

Synphony Model Compiler User Guide

Microsemi Edition I-2013.09M

October 2013

<http://solvnet.synopsys.com>

SYNOPSYS[®]

Copyright Notice and Proprietary Information

Copyright © 2013 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only.

Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____.”

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIMplus, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, Total-Recall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A
October 2013

Contents

Chapter 1: Getting Started

About The Symphony Model Compiler Tool	18
About the Software	18
Symphony Model Compiler and MATLAB	19
Symphony Model Compiler Design Flows	20
Symphony Model Compiler FPGA Design Flow	20
Design Requirements for RTL Generation	22
FPGA Design Flow Procedure	23
Finding Information	26
Getting Help	26

Chapter 2: SMC Blocks: Abs to Host Interface

Blocks — By Library	28
Communications	29
Control Logic	30
CORDIC	30
DSP Basics	31
Filtering	31
Floating Point Functions	32
Math Functions	33
Memories	34
Ports & Subsystems	35
Signal Operations	36
Sources	37
Transforms	38
Blocks — Alphabetical List	39
SMC Abs	42
SMC Accumulator	44
SMC Add	48

SMC Binary Logic	53
SMC Black Box	56
SMC Block Deinterleaver	62
SMC Block Interleaver	64
SMC CIC	66
SMC CIC2	70
SMC Commutator	77
SMC Comparator	84
SMC Concat	86
SMC Configurable FFT/IFFT	88
SMC Constant	94
SMC Convert	98
SMC Convolutional Deinterleaver	104
SMC Convolutional Encoder	106
SMC Convolutional Interleaver	109
SMC CORDIC Exp	111
SMC CORDIC Log	113
SMC CORDIC Polar	115
SMC CORDIC Rotator	117
SMC CORDIC SinCos	124
SMC CORDIC Sqrt	126
SMC CORDIC2	127
SMC Counter	131
SMC CRC Generator	138
SMC DDS	143
SMC DDS2	149
SMC Decommutator	163
SMC Delay	169

SMC Demux	171
SMC Depuncture	173
SMC Differentiator	176
SMC Divider	179
SMC DivMod	183
SMC Downsample	191
SMC Edge Detector	197
SMC Extract	200
SMC FDATool	203
SMC FFT	204
SMC FFT2	211
SMC FIFO	220
SMC FIR	226
SMC FIR Engine	235
SMC FIR Rate Converter	241
SMC FIR2	246
SMC Flow Control Buffer	275
SMC FP Add	286
SMC FP Compare	290
SMC FP Constant	292
SMC Fixed to FP	295
SMC FP Fused Mult Add	298
SMC FP Mult	301
SMC FP Port In	303
SMC FP Port Out	306
SMC FP to Fixed	309
SMC Gain	311
SMC Gold Sequence Generator	315

SMC HLS Subsystem	319
SMC Host Interface	326

Chapter 3: SMC Blocks: IIR to Viterbi Decoder

SMC IIR	340
SMC In	345
SMC Integrator	346
SMC Inverter	350
SMC Leading Zero Counter	352
SMC Log	354
SMC M Control	356
SMC Matrix Mult	360
SMC Mealy State Machine	364
SMC MinMax	367
SMC Moore State Machine	369
SMC Moving Average Filter	372
SMC Mult	378
SMC Mux	381
SMC Negate	386
SMC Out	388
SMC Parallel FIR	389
SMC Parallel to Serial	392
SMC Permutation	394
SMC PN Sequence Generator	396
SMC Port In	399
SMC Port Out	403
SMC Pow	405
SMC Pulse Generator	409
SMC Puncture	412

SMC RAM	414
SMC Ramp	419
SMC Random	422
SMC Recast	424
SMC Reed-Solomon Decoder	428
SMC Reed-Solomon Encoder	435
SMC Register	441
SMC Reshape	443
SMC RFIR	448
SMC ROM	453
SMC RTL Encapsulation	456
SMC Sample and Hold	465
SMC Saturate	467
SMC Sequence	470
SMC Serial to Parallel	473
SMC Shift Register	476
SMC Shifter	484
SMC SHLSTool	486
SHLSTool Toolbox Interface	487
Implementation Options Dialog Box	490
SMC Sign	507
SMC Signal Update	509
SMC SinCos	513
SMC SinCos2	516
SMC Single Clock Downsample	526
SMC Single Clock Upsample	529
SMC Smart Black Box	532
SMC Sqrt	539
SMC Subsystem	543

SMC Sum of Products	544
SMC Switch	548
SMC SynCoSimTool	550
SMC SynFixPtTool	554
SMC Test Vector Capture	556
SMC Upsample	557
SMC Vector Concat	561
SMC Vector Expand	567
SMC Vector Extract	570
SMC Vector Split	572
SMC Viterbi Decoder	574
Common Parameters	583
Output Format Options	583
Overflow Saturation Options	585
Underflow Rounding Options	585
Special Variables	588

Chapter 4: SMC Functions

shls_bitrev	590
shls_convert	592
shlsdemo	594
shlsdoc	596
shllib	597
shlsroot	599
shlstool	600
shlsver	602
syn_get_coefs	604
syn_get_datatype	605
syn_get_dspstartup	606
syn_get_wordlength	608
syn_read_hex	610

syn_set_atm	612
Timing Engine Configuration Dialog Box	612
syn_set_dspstartup	614
syn_set_portcapture	615
syn_set_portregister	616
syn_unlink	617
syn_write_wave	618

Chapter 5: Constraints

HLS Constraints File	620
Synphony Model Compiler Constraints	622
add_register_and_balance_parallel_paths	622
areabased_fir_arch_selection_atm Constraint	623
fir_architecture Constraint	623
multi_cycle_path Constraint	624
pattern_annotation Constraint	626
retime_across_blackbox	627
retiming_scale_factor Constraint	628
shls_retiming_lock Constraint	628
Multicycle Path Constraints	632
Specifying Multicycle Path Constraints	632
Automatically Inferring Multicycle Path Constraints	633
Forward-Annotation	636

Chapter 6: Synthesizing the Design

Configuring Synphony Model Compiler	638
Configuring Settings for Simulink Simulation	638
Configuring SMC Timing Modes for FPGAs	638
Setting Default Display Modes	640
Basic Procedures	641
Starting a Synphony Model Compiler Design	641
Working with Synphony Model Compiler Blocks	642
Setting Options for an Implementation	644
Setting up Implementations	644
Resolving Read/Write Conflicts in FPGA RAMs	647
Including Comments in the Generated RTL	649
Keeping Signal Names in Generated RTL	650

Using Constraints	653
Using Retiming	655
Optimizing with Retiming	655
Using Automatic Gate-level Retiming	660
Using Folding	662
Optimizing with Folding	662
Using Pattern Folding	665
Using Annotations for Folding	668
Optimizing with Multichannelization	674
Running Synthesis with SHLSTool	677
Synthesizing with a Host Interface Block	678

Chapter 7: Underlying DSP Fundamentals

Clock Domains	682
Resets in the SMC Tool	683
Global and Local Resets	683
Synchronous and Asynchronous Resets	684
Reset Implementation in RTL Code	685
Resets and RTL Testbenches	686
Clock and Reset Management	686
Clock_reset Module Interface	688
Reset Functionality with the Clock_reset Module	689
Clock Functionality with the Clock_reset Module	689
Clock/Reset Circuitry Files	690
Clock_reset Module Limitations	690
Log File Messages for the Clock_reset Module	691
Data Types	695
Fixed-Point and Floating-Point Representation	695
Symphony Model Compiler Data Type Implementation	696
Fixed-Point Data Type	696
Data Type Casting: Setting the Output Data Type	697
Matrix Data Types	698
CORDIC Algorithms	701
CORDIC Definitions	702
Unified CORDIC Applications	711
Multi-Rate Design	717
Sample Rate Terminology	717

Clock Generation and Clock Reset	721
Polyphase Filtering	724
Hierarchy Preservation	728
Subsystem Consolidation	729
Block Consolidation	730
Constant Propagation	731
RAMs	733
RAM Definitions	733
RAM Access Control	736
Port Use in Different RAM Configurations	737
Bus Protocols	738
AXI4-Lite Protocol	738
APB Protocol	743
AVLON-MM Protocol	745
Generic Interface Protocol	748

Chapter 8: Designing with the SMC Tool

Defining Clocks and Resets	754
Specifying a clock_reset Module	755
Defining Reset Signals	758
Designing Filters	760
Implementing FIR Filters with the FIR2 Block	760
Implementing FIR Filters with the FIR Block	764
Implementing Polyphase FIR Filters	767
Defining FIR Filter Coefficients with FDATool	768
Implementing IIR Filters	769
Defining IIR Filter Coefficients with FDATool	771
Working with Vectors	773
Creating Vector Signals	773
Using Math Operations on Vector Signals	774
Specifying ROM Data with syn_read_hex	776
Using Black Boxes and Third-Party IP	777
Integrating Black Boxes in the Design	777
Setting Black Box Parameters	780
Configuring a Black Box - Example	782
Using Optimizations with Black Boxes	784

Managing Subsystems and Hierarchy	786
Using the HLS Subsystem Block	786
Using the Symphony Subsystem Block	792
Tagging Subsystems with FPGA Synthesis Attributes	796

Chapter 9: Working with Custom Blocks

Primitives and Custom Blocks	800
Design Flow for Building Custom Blocks	804
Set up a Custom Library	805
Create a Custom Block	806
Define Basic Content for Custom Blocks	812
Define Content for Parameterized Blocks	816
Define Content for Reconfigurable Blocks	820
Designing with Custom Blocks	823
Maintaining Custom Libraries	824
Maintaining Independent Custom Libraries	824
Converting Custom Libraries	825
The MySign M-Generator	826

Chapter 10: Analyzing and Verifying the Design

Using Quantization Analysis Tools	832
Specifying Fixed-Point Options	832
Validating Algorithms with the Fixed-Point Toolbox	834
Using Plots	835
Using Smart Black Boxes for Cosimulation	837
Incorporating Smart Black Boxes in the Design	837
Configuring the Cosimulation Interface	839
Creating Smart Black Box Configuration Files	841
About Cosimulation with ModelSim	842
Simulating HLS Subsystem Blocks	844
Viewing Simulink Signals in a Waveform Viewer	846

Chapter 11: Working with SMC Output

Checking the Log File	850
Verifying the RTL with a Test Bench	853
Working with the Output for FPGA Designs	856

Chapter 12: Using M Code Blocks

Using M Code Blocks	858
Using M Code Blocks in SMC Designs	858
Coding for Synthesis with M Code Blocks	860
M Coding Style	862
Ports and Timing	862
M Code Block Data Types	864
Combinatorial Logic	868
Persistent Variables	869
Memories	869
State Machines	870
Counters	878
MATLAB Function that Evaluates to a Constant	880
User-Defined Functions for M Code Blocks	880
Overridable Parameters	881
Using Persistent Variables	883
M Code for Persistent Variables	883
Precision Bounds for Persistent Variables	885
Access-Update Sequence for Persistent Variables	888
Conditional Assignments to Persistent Variables	890
M Code Examples	892
Hardware-Aware M Code	892
Quantization of Constants	893
M Language Support for M Code Blocks	893
Keywords, Variables, Functions, and Structures	894
Operator Support	894
Built-In Function Support	895
SMC Functions for M Code Blocks	898
M Language Limitations	898

Chapter 13: Working with C Output

Design Flow for Working with C Output	902
Generating C Output Data	903
Generating C Output	903
Generating Output Data Files for C Output	905
Verifying C Output Against RTL	905

Simulating C Output	906
Simulating C Output with GCC	906
Simulating C Output in Microsoft Visual Studio 2010	906
Supported APIs for C Output	913
CEvent	913
int CModelDeleteEvent	915
REGISTER_DESIGN	915
void * CModelCreateInstance	916
int CModelDeleteInstance	917
int CModelSetInput	918
char * CModelGetOutput	919
int CModelEvalNext	920
CModelGetErrMsg()	921
int CSimGetLicense()	923
int CSimReleaseLicense()	924
C Model API Usage	925
Using C Output in Simulink	927
Using C Output to Speed up Simulink Simulations	927
Generating the Simulink C Output Wrapper	928
Using C Output with SystemC	932
Using C Output with Verilog-C Interfaces	933
Simulating C Output with Verilog-C Interfaces	933
Verilog-C Interface Wrappers	934
Verilog-C Interface Wrapper Example	936
Verilog-C Interface Wrapper System Tasks	938

Appendix A: Blockset Summary

SMC Block Summary	946
-------------------------	-----

CHAPTER 1

Getting Started

The following topics provide a general introduction to the Symphony Model Compiler software:

- [About The Symphony Model Compiler Tool, on page 18](#)
- [Symphony Model Compiler Design Flows, on page 20](#)
- [Finding Information, on page 26](#)
- [Getting Help, on page 26](#)

About The Symphony Model Compiler Tool

This section briefly discusses the following topics:

- [About the Software, on page 18](#)
- [Symphony Model Compiler and MATLAB, on page 19](#)

About the Software

The Symphony product is a high-level tool for hardware DSP design. It is an add-on to the Simulink® product from The MathWorks®, and provides the designer with an automated path from high-level design and simulation to an architecturally-optimized, synthesizable, system-level HDL implementation. This tool provides performance and productivity benefits for designers who are implementing DSP circuits into FPGA devices. The software achieves significantly higher performance than alternative solutions and provides the designer with a mechanism to evaluate high-level area/performance trade-offs. The output is synthesizable HDL code ready for use with the Synopsys® Synplify Pro® synthesis software.

The software consists of the following components:

- A Simulink blockset
- An automated mechanism to produce a bit-exact, optimized HDL implementation when a Simulink model is created using this blockset
- An automated mechanism to capture test vectors during Simulink simulation
- Automatic HDL test bench generation to verify bit accuracy

Value for DSP Algorithm Designers

Using FPGAs for DSP design is a complex task, and the Symphony Model Compiler software makes it easy to maximize the optimizations possible with this design flow. For DSP algorithm designers implementing in FPGAs, the Symphony Model Compiler software does the following:

- Provides a familiar working environment. The Symphony Model Compiler tool plugs into the familiar Simulink and MATLAB environment, so the DSP algorithm designer need not learn a new tool or methodology.

- Automates the design flow by smoothly transitioning from the high-level arithmetic Simulink abstractions to the Synopsys FPGA synthesis tools, with which it is tightly integrated. It eliminates the need for the algorithm designer to learn about physical issues that affect the design.
- Is the only tool that offers a vendor-independent solution for a DSP FPGA implementation. The designer can experiment with different FPGA vendor technologies.
- Includes proprietary optimizations that improve area and performance.

Value for Hardware Engineers

For the hardware engineer, the Symphony Model Compiler software does the following:

- Eliminates costly iterations normally required to accurately translate the DSP algorithms, because it generates the necessary RTL code. It eliminates the extra cycles normally required to generate RTL that captures the algorithmic intent of the designer and also accounts for physical issues.
- Makes the hardware engineer's job easier with built-in optimizations that account for hardware considerations. The Symphony Model Compiler tool does DSP-level optimizations (z-domain) using implementation-level constraints like target technology and timing.
- Generates an optional testbench, which can be extremely useful in verifying bit accuracy.

Symphony Model Compiler and MATLAB

It is assumed that you have valid licenses for the appropriate versions of the MATLAB® and Simulink software from MathWorks and that you are familiar with these products. For FPGA targets, the use model assumes that the Symphony Model Compiler output will be synthesized with the Synplify Pro product from Synopsys, so the designer must have this product and be familiar with its use.

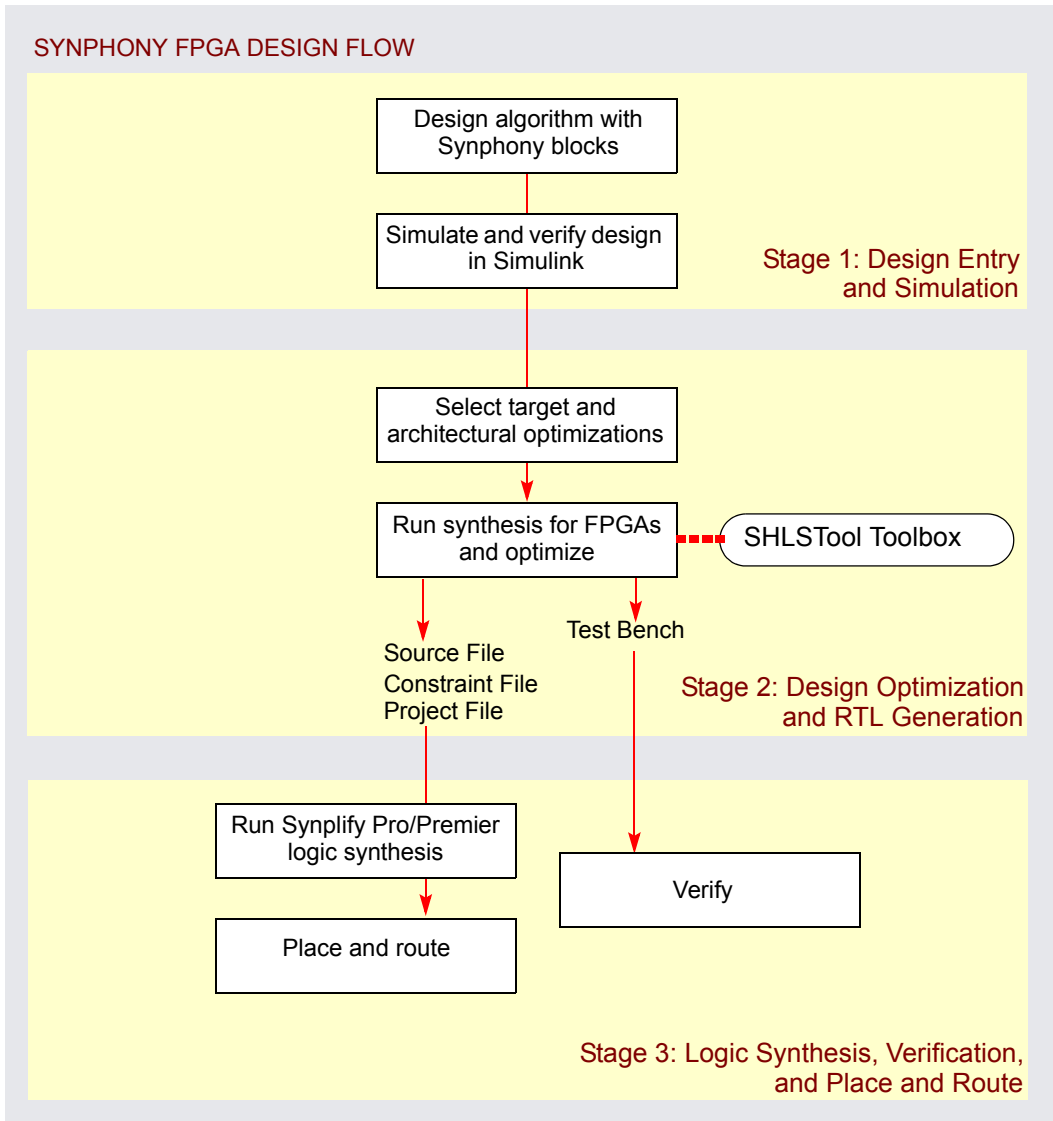
Symphony Model Compiler Design Flows

This section contains a flow description and a step-by-step procedure.

- [Symphony Model Compiler FPGA Design Flow, on page 20](#)
- [Design Requirements for RTL Generation, on page 22](#)
- [FPGA Design Flow Procedure, on page 23](#)

Symphony Model Compiler FPGA Design Flow

The following figure summarizes the flow for creating an FPGA design, generating RTL code, synthesis, and verification. For more details, see the procedure in [FPGA Design Flow Procedure, on page 23](#). To step through an example using the tool for an FPGA design, refer to the training materials packaged with the tool.



Stage 1: Design Entry and Simulation

For this first stage of the flow, use the Simulink software and the Symphony Model Compiler blockset to compose the design. You can use other Simulink blocksets for simulation and debugging, but the software only generates RTL

code for blocks from the Symphony blockset. Simulate and verify the design at least once with Simulink to ensure correct functionality. For additional details about the flow, see [FPGA Design Flow Procedure, on page 23](#).

Stage 2: Design Optimization and RTL Generation

The strengths of the Symphony Model Compiler software are optimization and RTL generation. To do this, add the SHLSTool block to the design.

Set system-level optimization settings and the target technology with the SHLSTool block. Use the same block to generate RTL code. The optimizations are targeted towards the FPGA design. For details of the flow, see [FPGA Design Flow Procedure, on page 23](#). The software generates RTL code and an optional test bench.

Stage 3: Logic Synthesis, Verification, and Place-and-Route

For this stage, you use synthesis, verification, and place-and-route tools. If you generated a test bench, run it in a VHDL simulator to verify the bit-exactness of the generated VHDL code with respect to the Simulink model. Use the RTL code for logic synthesis with the Synplify Pro software. After synthesis, verify the post-synthesis VHDL code generated by the synthesis software against the Symphony Model Compiler test bench. Then, use the synthesized netlist as input to the place-and-route tool of the FPGA vendor. For additional details about the flow, see [FPGA Design Flow Procedure, on page 23](#).

Design Requirements for RTL Generation

To generate RTL, you must follow these rules:

- The design must be bound by the Symphony Model Compiler Port In and Port Out blocks. You must define your design boundaries with Symphony Model Compiler Port In and Port Out blocks. If you do not do this, the Symphony Model Compiler tool cannot determine the ports for the RTL description. The generated RTL will not be correct.
- All the blocks that need to be synthesized into RTL must be from the Symphony Model Compiler blockset.
- Do not use the following characters in port, block, subsystem, or signal names in the Simulink model. If you do, the tool might not generate RTL code successfully.

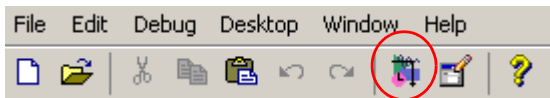
& Ampersand	! Exclamation mark
' Apostrophe	` Grave accent
* Asterisk	- Minus
\ Backslash	# Number sign
^ Caret	% Percent
: Colon	+ Plus
, Comma	? Question mark
{Curly bracket, open	; Semicolon
} Curly bracket, close	~ Tilde
\$ Dollar sign	

- Data types that are propagated through any of the Synphony blocks must have a word length that is greater than or equal to the fraction length.

FPGA Design Flow Procedure

The following procedure describes the steps required to follow the design flow ([Synphony Model Compiler FPGA Design Flow, on page 20](#)):

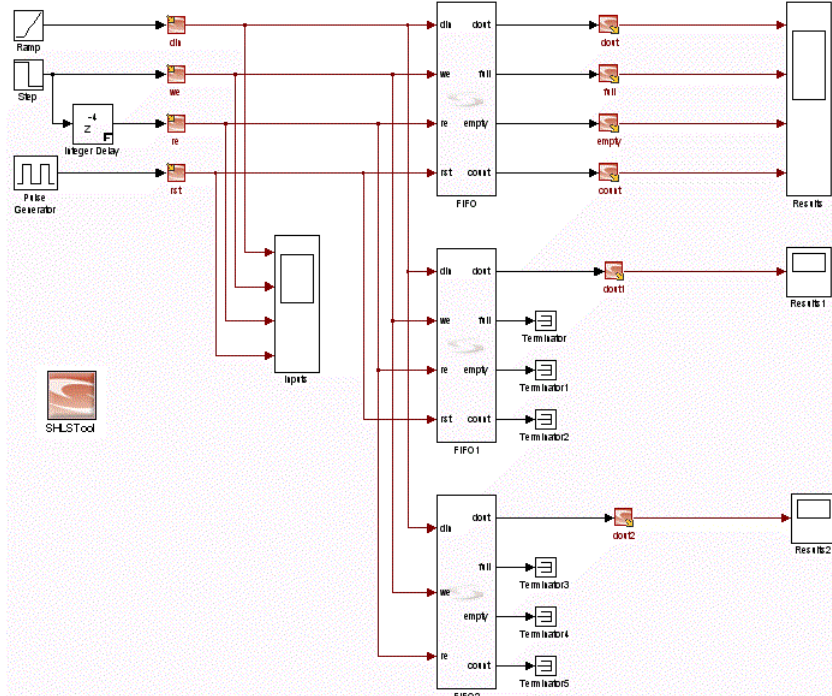
1. Start MATLAB and make sure you are in your design directory. Click the Simulink icon and open Simulink.



2. Set up the design.
 - Open a model window. For details, see [Starting a Synphony Model Compiler Design, on page 641](#).
 - Configure the tool settings and specify the timing mode, as described in [Configuring Synphony Model Compiler, on page 638](#).

- Build your circuit with Synphony Model Compiler blocks. For details, see [Working with Synphony Model Compiler Blocks, on page 642](#).
3. Verify the design in Simulink.
 - Set simulation parameters and simulate the design using the commands on the Simulate menu. For details, consult the Simulink documentation.
 - Use a Simulink simulation with scaled double precision.
 - Impose quantization on the design enabling fixed-point data type associations.
 - Verify the bit-accurate design with a Simulink simulation.
 4. Set up the implementation for RTL generation.
 - Make sure your design follows the requirements described in [Design Requirements for RTL Generation, on page 22](#).
 - In the Simulink window, click Synphony Blockset, and add the SHLSTool block to the design. One instance of this block controls the whole design.
 - Double-click the SHLSTool block in the model window to open the Synphony Model Compiler toolbox.
 - Set up the implementation and the options for it, as described in [Setting up Implementations, on page 644](#).
 - Click OK in the Implementation Options dialog box.
 5. Click Run in the Synphony Model Compiler window to generate RTL code and output files for the optimized design.

The software generates RTL code for the Synphony Model Compiler block components in the design. It does not generate RTL code for other blocks.



6. Run logic synthesis, verify, and place-and-route your design. For details about these tasks, consult the documentation for these tools.
 - Start the Synplify Pro or Synplify Premier software and use the source code, constraint, and project files generated in the previous step as input to synthesize your design. If you want to target a different family or device, you can reset that in the synthesis tool when you run synthesis.
 - Compare the test bench to post-synthesis gate-level simulation to verify results.
 - Place and route the design, using the tool appropriate to your design and vendor.

Finding Information

The following table shows you where to find information:

For...	See...
Procedures and tips on using the tool	Synthesizing the Design
Descriptions of individual Synphony Model Compiler blocks	Blocks — By Library Blocks — Alphabetical List
Descriptions of command line functions	SMC Functions
Help	Getting Help

Getting Help

The Synphony Model Compiler software includes documentation, which you can access in the following ways:

- For a printed copy, go to the MATLAB help (Help->Product Help) and select Synphony Model Compiler in the Contents tab of the Help system. Scroll down and open the PDF document (*Release Notes* or *User Guide*) you need. You can print out the PDF documents.
- For online help, open the Contents tab of the Help Navigator, scroll to Synphony Model Compiler, and select the topic you want.
- For context-sensitive online help about blocks in the Simulink library browser, click a block to see a one-line description displayed. Right-click on a block and select Help to display information about the block.
- For context-sensitive online help about blocks in the Simulink model window, right-click on the block and select Help. This displays information about the block.
- For context-sensitive online help on a dialog box, click the Help button.

CHAPTER 2

SMC Blocks: Abs to Host Interface

This chapter describes the Symphony Model Compiler blocks and the Symphony Model Compiler custom blocks, categorizing them by library and alphabetically. See the following:

- [Blocks — By Library, on page 28](#)
- [Blocks — Alphabetical List, on page 39](#)

Note the following:

- The Symphony Model Compiler library includes some toolboxes at the top level: SynCoSimTool, SHLSTool and SynFixPtTool. They are documented along with the other blocks.
- Some Symphony Model Compiler blocks are classified as custom blocks. For details, and a list of the custom blocks, see [Primitives and Custom Blocks, on page 800](#).
- Some blocks are specialized blocks, and the icons reflect the difference. For example, Black Box, M Control, and the port and subsystem blocks.
- The appendix [Blockset Summary, on page 945](#), contains a quick reference list of parameters like saturation and word length for different blocks.

Blocks — By Library

The Symphony Model Compiler blockset is organized into the block libraries described in the following table. You can access the libraries from the Simulink Library Browser. For an alphabetical list of individual blocks, see [Blocks — Alphabetical List, on page 39](#)).

Communications	Contains blocks specific to the communications industry.
Control Logic	Contains blocks that implement logic for controlling datapaths.
CORDIC	Contains blocks for specialized CORDIC math operations.
DSP Basics	Contains fundamental blocks used for most DSP functions.
Filtering	Contains blocks for designing and implementing filters.
Floating Point Functions	Contains blocks that perform various floating point computations of math functions.
Math Functions	Contains blocks for specialized math operations.
Memories	Contains blocks for memory structures like RAMs and FIFOs.
Ports & Subsystems	Contains port and black box blocks.
Signal Operations	Contains blocks for the manipulation of signals.
Sources	Contains blocks that generate constants and counters.
Transforms	Contains blocks for transforms that are important to DSP operations.
SMC SynCoSimTool	Specialized toolbox that manages the cosimulation interface between the smart black boxes in the design and ModelSim.
SMC SHLSTool	Specialized toolbox that controls the generation of RTL for synthesis. The toolbox lets you set options in the Implementation Options dialog box, described in Implementation Options Dialog Box, on page 490 .
SMC SynFixPtTool	Specialized toolbox that opens the Simulink fixed point interface.

Communications



This library contains specialized blocks used for DSP designs in the communications industry.

SMC Block Deinterleaver	Reshuffles a fixed number of interleaved input symbols to obtain the original sequence.
SMC Block Interleaver	Shuffles a fixed number of input symbols to a new permutation.
SMC Convolutional Deinterleaver	Reshuffles streaming input symbols according a to a predefined mapping scheme.
SMC Convolutional Encoder	Corrects feed-forward errors using k/n convolutional codes.
SMC Convolutional Interleaver	Shuffles streaming input symbols to a new permutation, using a predefined mapping scheme.
SMC CRC Generator	Generates CRC bits and appends them to the input data frames.
SMC Depuncture	Removes user-specified symbols from the input data stream and replaces them with zeroes.
SMC Gold Sequence Generator	Generates a Gold sequence, with specified polynomials u and v , of period $N = 2^n - 1$, called a preferred pair.
SMC PN Sequence Generator	Generates a sequence of pseudorandom (PN) binary numbers using a linear-feedback shift register (LFSR).
SMC Puncture	Removes user-specified bits from the input data stream.
SMC Reed-Solomon Decoder	Decodes the encoded signal using Reed-Solomon error-correcting codes.
SMC Reed-Solomon Encoder	Generates an encoded signal, using Reed-Solomon codes.
SMC Viterbi Decoder	Decodes convolutionally encoded input data.

Control Logic



This library contains blocks that provide control logic for outputs.

SMC M Control	Uses an M file to define a function for complex control logic.
SMC Mealy State Machine	Provides control logic where the output depends on the input and an internal state vector.
SMC Moore State Machine	Provides control logic where the output depends on the current state.

CORDIC



This library contains blocks for specialized CORDIC math operations.

SMC CORDIC Exp	Calculates the natural exponent of the input using the CORDIC algorithm.
SMC CORDIC Log	Calculates the natural logarithm of the input using the CORDIC algorithm.
SMC CORDIC Polar	Calculates $\sqrt{x^2+y^2}$ and $\text{atan}(y/x)$ where x and y are the inputs.
SMC CORDIC Rotator	Implements a fully pipelined CORDIC rotator.
SMC CORDIC SinCos	Implements a sine and/or cosine generator circuit using a fully parallel CORDIC algorithm in rotation mode.
SMC CORDIC Sqrt	Calculates the square root of the input using the CORDIC algorithm.
SMC CORDIC2	Implements a circular CORDIC (Coordinate Digital Rotation Computer).

DSP Basics



This library contains blocks for basic DSP operations.

SMC Add	Implements a full-precision signed adder or subtractor.
SMC Delay	Delays the input by the specified number of sample clock cycles.
SMC Gain	Implements a constant gain to the input.

Filtering



This library contains blocks for designing and implementing filters.

SMC CIC	Custom block that implements a CIC filter.
SMC CIC2	Implements a CIC filter with additional enhancements compared to the CIC block.
SMC Differentiator	Custom block that performs a discrete time differentiation of the input signal.
SMC FDATool	Opens the Simulink FDATool interface.
SMC FIR	Implements a finite impulse response (FIR) filter.
SMC FIR2	Implements fixed and reloadable coefficient FIR filters, including polyphase filters, multichannel filters, and symmetric coefficient filters.
SMC FIR Engine	Implements a finite impulse response (FIR) filter that uses the coefficients as vector input.
SMC FIR Rate Converter	Implements a polyphase FIR filter.
SMC IIR	Implements an infinite impulse response (IIR) filter.

SMC Integrator	Performs a discrete time integration of the input signal.
SMC Moving Average Filter	Implements a hardware efficient moving average filter.
SMC Parallel FIR	Implements a parallel input FIR filter.
SMC RFIR	Custom block that implements a reloadable finite impulse response FIR filter.

Floating Point Functions



This library contains blocks that perform various floating point computations of math functions.

SMC FP Add	Adds or subtracts two floating point values.
SMC FP Compare	Compares two floating point numbers and returns 1 if the selected condition holds true. Otherwise, 0 is returned.
SMC FP Constant	Sets a constant value for a specified floating point representation as the output.
SMC FP Fused Mult Add	Performs various multiply-add operations on three/four inputs.
SMC FP Port In	Converts Simulink double to SMC floating point format. Can be used instead of SMC Port In to define the RTL generation boundary of floating point designs.
SMC FP Port Out	Converts SMC floating point format to Simulink double. Can be used instead of SMC Port Out to define the RTL generation boundary of floating point designs.
SMC FP to Fixed	Converts an input SMC floating point format to a signed fixed point format for the specified word length and fraction length.
SMC Fixed to FP	Converts a fixed point input to the SMC floating point format with the specified representation.
SMC FP Mult	Multiplies two floating point values.

Math Functions



This library contains blocks for specialized math operations.

SMC Abs	Calculates the absolute value of the scalar input.
SMC Accumulator	Implements an accumulator with optional reset and enable.
SMC Add	Implements a full-precision signed multi-input adder. Selected inputs can be configured for addition or subtraction.
SMC Binary Logic	Calculates bitwise binary logic functions on the inputs.
SMC Comparator	Implements a programmable comparator.
SMC Divider	Calculates the fixed-point fractional division of two inputs, A and B.
SMC DivMod	Calculates the integer division and/or modulo function of two inputs, A and B.
SMC Gain	Implements a constant gain to the input.
SMC Log	Calculates the natural logarithm of the input.
SMC Matrix Mult	Implements matrix multiplication of a two-input matrix signal.
SMC MinMax	Custom block that calculates the minimum, maximum, or minimum and maximum of two inputs.
SMC Mult	Implements a full-precision multiplier.
SMC Negate	Computes the two's complement (arithmetic negation) of a signed input.
SMC Pow	Raises a value to the power of another value.
SMC Shifter	Performs a variable left or right shift on the input signal.
SMC Sign	Custom block that provides the 2-bit sign value (+1 or -1) for the input.
SMC SinCos	Calculates $\sin(2\pi f)$ or $\cos(2\pi f)$ for the input.

SMC SinCos2	Creates sin and cos waveforms based on the input phase and amplitude values.
SMC Sqrt	Calculates the square root of the input.
SMC Sum of Products	Multiplies inputs with gain values and calculates the sum of the computed products to provide a scalar output.

Memories



This library contains blocks for memory structures like RAMs and FIFOs.

SMC Delay	Delays the input by the specified number of sample clock cycles.
SMC FIFO	Implements a synchronous FIFO (First in First Out) memory queue.
SMC Flow Control Buffer	Provides forward or backward flow control.
SMC Permutation	Shuffles the incoming data according to a specified permutation vector.
SMC RAM	Implements a memory function through a storage array that has read and write access through ports.
SMC Register	Inserts a delay.
SMC ROM	Models a read-only memory (ROM) with a latency of one sample.
SMC Shift Register	Implements a delay line with dynamic or static access to intermediate taps.

Ports & Subsystems



This library contains port and black box blocks.

SMC Black Box	Provides a way to embed other blocks.
SMC HLS Subsystem	Lets you add a previously designed Symphony model to the current design and set implementation settings for it.
SMC Host Interface	Provides an interface to the host processor using a simpler bus protocol to configure the design.
SMC In	Provides a way to add an in port to a subsystem
SMC Out	Provides a way to add an out port to a subsystem
SMC Port In	Defines the input boundaries for the DSP design to be implemented in RTL.
SMC Port Out	Defines the output boundaries for the DSP design to be implemented in RTL.
SMC RTL Encapsulation	Embeds and simulates RTL blocks inside their Simulink model without the need of external RTL simulators or special Simulink features.
SMC Smart Black Box	Lets you embed third-party IP in a Symphony Model Compiler design.
SMC Subsystem	Allows you to add a subsystem to a Symphony Model Compiler design.
SMC Test Vector Capture	Toggles between setting or resetting Port In and Port Out Capture Test Vector mode for all Symphony Model Compiler ports

Signal Operations



This library contains blocks for the management of signals.

SMC Commutator	Sequentially switches the data from multiple input ports to a single output port, increasing the data rate of each output port accordingly.
SMC Concat	Concatenates the bits of up to 32 input signals.
SMC Convert	Changes the word size and data type of the input. You can apply a constant before the new word size and data type is casted.
SMC Decommutator	Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly.
SMC Demux	Implements a de-multiplexer of up to 2048 outputs with a latency of one sample.
SMC Downsample	Decreases the sample rate of the input by removing samples.
SMC Edge Detector	Outputs a unity amplitude pulse of one sample period to a synchronous transition from high to low or low to high.
SMC Extract	Extracts specified bits from the input signal.
SMC Leading Zero Counter	Computes the number of leading zeros for an unsigned input.
SMC Mux	Implements a multiplexer of up to 2048 inputs.
SMC Parallel to Serial	Implements a data packet splitter that divides the parallel data word at the input into small serial data packets in the order specified.
SMC Recast	Custom block that provides a value, based on the requested data type cast at the output and maintaining the same bits as provided at the input.
SMC Reshape	Changes the dimensionality of the input signal.
SMC Sample and Hold	Samples and holds the input signal.

SMC Saturate	Saturates the input signal to the values specified in the positive and negative saturation value fields.
SMC Serial to Parallel	Implements a data packet combiner that collects serial data packets at the input and merges them into a parallel data word at the output.
SMC Signal Update	Updates the specified elements of a vector or matrix input signal using a given update signal.
SMC Single Clock Downsample	Provides variable rate and single clock downsample operations.
SMC Single Clock Upsample	Provides variable rate and single clock upsample operations.
SMC Switch	Routes the signal through input or data port based on signal in the control port.
SMC Upsample	Increases the sample rate of the input by inserting zeroes.
SMC Vector Concat	Constructs vectors by bundling up to 2048 inputs together.
SMC Vector Expand	Converts scalar input to vector output.
SMC Vector Extract	Extracts selected ports for the output.
SMC Vector Split	Implements a de-multiplexer of up to 2048 outputs.

Sources



This library contains blocks that generate constants and counters.

SMC Constant	Implements a source with a constant value.
SMC Counter	Implements a resettable modulo counter with enable.
SMC DDS	Custom block that creates a direct digital synthesizer with sin and cos waves based on frequency, phase settings, and modulations.

SMC DDS2	Creates a direct digital synthesizer with sin and cos waveforms based on frequency, phase settings, and modulations. This block provides additional functionality and QoR improvements compared with the DDS block.
SMC Pulse Generator	Generates a single pulse.
SMC Ramp	Custom block that creates a ramp based on increments derived from a port or parameter
SMC Random	Custom block that creates a random integer of the requested word length.
SMC Sequence	Custom block that repeats a sequence of specified data

Transforms



This library contains blocks for transforms that are important to DSP operations.

SMC Configurable FFT/IFFT	Implements a fully pipelined Fast Fourier Transform (FFT) or Inverse Fast Fourier Transform (IFFT).
SMC FFT	Implements a fully pipelined Fast Fourier Transform.
SMC FFT2	Implements Fast Fourier Transform that supports both serial and parallel inputs.

Blocks — Alphabetical List

This list includes the toolboxes, as well as the blocks and custom blocks:

SMC Abs, on page 42	SMC Accumulator, on page 44
SMC Add, on page 48	SMC Binary Logic, on page 53
SMC Black Box, on page 56	SMC Block Deinterleaver, on page 62
SMC Block Interleaver, on page 64	SMC CIC, on page 66
SMC CIC2, on page 70	SMC Commutator, on page 77
SMC Comparator, on page 84	SMC Concat, on page 86
SMC Configurable FFT/IFFT, on page 88	SMC Constant, on page 94
SMC Convert, on page 98	SMC Convolutional Deinterleaver, on page 104
SMC Convolutional Encoder, on page 106	SMC Convolutional Interleaver, on page 109
SMC CORDIC Exp, on page 111	SMC CORDIC Log, on page 113
SMC CORDIC Polar, on page 115	SMC CORDIC Rotator, on page 117
SMC CORDIC SinCos, on page 124	SMC CORDIC Sqrt, on page 126
SMC CORDIC2, on page 127	SMC Counter, on page 131
SMC CRC Generator, on page 138	SMC DDS, on page 143
SMC DDS2, on page 149	SMC Decommulator, on page 163
SMC Delay, on page 169	SMC Demux, on page 171
SMC Depuncture, on page 173	SMC Differentiator, on page 176
SMC Divider, on page 179	SMC DivMod, on page 183
SMC Downsample, on page 191	SMC Edge Detector, on page 197
SMC Extract, on page 200	SMC FDATool, on page 203
SMC FFT, on page 204	SMC FFT2, on page 211
SMC FIFO, on page 220	SMC FIR, on page 226

SMC FIR Engine, on page 235	SMC FIR2, on page 246
SMC FIR Rate Converter, on page 241	SMC Flow Control Buffer, on page 275
SMC FP Add, on page 286	SMC FP Compare, on page 290
SMC FP Constant, on page 292	SMC Fixed to FP, on page 295
SMC FP Fused Mult Add, on page 298	SMC FP Mult, on page 301
SMC FP Port In, on page 303	SMC FP Port Out, on page 306
SMC FP to Fixed, on page 309	SMC Gain, on page 311
SMC Gold Sequence Generator, on page 315	SMC HLS Subsystem, on page 319
SMC Host Interface, on page 326	SMC IIR, on page 340
SMC In, on page 345	SMC Integrator, on page 346
SMC Inverter, on page 350	SMC Leading Zero Counter, on page 352
SMC Log, on page 354	SMC M Control, on page 356
SMC Mealy State Machine, on page 364	SMC Matrix Mult, on page 360
SMC Moore State Machine, on page 369	SMC MinMax, on page 367
SMC Mult, on page 378	SMC Moving Average Filter, on page 372
SMC Negate, on page 386	SMC Mux, on page 381
SMC Parallel FIR, on page 389	SMC Out, on page 388
SMC Permutation, on page 394	SMC Parallel to Serial, on page 392
SMC Port In, on page 399	SMC PN Sequence Generator, on page 396
SMC Pow, on page 405	SMC Port Out, on page 403
SMC Puncture, on page 412	SMC Pulse Generator, on page 409
SMC Ramp, on page 419	SMC RAM, on page 414
SMC Recast, on page 424	SMC Random, on page 422

SMC Reed-Solomon Encoder, on page 435	SMC Reed-Solomon Decoder, on page 428
SMC Reshape, on page 443	SMC Register, on page 441
SMC ROM, on page 453	SMC RFIR, on page 448
SMC Sample and Hold, on page 465	SMC RTL Encapsulation, on page 456
SMC Sequence, on page 470	SMC Saturate, on page 467
SMC Shift Register, on page 476	SMC Serial to Parallel, on page 473
SMC SHLSTool, on page 486	SMC Shifter, on page 484
SMC Signal Update, on page 509	SMC Sign, on page 507
SMC SinCos2, on page 516	SMC SinCos, on page 513
SMC Single Clock Upsample, on page 529	SMC Single Clock Downsample, on page 526
SMC Sqrt, on page 539	SMC Smart Black Box, on page 532
SMC Sum of Products, on page 544	SMC Subsystem, on page 543
SMC SynCoSimTool, on page 550	SMC Switch, on page 548
SMC Test Vector Capture, on page 556	SMC SynFixPtTool, on page 554
SMC Vector Concat, on page 561	SMC Upsample, on page 557
SMC Vector Extract, on page 570	SMC Vector Expand, on page 567
SMC Viterbi Decoder, on page 574	SMC Vector Split, on page 572

SMC Abs

Calculates the absolute value of the scalar or vector input.

Library

Symphony Model Compiler [Math Functions](#)

Description



The Symphony Model Compiler Abs block calculates the absolute value of the vector or scalar input. The output has the same signal dimension as the input, with each channel being the absolute value of the corresponding input channel.

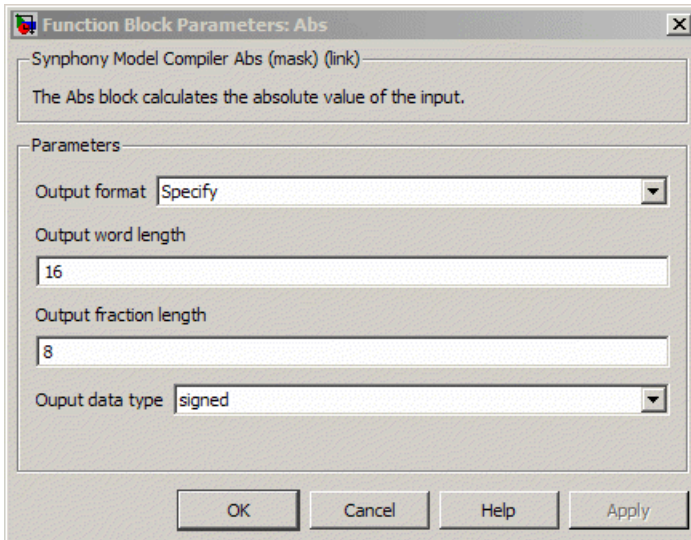
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Abs Parameters



For descriptions of the parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

The default output format for the Abs block is Automatic, where the tool keeps the input word length and fraction length with unsigned output. Thus, there is no lost bit for negative extremes, because there is no overflow or underflow.

If you use Specify to specify the output format of the block, and the integer length and/or fraction length you specify is less than the input values, the output is wrapped (no saturation) for overflow and/or truncated (no rounding) for underflow.

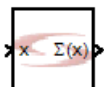
SMC Accumulator

Implements an accumulator with optional reset and enable.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Accumulator block implements an adder or subtractor-based accumulator with optional reset and enable ports.

$$y[n]=x[n-1]+y[n-1]$$

$$H(z)=z/(1-z)$$

Automatic Scalar Expansion

If the data input is a vector, and if one of the reset or enable ports is scalar, the reset and enable ports are expanded according to the size of the data input vector.

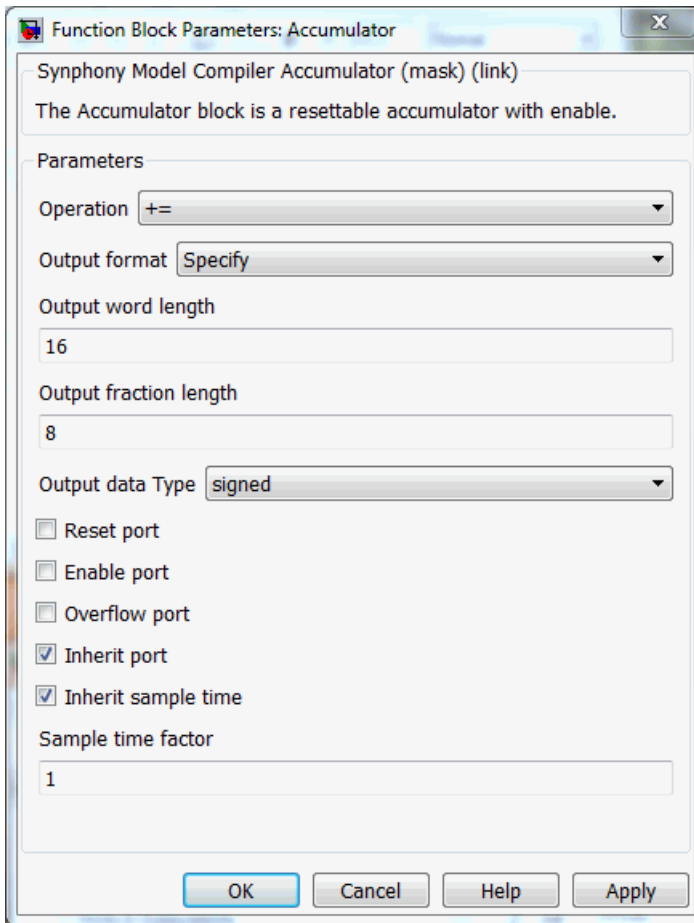
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block provides the result of the accumulating register.

Accumulator Parameters



Function Block Parameters: Accumulator

Symphony Model Compiler Accumulator (mask) (link)

The Accumulator block is a resettable accumulator with enable.

Parameters

Operation $+=$

Output format Specify

Output word length
16

Output fraction length
8

Output data Type signed

☐ Reset port

☐ Enable port

☐ Overflow port

☒ Inherit port

☒ Inherit sample time

Sample time factor
1

OK Cancel Help Apply

Operation

Configures the operation of the block. You can select from the following:

- $+=$
- $- =$

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Reset port

When enabled, the block is implemented with a reset port.

Enable port

When enabled, the block is implemented with an enable pin.

Overflow port

When enabled, the block is implemented with an output pin (ovf) for monitoring overflows.

Inherit port

Determines whether the tool creates an inherit port. The tool creates an inherit port when you enable the option. If you enable the option, the tool applies automatic scalar expansion to the inherit and data ports. If one input is scalar and the other is vector, the scalar input is expanded to the size of the vector input.

This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Special Variables, on page 588](#) for information about these variables.
- Use the inherit option to specify the Output data type. See [Output Data Type, on page 101](#) for a description of the option.

Inherit sample time

Determines whether the output inherits the sample time from the inherit port. Enabling this option means that the output port inherits the sample time from the inherit port, and disabling it means the output

port inherits sample time from the input. This option becomes available when you enable Inherit port.

Sample time factor

Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

SMC Add

Implements both signed single-input and multi-input adders. Selected inputs can be configured for addition or subtraction.

Library

Synphony Model Compiler [DSP Basics](#) and Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Add block implements a signed single-input or multi-input adder, whose inputs can be configured for addition or subtraction. The Add block can have up to 256 input ports. The inputs can be vectorized to a maximum size of 2048 for single-input adder (in sum of elements mode) and multi-input adder implementations.

Automatic Scalar Expansion

If enabled when the block has a mixture of scalar inputs and matrix or vector inputs, the tool expands the scalar inputs to the size of the vectors or matrices. The vector or matrix inputs must have the same size. You cannot have a combination of vector and matrix inputs.

In sum of elements mode, the elements of the input vector/matrix signal are summed up to a scalar. In multi-input adder mode, if the input signals are vectors/matrices, the corresponding elements of the input vectors/matrices are summed to give a vector/matrix output of the same size as the inputs.

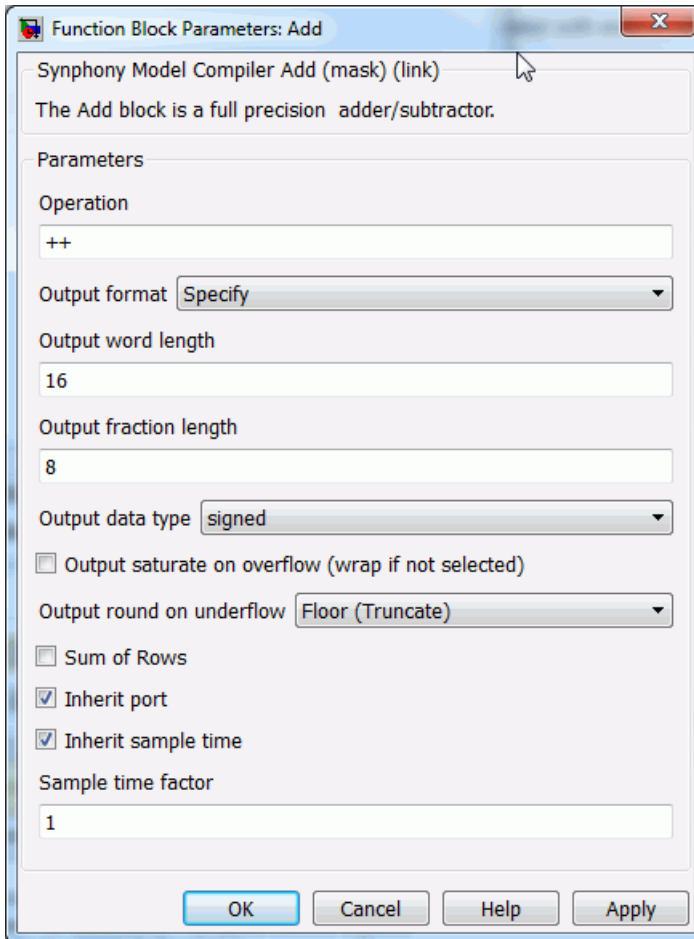
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Add Parameters



The dialog box titled "Function Block Parameters: Add" contains the following fields and options:

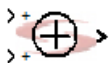
- Header: Symphony Model Compiler Add (mask) (link)
- Description: The Add block is a full precision adder/subtractor.
- Parameters section:
 - Operation: ++
 - Output format: Specify (dropdown)
 - Output word length: 16
 - Output fraction length: 8
 - Output data type: signed (dropdown)
 - ☐ Output saturate on overflow (wrap if not selected)
 - Output round on underflow: Floor (Truncate) (dropdown)
 - ☐ Sum of Rows
 - ☒ Inherit port
 - ☒ Inherit sample time
 - Sample time factor: 1
- Buttons: OK, Cancel, Help, Apply

Operation

Configures the operation of the multi-input adder. Specify a + or - for each input to the block; the number of inputs is determined by the number of + or - signs. The default is ++.

The + or - signs correspond to the adding or subtraction of the corresponding input port. For example, if you specify ++-, the block is implemented with three inputs. The output of the block is calculated as Input1 + Input2 -Input3. The inputs and the operation symbol on the block icon reflect the operation choices you made. For example:

++ Operation



-- Operation



++- Operation



+ Operation



If your design has a single input port feeding in the vector or matrix signal and if you set Operation to +, the output is the sum of the vector or matrix elements. If you set Operation to -, the output is the negative value of the sum of the vector or matrix elements.

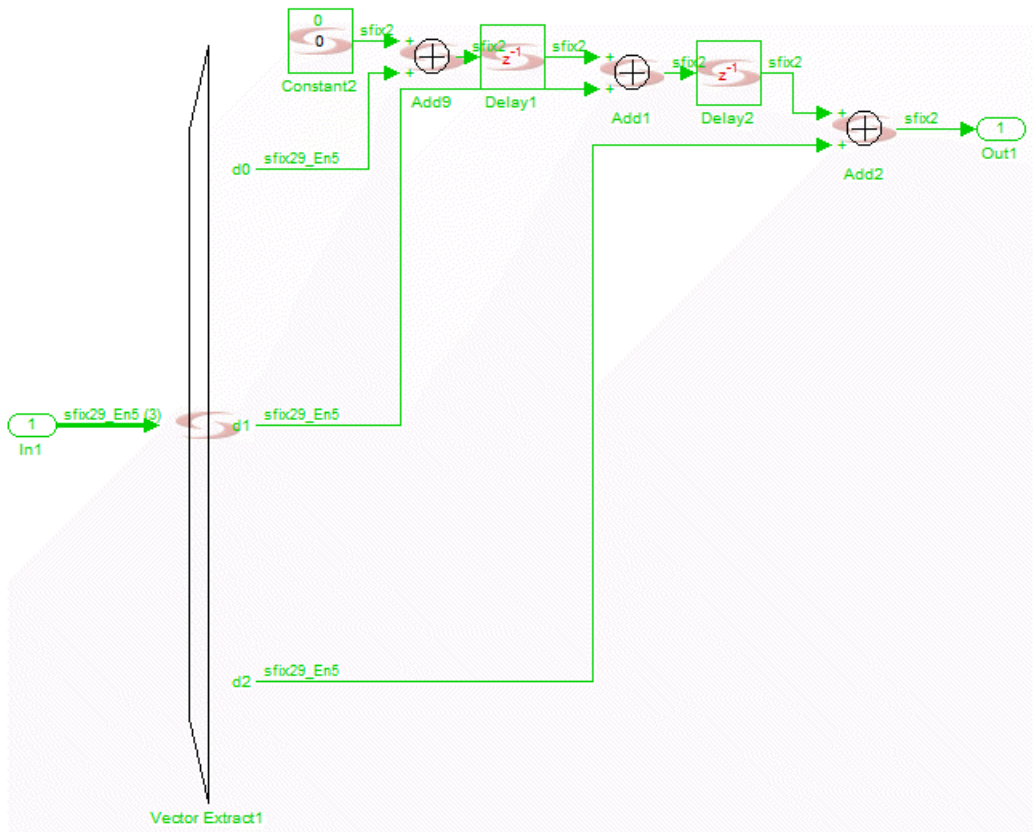
If your design has a single input port feeding a matrix signal and if you set Operation to + when Sum of Rows is enabled, the output is a vector whose size matches the number of columns in the input signal. Each element of the output vector is a sum of all the rows of that particular column of the input signal.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583. There is an exception to the general description provided in Output Format, on page 583 . If the output format is set to Automatic for an Add block, the data type of the first input is propagated to the output. The figure below illustrates the behavior.
Output word length	Output Word Length, on page 584.
Output fraction length	Output Fraction Length, on page 584.
Output data type	Output Data Type, on page 584.

In the following example, all the adders have Output Format set to Automatic. The input of the first adder is 0, because it goes through a Constant block, and the input data type is sfixed. The other input is through a port, with a data type of sfixed_en5. When this model is updated, all the adders have an output of sfixed, because that is the data type of the first input.



Output saturate on overflow, Output round on underflow

Determine how overflow and underflow are treated. These options are available if Output Format is set to Automatic or Specify. You can get overflow when Output Format is set to Automatic, because in this case the output data type for the Add block is inherited from the first input of the adder, so overflow can occur at the output.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	See Underflow Rounding Options, on page 585 for details about the rounding options available.

Sum of Rows

When enabled, the sum of all rows for each column in input matrix can be obtained at the output. The output is a vector whose size equals the number of columns in the input matrix. When this is enabled, you can only have a single input adder with + operation.

Inherit port

Determines whether the tool creates an inherit port. The tool creates an inherit port when you enable the option. If you enable the option, the tool applies automatic scalar expansion to the inherit and data ports. If one input is scalar and the other is vector, the scalar input is expanded to the size of the vector input.

This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Special Variables, on page 588](#) for information about these variables.
- Use the inherit option to specify the Output data type. See [Output Data Type, on page 101](#) for a description of the option.

Inherit sample time

Determines whether the output inherits the sample time from the inherit port. Enabling this option means that the output port inherits the sample time from the inherit port, and disabling it means the output port inherits sample time from the input. This option becomes available when you enable Inherit port.

Sample time factor

Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

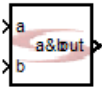
SMC Binary Logic

Calculates bitwise binary logic functions on both scalar and vector inputs.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Binary Logic block implements bitwise binary logic functions. The input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

If the block is fed by vector inputs, they must be the same size. In vectorized mode, the tool handles each input channel independently and calculates the corresponding output channel according to the specified expression, treating it as if a single Binary Logic block is replicated for each input channel.

Automatic Scalar Expansion

If the block has some scalar inputs and other vector inputs, the tool expands the scalar inputs to the size of the vectors. The vector inputs must be the same size.

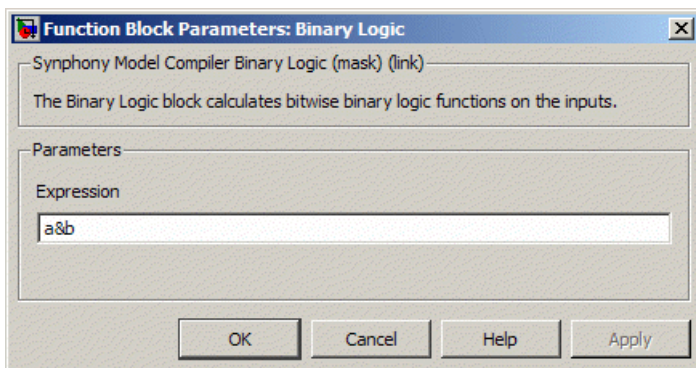
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Binary Logic Parameters



Expression

Specifies the logic operation performed by the block. For information about rules for the operation, see [Rules for Expressions, on page 55](#). The operation can be any of the following:

- Binary operations

Operator	Description
&	AND implements an AND operation, where the output is TRUE if all inputs are TRUE.
	OR implements an OR operation, where the output is TRUE if at least one input is TRUE. This is the default.
^	XOR implements an XOR operation, where the output is TRUE if an odd number of inputs are TRUE.
~&	NAND implements a NAND operation, where the output is TRUE if at least one input is FALSE.
~	NOR implements a NOR operation, where the output is TRUE if no inputs are TRUE. Remember that ~ is not equal to ~.
~^, ^~	XNOR implements an XNOR operation.

- Unary operations

Operator	Description
~	Not

- Reduction operations that do bitwise operations on a single operand to produce a single-bit output

&a = 0|1

|a

^a

~&a

~|a

~^a

Rules for Expressions

Follow these guidelines when you specify the binary logic operations:

- The inputs must be integers of the same size. You cannot use signed and unsigned integers together. If you do, you can get unexpected outputs, because the sign bit accepted as the part of the number.
- The expression must not start with an underscore (`_`).
- Precedence for the operators is from left to right.
- The operands for each binary operation must be the same size. For example, with the `a&b` expression, `a` and `b` must have same word length.
- Curly brackets `{}` are the expand operators. Operands inside curly brackets must be 1 bit wide, and they are expanded to the size of next expression.

Take `{a}&b` for example, where `a` is 1 bit and `b` is 8 bits. The expression takes `a` and expands it to 8 bits by adding the LSB value to the expanded bits. It then ands it with the operand `b`.

- The number of inputs is limited to 32.

SMC Black Box

Allows you to embed other blocks or IP in a Synphony Model Compiler design.

Library

Synphony Model Compiler [Ports & Subsystems](#)

Description



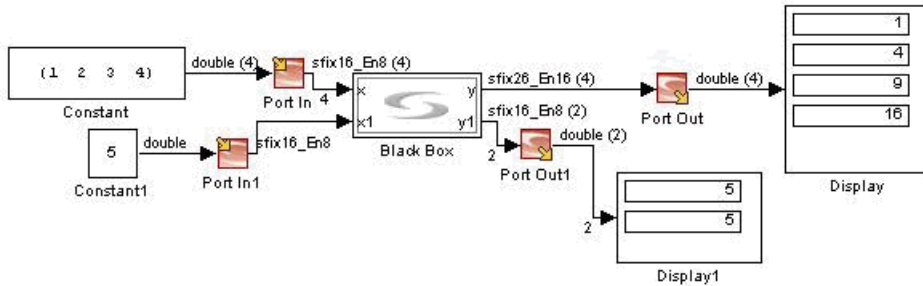
The Synphony Model Compiler Black Box block implements a black box, which allows you to embed other blocks in a Synphony Model Compiler design. For the purposes of simulation with Simulink, the black box is transparent; however, for RTL generation, the contents of the block will just be a black box. See the `<install_dir>\mathworks\toolbox\Synopsys\Synhls\demos\examples` directory for an example.

Use this block for IP for which you do not have access to the RTL code. If you have access to the RTL code, use the Smart Black Box block ([SMC Smart Black Box, on page 532](#)) instead. For details about using a black box in your design, see [Using Black Boxes and Third-Party IP, on page 777](#).

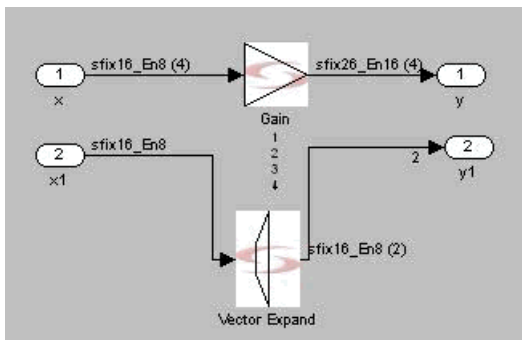
The Black Box consists of just an input and an output, to which you can add other blocks:



The Black Box supports vector inputs. If the input is a vector, the Input port in the black box is duplicated and connected to the signal coming outside the black box. Output vectors are handled in the same way.



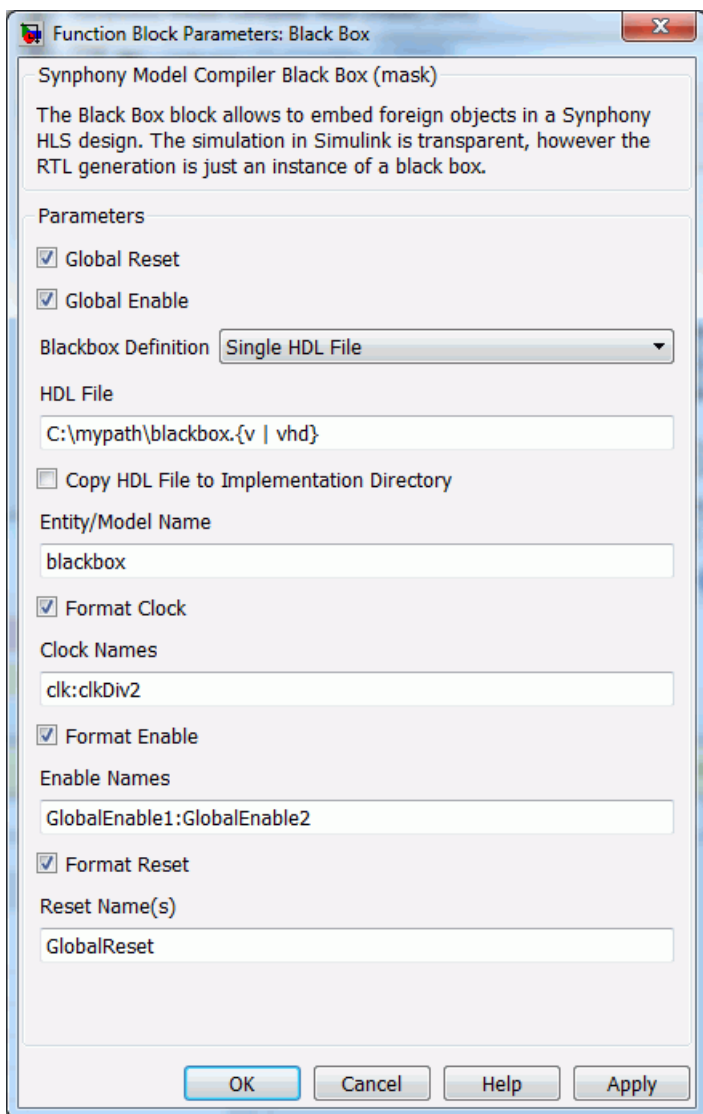
In the design shown above, the Port In block is the input to the x port of the Black Box block. In the HDL, the top-level input ports Port_In_e3, Port_In_e2, Port_In_e1, and Port_In_e0 are connected to the duplicated input ports x_e0, x_e1, x_e2, and x_e3, which are inside the black box.



Latency

Latency is determined by the contents of the black box.

Black Box Parameters



The dialog box titled "Function Block Parameters: Black Box" contains the following sections and controls:

- Synphony Model Compiler Black Box (mask)**
The Black Box block allows to embed foreign objects in a Synphony HLS design. The simulation in Simulink is transparent, however the RTL generation is just an instance of a black box.
- Parameters**
 - ☒ Global Reset
 - ☒ Global Enable
 - Blackbox Definition: **Single HDL File** (dropdown menu)
 - HDL File: **C:\mypath\blackbox.{v | vhd}** (text field)
 - ☐ Copy HDL File to Implementation Directory
 - Entity/Model Name: **blackbox** (text field)
 - ☒ Format Clock
Clock Names: **clk:clkDiv2** (text field)
 - ☒ Format Enable
Enable Names: **GlobalEnable1:GlobalEnable2** (text field)
 - ☒ Format Reset
Reset Name(s): **GlobalReset** (text field)

Buttons at the bottom: OK, Cancel, Help, Apply.

For information about setting these parameters, see [Setting Black Box Parameters, on page 780](#). The following are explanations of these parameters:

Global Reset

When enabled, the tool adds a single reset port to the instantiated black box and ties this port to the global reset in the RTL generated after DSP synthesis. It also makes the Format Reset option available, where you specify the global reset.

Global Enable

When enabled, the tool adds an enable port for each clock domain of the black box. The enable ports are tied to the global enable ports in the RTL generated after DSP synthesis. It also makes the Format Enable option available, where you can specify the global enables.

Black Box Definition

Determines the mode used to define the black box.

- Single HDL File specifies that the black box definition is in a single Verilog or VHDL file (.v or .vhd). Selecting this option makes the HDL File and Entity/Model Name options available, where you can specify additional parameters.
- Single EDIF File specifies that the black box definition is in a single EDIF file. Selecting this option makes the EDIF File, Simulation File, and Entity/Model Name options available, where you specify additional parameters.
- Import File List specifies that the black box definition is in multiple HDL and EDIF files. Selecting this option makes the Black Box File List and Entity/Model Name options available.
- Undefined specifies that there is no black box definition available, as when the black box is defined in some other black box block. Selecting this option makes the Entity/Model Name option available.

HDL File

Specifies the absolute path to the single HDL file that defines the black box; for example, C:\mypath\blackbox.v. This option is only available when you set Black Box Definition to Single HDL File. The file you specify is added to the project file and the simulator .do files.

Copy HDL File to Implementation Directory

When enabled, copies the HDL file specified in HDL File to the implementation directory. This options is only available when you set Black Box Definition to Single HDL File.

EDIF File

Specifies the absolute path to the single EDIF file (.edf or .edif) that defines the black box. This option is only available when you set Black Box Definition to Single EDIF File. The file you specify is added to the project file.

Simulation File

Specifies the absolute path to an HDL file that contains the behavioral simulation model for the black box defined in the EDIF file. This option is only available when you set Black Box Definition to Single EDIF File. The behavioral model can be a Verilog or VHDL file (.v or .vhd). The specified file is added to the simulator .do files.

Black Box File List

Specifies the absolute path to a single text file that lists all the Verilog, VHDL, and EDIF files that define the black box. This option is only available when Black Box Definition is set to Import File List.

The list must contain absolute paths to the files. Valid file extensions for black box definition files in the list file are .v, .vhd, .edf, and .edif. For example, if your black box is defined in three files called bb1.v, bb2.v and bb3.vhd, create and save a text file (bblist.txt) that contains the absolute paths to the black box definition files:

```
C:\mypath\bb1.v  
C:\mypath\bb2.v  
C:\mypath\bb3.vhd
```

Specify the path to the text file (C:\mypath\bblist.txt) in the Black Box File List field. All listed files are added to the project file. The Verilog and VHDL files are also added to the simulator .do files.

Entity/Model Name

Specifies the top-most entity or model for the black box in the RTL. The name you specify becomes the instance name for the black box and the name of the instantiated entity or model.

You can specify a variable entity or model name in this field using the `synStrEval` function. If you have a variable called `MyVariable` with the value `MySampleEntityName`, you can instantiate `MySampleEntityName` as the name for the top-level entity or model in the RTL by entering the following in this field:

```
synStrEval(MyVariable)
```


Format Clock

When enabled, the Clock Names option becomes available and lets you specify black box clock names. If it is disabled, the tool uses the Symphony Model Compiler convention for clock names, where the fastest clock is `clk`, and reduced frequency clocks are `clkDivN`.

Clock Names

Specifies clock names for the black box if you do not want to use the Symphony Model Compiler convention. This field becomes available when you select Format Clock. Type in the clock names, starting with the fastest clock, and using colons as separators. For example, if you have two clocks, `clk_sg` and `clk_2_sg`, type `clk_sg:clk_2_sg` in this field.

Format Enable

When enabled, the Enable Names option becomes available and lets you specify black box enable names. If it is disabled, the tool uses the Symphony Model Compiler convention for enable names, where the fastest domain enable signal is `GlobalEnable1`, and N-reduced enable signals are `GlobalEnableN`.

Enable Names

Specifies enable names for the black box if you do not want to use the Symphony Model Compiler convention. This field becomes available when you select Format Enable. Type in the enable names, starting with the time domain, and using colons as separators. For example, if you have two enables, `ce_sg` and `ce_2_sg`, type `ce_sg:ce_2_sg` in this field.

Format Reset

When enabled, the Reset Name option becomes available and lets you specify a name for the black box reset. If it is disabled, the tool uses the Symphony Model Compiler reset name, which is `GlobalResetSel`.

Reset Name

Specifies the reset name for the black box if you do not want to use the Symphony Model Compiler convention. This field becomes available when you select Format Reset. For example, if you have a reset called `grst`, type `grst` in this field.

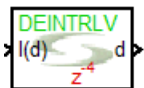
SMC Block Deinterleaver

Shuffles a fixed number of interleaved input symbols to obtain the original sequence.

Library

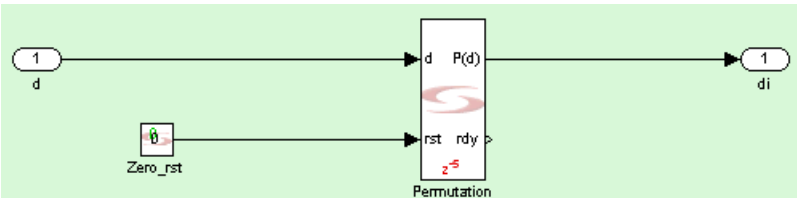
Synphony Model Compiler [Communications](#)

Description



This block shuffles a fixed number of input symbols according to the mapping you define to get the original sequence. This is a custom block; for information about custom blocks, see [Primitives and Custom Blocks, on page 800](#).

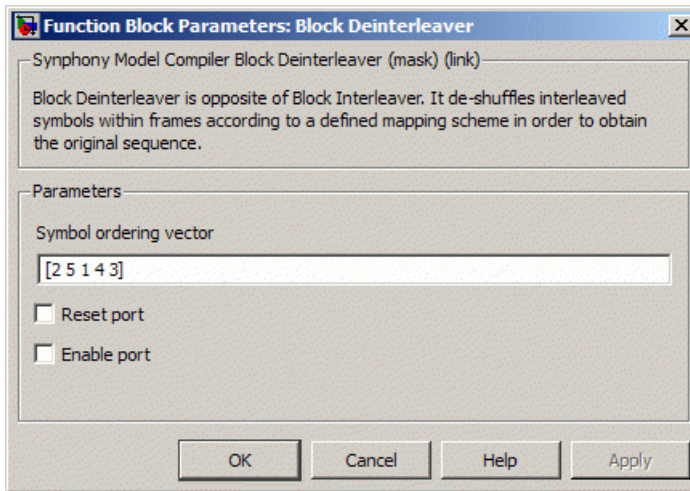
The following figure shows the internals of this block:



Icon Annotations

Note	Specifies that the block is a deinterleaver.
Latency	Is equal to the number of input symbols - 1.

Block Deinterleaver Parameters



Symbol ordering vector

Specifies the order for deinterleaving the input symbols. It operates on frames with a fixed number of symbols and shuffles them back to the original permutation, using all the symbols without missing any, and using each symbol only once.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable port. The enable port is connected to the enable signal of the internal shift registers.

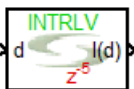
SMC Block Interleaver

Shuffles a fixed number of input symbols to a new permutation.

Library

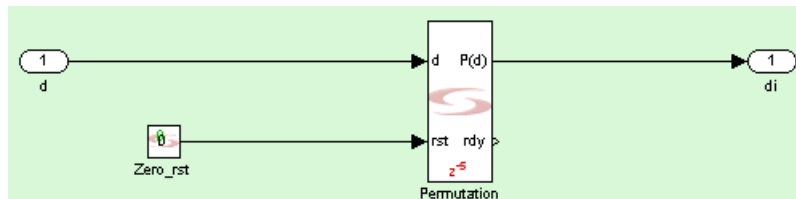
Synphony Model Compiler [Communications](#)

Description



This block shuffles a fixed number of input symbols to a new permutation, according to the mapping you define. This is a custom block; for information about custom blocks, see [Primitives and Custom Blocks, on page 800](#).

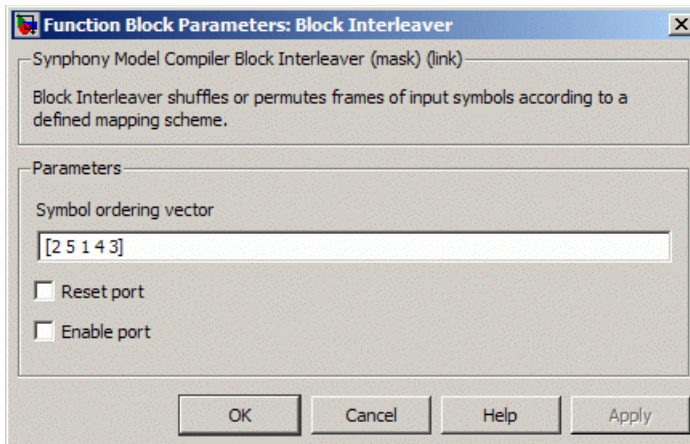
The following figure shows the internals of this block:



Icon Annotations

Note	Specifies that the block is an interleaver.
Latency	Varies with the number of input symbols.

Block Interleaver Parameters



Symbol ordering vector

Specifies the order for interleaving the input symbols. It operates on frames with a fixed number of symbols and shuffles them, using all the symbols without missing any, and using each symbol only once.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable port. The enable port is connected to the enable signal of the internal shift registers.

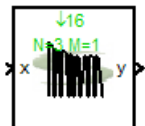
SMC CIC

Implements a CIC filter by applying cascaded integrator-comb (CIC) filtering on the input signal.

Library

Symphony Model Compiler [Filtering](#)

Description



This is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) that implements a CIC filter by applying cascaded integrator-comb filtering on the input signal. Cascaded Integrator-Comb filters are a type of linear phase FIR filter, and have a comb section and an integrator section. You can use this filter in either interpolating (upsample) or decimating (downsample) mode.

The SMC library also includes another CIC block, CIC2, with additional features. See [SMC CIC2, on page 70](#) for a description of this block.

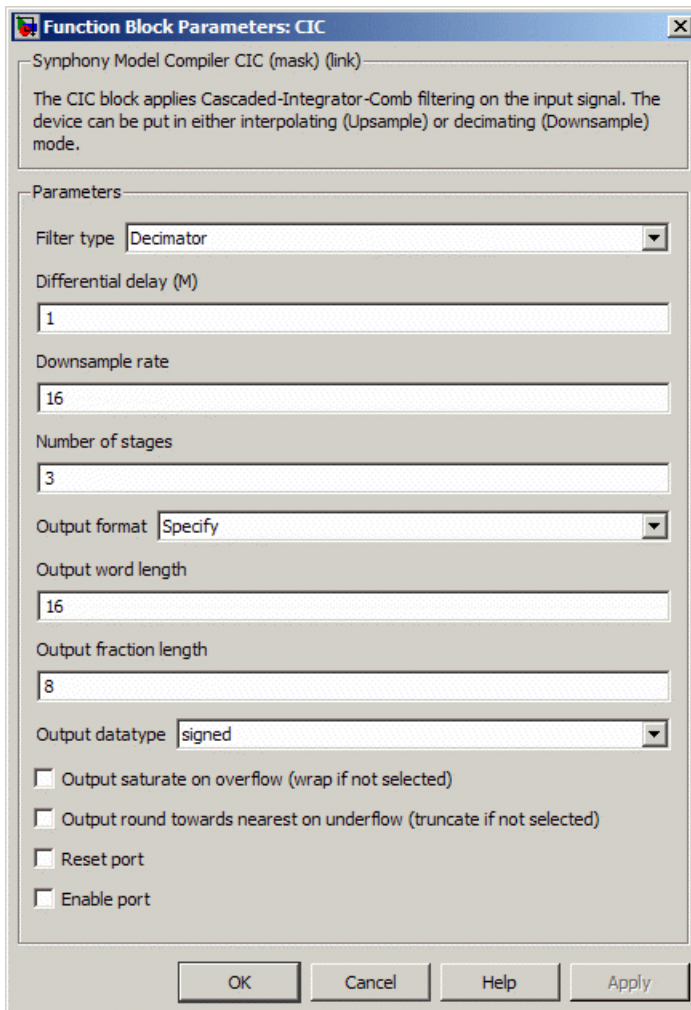
Automatic Scalar Expansion

If the data input is a vector and the reset or enable port is scalar, the tool expands the scalar reset or enable port to the size of the data input vector. The reset and enable can be either vector or scalar.

Latency

This block has no latency. In releases prior to 2.6, the CIC block had a latency of 1.

CIC Parameters



The dialog box titled "Function Block Parameters: CIC" contains the following elements:

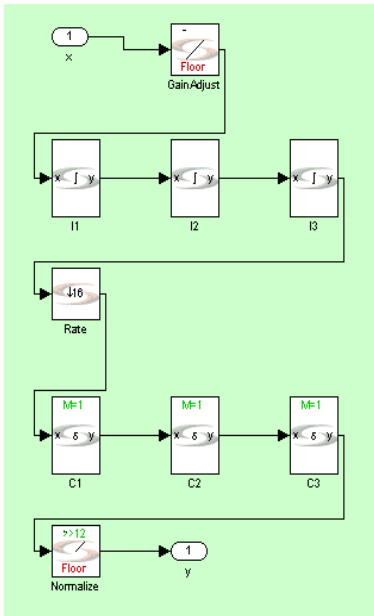
- Header:** "Synphony Model Compiler CIC (mask) (link)" with a close button.
- Description:** "The CIC block applies Cascaded-Integrator-Comb filtering on the input signal. The device can be put in either interpolating (Upsample) or decimating (Downsample) mode."
- Parameters Section:**
 - Filter type:** A dropdown menu set to "Decimator".
 - Differential delay (M):** A text input field containing "1".
 - Downsample rate:** A text input field containing "16".
 - Number of stages:** A text input field containing "3".
 - Output format:** A dropdown menu set to "Specify".
 - Output word length:** A text input field containing "16".
 - Output fraction length:** A text input field containing "8".
 - Output datatype:** A dropdown menu set to "signed".
 - Checkboxes:**
 - ☐ Output saturate on overflow (wrap if not selected)
 - ☐ Output round towards nearest on underflow (truncate if not selected)
 - ☐ Reset port
 - ☐ Enable port
- Buttons:** "OK", "Cancel", "Help", and "Apply" at the bottom.

Filter Type

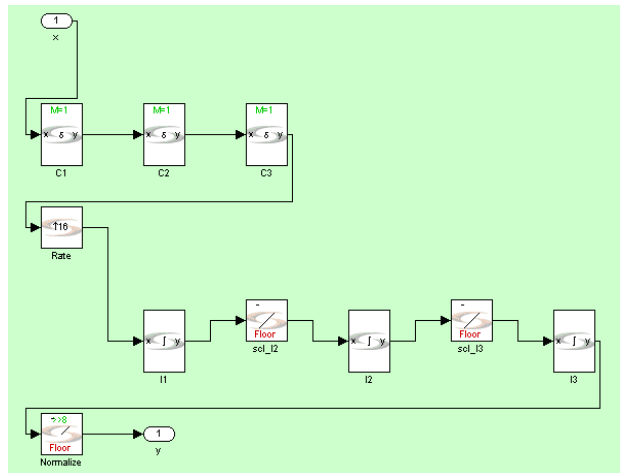
Determines the type of filter. The next figure shows how the filters are implemented, without resets and enables.

- **Decimator** uses downsampling mode and implements a CIC filter that performs a sample rate decrease on an input signal.

- Interpolator uses upsampling mode and implements a CIC filter that performs a sample rate increase on an input signal.



Decimator



Interpolator

Differential Delay (M)

Specifies the differential delay of the comb portion of the filter. Internally, the CIC filter uses differentiators, and the value of this parameter is passed to all differentiators in the CIC filter.

Upsample/Downsample Rate

Determines the interpolation or decimation rate for the filter, depending on the mode you selected in Filter Type.

Number of Stages

Specifies the number of filter stages. The CIC filter uses differentiators and integrators internally, and this number equals the number of differentiator/integrator pairs in the CIC filter.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round towards nearest on underflow

Determine how overflow and underflow are treated. These options are available when Output format is set to Specify.

Output saturate on overflow	Saturates the overflow when the option is enabled and wraps the overflow when it is disabled. See Overflow Saturation Options, on page 585 for details.
Output round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable port. The enable port is connected to the enable signal of the internal shift registers.

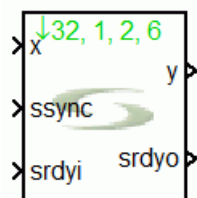
SMC CIC2

Implements a CIC filter by applying cascaded integrator-comb (CIC) filtering on the input signal.

Library

Synphony Model Compiler [Filtering](#)

Description



The CIC2 custom block implements a CIC filter by applying cascaded integrator-comb filtering on the input signal. Cascaded integrator-comb filters are a type of linear phase FIR filter, with a comb section and an integrator section. You can use this filter in either interpolating (upsample) or decimating (downsample) mode.

The CIC2 block offers many enhancements over the CIC block, such as enhanced flow control and mulichannel support. It also supports folding across differentiators or channels of the differentiator stages in decimation mode, and supports enabled datapath designs with either a partial clock frequency change or single clock operation for enabled inputs or outputs.

This block also supports a variable decimation factor and non-recursive decimation architectures. Currently, the tool only supports power of two decimation factors for non-recursive decimation.

CIC2 Flow Control

The following flow control ports are always available for the CIC2 block:

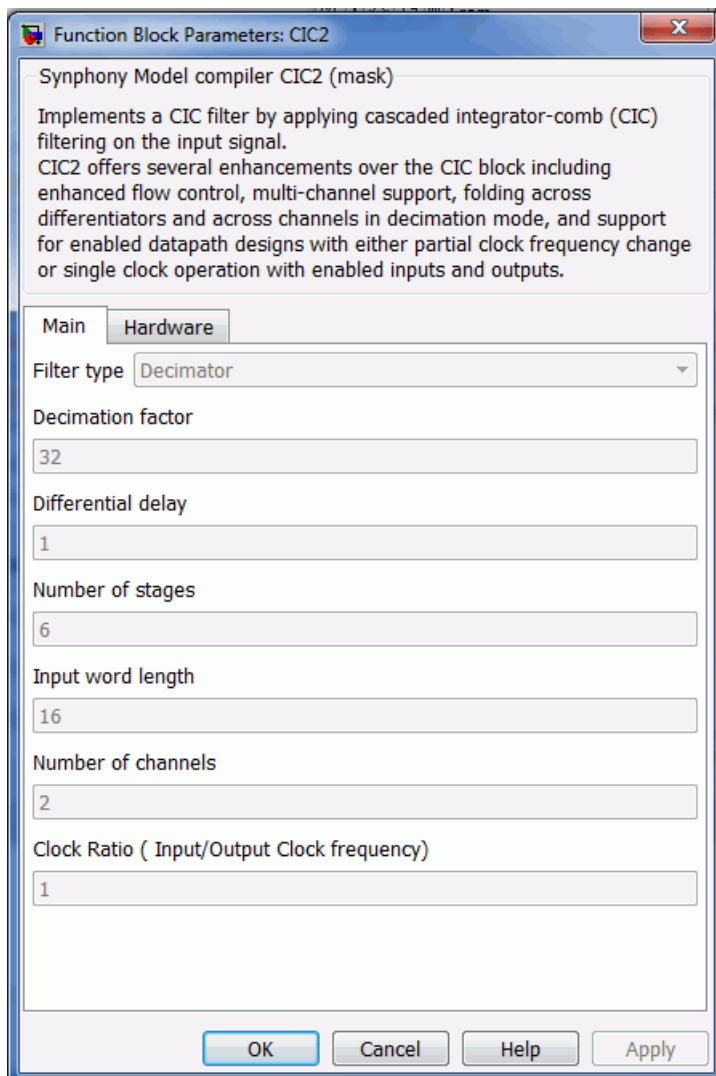
ssync	The ssync (source sync) input can be forced high to reset the integrator stages to zero.
srnyi	The srnyi (source ready) input port can qualify whether the input data in the current sample period is valid. An invalid input sample is indicated by srnyi going low. The presence of srnyi allows for interpolation without changing the clock rate by the corresponding amount.
srnyo	The srnyo (source ready) output port can qualify whether the current output sample is valid. An invalid output sample is indicated by srnyo going low. The presence of srnyo allows for decimation without changing the clock rate by the corresponding amount.

For a multichannel operation, flow control ports are scalar and the same flow control signal applies to all channels.

Icon Annotations

Top	The green annotations specify the following information: <ul style="list-style-type: none">• Decimation factor• Clock ratio• Number of channels,• Number of stages
-----	---

CIC2 Parameters



The image shows a dialog box titled "Function Block Parameters: CIC2". It has a description of the block and a "Main" tab with various parameters.

Symphony Model compiler CIC2 (mask)
Implements a CIC filter by applying cascaded integrator-comb (CIC) filtering on the input signal.
CIC2 offers several enhancements over the CIC block including enhanced flow control, multi-channel support, folding across differentiators and across channels in decimation mode, and support for enabled datapath designs with either partial clock frequency change or single clock operation with enabled inputs and outputs.

Main | Hardware

Filter type: Decimator

Decimation factor: 32

Differential delay: 1

Number of stages: 6

Input word length: 16

Number of channels: 2

Clock Ratio (Input/Output Clock frequency): 1

Buttons: OK, Cancel, Help, Apply

Main Tab

This tab sets parameters for filter type, decimation factor, differential delay, number of stages, input word length, number of channels, and clock ratio.

Filter Type

Determines the type of filter.

Decimator	Uses downsampling mode and implements a CIC filter that decimates the input signal. Decimators can be implemented with a full sample rate change. They can also be implemented with either a partial sample rate change or a single rate mode by specifying the appropriate Clock Ratio.
Interpolator	Uses upsampling mode and implements a CIC filter that interpolates the input signal. Interpolators can be implemented with a full sample rate change. They can also be implemented with either a partial sample rate change or a single rate mode by specifying the appropriate Clock Ratio.
Variable Rate Decimator	Implements a CIC filter where you can use the <code>ratei</code> input port to program the decimation factor. With this option, the tool sets Clock Ratio internally to 1, and you cannot specify the Folding and Pipelined implementation options on the Hardware tab. This option uses a variable shift operation at the output, which may cause the maximum speed that can be achieved to be significantly lower.
Non-recursive Decimator	Implements a power of two CIC decimator by using only feed-forward addition operations, thus avoiding large wordlength growth in the feedback integrator sections. You can program the $\log_2(\text{decimation factor})$ through a <code>ratei</code> input port. The decimation rates are all integer powers of two. With this option, the tool sets Clock Ratio internally to 1 and disables the Folding and Pipelined implementation options on the Hardware tab.

Decimation/Interpolation Factor

Determines the interpolation or decimation rate for the filter, depending on the implementation you selected for Filter Type.

Filter Type	Option Description
Decimator / Interpolator	Sets the decimation rate or interpolation rate for the filter
Variable Rate Decimator	Sets the maximum decimation rate for the filter.
Non-recursive decimator	Sets the decimation rate to $\log_2(\text{Maximum Decimation Factor})$.

Differential Delay (M)

Specifies the differential delay of the comb portion of the filter. Internally, the CIC filter uses differentiators, and the value of this parameter is passed to all differentiators in the CIC filter. This option is not available if you selected a non-recursive decimator in Filter Type.

Number of Stages

Specifies the number of filter stages. The CIC filter uses differentiators and integrators internally, and this number equals the number of differentiator/integrator pairs in the CIC filter.

Input Word Length

Specifies the word length of the input samples.

Number of Channels

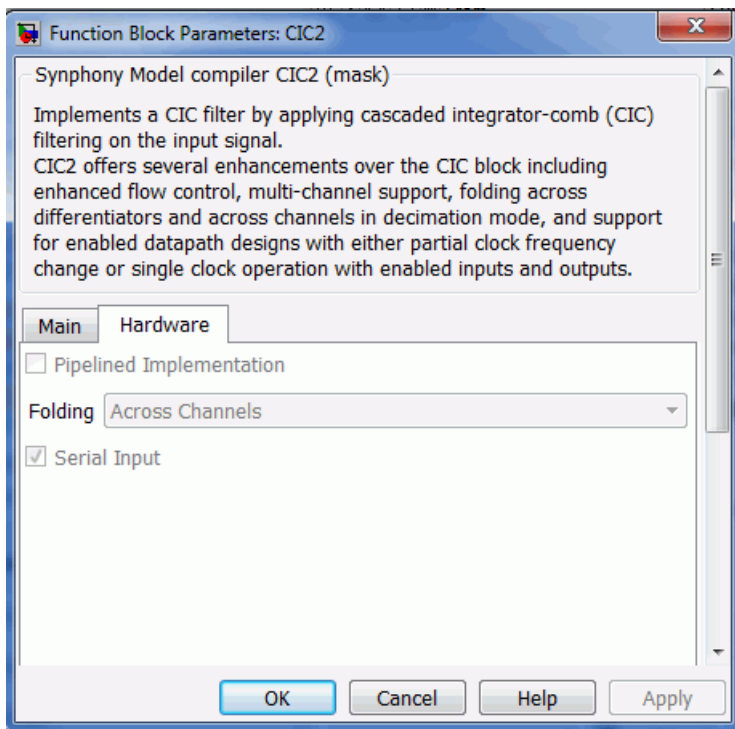
Specifies the number of channels implemented for the CIC filter. The input x must be a vector, for which its size is equal to the number of channels. The output sample y is scalar if decimators are selected for the fold across channel option. Otherwise, the size of the vector is same as the input.

Clock Ratio (Input/Output Clock Frequency)

Specifies the ratio of the input to output clock frequencies for the decimator and output to input clock frequencies for the interpolator. This value must be an integer factor of the interpolation/decimation factor. If the decimation factor is 16, then the values for the clock ratio can be 1, 2, 4, 8, 16.

Hardware Tab

This tab sets parameters for pipelined implementation, folding, and serial input.



Pipelined Implementation

Valid only in decimation mode. Specifies whether the tool inserts pipeline stages after every differentiator/integrator. This option is not available if you selected a variable rate decimator or non-recursive decimator in Filter Type.

Folding

Valid for decimators only, but it is not available for variable rate decimator or non-recursive decimator implementations. You can specify these folding options:

- None
- Fold across differentiators
This option is available only when the clock ratio is greater than $\text{ceil}(\text{number of stages}/2)$. In this case, a folding factor of the clock ratio is applied that folds the differentiators into a single MAC (multiply accumulate).
- Across channels
This option folds across channels for the differentiator bank.

Serial Input

This option is only available when you specify folding across channels and clock rate=1. The input x must be a scalar input of the commutated channels that results in a single clock implementation.

Non-recursive Decimator architecture

Specifies how decimation by two stages in a non-recursive architecture is to be implemented.

MCM	Each stage is implemented as a transpose MCM filter.
Cascaded Adder	Each stage is implemented as a cascaded chain of $(1+z^{-1}) z^{-1}$ stages.

The MCM options is more resource-efficient for designs with a large value for Number of stages, while Cascaded Adder yields more resource-efficient hardware when Number of Stages is small.

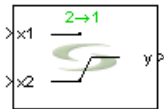
SMC Commutator

Sequentially switches the data from multiple input ports to a single output port, increasing the data rate of each output port accordingly (time division multiplexing). The Commutator block provides optional flow control, multi-channel, and single-clock multi-rate support.

Library

Synphony Model Compiler [Signal Operations](#)

Description



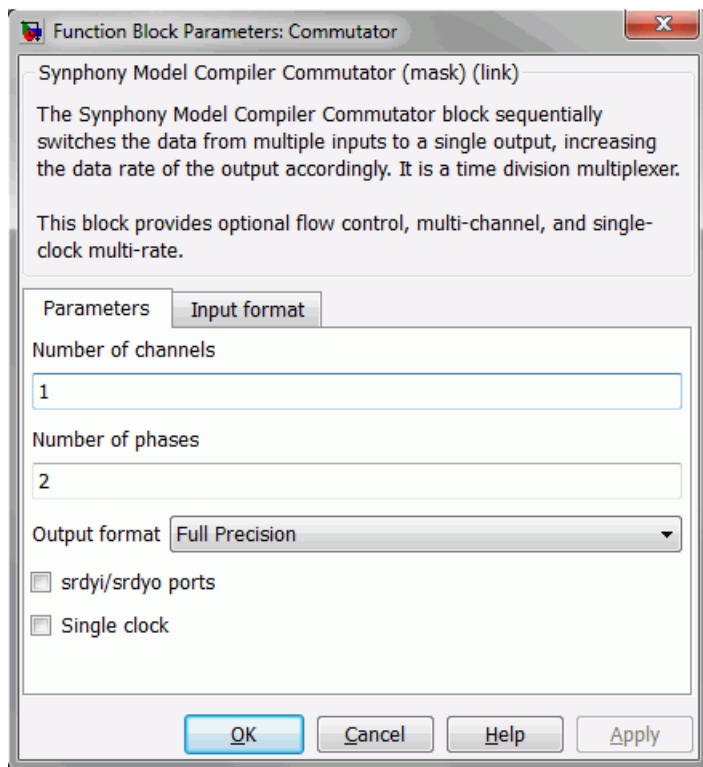
The Synphony Model Compiler Commutator block sequentially switches the data from multiple input ports to a single output port. In order to sequentially multiplex input data without missing a sample, the output data rate is increased by a factor of the number of input ports. This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition).

Icon Annotation

The icon for this block displays the following information:

Top Annotation	Shows the number of input ports to be multiplexed to a single output port.
Latency Annotation	Zero latency.

Commutator Parameters



Number of channels

Specifies the number of channels processed. The format of the input data depends on the Input format parameters described in the sections: [Scalar Input Format, on page 79](#), [Vector Input Format, on page 80](#), and [Matrix Input Format, on page 82](#).

Number of phases

Specifies the number of inputs or phases (per channel) from which data is multiplexed to the output. The format of the input data depends on the Input format parameters described in the sections: [Scalar Input Format, on page 79](#), [Vector Input Format, on page 80](#), and [Matrix Input Format, on page 82](#).

Output format, Output word length, Output fraction length, and Output data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

srdyi/srdo ports

When enabled, the block provides forward flow control. **srdyi** (Source Ready Input) indicates that the current input data is valid. **srdo** (Source Ready Output) is used to chain the Commutator block to other flow control blocks. When Single clock is enabled, these ports are required.

Single clock

When enabled, the block does not introduce a new sample time on the output. It creates a single-clock multi-rate implementation instead.

For Single clock mode, the inputs are provided in the fast domain and the **srdyi/srdo** ports are required. **srdyi** cannot be active more than 1 of N samples, where N is the number of inputs (per channel). If this requirement is not met, the behavior is undefined. **srdo** is pulse stretched by N samples; this indicates the output is valid.

Scalar Input Format

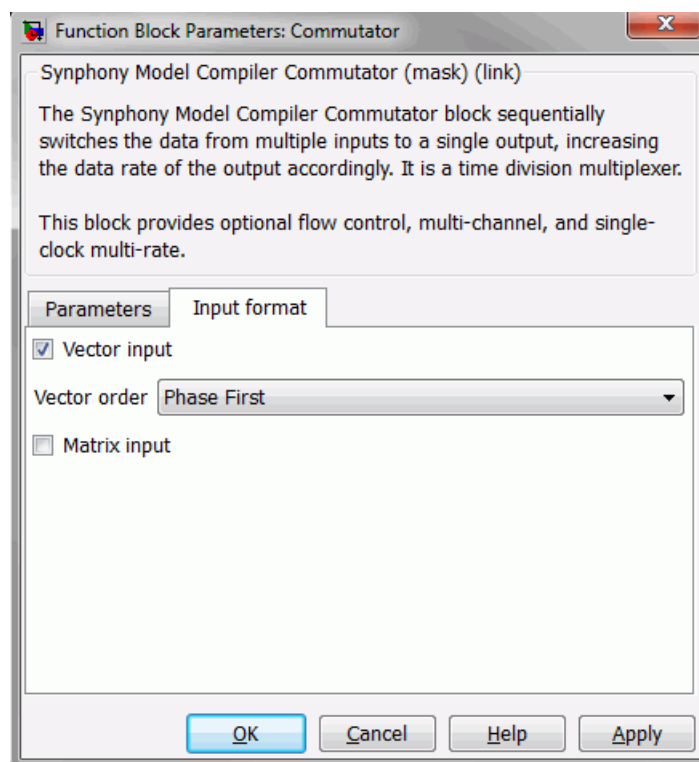
The way the Commutator block accepts input data depends on how many channels are required and the state of the Vector input and Matrix input parameters. See [Vector Input Format, on page 80](#) and [Matrix Input Format, on page 82](#). Scalar input is only available when the Vector input option is disabled.

Each input is a separate port to the block. $x_1 \dots x_N$ are the inputs to the first channel, $x_{1+N} \dots x_{2N}$ are the inputs to the second channel, and $x_{1+(C-1)*N} \dots x_{C*N}$ are the inputs to the last channel.

Example of a 3-phase, 2-channel scalar input:

Port	Input	Channel
x1	1	1
x2	2	1
x3	3	1
x4	1	2
x5	2	2
x6	3	2

Vector Input Format



Vector input

When enabled, the block accepts a single vector input for all data. The Vector order parameter determines the format of the input data.

Vector order

Vector input must be enabled and Matrix input disabled for this option to be available. A single port and the data is provided as a vector of length $N \times C$, where N is the number of phases and C is the number of channels. The Vector order parameter determines the order of elements in the vector.

Vector order	Description
Phase First	1 st phase $e1 \dots eC$ 2 nd phase $e1+C \dots eC+C$ N th phase $e1+(N-1)*C \dots eN*C$
Channel First	1 st channel $e1 \dots eN$ 2 nd channel $e1+N \dots e2N$ C th channel $e1+(C-1)*N \dots eC*N$

Example of a 2-phase, 3-channel, phase first vector input:

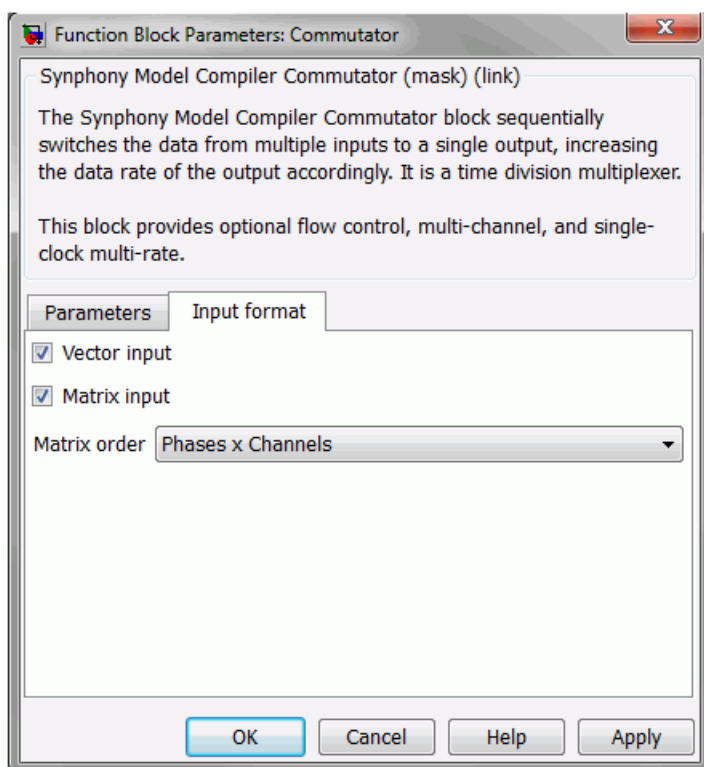
Element	Phase	Channel
1	1	1
2	1	2
3	1	2
4	2	1
5	2	2
6	2	3

Example of a 2-phase, 3-channel, channel first vector input:

Element	Phase	Channel
1	1	1
2	2	1
3	1	2

Element	Phase	Channel
4	2	2
5	1	3
6	2	3

Matrix Input Format



Matrix input

When enabled, the block accepts a single matrix input for all data. The Matrix order parameter determines the format of the input data. Vector input must be enabled for this option to be available.

Matrix order

Matrix input must be enabled for this option to be available. A single input port and the data is provided as a matrix. The Matrix order parameter determines the dimension of the matrix.

Matrix order	Description
Phases x Channels	NxC matrix, where N is the number of phases and C is the number of channels
Channels x Phases	CxN matrix, where C is the number of channels and N is the number of phases

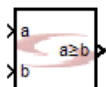
SMC Comparator

Implements a programmable comparator.

Library

Symphony Model Compiler [Math Functions](#)

Description



The Symphony Model Compiler Comparator block implements a comparator by comparing two signals and returning a single bit.

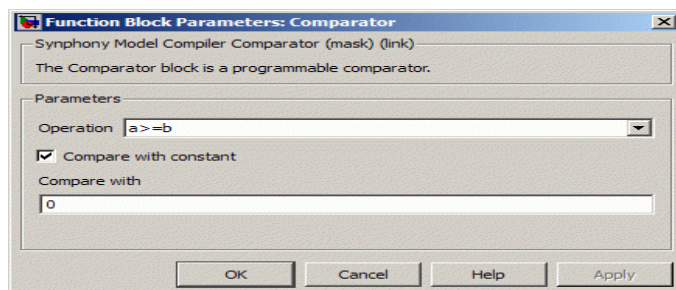
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Comparator Parameters



Operator

Determines the type of comparison to be performed on the two buses:

- $a == b$
- $a != b$
- $a < b$
- $a <= b$
- $a > b$
- $a >= b$ This is the default.

Compare with constant

When enabled, it compares a with the constant specified in Compare with, instead of b . Enabling this option makes the Compare with option available.

Compare with

Defines the constant to be used for comparison with a . This option becomes available only when Compare with constant is enabled. The defined constant is displayed on the block icon without being quantized, but while performing the specified operation it is first quantized to the input data type.

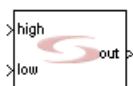
SMC Concat

Concatenates the bits of up to 32 input signals.

Library

Symphony Model Compiler [Signal Operations](#)

Description



The Symphony Model Compiler Concat block concatenates the bits of up to 32 input signals. This block converts the inputs to unsigned integers, by ignoring the binary point and maintaining the bit representation of the word. The output is an unsigned integer with the word length equal to the sum of the input word lengths. The software takes the bits of the hi input and makes them the most significant bits of the output. The bits of the lo input become the least significant bits of the output.

If the block has vector inputs, each vector element is concatenated with the corresponding one. Vector inputs must be the same size.

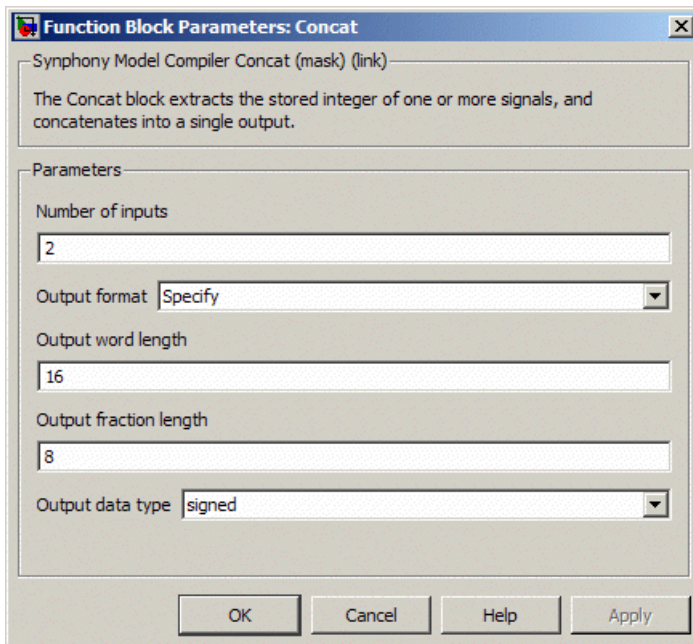
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Concat Parameters



Number of inputs

Specifies the number of input signals to be concatenated. The maximum number of input signals you can specify is 32. If you set the number of inputs to 1, the output is the stored integer bit representation of the input as an unsigned ufix value.

Output format, Output word length, Output fraction length, and Output data type

The output data format can be fully specified for this block. For descriptions of the following parameters, refer to this table:

Word length	Output Word Length, on page 584
Fraction length	Output Fraction Length, on page 584
Data type	Output Data Type, on page 584

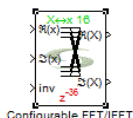
SMC Configurable FFT/IFFT

Implements a fully pipelined Fast Fourier Transform (FFT) or Inverse Fast Fourier Transform (IFFT).

Library

Symphony Model Compiler [Transforms](#)

Description



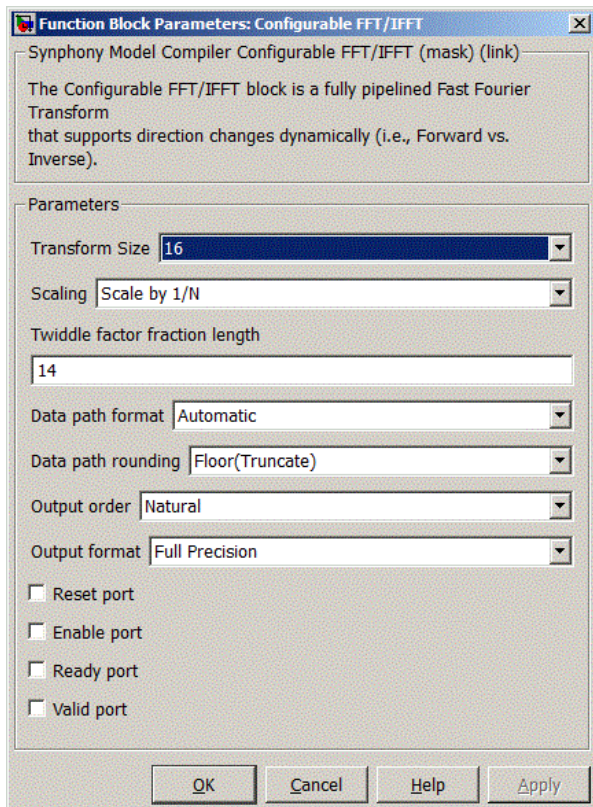
The Configurable FFT/IFFT block implements a fully pipelined Fast Fourier Transform (FFT) or Inverse Fast Fourier Transform (IFFT) based on whether `inv` port is low or high respectively. When it is used to perform the block-wise FFT of a streaming signal, you only require to reset the first block.

Latency

Latency is a complex function of frame size and output order:

Transform Size	Latency (Bit-reversed Output)	Latency (Natural Order Output)
16	19	36
32	38	71
64	70	135
128	137	266
256	265	522
512	524	1037
1024	1036	2061
2048	2063	4112
4096	4111	8208
8192	8210	16403
16384	16402	32787
32768	32789	65558
65536	65557	131094

Configurable FFT/IFFT Parameters



Transform Size

Sets the size of the FFT/IFFT block. For sizes of integer power 4, the software uses the Radix-4 algorithm. For other sizes, the software uses a Radix-2 stage, followed by Radix-4 stages.

Scaling

Specifies whether the FFT/IFFT is to be scaled. Scaling is applied after the butterfly stages to prevent bit growth from the beginning. Floor rounding (truncation) is used for the scaled data. See [Underflow Rounding Options, on page 585](#) for details.

N is the FFT/IFFT size. The following three options are available for scaling:

- Scale by $1/N$ divides the DFT summation by N.
- Scale by $1/\text{Sqrt}(N)$ divides the DFT summation by the square root of N.
- No scaling does not scale the FFT.

Twiddle factor fraction length

Determines the precision of the Configurable FFT/IFFT block by setting the fraction length for a twiddle factor, in bits. The specified value must be an integer between 1 and 50. Increasing the value of Twiddle factor fraction length increases precision. You can also specify the twiddle factor in terms of the variables `syn_inp_wl` and `syn_inp_fl`.

Data path format

Determines data path format. You can set one of these options:

- Automatic sets the data path format to the one that uses the maximum input and output fractions, and the smallest bit width that guarantees no overflow.
- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation after twiddle factor multiplications.
- Specify uses the user-defined data type to determine the cast for internal calculations. For this block, data path casting is done at the input, after the twiddle factor multiplications, and at the block output. Overflow only occurs at the points where data casting is done. The rest of the calculations are overflow-free, regardless of the specified data type.

Data path word length

Determines the word length of the data path in bits. It is only available when you set Data path format to Specify. You can also specify the word length in terms of the variables `syn_inp_wl`, `syn_inp_fl`, `syn_coef_wl`, and `syn_coef_fl`.

Data path fraction length

Sets the fraction length of the data path in bits. It is only available when you set Data Path Format to Specify. You can also specify the fraction length in terms of the variables `syn_inp_wl`, `syn_inp_fl`, `syn_coef_wl`, and `syn_coef_fl`.

Data path saturate on overflow

Determines how data path overflow is treated. When enabled, the option saturates the overflow. When disabled, it wraps the overflow. See [Overflow Saturation Options, on page 585](#) for details. This option is only available when you set Data path format to Specify.

Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 585](#) for details. This option becomes available when Data path format is set to Automatic or Specify.

Output Order

Sets the output order for the block. This option determines the latency of the block; see [Latency, on page 89](#) for a table of values.

- Natural is the default. It sets the output order of the FFT results to the natural order.
- Bit-reversed sets the pipelined FFT results to bit-reversed order.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584 . You can also specify word length in terms of variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_coef_wl</code> , and <code>syn_coef_fl</code> . See Special Variables, on page 588 , for details on these variables.
Output fraction length	Output Fraction Length, on page 584 . You can also specify fraction length in terms of variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_coef_wl</code> , and <code>syn_coef_fl</code> . See Special Variables, on page 588 , for details on these variables.
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding type

Determines how output overflow and underflow are treated. These options are only available when you set Output Format to Specify.

Output saturate on overflow	When enabled it saturates the overflow; when disabled, it wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Reset port

When enabled, it creates a local reset (`rst`) for the FFT block, clearing the pipeline. The reset is active high.

When disabled, the block outputs invalid data for the depth of the pipeline.

Enable port

When enabled, it creates an enable (`en`) port, which provides control over the Enable status of the block; however, you cannot use folding optimizations, as it leads to verification mismatches.

When disabled, the software does not create an `en` port and the FFT operation is always enabled.

Ready port

When enabled, this option outputs a ready pulse (`rdy`), and valid FFT data streams to the clock after the validity is asserted. A typical use of this pin is to feed the ready pin of a forward FFT to the reset pin of an inverse FFT. When disabled, the tool does not create a ready pin.

Valid port

When enabled, this option creates an active high signal (`vald`) that frames the valid output data. A typical use of this pin is to feed the valid pin of a forward FFT to the enable pin of an inverse FFT. When disabled, the tool does not create a valid pin.

SMC Constant

Sets a constant value of a specified data type as the output.

Library

Synphony Model Compiler [Sources](#)

Description



The Synphony Model Compiler Constant block sets the output of the block to a constant value of a specified data type.

```
Y[n] = <constant>
```

The value is cast to the specifications of the data format, and also displayed in the icon of the instance. The sample period of the constant is usually inherited through back inheritance from the rest of the design, but you can use the parameters to force the sample period.

Icon Annotation

The icon for this block displays the following information:

Top Annotation	Displays the constant value. If you enter an expression or a variable, you can use this annotation to identify the constant.
Latency Annotation	There is no latency, and the block drives the same value independent of the global reset or enable.

Constant Parameters

Source Block Parameters: Constant

Synphony Model Compiler Constant (mask) (link)

The Constant block drives a constant value of a specified data type.

Parameters

Constant value
0

Constant fraction length
0

Constant data type **unsigned**

☐ Constant round towards nearest on underflow (truncate if not selected)

Sample time (Use -1 to inherit)
-1

OK Cancel Help

Constant value

Sets the value to be driven (the output value of the block). For vectorized output, specify a row or column vector. For matrix output, specify a matrix value. Each value corresponds to a different channel.

Constant fraction length and Constant data type

The output data format must be fully specified for this block. See the following for details:

Constant fraction length	Output Fraction Length, on page 584
Constant data type	Output Data Type, on page 584

Constant round towards nearest on underflow

Determines how the underflow for the constant is treated. Enable the option to round the underflow using the Nearest algorithm, and disable it to round the overflow with the Floor (truncate) algorithms. See [Underflow Rounding Options, on page 585](#) for details.

Sample Time

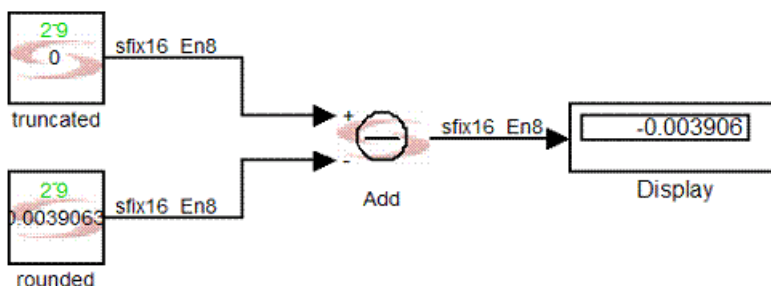
Sets the sample time. Use -1 to inherit.

Constant Block Examples

The following examples show the Constant block and the value of the green annotation at the top of this block.

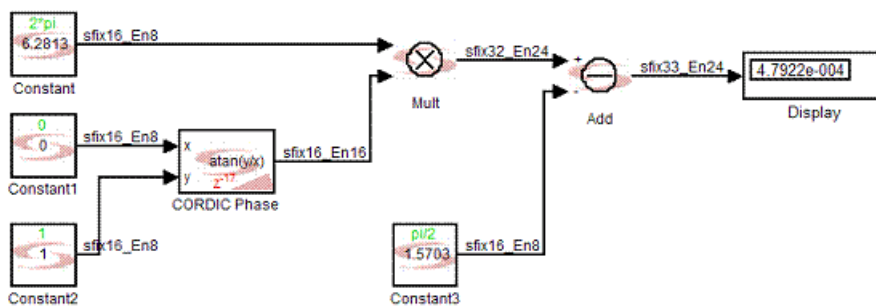
Example 1

The first example shows the importance of truncating versus rounding. The note shows a value outside the range of the given data format. Rounding will set the LSB anyway:



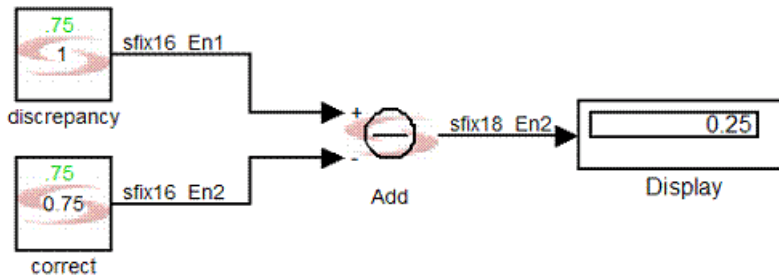
Example 2

The second example shows the convenience of the note when you use variables or built-in constants. Further, in this test case, the sample period can only be derived if specified in at least one of the constant blocks.



Example 3

This third example illustrates how the notes can reveal a quantization issue, when the note is different from the quantized value.



Diagnostics

Warning: value can not be represented with selected data type.

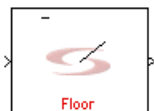
SMC Convert

Changes the word size and data type of the input. You can apply a constant shift before the new word size and data type are cast.

Library

Symphony Model Compiler [Signal Operations](#)

Description



The Symphony Model Compiler Convert block changes the word size and data type of the input. Most Symphony Model Compiler blocks have an option to provide a built-in cast on the output. This block explicitly casts a signal, with an optional shift. You can apply the constant shift before the word size and data type change.

The quantization of a signal is determined by the quantization propagated from input signals. Each block in the Symphony Model Compiler blockset calculates the quantization of the outputs based on block-specific rules and the quantization on the inputs. You can also manage the quantization of a signal directly with a block cast operation inside the block, or by putting the Convert block (Symphony Model Compiler Signal Operations library) at the output of the block.

Binary Point Examples

The position of the binary point determines how fixed-point numbers are interpreted. The binary point is the means by which fixed-point numbers are scaled. The following table shows how the binary point position affects the five-bit binary number 10110, using signed and unsigned arithmetic:

	Signed (two's complement) arithmetic	Unsigned arithmetic
10110.	$-2^4 + 2^2 + 2 = -10$	$2^4 + 2^2 + 2 = 22$
10.110	$-2 + 2^{-1} + 2^{-2} = -1.25$	$2 + 2^{-1} + 2^{-2} = 2.75$
1.0110	$-2^0 + 2^{-2} + 2^{-3} = -0.625$	$2^0 + 2^{-2} + 2^{-3} = 1.375$

The following table contains an example of input values and results:

A Convert block with these input parameters...	Gives you these results...
A <code>sfix5_0</code> signed input 10110 to the Convert block (word length = 5, fraction length = 0)	Input 10110. (-10)
A left shift over 3 bits	Shift 10.110
A cast towards a <code>sfix4_2</code> signed output (word length is 4, fraction length is 2)	Output 10.110

Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

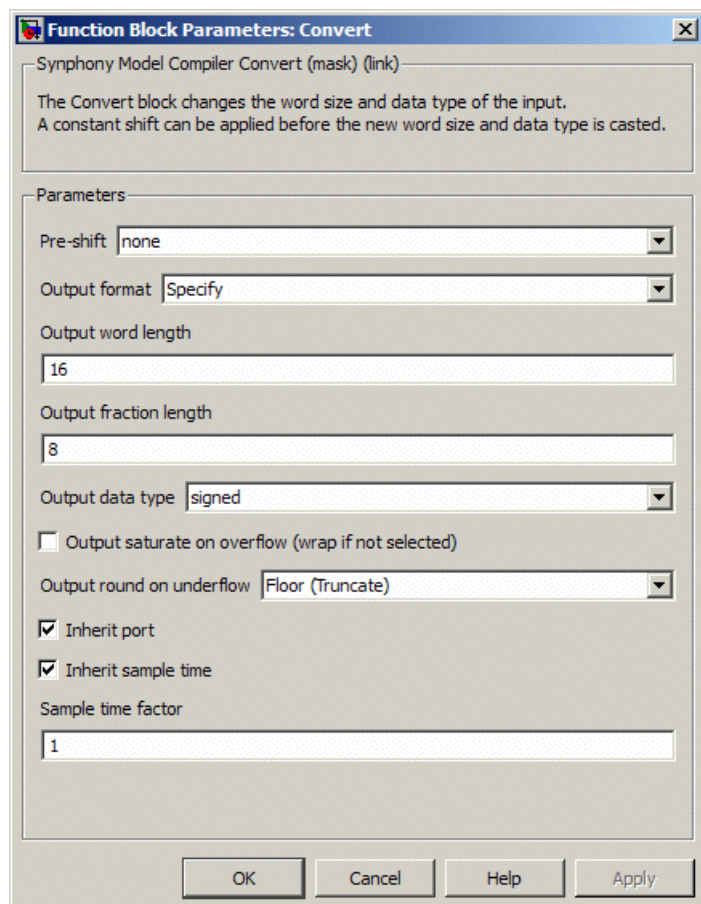
Icon Annotations

Note (green)	Specifies the number and direction of shift bits, if any.
Rounding (red)	Specifies the algorithm used for rounding.

Latency

This block has no latency.

Convert Parameters



The dialog box titled "Function Block Parameters: Convert" contains the following elements:

- A header bar with a close button (X).
- A description: "Synphony Model Compiler Convert (mask) (link)" and "The Convert block changes the word size and data type of the input. A constant shift can be applied before the new word size and data type is casted."
- A "Parameters" section with the following controls:
 - "Pre-shift" dropdown menu set to "none".
 - "Output format" dropdown menu set to "Specify".
 - "Output word length" text input field set to "16".
 - "Output fraction length" text input field set to "8".
 - "Output data type" dropdown menu set to "signed".
 - Checkbox "Output saturate on overflow (wrap if not selected)" is unchecked.
 - "Output round on underflow" dropdown menu set to "Floor (Truncate)".
 - Checkbox "Inherit port" is checked.
 - Checkbox "Inherit sample time" is checked.
 - "Sample time factor" text input field set to "1".
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

Pre-shift

The direction of the optional shift. The value can be one of the following:

- none. This is the default. It keeps the value of the input data intact.
- << does a left shift. Setting this value makes the Number of shift bits parameter available.
- >> does a right shift. Setting this value makes the Number of shift bits parameter available.

Number of shift bits

This parameter indicates the number of bits the input has to be shifted and only becomes available when you set Pre-shift to << or >>. For a right shift, the value of the most significant bit (MSB) is shifted in by the number of bits specified. For a left shift, zero is shifted in on the least significant bit (LSB) side.

You can also specify the number of shift bits in terms of one of these variables: `syn_inp_wl`, `syn_inp_fl`, or `syn_inp_dt`. If Inherit port is enabled, you can also use the `syn_inh_wl`, `syn_inh_fl`, or `syn_inh_dt` variables. See [Special Variables, on page 588](#) for information about them.

Output format, Output word length, and Output fraction length

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584. You can also specify it in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , and <code>syn_inp_dt</code> variables. If Inherit port is enabled, you can also use the <code>syn_inh_wl</code> , <code>syn_inh_fl</code> , and <code>syn_inh_dt</code> variables. The variables are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584. You can also specify it in terms of the variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , and <code>syn_inp_dt</code> . If Inherit port is enabled, you can also use the <code>syn_inh_wl</code> , <code>syn_inh_fl</code> , and <code>syn_inh_dt</code> variables. The variables are described in Special Variables, on page 588 .

Output Data Type

Determines the data type for the output.

Signed	See Output Data Type, on page 584 for details.
Unsigned	See Output Data Type, on page 584 for details.
Preserve	Preserves the input data type. If the input is signed, the output is also signed. If the input is unsigned, the output is also unsigned.
Inherit	Inherits the input data type from the inherit port. This option is only available when you enable Inherit port. See Inherit port, on page 102 for information about this port.

Output saturate on overflow, Output round on underflow

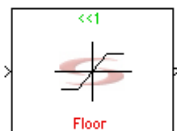
Determine how output overflow and underflow are treated. These options are available when you set Output Format to Automatic or Specify.

Output saturate on overflow	Saturates the overflow when the option is enabled and wraps the overflow when it is disabled. See Overflow Saturation Options, on page 585 for details.
-----------------------------	---

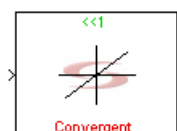
Output round on underflow	See Underflow Rounding Options, on page 585 for details about the rounding options available.
---------------------------	---

The symbol on the block icon reflects the saturation and rounding choices you make. For example:

Saturation on,
Floor rounding



Saturation off,
Convergent rounding



Inherit port

Determines whether the tool creates an inherit port. The tool creates an inherit port when you enable the option. If you enable the option, the tool applies automatic scalar expansion to the inherit and data ports. If one input is scalar and the other is vector, the scalar input is expanded to the size of the vector input.

This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Special Variables, on page 588](#) for information about these variables.
- Use the `inherit` option to specify the Output data type. See [Output Data Type, on page 101](#) for a description of the option.

Inherit sample time

Determines whether the output inherits the sample time from the inherit port. Enabling this option means that the output port inherits the sample time from the inherit port, and disabling it means the output

port inherits sample time from the input. This option becomes available when you enable Inherit port.

Sample time factor

Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

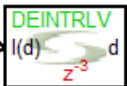
SMC Convolutional Deinterleaver

Reshuffles streaming input symbols according a to a predefined mapping scheme.

Library

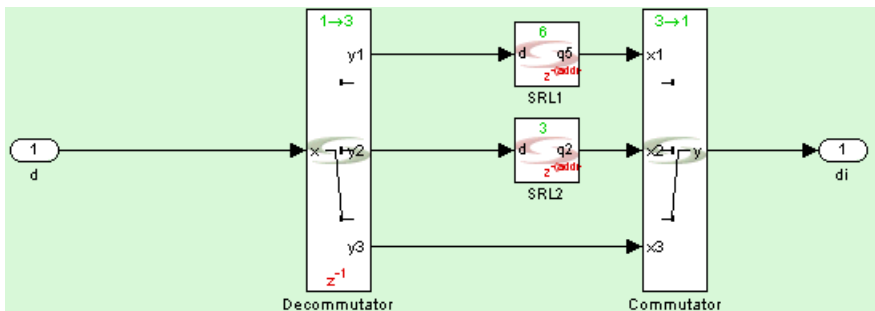
Symphony Model Compiler [Communications](#)

Description



This block reshuffles a fixed number of input symbols according to the mapping you define. This is a custom block; for information about custom blocks, see [Primitives and Custom Blocks, on page 800](#).

The following figure shows the internal modeling of this block:

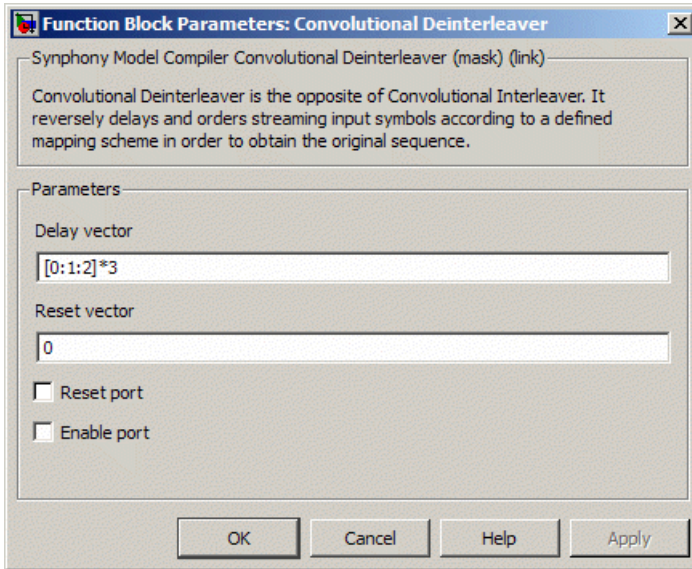


Icon Annotations

Note Specifies that the block is a deinterleaver.

Latency Depends on the number of inputs.

Convolutional Deinterleaver Parameters



Delay vector

Specifies the mapping scheme for the input symbols. It operates on streaming symbols and uses the order specified here, starting at the vector specified in Reset Vector.

Reset vector

Specifies the vector to be used for initialization.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable port. The enable port is connected to the enable signal of the internal shift registers.

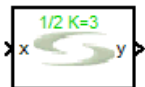
SMC Convolutional Encoder

Performs feed-forward convolutional encoding using k/n convolutional codes, with optional reset and enable ports.

Library

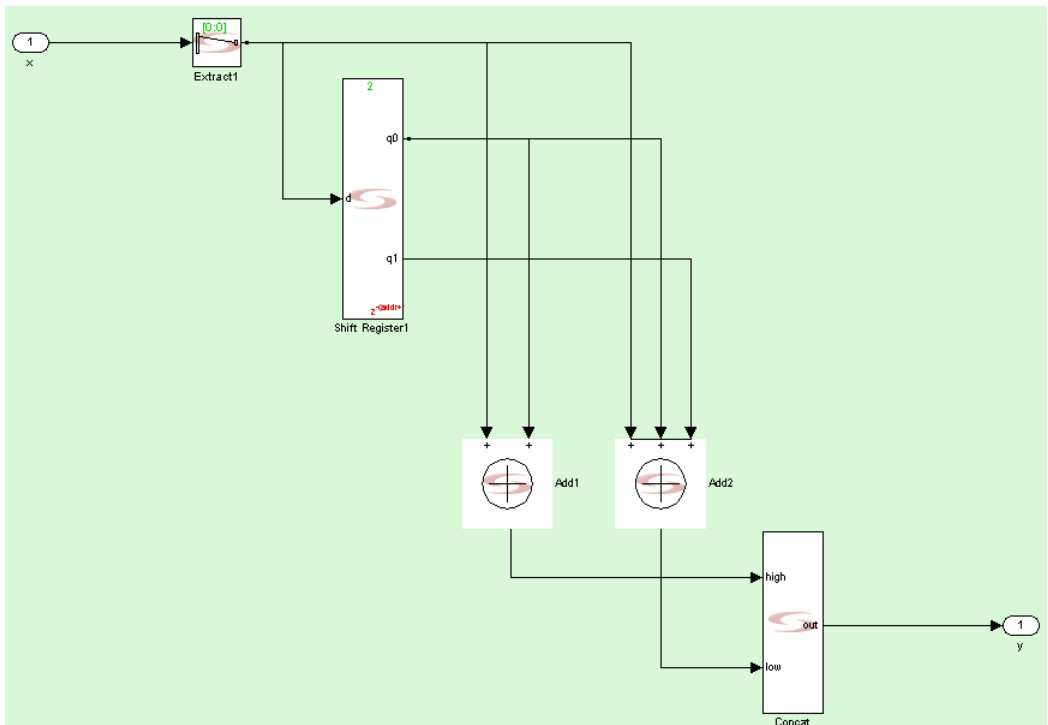
Synphony Model Compiler [Communications](#)

Description



The Synphony Model Compiler Convolutional Encoder is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) that encodes the input data stream with k/n convolutional codes, where k is the number of input bits and n is the number of output bits. It includes optional reset and enable ports.

The following shows how this custom block is implemented:

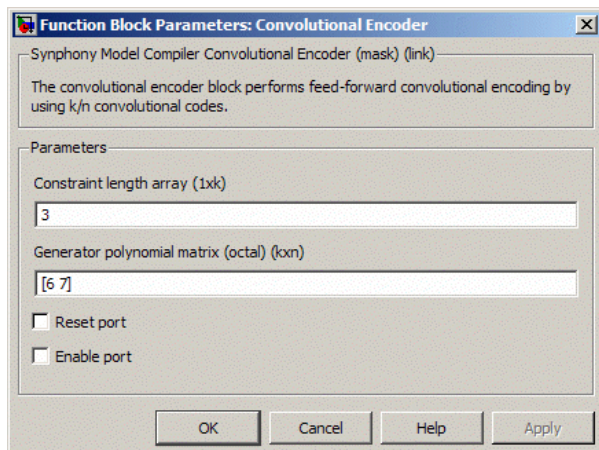


Icon Annotation

The icon for this block displays the following information:

Note	Is Code Rate - Constraint Lengths (e.g. 1/2 code rate with K=3 constraint length)
Latency	This block has no latency.

Convolutional Encoder Parameters



Constraint length array

Determines the 1xk vector which holds the constraint length values for each input.

Generator polynomial matrix

Sets the kxn matrix that specifies the input contributions for each output. The values of the generator polynomial should be specified in the octal number system.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift registers.

SMC Convolutional Interleaver

Shuffles streaming input symbols to a new permutation, using a predefined mapping scheme.

Library

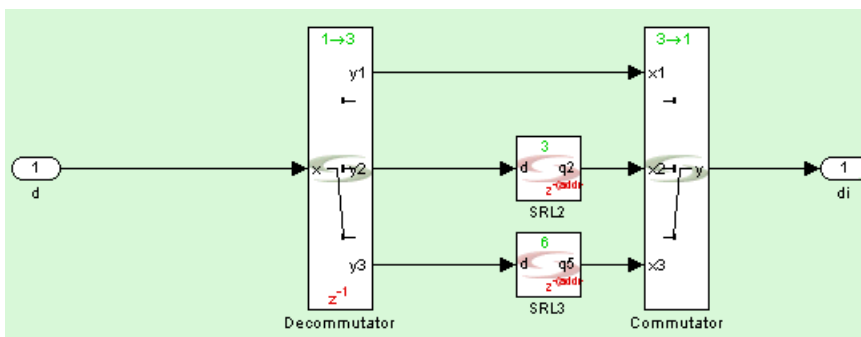
Synphony Model Compiler [Communications](#)

Description



This block shuffles streaming input symbols according to the mapping you define. This is a custom block; for information about custom blocks, see [Primitives and Custom Blocks, on page 800](#).

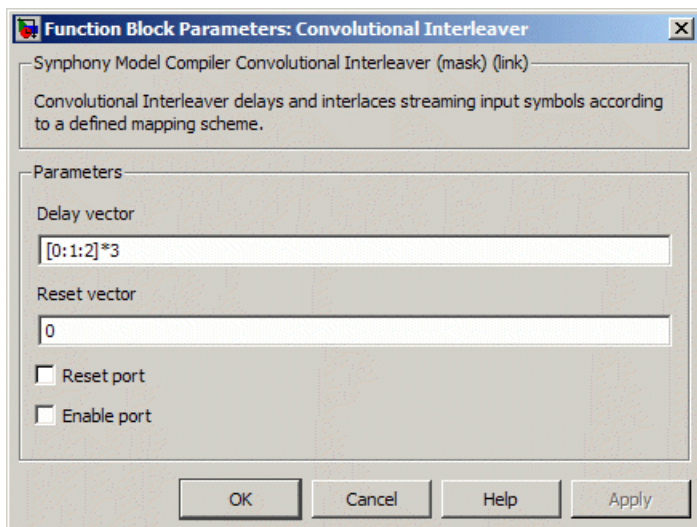
The following figure shows the internals of this block:



Icon Annotations

Note	Specifies that the block is an interleaver.
Latency	Varies with the number of input symbols.

Convolutional Interleaver Parameters



Delay vector

Specifies the order for shuffling the input symbols. It operates on streaming symbols and uses the order specified here, starting at the vector specified in Reset Vector.

Reset vector

Specifies the vector to be used for initialization.

Reset port

When enabled, the block is implemented with a reset port. The reset port is connected to the reset signal of the internal shift registers.

Enable port

When enabled, the block is implemented with an enable port. The enable port is connected to the enable signal of the internal shift registers.

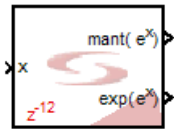
SMC CORDIC Exp

Calculates the natural exponent of the input using a CORDIC algorithm.

Library

Synphony Model Compiler [CORDIC](#)

Description



The Synphony Model Compiler CORDIC Exp block uses a CORDIC algorithm to calculate the natural exponent of the input. See [CORDIC Algorithms, on page 701](#) for a description of the algorithms.

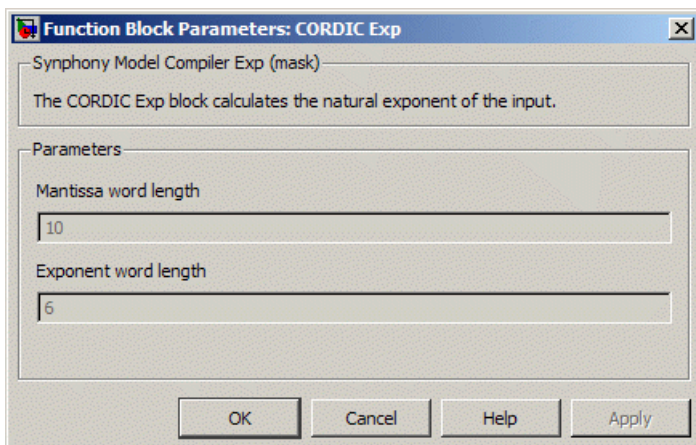
The result from this block is output in the form of a mantissa and an exponent, where $x = \text{mant} \cdot 2^{\text{exp}}$. The mantissa is a fraction, with the most significant bit of the mantissa to the left of the binary point. The exponent is an integer. The number of iterations is equal to the word length of the mantissa.

Icon Annotations

The icon for this block displays the following information:

Latency Annotation	Latency is based on accuracy. It is equal to the number of mantissa bits + 2.
--------------------	---

CORDIC Exp Parameters



Mantissa word length

Number of bits requested for the mantissa fraction.

Exponent word length

Number of bits requested for the exponent integer.

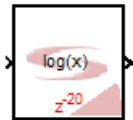
SMC CORDIC Log

Calculates the natural logarithm of the input using a CORDIC algorithm.

Library

Synphony Model Compiler [CORDIC](#)

Description



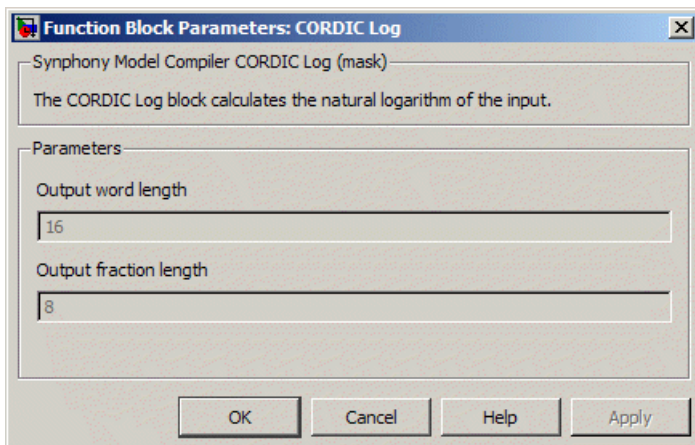
The Synphony Model Compiler CORDIC Log block calculates the natural logarithm of the input, using a CORDIC algorithm. See [CORDIC Algorithms, on page 701](#) for a description of CORDIC algorithms. The number of iterations is equal to output word length.

Icon Annotations

The icon for this block displays the following information:

Latency Annotation	The latency of the block is based on the number of iterations. It is equal to the output word length + 4.
--------------------	---

CORDIC Log Parameters



For descriptions of the parameters, see the following:

Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584

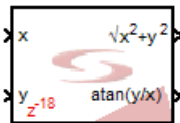
SMC CORDIC Polar

Performs rectangular-to-polar conversion. It calculates $\sqrt{x^2+y^2}$ and $\text{atan}(y/x)$ where x and y are inputs.

Library

Synphony Model Compiler [CORDIC](#)

Description



The Synphony Model Compiler CORDIC Polar block uses the CORDIC algorithm to perform rectangular-to-polar conversions. See [CORDIC Algorithms, on page 701](#) for a description of the algorithms. The CORDIC algorithm is used for computation, and the implementation is fully pipelined.

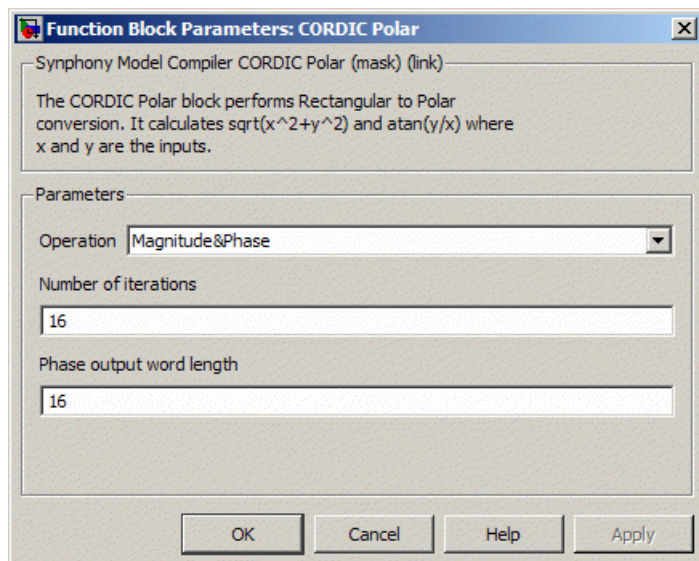
When $y=0$, the value of CORDIC phase goes back and forth between -0.5 and 0.5 for the values of $x<0$. This is because of numerical instability in the CORDIC algorithm, as 0.5 and -0.5 correspond to the same angle $(-\pi, \pi)$. If this causes a problem in the application, use a mux as a workaround. This enables the output to be 0.5 or -0.5 when $x<0$ and $y=0$.

Icon Annotations

The icon for this block displays the following information:

Latency Annotation	The latency of the block is based on the number of iterations. It is equal to the number of iterations + 2.
--------------------	---

CORDIC Polar Parameters



Operation

Determines the kind of rectangular-to-polar operation to be performed.

- Magnitude & Phase calculates $\sqrt{x^2+y^2}$ and $\text{atan}(y/x)$ where x and y are inputs.
- Magnitude calculates $\sqrt{x^2+y^2}$ where x and y are the inputs.
- Phase calculates $\text{atan}(y/x)$ where x and y are scalar inputs.

Number of iterations

This field defines the number of cascaded rotator stages, and affects precision. It is recommended that you set the number of iterations to be equal or close to the input word length. The number of iterations affects the latency of the block, as described in [Icon Annotations, on page 115](#).

Output word length

Determines the total word length for the fixed point data type.

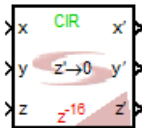
SMC CORDIC Rotator

Implements a fully pipelined CORDIC rotator.

Library

Synphony Model Compiler [CORDIC](#)

Description



The Synphony Model Compiler CORDIC Rotator block implements a fully pipelined CORDIC engine using the CORDIC algorithm in either rotation or vectoring mode. Use this building block to elegantly compute a variety of functions. This block is intended for advanced users, and requires familiarity with CORDIC architecture. See [CORDIC Algorithms, on page 701](#) for a description of CORDIC algorithms.

CORDIC algorithms are designed to rotate vectors in a plane, through a set of shift-add operations. CORDIC functions can be hardware-efficient because they do not need a multiplier, but they require latency to execute the CORDIC iterations.

Circular, Linear, and Hyperbolic Coordinate Systems

The Synphony Model Compiler tool supports circular, linear and hyperbolic coordinate systems. For additional background information about the algorithms, see [CORDIC Algorithms, on page 701](#).

The Synphony blocks do not apply any techniques to modify the range of the inputs, like quadrant folding or pre-shift, because these techniques are specific to the application or function to be implemented with the block. You must use the block with external manipulation to ensure the desired convergence range.

Convergence Range			
Circular		Linear	Hyperbolic
x	$ y/x > 5.74$ for $x < 0$	$ y/x < 1$	$ y/x < .81$
y			
z	$[-1.7433, 1.7433]/2\pi = [-.2775, .2775]$	$[-1, 1]$	$[-1.1182, 1.1182]$

Circular and hyperbolic systems are executed with pseudo-rotations, and the block does not apply gain compensation in any of the stages. This means that the block exposes the typical CORDIC gain associated with CORDIC equations, and you must externally compensate for the gain, if this is required. Linear systems do not have a gain associated with the equations, so there is no need for compensation.

Gain			
Number of iterations	Circular	Linear	Hyperbolic
1	1.4142	1	.8660
2	1.5811	1	.8358
3	1.6298	1	.8319
4	1.6425	1	.8303
5	1.6457	1	.8287
6	1.6465	1	.8283
7	1.6467	1	.8282
8	1.6467	1	.8282

To ensure convergence, hyperbolic systems require some of the iterations in the CORDIC algorithm to be repeated. The Symphony Model Compiler Rotator block does this automatically.

$$X'[n] = \text{iterate}(X_i - m \cdot Y_i \cdot d_i \cdot 2^{-i})$$

$$Y'[n] = \text{iterate}(Y_i + X_i \cdot d_i \cdot 2^{-i})$$

$$Z'[n] = \text{iterate}(Z_i - d_i \cdot e_i)$$

$$X_0 = X[n]$$

$$Y_0 = Y[n]$$

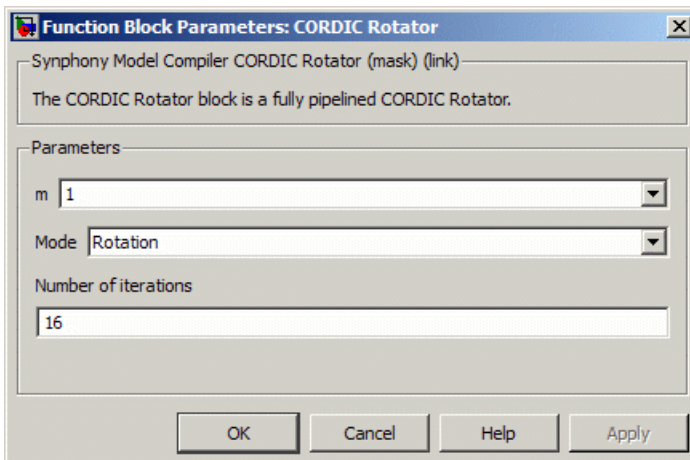
$$Z_0 = Z[n]$$

Icon Annotations

The icon for this block displays the following information:

Note	<p>Reflects the selected coordinate system:</p> <ul style="list-style-type: none"> • CIR Circular. This is the default. The rotation unit is $\tan^{-1}2^{-i}$. • LIN Linear. The rotation unit is 2^{-i}. • HYP Hyperbolic. The rotation unit is $\operatorname{atan}^{-1}2^{-i}$.
Image	<p>Reflects the selected mode:</p> <ul style="list-style-type: none"> • $z' \rightarrow 0$ Rotation mode (iterating to make $z' = 0$) • $y' \rightarrow 0$ Vectoring mode (iterating to make $y' = 0$)
Latency	Latency is equal to the number of iterations selected.

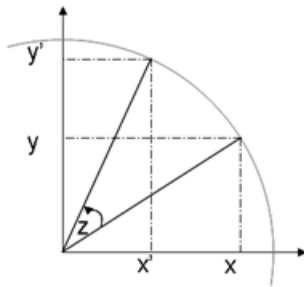
CORDIC Rotator Parameters



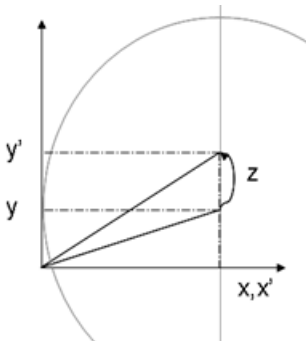
m

Determines the coordinate system used by the block. For further details about coordinate systems, see [Circular, Linear, and Hyperbolic Coordinate Systems, on page 117](#).

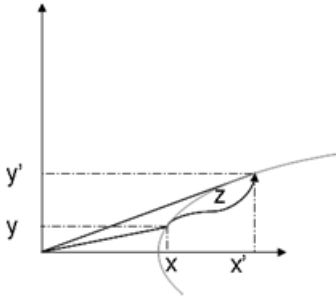
- 1 specifies a circular (trigonometric) coordinate system. The z input represents a circular coordinate (angle) expressed in normalized radians. The fraction $[-1,1]$ corresponds to $[-\pi,\pi]$; however the rotator only converges for inputs in the range of $[-.2775,.2775]$, which corresponds to $[-1.743,1.743]$ radians or $[-99.9,99.9]$ degrees. The vector rotation over a circle will have a CORDIC gain.



- 0 specifies a linear coordinate system. The z input represents a linear coordinate (angle) expressed as a normalized radius. The fraction $[-1,1]$ corresponds to $[y-x:y+x]$ and $|y/x| < 1$ is required for the rotator to converge. The vector rotates on a line through the first coordinate.



- -1 specifies a hyperbolic coordinate system. The z input presents a hyperbolic coordinate that must be in the range $[-1,1]$ for the rotator to converge. The vector rotates on a hyperbole, and will have a CORDIC gain.



Mode

Determines the rotation mode.

- Rotation applies a rotation Z ($Z'=0$) on the given vector coordinates (X,Y) , and calculates the resulting vector coordinates (X',Y') .
- Vectoring rotates the vector (X,Y) to the X-axis ($Y'=0$), and calculates the required angle (Z') to do this.

Number of iterations

Specifies the number of CORDIC rotations to be executed.

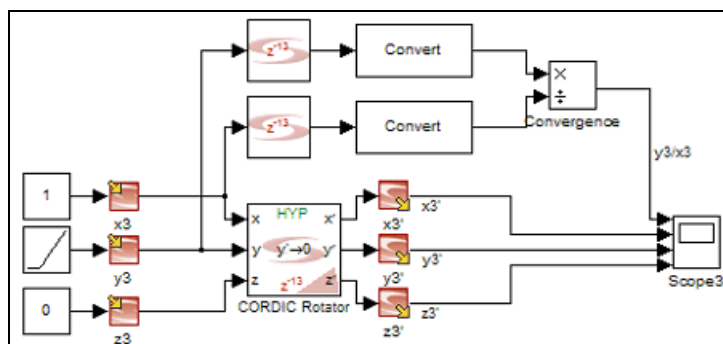
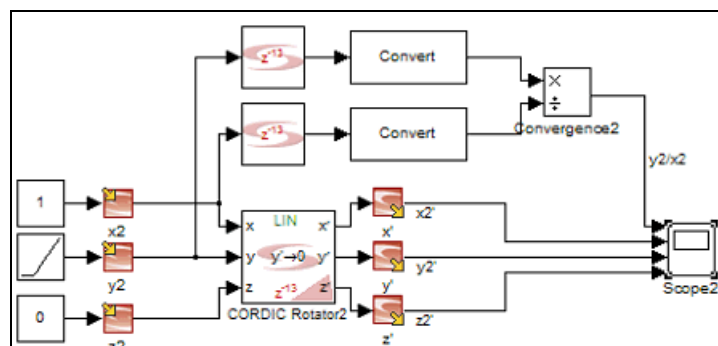
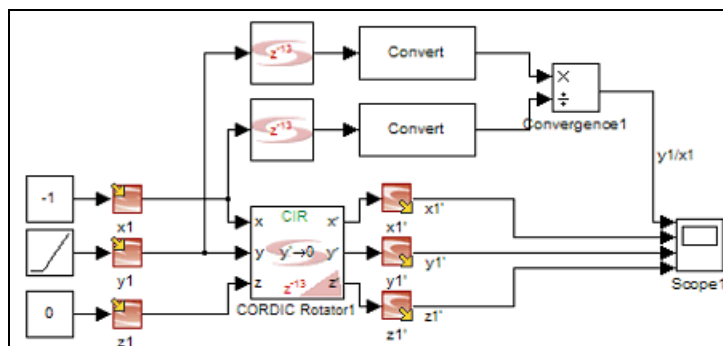
Data Format

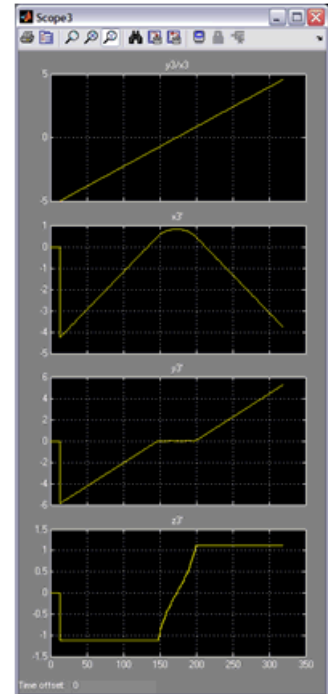
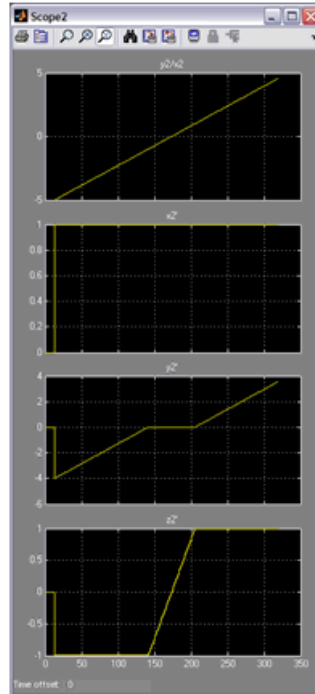
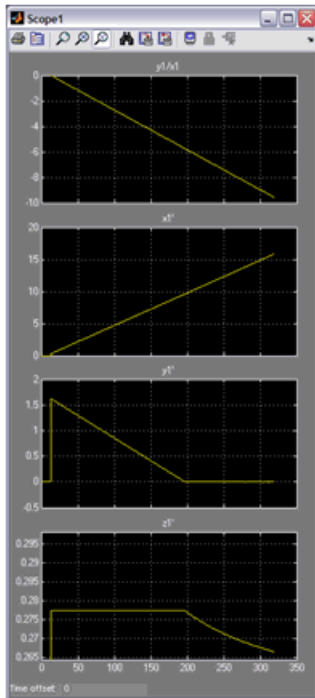
The data formats for rotation and vectoring modes are as follows:

- The data format for x' and y' has a fraction length of $\max(\text{FL}(x), \text{FL}(y))$. The integer portion is also the maximum of both respective inputs. The data type is always signed.
- The data format for z' has the same WL and FL as z , but the data type is always signed.

Examples

The following examples illustrate the convergence check for vectoring mode in the three coordinate systems.





Circular	For $x=-1$ and $0 \leq y \leq 10$	Convergence is ($y'=0$) for $ y/x > 5.74$.
Linear	For $x=1$ and $-5 \leq y \leq 5$	Convergence is ($y'=0$) for $ y/x < 1$.
Hyperbolic	For $x=1$ and $-5 \leq y \leq 5$	Convergence is ($y'=0$) for $ y/x < .81$.

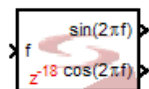
SMC CORDIC SinCos

Implements a sine and/or cosine generator circuit using a fully parallel CORDIC algorithm in rotation mode.

Library

Synphony Model Compiler [CORDIC](#)

Description



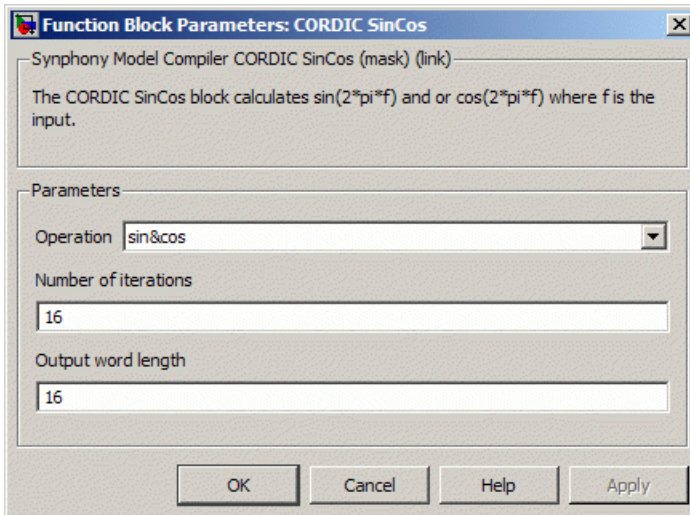
The Synphony Model Compiler CORDIC SinCos block implements a sine and/or cosine generator circuit using a fully parallel CORDIC algorithm in rotation mode. (See [CORDIC Algorithms, on page 701](#) for a description of the algorithms.) It calculates $\sin(2\pi f)$ and/or $\cos(2\pi f)$ where f is an input. The implementation is fully pipelined. The output is signed, with the fraction length being two less than the total word length requested. This allows coverage of the full output range of possible values $([-1 \ 1])$.

Icon Annotations

The icon for this block displays the following information:

Latency Annotation	The latency of the block is equal to the number of CORDIC rotations + 2.
--------------------	--

CORDIC SinCos Parameters



Function

You can select one of the following:

- sin&cos
- sin
- cos

Number of iterations

Defines the number of cascaded rotator stages, and affects precision. It is recommended that you set the number of iterations equal or close to output word length.

Output word length

The output is signed, with fraction bits being two less than the total word length.

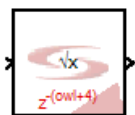
SMC CORDIC Sqrt

Calculates the square root of the input using the CORDIC algorithm.

Library

Synphony Model Compiler [CORDIC](#)

Description



The Synphony Model Compiler CORDIC Sqrt block calculates the square root of the input using a fully pipelined CORDIC algorithm for the implementation. See [CORDIC Algorithms, on page 701](#) for a description of the algorithms.

The output word length is half of the input word length, and the number of output fraction bits is half of the number of input fraction bits. For odd input word length and input fraction bit values, the output word length and number of fraction bits are rounded upwards. For example, if the input word length is 9 and the number of input fraction bits is 3, then the output word length is 5 and the number of output fraction bits is 2. The number of cascaded rotators is the same as the output word length. The output words length affects the latency of the block.

Icon Annotations

The icon for this block displays the following information:

Latency Annotation	The latency of the block is equal to the output word length (OWL) + 4.
--------------------	--

SMC CORDIC2

Implements a circular CORDIC (Coordinate Digital Rotation Computer).

Library

Synphony Model Compiler [CORDIC](#)

Description



The Synphony Model Compiler CORDIC2 is a custom block that creates a circular CORDIC implementation. See [CORDIC Algorithms, on page 701](#) for a description of the CORDIC algorithms. CORDIC algorithms are designed to rotate vectors with a set of shift-add operations in a plane. Because CORDIC functions do not need a multiplier, they can be hardware-efficient, but there is extra latency to execute the CORDIC iterations.

The SMC CORDIC2 output is scaled by the gain inherent with any CORDIC operation. You must scale the x and y (or mag) outputs by approximately 0.602 to compensate for the CORDIC gain. The phase (z) inputs and outputs in CORDIC2 are all modulo 2π and scaled by 2π , where $2\pi \cdot \text{angle}$ represents the actual value of the angle in radians.

The word length of the input coordinates (x, y) and z must be promoted by 1. To correctly promote the word length, use a Convert block by specifying `syn_inp_wl+1` and `syn_inp_fl` for the word length. The data type for the input coordinates must be set to cast to signed, because that is what the CORDIC2 block expects.

CORDIC2 provides the following enhancements compared to SMC CORDIC:

- Configures the number of pipeline stages for the mask from 0 (full combinational implementation) to any positive integer.
- Configures the rounding mode for x, y, and angle outputs computed for each stage from the mask allowing speed vs. accuracy trade off.

- Supports flow control.
- Supports both one and four quadrant operations.
- Supports dynamic vectoring and rotation mode.
- Supports the enabled data path folded structure.
- Provides multichannel support.

CORDIC2 Flow Control

The CORDIC2 block provides the following optional flow control ports:

srdyi	The srdyi (source ready) input port determines whether the input data in the current sample period is valid. An invalid input sample is indicated by srdyi going low. For Enabled data path folded mode, you must keep srdyi low for at least (number of iterations-1) the number of clock cycles between two srdyi HIGH events.
--------------	--

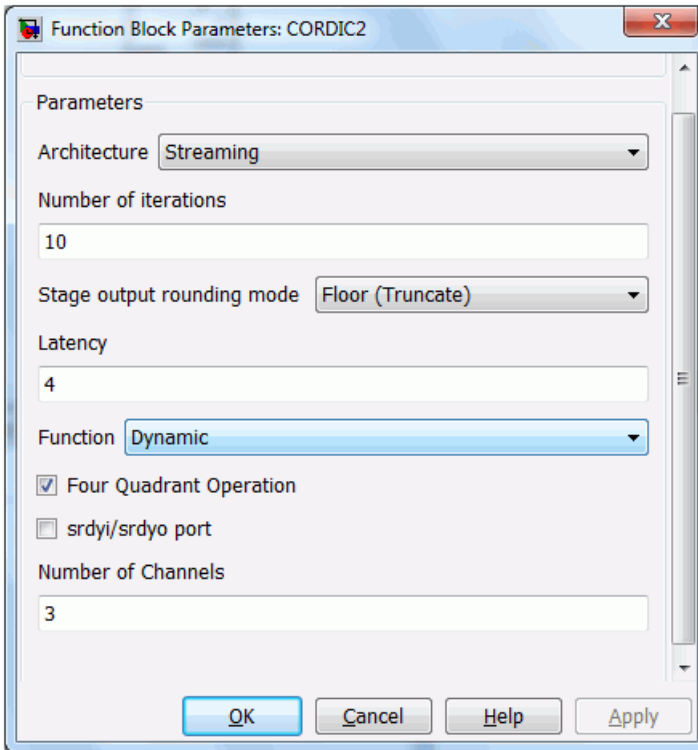
srdyo	The srdyo (source ready) output port determines whether the current output sample is valid. An invalid output sample is indicated by srdyo going low.
--------------	---

For a multichannel operation, the flow control signals are serial and apply for all the channels.

Latency

For Streaming mode, the latency of CORDIC2 is equal to the latency parameter value you entered for the mask. For Enabled data path folded mode, the latency is equal to (number of iterations + 2).

CORDIC2 Parameters



Architecture

Specifies how the CORDIC block is implemented:

- Streaming implements each iteration as a separate stage in the pipeline, that takes input from the previous stage and feeds the output to the next stage in the pipeline. The maximum effective throughput can be achieved at a cost of .
- Enabled data path folded reuses one stage to implement all the iterations, resulting in a folded iteration structure. This mode requires that `srdyi/srdyo` always be available, since the CORDIC only processes one valid input every number of iterations clock cycles. Throughput can be reduced effectively, at the most once every number of iterations clock cycles. However, the entire CORDIC is implemented with only three adders.

Stage output rounding mode

Specifies the rounding mode for x, y and angle (z) at the output of each iteration stage. Select one of the following options: Floor(Truncate), Nearest, Convergent, Fix, Ceil, or Round.

Latency

Specifies the latency for the CORDIC, which is made available only with the Streaming architecture. The pipeline registers that account for the latency are distributed uniformly among the iteration stages to optimize timing performance.

Functions

Specifies the operation to be implemented for the CORDIC:

- Vectoring implements $\text{mag}(x') = \sqrt{x^2 + y^2}$, $y' = 0$, $\text{phase}(z') = \arg(x + jy) + z$.
- Rotation implements $x' = x \cos(z) - y \sin(z)$, $y' = y \cos(z) + x \sin(z)$, $z' = 0$.
- Dynamic performs vectoring or rotation dynamically at runtime.
 - For vectoring, feed a one (1) to the veci port.
 - For rotation, feed a zero (0) to the veci port.

Veco is the delayed version of veci for chaining multiple CORDICs.

Four quadrant operation

Specifies whether to implement the default of one quadrant CORDIC or use wrapper logic to implement a four quadrant rotation for vectoring.

srdyi/srdyo port

Specifies that flow control ports are available when you select the Streaming architecture.

Number of channels

Specifies the number of channels implemented for the CORDIC. All data inputs must be vectors of a width equal to the number of channels.

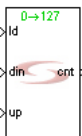
SMC Counter

Implements a configurable counter with enable and reset.

Library

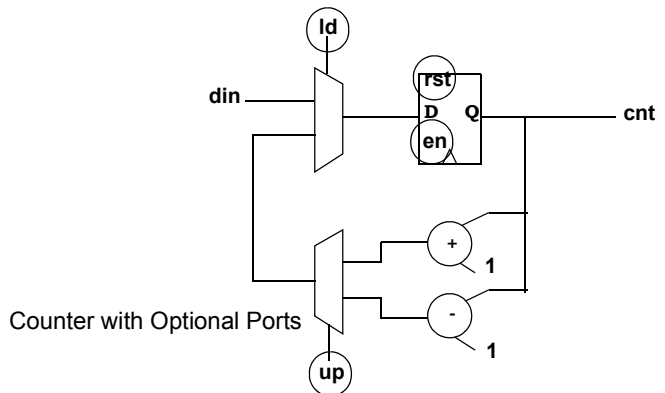
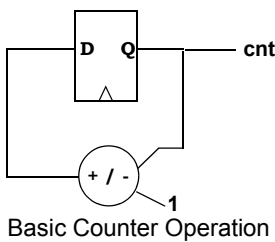
Synphony Model Compiler [Sources](#)

Description



The Synphony Model Compiler Counter block implements a configurable counter with enable and reset, and provides looping control for many algorithms. It has the following:

- Optional ports: load, up, reset, and enable



- Operations: counting up, down, and programmable
- Initial and terminal values for the count
- Cast on the output for sizing

Constant Propagation

This block supports constant propagation ([Constant Propagation, on page 731](#)).

Automatic Scalar Expansion

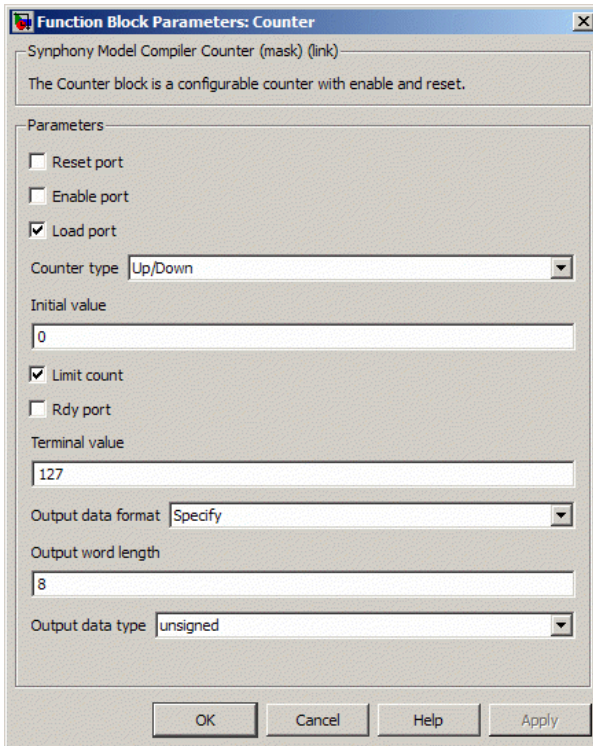
Any of the inputs to the block can be a vector input. If one of the inputs is a vector, the tool expands the others to the size of the vector input.

Icon Annotations

The icon for this block displays the following information:

Top Annotation	This annotation shows the limit at which the counter is reset
Latency Annotation	The latency of the block is only relevant if there is an input: din port with the load operation. The load inherently enforces a latency of 1.

Counter Parameters



The dialog box titled "Function Block Parameters: Counter" contains the following elements:

- Header: "Synphony Model Compiler Counter (mask) (link)" with a close button (X).
- Text: "The Counter block is a configurable counter with enable and reset."
- Section: "Parameters" containing:
 - Checkboxes: "Reset port" (unchecked), "Enable port" (unchecked), "Load port" (checked).
 - Dropdown: "Counter type" set to "Up/Down".
 - Text field: "Initial value" set to "0".
 - Checkboxes: "Limit count" (checked), "Rdy port" (unchecked).
 - Text field: "Terminal value" set to "127".
 - Dropdown: "Output data format" set to "Specify".
 - Text field: "Output word length" set to "8".
 - Dropdown: "Output data type" set to "unsigned".
- Buttons: "OK", "Cancel", "Help", and "Apply" at the bottom.

Reset Port

When enabled, it creates a synchronous reset (**rst**) port, which provides a local block reset. Specify the value of the reset with the Initial Value option, or leave the default of 0. When disabled, the software does not create a **rst** port, and the content of the block is determined solely by the count operation.

Enable Port

When this option is enabled, it creates an enable (**en**) port, which provides control over the Enable status of the block. If this option is disabled, the software does not create an **en** port and the counter block is always enabled.

Load Port

When enabled, the software creates an ld port and a din port. The synchronous ld port loads the block with the value of the input port, din.

When disabled, the software does not create the ld and din ports. The content of the register is determined by either the count or reset operations.

The Load Pin, Reset Pin, and Enable Pin priorities are shown in the following table:

Reset	Enable	Load	Function
0	0	0	Disabled; maintain output
0	0	1	Disabled; maintain output
0	1	0	Enabled; increment/decrement output. Default if you do not enable any optional pins.
0	1	1	Load; din to output
1	0	0	Reset; output initial value
1	0	1	Reset; output initial value
1	1	0	Reset; output initial value
1	1	1	Reset; output initial value

Counter Type

Determines the type of counter.

- Up/Down implements an up/down counter and creates an up port. The direction of the count is determined by the value driven on the up port.

Up Pin Value	Function
0	Count down
1	Count up

- Up hard codes an up counter, and does not create an up port.
- Down hard codes a down counter, and does not create an up port.

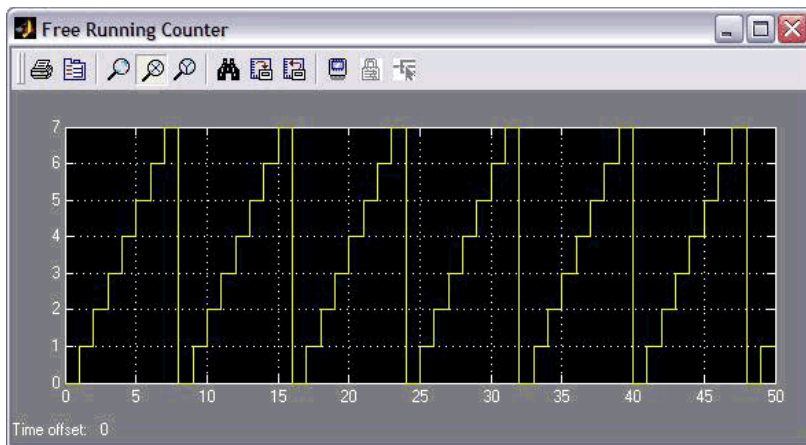
Initial Value

Sets the starting point for the counter block after any reset, including a Terminal Value reset, described below. The default value is 0.

Limit Count

When enabled, it displays the Terminal Value option, which forces a reset when the counter reaches that value.

When disabled, it executes a free-running counter which is a bare configuration (shown in the following figure). You can further manipulate this counter with the Load Pin and Reset Pin options.



Rdy Port

Creates a rdy port when it is enabled. When enabled, the ready is asserted when the output reaches the limit count.

Terminal Value

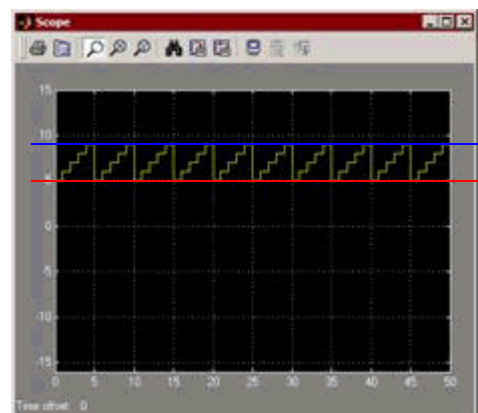
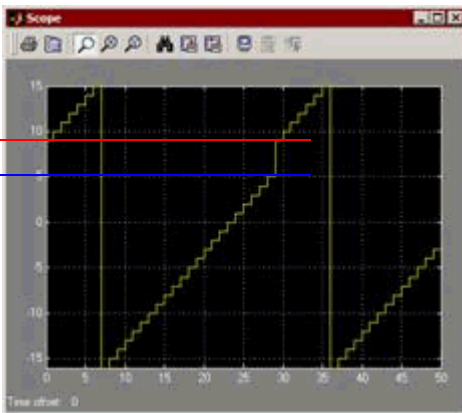
Resets the counter when it reaches the specified value; it resets the count to the Initial Value. The default terminal value is 127. To use this option, you must enable Limit Count.

—— Initial Value —— Terminal Value

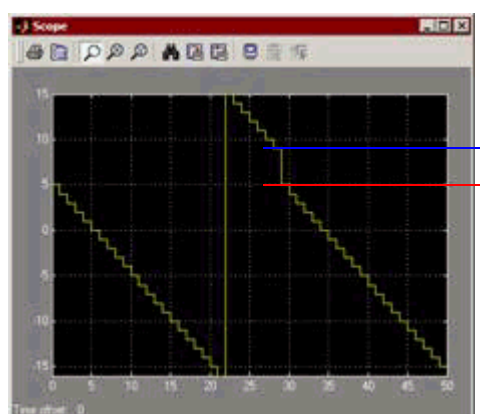
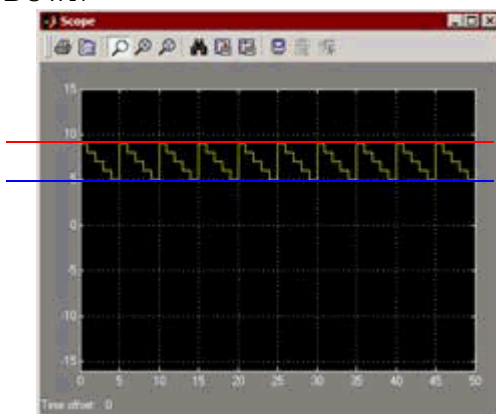
$T < I$

$I < T$

UP



DOWN



Output Data Format

Determines data type and word length. Unlike other Synopsys blocks, the data type is not propagated from an input but derived from other block parameters. You can set this option to Automatic or Specify.

- Automatic

The software derives the data type from a combination of the initial value, terminal value, and the port `din`. The Automatic option calculates the smallest data type that accommodates all the values above. This option is only available when the Load Pin option is enabled and no terminal value is specified. The data type and word length are determined as follows:

Data type	<p>Is signed if either the initial or terminal value is negative, or if the input port is signed.</p> <p>Is unsigned if both the initial and terminal values are positive, or if the input port is unsigned.</p> <p>The data type values are as follows, where N is a finite word length:</p> <p>Unsigned: $(2^N) - 1$</p> <p>Signed: $-2^{(N-1)}$ to $(2^{(N-1)} - 1)$</p>
Word length	Is based on the smallest power that accommodates the initial and terminal values and the port <code>din</code> , and varies with the data type.

- Specify

The Word Length and Data Type options become available.

Output word length

Specifies the output word length.

Output Data Type

Specifies whether the output is signed or unsigned.

- signed specifies Two's complement signed representation, and sets the sign bit to the MSB. This format specifies that an n-bit binary number be interpreted as a value in the range $[-2^{(n-1)}, (2^{(n-1)} - 1)]$. Numbers with their most significant bit equal to 1 indicate a negative value, which is obtained by subtracting 2^n from the unsigned value of the number. For example, if a is a signed 3-bit binary number, $a=110$ means $6 - 2^3 = -2$.
- unsigned specifies that an n-bit binary number be interpreted as a value in the range $[0, (2^n - 1)]$. If a is an unsigned 3-bit binary number, $a=110$ means $1*2^2 + 1*2^1 + 0*2^0 = 6$.

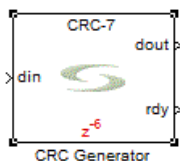
SMC CRC Generator

Generates CRC bits according to the generator polynomial parameter and appends them to the input data frames in the encoder mode and decodes the received data frames in the decoder mode and asserts the error signal if necessary.

Library

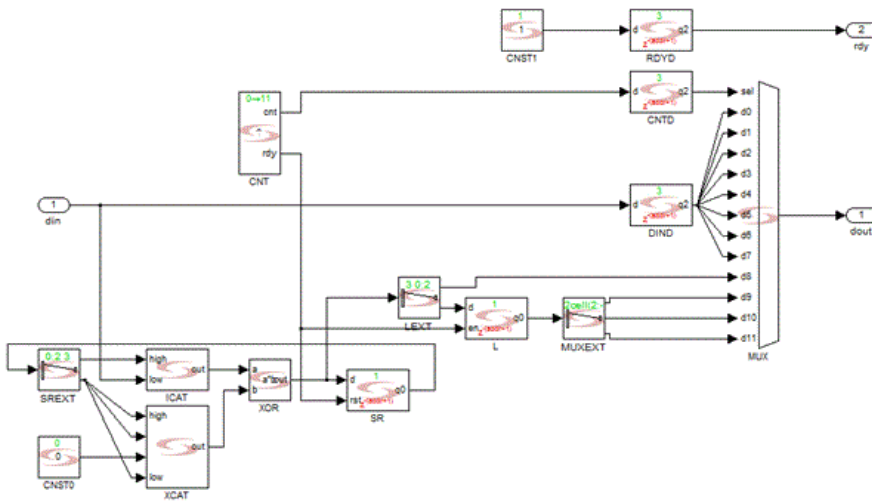
Synphony Model Compiler [Communications](#)

Description



The CRC Generator generates CRC bits according to the generator polynomial parameter and appends them to the input data frames in the encoder mode. It decodes the received data frames in the decoder mode and asserts the error signal if necessary. As the generator polynomial is a primitive polynomial, the constant and the leading terms in the polynomial must be 1.

This block is a custom block (see Primitives and Custom Blocks for a definition). The following figure shows the internal modeling:



The behavior of the block is as follows:

Input data must be `enc_len` bits long, where `enc_len = msg_len + crc_len` and the `crc_len` bits are all 0s.

Output data depends on the selected mode of operation, which are Data and CRC or CRC only. Based on the selected operation mode the output is either the input data with the computed CRC bits appended or just the CRC bits.

The enable signal stalls the entire datapath when deasserted. The ready signal synchronizes downstream data to follow the behavior of the enable signal.

Consecutive frames of data may be fed into the CRC block continuously. The enable signal frames the input data and the ready signal frames the output data.

The sequence of operations in the Encoder Mode are as follows:

1. The input signal is accepted while enable is asserted.
2. The serial data is input, MSB first.
3. After the minimum required latency, the ready signal, which is conditioned by the enable signal, is asserted and the calculated result is output as serial bits, MSB first.

4. The ready signal is deasserted when the reset port is asserted for one or more samples.

The sequence of operations in the Decoder Mode are as follows:

1. The input signal is accepted while enable is asserted.
2. The serial data is input, MSB first.
3. After the minimum required latency, the ready signal is asserted (conditioned by the enable signal) and the calculated result is output as serial bits, MSB first. If there are errors, the error signal is asserted during the period when the CRC is present on the output.
4. The ready and error signals are deasserted when the reset port is asserted for one or more samples.

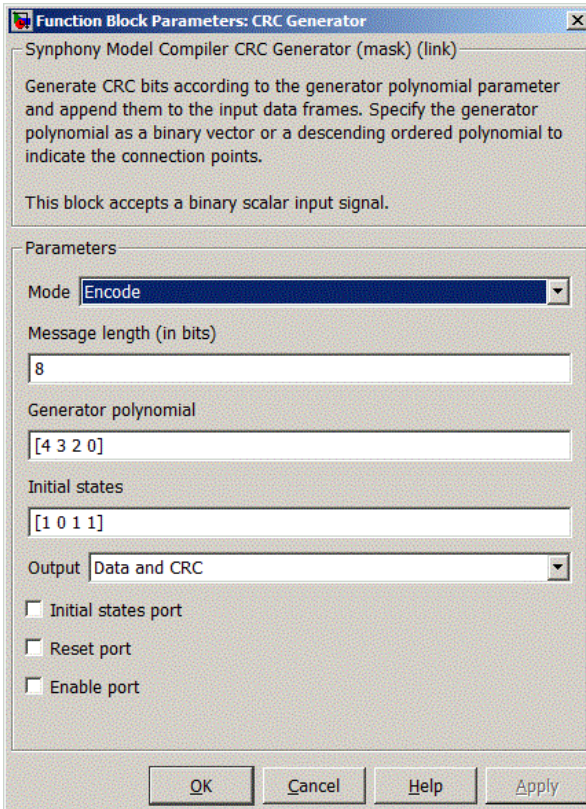
Initialization

When you need to load the initial value, input the initial value as serial data, MSB first, on the initial value port. The block looks for a rising edge on load enable and begins shifting N data values coincident with the load enable and continues for a total of N active states. N is the length of the CRC and the generator polynomial determines it. The active state is one when the enable port is asserted (if there is no enable port, then the block is always active).

Latency

This block of N-1, where N is the length of the shift register.

CRC Generator Parameters



The screenshot shows a dialog box titled "Function Block Parameters: CRC Generator". It contains a description of the block's function and a list of parameters. The parameters include a Mode dropdown set to "Encode", a Message length input set to 8, a Generator polynomial input set to [4 3 2 0], an Initial states input set to [1 0 1 1], and an Output dropdown set to "Data and CRC". There are also three unchecked checkboxes for "Initial states port", "Reset port", and "Enable port". At the bottom are buttons for "OK", "Cancel", "Help", and "Apply".

Function Block Parameters: CRC Generator

Synphony Model Compiler CRC Generator (mask) (link)

Generate CRC bits according to the generator polynomial parameter and append them to the input data frames. Specify the generator polynomial as a binary vector or a descending ordered polynomial to indicate the connection points.

This block accepts a binary scalar input signal.

Parameters

Mode: Encode

Message length (in bits): 8

Generator polynomial: [4 3 2 0]

Initial states: [1 0 1 1]

Output: Data and CRC

☐ Initial states port

☐ Reset port

☐ Enable port

OK Cancel Help Apply

Mode

Specifies whether you want to run the block as a CRC encoder or decoder.

Message Length

Specifies the length of the message in bits.

Generator polynomial

Represent the shift register connections.

You can specify the Generator polynomial parameter using either:

- A binary vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the

length of this vector is one more than the degree of the generator polynomial and the entry is one if there is a connection tap for the corresponding power; otherwise, 0.

- A vector containing the exponents of z for the non-zero terms of the polynomial in descending order of powers. The last entry must be 0.

For example, [1 1 0 0 0 0 0 1] and [7 6 0] represent the same polynomial, $p(z) = z^7 + z^6 + 1$.

Initial States

Specifies the initial values of the registers. It is a binary vector and must satisfy the following criteria:

- The length of the initial states vector must equal the degree of the generator polynomial.
- At least one element of the initial states vector must be 1 in order to generate a non-zero sequence.

Initial states port

When enabled, initial state for of the shift register is provided through the `init_d` port instead of the Initial States mask parameter.

Reset port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

SMC DDS

Creates sin and cos waves based on frequency and phase settings and modulations.

Library

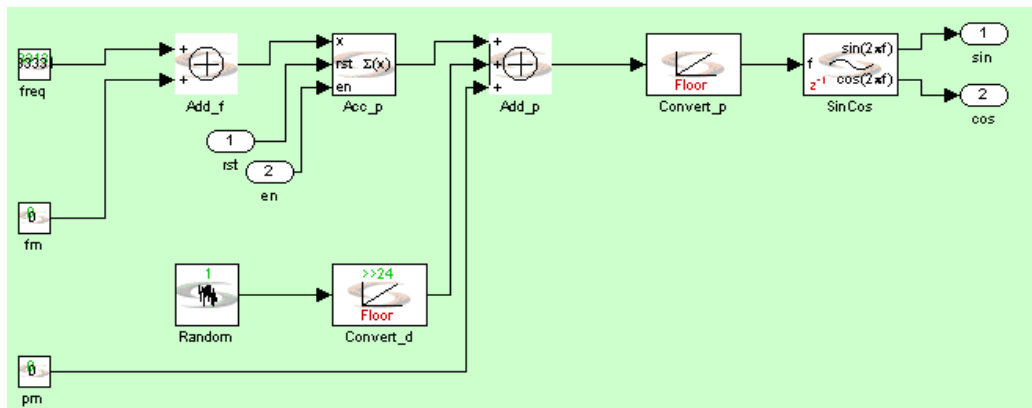
Synphony Model Compiler [Sources](#)

Description



This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) creates a direct digital synthesizer (DDS) by generating discrete-time sin and cos waveforms using a phase accumulator and waveform generator. Phase accumulation accepts frequency and phase inputs and optional frequency and modulation inputs. You can specify accuracy independently for the frequency precision, waveform phase precision, and waveform amplitude precision. You can also choose to flatten spurious noise components caused by phase quantization by selecting phase dithering.

The following figure shows a version of this block with all options enabled. The components vary depending on the options you choose, and the block icon reflects the choices made.



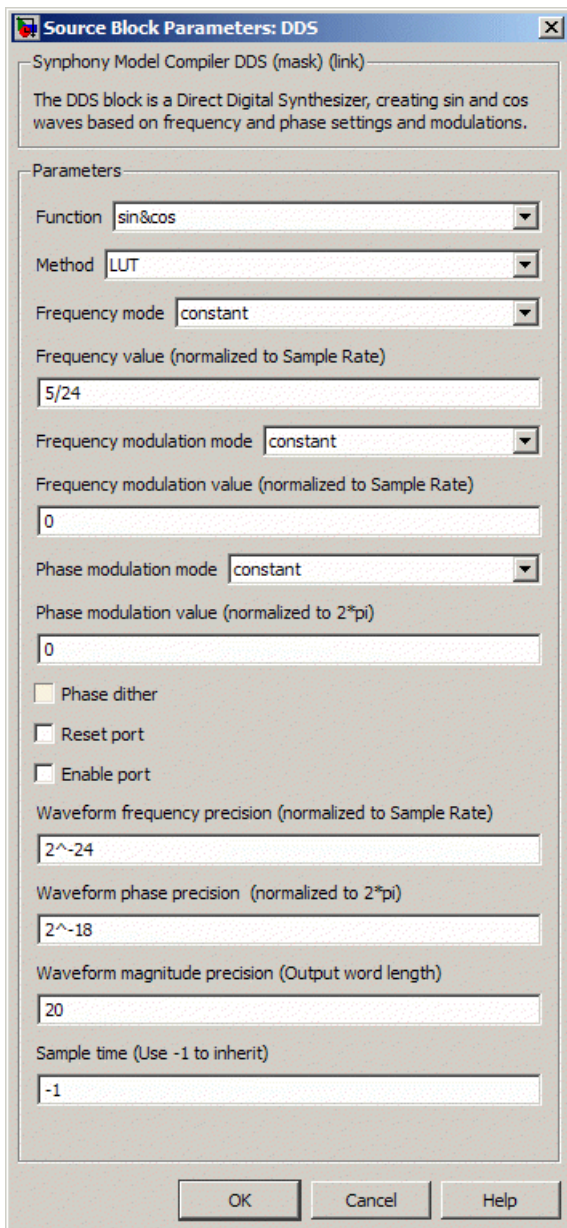
The SMC blockset also includes the DDS2 block, with improved functionality. See [SMC DDS2, on page 149](#) for a description of this block and the differences between the two.

Latency

The latency of the DDS block is determined as follows:

- If there is a LUT, the latency is 1.
- If there is a CORDIC waveform mag, the latency is equal to precision + 2.

DDS Parameters



Source Block Parameters: DDS

Synphony Model Compiler DDS (mask) (link)

The DDS block is a Direct Digital Synthesizer, creating sin and cos waves based on frequency and phase settings and modulations.

Parameters

Function:

Method:

Frequency mode:

Frequency value (normalized to Sample Rate):

Frequency modulation mode:

Frequency modulation value (normalized to Sample Rate):

Phase modulation mode:

Phase modulation value (normalized to 2π):

☐ Phase dither

☐ Reset port

☐ Enable port

Waveform frequency precision (normalized to Sample Rate):

Waveform phase precision (normalized to 2π):

Waveform magnitude precision (Output word length):

Sample time (Use -1 to inherit):

OK Cancel Help

Function

Specifies the function that the DDS block calculates. You can choose one of the following:

- sin
- cos
- sin&cos

Method

Specifies the method used to generate the waveforms:

- LUT uses a lookup table containing the DDS output values to generate the waveforms.
- CORDIC uses CORDIC algorithms to generate the waveforms.

Frequency Mode

Determines how to set the frequency in units of normalized frequency. These values are cast into the format specified in *Waveform Frequency Precision*.

- Constant sets the frequency to the hard-coded value specified in *Frequency Value*.
- Port sets the frequency dynamically to the frequency of the input port.

Frequency Value

Sets a constant value for the frequency in units of normalized frequency.

Frequency Modulation Mode

Specifies how to set the frequency modulation.

- None does not modulate the frequency.
- Constant uses the hard-coded value set in *Frequency Modulation Value* to modulate the frequency.
- Port uses the frequency dynamically set by the input port to modulate the frequency.

Frequency Modulation Value

Sets the value for frequency modulation when *Frequency Modulation Mode* is set to Constant. It specifies an offset in units of normalized frequency

that is added to the frequency and input to the phase accumulator. The value is cast into the data width specified in Waveform Frequency Precision.

Phase Modulation Mode

Specifies how to set phase modulation.

- None does not do any phase modulation.
- Constant uses the hard-coded value set in Phase Modulation Value for phase modulation.
- Port uses the frequency dynamically set by the input port for phase modulation.

Phase Modulation Value

Sets the value for phase modulation when Phase Modulation Mode is set to Constant. It specifies the phase offset constant, in units normalized to 2π . The phase offset is added to the output of the phase accumulator. The value is cast into the data width specified by Waveform Phase Precision.

Phase Dither

Determines whether you improve the DDS spurious free dynamic range by using phase dithering. When you enable this option, the software spreads the spurs through the available bandwidth to prevent phase error from being introduced by the quantizer. The tool adds the dithering sequence before quantization, and then uses the quantized value to index into the sine/cosine lookup table or CORDIC algorithm, so that the phase-space is mapped to time.

When disabled, the tool does not dither the signal.

Reset Port

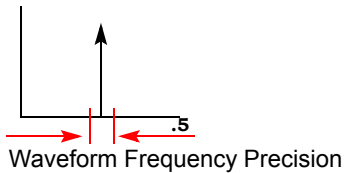
When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

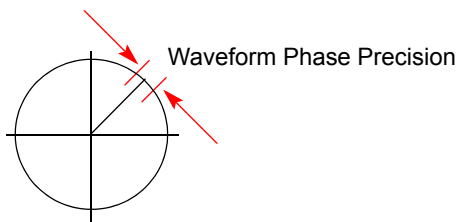
Waveform Frequency Precision

Specifies the frequency resolution of the DDS block, normalized to the sample frequency. For example, a value of $1/1000$ will give you a resolution of 1 Hz at a 1KHz sample rate or 10Hz at 10KHz sample rate.



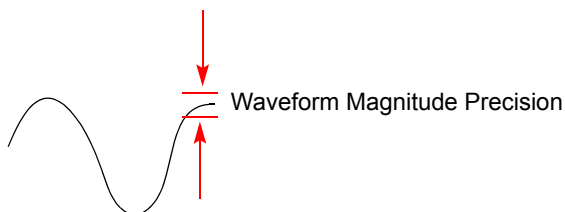
Waveform Phase Precision

Specifies the input phase precision of the waveform generator, normalized to 2π .



Waveform Magnitude Precision

Specifies the bits used for the quantization of the waveform output.



Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you specify reset or enable ports.

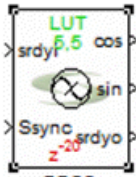
SMC DDS2

Creates sin and cos waveforms based on frequency, phase settings, and modulations, with more functionality than the DDS block.

Library

Synphony Model Compiler [Sources](#)

Description



The DDS2 block creates a direct digital synthesizer (DDS) by generating discrete-time sin and cos waveforms using a phase accumulator and waveform generator. Phase accumulation accepts frequency and phase inputs and optional frequency and modulation inputs. You can specify accuracy independently for the frequency precision, waveform phase precision, and waveform amplitude precision. You can also choose to flatten spurious noise components caused by phase quantization by selecting phase dithering.

Differences Between the DDS2 and DDS Blocks

The DDS2 block provides additional functionality and has better QoR than the DDS block.

- The DDS2 block provides significant area and performance improvements compared with the DDS block, especially for high-performance and multichannel designs. For details, see [DDS2 Multichannel Designs, on page 150](#).
- The DDS2 block offers additional functionality:
 - You can specify the target SFDR; the precision required to achieve the target SFDR is automatically calculated by the DDS2 block.

- This block supports signed frequency and phase input, so that it can be used as frequency modulator.
- It incorporates a LUT compression mode, which uses an additional complex multiplier. This significantly reduces the lookup table size: for a phase word length of 20 bits, LUT size is reduced from 2^{20} to 3×2^{10} (1M to 3K).
- The DDS2 block supports forward flow control through the optional `srdyi`, `ssync`, and `srdo` ports. This eliminates the local reset and local enable ports that are on the DDS block. The new flow control mechanism is more flexible and easier to integrate than the previous DDS methodology. For details, see [DDS2 Flow Control, on page 151](#).

DDS2 Multichannel Designs

To generate a multichannel DDS, you must set the number of channels to be greater than 1, and enable the Fold across channel option. The phase dither generation logic and Sin-Cos generation block (CORDIC-based or LUT-based) are shared across all the channels. When Fold across channel is enabled, the output is multiplexed by the folding factor you specify. For example:

If you specify 8 channels and a folding factor of 4, the output vector size will be 2. The first element of the vector has outputs for channels [1, 2, 3, 4], time-multiplexed in the same order; and the second element has the outputs [5, 6, 7, 8], similarly time-multiplexed. The output sample time is $1/4^{\text{th}}$ of the sample time value provided on the mask parameter or input ports.

For multichannel designs, the dimensions of the `ssync` or `srdyi` ports must be the same as the number of channels. If the same `ssync` or `srdyi` input is provided to all channels, you must connect a Vector Expand block that expands the dimensions of the `ssync` or `srdyi` ports to match the number of channels.

Note the effects of mode settings (Frequency, Frequency Modulation and Phase Modulation) on the implementation of multichannel designs:

Mode	Other Settings	Description
Constant	Constant dimension: 1 Number of channels: >1	The tool automatically performs scalar expansion. This means that the same value will be applied to all the channels.

Constant	Constant dimension: >1, but < number of channels	The other channels are provided with 0 inputs.
Constant	Constant dimension: > number of channels	The tool prunes the size of the constant to equal the number of channels.
Port	Input dimension must be the same as the number of channels	The tool matches inputs to the channels.

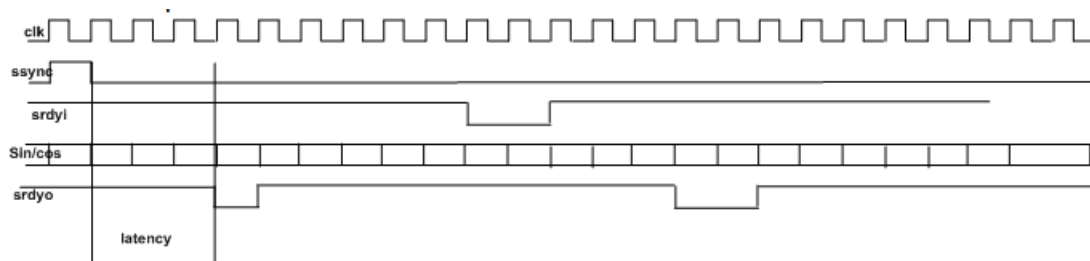
DDS2 Flow Control

DDS2 supports forward flow control through the optional `srdyi` and `ssync` input port and the output `srdyo` port. The following table describes the ports:

<code>srdyi</code>	<p>Informs the DDS2 block about the phase accumulator clock cycle. It does not progress to the next phase value unless <code>srdyi</code> is held high. When <code>srdyi</code> goes low, the <code>srdyo</code> goes low after #DDS2 Latency number of clock cycles, denoting that the corresponding output is invalid. You can use the <code>srdyi</code> mechanism to intermittently stall the DDS2 output and ensure there is no phase discontinuity on restart.</p>
<code>ssync</code>	<p>Re-initializes the phase accumulator. This means that you can start the phase accumulator again from the initial phase modulation value you specified, or 0 if it was not provided. However, <code>ssync</code> does not reset the DDS2 pipeline and the phase values that were in the processing pipeline when you applied the <code>ssync</code> will continue to be processed. When <code>ssync</code> goes high, the <code>srdyo</code> goes low after # DDS2 Latency number of clock cycles, since there is a glitch of one clock cycle for every cycle that <code>ssync</code> is high.</p> <p>When the Latch inputs only on <code>ssync</code> option is enabled, the frequency, frequency modulation, or phase modulation port values will be registered only when <code>ssync</code> is high; otherwise any change in the port is ignored. If the corresponding value is provided as a constant on the mask parameter, <code>ssync</code> has no effect as well.</p>
<code>srdyo</code>	Is the corresponding output port for <code>srdyi</code> and <code>ssync</code> .

For multiple channels, you can provide independent `srdyi` and `ssync` ports for each channel, by specifying a vector equal to the number of channels. When the Fold across channel option is enabled, the `srdyo` is multiplexed like the sin/cos output and is synchronous to the output. The sample time is the same as the `srdyo` output.

The following timing diagram illustrates the flow control operation:



Icon Annotations

The icon for this block displays the following information:

Top Annotation	The green annotation specifies the type of sin-cos generation, the folding factor, and the number of channels.
----------------	--

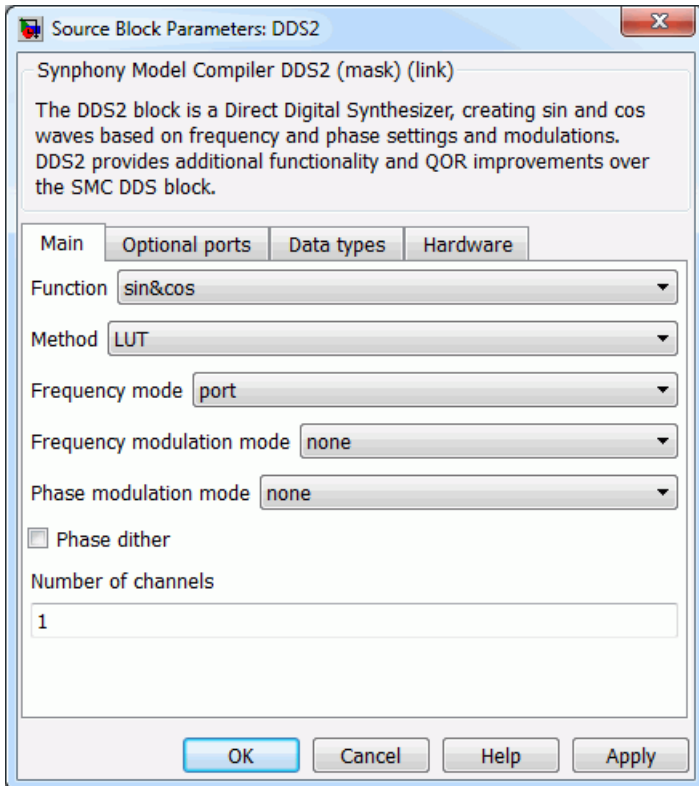
Latency Annotation	The red annotation at the bottom of the block specifies the latency value.
--------------------	--

DDS2 Parameters

The parameters for this block are displayed on four tabs: [Main Tab](#), on page 152, [Optional Ports Tab](#), on page 157, [Data Types Tab](#), on page 158, and [Hardware Tab](#), on page 160.

Main Tab

This tab displays general settings.



Main	Optional ports	Data types	Hardware
Function	sin&cos		
Method	LUT		
Frequency mode	constant		
Frequency value (normalized to Sample Rate)			
1/32			
Frequency modulation mode	constant		
Frequency modulation value (normalized to Sample Rate)			
0			
Phase modulation mode	constant		
Phase modulation value (normalized to 2*pi)			
0			
<input checked="" type="checkbox"/> Phase dither			
Phase dither bits	Specify		
Phase dither word length			
3			
Number of channels			
1			

Function

Specifies the function that the DDS2 block calculates. You can choose one of the following:

- sin
- cos
- sin&cos

Method

Specifies the method used to generate the waveforms:

- LUT uses a lookup table containing the DDS output values to generate the waveforms. When you select this option, LUT-specific options become available on the Hardware tab.
- CORDIC uses CORDIC algorithms to generate the waveforms. When you select this option, CORDIC options become available on the Hardware tab.

Frequency Mode

Determines how to set the frequency in units of normalized frequency. These values are cast into the format specified in Waveform Frequency Precision.

- Constant sets the frequency to the hard-coded value specified in Frequency Value.
- Port sets the frequency dynamically to the frequency of the input port.

For additional information about mode settings in multichannel designs, see [DDS2 Multichannel Designs, on page 150](#).

Frequency Value

Sets a constant value for the frequency in units of normalized frequency. This option requires that Frequency Mode be set to Constant.

Frequency Modulation Mode

Specifies how to set the frequency modulation.

- None does not modulate the frequency.
- Constant uses the hard-coded value, which is set in Frequency Modulation Value to modulate the frequency.
- Port uses the frequency dynamically set by the input port to modulate the frequency.

For additional information about mode settings in multichannel designs, see [DDS2 Multichannel Designs, on page 150](#).

Frequency Modulation Value

Sets the value for frequency modulation when Frequency Modulation Mode is set to Constant. It specifies an offset in units of normalized frequency,

which is added to the frequency and input to the phase accumulator. The value is cast into the data width specified in Waveform Frequency Precision. This option requires that Frequency Modulation Mode be set to Constant.

Phase Modulation Mode

Specifies how to set phase modulation.

- None does not do any phase modulation.
- Constant uses the hard-coded value set in Phase Modulation Value for phase modulation.
- Port uses the frequency dynamically set by the input port for phase modulation.

For additional information about mode settings in multichannel designs, see [DDS2 Multichannel Designs, on page 150](#).

Phase Modulation Value

Sets the value for phase modulation when Phase Modulation Mode is set to Constant. It specifies the phase offset constant, in units normalized to 2π . The phase offset is added to the output of the phase accumulator. The value is cast into the data width specified by Waveform Phase Word Length. This option requires that Phase Modulation Mode be set to Constant.

Phase Dither

Determines whether you improve the DDS spurious free dynamic range by using phase dithering. When you enable this option, the software spreads the spurs through the available bandwidth to prevent phase error from being introduced by the quantizer. The tool adds the dithering sequence before quantization, and then uses the quantized value to index into the sine/cosine lookup table or CORDIC algorithm, so that the phase-space is mapped to time.

When disabled, the tool does not dither the signal.

Phase Dither Bits

When Phase Dither is enabled, you can determine how the precision of the dither output is computed.

- Automatic lets the tool automatically compute the difference between the waveform frequency word length and the waveform phase word length whose value can range between 2 and 19.
- Specify lets you choose the phase dither word length.

Phase Dither Word Length

If Phase Dither Bits is set to Specify, you can set the word length for the dither generator output. When you specify a dither word length that is outside the range [2, 19], the tool automatically sets the word length to 2 and 19 respectively.

For both automatic and specify Phase Dither Bits modes, if the tool needs to limit the value to 2 or 19:

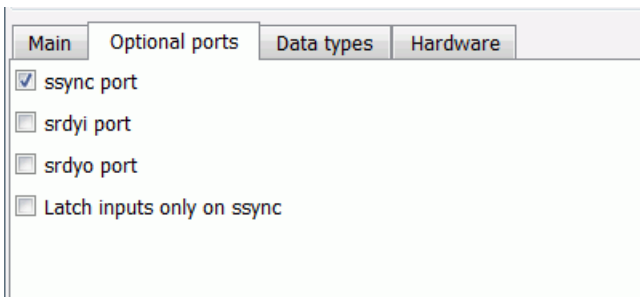
- The waveform phase word length value you specified may be ignored and the value is automatically set to frequency word length -2 or frequency word length -19.
- Unless setting this value leads to a zero or negative value of the phase word length, in which case the value you specified is retained.

Number of Channels

Specifies the number of output channels required.

Optional Ports Tab

The ports on this tab provide flow controls for the block. See [DDS2 Flow Control, on page 151](#) for additional information about these ports.



ssync port

When enabled, the block includes the ssync input port. When enabled, it also makes the corresponding srdyo output port available.

srdyi port

When enabled, the block includes the srdyi input port. When enabled, it also makes the corresponding srdyo output port available.

srdyo port

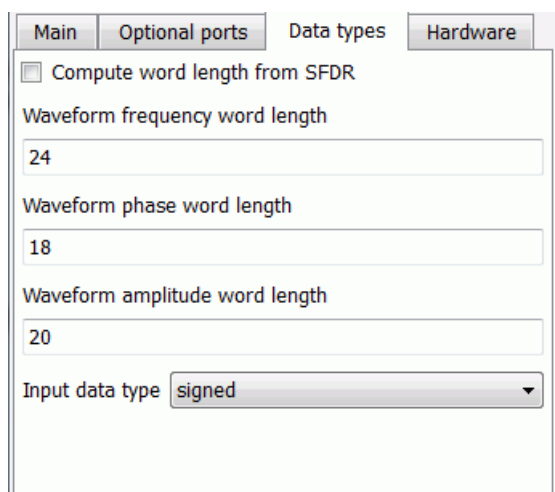
When this option is enabled, the srdyo output port available for the block. If either the ssync or srnyi port is enabled, the srdyo port is always available at the output.

Latch inputs only on ssync

When enabled, the tool accepts and registers the input at the frequency, phase modulation, or frequency modulation port only when ssync is high.

This option becomes available when ssync port is enabled and Frequency mode, Frequency modulation mode, or Phase modulation mode (on the Main tab) is set to port.

Data Types Tab



The screenshot shows the 'Data types' tab of a configuration window. It contains a checkbox labeled 'Compute word length from SFDR' which is currently unchecked. Below this are three text input fields: 'Waveform frequency word length' with the value '24', 'Waveform phase word length' with the value '18', and 'Waveform amplitude word length' with the value '20'. At the bottom, there is a dropdown menu for 'Input data type' which is currently set to 'signed'.

Compute word length from SFDR

Determines whether to let the tool automatically compute word length from the SFDR value or allow you to specify the waveform phase word length. When this option is enabled, only the SFDR parameter is available on the mask. Otherwise when disabled, the Waveform phase word length parameter is available on the mask.

SFDR

Specifies the Spurious Free Dynamic Range (SFDR) that is expected at the output. This option does not guarantee that the specified SFDR value will be met at the output; you must also specify an appropriate frequency word length and output word length value to get the appropriate SFDR at the output. The tool only uses the SFDR value to compute the waveform phase word length with the following formula:

$$\text{Waveform Phase Word Length} = \text{ceil}(\text{SFDR}/6)$$

Waveform frequency word length

Specifies the word length of the frequency, either as a constant or for the port. This value also specifies the resolution of the phase increment in the DDS, so this value can be computed as described below.

Assume that f_s is the sampling rate and f is the frequency needed for the output to be generated. The frequency word length must be greater than or equal to $-\text{ceil}(\log_2(f/f_s))$ to prevent any SFDR degradation. However, the phase increment value may not be full precision; this is possible only when f/f_s is a power of 2.

Waveform phase word length

Specifies the word length of the phase accumulator output. If the desired phase precision for the accumulator in radians is ϕ , then set the value to $-\text{ceil}(\log(\phi/(2\pi)))$. The phase word length exponentially affects the size of the sin-cos lookup table, but does not affect the number of CORDIC stages or CORDIC latency.

Waveform amplitude word length

Specifies the bits used for quantization of the waveform output.

Input data type

Determines whether the input is signed or unsigned, when at least one of the inputs is available through a port.

If all the inputs are constant, the tool automatically determines this. If the input data type is set to signed and some of the input constants are unsigned, the tool automatically promotes the constant as signed. If the input data type is unsigned but one of the input constants is signed, the tool automatically promotes the data type of input ports as signed.

Hardware Tab

This tab displays different options, according to what you selected for the Method option (Main tab):

LUT

The screenshot shows the 'Hardware' tab of the SMC DDS2 configuration window. It is divided into two main sections: 'LUT' and 'CORDIC'. The 'LUT' section on the left has tabs for 'Main', 'Optional ports', 'Data types', and 'Hardware'. The 'CORDIC' section on the right has tabs for 'Main', 'Optional ports', 'Data types', and 'Hardware'. In the 'LUT' 'Main' tab, the 'Fold across channels' checkbox is unchecked, while 'Pipeline sin-cos LUT' and 'Split phase LUT compression' are checked. The 'Target device' is set to 'Virtex 5' and the 'Optimization target' is set to 'Area'. In the 'CORDIC' 'Main' tab, the 'Fold across channels' checkbox is unchecked, 'CORDIC parameters' is set to 'Specify', 'Number of stages' is 25, 'CORDIC Latency' is 22, and 'Stage output rounding' is set to 'Round'.

Fold across channels

When enabled, specifies that channels are time division multiplexed. This option requires that the number of channels be greater than 1, and makes the Folding factor and sample time options available.

The folding options apply to both the LUT and CORDIC methods for the DDS2 block.

Folding Factor

Specifies the time division multiplexing factor across channels. This option requires that Fold across channels be enabled.

The number of channels must be an integer multiple of the folding factor. Otherwise, the tool performs zero padding at the input to increase the number of channels to be a multiple of the folding factor. The zeros are not removed at the output.

Sample Time

Specifies sample time. Use -1 to inherit. This option is unavailable if you specify any option that results in an input port being available.

Pipeline sin-cos LUT

(LUT Method)

When this option is enabled, the tool internally uses a fully pipelined optimized quarter-wave LUT architecture that significantly increases the maximum achievable clock frequency. Turn on this option for optimal FPGA mapping, unless the DDS2 block is used in a feedback path such as a Costas loop.

This option automatically becomes available when Fold across channels option is enabled and LUT is selected as the Method on the Main tab.

Split phase LUT compression

(LUT Method)

When enabled, the tool reduces the LUT size from 2^{phaseWL} to $3 \cdot 2^{(\text{phaseWL}/2)}$, at the cost of an extra complex multiplier. Use this option to lower block RAM utilization when you have large phase word lengths.

This option is available when LUT is selected as the Method on the Main tab.

If Pipeline sin-cos LUT and Split phase LUT compression are both enabled, additional target options become available.

Target device

(LUT Method)

Selects an FPGA target for which the generated RTL is optimized.

This option is available when LUT is selected as the Method on the Main tab, and both Pipeline sin-cos LUT and Split phase LUT compression are enabled.

Optimization Target

(LUT Method)

Specifies whether the generated RTL is optimized for Speed or for Area.

This option is available when LUT is selected as the Method on the Main tab, and both Pipeline sin-cos LUT and Split phase LUT compression are enabled.

CORDIC parameters

(CORDIC Method)

Specifies hardware options when you set CORDIC as the Method on the Main tab.

- Automatic
The tool automatically selects the optimal options depending on the other parameters.
- Specify
Additional options become available where you can set individual options.

Number of Stages

(CORDIC Method)

Specifies the number of stages in the CORDIC DDS2 block. This option is available when CORDIC is selected as the Method on the Main tab, and CORDIC parameters is set to Specify.

CORDIC Latency

(CORDIC Method)

Sets the latency of the CORDIC DDS2 block. This option is available when CORDIC is selected as the Method on the Main tab, and CORDIC parameters is set to Specify.

Stage output Rounding

(CORDIC Method)

Sets the rounding mode to use on the x, y, and angle output of each stage of the CORDIC DDS2 block. This option is available when CORDIC is selected as the Method on the Main tab, and CORDIC parameters is set to Specify.

You can set this option to Floor(Truncate), Nearest, Convergent, Fix, Ceil, or Round. See [Underflow Rounding Options, on page 585](#) for descriptions of the algorithms.

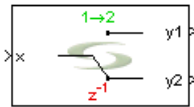
SMC Decommutator

Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly (time division demultiplexing). The Decommutator block provides optional flow control, multichannel, and single-clock multi-rate support.

Library

Synphony Model Compiler [Signal Operations](#)

Description



This block is a time division demultiplexer that sequentially switches the data at the input port to multiple output ports. Each output port data rate is reduced by a factor of the number of output ports. Data at the output ports have the same format as the input.

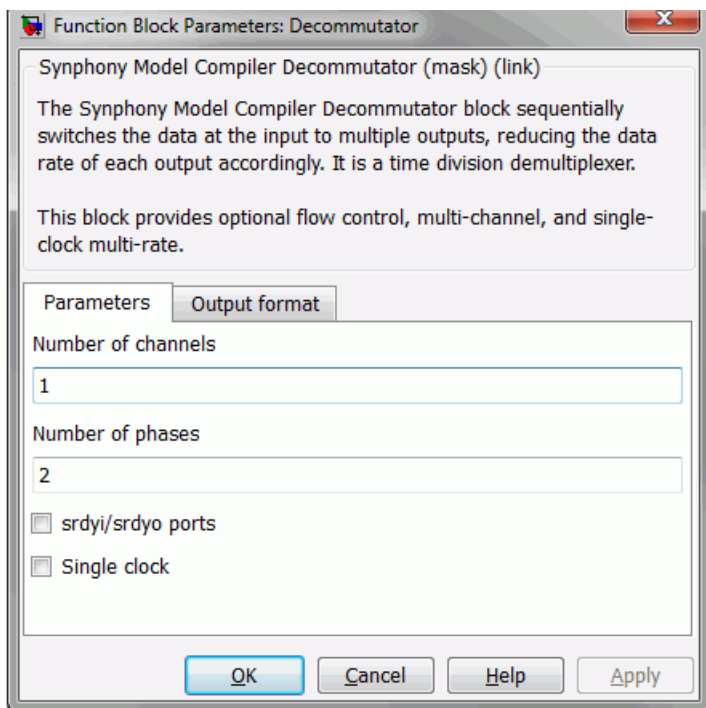
This block is a custom block (see [Primitives and Custom Blocks](#), on page 800 for a definition).

Icon Annotation

The icon for this block displays the following information:

Top Annotation	The annotation at the top of the block indicates the number of output ports demultiplexed from a single input port.
Latency Annotation	One sample latency with respect to the output sample domain for multi-clock mode. Zero latency for single-clock mode.

Decommutator Parameters



Number of channels

Specifies the number of channels processed. The format of the output data depends on the Output format parameters described in the sections: [Scalar Output Format, on page 165](#), [Vector Output Format, on page 166](#), and [Matrix Output Format, on page 168](#).

Number of phases

Specifies the number of outputs or phases (per channel) from which data is demultiplexed to the output. The format of the output data depends on the Output format parameters described in the sections: [Scalar Output Format, on page 165](#), [Vector Output Format, on page 166](#), and [Matrix Output Format, on page 168](#).

srdyi/srdyo ports

When enabled, the block provides forward flow control. srdyi (Source Ready Input) indicates that the current input data is valid. srdyo (Source

Ready Output) is used to chain the Decommutator block to other flow control blocks.

Single clock

When enabled, the block does not introduce a new sample time on the output. It creates a single-clock multi-rate implementation instead.

For Single clock mode, the outputs are provided in the fast domain.

Scalar Output Format

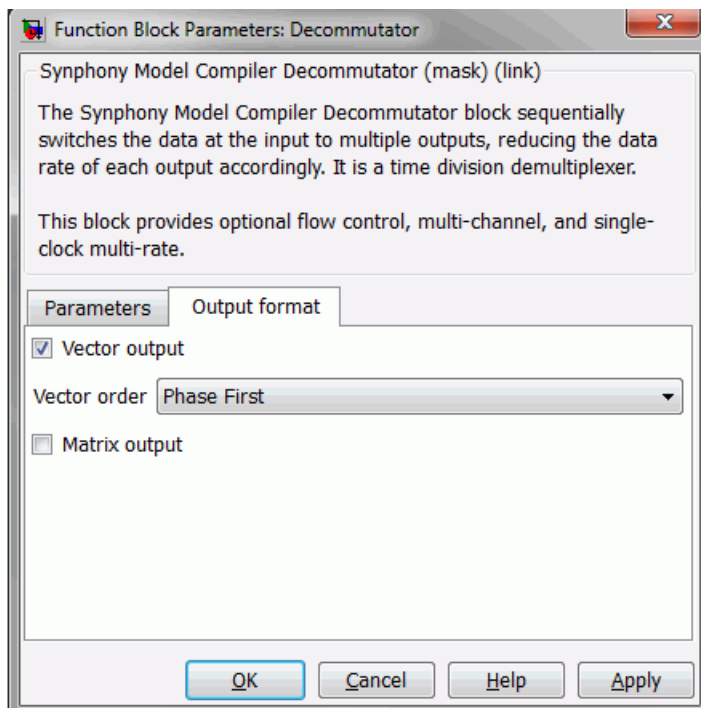
The Decommutator block can provide data to the output depending on how many channels are required and the state of the Vector Output and Matrix Output parameters. See [Vector Output Format, on page 166](#) and [Matrix Output Format, on page 168](#). Scalar output is only available when the Vector output option is disabled.

Each output is a separate port to the block. $y_1 \dots y_N$ are the outputs to the first channel, $y_{1+N} \dots y_{2N}$ are the outputs to the second channel, and $y_{1+(C-1)*N} \dots y_{C*N}$ are the outputs to the last channel.

Example of a 3-phase, 2-channel scalar output:

Port	Input	Channel
y1	1	1
y2	2	1
y3	3	1
y4	1	2
y5	2	2
y6	3	2

Vector Output Format



Vector output

When enabled, the block provides a single vector output for all data. The Vector order parameter determines the format of the output data.

Vector order

Vector output must be enabled and Matrix output disabled for this option to be available. A single port and the data is provided as a vector of length $N \times C$, where N is the number of phases and C is the number of channels. The Vector order parameter determines the order of elements in the vector.

Vector order	Description
Phase First	1 st phase $e_1 \dots e_C$ 2 nd phase $e_{1+C} \dots e_{2C}$ N^{th} phase $e_{1+(N-1)*C} \dots e_{N*C}$
Channel First	1 st channel $e_1 \dots e_N$ 2 nd channel $e_{1+N} \dots e_{2N}$ C^{th} channel $e_{1+(C-1)*N} \dots e_{C*N}$

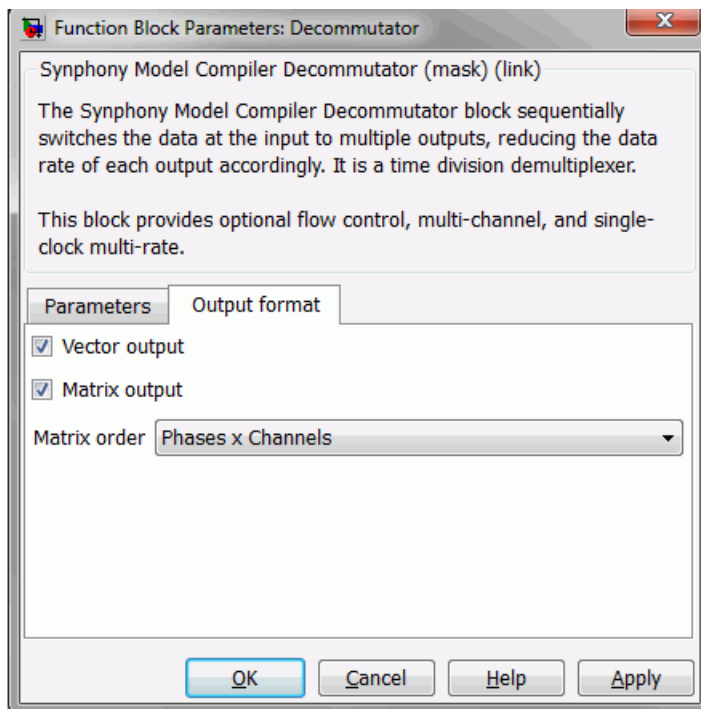
Example of a 2-phase, 3-channel, phase first vector output:

Element	Phase	Channel
1	1	1
2	1	2
3	1	2
4	2	1
5	2	2
6	2	3

Example of a 2-phase, 3-channel, channel first vector output:

Element	Phase	Channel
1	1	1
2	2	1
3	1	2
4	2	2
5	1	3
6	2	3

Matrix Output Format



Matrix output

When enabled, the block provides a single matrix output for all data. The Matrix order parameter determines the format of the output data. Vector output must be enabled for this option to be available.

Matrix order

Matrix output must be enabled for this option to be available. A single output port and the data is provided as a matrix. The Matrix order parameter determines the dimension of the matrix.

Matrix order	Description
Phases x Channels	NxC matrix, where N is the number of phases and C is the number of channels
Channels x Phases	CxN matrix, where C is the number of channels and N is the number of phases

SMC Delay

Delays the input by the specified number of sample clock cycles.

Library

Synphony Model Compiler [DSP Basics](#), Synphony Model Compiler [Memories](#)

Description



The Synphony Model Compiler Delay block delays the input by a specified number of sample clock cycles. Initial values in the delay line are zero. You can also use the Register block ([SMC Register, on page 441](#)) to specify a delay, but it is recommended that you use the Delay block wherever possible.

shls_retiming_lock Constraint

You can specify whether retiming affects Delay blocks by setting the `shls_retiming_lock` constraint. See [shls_retiming_lock Constraint, on page 628](#) for details.

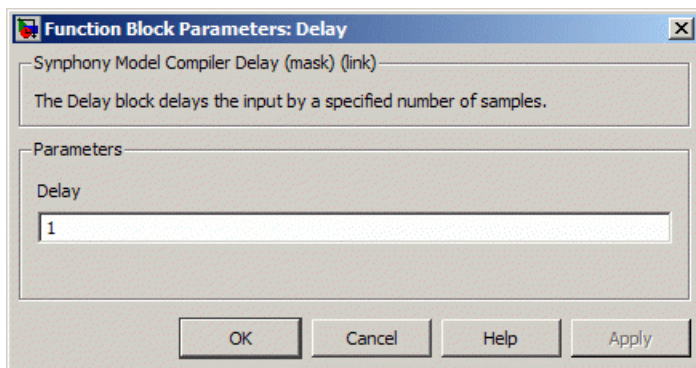
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

There is no extra latency for this block; its latency is determined by its functionality.

Delay Parameters



Delay

Sets the number of sample clock cycles by which the signal is delayed.

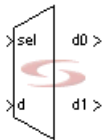
SMC Demux

Implements a de-multiplexer of up to 2048 outputs.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Demux block is a de-multiplexer of up to 2048 outputs. The sel input determines which of the outputs gets the value of the d data input. This value becomes available on the output, and is retained until overwritten.

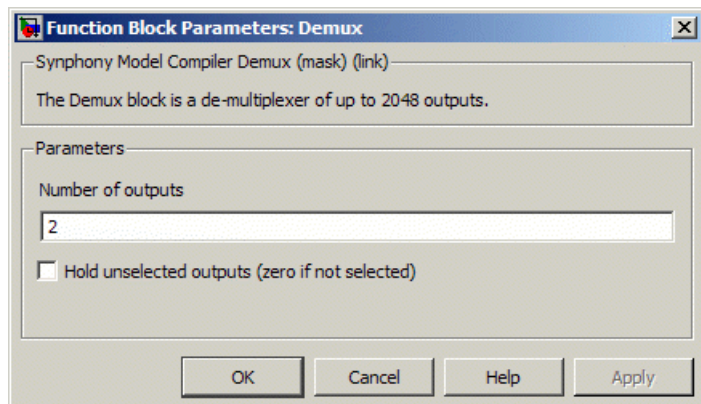
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Demux Parameters



Number of outputs

Determines the number of outputs required. You can specify up to 2048 outputs.

Hold unselected outputs

When enabled, the non-selected output holds the previous value on the output.

If the option is disabled, non-selected outputs drive a zero.

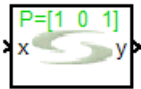
SMC Depuncture

Removes specified bits from the input data stream and replaces them with zeroes.

Library

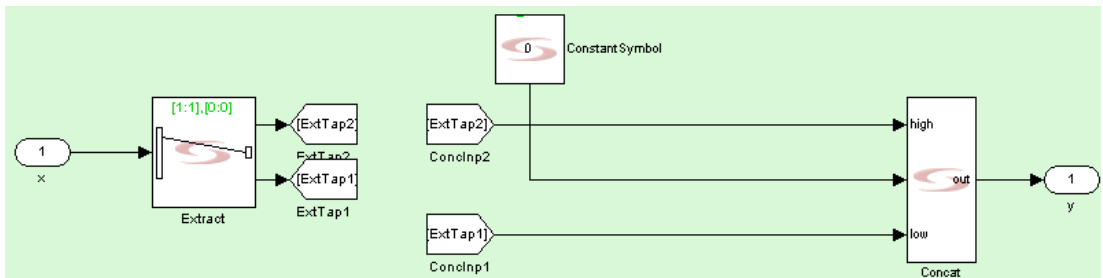
Synphony Model Compiler [Communications](#)

Description



Using the puncture matrix you specify, the Depuncture block inserts zeroes in the locations you specify. The output rate is the same as input sample rate.

This block is a custom block. (See [Primitives and Custom Blocks](#), on page 800 for a definition.) The following figure shows how the block is modeled:



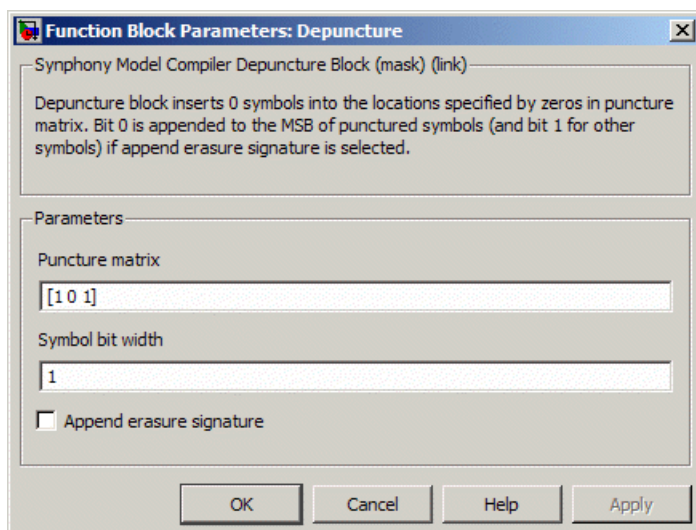
Icon Annotations

The icon for this block displays the following information:

Top annotation	The green annotation shows the puncture pattern set for the block.
----------------	--

Latency	This block has no latency.
---------	----------------------------

Depuncture Parameters



Puncture matrix

Determines the pattern of bits to be inserted into the input data stream. It assumes a punctured input with the bit width equal to the number of ones in the puncture matrix. It creates the output signal by inserting bit zeros in every location where the puncture matrix specifies a 0. Each row of the puncture matrix operates on a different bit in the input data word with the last row corresponding to the LSB of the input data word.

Each 0 indicates a bit to be inserted. For example, an input of UFix_2_0 and a puncture matrix of [1 0 1] results in the insertion of a 0 bit into the input stream and a 3-bit punctured output of UFix_3_0.

You can feed the output of the Puncture block directly to the Depuncture block with no extra blocks between. You must use the same puncture matrix for both blocks to ensure proper decoding.

Symbol bit width

Specifies the width for the symbol bit. The default symbol bit width for the block is 1. If you specify a symbol bit width that is greater than 1, the block operates on symbols of the specified bit length. This is usually the case for soft decoded symbols.

Append erasure signature

When this option is enabled, the block inserts the puncturing status in the output stream. For punctured symbols, it appends bit 0 to the MSB, and for non-punctured symbols, it appends bit 1 to the MSB.

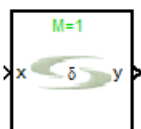
SMC Differentiator

Performs a discrete time differentiation of the input signal.

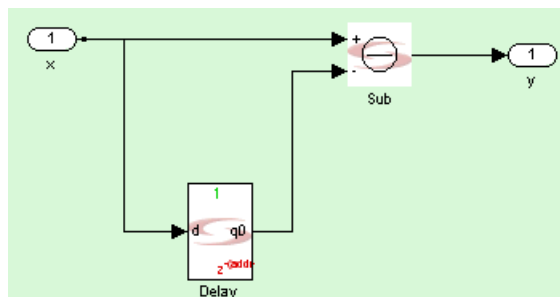
Library

Synphony Model Compiler [Filtering](#)

Description



This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) performs a discrete time differentiation of the input signal.



Automatic Scalar Expansion

If the data input is a vector and the reset or enable port is scalar, the tool expands the scalar reset or enable port to the size of the data input vector. The reset and enable can be either vector or scalar.

Latency

This block has no latency.

Differentiator Parameters

Function Block Parameters: Differentiator

Synphony Model Compiler Differentiator (mask) (link)

The Differentiator block does a discrete time differentiation of the input signal.

Parameters

Differential Delay (M)

1

Output format Specify

Output word length

16

Output fraction length

8

Output datatype signed

☐ Output saturate on overflow (wrap if not selected)

☐ Output round towards nearest on underflow (truncate if not selected)

☐ Reset port

☐ Enable port

OK Cancel Help Apply

Differential Delay (M)

This block uses a shift register block internally, and the Differential Delay parameter sets the latency of the shift register block.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round towards nearest on underflow

Determine how overflow and underflow are treated. The options are only available when Output format is set to Specify. For descriptions of these parameters, see the following:

Output saturate on overflow	Saturates or wraps the overflow; see Overflow Saturation Options, on page 585 for details.
Output round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

SMC Divider

Calculates the fixed-point fractional division of two inputs, A and B.

Library

Synphony Model Compiler [Math Functions](#)

Description



The block takes two scalar or vector inputs with fractional parts and performs fixed-point fractional division on them. The block only supports real inputs, not complex inputs. In Reciprocal mode, the block takes a single input and outputs the reciprocal of that input.

Constant Propagation

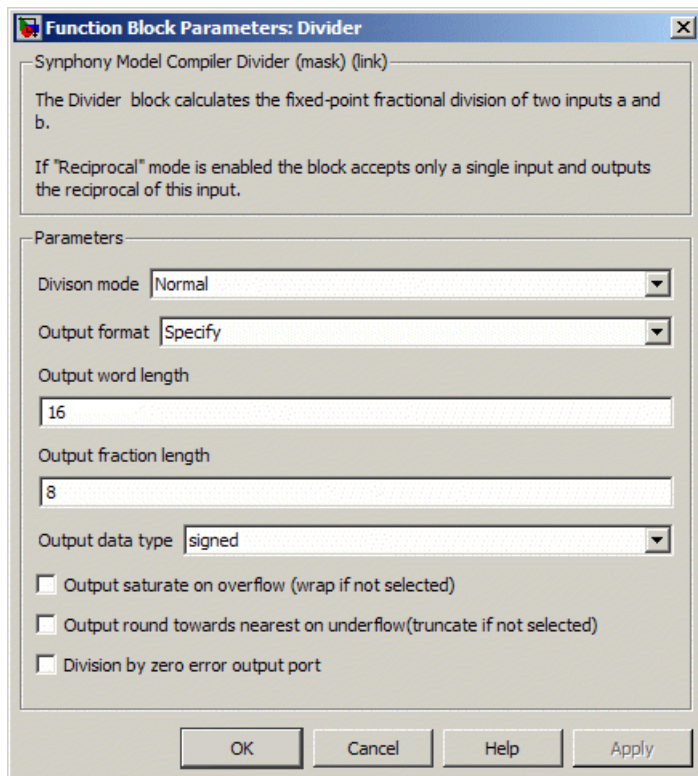
This block propagates constants as described in [Constant Propagation, on page 731](#).

Icon Annotations

The icon for this block displays the following information:

Input and Output Annotations	The input and output annotations reflect the mode set: Normal or Reciprocal. See Division mode, on page 180 for descriptions of the modes and icons.
Latency Annotation	There is no latency introduced by this block.

Divider Parameters

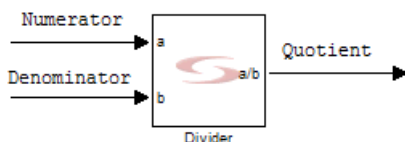


Division mode

Determines the operating mode for the block. The icon reflects the selected operating mode.

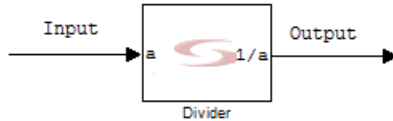
- Normal (default)
The block takes two inputs (numerator/dividend, denominator/divisor) and produces a single division output (quotient/result) as follows, where e is error due to finite precision and is not output:

$$N = Q * D + e$$



– Reciprocal

The block takes a single input and returns the reciprocal of the input (1/input).



Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	You can only set this option to Specify. See Output Format, on page 583 for details.
---------------	--

Output word length	Output Word Length, on page 584 . You can also specify word length in terms of the variables listed below.
--------------------	--

Output fraction length	Output Fraction Length, on page 584 . You can also specify word length in terms of the variables listed below.
------------------------	--

Output data type	Output Data Type, on page 584
------------------	---

You can also specify output word length and output fraction length with the following variables that specify data type information for the block inputs:

syn_num_wl	Word length of the numerator input
------------	------------------------------------

syn_num_fl	Fraction length of the numerator input
------------	--

syn_num_dt	Numerator input sign: 1 - signed input; 0 - unsigned input
------------	--

syn_den_wl	Word length of the denominator input
------------	--------------------------------------

syn_den_fl	Fraction length of the denominator input
------------	--

syn_den_dt	Denominator input sign: 1 - signed input; 0 - unsigned input
------------	--

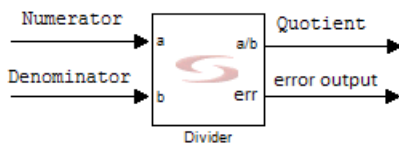
Output saturate on overflow, Output round towards nearest on underflow

Determine how overflow and underflow are treated. The options are only available when Output format is set to Specify. For descriptions of these parameters, see the following:

Output saturate on overflow	Saturates or wraps the overflow; see Overflow Saturation Options, on page 585 for details.
Output round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Division by zero error output port

When enabled, creates an error output port that indicates a division by zero operation. During Simulink simulation, if the block receives 0 as the denominator, it outputs high through this port. The block icon reflects the port if you enable this option:



SMC DivMod

Calculates the integer division and/or modulo function of two inputs, A and B.

Library

Synphony Model Compiler [Math Functions](#)

Description



This block calculates the integer division and/or modulo function of two inputs, A and B. The block also supports F-division. The divide and modulo functions provide the relationship stated by the division and modulus algorithm: given two integers A and B, with B not equal to 0, there are unique integers Q and R such that

$$\begin{aligned} A[n] &= Q[n] * B[n] + R[n] \\ A[n] &= (A[n] \text{ DIV } B[n]) * B[n] + A[n] \text{ MOD } B[n] \\ A[n] &= (A/B)[n] * B[n] + A[n] \% B[n] \end{aligned}$$

For a detailed discussion of division algorithms, see [Overview of Division Algorithms, on page 188](#). Computer languages are notoriously inconsistent in their implementation of mod for negative numbers. Synphony currently uses T-division, which is described more fully in [Division algorithm, on page 185](#).

The block supports vector inputs. If there are vector inputs, division and modulus algorithms are applied for each corresponding vector element.

Constant Propagation

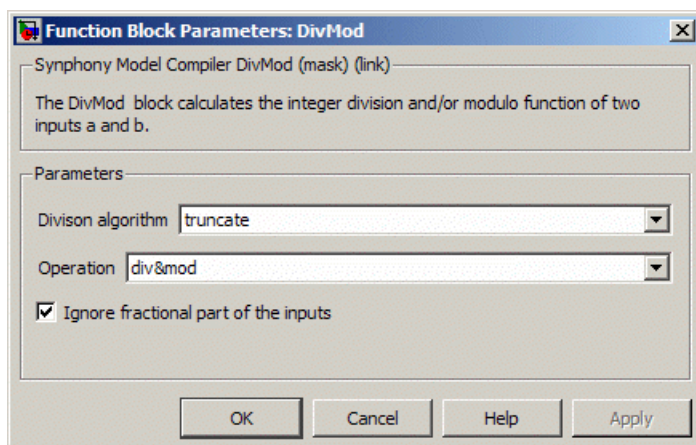
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Icon Annotations

The icon for this block displays the following information:

Top Annotation	The green annotation at the top of the block indicates the division algorithm being used for the operation: T = truncate F = floor
Latency Annotation	There is no latency introduced by this block. However, it can significantly slow down performance for large divisors, and you might require retiming through extra latency to get reasonable performance in the design.

DivMod Parameters



Division algorithm

Specifies the division algorithm to be used. You can select either Truncate or Floor. The behavior varies, based on the setting of the Ignore fractional parts of the inputs option:

- With Ignore fractional parts of the inputs enabled (default):

Truncate	Uses T-division, which truncates any fraction of the quotient. This is the same as rounding towards zero. This behavior matches the MATLAB <code>fix(A/B)</code> and <code>rem(A,B)</code> functionality. The algorithm satisfies $A = Q*B + R$ equality, where $A*R \geq 0$ (if nonzero, A and R have the same sign.) See T-Division , F-Division , and E-Division , on page 189 for details.
----------	---

Floor	Uses F-division, where the quotient is rounded towards minus infinity. This matches the MATLAB <code>floor(A/B)</code> and <code>mod(A,B)</code> functionality. The algorithm satisfies $A = Q*B + R$ equality, where $B*R \geq 0$ (if nonzero, B and R have the same sign.) See T-Division , F-Division , and E-Division , on page 189 for details.
-------	---

If the inputs of the DivMod block are not integers, the tool ignores the fraction and executes the operation on the value of the integer portion of the signals.

For the exception value $B=0$, the division Q becomes the largest negative number for a negative dividend, zero for a zero dividend and the largest positive number for a positive dividend. The modulus R is always equal to A , satisfying $A=Q*B+R$.

- With Ignore fractional parts of the inputs disabled:

Truncate	Uses T-division, which truncates any fraction of the quotient. This is the same as rounding towards zero. See T-Division , F-Division , and E-Division , on page 189 for details. This operation matches this MATLAB functionality: <code>rem()->mod(a%b)</code> block output.
----------	---

Floor	Uses F-division, where the quotient is rounded towards minus infinity. See T-Division , F-Division , and E-Division , on page 189 for details. This operation matches this MATLAB functionality: <code>mod()->mod(a%b)</code> output of the block.
-------	---

Operation

Allows you to set the mode of operation, where only the ports that are needed are made available.

- div sets the mode where only the division output port is required.
- mod sets the mode where only the modulus output port is required.
- div&mod sets the mode where both the division and modulo output ports are required.

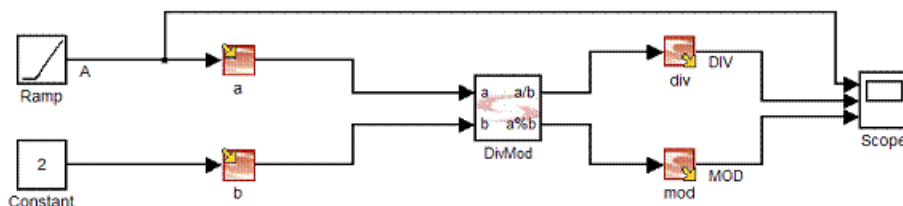
Ignore fractional part of the inputs

Lets you use fractional inputs or ignore them. For details on how the setting affects the division operation, see the description for [Division algorithm, on page 185](#).

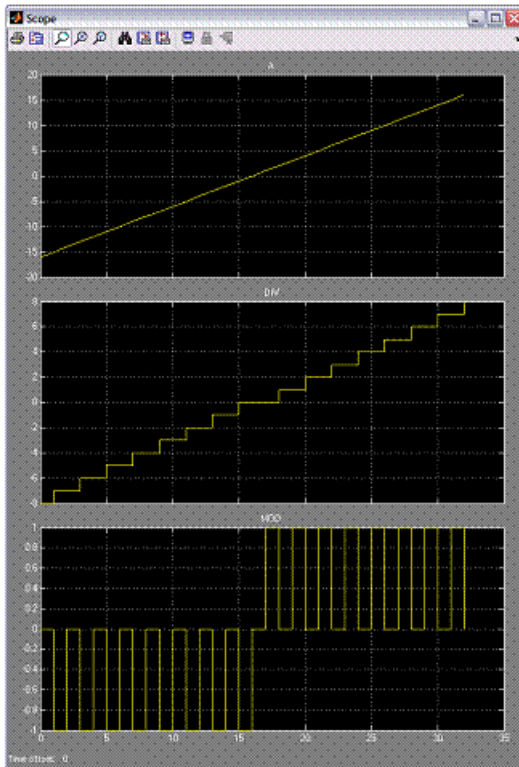
- Enabled (default):
Executes the operation on just the integer part of the input, ignoring the fractional part. This mode is compatible with older versions of the tool. If you open an older design, the blocks automatically have this setting enabled, and you do not have to update your library.
- Disabled:
Takes the fractional part of the input signals into account when it executes the operation. When you use this setting, the blocks will not be backwards-compatible with older libraries that do not have this functionality.

Example

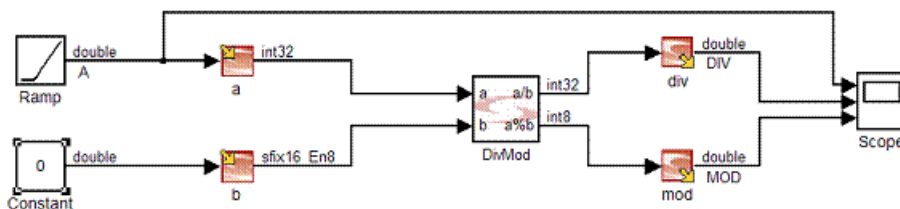
Consider the following sweep:

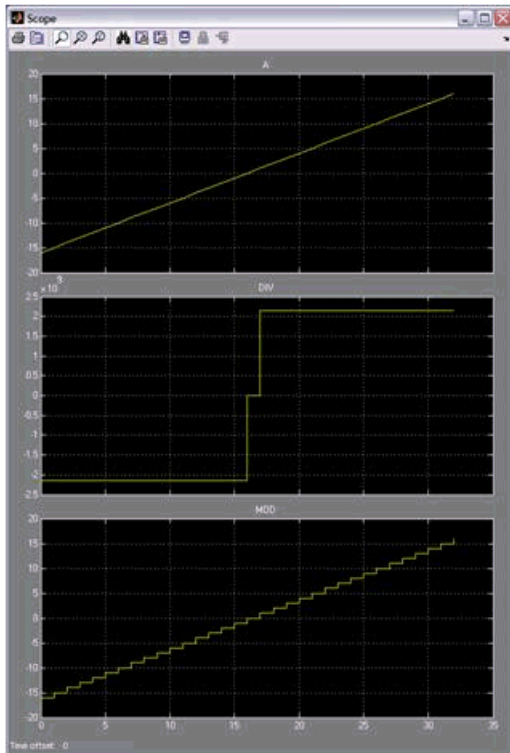


The result of the F-division and corresponding mod implemented by Symphony shows the truncation of the fractions:



Notice the behavior when the divisor is zero:





Overview of Division Algorithms

This section starts with basic definitions, and then discusses different approaches to division. It also provides examples of different kinds of division results.

Division Definitions

In the following discussion, the integer Q is the quotient, R is the modulus, B is the divisor, and A is the dividend.

However, the value of the modulo function depends on the definition of the division¹:

```

QT = trunc(A/B)      T-division (Truncated, ISO C99)
QF = floor(A/B)      F-division (Floor)
QR = round(A/B)      R-division (Round)
QC = ceil(A/B)       C-division (Ceil)
R = A - Q*B

```

There is also a modulo-centric approach to this problem²:

```

0 ≤ R < |B|
QE = (A - R) / B      E-division (Euclidean)

```

T-Division, F-Division, and E-Division

T-division or truncate division is an ISO standard and used in modern processors. Hence the ANSI C functions “/” and “%” tend to be implemented with T-division. The MATLAB rem function is based on the T-division. The following properties hold:

```

A divT(-B) = (-A) divT B = -(A divT B)
A modT(-B) = A modT B

```

F-division or floor division is described and promoted by Knuth³. The MATLAB mod function is based on F-division.

E-division or Euclidean division has the following mathematical advantages:

```

A divE(-B) = -(A divE B)
A modE(-B) = A modE B

```

1. Daan Leijen. *Division and Modulus for Computer Scientists*. University of Utrecht, 2001.
2. Raymond T. Boute. *The Euclidean Definition of the Functions div and mod*. In ACM Transactions on Programming Languages and Systems (TOPLAS), 14(2):127-144, New York, NY, USA, April 1992. ACM press.
3. Donald E Knuth. *The Art of Computer Programming, Vol 1, Fundamental Algorithms*. Addison-Wesley, 1972.

Example: Division Algorithm Variations

The differences between some common division algorithms are best illustrated through an example:

A,B	(QT,RT)	(QF,RF)	(QR,RR)	(QC,RC)	(QE,RE)
(+13,+2)	(+6,+1)	(+6,+1)	(+7,-1)	(+7,-1)	(+6,+1)
(+13,-2)	(-6,+1)	(-7,-1)	(-7,+1)	(-6,+1)	(-6,+1)
(-13,+2)	(-6,-1)	(-7,+1)	(-7,+1)	(-6,-1)	(-7,+1)
(-13,-2)	(+6,-1)	(+6,-1)	(+7,+1)	(+7,+1)	(+7,+1)

Diagnostics

You see the following warning message when one or both of the DivMod block inputs has a fractional part:

Warning: block 'test/DivMod': Fractional type fed into Integer division. Fraction bits will be ignored

You see the following warning message when the denominator of the DivMod block has a zero value:

Warning: block 'test/DivMod': Division by zero!

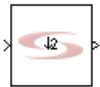
SMC Downsample

Decreases the sample rate of the input by removing samples.

Library

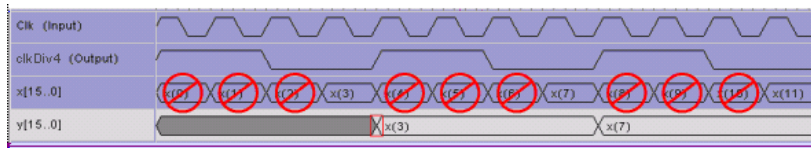
Synphony Model Compiler [Signal Operations](#)

Description

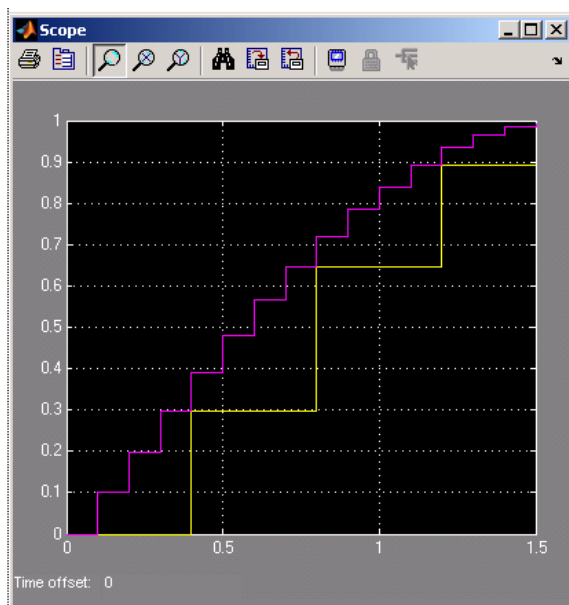


The Synphony Model Compiler Downsample block decreases (downsamples) the sample rate of the input by removing samples. If M is the specified downsampling rate, for every M samples at the input, the software keeps 1 sample at the output. This means that the sample rate at the output is the input sample rate divided by the downsampling rate M . The kept sample can be one of the samples of M which is specified by the sample offset.

This figure shows the corresponding signal manipulation for a downsample rate of 4 and a sample offset of 3, along with the downsample implementation clock and signal dependencies:



The following figure shows a practical simulation result for the implementation:



The software uses a delay at the input (based on the input sample rate), followed by a standard downsample operation, where it keeps the first sample of every input frame (M samples), and discards the other (M-1) samples from the input frame.

For information about using the Downsample block in multi-rate designs, see [Multi-Rate Design, on page 717](#).

Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

The latency of the Downsample block is determined by the offset:

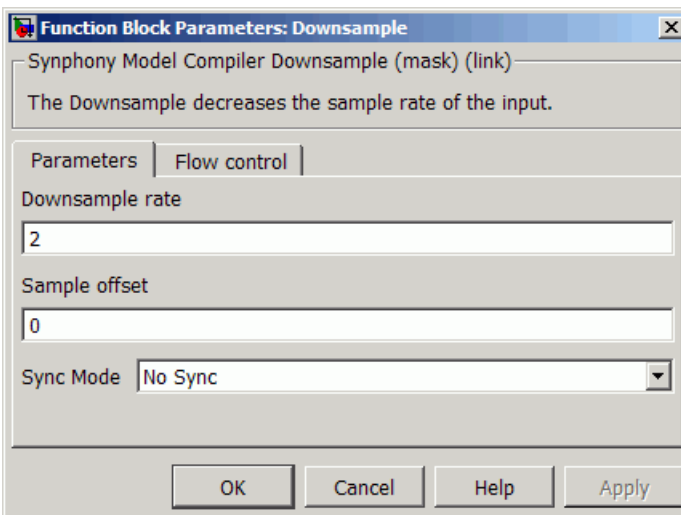
Offset	Latency
0	0
All other cases	Downsample rate - sample offset at the input

Downsample Parameters

This block has two tabs, Parameters and Flow control.

Parameters Tab

Set general downsampling parameters on this tab.



Downsample rate

Specifies the value by which the input sample rate is divided to get the output sample rate.

Sample offset

Specifies the sample offset. For a description and a discussion of sample rates, see [Multi-Rate Design, on page 717](#).

Sync Mode

Specifies the synchronization mode for the output. When the clock counter reaches the position you specify in this options, the synchronized output produces 1. You can choose one of the following modes:

Mode	Description
No Sync	There is no synchronized output.
When Output Changes	The sync output produces 1 pulse for every sample taken.
Aligned with Offset	The sync output is synchronized with the offset.
Right before Offset	The sync output is synchronized with one sample before the offset

Refer to the waveforms shown in [Downsample Timing Waveforms, on page 195](#) for additional information about these modes.

Flow Control Tab

This tab covers flow control.

Forward flow control

Determines forward flow control. When enabled, creates two extra ports: `srnyi` (source ready in) and `srnyo` (source ready out). With forward flow control, the downsampling operation is controlled by the preceding block through the `srnyi` signal. Based on this signal, input data is downsampled and the output signal `srnyo` is asserted.

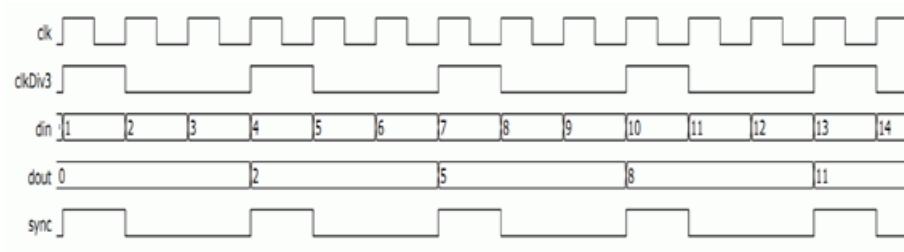
In forward flow control, data input is considered to be valid only when input signal `srnyi` is asserted. If the downsampling factor is M , one of the M valid input samples is placed on the `dout` output bus. At the same time, the output signal `srnyo` is asserted.

See [Downsample with Forward Flow Control Enabled, on page 196](#) for a timing waveform.

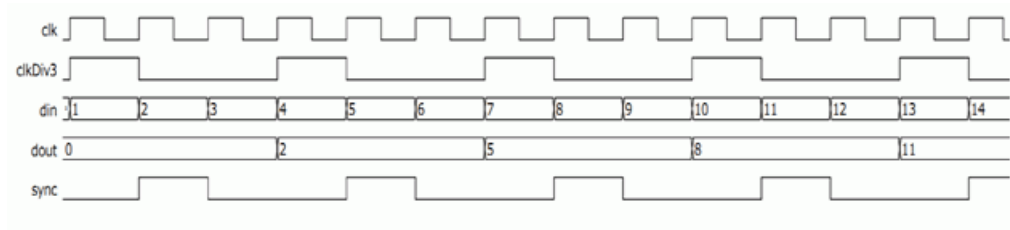
Downsample Timing Waveforms

The following figures show timing waveforms for the Downsample block. For all the examples shown, the downsampling factor is 3, and the sample offset is 1.

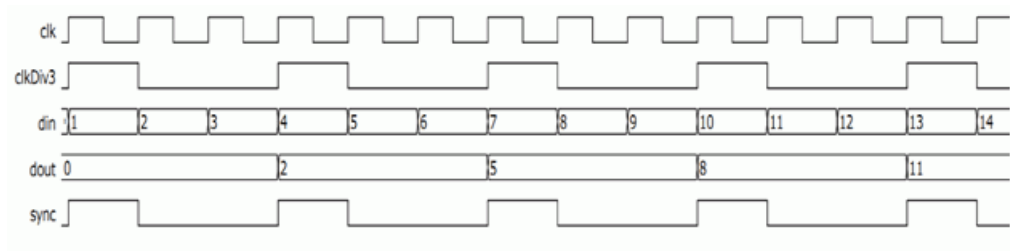
Sync Mode = When Output Changes



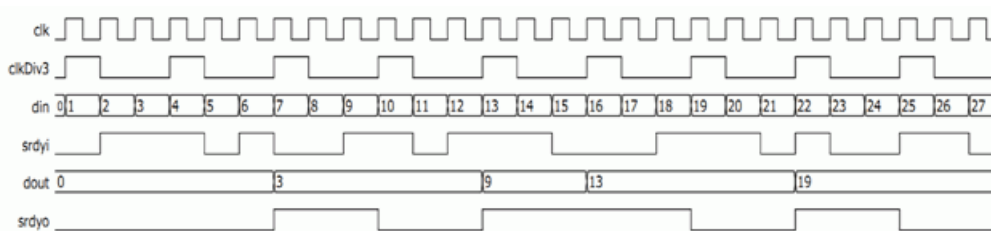
Sync Mode = Aligned with Offset



Sync Mode = Right Before Offset



Downsample with Forward Flow Control Enabled



SMC Edge Detector

Outputs a unity amplitude pulse of one sample period in response to a synchronous transition from high to low or low to high.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Edge Detector block outputs a unity amplitude pulse for one sample period in response to a synchronous transition from high to low or low to high. This block detects edges by comparing the input state at the previous and the current sampling instances. Transition types are:

- Rising
- Falling
- Either

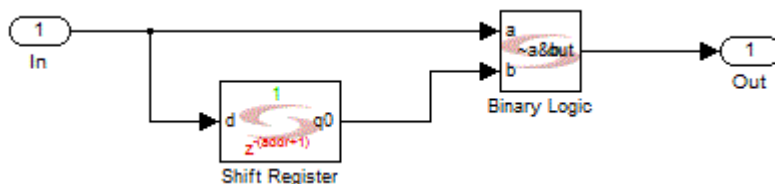
If transition type is either rising or falling, the initial condition is chosen to be low and high respectively.

If the transition type is either, specify the desired initial condition as low or high.

A pulse is always generated at the first sampling period if:

- The Edge type is set to Rising and the first input state is high OR
- The Edge type is set to Falling and the first input state is low OR
- The Edge type is set to Either and the initial condition and the first input state are different.

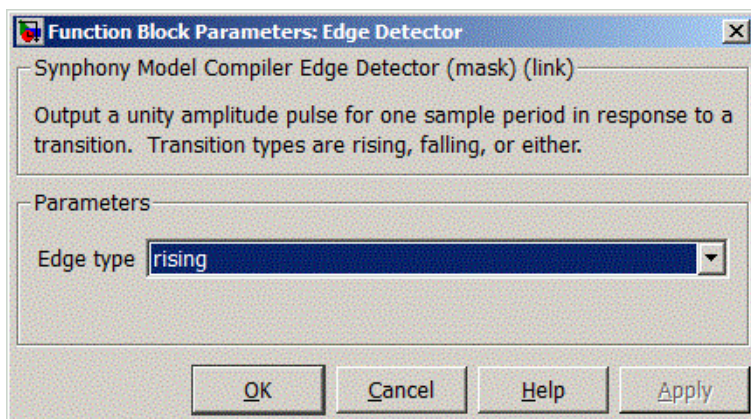
This block is a custom block (see [Primitives and Custom Blocks, on page 800](#), for a definition). The following figure shows the internal modeling when edge type is falling:



Latency

This block has no latency.

Edge Detector Parameters



Edge type

Specifies whether the edge type is Rising, Falling or Either. A unit amplitude pulse of one sample width is generated only if:

- The Edge type is Rising and the input state is low for the previous sampling instance and is high for the current instance.
- The Edge type is Falling and the input state is high for the previous sampling instance and is low for the current instance.

- The Edge type is Either and the input state is different for the previous sampling instance and the current instance.

Initial Condition

This option is available when Edge type is selected as Either. You can specify the previous state as high or low for the first sampling instance.

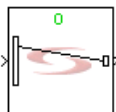
SMC Extract

Extracts specified bits from the input signal.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Extract block extracts specified bits from the input signal. The output is unsigned, but you can recast the output using the Recast block ([SMC Recast, on page 424](#)).

Constant Propagation

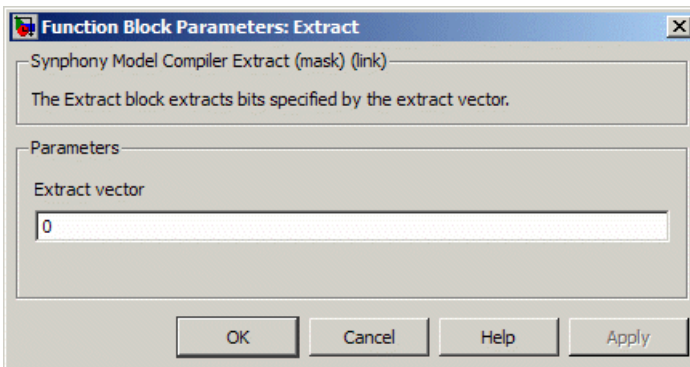
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Icon Annotations

The icon for this block displays the following information:

Top Annotation	The green annotation at the top of the block specifies the bit vector for extraction. See Extract Vector, on page 201 for more detail.
Latency Annotation	There is no latency introduced by this block.

Extract Parameters



Extract Vector

Specifies the bit vector to extract from the incoming signal. You cannot extract bit indices greater than the input word length. You cannot extract negative bit indices. The output data type depends on the size of the extract vector.

You can also specify the following in this field:

- `syn_inp_wl == input wordlength`
- `syn_inp_fl == input fraction length`
- `syn_inp_dt == input data type (1 if signed)`

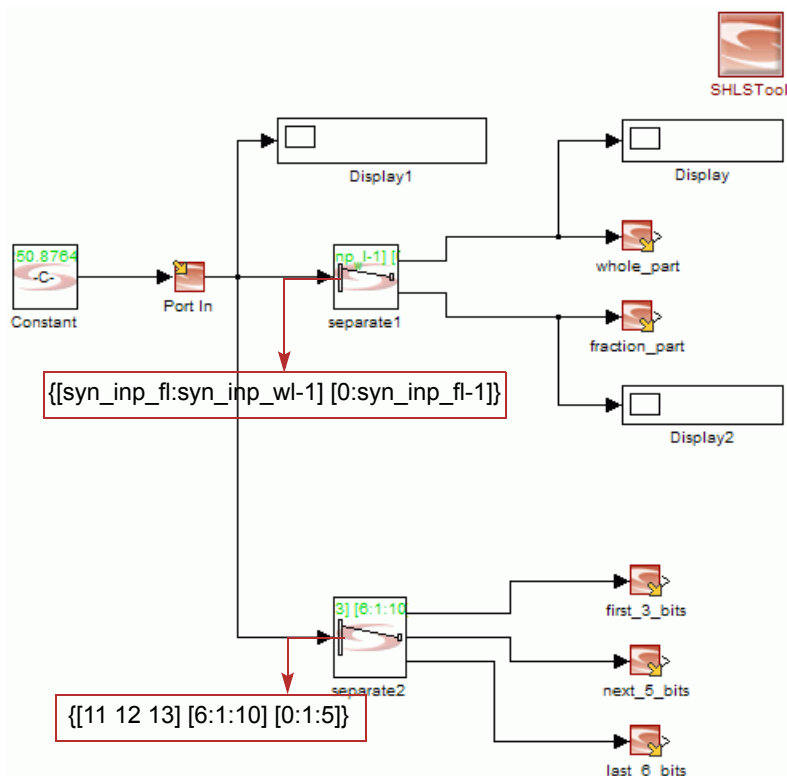
For example:

<code>0:syn_inp_wl-1</code>	Recovers whole signal
<code>[0:syn_inp_fl-1]</code>	Recovers fraction bits
<code>syn_inp_wl-1:-2: 0</code>	Recovers even bits in back order
<code>syn_inp_dt * 3 + 2: "secret formula J"</code>	

To generate multiple outputs for the Extract block, specify a cell array, where *n* extract vectors results in *n* outputs. The extract vectors do not have to be the same. Specify the cell array using the `{ }` operator, and use square brackets `[]` to indicate the extract vectors for each output. The following example shows the syntax for a group of *n* vectors:

```
{ [ ] [ ] ... [ ] }
```

In the following design, `separate1` and `separate2` have multiple outputs, because their bit vectors have been defined as `{[syn_inp_fl:syn_inp_wl-1] [0:syn_inp_fl-1]}` and `{[11 12 13] [6:1:10] [0:1:5]}` respectively:



SMC FDATool

Opens the Simulink FDATool interface.

Library

Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler FDATool block opens the Simulink FDATool interface where you can design, analyze, and implement floating-point FIR and IIR filters.

The Synphony Model Compiler blockset includes an IIR and various FIR blocks, but this block provides an interface to the FDATool software, which is part of the MATLAB® Signal Processing Toolbox. The Synphony Model Compiler FDATool block will not function properly unless the Signal Processing toolbox is installed. The Simulink FDATool provides a powerful graphical interface for defining digital filters. Through this block, you can use the Simulink FDATool interface to define an FDATool object, and then store it as part of the Synphony Model Compiler model.

For procedural information about using this block, see [Defining FIR Filter Coefficients with FDATool](#), on page 768.

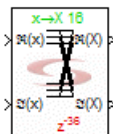
SMC FFT

Implements a fully pipelined Fast Fourier Transform (FFT).

Library

Symphony Model Compiler [Transforms](#)

Description



The Symphony Model Compiler FFT block implements a fully pipelined Fast Fourier Transform. When it is used to perform the block-wise FFT of a streaming signal, you only require a reset for the first block.

Automatic Scalar Expansion

If the input to the FFT block is a vector, both the real and imaginary inputs must be vectors. The reset and enable can be either vector or scalar. If the reset or enable is scalar, the tool expands it to the size the real or imaginary vector inputs.

Constant Propagation

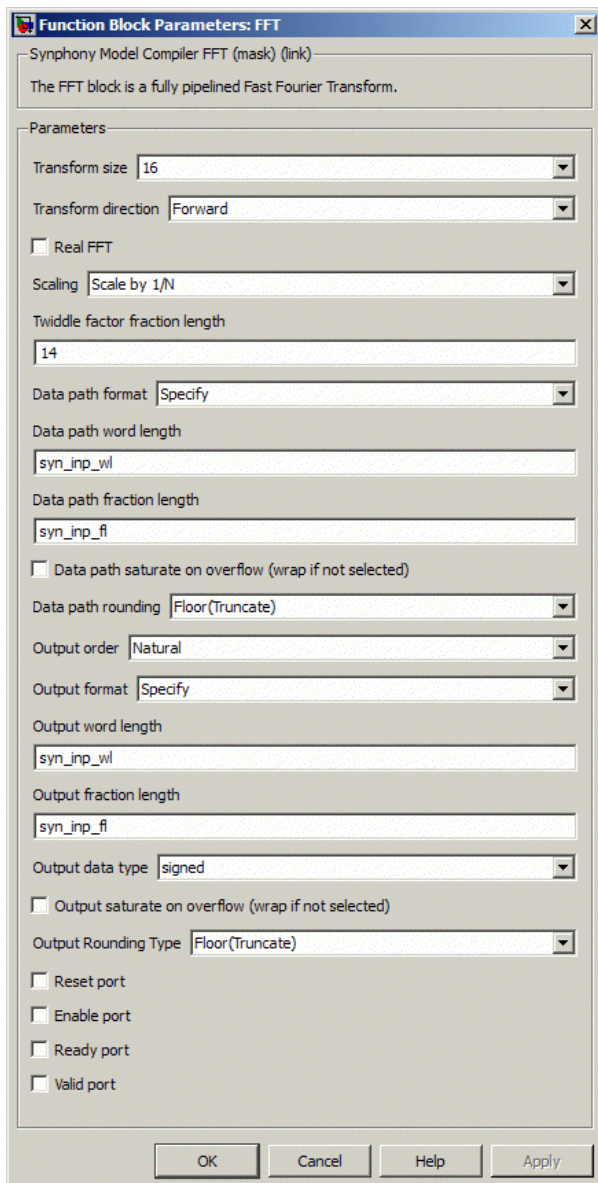
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

Latency is a complex function of frame size and output order:

Transform Size	Latency (Bit-reversed Output)	Latency (Natural Order Output)
16	19	36
32	38	71
64	70	135
128	137	266
256	265	522
512	524	1037
1024	1036	2061
2048	2063	4112
4096	4111	8208
8192	8210	16403
16384	16402	32787
32768	32789	65558
65536	65557	131094

FFT Parameters



The dialog box is titled "Function Block Parameters: FFT". It contains a link to the "Synphony Model Compiler FFT (mask) (link)" and a description: "The FFT block is a fully pipelined Fast Fourier Transform."

Parameters

- Transform size: 16
- Transform direction: Forward
- ☐ Real FFT
- Scaling: Scale by 1/N
- Twiddle factor fraction length: 14
- Data path format: Specify
- Data path word length: syn_inp_wl
- Data path fraction length: syn_inp_fl
- ☐ Data path saturate on overflow (wrap if not selected)
- Data path rounding: Floor(Truncate)
- Output order: Natural
- Output format: Specify
- Output word length: syn_inp_wl
- Output fraction length: syn_inp_fl
- Output data type: signed
- ☐ Output saturate on overflow (wrap if not selected)
- Output Rounding Type: Floor(Truncate)
- ☐ Reset port
- ☐ Enable port
- ☐ Ready port
- ☐ Valid port

Buttons: OK, Cancel, Help, Apply

Transform Size

Sets the size of the FFT block. For sizes which are an integer power of 4, the software uses the Radix-4 algorithm. For other sizes, the software uses a Radix-2 stage, followed by Radix-4 stages.

Transform Direction

Sets the operation direction for the block.

- Inverse sets the transform direction to inverse.
- Forward is the default, and sets the transform direction forward.

Real FFT

Enable this option for FFTs with non-complex inputs. When enabled, the imaginary input port disappears and internally it feeds 0 as imaginary input. Using this option results in some advantages:

- Reduced memory consumption in baseline and multi-channel mode for natural orders
- Reduced memory consumption in folded modes for natural orders.
- Reduced memory consumption in baseline and multi-channel mode for bit reversed orders
- Fewer multipliers for odd power-of-2 FFT in baseline and multi-channel mode. e.g. 128 pt FFT uses 12 multipliers, a 128 pt real FFT uses 10 multipliers.

Scaling

Specifies whether or not the FFT is to be scaled. Scaling is applied after the butterfly stages to prevent bit growth from the beginning. Floor rounding (truncation) is used for the scaled data. See [Underflow Rounding Options, on page 585](#) for details.

N is the FFT size.

- Scale by $1/N$ divides the DFT summation by N.
- Scale by $1/\text{Sqrt}(N)$ divides the DFT summation by the square root of N.
- No scaling does not scale the FFT.

Twiddle Factor Fraction Length

Determines the precision of the FFT block by setting the fraction length for a twiddle factor, in bits. The specified value must be an integer

between 1 and 50. To increase precision, increase the fraction length for the twiddle factor. You can also specify the twiddle factor in terms of the variables `syn_inp_wl` and `syn_inp_fl`.

Data Path Format

Determines data path format. You can set one of these options:

- Automatic sets the data path format to one that uses the maximum of input and output fractions, and the smallest bit width that guarantees no overflow.
- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation after twiddle factor multiplications.
- Specify uses the user-defined data type to determine the cast for internal calculations. For this block, data path casting is done at the input, after the twiddle factor multiplications, and at the block output. Overflow only occurs at the points where data casting is done. The rest of the calculations are overflow-free, regardless of the specified data type.

Data Path Word Length

Determines the word length of the data path in bits. It only becomes available when you set Data Path Format to Specify. You can also specify the word length in terms of the variables `syn_inp_wl`, `syn_inp_fl`, `syn_coef_wl`, and `syn_coef_fl`.

Data Path Fraction Length

Sets the fraction length of the data path in bits. It only becomes available when you set Data Path Format to Specify. You can also specify the fraction length in terms of the variables `syn_inp_wl`, `syn_inp_fl`, `yn_coef_wl`, and `syn_coef_fl`.

Data path saturate on overflow

Determines how data path overflow is treated. Enable the option to saturate the overflow, and disable it to wrap the overflow. See [Overflow Saturation Options, on page 585](#) for details. This option is only available when you set Data Path Format to Specify.

Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 585](#) for details. This option becomes available when Data path format is set to Automatic or Specify.

Output Order

Sets the output order for the block. This option determines the latency of the block; see [Latency, on page 204](#) for a table of values.

- Natural is the default. It sets the output order of the FFT results to the natural order.
- Bit-reversed sets the pipelined FFT results to bit-reversed order.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584 . You can also specify word length in terms of variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_coef_wl</code> , and <code>syn_coef_fl</code> . These variables are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584 . You can also specify fraction length in terms of variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_coef_wl</code> , and <code>syn_coef_fl</code> . These variables are described in Special Variables, on page 588 .
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding type

Determine how output overflow and underflow are treated. The options are only available when you set Output Format to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details. The saturation option you specify is applied separately at the output, after the data path saturation option you specified in Data path saturate on overflow.
Output rounding type	See Underflow Rounding Options, on page 585 for details about the rounding options available. The rounding option is applied separately to the output, after the data path rounding option you specified in Data path rounding.

Reset port

When enabled, it creates a local reset (*rst*) for the FFT block, clearing the pipeline. The reset is active high. If you disable this option, the block outputs non-valid data for the depth of the pipeline.

Enable port

When enabled, it creates an enable (*en*) port, which provides control over the Enable status of the block. If you enable this pin, you cannot use folding optimizations, because it leads to verification mismatches.

If this option is disabled, the software does not create an *en* port and the FFT operation is always enabled.

Ready port

When enabled, this option outputs a ready pulse (*rdy*), and valid FFT data streams out on the clock after the valid is asserted. A typical use of this pin is to feed the ready pin of a forward FFT to the reset pin of an inverse FFT. When disabled, the tool does not create a ready pin.

Valid port

When enabled, this option creates an active high signal (*vld*) that frames the valid output data. A typical use of this pin is to feed the valid pin of a forward FFT to the enable pin of an inverse FFT. If this option is disabled, the tool does not create a valid pin.

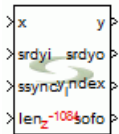
SMC FFT2

Implements a Fast Fourier Transform that supports both serial and parallel inputs.

Library

Synphony Model Compiler [Transforms](#)

Description



The FFT2 block is a high-speed Radix2 DIF streaming FFT block. This block provides additional functionality and better results than the FFT block. Key features include the following:

- Parallel processing of 1, 2, 4, 8, 16, or 32 input samples
This yields very high throughput for FFT designs, which is required by many spectral monitoring applications. For example, you can use this block to implement a 1024-point block with 32 parallel inputs, that runs at 300 MHz on the FPGA and can process 9.6 GSamples/sec.
- Variable transform length, from 2*number of parallel inputs to 64K
- Multichannel FFT operations
See [Multichannel FFT, on page 212](#) for details.
- Easy interface to common interface protocols through flow control
See [FFT2 Flow Control, on page 213](#) for details.
- Optimized area and performance for the target device

Multichannel FFT

For a multichannelized operation, set the number of parallel inputs to 1 and the multiplexed channel parameter to a value greater than 1. With multiple channels, the FFT2 block uses a faster clock to share the twiddle factor generation and multiplication logic for each butterfly. This results in significant area savings over the instantiation of multiple independent FFTs.

Transform Length and Input/Output Data Formats

To configure transform length at runtime, disable the Constant Transform Length option. Transform length is now configured at runtime through the `len` (length) port. The transform length values can range from twice the number of parallel inputs to the maximum transform size specified.

When runtime reconfiguration of transform length is not required, enable Constant Transform Length. This setting offers significant area and timing benefits. When this option is enabled, the `ssynci` and `ssynco` port are available, and you must specify the input to the `length` port as follows:

$\text{ceil}(\log_2(\text{actual transform length}))$

Also, the nominal latency does not change for the block even if the `len` input has changed. Configure the input vector `x` as follows:

<number of channels / number of parallel inputs of I> < Same number of Q >

For example, if you specify 4 parallel inputs, you must provide the `x` vector in the form `{I0,I1,I2,I3,Q0,Q1,Q2,Q3}`. Use the same format when the number of parallel inputs is 1 and the multiplexed channels specified is greater than 1.

The `y` output also uses this format. All inputs except the `x` input must be dimension 1. For DIF FFT operations, the input is always in-order, while the output is bit-reversed order. The `y_index` can provide the index for the output value that is its real location in the current frame. Therefore, bit-reverse ordering is easily done at the output, using a ping-pong memory, the `y_index` as the write address, and an up counter as the read address.

Transform sizes must be an integer power of 2 up to a maximum of 2^{16} .

For a serial input FFT2 block, the output is in bit-reversed order. For a parallel input FFT2, the tool uses the following calculation. For an FFT2 block with transform length N and K complex parallel inputs, the output is a $2K$ vector. The first K outputs are the real components and the next K outputs will be the imaginary components of the FFT.

Within each component, an output value of $(\text{bitreverse floor}(i/k) + p)$ is available at time instance l , and this is the p th element of this output vector:
 $0 \leq l < N/K$, $0 \leq p < K$.

Bit Growth Handling

Each butterfly in the FFT2 block incorporates automatic bit growth, where the output word length is one more than the input word length. The fraction length remains the same, and simple truncation occurs after twiddle factor multiplication.

The output word length can be $\text{ceil}(\log_2(\text{Maximum transform length}))$ greater than the input word length. As the output is also unscaled, this can result in an equivalent gain of $1/\sqrt{\text{actual length}}$ compared to the theoretical value. When the actual length value available through the `len` port is less than the maximum transform length, the $\text{ceil}(\log_2(\text{Maximum transform length/actual length}))$ number of bits from the most significant output can safely be removed without the danger of wrapping.

Depending on the input word length for each butterfly, the twiddle factor multiplication circuit is derived automatically and optimized for the bit width needed in the DSP macro capabilities of the target device. If the word length exceeds the maximum input word length of the target DSP, the tool automatically splits the inputs for optimal DSP mapping.

FFT2 Flow Control

The FFT2 block provides forward flow control through the `srdyi` and `ssynci` input ports and the `srdyo`, `ssynco`, and `sofo` output ports. Descriptions about these ports for variable-length FFT are provided below:

<code>srdyi</code>	Indicates whether the current input sample is valid input for the frame.
<code>ssynci</code>	Indicates when a new value for the transform length can be loaded.

srdo	Is the current output sample for the frame.
ssynci	Is the delayed version of the ssynci input
sofo	Designates the clock cycle when the first valid output for the current frame is available.

An invalid input is denoted by **srdo** going low. **srdo** going low denotes that the corresponding output is invalid. When Constant option length is disabled, the **ssynci** input is active high to indicate when a new value for the transform length can be loaded into the block. **ssynci** can also reset the frame counter for constant length FFTs.

The **srdo** input can go down at any point in the frame, but **ssynci** must be asserted after the current frame is provided in full at the input and before the next input frame starts. This allows the **len** input to change for every FFT frame that is processed. If **ssynci** is asserted in the middle of a frame, the input frame counter resets immediately and a new frame starts processing from the next cycle, with the new length value loaded. The current frame in the pipeline is only partially processed, and the output has invalid values.

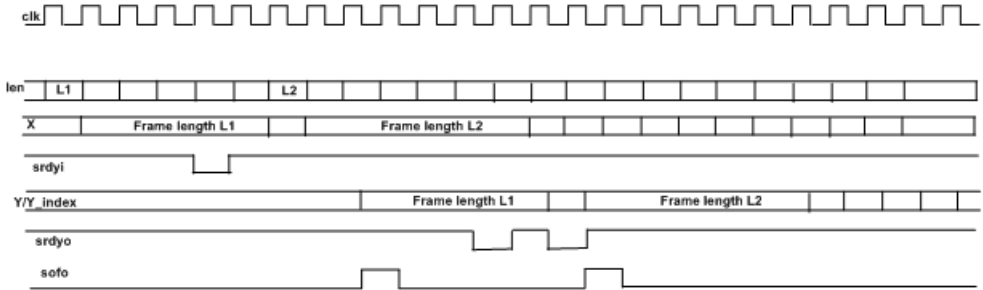
For variable length FFT, assert **ssynci** at the beginning of every frame only when you want to change the transform length. If not, the internal frame counter automatically flags the start of the next frame. The length can only be changed between **frame1** and **frame2** after all the data of **frame1** is provided, and before any **frame2** data is available. This means there must be a 1-cycle gap between **frame1** and **frame2**, if the length needs to change.

Invalid output is denoted by **srdo** going low. Since FFT is an intrinsically framed operation, **srdo** may not go low after **srdo** goes low plus latency, although the average number of clock cycles that **srdo** and **srdo** are high will be the same over time to maintain throughput. **sofo** designates the clock cycle when the first valid output for the current frame is available. **ssynci** is the delayed version of the **ssynci** input, which can be used as a **ssynci** input for a cascaded FFT2 operation. Note that the **ssynci** is available only after **ssynci** has been applied, while **sofo** becomes available at the beginning of every frame.

For constant transform length, the **ssynci** port is optional. If selected, assert the **ssynci** port to synchronize the FFT operation to an external sync event. There is no need to assert **ssynci** for a normal streaming operation; assert it only if the sync event causes a timing change between frames. Asserting **ssynci** interrupts the processing of the input data. Data processing resumes

on the first cycle in which `srdyi` is asserted after `ssynci` was deasserted. So if `ssynci` is unnecessarily asserted when there is no timing change, it will unnecessarily disrupt the operation of the FFT.

The following timing diagram illustrates the flow control operation:



Icon Annotations

The icon for this block displays the latency value, in red.

Latency

Latency is calculated as shown below, where processing pipeline delay depends on the input word length and the target device:

maximum transform length/number of parallel inputs + processing pipeline + 1 delay through butterflies

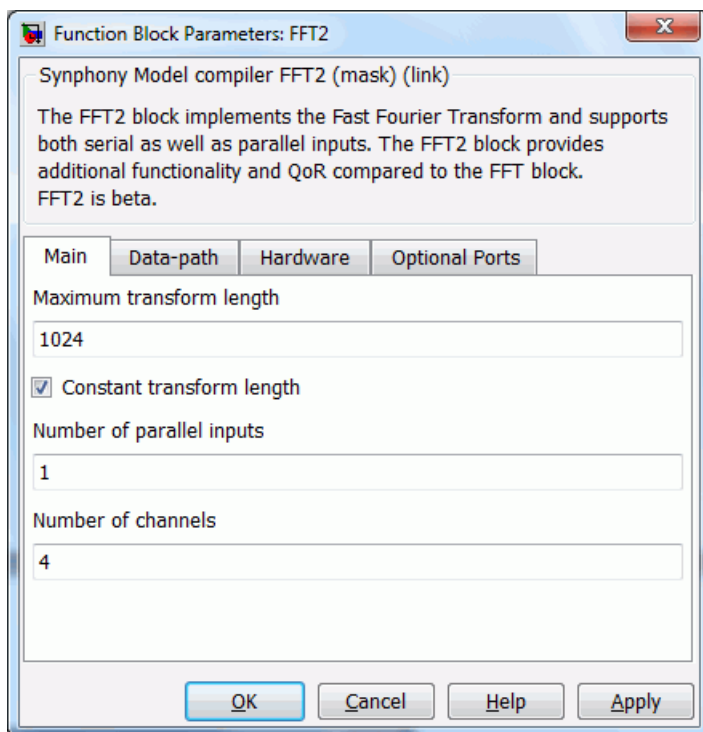
The actual latency of the FFT2 block is also affected by `ssynci` and `srdyi`, as shown below:

<annotated latency> + <number of clock cycles ssynci is high> + <number of clock cycles srdyi is low>

FFT2 Parameters

The FFT2 block parameters are available on various tabs.

Main Tab



Maximum Transform Length

Specifies the maximum transform length of the FFT operation. The value must be an integer power of 2, with a maximum of 2^{16} . The minimum value is $\max(4, 2 \times \text{number of parallel inputs})$. See [Transform Length and Input/Output Data Formats, on page 212](#) for additional information.

Constant Transform Length

Specifies whether to configure the transform length at runtime. When enabled, the `len` and `ssynci` ports are not available. See [Transform Length and Input/Output Data Formats, on page 212](#) for more information.

Number of Parallel Inputs

Specifies the number of parallel inputs to the FFT2 block. Valid values are 1, 2, 4, 8, 16, 32. The input vector size must be twice the value specified for this option. Effective throughput is calculated as follows:

clock frequency* number of parallel inputs

Number of Channels

Specifies the number of channels for the FFT operation. Set a positive integer value. This option is available only when the number of parallel inputs is equal to 1.

If the number of channels is greater than 1, the Fold Across Channel option on the Hardware tab lets you specify whether the twiddle factor multipliers should be reused across channels.

Datapath Tab

The screenshot shows the 'Datapath' tab of the SMC FFT2 configuration window. It contains the following settings:

- Input word length:** 16
- Twiddle factor word length:** 18
- Butterfly bit-growth:** Automatic
- Twiddle-factor output rounding mode:** Floor (Truncate)
- Output Scaling:** 1/N

At the bottom of the window are four buttons: OK, Cancel, Help, and Apply.

Input Word Length

Specifies the word length for the input. If you specify a word length that is less than the actual input word length, the block continues to function properly, but you may lose some optimizations.

Twiddle Factor Word Length

Specifies the word length of the twiddle factor, where the fraction length is set to 2 less than the word length. The maximum value allowed is 18.

Butterfly bit growth

Determines the word length of the butterfly output.

- Automatic
The word length of each butterfly output is one more than the input. The tool adjusts the decimal point position according to the output scaling factor you select.
- None
The word length of each butterfly output is the same as the input, so that the final output word length is the same as the input.

Twiddle-factor output rounding mode

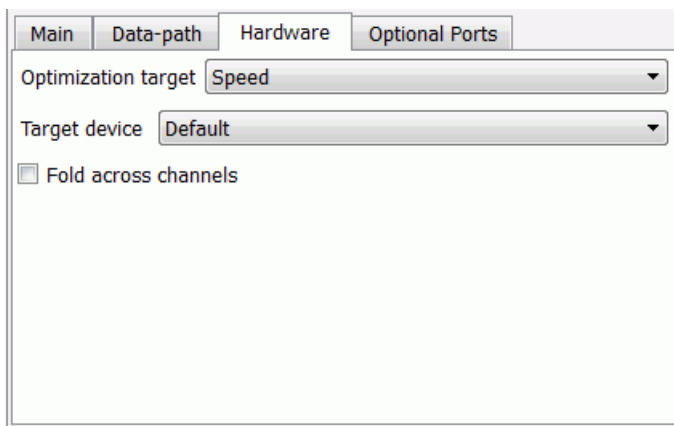
Specifies the rounding algorithm to use after each twiddle factor multiplication: Floor(Truncate), Nearest, Convergent, Fix, Ceil, Round. See [Underflow Rounding Options, on page 585](#) for descriptions.

Output Scaling

Specifies the output scaling factor.

- None
The output is not scaled.
- 1/N
The output is scaled by 1/N. For 1/N scaling in case of variable transform length, N denotes the actual transform length available through the len port
- 1/sqrt(N)
This option is only available for constant transform length.

Hardware Tab



The screenshot shows the 'Hardware' tab selected in a configuration window. The window has four tabs: 'Main', 'Data-path', 'Hardware', and 'Optional Ports'. The 'Hardware' tab is active. Inside the tab, there are two dropdown menus: 'Optimization target' set to 'Speed' and 'Target device' set to 'Default'. Below these is a checkbox labeled 'Fold across channels' which is currently unchecked.

Optimization target

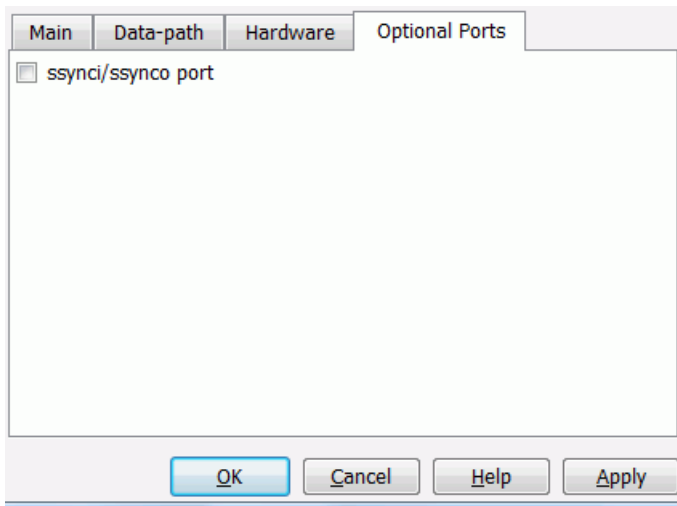
Specifies whether the generated RTL is optimized for speed or area. Setting this option to **Area** can result in lower latency and register count, but the tradeoff is lower speed.

Target Device

Specifies the target device for the FFT2 block.

Fold across Channels

For multi-channel FFTs, specifies whether to reuse twiddle factor multipliers across channels in each butterfly.

Optional PortsTab**ssynci/ssynco Port**

Specifies whether the block includes **ssynci** and **ssynco** input and output ports, respectively. This option is only available when **Constant transform length** is enabled. If **Constant transform length** is disabled, this option becomes unavailable because the **ssynci** and **ssynco** ports are always present. See [FFT2 Flow Control, on page 213](#) for additional information about these ports.

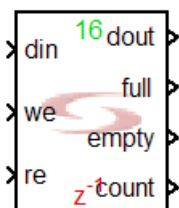
SMC FIFO

Implements a single-rate or multirate FIFO (First in First Out) memory queue.

Library

Synphony Model Compiler [Memories](#)

Description



The Synphony Model Compiler FIFO block implements a single-rate or multi-rate FIFO memory queue. FIFOs are usually used to buffer input and output signals at both the transmit and receive ends of a digital system. For example, in a communication system a transmitter may send data in bursts faster than the receiver can handle it. Such a system requires FIFO buffers that can accept short bursts of high-speed data and then allow the data to be read out as needed. In addition the signals in the data stream burst remain in order, so that the first word entered into the buffer is the first word read out from the buffer. A device that performs these operations is called a *first in-first out* buffer of FIFO. The advantage of using the FIFO block is that it decouples the reader and writer, so that the two operations do not have to operate in lock step.

The tool implements the FIFO block as either a single-rate or multi-rate FIFO, based on the data rates of the read enable and write enable ports. If the two ports have different rates, the tool implements a multi-rate FIFO in the RTL. If the rates are the same, it implements a single-rate FIFO.

When the we (write enable) input is 1, the FIFO stores the value from the din port to the next available empty memory location in the FIFO. When the re (read enable) input is 1 the FIFO reads the next value to the dout port from a memory location, in the order the FIFO was written.

When the FIFO is completely empty, the FIFO sets the empty output to 1, and ignores any read attempts.

The count output indicates the number of items in the FIFO queue. You can use the count output to implement buffer management algorithms.

Automatic Scalar Expansion

If the data input is a vector and the reset, read enable, or write enable port is scalar, the tool expands the scalar reset or enable port to the size of the data input vector. The reset and enables can be either vector or scalar.

Constant Propagation

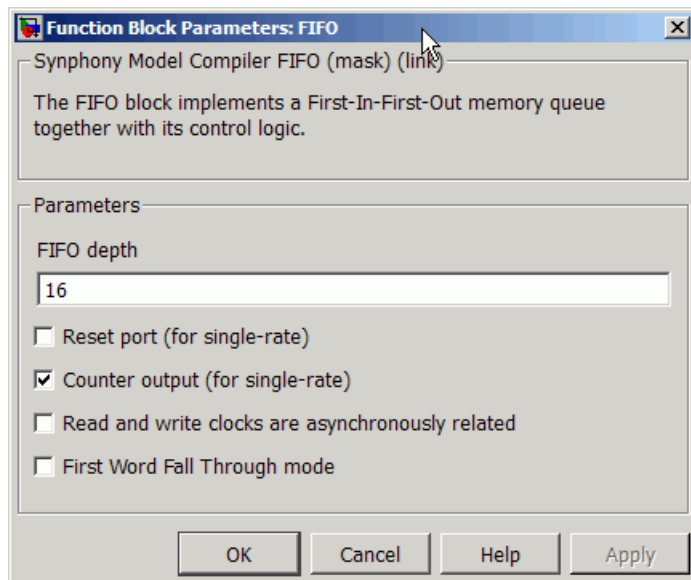
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Icon Annotations

The icon for this block displays the following information:

Note (green)	Indicates the number of words that can be stored in the FIFO.
Latency (red)	The latency for this block is 1.

FIFO Parameters



FIFO Depth

Determines the number of words that can be stored in the FIFO. The width of the words is determined by the driver of the `din` port.

Reset Port

When enabled, it creates a synchronous reset pin on the FIFO. The icon changes to reflect the pin. Enable this option for single-rate FIFOs only.

Counter output

When enabled, it creates a synchronous count port on the FIFO. The icon changes to reflect this. Enable this option for single-rate FIFOs only.

Read and write clocks are asynchronously related

When enabled, it indicates an asynchronous clock relationship between the FIFO read and write clocks. When enabled, the retiming algorithm treats the block as an asynchronous FIFO, and does not retime and move registers across the block.

First Word Fall Through mode

When disabled, the FIFO uses the standard procedure for reading and writing. This means that when the **we** (write enable) pin is high and the FIFO is not full, input data is written to the FIFO. When the **re** (read enable) pin is asserted and the block is not empty, data is read from the FIFO and driven to the output. The output is obtained in the next cycle, so there is a one-cycle read latency.

When enabled, the tool uses First Word Fall Through (FWFT) read mode. FWFT is a read mode that gets rid of the typical one-cycle read latency and drives the first word written to the FIFO to the output without waiting for a read operation. At every read operation, the tool reads the next word from the FIFO, as long as it is available.

See the waveforms in [FIFO Operation in FWFT Mode, on page 224](#) for additional information.

FIFO Timing Waveforms

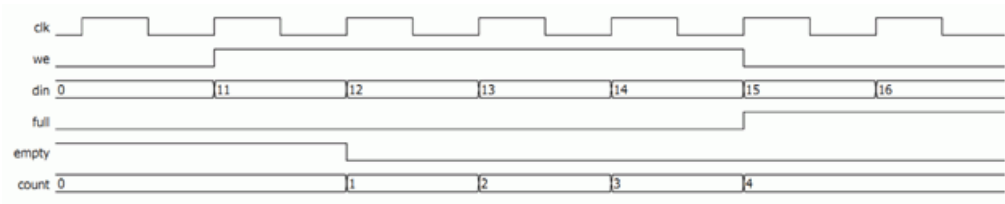
The timing waveforms describe FIFO operation in both normal and FWFT modes.

Normal FIFO Operation

The following timing waveforms represent normal FIFO operation. The FIFO is a single-rate FIFO with a depth of 4, and FWFT mode disabled.

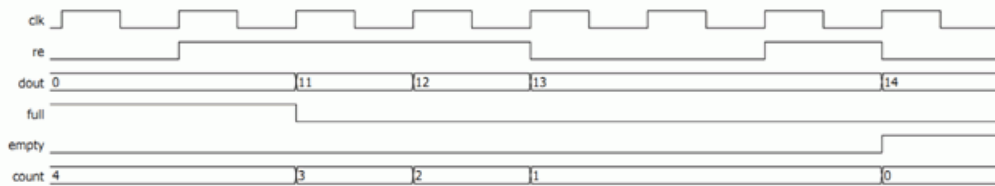
- Write Operation

This waveform shows how data is written to the FIFO according to the **we** signal, and how the operation affects the **full**, **empty**, and **count** signals. This figure assumes that the FIFO is initially empty.



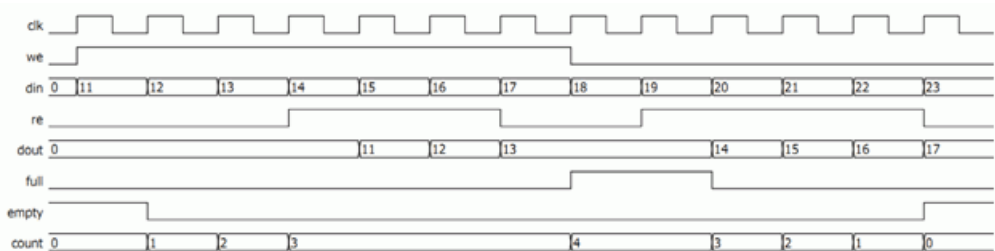
- Read Operation

This waveform assumes that four words (11, 12, 13, and 14) are already written to the FIFO and that it is full. The waveform shows how data is read from the FIFO according to the `re` signal and how the operation affects the full, empty, and count signals.



- Overall Operation

This waveform shows how data is written to and read from the FIFO. It assumes that the FIFO is empty initially.

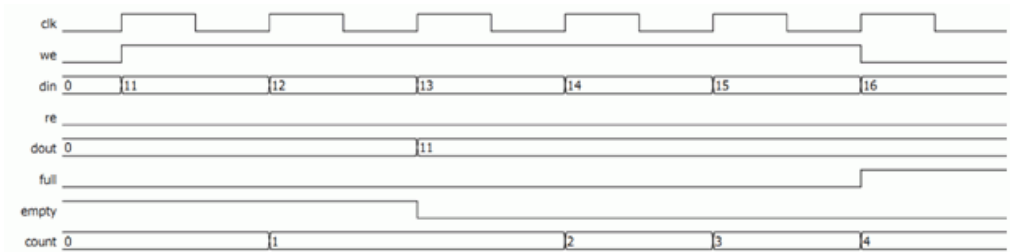


FIFO Operation in FWFT Mode

The following timing waveforms represent FIFO operation when FWFT mode is enabled. The FIFO is a single-rate FIFO with a depth of 4.

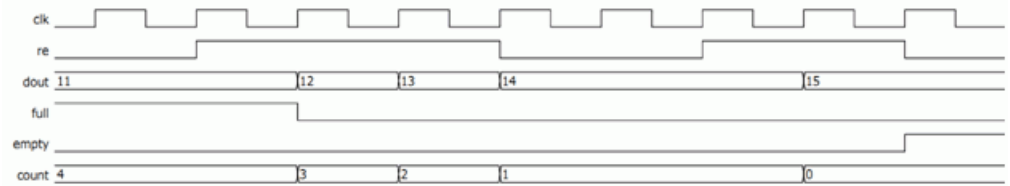
- Write Operation

This waveform shows how data is written to the FIFO according to the `we` signal, and how the operation affects the full, empty, and count signals. This figure assumes that the FIFO is initially empty. Note that in FWFT mode, the FIFO becomes full when five words are written, even though the specified depth is four. The effective depth is always one more than specified in FWFT mode.



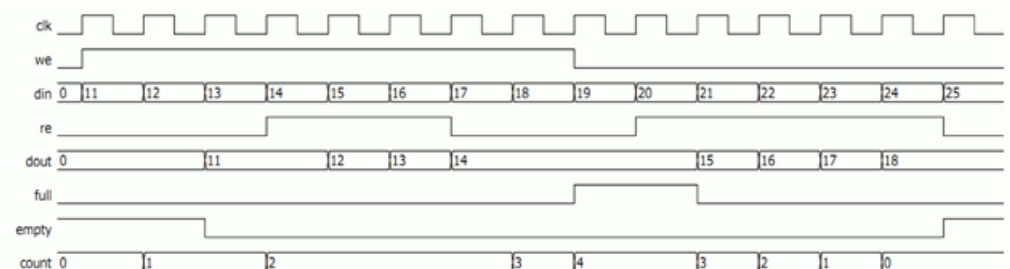
- Read Operation

This waveform assumes that five words (11, 12, 13, 14, and 15) are already written to the FIFO and that it is full. No further words are written to the FIFO. The waveform shows how data is read from the FIFO according to the *re* signal and how the operation affects the *full*, *empty*, and *count* signals.



- Overall Operation

This waveform shows how data is written to and read from the FIFO. It assumes that the FIFO is empty initially.



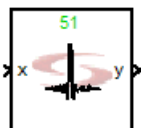
SMC FIR

Implements a finite impulse response (FIR) filter.

Library

Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler FIR block implements a finite impulse response filter, using either a Transposed Form FIR or a Direct Form FIR or a MCM Form FIR for the maximum hardware performance. Typically, this results in better area and timing. You define filter coefficients with either a coefficient vector or a coefficient matrix, according to the application. The coefficients can be extracted from an FDATool instance with the `syn_get_coefs` command.

To implement polyphase FIRs, use the FIR Rate Converter custom block, as described in [Implementing Polyphase FIR Filters, on page 767](#). To implement adaptive or reloadable FIRs, use the RFIR custom block, which is described in [SMC RFIR, on page 448](#).

Automatic Scalar Expansion

If the data input is a vector and the reset or enable port is scalar, the tool expands the scalar reset or enable port to the size of the data input vector. The reset and enable can be either vector or scalar.

FIR Architecture

The tool automatically selects an FIR architecture for each implementation. The optimal configuration meets timing and minimizes area. The tool selects it based on factors like the target device, sample rates, Advanced Timing Mode (ATM), and system-wide optimization settings, like retiming, folding, and multi-channelization. You can also override automatic architecture

selection by using a constraint to specify the architecture you want. The selected micro-architecture for the FIR block is printed in the log file as shown below:

```
@N: <FIR architecture> architecture is selected for the FIR  
    block <block name>
```

The effect of some of the factors that affect architecture selection is described here:

- Advanced Timing Mode

For Microsemi devices, the Symphony Model Compiler tool typically implements MCM form FIRs for baseline implementations.

When ATM is enabled ([SMC SHLSTool, on page 486](#)), the Symphony Model Compiler tool explores various micro-architectures for each implementation before selecting an architecture. It considers the direct form, transposed form and MCM form micro-architectures, explores ripple-carry or carry-save adder as options for the adder tree, and finally chooses a micro-architecture that meets timing and minimizes area.

It does this simultaneously with other operations it would normally do when ATM is not enabled, like exploiting positive and negative symmetric coefficients and optimizations for values of zero, powers of 2, or shifted versions of other coefficients. The tool also considers retiming and pipelining.

If you enable Advanced Timing mode, the tool also uses the area estimates from Synplify Pro synthesis. If you do not want to use the area estimates, disable this behavior with the `areabased_fir_arch_selection_atm` constraint ([areabased_fir_arch_selection_atm Constraint, on page 623](#)).

- Architecture Constraint

You can specify a particular architecture using the `fir_architecture` constraint. See [fir_architecture Constraint, on page 623](#) for the syntax. When you use this constraint, the tool uses the specified architecture.

- Folding

If you enabled the Folding option, the tool always uses the transpose architecture, regardless of what you specified.

Constraints for FIR Architecture

There are two Tcl constraints you can specify for FIR architecture, `fir_architecture` and `areabased_fir_arch_selection_atm`. For details about these constraints, refer to [fir_architecture Constraint, on page 623](#) and [areabased_fir_arch_selection_atm Constraint, on page 623](#).

Icon Annotations

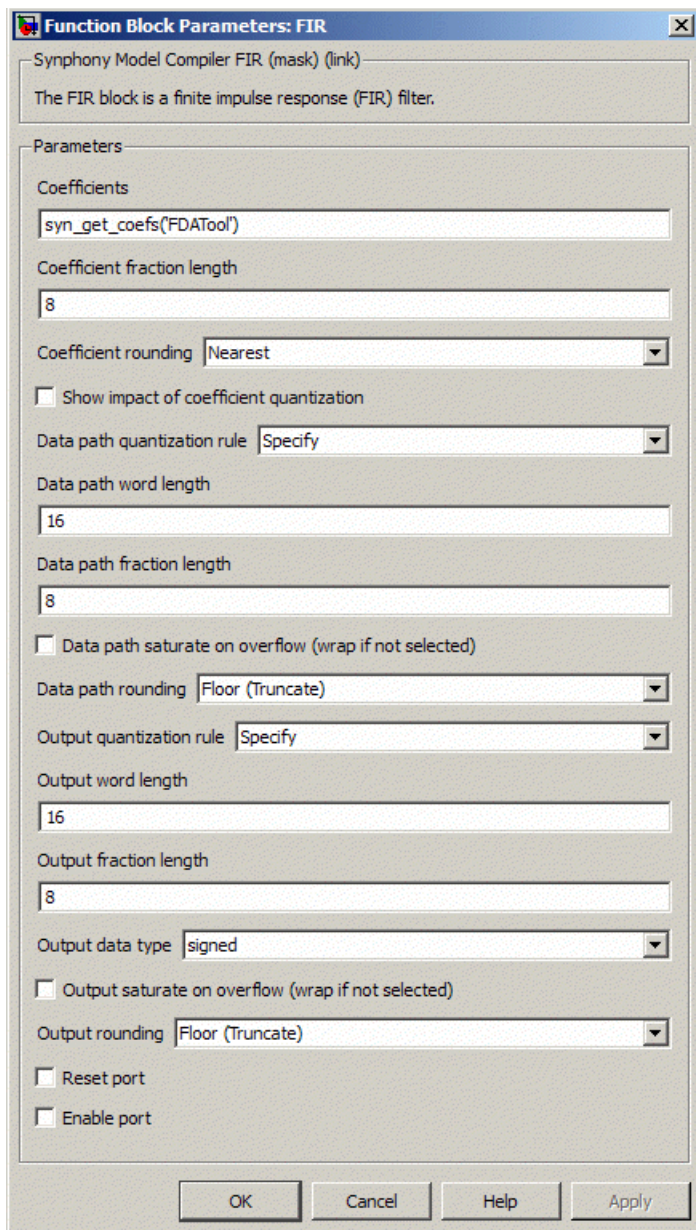
The icon for this block displays the following information:

Top Annotation	The green annotation at the top indicates the number of taps.
Latency Annotation	There is no latency introduced by this block.

Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

FIR Parameters



Function Block Parameters: FIR

Synphony Model Compiler FIR (mask) (link)

The FIR block is a finite impulse response (FIR) filter.

Parameters

Coefficients

`syn_get_coefs('FDATool')`

Coefficient fraction length

8

Coefficient rounding Nearest

☐ Show impact of coefficient quantization

Data path quantization rule Specify

Data path word length

16

Data path fraction length

8

☐ Data path saturate on overflow (wrap if not selected)

Data path rounding Floor (Truncate)

Output quantization rule Specify

Output word length

16

Output fraction length

8

Output data type signed

☐ Output saturate on overflow (wrap if not selected)

Output rounding Floor (Truncate)

☐ Reset port

☐ Enable port

OK Cancel Help Apply

Coefficients

Specifies the filter coefficients. Provide a vector or matrix with the filter coefficients. You can also enter the `syn_get_coefs` function here to extract the coefficients from an `FDATool` instance. See [Defining FIR Filter Coefficients with FDATool, on page 768](#) for information about extracting coefficients, and [syn_get_coefs, on page 604](#) for the function syntax. If you define the coefficients as a matrix or vector, there are various possibilities for the sizes of input and coefficient signals:

- The coefficient array is a row vector (dim:1xN) and the input is a one-dimensional signal. This is regular operation.
- The coefficient array is a row vector (dim:1xN) and the input is an M-dimensional signal. This results in multiple channels, each operating with the same set of coefficients. The same coefficient array vector is applied to each dimension of the M-dimensional input signal.
- The coefficient array is a matrix (dim:MxN) and the input is an M-dimensional signal. This results in multiple channels, each operating with a different set of coefficients. Each row of the parameter matrix is applied to a different signal dimension in the m-dimensional input signal.

Coefficient fraction length

Specifies the fraction length for the coefficient. You can type the value in, or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, and `syn_inp_dt` variables, which are described in [Special Variables, on page 588](#).

The tool selects the coefficient word length automatically. The word length is the smallest length possible for the value, and varies depending on whether the value is signed or unsigned.

Coefficient rounding

Determines how the coefficient value is rounded. See [Underflow Rounding Options, on page 585](#) for details.

Show impact of coefficient quantization

When you enable this option, the spectrum window displays the coefficients with and without quantization, so you can compare them.

Data path quantization rule

Determines the format for data path quantization:

- Automatic sets the data path format to one that uses the maximum of input and output fractions, and the smallest bit width that guarantees no overflow.
- Algorithmic Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the user-defined data type to cast the adder and multiplier outputs. It makes the Data Path Word Length and Data Path Fraction Length options available.

Data path word length

Determines the word length of the data path in bits. It only becomes available when you set Data path quantization rule to Specify. You can type the value in or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

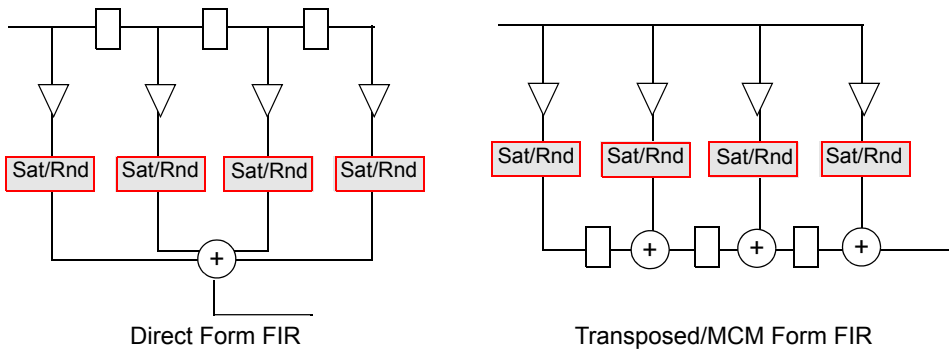
Data path fraction length

Sets the fraction length of the data path in bits. It only becomes available when you set Data path quantization rule to Specify. You can type the value in or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, and `syn_coef_fl` variables, which are described in [Special Variables, on page 588](#).

Data path saturate on overflow

Determines how data path overflow is treated. Enable the option to saturate the overflow, and disable it to wrap the overflow. See [Overflow Saturation Options, on page 585](#) for details.

The following figure shows where the datapath saturation and rounding options are applied:



Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 585](#) for details. This option is not available if Data path quantization rule is set to Algorithmic Full Precision.

Output quantization rule

Determines the format for output quantization:

- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the data type (specified in Output Format) internally to store adder and multiplier outputs. It makes the Output Word Length, Output Fraction Length, Output saturate on overflow, and Output rounding options available.

Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

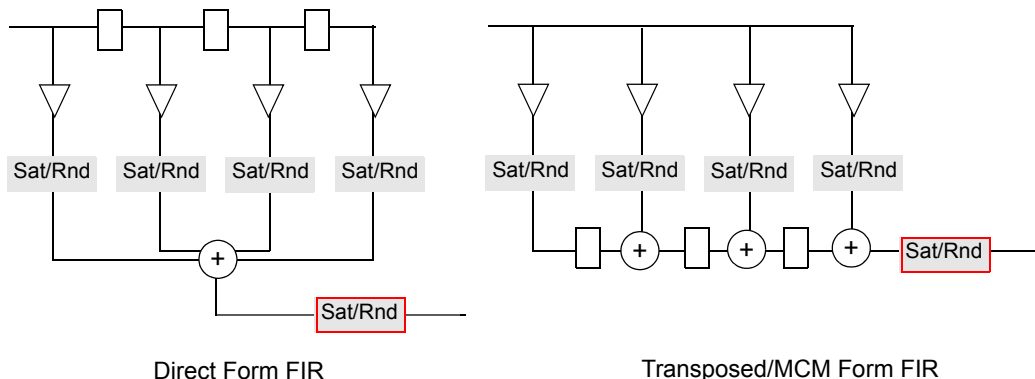
Output word length	Output Word Length, on page 584 You can also specify word length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584 You can also specify fraction length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding

Determine how output overflow and underflow are treated. These options become available when you set Output quantization rule to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output rounding	See Underflow Rounding Options, on page 585 for details about the rounding options available.

The following figure shows where the output saturation and rounding options are applied, after the data path saturation and rounding options:



Reset Port

When enabled, the FIR is implemented with a reset pin. The block icon reflects the change.

Enable Port

When enabled, the FIR is implemented with an enable pin. The block icon reflects the change.

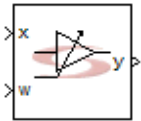
SMC FIR Engine

Implements an FIR filter using coefficients that are input as vectors.

Library

Synphony Model Compiler [Filtering](#)

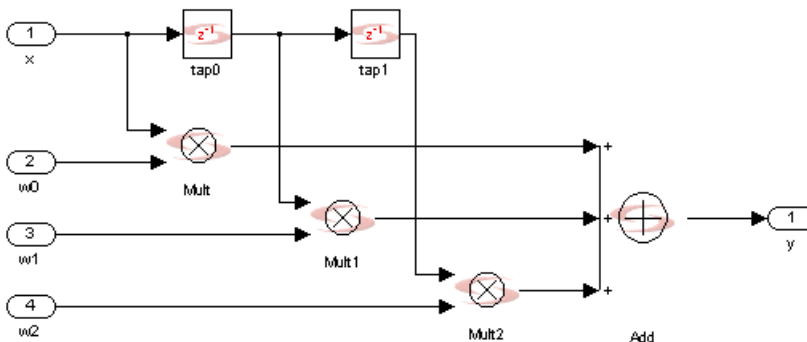
Description



The Synphony Model Compiler FIR Engine block implements a variable-coefficient direct-form finite impulse response filter. The inputs to the block are

- The input signal to be filtered.
- Filter coefficients that are fed to the block packed in a vector. This input is asynchronous or directly fed to the multipliers.
- Optional reset input.
- Optional enable input.
- Optional output for the stored inputs in a vector signal equal to the filter tap length

The following shows a sample diagram for a 3-tap FIR engine.



Folding

In order to use FIR Engine in a folded design, each instance should be able to receive 2 latencies.

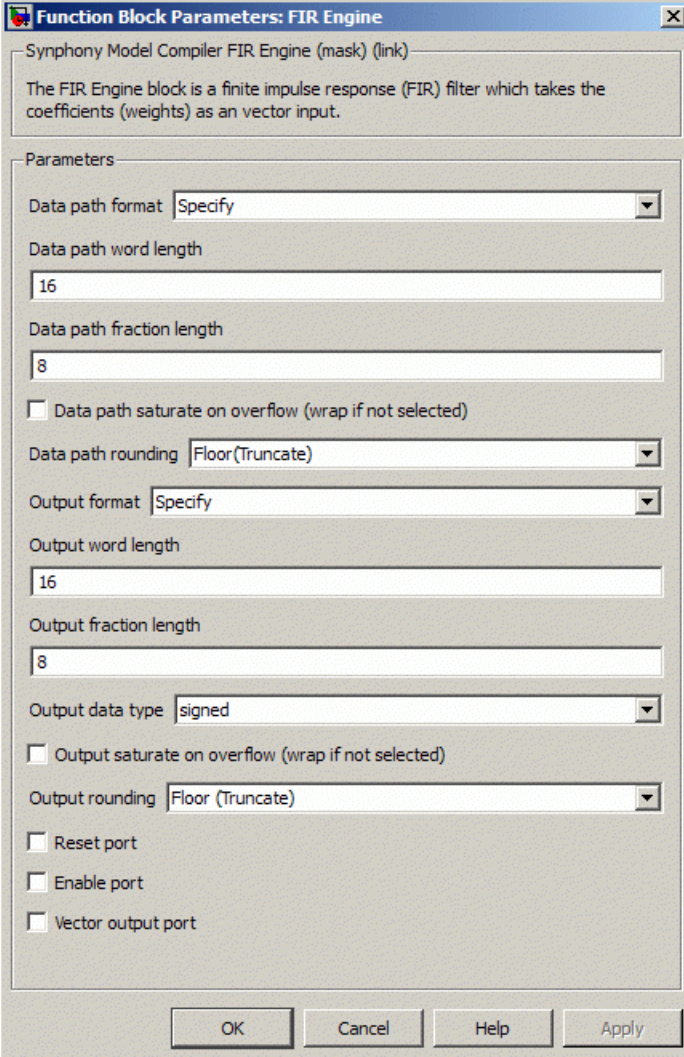
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block does not introduce any latency.

FIR Engine Parameters



The dialog box titled "Function Block Parameters: FIR Engine" contains the following elements:

- A header bar with a close button (X).
- A description box: "Synphony Model Compiler FIR Engine (mask) (link)" and "The FIR Engine block is a finite impulse response (FIR) filter which takes the coefficients (weights) as an vector input."
- A "Parameters" section with the following controls:
 - "Data path format" dropdown menu set to "Specify".
 - "Data path word length" text box containing "16".
 - "Data path fraction length" text box containing "8".
 - Checkbox "Data path saturate on overflow (wrap if not selected)" which is unchecked.
 - "Data path rounding" dropdown menu set to "Floor(Truncate)".
 - "Output format" dropdown menu set to "Specify".
 - "Output word length" text box containing "16".
 - "Output fraction length" text box containing "8".
 - "Output data type" dropdown menu set to "signed".
 - Checkbox "Output saturate on overflow (wrap if not selected)" which is unchecked.
 - "Output rounding" dropdown menu set to "Floor (Truncate)".
 - Checkbox "Reset port" which is unchecked.
 - Checkbox "Enable port" which is unchecked.
 - Checkbox "Vector output port" which is unchecked.
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

Data Path Format

Determines data path format. You can set one of these options:

- Automatic sets the data path format to one that uses the maximum of input and output fractions, and the smallest bit width that guarantees no overflow.

- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the user-defined data type to cast the adder and multiplier outputs. It makes the Data Path Word Length and Data Path Fraction Length options available.

Data Path Word Length

Determines the word length of the data path in bits. It only becomes available when you set Data Path Format to Specify. You can type the value or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, `syn_coef_dt`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

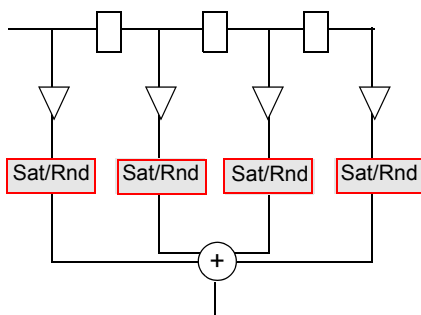
Data path fraction length

Sets the fraction length of the data path in bits. It only becomes available when you set Data path format to Specify. You can type the value or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, `syn_coef_dt`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

Data path saturate on overflow

Determines how data path overflow is treated. Enable the option to saturate the overflow, and disable it to wrap the overflow. See [Overflow Saturation Options, on page 585](#) for details.

The following figure shows where the datapath saturation and rounding options are applied:



Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 585](#) for details. This option is not available if Data path format is set to Full Precision.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

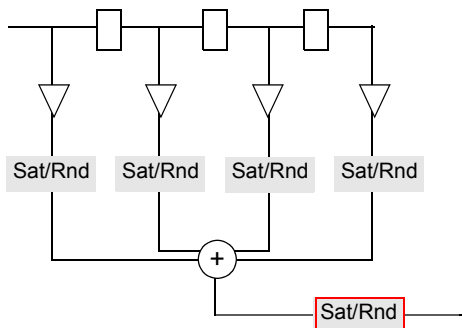
Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584 You can also specify word length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , <code>syn_coef_dt</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584 You can also specify fraction length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , <code>syn_coef_dt</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding

Determine how output overflow and underflow are treated. The options become available when you set Output format to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output rounding	See Underflow Rounding Options, on page 585 for details about the rounding options available.

The following figure shows where the output saturation and rounding options are applied, after the data path saturation and rounding options:



Reset Port

When enabled, the FIR is implemented with a reset input. The block icon reflects the change. The reset pin only affects the internal shift register of the direct-form FIR implementation. If the input $x[0]$ and the coefficients of the filter are non-zero, you might see a non-zero output from the combinatorial path from the input to the filter output.

Enable Port

When enabled, the FIR is implemented with an enable input. The block icon reflects the change. The enable input only affects the internal shift register of the direct-form FIR implementation. The output of the filter may change as the data input and/or coefficients change.

Vector Output Port

When enabled, the FIR Engine represents the taps of the internal shift register as a vector output signal (xv), with the first element of the output vector being equal to the input. The block icon reflects the change.

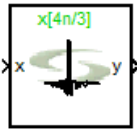
SMC FIR Rate Converter

Implements a polyphase FIR filter by inserting upsamplers and downsamplers.

Library

Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler FIR Rate Converter is a custom block that implements a polyphase filter, by using upsample and downsample blocks with the FIR block to implement interpolators, decimators, and resamplers. See [Primitives and Custom Blocks, on page 800](#) for information about custom blocks.

You can use the FIR block to implement polyphase decimators and interpolators: see [Implementing Polyphase FIR Filters, on page 767](#) for details. This block supports vector input.

Latency

This block has zero latency.

FIR Rate Converter Parameters

Function Block Parameters: FIR Rate Converter

Symphony Model Compiler FIR Rate Converter (mask) (link)

The FIR Rate Converter block applies an upsample in front of an FIR (Interpolator), or a downsample after the FIR (Decimator), or a combination of both (Resampler). The architectural implementation of this function is a polyphase FIR filter.

Parameters

Filter type: **Resampler**

Upsample rate: **4**

Downsample rate: **3**

Coefficients: **syn_get_coefs('FDATool')**

Coefficient fraction length: **8**

Coefficient rounding: **Nearest**

☐ Show impact of coefficient quantization

Data path quantization rule: **Automatic**

Data path rounding: **Floor (Truncate)**

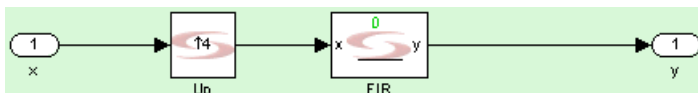
Output quantization rule: **Full Precision**

OK **Cancel** **Help** **Apply**

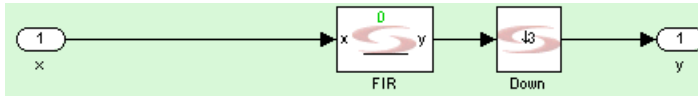
Filter Type

Specifies the type of polyphase filter to be implemented.

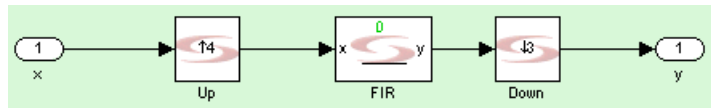
- Interpolator applies an Upsample block before an FIR to implement the polyphase filter.



- Decimator inserts a Downsample block after an FIR to implement the polyphase filter.



- Resampler inserts an Upsample block before and a Downsample block after an FIR to implement the polyphase filter.



Upsample rate

Specifies the value by which the input sample rate is multiplied to get the output sample rate. This option is only available when Filter Type is set to Interpolator or Resampler.

Downsample rate

Specifies the value by which the input sample rate is divided to get the output sample rate. This option is only available when Filter Type is set to Decimator or Resampler.

Coefficients

Specifies the filter coefficients. Provide a vector with the filter coefficients. You can also enter the `syn_get_coefs` command here to extract the coefficients from an FDATool instance. See [Defining FIR Filter Coefficients with FDATool, on page 768](#) for information about extracting coefficients, and [syn_get_coefs, on page 604](#) for details of the function syntax.

Coefficient fraction length

Specifies the fraction length for the coefficient. You can also specify the fraction length in terms of the `syn_inp_wl`, `syn_inp_fl`, and `syn_inp_dt` variables, which are described in [Special Variables, on page 588](#).

Coefficient rounding

Determines how the coefficient value is rounded. See [Underflow Rounding Options, on page 585](#) for details.

Show impact of coefficient quantization

When enabled, the spectrum window displays the coefficients with and without quantization.

Data path quantization rule

Determines how the data path is quantized. You can set one of these options:

- Automatic sets the data path format to one that uses the maximum of input and output fractions, and the smallest bit width that guarantees no overflow.
- Algorithmic Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the user-defined data type to cast adder and multiplier outputs for internal calculations. It makes the Data Path Word Length and Data Path Fraction Length options available.

Data Path Word Length

Determines the word length of the data path in bits. It only becomes available when you set Data path quantization rule to Specify. You can type the value or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

Data path fraction length

Sets the fraction length of the data path in bits. It only becomes available when you set Data path quantization rule to Specify. You can type the value or specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, and `syn_coef_fl` variables, which are described in [Special Variables, on page 588](#).

Data path saturation on overflow

Determines how data path overflow is treated. See [Overflow Saturation Options, on page 585](#) for details. This option is only available when you set Data path quantization rule to Specify.

Data path rounding

Determines how data path underflow is treated. See [Underflow Rounding Options, on page 585](#) for details. This option is only available when you set Data path quantization rule to Automatic or Specify.

Output Parameters

For descriptions of these parameters, see the following:

Output quantization rule	Output Format, on page 583
Output word length	Output Word Length, on page 584 You can also specify word length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584 You can also specify fraction length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output data type	Output Data Type, on page 584

Output saturation on overflow

Determines how output overflow is treated. See [Overflow Saturation Options, on page 585](#) for details. This option is only available when you set Output quantization rule to Specify.

Output rounding

Determines how output underflow is treated. See [Underflow Rounding Options, on page 585](#) for details. This option is only available when you set Output quantization rule to Specify.

Output quantization rule

Determines the format for output quantization:

- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the data type (specified in Output Format) internally to store adder and multiplier outputs. It makes the Output Word Length, Output Fraction Length, Output saturate on overflow, and Output rounding options available.

SMC FIR2

Implements fixed and reloadable coefficient FIR filter blocks, including polyphase filters, multichannel filters, and symmetric coefficient filters.

Library

Synphony Model Compiler [Filtering](#)

Description



The SMC FIR2 block is a custom block that lets you generate highly parametrizable FIR filters. It includes the functionality of the FIR, FIR Engine, and FIR Rate Converter blocks. It also supports multichannel filtering. The architectures are described in [FIR2 Architectures, on page 246](#).

See [FIR2 Functional Overview, on page 246](#) for more information.

FIR2 Functional Overview

For a baseline single phase FIR (all the oversampling factors on the Hardware tab are set to 1), you can add a tap delay line as an optional output. For polyphase decimator FIRs, you can add individual phases as optional output.

You can use the FIR2 block to design a wide range of filters. For details, see [Implementing FIR Filters with the FIR2 Block, on page 760](#).

FIR2 Architectures

The FIR2 block allows you to choose from several different filter architectures to generate optimal RTL for area-efficient, high-performance filters that map efficiently to your selected hardware target. Supported architectures include the following:

- Systolic form, for high speed filters
- Direct, Transpose and Systolic architectures with reloadable coefficients

You specify the kind of architecture to implement with the FIR Architecture option ([FIR architecture, on page 266](#)). You can choose to implement single-phase FIRs, polyphase decimators, or polyphase interpolators, using the Polyphase filter type option ([Polyphase filter type, on page 257](#)).

You can control the level of parallelism used to implement the filter with the oversampling options on the Hardware tab. Use them to control the level of sharing implemented. You can choose between fully parallel, fully serial, and intermediate implementations. You can also use these options to specify the sharing of resources across the channels of a multichannel filter or polyphase filter bank.

Constant Coefficient and Reloadable Coefficient Filters

You can use the FIR2 block as a constant coefficient filter as well as a reloadable filter. The Reloadable Coefficients option specifies the kind of filter to implement ([Reloadable coefficients, on page 250](#)).

- For constant coefficients, the tool infers the tap length of the filter from the number of columns in a specified matrix. You can extract the coefficients from an FDATool instance using the `syn_get_coefs` command.

The tool also infers the number of coefficient sets from which to select the coefficients, based on the number of rows divided by the number of channels specified. If the number of available coefficient sets is more than 1, the tool automatically provides a port to specify the selected coefficient set number. This allows you to select from a fixed set of coefficients with optimized implementation at runtime.

The SMC tool also uses the specified coefficients to infer whether the filter structure should be symmetric, antisymmetric or half band. See [Main Tab, on page 249](#) for descriptions of the parameters to specify constant coefficient filters.

- For reloadable coefficient filters, the coefficients are fed through an input port, and all the parameters must be explicitly specified. See [Reloadable coefficients, on page 250](#) for descriptions of the parameters for specifying constant coefficient filters.

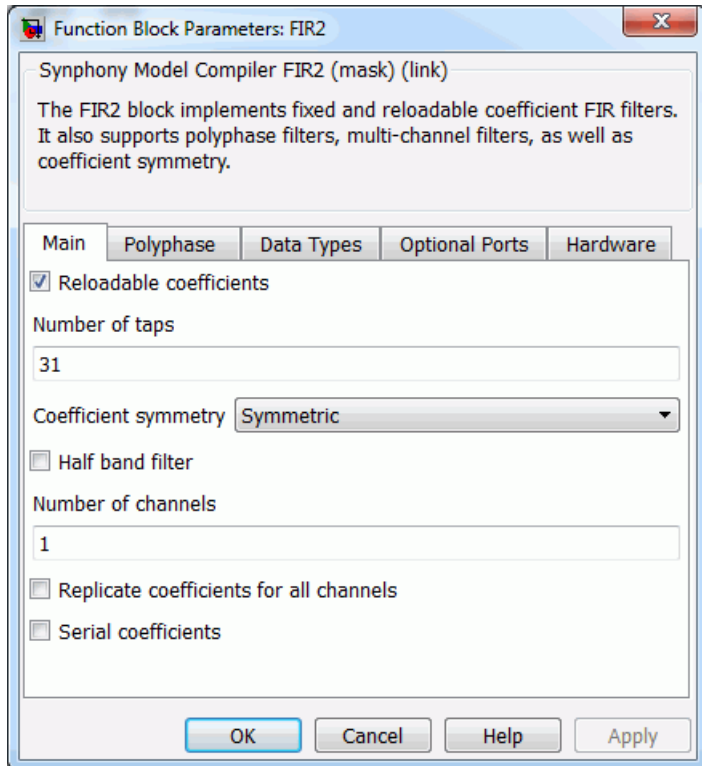
Icon Annotations

The icon for this block displays the following information:

Top Annotation	The green annotations at the top indicate the number of taps, the hardware oversampling factor, the number of channels, and the decimation/interpolation factor for polyphase filters.
Bottom Annotation	<p>The red annotation at the bottom of the icon indicates the latency. Latency values for polyphase decimators only reflect the latency of the summed output. If both summed outputs and individual phase options are selected, the latency annotation only reflects the summed output latency.</p> <p>This latency annotation does not display in the following cases:</p> <ul style="list-style-type: none">• If the <code>load_coef</code> or <code>srnyi</code> port options are enabled• If a polyphase decimator with the individual phase option is specified

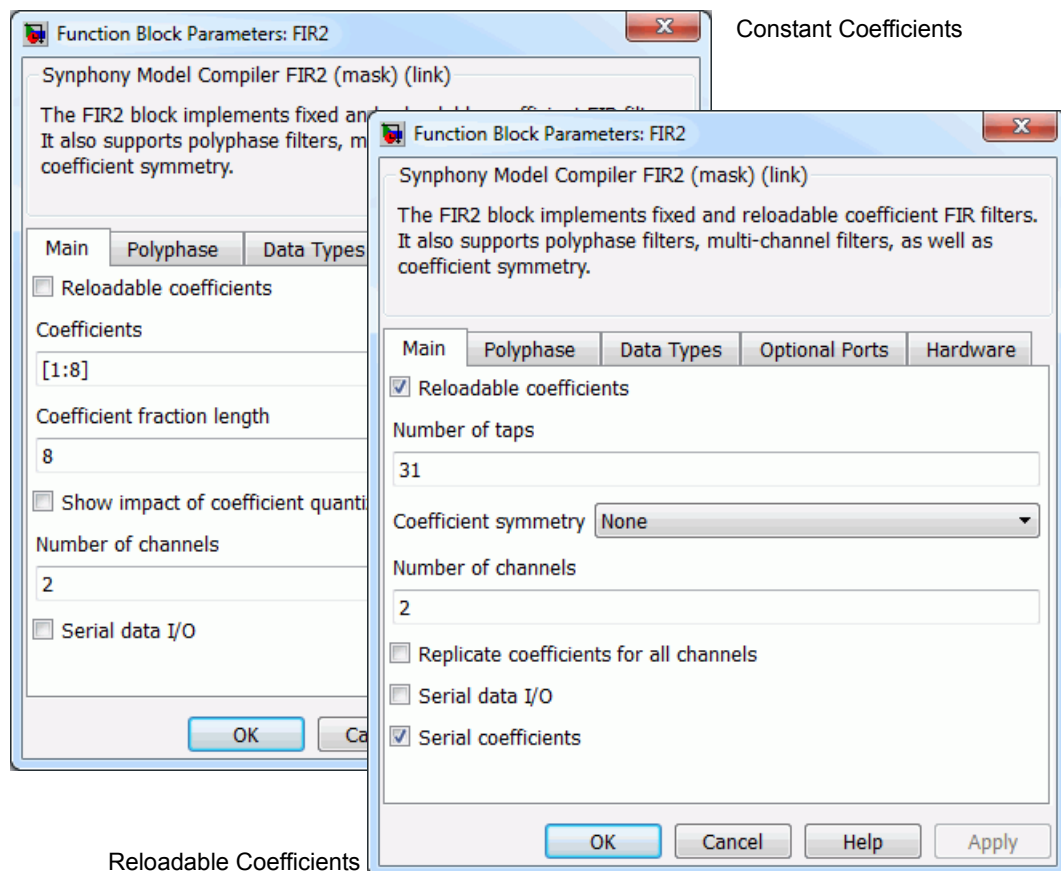
FIR2 Parameters

The interface consists of multiple tabs where you can specify the parameters you need.



Main Tab

This tab sets parameters for coefficients, especially fixed and reloadable coefficients. The available options vary, depending on whether you select constant or reloadable coefficients (Reloadable coefficients option).



Reloadable coefficients

Determines the kind of filter implemented and also determines what other options are available. See [Constant Coefficient and Reloadable Coefficient Filters, on page 247](#) for a description of these filters and how they differ.

- When disabled, the tool implements a filter with fixed coefficients. You enter the coefficients using parameters like Coefficient. This is different from specifying reloadable coefficient filters, where you use a coefficient input port.
- When enabled, the tool implements an FIR with reloadable coefficients. It creates a vector port w for the coefficient input. The

coefficient input is not latched anywhere within the filter, and any updates to the coefficient vector immediately affect the filter output.

Coefficients

Constant coefficient filters only (Reloadable coefficients disabled).

Specifies the filter coefficients for a constant coefficient FIR. You must enter all the coefficients, not just the unique set, because the tool automatically determines if the filter is symmetric, antisymmetric, or half band from the input coefficient matrix you provide. The tool also infers the number of taps from the coefficient matrix.

There are three ways to specify coefficients in this field:

- As a vector.
- As a matrix. The number of matrix rows divided by the number of channels denotes the number of selectable FIR coefficient sets, and the number of columns denotes the number of taps.
- With the `syn_get_coefs` function, which extracts the coefficients from an FDATool instance. See [Defining FIR Filter Coefficients with FDATool, on page 768](#) for information about extracting coefficients, and [syn_get_coefs, on page 604](#) for the function syntax.

You must provide the expanded coefficient set, because the tool infers the filter structure from the input coefficient.

Coefficient fraction length

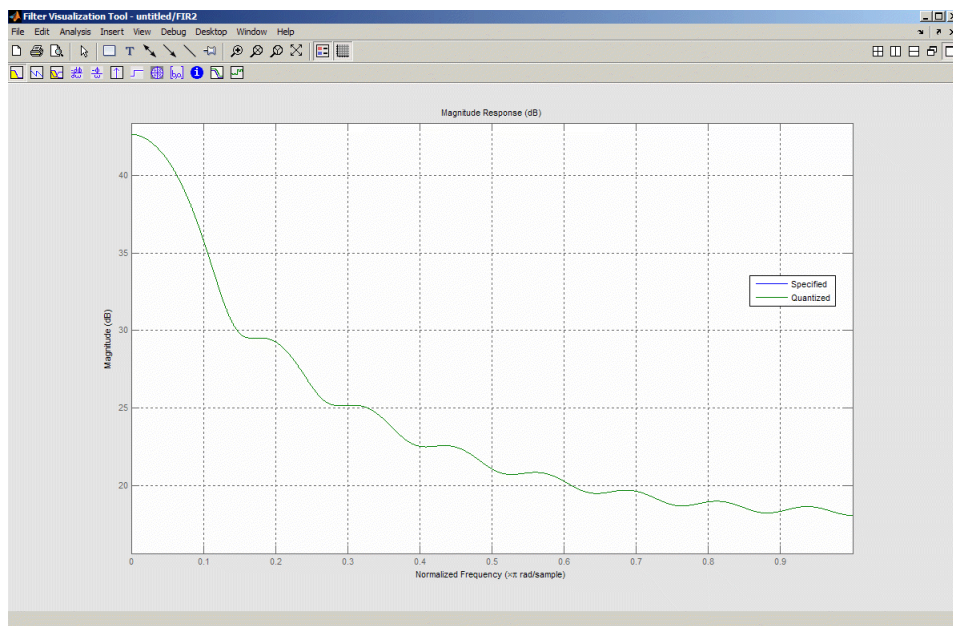
Constant coefficient filters only (Reloadable coefficients disabled).

Specifies the fraction length for the coefficient in a fixed coefficient FIR. The tool selects the coefficient word length automatically. The word length is the smallest length possible for the coefficient with the largest magnitude.

Show impact of coefficient quantization

Constant coefficient filters only (Reloadable coefficients disabled).

When enabled, it opens a magnitude spectrum window. This is only available for a single row of coefficients. The window lets you graphically compare the effect of quantization on the frequency response of the fixed coefficient FIR, as it displays the spectrum both with and without quantization.



Number of channels

Reloadable and constant coefficient filters

Specifies the number of channels in the FIR2 core. If you specify more than one channel, the tool implements a multichannel filter. For multichannel filters, the coefficient input port must be specified as a matrix, where the matrix size is the number of channels x number of coefficients.

For multichannel constant coefficient filters, note the following behavior:

- If you provide a coefficient matrix (Number of channels x Number of taps) with only one row, the tool replicates the coefficient across all channels.
- If you specify fewer rows in the coefficient matrix than the number of channels, the tool pads the remaining channels with all zero coefficients.
- If you specify more rows in the matrix than the number of channels, the tool, after necessary zero padding, infers that multiple sets of coefficients have been provided, with the first Number of channel rows belonging to the first set, and so on. The tool creates an additional port called `coef_sel`, which allows you to specify which of the coefficients sets to use at runtime. The valid input values to `coef_sel` is

0..number of sets -1, where number of sets is computed as $\text{ceil}(\text{coef matrix rows}/\text{number of channels})$.

Serial data I/O

Reloadable and constant coefficient filters

Lets you provide the inputs and obtain the outputs in a time-multiplexed serial fashion, so as to avoid expensive muxes and demuxes at the inputs and outputs. When enabled, the I/O data vector size is equal to Number of channels / Hardware oversampling factor across channels ([Number of channels, on page 252](#) and [Hardware oversampling factor across channels, on page 268](#)).

For reloadable coefficient filters where the Serial coefficient option is enabled, the setting of this option affects the dimensions of the w port. In this case, the tool implements a scalar w port if Serial data I/O is on; if it is off, it implements a vector w port with the vector size equal to the number of channels.

This option is available when Number of channels is more than 1, Polyphase filter type is set to None ([Polyphase filter type, on page 257](#)), folding across channels is feasible, and the number of channels is an integer multiple of Hardware oversampling factor across channels.

Number of taps

Reloadable coefficient filters only (Reloadable coefficients enabled).

Specifies the total number of taps before accounting for symmetry or half band coefficients in the filter.

Coefficient symmetry

Reloadable coefficient filters only (Reloadable coefficients enabled).

You can set three options for coefficient symmetry:

None	Specifies no coefficient symmetry. The input to the w port is the Number of taps x Number of channels.
Symmetric	Specifies symmetric coefficients. The input to the w port must be a matrix of this size: Number of channels x ceil(Number of taps / 2). The vector input w is reversed and appended to the original coefficient vector to create a symmetric coefficient set. If the value in Number of taps is odd, the tool does not replicate the last element of the vector input w.
Antisymmetric	Specifies antisymmetric coefficients. The input to the w port must be a matrix of this size: Number of channels x ceil(Number of taps / 2). The vector input w is negated, reversed, and appended to the original coefficient vector to create an antisymmetric coefficient set. If the Number of taps value is odd, the tool does not negate and replicate the last element of the vector input w.

Half Band Filter

Reloadable coefficient filters only (Reloadable coefficients enabled).

When enabled, implements a standard half band filter. This option is only available if Number of taps is set to an odd value and Coefficient symmetry is set to Symmetric or Antisymmetric. If these conditions are met, the coefficient for the centre tap is always 0.5 and the coefficient for every alternate tap is zero.

If you enable this option, ensure that the w port has a matrix input of this size: Number of channels x ceil(Number of taps - 1) / 4.

Replicate coefficients for all channels

Reloadable coefficient filters only (Reloadable coefficients enabled).

Specifies whether the same coefficient must be applied to all channels. When enabled, the input to the w port is a vector whose size varies:

Option Setting	Input Vector Size for Port w
Coefficient symmetry: None	Number of taps
Coefficient symmetry: Symmetric/Antisymmetric	ceil (Number of taps / 2)
Half band enabled	ceil (Number of taps - 1) / 4

Serial coefficients

Reloadable coefficient filters only (Reloadable coefficients enabled).

Lets you provide the coefficient set for the w input port serially.

- If this option is disabled, you must provide the entire coefficient set as a matrix, which means that all coefficients are available in parallel. If the `load_coef` port is available ([load_coef/srdyo, on page 264](#)), the tool latches all coefficients internally when `ssync` is high. If the `ssync` port is not available, the tool does not latch coefficients internally, and any update to the w port is automatically propagated to the filter.
- When this option is enabled, specify the coefficients for the w port in the same sequence as the taps, with the coefficient for the most recent tap provided first. Note that for serial coefficients, the `load_coef` and `srdyo` ports are always available. The coefficients must be consecutive; that is, `load_coef` must be continuously high for the required number of clock cycles. When this option is enabled, the dimension and the sample rate of the x and w inputs are the same.

You cannot enable this option for certain architectures. See [Limitations to Serial Coefficient, on page 256](#) for a description.

When enabled, the block stores the coefficients as appropriate for subsequent filtering. The effect of various settings is described below:

Option Settings	Description
Serial coefficients on	The block loads new coefficients when <code>load_coef</code> goes high, and continues to load them until all coefficients are received. In subsequent clock cycles, if <code>load_coef</code> goes high again, a new load operation starts. <code>srdyo</code> stays low until the required coefficients are loaded, regardless of the state of <code>srdyi</code> . When targeting Microsemi SmartFusion2 devices, the <code>load_coef</code> signal is required to be a pulse. This is unlike other cases, when a long impulse is needed of length equal to the number of taps. During this period, any input data is ignored.
Serial coefficients on Hardware oversampling across channels =1 Serial data I/O off Replicate coefficients for all channels off	The <code>w</code> input vector size is equal to the number of channels. The coefficients must be provided (<code>load_coef</code> remains high) over Number of taps clock cycles. Note that the Serial coefficients option is not available with the symmetric, antisymmetric, or halfband coefficient options.
Serial coefficients on Hardware oversampling across channels = Number of channels Serial data I/O on	The <code>w</code> input is fully serialized. The vector input size is 1 and the number of clock cycles required to load the coefficients is multiplied by the number of channels. The coefficients are loaded in this sequence: <code><tap1 _ channel 1 ...n, tap2_channel 1...n, ...></code> .

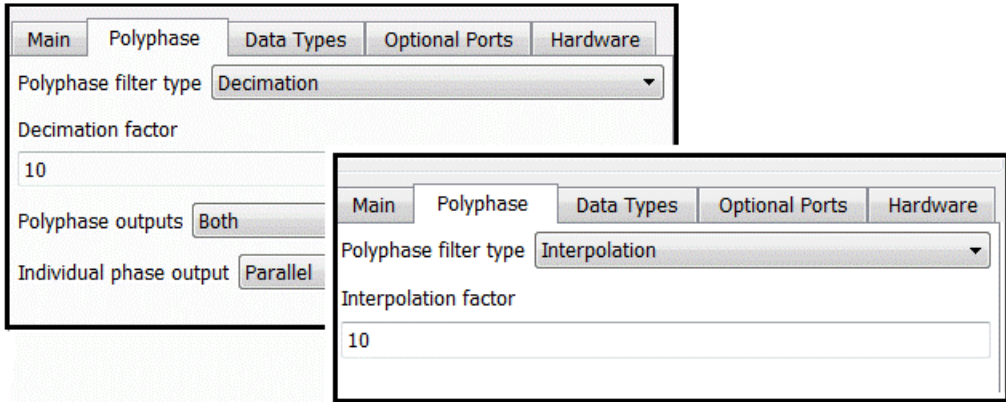
Limitations to Serial Coefficient

The Serial coefficients option ([Serial coefficients, on page 255](#)) and the `load_coef` port option ([load_coef/srdyo, on page 264](#)) are only available for the following FIR architectures:

- Single-phase baseline asymmetric direct FIR
- Single-phase baseline asymmetric systolic FIR
- Single-phase baseline asymmetric transpose FIR targeting Microsemi SmartFusion2 devices
- Polyphase decimation asymmetric systolic FIR, with fold across phases, and folding factor equal to the number of phases
- Single-phase fold-across-channel systolic asymmetric FIR, with folding factor = number of channels
- Single-phase folded asymmetric systolic FIR

Polyphase Tab

Specify options for polyphase filter architecture on this tab.



Polyphase filter type

Determines the filter architecture:

None	Implements a single phase FIR. If this option is selected, no other options are available on this tab.
------	--

Decimation	Implements a polyphase decimator filter.
------------	--

Interpolation	Implements a polyphase interpolator filter.
---------------	---

Decimation factor/Interpolation factor

Specifies the number of phases in the polyphase FIR.

Either the Decimation factor or Interpolation factor option becomes available, to match the Polyphase filter type setting.

If you specify a factor greater than 1, the tool creates a polyphase filter bank with appropriate resource sharing across the filter bank or within each filter, depending on the hardware oversampling factors specified on the Hardware tab.

Polyphase outputs

The outputs vary, depending on the kind of polyphase filter chosen:

– Decimation

When you select a polyphase decimator implementation, the FIR is decomposed into a number of parallel FIRs, or phases. This option specifies how the separate phase outputs are concatenated:

Individual Phases	Each phase of the polyphase FIR is available separately, with all the phase outputs serialized using a commutator.
Summed Output	The output is the scalar sum of phase outputs.
Both	Both individual phases (ph) and summed output (y) are available.

If the Number of channels is set to greater than 1 for a polyphase decimator, the individual phase output is a matrix where the number of rows equals Number of channels, and the number of columns equals the number of phases specified in Decimation factor.

– Interpolation

If you select a polyphase interpolation filter implementation, there is only one type of output available. The tool obtains this output by serializing the output of all the phases in the filter.

Individual phase output

Determines whether individual phase outputs are time-multiplexed. This option is only relevant when the Polyphase outputs is set to Individual Phases or Both for a decimator. You have two choices:

Serial	Each phase of the polyphase FIR is time-multiplexed and available serially at the input sample rate.
Parallel	Phase outputs are available as a vector at the decimated sample rate.

Data Types Tab

Set word length and data type options for various outputs on this tab.

Data path quantization rule	Specify
Pre-adder format	Specify
Pre-adder word length	syn_inp_wl
Pre-adder fraction length	syn_inp_fl
Pre-adder data type	signed
<input type="checkbox"/> Pre-adder saturate on overflow (wrap if not selected)	
Pre-adder round on underflow	Floor (Truncate)
Pre-adder shift bits (negative values indicate left shift)	0
Multiplier format	Specify
Multiplier word length	syn_inp_wl
Multiplier fraction length	syn_inp_fl
Multiplier data type	signed
<input type="checkbox"/> Multiplier saturate on overflow (wrap if not selected)	
Multiplier round on underflow	Floor (Truncate)
Multiplier shift bits (negative values indicate left shift)	0
Sum format	Specify
Sum word length	syn_inp_wl
Sum fraction length	syn_inp_fl
Sum data type	signed
<input type="checkbox"/> Sum saturate on overflow (wrap if not selected)	
Sum round on underflow	Floor (Truncate)
Sum shift bits (negative values indicate left shift)	0
Output format	Specify
Output word length	syn_inp_wl
Output fraction length	syn_inp_fl
Output data type	signed
<input type="checkbox"/> Output saturate on overflow (wrap if not selected)	
Output round on underflow	Floor (Truncate)
Output shift bits (negative values indicate left shift)	

Data path quantization rule

Sets the datapath format to one of the following choices:

- Full Precision
The software does not truncate after adding the partial products.
- Specify
Lets you specify word length and data type for pre-adder output, multiplier output, partial sum outputs, and final outputs.

Pre-adder format

The pre-adder options only apply when the filter coefficient structure is symmetric. In other cases they are ignored. This option sets the adder output format to one of the following choices:

- Full Precision
The software does not truncate after adding inputs corresponding to symmetric taps.
- Specify
Sets the adder output word length, fraction length, and other parameters as specified in the corresponding options:

Option	Description
Pre-adder word length	FIR2 Word length, on page 262
Pre-adder fraction length	FIR2 Fraction Length, on page 262
Pre-adder data type	FIR2 Data Type, on page 262
Pre-adder saturate on overflow	FIR2 Saturate on Overflow, on page 262
Pre-adder shift bits	FIR2 shift bits, on page 263

Multiplier format

Sets the multiplier output format to one of the following choices:

- Full Precision
The software does not truncate after multiplying the tap delay outputs with the filter coefficients.

- Specify
Sets the multiplier output word length, fraction length, and data type as specified in the corresponding options:

Option	Description
Multiplier word length	FIR2 Word length, on page 262
Multiplier fraction length	FIR2 Fraction Length, on page 262
Multiplier data type	FIR2 Data Type, on page 262
Multiplier saturate on overflow	FIR2 Saturate on Overflow, on page 262
Multiplier round on underflow	FIR2 Round on Underflow, on page 263
Multiplier shift bits	FIR2 shift bits, on page 263

Sum format

Sets the partial sum output format to one of the following choices:

- Full Precision
The software does not truncate after adding a pair of partial products.
- Specify
Sets the sum output word length, fraction length, data type, shift bit, overflow, and underflow, as specified in the corresponding options.

Option	Description
Sum word length	FIR2 Word length, on page 262
Sum fraction length	FIR2 Fraction Length, on page 262
Sum data type	FIR2 Data Type, on page 262
Sum saturate on overflow	FIR2 Saturate on Overflow, on page 262
Sum round on underflow	FIR2 Round on Underflow, on page 263
Sum shift bits	FIR2 shift bits, on page 263

Output format

Lets you specify the output format. There are two choices:

- Full Precision
The output is not truncated.

- Specify
The final output is truncated according to the parameters you specify for the following options:

Option	Description
Output word length	FIR2 Word length, on page 262
Output fraction length	FIR2 Fraction Length, on page 262
Output data type	FIR2 Data Type, on page 262
Output saturate on overflow	FIR2 Saturate on Overflow, on page 262
Output round on underflow	FIR2 Round on Underflow, on page 263
Output shift bits	FIR2 shift bits, on page 263

FIR2 Word length

Determines the word length of the data path in bits. For details, see [Output Word Length, on page 584](#).

You can also specify it in terms of the `syn_inp_wl` and `syn_inp_fl` variables. The variables are described in [Special Variables, on page 588](#).

FIR2 Fraction Length

Sets the fraction length of the data path in bits. For details, see [Output Fraction Length, on page 584](#).

You can also specify it in terms of the variables `syn_inp_wl` and `syn_inp_fl` variables. The variables are described in [Special Variables, on page 588](#).

FIR2 Data Type

Sets the data type for the output. See [Output Data Type, on page 584](#) for descriptions of the data types.

FIR2 Saturate on Overflow

Determines how data path overflow is treated. Enable the option to saturate the overflow, and disable it to wrap the overflow. See [Overflow Saturation Options, on page 585](#) for details. The symbol on the block icon reflects the saturation choice you make.

FIR2 Round on Underflow

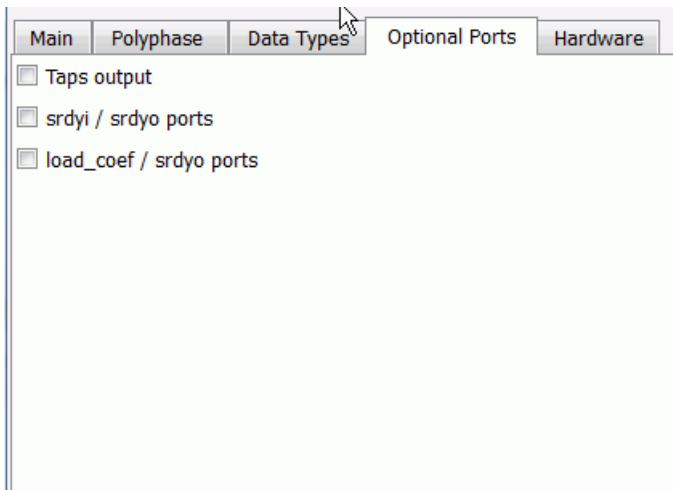
Uses the specified algorithm to round the underflow; see [Underflow Rounding Options, on page 585](#) for descriptions of the algorithms. The symbol on the block icon reflects the rounding choices you make.

FIR2 shift bits

Indicates the number of bits by which the input has to be shifted. For a right shift, the value of the most significant bit (MSB) is shifted in by the number of bits specified. For a left shift, specify a negative number. For left shifts, the zero is shifted in on the least significant bit (LSB) side.

Optional Ports Tab

Set output tap options on this tab. Microsemi SmartFusion2 devices have additional ports, which are described in [Microsemi Output Ports, on page 271](#).



Taps Output

Provides an optional output of the tap delay line, as a vector for single channel filters and as a matrix for multichannel filters. For the matrix output, the number of rows is equal to Number of channels, and number of columns is equal to Number of taps. This option is only available for single-phase baseline filters (Hardware oversampling factors = 1, Polyphase type = None).

srdyi/srdyo ports

When enabled, inserts an `srdyi` (Source ready in) port and a corresponding `srdyo` (Source ready out) port for the block. You can only specify this option for the architectures listed in [Limitations to Serial Coefficient, on page 256](#).

Use the `srdyi` input to denote data inputs that are invalid and should not be processed through the tap delay line. The `srdyo` port indicates which output samples are invalid. It follows `srdyi` with the FIR latency.

You must provide inputs to these ports independently on a per-channel basis. Both `srdyi` and `srdyo` are vectors whose size is equal to the number of channels. To specify the same `srdyi` across all channels, connect a Vector Expand block at the input. If the Serial data I/O option is on, both `srdyi` and `srdyo` are serialized in the same fashion as the data.

When this option is not enabled, the tool processes all data inputs through the tap delay line.

This option is available for the following configurations/architectures:

- Single-phase baseline asymmetric FIR
- Polyphase decimation asymmetric systolic FIR, with fold across phases, and folding factor equal to the number of phases
- Single-phase fold-across-channel systolic symmetric FIR with the limitation of the same `srdyi` signal for all channels
- Single-phase folded asymmetric systolic FIR

load_coef/srdyo

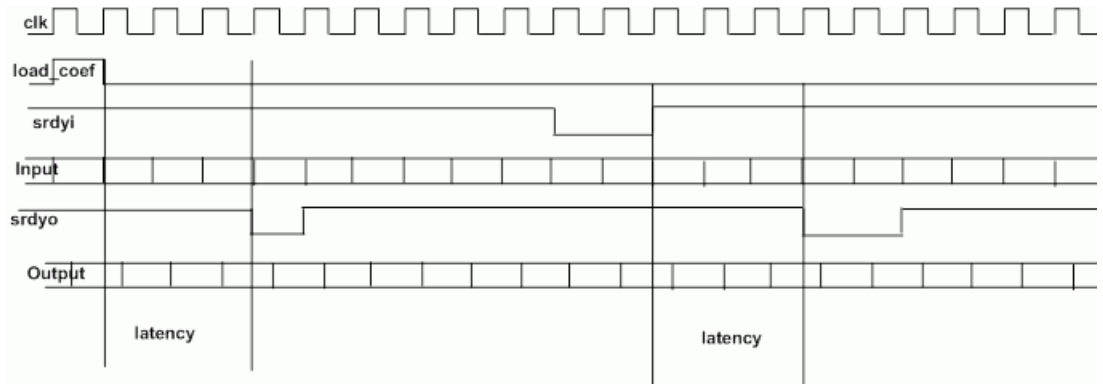
When enabled, inserts a `load_coef` port and a corresponding `srdyo` port for the block. The tool adds extra registers at the `w` input, as described below:

Reloadable Coefficients	If Serial coefficients is off, the tool inserts an extra register at the <code>w</code> input. The block only loads new coefficient values for <code>w</code> when <code>load_coef</code> is high.
Constant Coefficients	If the <code>coef_sel</code> port is available, the tool adds an extra register at the <code>coef_sel</code> input, and only loads new values when <code>load_coef</code> is high.

You can only specify this option for the architectures listed in [Limitations to Serial Coefficient, on page 256](#).

For both fixed and reloadable coefficients, the corresponding `srdyo` output determines which outputs are valid. In the clock cycles where `load_coef` is high, `srdyi` is neglected and `srdyo` goes low for the same number of cycles after FIR2 latency.

The following figure shows an interface timing diagram for the block signals:



If the option is not enabled, the coefficients provided at the `w` port are not registered internally and the coefficient update is immediate and not controlled.

Hardware Tab

Specify options for FIR architecture and hardware oversampling factors on this tab. This tab also includes some device-specific options.

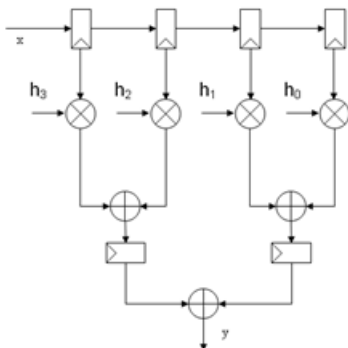
The screenshot shows the 'Hardware' tab of the SMC FIR2 configuration window. It features a tabbed interface with 'Main', 'Polyphase', 'Data Types', 'Optional Ports', and 'Hardware'. The 'FIR architecture' dropdown is set to 'Direct'. Below it are three input fields for hardware oversampling factors, all set to '1': 'Hardware oversampling factor within phases/filters', 'Hardware oversampling factor across phases', and 'Hardware oversampling factor across channels'. At the bottom, there is a checkbox labeled 'Register output' which is currently unchecked.

FIR architecture

Determines the FIR architecture. See [FIR Architecture, on page 226](#) for general information about FIR architectures. For target-specific information about architectures, see [Microsemi SmartFusion2 Options, on page 269](#).

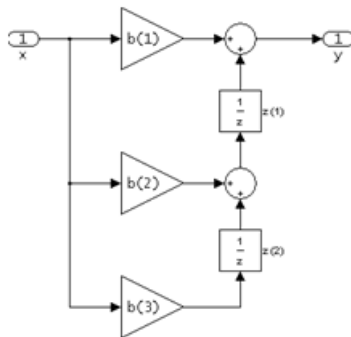
You can set one of these architecture implementations:

- Direct

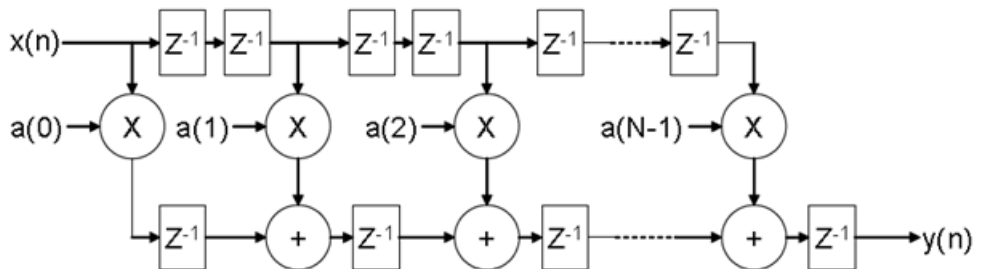


- Transpose

This is only supported for baseline single-phase and polyphase FIRs.



- Systolic



Target Device

The Target Device option on the Hardware tab of the SMC FIR2 block, lets you select a target vendor-specific FPGA for the best performance and optimization. Choose one of the following options:

- Microsemi SmartFusion2

See [Microsemi SmartFusion2 Options, on page 269](#) for information about device-specific options.

-

Default

The generic Default option supports all architectures. If a particular FIR architecture/configuration is not supported for the selected target architecture, the tool uses the hardware corresponding to the Default option.

Hardware oversampling factor within phases/filters

Specifies a hardware oversampling factor within a phase or filter, which helps determine the frequency by which the processing clock can be multiplied with regard to the input sample rate. See [FIR2 Hardware Oversampling Factors, on page 268](#) for additional information.

Hardware oversampling factor across phases

Specifies a hardware oversampling factor across phases, which helps determine the frequency by which the processing clock can be multiplied with regard to the input sample rate. See [FIR2 Hardware Oversampling Factors, on page 268](#) for additional information.

Hardware oversampling factor across channels

Specifies a hardware oversampling factor across channels, which helps determine the frequency by which the processing clock can be multiplied with regard to the input sample rate. See [FIR2 Hardware Oversampling Factors, on page 268](#) for additional information.

Register output

When enabled, implements a register at the output of the FIR filter. Enable this option if RTL generation fails with a message about an infeasible path from the FIR2 block to the next block.

FIR2 Hardware Oversampling Factors

The hardware oversampling factors on the Hardware tab of the FIR2 block define the factor by which the frequency of the processing clock can be multiplied with regard to the input sample rate. This allows the tool to correspondingly reduce the number of multipliers by folded the filters.

By default, folding infers block RAMs for tap memories. If you do not want to implement a folded filter using block RAMs for FPGAs, you must explicitly specify this for your Synplify Pro or Synplify Premier synthesis run using the `syn_ramstyle` constraint.

The following table lists implementations for certain folding factor settings:

Hardware Oversampling Setting Resulting Implementation

All hardware oversampling factors = 1	Baseline or non-folded implementation
Within phases/filters > 1	Single phase filters = Folded within the taps Polyphase decimator/interpolator = Folded within each phase.
Across phases > 1	For polyphase decimators/interpolators, folds across phases.
Across channels >1	For single phase multichannel filters, folds across channels.

If more than one of the factors is greater than 1, the tool only retains the folding factor with the maximum value, and resets the others to 1. If the folding factor does not exactly divide the corresponding dimension, the tool adds zero padding to allow folding.

Microsemi SmartFusion2 Options

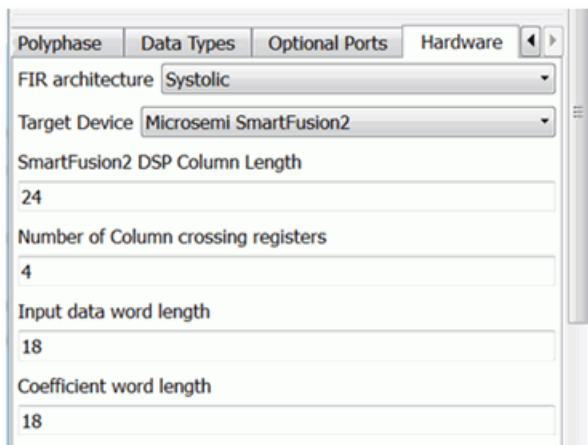
Microsemi SmartFusion2 Architectures

The following configurations or architectures are supported for the Microsemi SmartFusion2 device.

No.	Architecture	Settings
1	Systolic asymmetric baseline	<ul style="list-style-type: none"> • Coefficient symmetry - None (or non-symmetric constant coefficients) • Polyphase filter type - None • FIR architecture - Systolic
2	Systolic symmetric baseline	<ul style="list-style-type: none"> • Coefficient symmetry - Symmetric or Antisymmetric (or symmetric/antisymmetric constant coefficients) • Polyphase filter type - None • FIR architecture - Systolic
3	Transpose asymmetric baseline	<ul style="list-style-type: none"> • Coefficient symmetry - None (or non-symmetric constant coefficients) • Polyphase filter type - None • FIR architecture - Transpose

Microsemi SmartFusion2 Hardware Options

The Hardware tab displays the following additional options for this target:



SmartFusion2 DSP Column Length

Specifies the number of Mathblocks (MACCs) in a row. The embedded MACCs for SmartFusion2 devices are arranged in rows across the fabric. The number of rows and column length for each row (number of MACC per row) varies depending on the device. See the *SmartFusion2 SoC FPGA Fabric User's Guide* for the Mathblock (MACC) resources available for a particular device.

Number of column crossing registers

Specifies the number of registers inserted between two columns.

Input data word length

Specifies the word length of the input data. Internally, the FIR engine is limited to a word length of 44. If you select word lengths and number of taps that exceed the internal bit-growth upper limit of 44-bits, then FIR2 issues a warning and the accuracy of results may not be desired.

Coefficient word length

Specifies the word length of the input coefficients and internally is used to calculate the bit-growth when Reloadable coefficients is selected. This option is not available for Constant Coefficients, since it is automatically calculated based on the coefficient entered and fraction length.

Microsemi Output Ports

The tool includes the following additional output ports for Microsemi SmartFusion2:

Krdyo [Output]

When you select Serial Coefficients for the Microsemi SmartFusion2 architecture, the krdyo port becomes available. This port determines whether input data is ignored during the period that serial coefficients are loaded into the FIR engine. Valid input data is indicated by one (1), where valid output data is generated for that input. Invalid input data is indicated by zero (0), when the input data is ignored by the FIR engine.

Transients [Output]

When you select Serial Coefficients for the Microsemi SmartFusion2 architecture, the transients port becomes available. This port determines whether invalid output data is generated during the period that serial coefficients are loaded into the FIR engine. A transient (invalid output data) is indicated by one (1), when the output data is ignored by the FIR engine.

FIR2 Limitations and Workarounds

The FIR2 functionality has some limitations. They are described below, along with suggestions to work around the problem.

- [srdyi/srdyo Ports, on page 271](#)
- [Serial Coefficients, on page 272](#)
- [Other Coefficient Issues, on page 273](#)

srdyi/srdyo Ports

These are some limitations to specifying srdyi/srdyo ports for the FIR2 block:

- The tool does not support srdyi/srdyo ports for systolic self-folded FIRs when the Hardware oversampling factor within filters < Number of taps / Hardware oversampling factor within filters.

Workaround:

Decompose the impulse response in sections of size fofa*fofa. Use the decomposed sections for parallel FIR2 instances, with a register of delay

fofa*fofa and enabled by srdyi between two consecutive instances. Add the output of the FIR2 instances.

- The tool only supports srdyi/srdyo for summed output in polyphase FIRs with hardware oversampling factor across phases = number of phases. You cannot use these ports if you specify polyphase output to be Individual Phase or Both.

Workaround:

Split the phases outside the FIR2 instance and then use the multi-channel FIR with hardware oversampling factor = number of channels = number of phases in the original FIR description.

Serial Coefficients

The following describe some issues with serial coefficients:

- For serial coefficients in systolic self-folded FIRs, from the second coefficient load event onwards, when load_coef goes high again (to signal the start of a new load), the sample that comes in the clock cycle immediately prior to load_coef going high is filtered with the new coefficients, instead of the old coefficients.

Workaround:

Design the overall system so that it can tolerate the loss of one sample just prior to loading a new set of coefficients.

- For serial coefficients in polyphase FIRs with oversampling factor across phases = number of phases, from the second coefficient load event onwards, when load_coef goes high again (to signal the start of a new load), the sample that comes in the clock cycle immediately prior to load_coef going high is filtered with the new coefficients, instead of the old coefficients.

Workaround:

Design the overall system so that it can tolerate the loss of one sample just prior to loading a new set of coefficients.

- For serial coefficients in MAC FIRs, from the second coefficient load event onwards, when load_coef goes high again (to signal the start of a new load), the sample that comes in the clock cycle immediately prior to load_coef going high is filtered with the new coefficients, instead of the old coefficients.

Workaround:

Design the overall system so that it can tolerate the loss of one sample just prior to loading a new set of coefficients.

- The SMC tool does not support serial coefficients are not supported for systolic self-folded FIRs if the number of taps is not an integer multiple of the hardware oversampling factor within phases/filters.

Workaround:

Increase the number of taps, to ensure that this relation is achieved. Then feed zeros serially for the corresponding coefficients.

- The tool does not support serial coefficients for polyphase FIRs if the number of taps is not an integer multiple of the number of phases.

Workaround:

Increase the number of taps to ensure that this relation is achieved. Then feed zeros serially for the corresponding coefficients.

- For polyphase decimation FIRs with hardware oversampling factor across phases = number of phases, and the serial coefficient option enabled, the number of valid input samples between the de-assertion of `load_coef` (indicating the end of a serial coefficient load operation) and its re-assertion (indicating the start of the next serial coefficient load operation) must be an integer multiple of the number of phases. If not, the filter does not perform as expected.

Other Coefficient Issues

The following describe miscellaneous issues:

- For all architectures other than baseline direct FIRs, if `coef_sel` is changed (for selectable coefficient set) or a new set of coefficients are applied for parallel reloadable coefficients, either with or without `load_coef` input, the next few samples will be invalid for a length equal to taps/fofa within filters or phases. However `srdyo` does NOT go low to indicate that those samples are invalid.

Workaround:

Account for the invalid samples outside the FIR2 instance. The number of clock cycles between the changing of `coef_sel` or the input coefficient set and the output is a function of the processing pipeline latency and is not same as the annotated latency of the FIR. Derived this value through simulation.

- The tool supports different `srnyi` sequences in different channels for multichannel filters with fold across channels. However the `load_coef` input must be the same for all the channels even though its port dimension is the same as `srnyi`.

Workaround:

Generate a scalar sequence for `load_coef` and pass it through a Vector Expand block before connecting it to the `load_coef` input.

SMC Flow Control Buffer

Provides forward and backward flow control for bursty data streams.

Library

Synphony Model Compiler [Memories](#)

Description

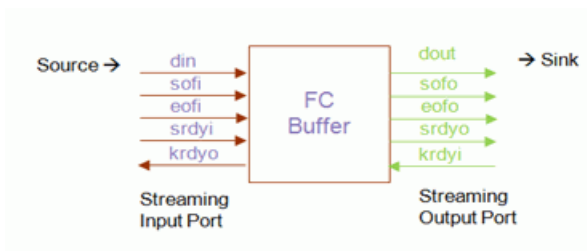


The Synphony Model Compiler Flow Control Buffer is a custom block that provides forward and backward flow control to manage multiple streams of data and account for worst-case stalls.

- When forward control is enabled, the block uses the `srdyi` and `srdyo` input and output signals. It stores the data in the buffer based on input signal `srdyi`. Data in the buffer is always driven to the output, whenever it is available. The `srdyo` indicates availability of the data to be read by downstream logic.
- When backward flow control is enabled, the block uses the `krdyi` and `krdyo` signals. The block stores the input data as long as the threshold is not reached. It reads the data from the buffer according to the `krdyi` input signal. The `krdyo` output signal indicates the availability of space in the buffer so upstream logic can write to it.

Block Signals

The following figure and table describe the optional signals you can define for this block to control data flow. All control signals are synchronous one-bit scalar signals. The `srdyi` and `srdyo` signals control forward flow and the `krdyi` and `krdyo` signals control backward flow. The `sofi/sofo` and `eofi/eof` signals are frame delimiter signals that go along with the data in the buffer and are independent of flow control.



srdyi	Source ready in Input signal that enables the buffer write operation. If it is not defined, the buffer write is always enabled. If the buffer has no space, input data is not stored in the buffer.
srdyo	Source ready out Output signal that is asserted only when the buffer has data to be read.
krdyi	Sink ready in Input signal that triggers the reading of data from the buffer.
krdyo	Sink ready out Output signal that is asserted when the buffer is not full.
sofi	Start of frame in Input signal that should be asserted for the first word of each frame.
sofo	Start of frame out Output signal that is asserted when a word with sofi asserted is read from the buffer. It is up to you to generate the appropriate sofi signal.
eofi	End of frame in Input signal that should be asserted for the last word of each frame.
eoyo	End of frame out Output signal that is asserted when a word with eofi asserted is read from the buffer. It is your responsibility to generate the appropriate eofi signal.

Flow Control Buffer Operating Modes

The operating mode for the Flow Control Buffer block is based on the input and output dimensions. The mode is automatically inferred from these dimensions. Currently, the tool supports the following operating modes:

- **Scalar - Scalar Operating Mode**

This mode of operation is based on scalar input (din) and output (dout). For this mode, the din sample rate must be the same as srnyi. The dout sample rate is determined by the sample rate of the krnyi input signal.

- **Scalar - Vector Operating Mode**

This mode of operation is based on scalar input (din) and vector output (dout). The input is stacked into a vector, according to the specified output vector width. The output sample rate is the input rate divided by the vector width.

The din, srnyi, and krnyo signals come in the higher sample rate domain and the dout, srnyo and krnyi signals come in the lower sample rate domain. Make sure that the din sample rate is the same as that of srnyi, and that the krnyi sample rate is equal to the srnyi sample rate divided by the output vector width.

The soft/sofo and eofi/eofy signals are not available for this mode.

- **Scalar - Matrix Operating Mode**

This mode of operation is based on scalar input (din) and matrix output (dout). Based on the output dimensions, the input is stacked into a vector and then converted to a matrix. The matrix is given out on dout when srnyo is asserted. The output sample rate is equal to the input rate divided by the number of elements desired in the output.

The din, srnyi, and krnyo signals come in the higher sample rate domain and the dout, srnyo and krnyi signals come in the lower sample rate domain. Make sure that the din sample rate is the same as that of srnyi, and that the krnyi sample rate is equal to the srnyi sample rate divided by the number of elements of output.

The soft/sofo and eofi/eofy signals are not available for this mode.

- **Vector - Vector Operating Mode**

This mode is based on vector data input and output. The input and output vectors must be the same width. A separate buffer is instantiated

for each element of the input vector. Each buffer is controlled by the same control signals.

For this mode, the `din` sample rate must be the same as `srdyi`. The `dout` sample rate is determined by the `krdyi` sample rate.

- Vector - Scalar Operating Mode

This mode of operation is based on vector input (`din`) and scalar output (`dout`). In this mode of operation, input vector is scalarized. This scalar is given out on `dout` with `srdyo` asserted. The output sample rate is equal to the input sample rate multiplied by the number of elements in the input.

The `din`, `srdyi` and `krdyo` signals come in the lower sample rate domain and the `dout`, `srdyo` and `krdyi` signals come in the higher sample rate domain. Make sure that the `din` sample rate is the same as that of `srdyi`, and that the `krdyi` sample rate is equal to the `srdyi` sample rate multiplied by the number of elements in the input.

The `sofi/sofo` and `eofi/eofa` are not available for this mode.

- Matrix - Matrix Operating Mode

This mode is based on matrix input and output. The input and output matrices must have the same dimensions. A separate buffer is instantiated for each element of the input matrix. Each buffer is controlled by the same control signals.

For this mode, the `din` sample rate must be the same as that of `srdyi`. The `dout` sample rate is determined by the sample rate of the `krdyi` input signal.

- Matrix - Scalar Operating Mode

This mode of operation is based on matrix input (`din`) and scalar output (`dout`). In this mode, the input vector is scalarized in a row-wise manner. This scalar is given out on `dout` with `srdyo` asserted. The output sample rate is equal to the input sample rate multiplied by the number of elements in the input.

The `din`, `srdyi` and `krdyo` signals come in the lower sample rate domain and the `dout`, `srdyo` and `krdyi` signals come in the higher sample rate domain. Make sure that the `din` sample rate is the same as that of `srdyi`, and that the `krdyi` sample rate is equal to the `srdyi` sample rate multiplied by the number of elements in the input.

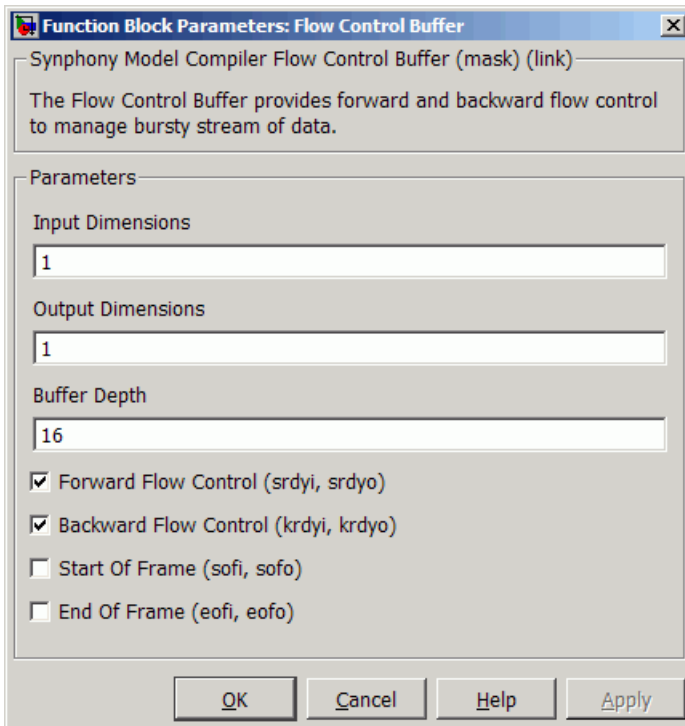
The `sofi/sofo` and `eofi/eofa` are not available for this mode.

Icon Annotation

Top annotation

Buffer depth

Flow Control Buffer Parameters



Input Dimensions

Specifies the input dimensions. You can specify the dimensions as a single integer or as a pair of integer values:

Input Dimensions	Input
Scalar input (1)	Scalar input
Input vector width (integer)	Vector input
Matrix [m,n], where m and n are the number of rows and columns, respectively.	Matrix input

The tool uses the input and output dimensions to determine the operating mode. See [Flow Control Buffer Operating Modes, on page 277](#).

Output Dimensions

Specifies the output dimensions, either as a single integer or as a pair of integer values.

Output Dimensions	Output
Scalar output (1)	Scalar output
Output vector width	Vector output
Matrix [m,n], where m and n are the number of rows and columns, respectively.	Matrix output
-1	Inherit dimensions from input

The software uses the output dimensions along with the input dimensions to determine the operating mode. See [Flow Control Buffer Operating Modes, on page 277](#) for details.

Buffer Depth

Specifies the number of samples that can be stored in the buffer. It represents the extra buffering space required. Indirectly, the buffer depth determines the depth of the FIFO block instantiated inside the custom Flow Control Buffer block.

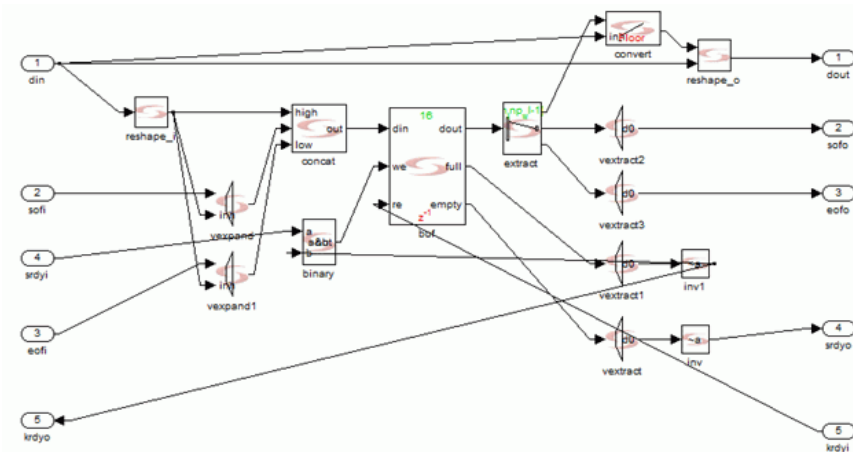
If only one type of flow control is enabled, buffering is not necessary. These scenarios include the following:

- When forward flow control is enabled, the upstream block can send data to the buffer with its corresponding `srdyi` signal and then passes this data to the downstream block using its corresponding `srdo` signal. Nothing gets stored in the buffer, so the downstream block is always available and ready to accept this data. This is true even when scalar to vector conversions occur. The buffer only needs to send the aggregate number of scalars to vectors with the correct `srdo` signal.
- When backward flow control is enabled, the downstream block can notify the buffer that it is not ready to accept data with its `krdyi` signal and then passes this signal to the upstream block with its `krdo` signal. The upstream block must not send data to the downstream block when it is not ready and be available to send data when the downstream block is ready to accept it.

Buffering only occurs when both types of flow controls are enabled.

- If both types of flow control are enabled, the forward flow control signals are used to write data into the buffer and communicate the availability of data to downstream logic, and the backward flow control signals are used to read data from the buffer and indicate when the buffer is available for more data to be written to it.

The custom block implementation varies, according to the block parameters you set. The following figure shows the implementation with all the flow control signals enabled:



The Flow Control Buffer block can be used to combine serial input or split the input, as with the Serial-to-Parallel and Parallel-to-Serial blocks. The difference is that the Serial-to-Parallel and Parallel-to-Serial blocks work with bit concatenation and bit extraction, and are used for bit level serialization and parallelization, while the Flow Control Buffer block is used to serialize and parallelize any data samples with control.

Forward Flow Control (srdyi, srdyo)

When enabled, specifies forward flow control for the block. The tool adds the srdyi and srdyo signals to the block, for input and output respectively. When the srdyi signal is asserted, input data is written to the FIFO. The srdyo signal is asserted when data is available to be read.

If the option is disabled, the tool does not add the srdyi and srdyo signals to the block, and buffer write is always enabled.

You must enable at least one type of flow control.

Backward Flow Control (krdyi, krdyo)

When enabled, specifies backward flow control for the block. The tool adds the `krdyi` and `krdyo` signals to the block, for input and output respectively. When the `krdyi` signal is asserted, data is read from the FIFO. The output `krdyo` signal is deasserted when there is no more space in the buffer to write data.

If the option is disabled, the tool does not add the `krdyi` and `krdyo` signals to the block, and the buffer is always enabled.

You must enable at least one type of flow control.

Start of Frame (sofi, sofo)

When this option is enabled, the tool adds the `sofi` and `sofo` start of frame signals to the block. These signals are optional signals. When it is disabled, the start-of-frame signals are not added to the block.

The `sofi` signal does not control the state of the buffer. The output `sofo` signal is asserted when data with `sofi` asserted is read from the buffer.

End of Frame (eofi, efof)

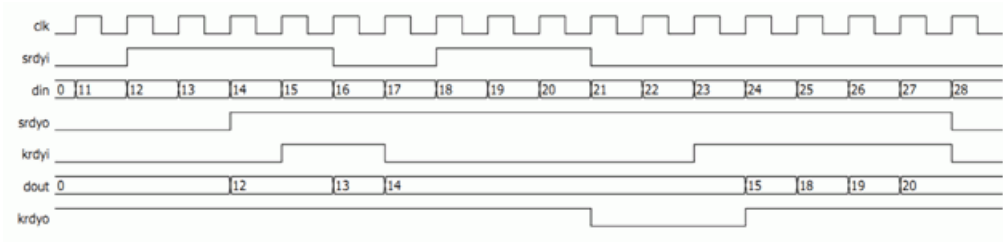
When enabled, specifies the end of the frame and adds the `eofi` and `efof` signals to the block. These signals are optional signals. The `eofi` signal does not control the state of the buffer. The output `efof` signal is asserted when data with `eofi` asserted is read from the buffer.

Flow Control Buffer Waveforms

The following timing waveforms illustrate different modes of operation and flow controls with the Flow Control Buffer block.

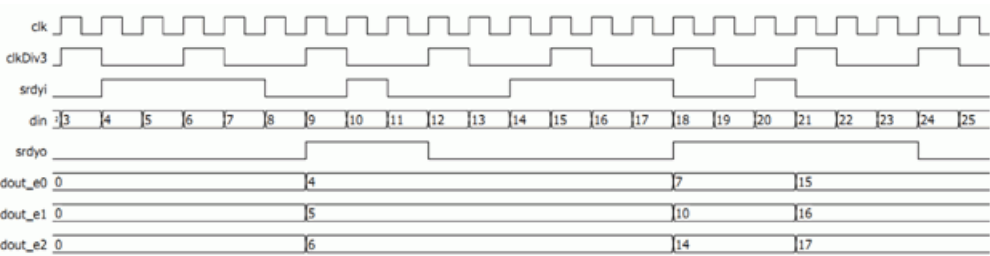
Scalar - Scalar Mode with Forward and Backward Flow Control

Input dimensions	1
Output dimensions	1
Buffer depth	4
Forward flow control	On
Backward flow control	On



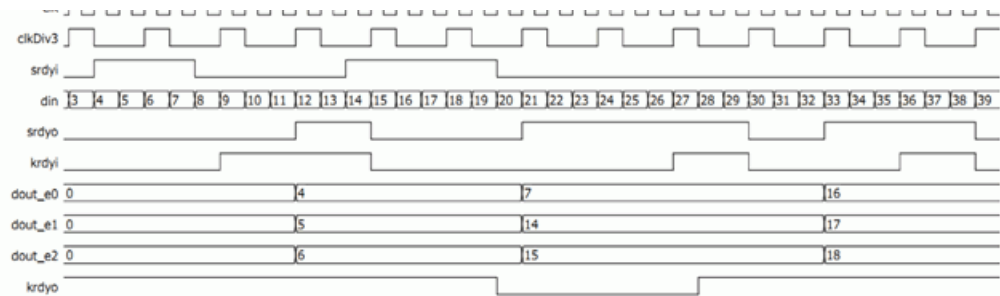
Scalar - Vector Mode with Forward Flow Control

Input dimensions	1
Output dimensions	3
Forward flow control	On
Backward flow control	Off



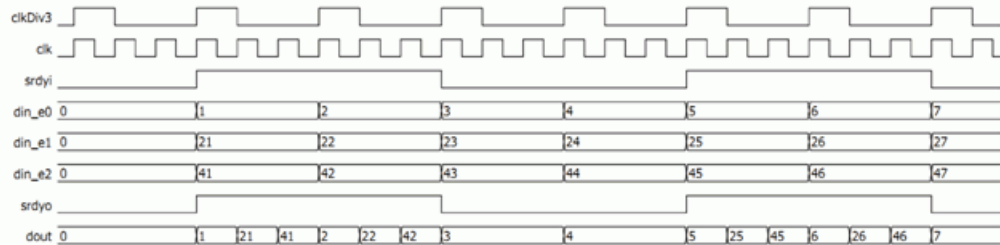
Scalar - Vector Mode with Forward and Backward Flow Control

Input dimensions	1
Output dimensions	3
Buffer depth	4
Forward flow control	On
Backward flow control	On



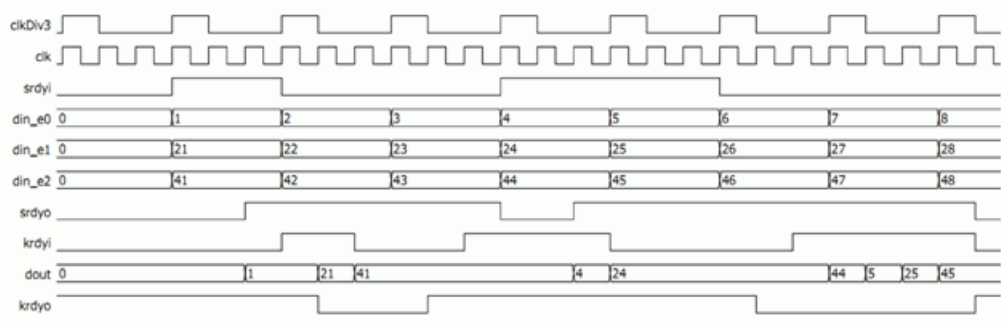
Vector - Scalar Mode with Forward Flow Control

Input dimensions	3
Output dimensions	1
Forward flow control	On
Backward flow control	Off



Vector - Scalar Mode with Forward and Backward Flow Control

Input dimensions	3
Output dimensions	1
Buffer depth	4
Forward flow control	On
Backward flow control	On



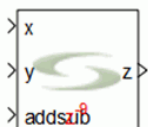
SMC FP Add

Adds or subtracts two floating point values.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



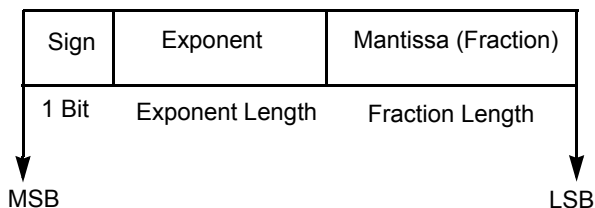
This block adds or subtracts two floating point values. The representations are the same for the input and output floating points. You can also use this block to perform dynamic addition or subtraction with the addsub option.

Floating Point Input and Output Formats

For all SMC floating point computation blocks, the format of the floating point input or output is always an unsigned integer of length N bit, where

$$N = 1 + \text{Length of the Exponent Field} + \text{Length of the Fraction (Mantissa) field}$$

These fields are arranged in the order shown below. The length of the exponent and mantissa fields specify the floating point representation.



For example, the floating point representation is described for the following:

IEEE Single Precision Floating Point	If the exponent length is 8 bits and the mantissa length is 23 bits, this implies that an SMC floating point input or output has a datatype of uint32 on the Simulink diagram.
IEEE Double Precision Floating Point	If the exponent length is 11 bits and the fraction length is 52 bits, this implies that an SMC floating point input or output has a datatype of uint64 on the Simulink diagram.

A quantity X with SMC floating point format can be represented as follows:

$$\text{sign}(X) * \text{fraction}(X) * 2^{\text{exponent}(X)}$$

The floating point representation is determined by the following conditions:

- Sign(X) is 1 if $X < 0$. Otherwise, it is 0.
- Fraction(X) is represented as 1.Y, where (.) is the binary fraction point. Note that for the representation of the mantissa, only the binary form of Y is used with the MSB denoting the position of 2^{-1} .
- Exponent(X) is represented as unsigned number biased by

$$2^{\text{exponent word length} - 1} - 1, \text{ with a range of } 0 \text{ to } 2^{\text{exponent word length} - 1}$$

Additionally, the following optional signals are available for the input/output:

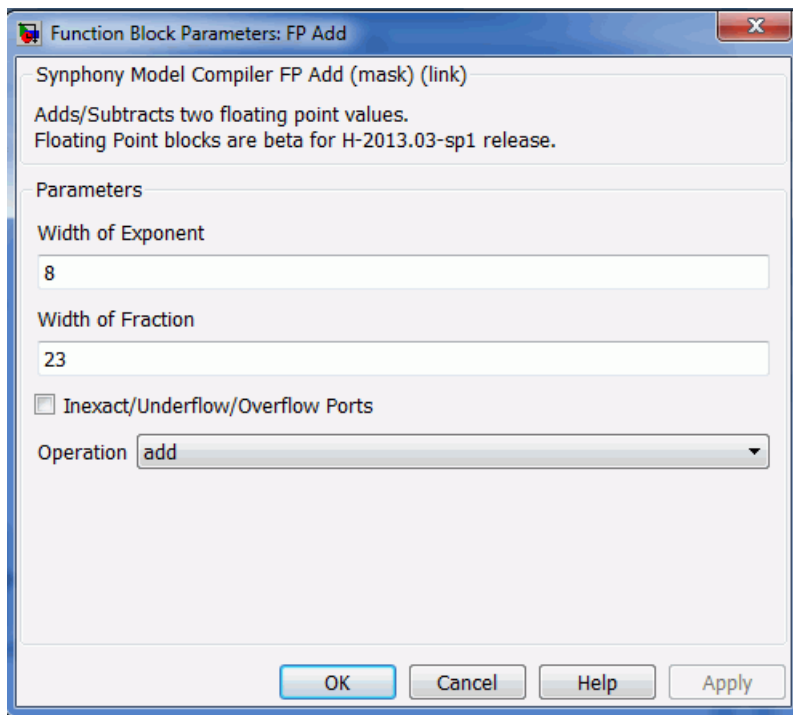
inexact	This signal goes high whenever the corresponding floating point representation is approximate. This means that the mantissa value was rounded to fit into the given format.
overflow	This signal goes high when the biased exponent value in a floating point representation is greater than the maximum value that can be expressed with the number of bits assigned to the exponent.
underflow	This signal goes high even when the biased exponent value goes to zero. The mantissa cannot be represented by a number greater than or equal to 1.0.

For all SMC floating point computations, convergent rounding generates results for rounding the mantissa as defined by the Matlab function `convergent(x)`.

Latency

The FP Add block has a latency of 9.

FP Add Parameters



Width of Exponent

Number of bits allocated for the exponent.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Inexact/Underflow/Overflow Ports

When enabled, the inexact, underflow, and overflow ports are available for each floating point number at the input/output.

Operation

Selects one of the following operations:

add	Adds the two inputs.
sub	Subtracts the two inputs.
addsub	Performs dynamic addition or subtraction for the two inputs. The operation performed depends on the value of the addsub input: <ul style="list-style-type: none">• 0 - The inputs are added.• 1 - The second input is subtracted from the first.

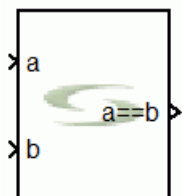
SMC FP Compare

Compares two floating point numbers and returns 1 if the selected condition holds true. Otherwise, returns 0.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Compare block compares two floating point numbers and returns 1 if the selected condition holds true. Otherwise, 0 is returned.

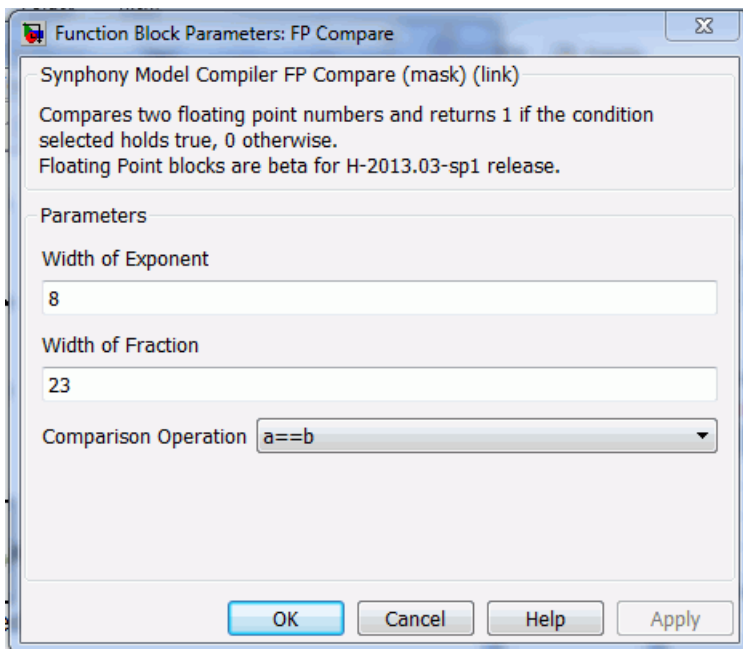
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats, on page 286](#).

Latency

The FP Compare block has no latency

FP Compare Parameters



Width of Exponent

Number of bits allocated for the component.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Comparison Operation

The following comparison operations are available: =, !=, <, <=, >, >=

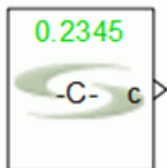
SMC FP Constant

Sets a constant value of the specified floating point representation as the output.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Constant block sets a constant value of the specified floating point representation as the output.

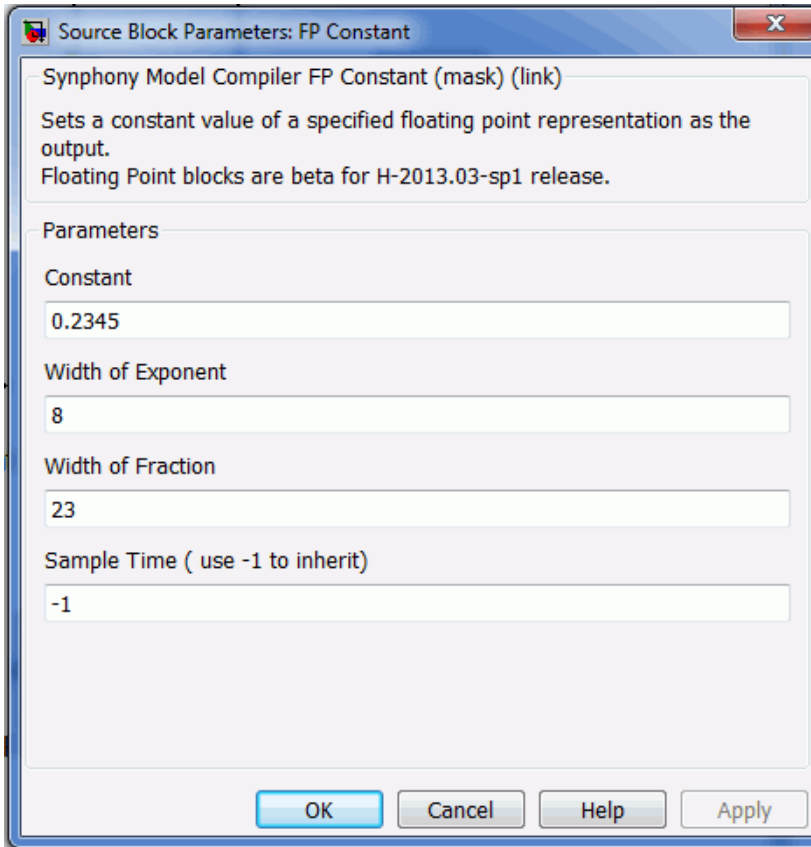
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats, on page 286](#).

Latency

This block has no latency.

FP Constant Parameters



Source Block Parameters: FP Constant

Synchrony Model Compiler FP Constant (mask) (link)

Sets a constant value of a specified floating point representation as the output.
Floating Point blocks are beta for H-2013.03-sp1 release.

Parameters

Constant
0.2345

Width of Exponent
8

Width of Fraction
23

Sample Time (use -1 to inherit)
-1

OK Cancel Help Apply

Constant

The value of the constant applied to the specified floating point representation provided for the output.

Width of Exponent

Number of bits allocated for the exponent.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Sample Time (use -1 to inherit)

The sample time of the output. When specified as -1, this means that sample time is inherited from the rest of the model. Any other legal value must be greater than zero.

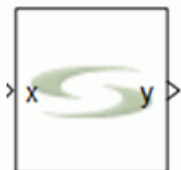
SMC Fixed to FP

Converts a Fixed Point format to SMC Floating Point format with the specified representation.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler Fixed to FP block converts a Fixed Point format to SMC Floating Point format with the specified representation.

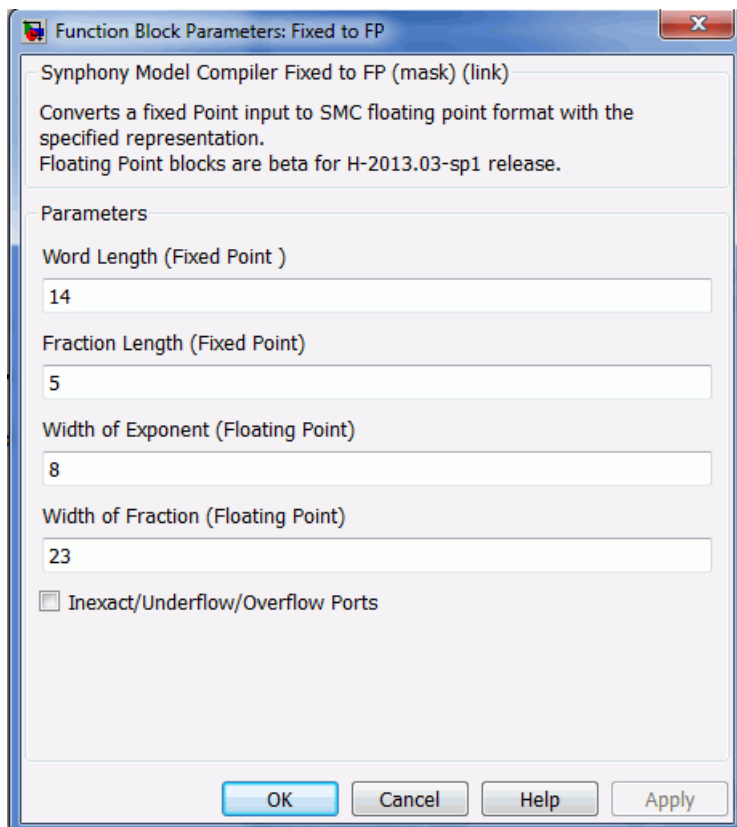
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats, on page 286](#).

Latency

This block has no latency.

Fixed to FP Parameters



Word Length (Fixed Point)

Word length for the fixed point input.

Fraction Length (Fixed Point)

Fraction length for the fixed point input.

Width of Exponent (Floating Point)

Number of bits allocated to the exponent for the floating point output.

Width of Fraction (Floating Point)

Number of bits allocated to the fraction for the floating point output.

Inexact/Underflow/Overflow Ports

When enabled, the inexact, underflow, and overflow ports are available for each floating point number at the input/output.

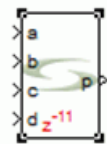
SMC FP Fused Mult Add

Performs various multiply-add operations on three or four inputs.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Fused Mult Add block performs fused Mult Add operations on three/four inputs. You can select the operation from the mask parameter. Using the FP Fused Mult Add block can provide substantial area savings compared with the individual FP Mult or FP Add blocks, but this block may cost higher approximation errors. The floating point representations are the same for the input and output floating points. Multiple FP Fused Mult Add blocks can be combined to perform floating point dot product operation. You can use two FP Fused Mult Add blocks to perform floating point for complex multiplications.

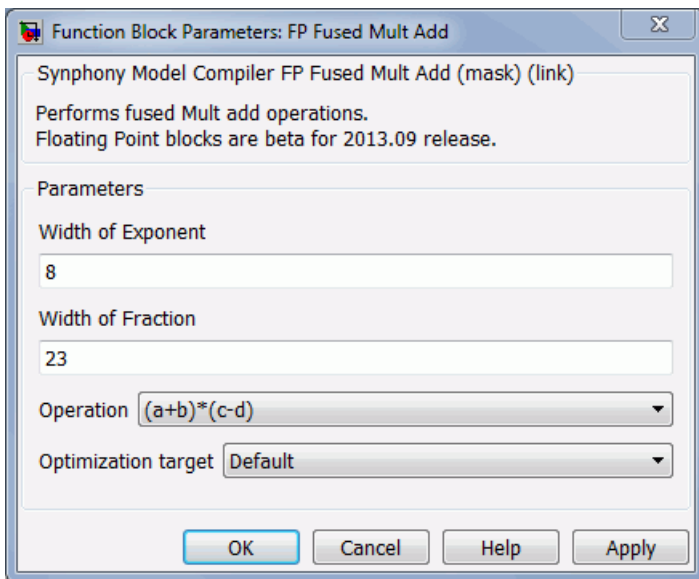
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats](#), on page 286.

Latency

This FP Fused Mult Add block has a latency of 14 for $a*b+c$, $a*b+c*d$ and 11 for the other operations.

FP Fused Mult Add Parameters



Width of Exponent

Specifies the number of bits to allocate for the exponent.

Width of Fraction

Specifies the number of bits to allocate for the fraction (mantissa).

Operation

Lets you to select one of the following operations:

$(a+b)*(c-d)$	
$(a-b)*(c-d)$	
$(a-b)*(c+d)$	
$(a+b)*(c+d)$	
$a*(b-c)$	
$a*(b+c)$	
$a*b+-c*d$	Both add and sub outputs are available in parallel.
$a*b+-c$	Both add and sub outputs are available in parallel.

Optimization Target

Specifies the target device for the FP Fused Mult Add block.

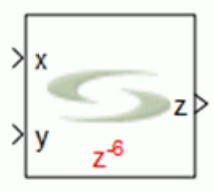
SMC FP Mult

Multiplies two floating point values.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Mult block multiplies two floating point values. The input and output floating point representations are the same.

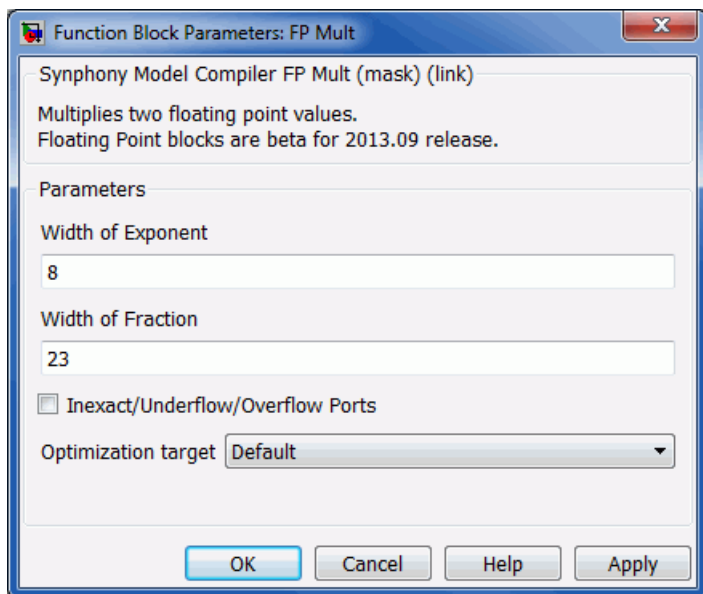
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats, on page 286](#).

Latency

The Floating Point Mult block has a latency of 6.

FP Mult Parameters



Width of Exponent

Number of bits allocated for the exponent.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Inexact/Underflow/Overflow Ports

When enabled, the inexact, underflow, and overflow ports are available for each floating point number at the input/output.

Optimization Target

Specifies the target device for the FP Mult block.

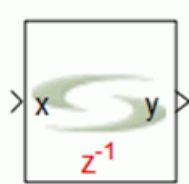
SMC FP Port In

Converts the format from Simulink double to SMC floating point.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Port In block converts Simulink double to SMC floating point format. You can use this block instead of SMC Port In to define the RTL generation boundary of floating point designs.

Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats](#), on page 286.

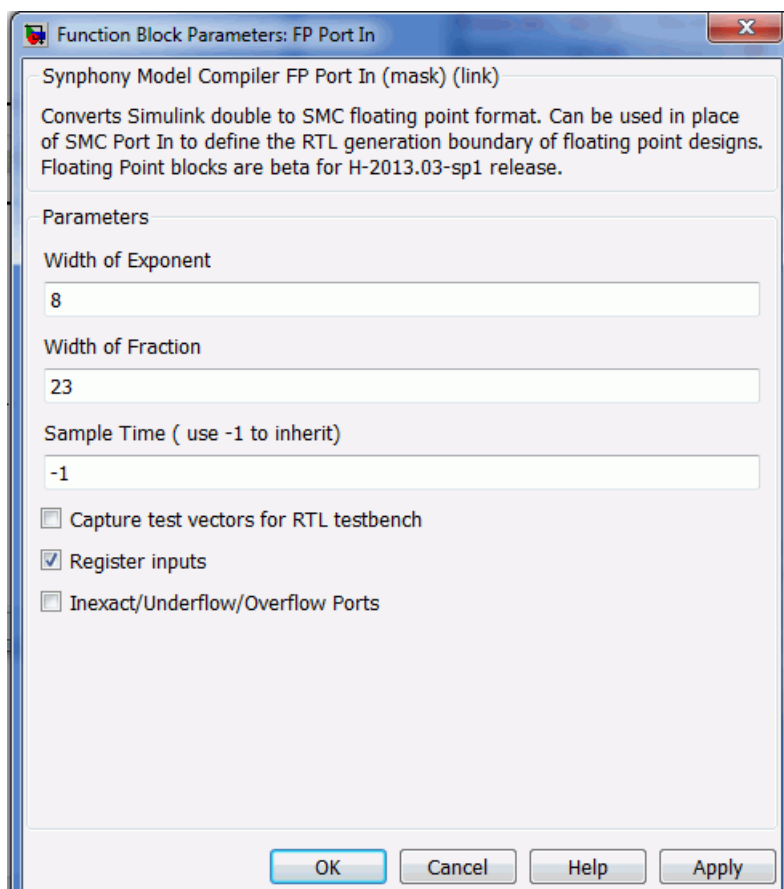
RTL Generation

The RTL generated for FP Port In is a module that has the same name as the FP Port In instance. The block ports are determined by the setting of the Inexact/Underflow/Overflow parameter, which creates additional ports when this parameter is enabled. When performing RTL integration, make sure that `FP_prt_In instanceName_porty` is the top-level port.

Latency

This FP Port In block has a latency of 1, if the register input is selected. Otherwise, the latency is 0.

FP Port In Parameters



Width of Exponent

Number of bits allocated for the exponent.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Sample Time (use -1 to inherit)

The sample time of the output. When specified as -1, this means that sample time is inherited from the rest of the model. Any other legal value must be greater than zero.

Capture Test Vectors for RTL Testbench

When enabled, each input captures the test vectors on the sample clock and saves them in a file. The software can use this file when it generates RTL to create stimuli for the RTL design. The .dat files for the test vectors are stored in the *modelFileDir/test_vectors* directory.

Register Input

When enabled, the input is registered. With registered input, the block has a latency of 1. The registers are generated with an attached `syn_keep` directive that instructs the synthesis tools not to move these registers during retiming.

Inexact/Underflow/Overflow Ports

Determines whether inexact, underflow, and overflow ports are created for each floating point number at the input and output.

- When the option is disabled, the module has only one data input (port `x`) and one output `y`. The `y` output is connected to the rest of the design. When performing RTL integration, make sure that `FP_prt_In instanceName_prt` is the top-level port.
- If the option is enabled, the tool creates corresponding 1-bit input and output ports as shown here:

Input Port Names	Output Port Names
<code>portinex</code>	<code>inexact</code>
<code>portufl</code>	<code>underflow</code>
<code>portofl</code>	<code>overflow</code>

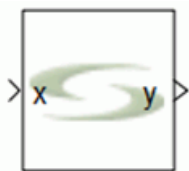
SMC FP Port Out

Converts SMC floating point format to Simulink double.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP Port Out block converts SMC floating point format to Simulink double. You can use this block instead of SMC Port Out to define the RTL generation boundary of floating point designs.

Latency

This Float Point Out block has a latency of 1, if the register output is selected. Otherwise, the latency is 0.

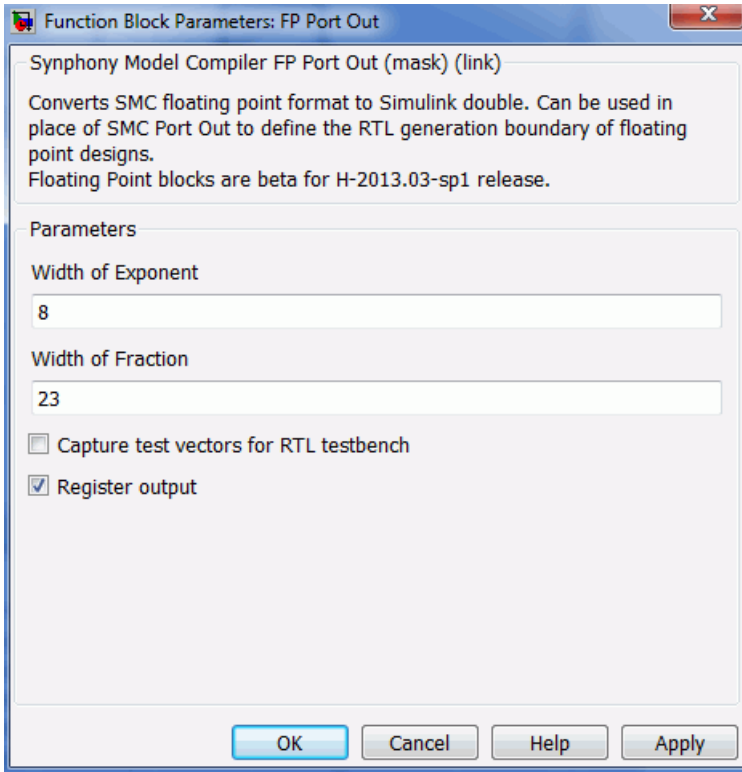
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats](#), on page 286.

RTL Generation

The RTL generated for FP Port Out is a module that has the same name as the FP Port Out instance. This module will only have one data output (portx) and one input x. The x input is connected to the rest of the design. When performing RTL integration, make sure that the `FP_Port_Out_instanceName_portx` is the top-level output port.

FP Port Out Parameters



Width of Exponent

Number of bits allocated for the exponent.

Width of Fraction

Number of bits allocated for the fraction (mantissa).

Capture Test Vectors for RTL Testbench

When enabled, each input captures the test vectors on the sample clock and saves them in a file. The software can use this file when it generates RTL to create stimuli for the RTL design. The .dat files for the test vectors are stored in the *modelFileDir/test_vectors* directory.

Register Output

When enabled, the output is registered. With registered output, the block has a latency of 1. The registers are generated with an attached `syn_keep` directive that instructs the synthesis tools not to move these registers during retiming.

SMC FP to Fixed

Converts an input SMC Floating Point format to a signed Fixed Point format with the specified word length and fraction length.

Library

Synphony Model Compiler [Floating Point Functions](#).

Description



The Synphony Model Compiler FP to Fixed block converts an input SMC Floating Point format to a signed Fixed Point format with the specified word length and fraction length.

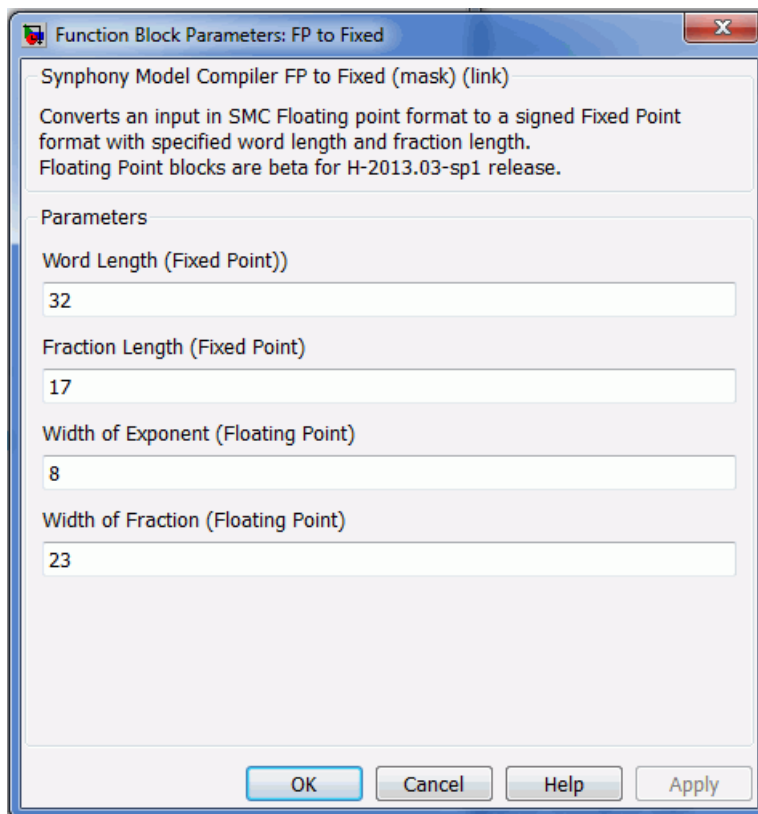
Floating Point Formats

For information about SMC floating point input and output formats, see [Floating Point Input and Output Formats](#), on page 286.

Latency

This block has no latency.

FP to Fixed Parameters



Word Length (Fixed Point)

Word length for the fixed point output.

Fraction Length (Fixed Point)

Fraction length for the fixed point output.

Width of Exponent (Floating Point)

Number of bits allocated to the exponent for the floating point input.

Width of Fraction (Floating Point)

Number of bits allocated to the fraction for the floating point input.

SMC Gain

Implements a constant gain to the input.

Library

Synphony Model Compiler [DSP Basics](#) and Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Gain block provides a constant gain by multiplying the input by the specified gain factor. For trivial gain values, the software does the following optimizations:

0	Output is connected to 0
1	Output is connected to input
$2^{\pm n}$	Output is shifted left/right by n (n being an integer)

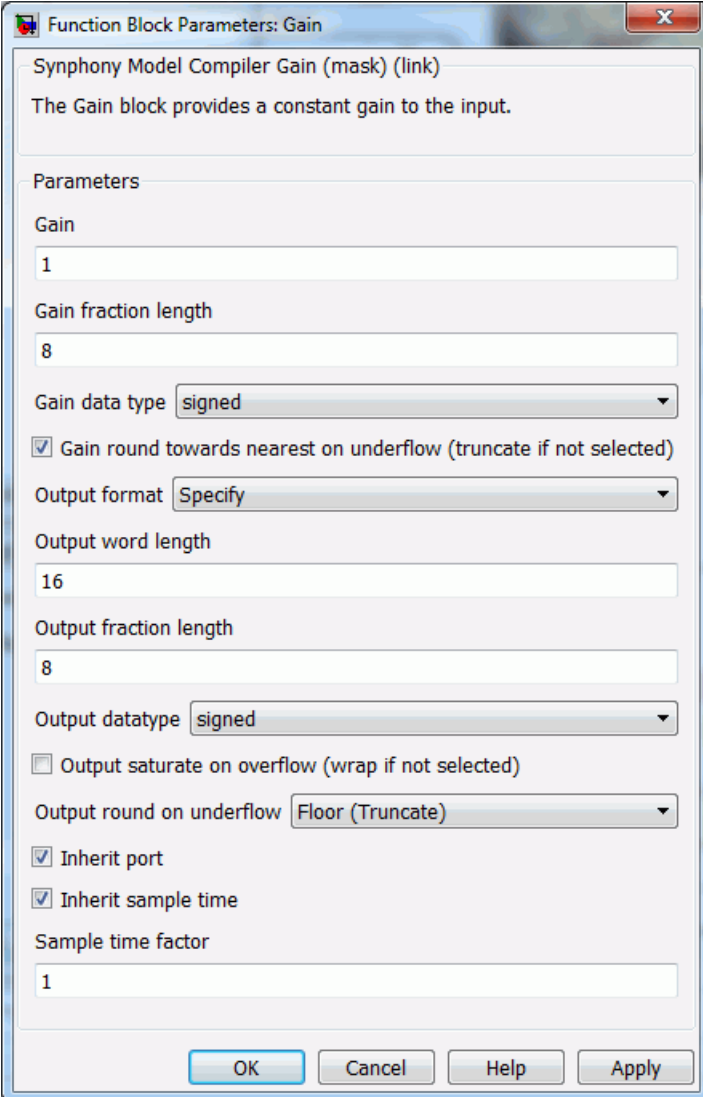
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Gain Parameters



The dialog box titled "Function Block Parameters: Gain" contains the following parameters:

- Symphony Model Compiler Gain (mask) (link)**
The Gain block provides a constant gain to the input.
- Parameters**
 - Gain**: 1
 - Gain fraction length**: 8
 - Gain data type**: signed
 - ☒ Gain round towards nearest on underflow (truncate if not selected)
 - Output format**: Specify
 - Output word length**: 16
 - Output fraction length**: 8
 - Output datatype**: signed
 - ☐ Output saturate on overflow (wrap if not selected)
 - Output round on underflow**: Floor (Truncate)
 - ☒ Inherit port
 - ☒ Inherit sample time
 - Sample time factor**: 1

Buttons: OK, Cancel, Help, Apply

Gain

Specifies the factor by which the input is multiplied to implement the gain. If the input to the block is a vector or matrix, you can specify the gain value as a column or row vector/matrix, with different gain factors for each

channel. The value you enter in this field must either be a scalar or have the same dimensions as the input.

Gain fraction length

Specifies the accuracy of the fraction requested for the coefficient value. The software infers the total word length of the coefficient automatically inferred from the value.

Gain data type

Determines the data type for the gain value (specified in the Gain option) for the block. You can set it to signed or unsigned.

Gain round towards nearest on underflow

Determines how the underflow for the gain is treated. Enable the option to round the underflow using the Nearest algorithm, and disable it to round the overflow with the Floor (truncate) algorithms. See [Underflow Rounding Options, on page 585](#) for details.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round on underflow

Determine how overflow and underflow are treated. These options are only available when Output format is set to Specify.

Output saturate on overflow	When enabled it saturates the overflow; when disabled, it wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Inherit port

Determines whether the tool creates an inherit port. The tool creates an inherit port when you enable the option. If you enable the option, the tool applies automatic scalar expansion to the inherit and data ports. If one input is scalar and the other is vector, the scalar input is expanded to the size of the vector input.

This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Special Variables, on page 588](#) for information about these variables.
- Use the `inherit` option to specify the Output data type. See [Output Data Type, on page 101](#) for a description of the option.

Inherit sample time

Determines whether the output inherits the sample time from the inherit port. Enabling this option means that the output port inherits the sample time from the inherit port, and disabling it means the output port inherits sample time from the input. This option becomes available when you enable Inherit port.

Sample time factor

Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

SMC Gold Sequence Generator

Generates a Gold sequence with specified polynomials u and v , of period $N = 2n - 1$, called a preferred pair.

Library

Synphony Model Compiler [Communications](#)

Description



The Gold Sequence Generator block generates a Gold sequence. The Gold sequence is an XOR between two maximal length PN sequences (see [SMC Gold Sequence Generator, on page 315](#)) with specified polynomials u and v , of period $N = 2n - 1$, called a preferred pair. The set of gold sequences for a given pair of component PN sequences (u, v) defined as

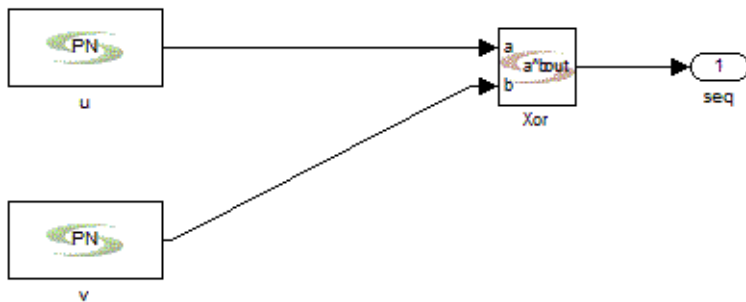
$$G(u, v) = \{u, v, u \oplus v, u \oplus Tv, u \oplus T^2v, \dots, u \oplus T^{N-1}v\}$$

where,

- T represents cyclical left shift of the corresponding component sequence by one place,
- \oplus represents element-wise XOR operation of two vectors of same length. Note that $G(u, v)$ contains $N + 2$ sequences of period N .

The Gold Sequence Generator block outputs one of these sequences according to the initial states (1) and (2) parameters. This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for definition).

The following figure shows the internal modeling:



Latency

This block has no latency.

Gold Sequence Generator Parameters

Source Block Parameters: Gold Sequence Generator

Synphony Model Compiler Gold Sequence Generator (mask) (link)

Generate a Gold sequence from a set of sequences by specifying a preferred pair of polynomials.

The polynomial parameter values represent the shift register connections. Enter these values as either a binary vector or a descending ordered polynomial to indicate the connection points.

The initial states parameters are binary vectors that represent the starting state of the shift registers.

Parameters

Preferred polynomial (1)

[7 3 0]

Initial states (1)

[0 0 0 0 0 1]

Preferred polynomial (2)

[7 3 2 1 0]

Initial states (2)

[0 0 0 0 0 1]

☐ Reset port

☐ Enable port

Sample time (Use -1 to inherit)

1/100e6

OK Cancel Help

Preferred polynomial (1)

Represents the generator polynomial used for constructing the first preferred pair of PN sequences (u). See [Generator polynomial, on page 141](#) for details.

Preferred polynomial (2)

Represents the generator polynomial used for constructing the second preferred pair of PN sequences (v). See [Generator polynomial, on page 141](#) for details.

Initial States (1)

Represents the [Initial states, on page 398](#) used for constructing the first preferred pair of PN sequences (u).

Initial States (2)

Represents the [Initial states, on page 398](#) used for constructing the second preferred pair of PN sequences (v).

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you specify reset or enable ports.

SMC HLS Subsystem

Lets you add a previously designed Symphony model to the current design and set implementation settings for it.

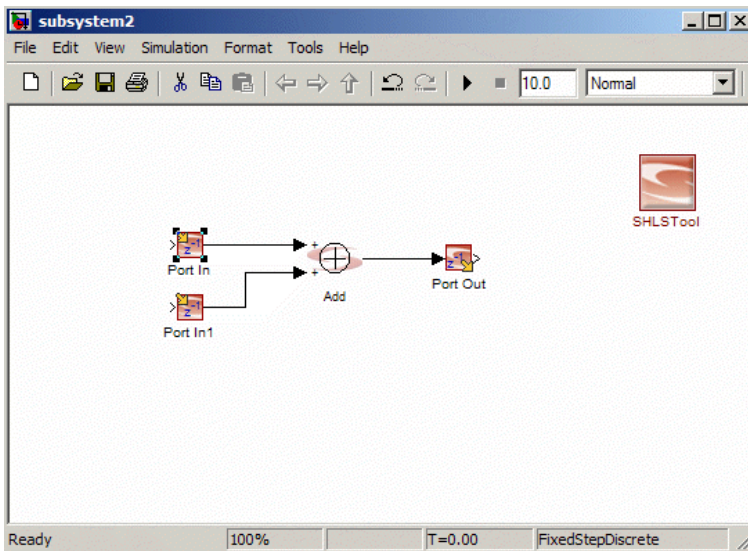
Library

Symphony Model Compiler [Ports & Subsystems](#)

Description



The Symphony Model Compiler HLS Subsystem block lets the designer instantiate a model that has already been designed. The tool synthesizes the block using the original model. For simulation, it copies the contents of the HLS Subsystem block and uses the copy for simulation.



The HLS Subsystem block is intended to be used in a bottom-up design flow, where you implement independent modules and optimize them separately, and then assemble them at the top level. For information about using the block, see [Using the HLS Subsystem Block, on page 786](#).

The `shls.log` file documents details of the HLS Subsystem blocks used in the design. For more information about this and the file structure the tool uses when you include HLS Subsystem blocks, see [Using the HLS Subsystem Block, on page 786](#).

Tool Simulation Process for HLS Subsystem Blocks

The tool uses the imported subsystem model file as the simulation model of the subsystem. For information about simulating the block design, see [Simulating HLS Subsystem Blocks, on page 844](#). The tool goes through the following steps when it simulates the block:

1. If Lock HLS Subsystem button is disabled, the tool automatically imports the subsystem model file as follows:
 - It loads the subsystem model file and compiles it to evaluate all parameter values. It copies each block to under the subsystem, and replaces parameter specifications with local variables for the subsystem to preserve actual values.
 - The tool converts the subsystem Port In and Port Out blocks to subsystem ports. It also adds From and GoTo blocks if the subsystem has Port In and Port Out blocks that are not at the top level. Additionally, it inserts Convert blocks after the subsystem input ports to enforce the data types specified at the Port In blocks.
 - HLS Subsystem modules must have registered I/Os. The tool inserts Delay blocks at inputs and outputs to account for I/O registers.
2. If Lock HLS Subsystem button is enabled, the tool preserves the existing simulation model, and does not regenerate the subsystem block.
3. For multichannelization, see [Vectorize scalar ports for multichannelized subsystems, on page 324](#). When multichannelization optimization is used for a reference subsystem, the tool instantiates a copy of the subsystem for each channel to simulate the behavior.
4. The tool compares the data type and rate of each HLS Subsystem I/O port to the data type and the rate of the associated port in the subsystem model and then propagates the data types and rates.

It generates an error if it cannot find the model file. If there is a mismatch between data types or data rates of the subsystem ports and the top-level signals, the tool stops simulation and issues an error message.

5. It then continues with the rest of the simulation process as usual.

Tool Synthesis Process for HLS Subsystem Blocks

When RTL is generated at the top level, the tool goes through the following steps to synthesize the subsystem:

1. Updates the model and compiles it.
2. Runs simulation as described in [Tool Simulation Process for HLS Subsystem Blocks, on page 320](#). The tool uses the imported subsystem model file as the simulation model of the subsystem.

The optimization parameters set for the subsystems are considered part of the behavior of the top level. For example, the number of channels or the latency of the subsystem is reflected in the top-level simulation.

3. Checks that top-level and subsystem parameters for synthesis match in the active subsystem implementation.

If there is a mismatch the synthesis parameters of the top level override the parameters of the subsystem, and you see a warning message in the synthesis log file.

4. Generates RTL for the subsystems.
 - If Lock HLS Subsystem was disabled, the process generates RTL for each HLS Subsystem. For each HLS Subsystem, the tool uses the subsystem model file to generate RTL.
 - If Lock HLS Subsystem was enabled, the tool does not regenerate RTL for each HLS Subsystem, but uses the RTL generated previously.
5. Generates RTL for the top level.
6. Generates a test bench for the top level if you selected that option. The tool uses the do files generated for the HLS Subsystem as input in this process.

Limitations to Using the HLS Subsystem Block

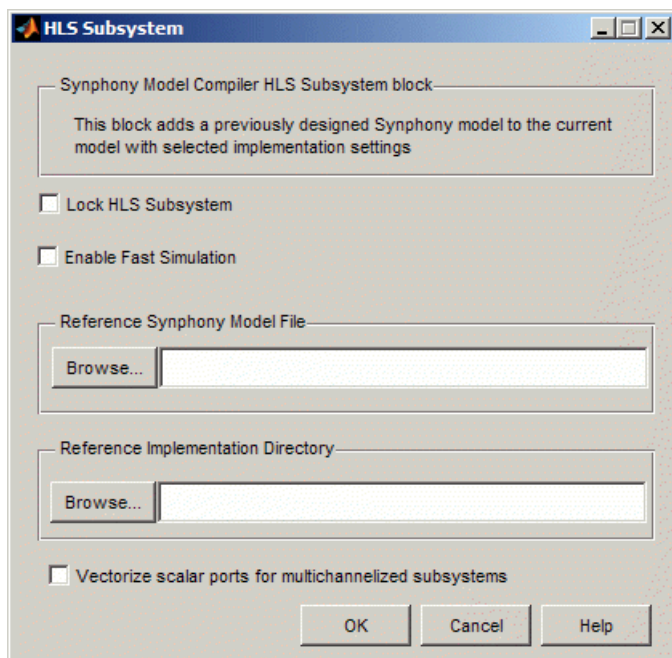
Currently there are some limitation to using the HLS Subsystem block:

- An HLS Subsystem block cannot contain another HLS Subsystem block; in other words, you cannot have nested HLS Subsystem blocks.
- Subsystem models cannot contain black boxes, but you can have black boxes at the top level.
- Subsystem models cannot contain Viterbi blocks, but you can have Viterbi blocks at the top level.
- The tool does not support folding, retiming, and multichannelization at the top level when you have HLS Subsystem blocks in the design.

Latency

The latency of this block is determined by the subsystem design, and the optimizations specified for it.

HLS Subsystem Parameters



Lock HLS Subsystem

Determines whether RTL is regenerated for the block when you run synthesis on the top-level design:

- If Lock HLS Subsystem is disabled, the tool generates RTL for each HLS Subsystem block. It uses the subsystem model file to generate RTL for each HLS Subsystem block. It first generates RTL for the subsystem, using the specified model file as the top level. It then instantiates this RTL in the top-level RTL.
- If Lock HLS Subsystem is checked, the tool does not regenerate RTL for each HLS Subsystem, but uses the RTL information for the block generated from the previous top-level synthesis run. Use this option when your subsystem design is not going to change. This option speeds up runtime. Do not use this option unless you have run top-level synthesis and generated RTL for the subsystem at least once before.

Enable Fast Simulation

Determines whether a C-model is used to simulate the block.

- When the option is disabled, the tool uses the Simulink model of the HLS Subsystem block for simulation.
- When it is enabled, the tool uses the C-model and S-function wrapper of the subsystem for simulation instead of the Simulink model. Although the generation of the C-model can take some time, especially for large subsystems, the model only needs to be generated once. This means that checking Enable Fast Simulation can speed up runtime, because the C-model does not have to be regenerated. Each HLS Subsystem block for which fast simulation is enabled consumes a C output license.

In both cases, the tool instantiates a subsystem block under the HLS Subsystem block hierarchy called HLSSimModel, which is used exclusively for simulation. The generation of the HLSSimModel information varies with the setting you choose:

Without Fast Simulation	Fast Simulation Mode
Copies all blocks in the HLS Subsystem model file to the HLS Subsystem block.	Uses the C-model, which is instantiated in HLSSimModel.
Converts subsystem Port In and Port Out blocks to subsystem ports. Adds From and Go To blocks if the ports are not at the top level of the subsystem.	
Adds Convert blocks after the subsystem input ports to enforce the data types and sample time.	The S-function wrapper implements the data type and sample time consistency checks.
Adds delay for registered input and output ports.	Included in C-model.
Adds implementation latency to the output as delay.	Included in C-model.
For multichannel implementations, replicates HLSSimModel, according to the number of channels required.	C-model includes ports for each of the channels.

See [Simulating HLS Subsystem Blocks, on page 844](#) for a step-by-step procedure.

Reference Symphony Model File

Specifies the model file (.mdl) that describes the behavior of the subsystem. The tool uses the parameters from the specified implementation for the model. Use the **Browse** button if needed to locate the implementation. You can specify relative or absolute paths in this field.

Reference Implementation Directory

Specifies which implementation to use for the specified model file by indicating the appropriate implementation directory. Use the **Browse...** button if needed to locate the directory. You can specify relative paths in this field.

Vectorize scalar ports for multichannelized subsystems

Determines how scalar ports are implemented for multichannelized subsystems:

- When enabled, displays the input or output scalar ports for multichannel reference implementations as vector signals on the HLS Subsystem block. Each vector element represents one channel data. This option does not affect vector/matrix ports.
- When disabled, all scalar ports for each channel are populated as separate ports.

SMC Host Interface

Provides a slave interface to simpler bus protocols that let you interface with the host processor and configure the design.

Library

Symphony Model Compiler [Ports & Subsystems](#)

Description

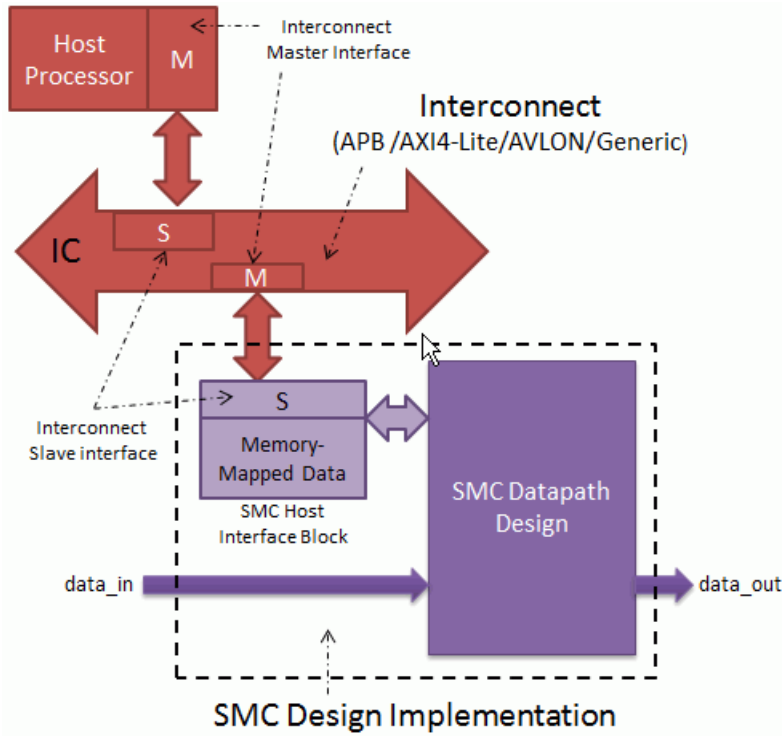


The Host Interface provides a slave interface to a simpler interconnect protocol that lets the design interface with the host processor and load the memory-mapped configuration registers and parameters required by the SMC design. For information about using this block in an SMC design, see [Synthesizing with a Host Interface Block, on page 678](#).

This block supports the following bus protocols in slave mode: AXI4-Lite, APB, AVALON, and Generic Interface. The bus protocols are described in detail in [Bus Protocols, on page 738](#). You can enter the memory map and bus interface information, and the memory map registers are output to the SMC design at asynchronous sample rates. The block also allows you to add synchronizers for single-bit memory map elements.

You cannot use workspace variables to specify Host Interface block parameters.

The following schematic illustrates how the block fits into the system at the next level of abstraction, providing a protocol-specific slave interface module and storage for memory-mapped configuration data:



When simulating and synthesizing a model that includes the Host Interface block, note the following:

- Never register the input and output bus protocol interface signals in the SMC design. This is because all the protocols have a handshake mechanism which would no longer be compliant with the protocol if the bus interface signals are delayed. This leads to unexpected results.
- You cannot use multichannelization during synthesis. Multichannelizing the block would mean replicated memory-mapped registers with the same address, which is not supported.
- You cannot use folding. Folding creates registers on the ports and typically introduces delay on the bus interface signals, which leads to the failure to meet bus protocol specifications.
- You cannot use retiming, because this optimization tends to create registers on the bus protocol signal outputs from the Host Interface block.

The registers violate the handshake mechanism between the Host Interface block and the bus master.

- You can use any reset sensitivity and polarity for models that incorporate the Host Interface block.

Latency

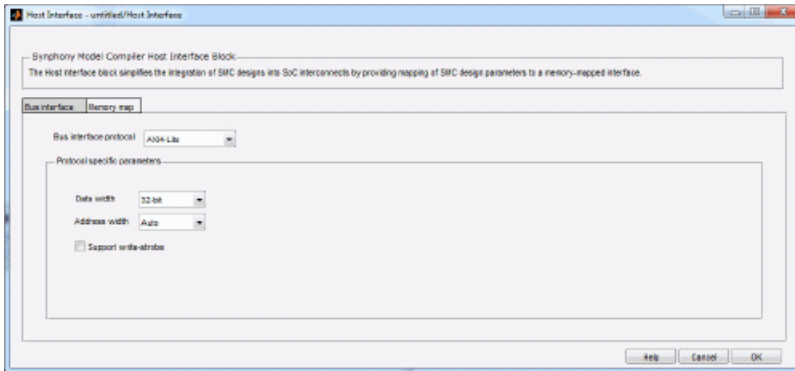
The block does not have a fixed latency for the response to a request from the initiator, which is either a bus interconnect or a master. The interaction of the Host Interface block with the initiator is affected by two factors:

- The bus protocol and the handshake mechanism defined by the bus protocol.
- The current state of the block: whether it is in idle state or currently processing the previous request of the initiator.

The memory map configuration data is first stored in the bus clock domain. The block then adds an additional output register, clocked by the destination clock of the respective memory map elements, before outputting the configuration data to the SMC design. The additional register is to avoid mismatches between RTL and Simulink simulations. Simulink has a limitation where it introduces one additional latency when simulating signals at a rate different than the rate at which the signal is driven, and the block adds the extra register to match that behavior. The register is not required for single-bit memory map elements that are synchronized, because in Simulink, the signal is driven by the same rate at which it is simulated, because of the synchronizer flop.

This addition of the extra output register stage makes memory map configurations available to the SMC design on the following rising edge of the destination clock after the host configures them. For single bit memory map elements for which you have specified a synchronizer block, the memory map register bit is available to the SMC design after the synchronizer delay.

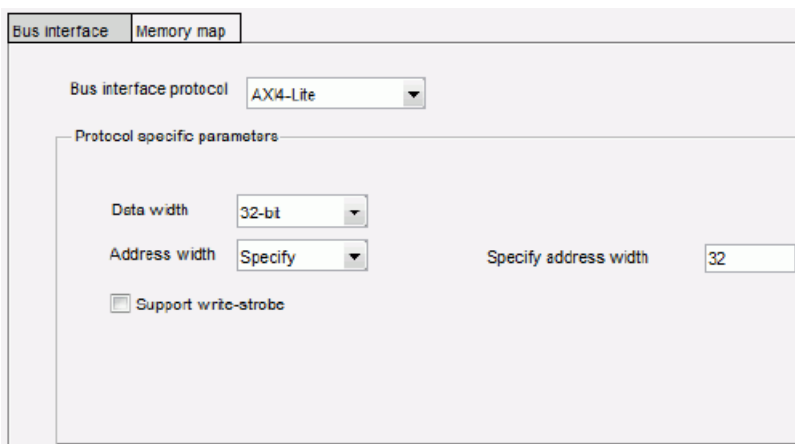
Host Interface Parameters



The block mask has two tabs, [Bus Interface Tab, on page 329](#) and [Memory Map Tab, on page 331](#).

Bus Interface Tab

The parameters on this tab affect the bus interface.



Bus interface protocol

Specifies the slave protocol you need to implement. The block updates its bus interface based on the protocol you select. For descriptions of the four bus protocols available, see the following sections:

- [AXI4-Lite Protocol, on page 738](#)

- [APB Protocol, on page 743](#)
- [Generic Interface Protocol, on page 748](#)

Datawidth

Sets the word size for the bus protocol. Possible values are 8-bit, 16-bit, 32-bit, and 64-bit, but not all protocols support all values. The only choices for AXI4Lite are 32bit and 64bit. APB does not support 64bit.

Address width

Determines the address bus width. You have two choices:

- Auto
The block automatically calculates the minimum address width based on the specified base address, data width and the maximum word address and uses it as the address width. The following formula is used to calculate the minimum address width:

$$\text{ceil}(\log_2(\text{baseaddress} + (\text{maximum word address} * \text{datawidth}/8)))$$

- Specify
Lets you explicitly specify the address bus width in Specify address width.

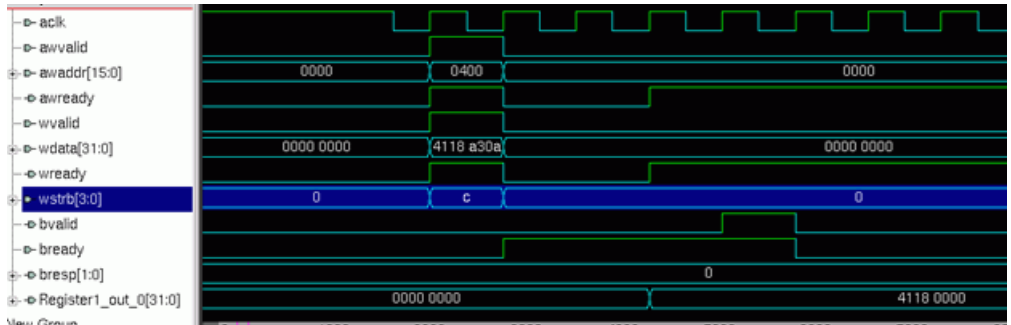
Specify address width

Specifies address bus width when Address Width is set to Specify.

Support write strobe

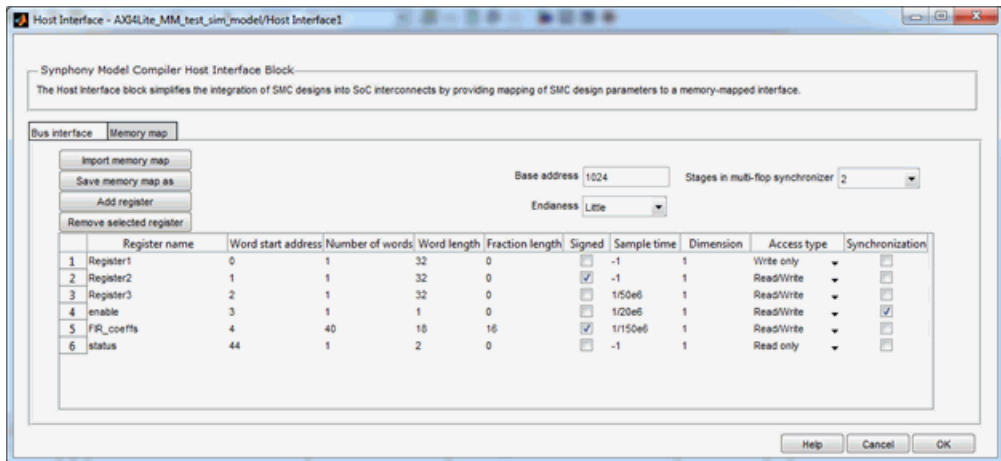
Determines support for write strobes or byte enables during write transactions.

- When disabled, the block assumes that all the bytes in the data word are selected for writing.
- When enabled, the block supports byte enables during the write transactions. The following shows a simulation waveform during write transaction for AXI4Lite protocol when write strobes are used:



Memory Map Tab

Lets you set memory mapping configurations in a tabular format.



Import memory map

Loads previously saved memory map data from a csv file or an xml file that is compatible with IP-XACT.

Save memory map as

Lets you save the memory map data you specified in the table in a csv file or as an xml file that is compatible with IP-XACT.

Add register / Remove selected register

Adds new memory map elements to or deletes selected elements from the table.

Base address

Specifies the base address of the slave. A byte-precise address must be specified, but the address has to be word aligned. If not, the block errors out without generating the RTL.

Take a case where the data width is 32. As the word size is 4 bytes, the specified base address must be divisible by 4. This means that the last two bits of the base address must be zero.

Endianness

Determines how the bytes of the data bus are mapped.

- Little
The data bus bytes are directly mapped to the memory map register bytes.
- Big
The i th byte of the data bus is mapped to $(N-i)$ th byte of the memory map registers. N is the number of bytes of the data bus.

Stages in multi-flop synchronizer

Sets the number of flops required for the multi-flop synchronizer. The option is valid only if Synchronization is checked for a memory map element.

Register name

Specifies the name of the memory map register. Only alphanumeric characters and underscores are allowed. The name cannot start with a number. It must not start or end with an underscore. The name must not contain consecutive underscores or spaces.

The memory map port name in the Host Interface RTL follows this convention:

register_name_in|out_eIndex

<i>register_name</i>	Specified register name
in	Read only access type
out	Write only, or read/write access type
<i>index</i>	0, 1, 2 ... <number of words in the memory map element - 1>

The memory map output/input name of the Simulink block is different. For single-word memory map elements, the block name is specified as:

`<register_name>_in|out_e0`

For memory map elements where the number of words is greater than 1, the block name is specified without the index number as shown below. This is because the Simulink block vectorizes the memory map elements with number of words greater than 1 and creates a single port.

`<register_name>_in|out`

Word start address

Specifies a value that is multiplied by the number of bytes in the data bus to get the byte-precise address of the word. The base address is added to this value to get the absolute address.

Absolute start address of a register= (word start address) x (number of bytes in data bus) + (base address)

If the memory map element has multiple words, the absolute address of the subsequent word is computed by incrementing the word address of previous word by 1.

For example, consider a 40-tap FIR filter with configurable coefficients coming from the host processor. If the word start address is 3, data bus width is 32, and the base address is 1024 (this value must be divisible by number of bytes in the databus i.e. 4), then the absolute addresses are as follows:

Absolute address of 1st coeff = $3 \times 4 + 1024 = 1036$

Absolute address of 2st coeff = $4 \times 4 + 1024 = 1040$

...

...

Absolute address of 40th coeff = $42 \times 4 + 1024 = 1192$

You can create unused memory map locations in the IP for future expansion by specifying a word start address that does not consecutively follow the previous element in the memory. For example, take a 40-tap FIR filter with configurable coefficients where the number of taps is expected to go up to 64 taps. In this case, specify the word start address of the memory map element following the coefficients in the table as Word start address of the coefficients + 64, instead of Word start address of the coefficient + 40.

The Host Interface block does not issue errors for write transactions to unused spaces. For read transactions to unused locations, it responds with 0 data.

Number of words

Specifies the number of words for the current memory map element. For example, for a 40-tap FIR filter with configurable coefficients coming from the host processor, the number of words entered would be 40.

Word length, Fraction Length, and Signed

These data type options for the output of the memory map element are considered during Simulink simulations. The output port width (or the input port width in the case of read-only memory map) of the element in the Host Interface RTL is equal to the specified word length.

The word length must be less than or equal to width of the data bus (data width). Fractional length must not exceed the word length. The single-bit memory map elements cannot be signed.

Sample time

Specifies a sample time to be used during Simulink simulations to update the memory map ports available at the output of the block. Internally, these memory map registers are driven by the bus clock domain.

Specify -1 in this field to automatically infer the sample time of the bus interface. If the sample time is different from the bus interface sample time, no synchronizer stage is inserted in the Host Interface block. For read-only access memory map elements, you must specify -1 in this field, because the sample time will be propagated from the SMC design in this case.

Dimension

For multiple-word memory map elements, specifies the dimension of output to be used in the Simulink model. For single-word memory map elements, its value must be 1.

Enter the dimension in [a b] format, with a being the number of rows and b the number of columns. The total number of elements (a x b) must be equal to the value entered in Number of words. The individual word outputs are arranged in the row-first order in the output.

The following examples illustrate how to specify this value:

- Memory map element infers the coefficient outputs for a 40-tap FIR filter
Set the dimensions to [1 40] or [40 1], depending on whether the Filter accepts coefficients as row vectors or column vectors.
- Memory map element infers the coefficient output of a 4 channel FIR filter with each channel having 10 taps
Set the dimensions to [4 10] or [10 4], depending on how the multichannel FIR filter accepts coefficient vectors, and whether the coefficients are arranged in channel-interleaved or non-interleaved order in the memory map registers in the Host Interface block.

Access type

Defines the access of the host processor or bus interface to the memory map element.

Read only	Use this value if the memory map element value is driven by the SMC design. The tool infers an input port to the Host Interface block in this case.
Write only	Use this setting when the host processor is restricted to reading the value from the memory map. An output port is inferred in this case.
Read/Write	Specify this setting if the host processor is allowed to read as well as write to the memory map register. The tool infers an output port to the Host Interface block in this case.

Read/Write is especially useful for debugging the software. For example, it would be useful in a scenario where the SMC design assumes a 24-bit coefficient and the host processor assumes a 32-bit word length for the coefficient. In this case, this setting enables the software to read back the FIR coeffs it has written to ensure that the values are not corrupted.

The Read/Write setting does not incur any more area and timing overhead than the Write only setting, so use Write only if you explicitly need to restrict the host processor from reading back the values.

Synchronization

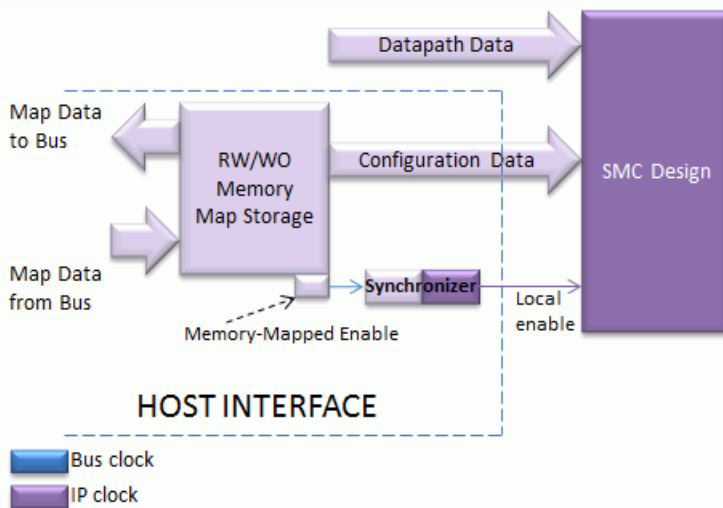
Determines whether memory map outputs that drive the SMC design are synchronized. You can only enable this option if all of the following conditions are met:

- Word length must be 1

Typically, the host processor has a configuration methodology that ensures that the effect of the newly configured data on the IP is fully controlled by a single memory-mapped bit. Given this, it is sufficient to synchronize this single bit instead of the entire configuration data. Some illustrative examples are described below:

Example1: Memory-Mapped Enable Control

The host uses memory-mapped enable to control the data processing of the SMC design. The following figure shows the interaction between the Host Interface block and the SMC design in this scenario:

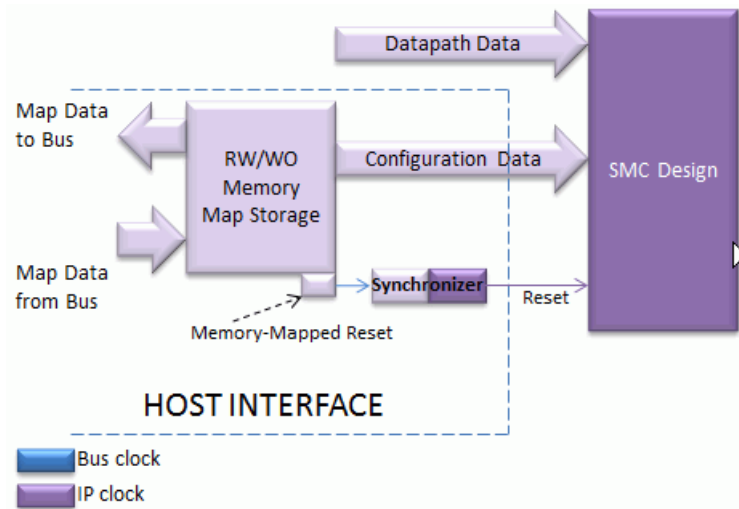


The host de-asserts memory map enable, and then configures the rest of the configuration data. Finally, the host asserts the memory map enable bit. It is assumed that the host is aware of the IP clock

frequency so that it waits until the enable bit reaches to the SMC design before starting configuration.

Example 2: Memory-Mapped Reset Control

The host uses the memory-mapped reset to control the data processing of the SMC design. The following schematic is similar to the previous one, except the synchronized bit is used to reset the design instead of stalling it.



The host first asserts the memory map reset. This bit resets the SMC design and keeps it in this state. Next the host configures the rest of the configuration data. Finally, the host de-asserts the memory map reset bit, which releases the design from reset. The design then starts processing with the new configuration data.

- Access type must not be Read only

Currently the Host Interface block does not allow the data coming from the SMC design to be synchronized.

- Sample time must not be -1

If the specified sample time is -1, the tool uses the bus interface sample time, so synchronization does not make sense.

- Number of words is 1

Currently the Host Interface block does not allow synchronization when the number of words is greater than 1.

CHAPTER 3

SMC Blocks: IIR to Viterbi Decoder

This chapter describes the Symphony Model Compiler blocks in alphabetical order, starting with the `ln` block. The other blocks are described in [Chapter 2, *SMC Blocks: Abs to Host Interface*](#).

Refer to the following sections for complete lists of the blocks, and other block-related information:

- [Blocks — By Library, on page 28](#)
- [Blocks — Alphabetical List, on page 39](#)
- [Primitives and Custom Blocks, on page 800](#)
- [Blockset Summary, on page 945](#)

The next block description is for the `ln` block.

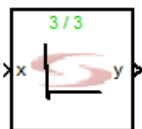
SMC IIR

Implements an infinite impulse response (IIR) filter.

Library

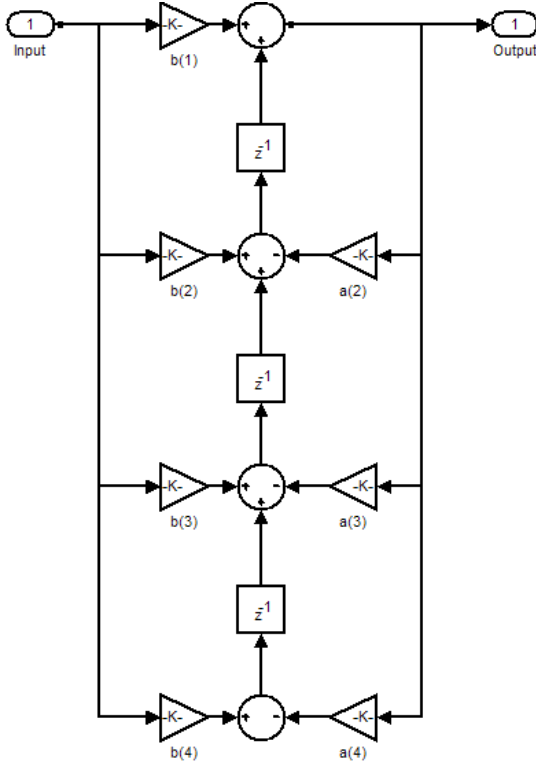
Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler IIR block implements an infinite impulse response filter, using a Direct Form II Transposed architecture. Typically, this results in smaller memory utilization.

Currently, the internal data path specification of the IIR is determined by the input specification. It is recommended that you use some fractional part on the input port of the IIR. Do not use the output format to change the data type, because the selected output format tries to increase the resolution of the output compared to input.



Forward and feedback coefficients are defined by coefficient vector or coefficient matrix parameters, according to the application. The coefficients can be extracted from an FDATool instance with the `syn_get_coefs` command. When you select Automatic for the output data type, the output word length is determined by adding an estimated amount of guard bits on top of the input word length.

Matrix and Vector Coefficient Definitions

There are various possibilities for input and coefficient signal sizes:

- Both forward and feedback coefficients are row vectors (dim:1xN) and the input is a one-dimensional signal. This is normal operation.
- Both forward and feedback coefficients are row vectors (dim:1xN) and the input is an M-dimensional signal. This results in multiple channels, each operating with the same set of forward and feedback coefficients.

The same forward and feedback coefficient array vectors are applied to each dimension of the M-dimensional input signal.

- Both forward and feedback coefficients are matrices (dim:MxN) and the input is an M-dimensional signal. This results in multiple channels, each operating with a different set of forward and feedback coefficients. Each row of the forward and feedback parameter matrices are applied to a different signal dimension in the m-dimensional input signal.[o5]

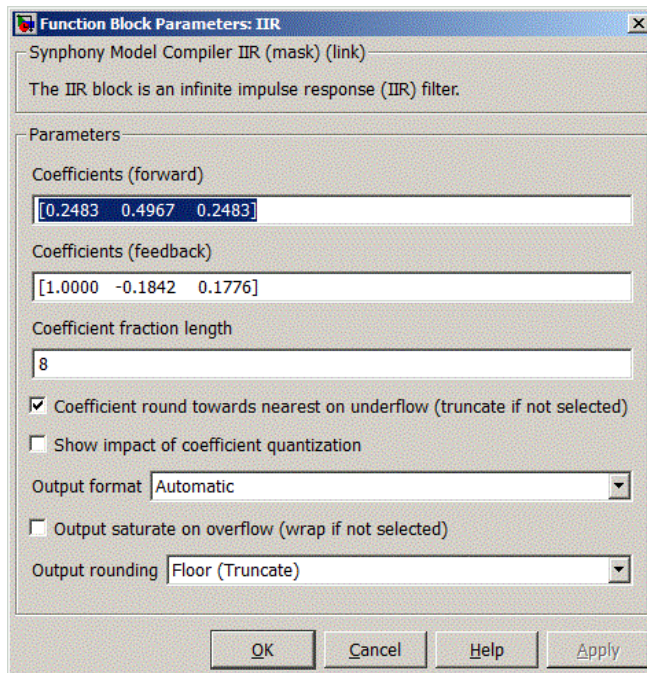
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

IIR Parameters



The dialog box titled "Function Block Parameters: IIR" contains the following elements:

- A link: "Symphony Model Compiler IIR (mask) (link)".
- A description: "The IIR block is an infinite impulse response (IIR) filter."
- A "Parameters" section with the following controls:
 - "Coefficients (forward)": A text field containing the values [0.2483 0.4967 0.2483].
 - "Coefficients (feedback)": A text field containing the values [1.0000 -0.1842 0.1776].
 - "Coefficient fraction length": A text field containing the value 8.
 - A checked checkbox: "Coefficient round towards nearest on underflow (truncate if not selected)".
 - An unchecked checkbox: "Show impact of coefficient quantization".
 - "Output format": A dropdown menu set to "Automatic".
 - An unchecked checkbox: "Output saturate on overflow (wrap if not selected)".
 - "Output rounding": A dropdown menu set to "Floor (Truncate)".
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

Coefficients (forward)

Specifies the forward coefficients in one of these ways:

- Type in a vector with the filter coefficients. See [Matrix and Vector Coefficient Definitions, on page 341](#).
- If the input to IIR is vectorized, either type in a matrix with each row specifying forward IIR coefficients for the corresponding channels, or type in a vector as usual for creating identical IIR channels. When the coefficients are defined in a matrix, the number of rows must be equal to the input vector size. See [Matrix and Vector Coefficient Definitions, on page 341](#) for additional details.
- Alternatively, type the `syn_get_coefs` command in this field to extract the coefficients from an `FDATool` instance. If you are using this method, the filter structure must be converted to a single section. For information about using the `FDATool` to convert the structure to a single section and extract coefficients, see [Defining FIR Filter Coefficients with `FDATool`, on page 768](#). See `syn_get_coefs`, on [page 604](#) for details of the function syntax.

Coefficients (feedback)

Specifies the feedback coefficients in one of the following ways:

- Type in a row vector with the filter coefficients. The first element of the feedback vector must be 1. If it is any other value, the software scales the vector so that the first element is 1. Do not set the first element to 0, as it can result in unexpected behavior.
- If the input to IIR is vectorized, either type in a matrix with each row specifying feedback IIR coefficients for the corresponding channel, or type in a vector as usual for creating identical IIR channels. When the coefficients are defined in a matrix, the number of rows must equal the input vector size. See [Matrix and Vector Coefficient Definitions, on page 341](#) for additional details.
- Alternatively, type the `syn_get_coefs` command in this field to extract the coefficients from an `FDATool` instance. For more information about this command, check the references listed for [Coefficients \(forward\), on page 343](#).

Coefficient fraction length

Specifies the fraction length for the coefficient.

Coefficient round towards nearest on underflow

Determines how the underflow for the coefficient is treated. Enable the option to round the underflow using the Nearest algorithm, and disable it to round the overflow with the Floor (truncate) algorithms. See [Overflow Saturation Options, on page 585](#) for details.

Show impact of coefficient quantization

When enabled, the spectrum window displays the coefficients with and without quantization.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583. Do not use the output format to change the data type. See Description, on page 340 for details.
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding

Determine how overflow and underflow are treated. For descriptions of these parameters, see the following:

Output saturate on overflow	Saturates (option enabled) or wraps (option disabled) the overflow. See Overflow Saturation Options, on page 585
Output rounding	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 .

SMC In

Allows you to add an in port to a subsystem to a Synphony Model Compiler design.

Library

Synphony Model Compiler [Ports & Subsystems](#)

Description



The Synphony Model Compiler In block provides an in port for a subsystem added to a Synphony Model Compiler design. For details about this block, refer to the Simulink documentation for the Inport block in the Simulink Ports & Subsystems library.

Latency

This block has no latency.

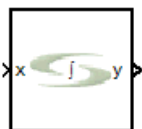
SMC Integrator

Performs a discrete time integration of the input signal.

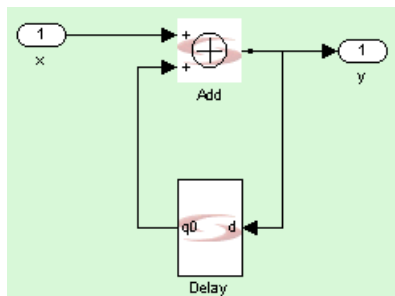
Library

Synphony Model Compiler [Filtering](#)

Description



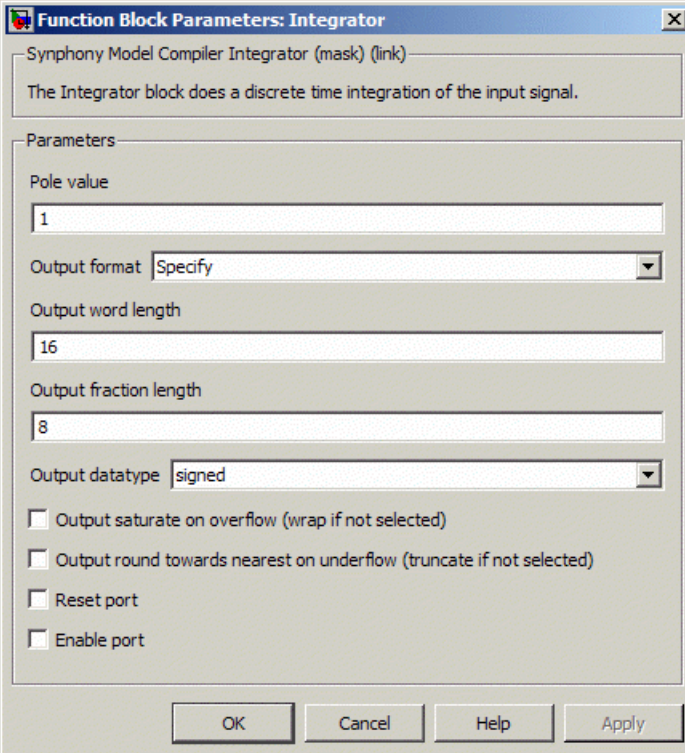
This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) performs a discrete time integration of the input signal.



Latency

This block has no latency.

Integrator Parameters



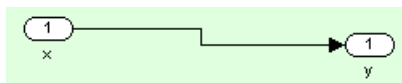
The dialog box titled "Function Block Parameters: Integrator" contains the following elements:

- A header bar with a close button (X).
- A description: "Synphony Model Compiler Integrator (mask) (link)" and "The Integrator block does a discrete time integration of the input signal."
- A "Parameters" section with the following controls:
 - "Pole value": A text input field containing the value "1".
 - "Output format": A dropdown menu currently showing "Specify".
 - "Output word length": A text input field containing the value "16".
 - "Output fraction length": A text input field containing the value "8".
 - "Output datatype": A dropdown menu currently showing "signed".
 - Four unchecked checkboxes:
 - ☐ Output saturate on overflow (wrap if not selected)
 - ☐ Output round towards nearest on underflow (truncate if not selected)
 - ☐ Reset port
 - ☐ Enable port
- Four buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

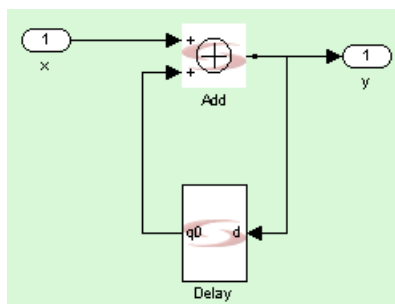
Pole Value

Determines how the block is implemented. The pole value must be between 0 and 1.

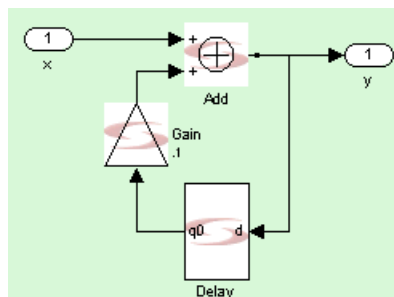
The following figure shows implementations with different pole values. A value of 0 is implemented as feedthrough without integration. A value of 1 is full integration, where the pole on the unit circle causes instability for DC frequencies. With a value of .9, you get a leaky integration, because the pole close to the unit circle causes high gain close to DC frequencies. With a value of .1, you get a very leaky integration, with very little distinction between different signal gains.



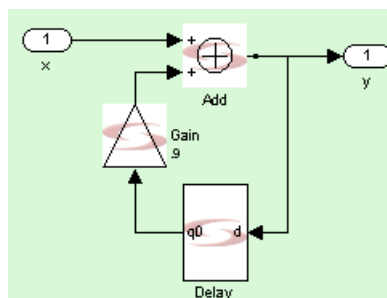
Pole Value = 0



Pole Value = 1



Pole Value = .1



Pole Value = .9

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round towards nearest on underflow

Determine how overflow and underflow are treated. These options are only available when you set Output Format to Specify.

Outputsaturate on overflow	Saturates or wraps the overflow; see Overflow Saturation Options, on page 585 .
-------------------------------	---

Output round towards nearest on underflow	With the option enabled, the software uses the Nearest algorithm to round the underflow; when disabled, it uses the Floor (Truncate) algorithm. See Underflow Rounding Options, on page 585 for details.
--	--

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

SMC Inverter

Calculates the inverse (one's complement) of the input.

Library

Symphony Model Compiler [Math Functions](#)

Description



The Symphony Model Compiler Inverter block calculates the one's complement of the input, which is a bitwise inversion of the input. The block supports vector inputs.

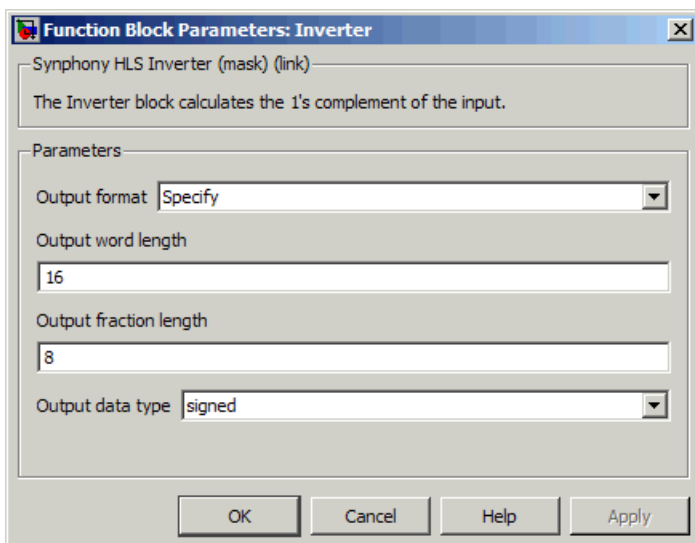
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Inverter Parameters



For descriptions of the parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

SMC Leading Zero Counter

Computes the number of leading zeros for an unsigned input.

Library

Symphony Model Compiler [Signal Operations](#)

Description

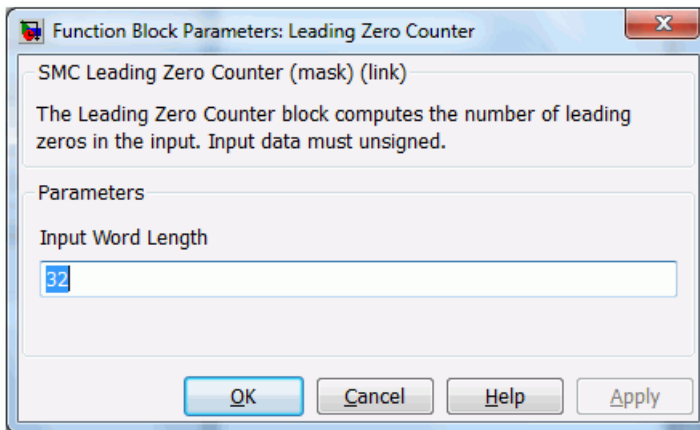


The Symphony Model Compiler Leading Zero Counter custom block computes the number of leading zeros for an unsigned input, which is the number of zeros starting from the MSB before it encounters one. This block is valuable when it is used in conjunction with a variable shifter to align numbers that always have a specific number of zeros before the most significant one. This is an essential computation for floating point and other operations that depend on number formats. The block can also be used to compute $\text{floor}(\log_2(\text{input}))$. Radix 4 decomposition during the first stage, followed by radix 2 decomposition for the subsequent stages can reduce the critical path to $\log_2(\text{input wordlength})-1$.

Latency

This block has no latency.

Leading Zero Counter Parameters



Input Word Length

Specifies the word length for the input.

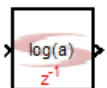
SMC Log

Calculates the natural logarithm of the input.

Library

Synphony Model Compiler [Math Functions](#)

Description

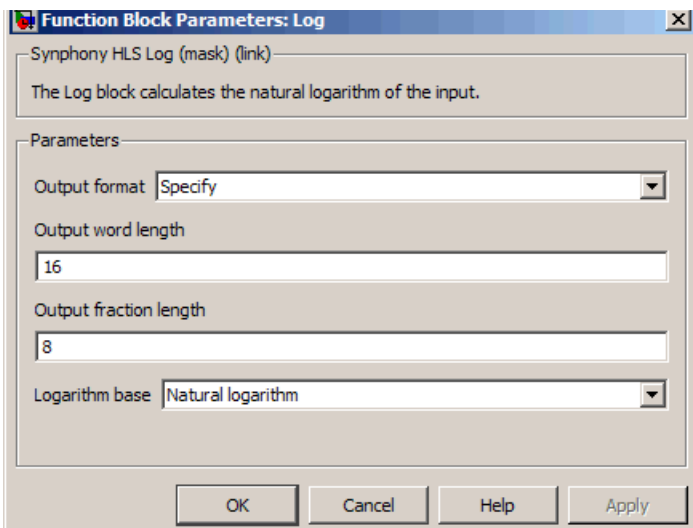


The Synphony Model Compiler Log block calculates the natural logarithm of the input. This implementation is based on a look-up table, the size of which is determined by the input fraction length and the output word length.

Latency

The latency of the Log block is 1.

Log Parameters



Output format, Output word length, and Output fraction length

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583 The output is signed. If the input is a signed data type and the input is negative, the Log block has a zero output, and you see a warning in the MATLAB command window.
Output word length	Output Word Length, on page 584 If Output format is Automatic, the word length is the same as the input. If Output format is Specify, the word length is as specified.
Output fraction length	Output Fraction Length, on page 584 If you set this option to a value greater than 8, the output accuracy is limited to 8 fraction bits and you see a warning message about the accuracy in the MATLAB command window. If Output format is Automatic, the fraction length is the same as the input. If Output format is Specify, the fraction length is as specified.

Logarithm base

Sets the logarithm used for calculation. You can set it to any of the following:

- Natural Logarithm
- Base 10
- Base 2

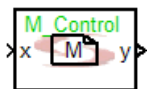
SMC M Control

Provides control logic using an M function.

Library

Synphony Model Compiler [Control Logic](#)

Description



The Synphony Model Compiler M Control block provides control logic that is written as an M function. You can use this block to implement complex control-intensive functions using the MATLAB M language. For details on writing the M functions, see [Using M Code Blocks, on page 858](#).

At every simulation tick, the block converts the fixed-point data at the block inputs to double, executes the M-control function on this double data, and then converts the output double data to fixed-point again for the rest of the model. This implementation improves simulation times.

You can use the matrix data type as input and output, and for internal operations. The M Control block infers and implement matrix operations like matrix multiplication and transpositions of matrices.

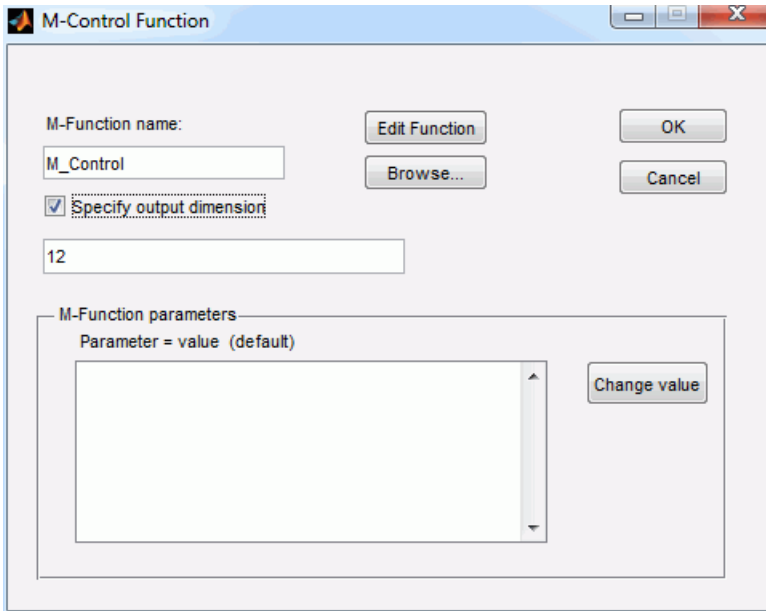
The M Control block treats all one-dimensional arrays as 1xN matrices. A vector input to the block is considered a 1xN matrix. Similarly, if the output of the M Control block is a one-dimensional array, it is output as a 1xN matrix.

Latency

Each output of this block can have either no latency or a latency of one.

Latency	Description
1	An output referencing one or more state-holding elements inferred from persistent variables has a latency of one.
0	In all other cases the output has no latency.

M Control Parameters



M-Control Function

M-Function name:

☒ Specify output dimension

M-Function parameters

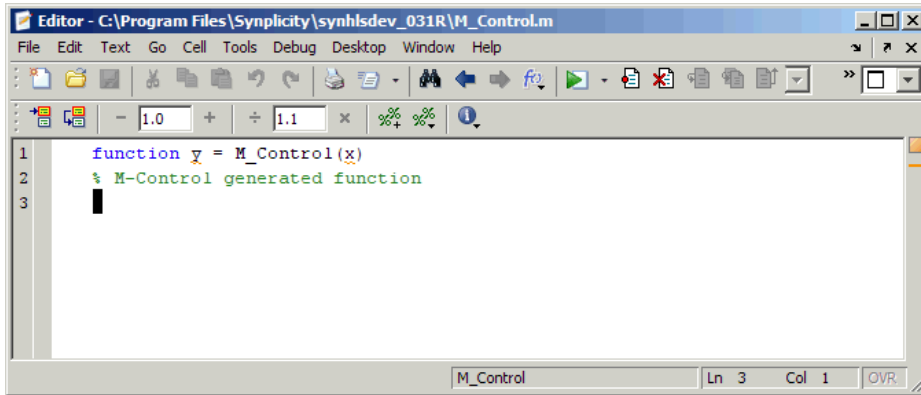
Parameter = value (default)

M Function Name

Specifies the name of the M function used to define the control function. The function name must not be the same as the design name, nor can it be a reserved MATLAB keyword. For information about writing M functions for this block, see [Using M Code Blocks, on page 858](#).

Edit Function

Opens a text window with the M function where you can type in or edit a function.



Browse

Lets you browse and search for the M function.

Specify output dimensions

Specifies the dimensions of the output ports.

- When it is disabled, the tool assumes that all output ports are scalar.
- When enabled, you can specify the dimensions in the field provided. To specify dimensions for multiple output ports, specify them in output port order, separated by commas. If the number of output ports does not match the number of dimensions specified, you get an error message.

The block can calculate the output port dimensions, but the information needed to do this might not be available early in the flow when it is required to propagate Simulink dimensions.

If you specify output port dimensions in this option, the block checks if this matches the dimensions received through propagation. If there is a mismatch, you get an error message about the mismatch.

M-Function Parameters

Displays the list of overridable parameters defined for the selected M function. The parameter variable and its value is displayed in each item. When you override a value with the Change Value button, the original value is shown in parentheses. For more information about overriding M function parameters, see [Overridable Parameters, on page 881](#).

Change Value

Lets you override the value of the selected parameter in the parameter list. See [Overridable Parameters, on page 881](#) for details about defining overridable parameters.

SMC Matrix Mult

Performs matrix multiplication of a two-input matrix signal.

Library

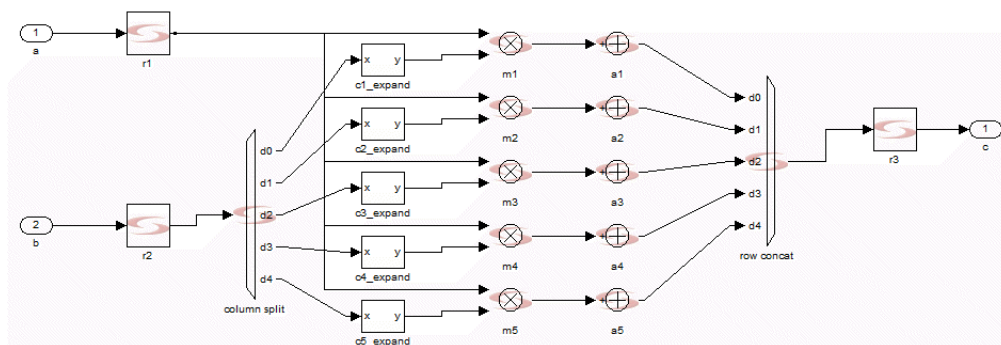
Symphony Model Compiler [Math Functions](#)

Description



The Matrix Mult block implements a full matrix multiplier. Matrix multiplication takes the first $M \times N$ matrix and the second $N \times P$ matrix as input signals and outputs an $M \times P$ product. There is a common dimension to the inputs: the number of columns of the first input matrix is always equal to the number of rows in the second input matrix. The inputs can be a $[M \times N]$, $[1 \times N]$, or $[N \times 1]$ matrix. The SMC Mult block also accepts matrix inputs, but its output is an element by element multiplication of the input matrices, not a matrix multiplication.

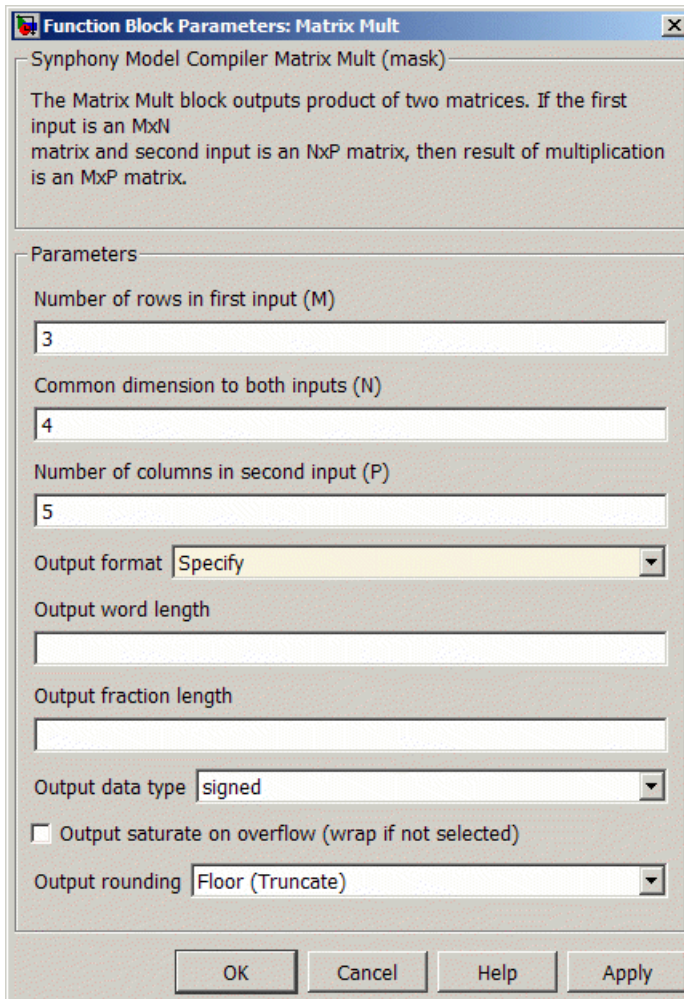
This block is a custom block, built as shown below. See [Primitives and Custom Blocks](#), on page 800 for a definition of custom blocks.



Latency

The default latency is 0.

Matrix Mult Parameters



Function Block Parameters: Matrix Mult

Symphony Model Compiler Matrix Mult (mask)

The Matrix Mult block outputs product of two matrices. If the first input is an $M \times N$ matrix and second input is an $N \times P$ matrix, then result of multiplication is an $M \times P$ matrix.

Parameters

Number of rows in first input (M)

Common dimension to both inputs (N)

Number of columns in second input (P)

Output format

Output word length

Output fraction length

Output data type

☐ Output saturate on overflow (wrap if not selected)

Output rounding

OK Cancel Help Apply

Number of rows in first input (M)

Specifies the number of rows in the first input matrix signal. If the first input matrix is $M \times N$, M is the number of rows in the first input.

Common dimension to both inputs (N)

Specifies the number of columns for the first input or the number of rows in the second input signal. If the first input matrix is $M \times N$ and the second is $N \times P$, N is the common dimension.

Number of columns in second input (P)

Specifies the number of columns in the second input matrix signal. If the first input matrix is $M \times N$ and the second is $N \times P$, P is the number of columns in the second input.

Output format

Specifies the precision and bitwidth of the output. The quantization is applied to all final adder in the Matrix Mult block.

- Full Precision
The output is a full-precision multiply and add operation inferred from the *a* and *b* input formats and propagated through all the internal operations.
- Specify
Lets you specify various quantization parameters, like output word length, fraction length, and so on.

Output word length, Output fraction length, Output data type

These options become available when you set Output quantization rule to Specify. For descriptions of these parameters, see the following:

Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturation on overflow, Output rounding

Determine how overflow and underflow are treated in the pruning of bits during internal operations. These options become available when Output format is set to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow, disable the option. See Overflow Saturation Options, on page 585 for details.
-----------------------------	--

Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.
---------------------------	--

Example

See [Example: 2-D DCT Using Matrix Data Types, on page 698](#) for an example that uses the Matrix Mult block.

SMC Mealy State Machine

Provides control logic where the output depends on the input and an internal state vector.

Library

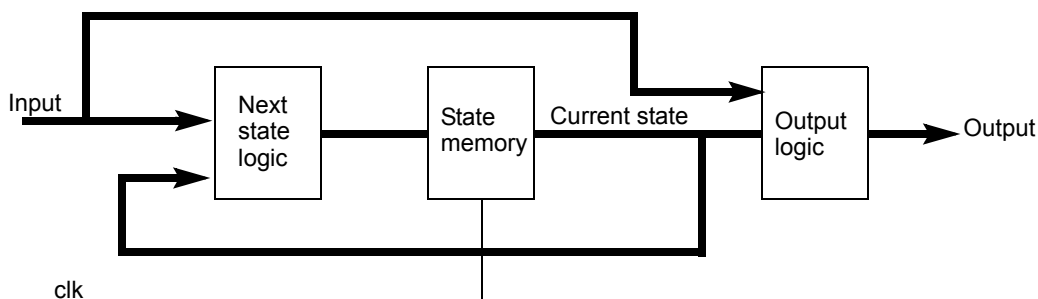
Synphony Model Compiler [Control Logic](#)

Description



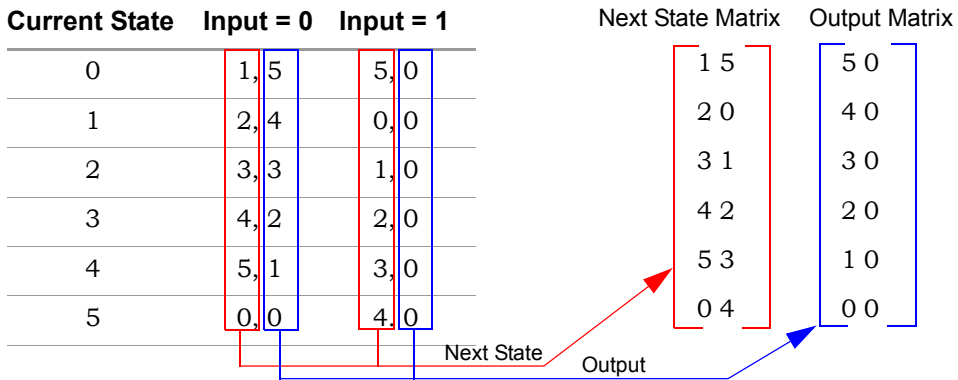
The Synphony Model Compiler Mealy State Machine block provides control logic to implement a Mealy state machine, where the output is a function of the present state and the input.

Mealy State Machine Diagram



Deriving Next State and Output Matrices

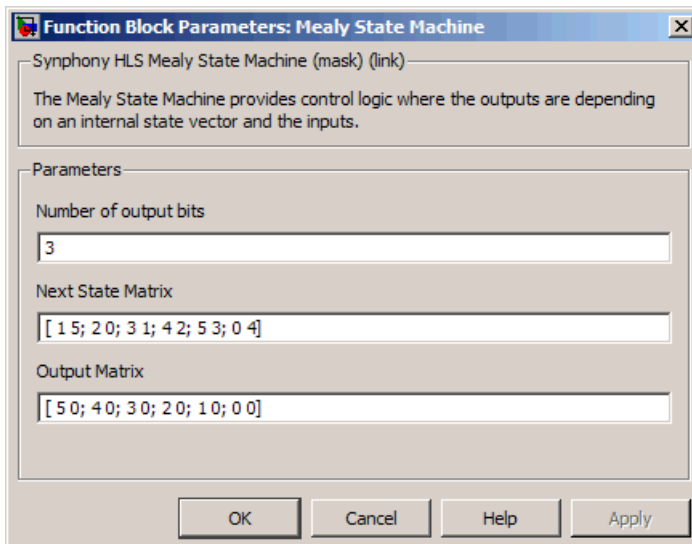
You configure the block by providing the next state and output matrices, which are defined by the next state/output table for the state machine. The rows of the matrices correspond to the current state, and the columns correspond to the input value. For example:



Latency

This block has no latency.

Mealy State Machine Parameters



Number of output bits

Each output bit is a control bit, calculated by the transformations defined below.

Next State Matrix

Defines state transition rules for the state machine. The number of rows in this matrix is equal to the current state. The number of columns is equal to the number of possible inputs. An input must be an unsigned integer between 0 and <number of columns> - 1. For each state-input pair, Next State Matrix defines what the next state should be.

Output Matrix

This is similar to Input matrix, but it defines the output for each state-input pair. You must be able to represent the values in the Output Matrix by the Number of Output Bits parameter.

SMC MinMax

Determines the minimum, maximum, or minimum and maximum of two inputs.

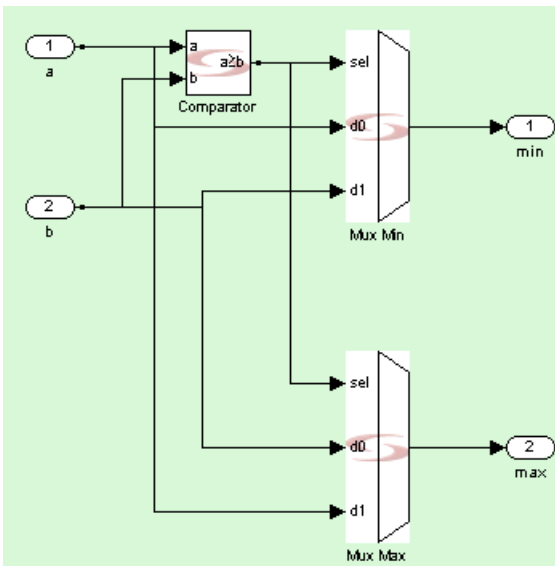
Library

Synphony Model Compiler [Math Functions](#)

Description



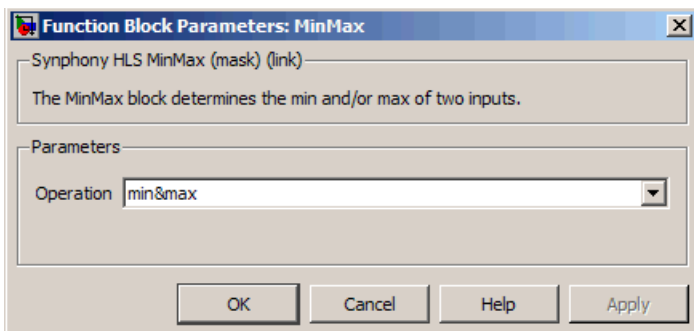
This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) determines the minimum, maximum, or minimum and maximum of two inputs.



Latency

This block has no latency.

Minmax Parameters



Function

Determines which operation is performed on the inputs. The icon changes to reflect your choice.

- min&max determines the minimum and maximum of the two inputs.
- min specifies the minimum of the two inputs.
- max specifies the maximum of the two inputs.

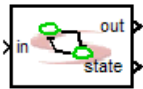
SMC Moore State Machine

Provides control logic where the outputs depend on the current state.

Library

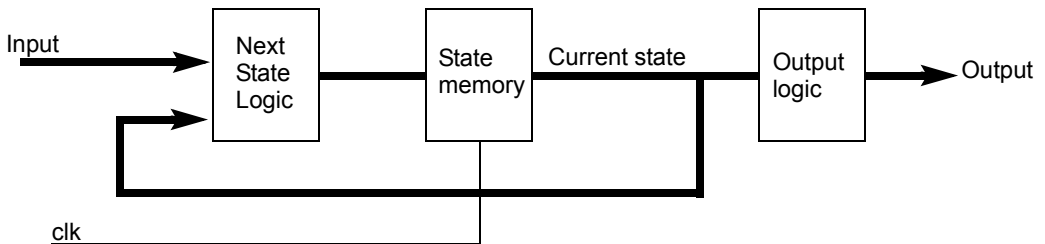
Synphony Model Compiler [Control Logic](#)

Description



The Synphony Model Compiler Moore State Machine block provides control logic where the output depends on the state variable.

Moore State Machine Diagram



Deriving a Next State Matrix and Output Array

You configure the block by providing the next state matrix and an output array. They are derived from the next state/output table for the state machine. The rows of the matrices correspond to the current state, and the columns correspond to the input value. The output array has only one value, because the input value does not affect the output.

For example:

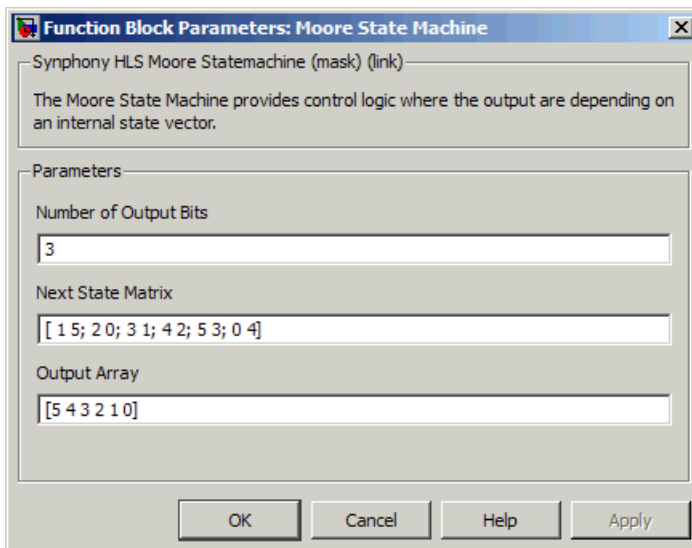
State	Input = 0	Input = 1	Output	Next State Matrix	Output Matrix
0	1	5	5	1 5	5
1	2	0	4	2 0	4
2	3	1	3	3 1	3
3	4	2	2	4 2	2
4	5	3	1	5 3	1
5	0	4	0	0 4	0

Diagram illustrating the state transitions and outputs for the SMC Moore State Machine. The state vector (0 to 5) is shown. The Next State Matrix and Output Matrix are also shown. A red arrow labeled "Next State" points from the state vector to the Next State Matrix. A blue arrow labeled "Output" points from the state vector to the Output Matrix.

Latency

This block has no latency.

Moore State Machine Parameters



Function Block Parameters: Moore State Machine

Synphony HLS Moore Statemachine (mask) (link)

The Moore State Machine provides control logic where the output are depending on an internal state vector.

Parameters

Number of Output Bits
3

Next State Matrix
[1 5; 2 0; 3 1; 4 2; 5 3; 0 4]

Output Array
[5 4 3 2 1 0]

OK Cancel Help Apply

Number of output bits

Each output bit is a control bit, calculated by the transformations defined below.

Next State Matrix

Defines state transition rules for the state machine. The number of rows in this matrix is equal to the number of states. The number of columns is equal to the number of possible inputs. An input must be an unsigned integer between 0 and <number of columns> - 1. For each state-input pair, Next State Matrix defines what the next state should be.

Output Array

This array has only value per state, because the input does not affect the output (see [Deriving a Next State Matrix and Output Array, on page 369](#)). The values in the array are separated by spaces. You must be able to represent the Output Array values by the Number of Output Bits parameter.

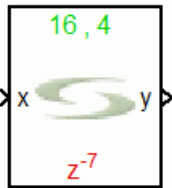
SMC Moving Average Filter

Implements a hardware-efficient moving average filter.

Library

Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler Moving Average Filter custom block implements a moving average filter that computes the sum of the last N input samples, and is based on a sliding window in a hardware-efficient manner using the recurrence relation:

$$y(n) = \sum_{k=0}^{N-1} x(n-k) = y(n-1) + x(n) - x(n-N)$$

To compute the unscaled mean of N values, set gain to 1/N.

If the number of channels is greater than 1, then the block assumes that channels are commutated at the input. The output is serial in this case.

Moving Average Filter Flow Control

The Moving Average Filter block optionally provides the following flow control ports:

ssynci	The ssynci (source ssync) input can be forced high to reset the internal filter storage to zero.
ssynco	The corresponding output goes to high latency clock cycles after ssynci goes high. This is used to cascade multiple stages of the moving average filters.
srdyi	The srdyi (source ready) input port qualifies whether the input data in the current sample period is valid. An invalid input sample is indicated by srdyi going low.
srdyo	The srdyo (source ready) output port qualifies whether the current output sample is valid. An invalid output sample is indicated by srdyo going low.

For a multichannel operation, flow control signals are commutated in the same way as the data.

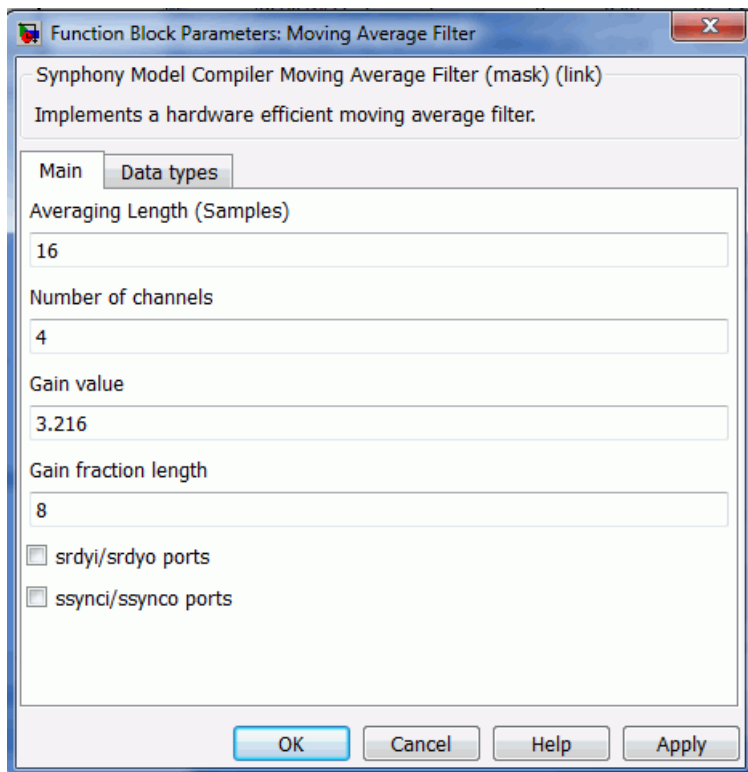
Block Annotation

Green annotation	The first number is the length of averaging, and the second the number of channels
------------------	--

Latency

The Moving Average Filter block has a latency of $3 + \text{number of channels}$.

Moving Average Filter Parameters



Main Tab

This tab sets parameters for average length of samples, number of channels, gain value and fraction length, and available ports.

Averaging Length

The length for samples of the sliding window from which the sum is computed.

Number of Channels

Specifies the number of channels for the moving average filter.

Gain Value

Specifies the gains for the input stage.

Gain Fraction Length

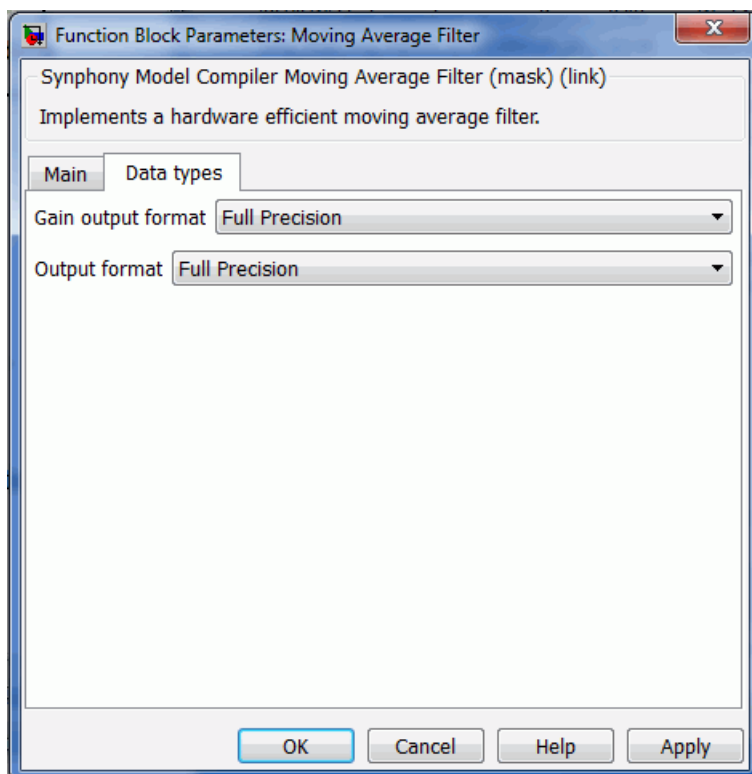
Specifies the gains for the fraction length.

srnyi/srdyo Ports

When enabled, adds the srnyi and srdyo ports to the block interface.

ssynci/ssynco Ports

When enabled, adds the ssynci and ssynco ports on the block interface.



Data Types Tab

This tab sets parameters for output format, word length, and fraction length.

Output format, Output word length, and Output fraction length

The gain output parameters refer to the data type after the input is multiplied by the specified gain value. The output format parameters refer to the data type of the feedback adder and the final output.

For descriptions of these output parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584. You can also specify it in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , and <code>syn_inp_dt</code> variables. If Inherit port is enabled, you can also use the <code>syn_inh_wl</code> , <code>syn_inh_fl</code> , and <code>syn_inh_dt</code> variables. The variables are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584. You can also specify it in terms of the variables <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , and <code>syn_inp_dt</code> . If Inherit port is enabled, you can also use the <code>syn_inh_wl</code> , <code>syn_inh_fl</code> , and <code>syn_inh_dt</code> variables. The variables are described in Special Variables, on page 588 .

Output Data Type

Determines the data type for the output.

Signed	See Output Data Type, on page 584 for details.
Unsigned	See Output Data Type, on page 584 for details.
Preserve	Preserves the input data type. If the input is signed, the output is also signed. If the input is unsigned, the output is also unsigned.
Inherit	Inherits the input data type from the inherit port. This option is only available when you enable Inherit port.

Output saturate on overflow, Output round on underflow

Determine how output overflow and underflow are treated. These options are available when you set Output Format to Automatic or Specify.

Output saturate on overflow	Saturates the overflow when the option is enabled and wraps the overflow when it is disabled. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	See Underflow Rounding Options, on page 585 for details about the rounding options available.

SMC Mult

Implements a full-precision multiplier.

Library

Symphony Model Compiler [Math Functions](#)

Description



The Symphony Model Compiler Mult block implements a full-precision multiplier. It computes the product of the data on its two input ports, and feeds it to the output port. The inputs can be of different word lengths, as determined by their drivers. The output word length is the sum of input word lengths. Similarly, the output fraction length is the sum of the input fraction lengths.

Automatic Scalar Expansion

If the block has one scalar input and one vector or matrix input, the tool automatically expands the scalar input to the size of the vector or matrix. You cannot have a combination of matrix and vector inputs.

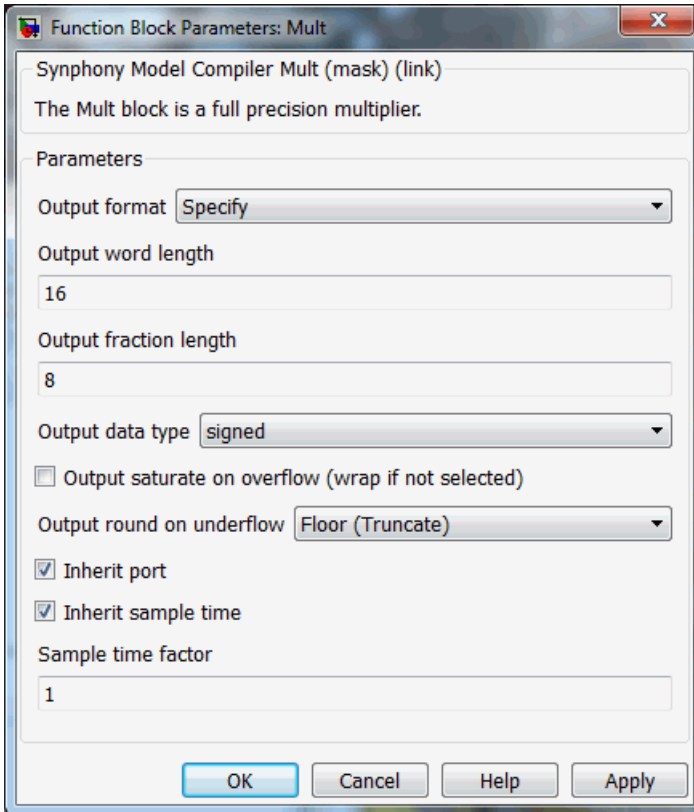
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Mult Parameters



Output format, Output word length, Output fraction length, and Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Data type	Output Data Type, on page 584

Output saturate on overflow, Output round on underflow

Determine how overflow and underflow are treated. These options become available when Output format is set to Specify.

Output saturate on overflow	Enable the option to saturate, and disable it to wrap the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

Inherit port

Determines whether the tool creates an inherit port for the block. This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Recast Output Variables, on page 427](#) for information about these variables.
- Use the inherit option to specify the Output data type. See [Data Type, on page 426](#) for a description of the option.

Inherit sample time

When enabled, inherits the sample period from the input signal.

Sample time factor

Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

SMC Mux

Implements a multiplexer of up to 2048 inputs.

Library

Synphony Model Compiler [Signal Operations](#)

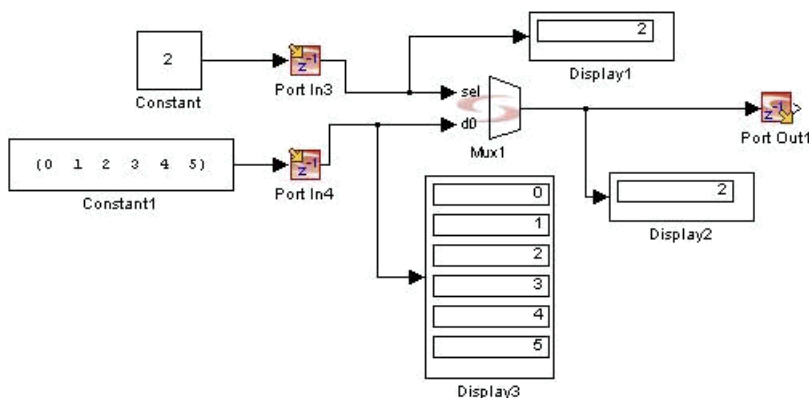
Description



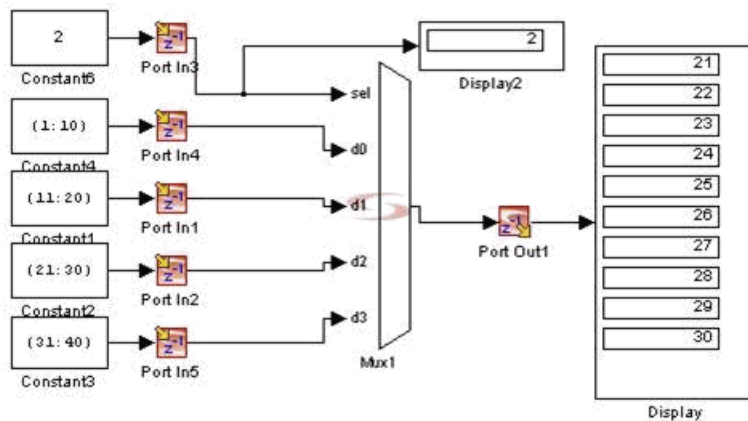
The Synphony Model Compiler Mux block implements a multiplexer of up to 2048 inputs. The sel input determines which of the data inputs gets multiplexed into the output.

Automatic Scalar Expansion

If the input to the Mux is one row vector, the sel line selects one of the elements of the vector. The following figure shows the second element of the input vector selected as output.



If the Mux input has many vectors, the sel line selects one of the vectors as the output. The next figure shows the second input vector selected as output.



Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Mux Parameters

Function Block Parameters: Mux

Synphony HLS Mux (mask) (link)

The Mux block is a multiplexer of up to 2048 inputs.

Parameters

Number of inputs
2

Output format Specify

Output word length
16

Output fraction length
8

Output data type signed

OK Cancel Help Apply

Number of inputs

Sets the number of inputs that are multiplexed. You can specify up to 2048 inputs. The inputs do not have to operate at the same sample rate as long as the `sel` line operates at the fastest clock (lowest sample time) of all the clocks entering the block, and the `sel` clock line is an integer multiple of any other clock entering the block. When the `sel` line is set to an out-of-bounds value, the Mux block outputs the first data line (`d0`).

If you specify a single input, the tool implements a single input mux. See [Single Input Mode Mux, on page 384](#) for further information.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

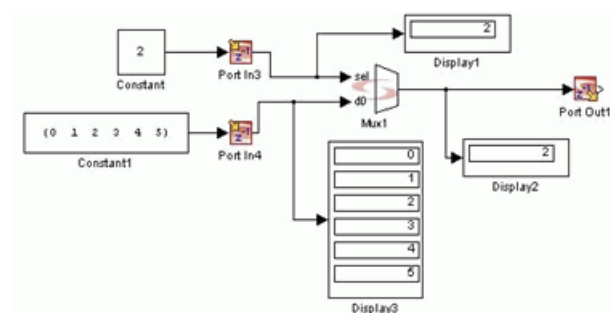
Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Single Input Mode Mux

When you specify a single input for the multiplexer, the tool implements a special multiplexer with a single data input and select input. The data input can be vector or matrix, and the select line must be scalar or vector. The following examples illustrate different scenarios.

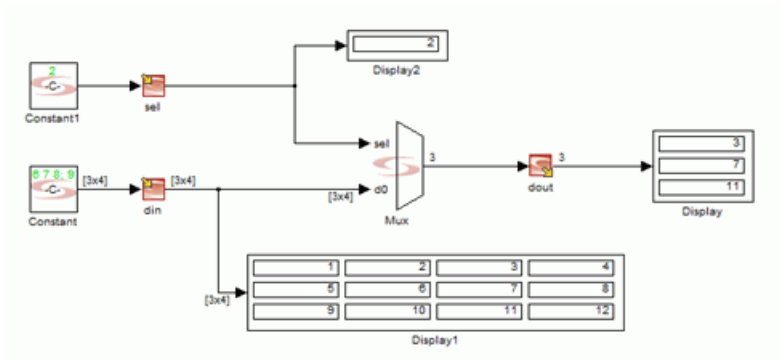
Scalar Select Line and Vector Data Input

If the data input to the mux is vector and the select input is scalar, the sel line selects one of the elements of the vector. This figure shows the third element of the input vector selected as output.



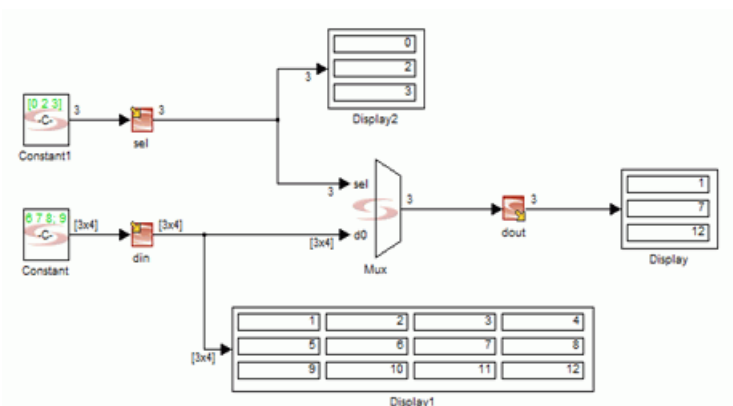
Scalar Select Line and Matrix Data Input

If the input to the mux is matrix and select input is scalar, the sel line selects one of the matrix columns. The following figure shows the third column of the input matrix selected as output. It shows the mux is equivalent to three (number of rows in matrix input) independent multiplexers, with the same select line going to each of them.



Vector Select Line and Matrix Data Input

If the input to the mux is matrix and the select input is vector, the *n*th element in the sel line selects one of the elements in the *n*th row of the matrix. The next figure shows the first element in the sel line (0) selects the first element in the first row, and the second element in the sel line (2) selects the third element in the second row, and so on. This example shows the mux is equivalent to three (number of rows in matrix input) independent multi-plexers, with an independent select line.



SMC Negate

Computes the two's complement (arithmetic negation) of an input.

Library

Symphony Model Compiler [Math Functions](#)

Description



The Symphony Model Compiler Negate block takes the two's complement of a signed input. For a signed value, this means that it multiplies the input by -1. This block supports vector input.

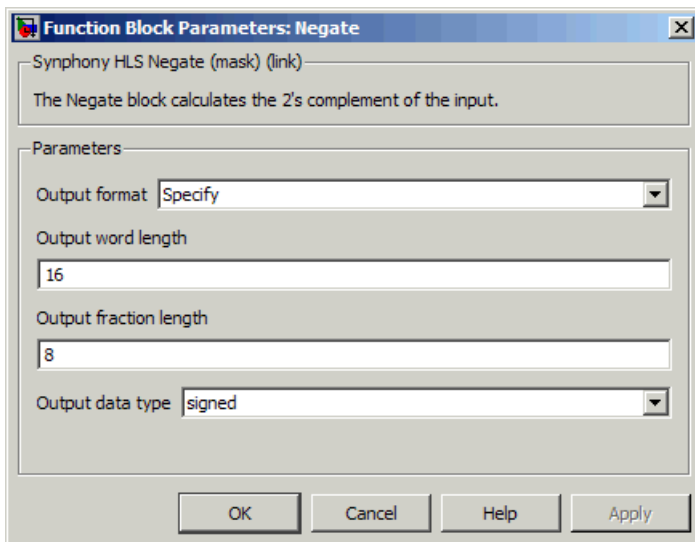
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Negate Parameters



For descriptions of the parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

SMC Out

Allows you to add an out port to a subsystem to a Symphony Model Compiler design.

Library

Symphony Model Compiler [Ports & Subsystems](#)

Description



The Symphony Model Compiler Out block provides an out port for a subsystem in a Symphony Model Compiler design. For details about this block, refer to the Simulink documentation for the Outport block in the Simulink Ports & Subsystems library.

Latency

This block has no latency.

SMC Parallel FIR

Implements a parallel input FIR filter.

Library

Synphony Model Compiler [Filtering](#).

Description



The Synphony Model Compiler Parallel FIR custom block implements high throughput FIR filters for processing Gsamples/sec. This block can accept N number of user-specified input values in parallel and produces N output values for each cycle that delivers throughput of $N \times \text{sample-rate}$. For example, suppose the clock frequency is 400 MHz and the number of parallel inputs is 32, then the filter throughput is 12.8 Gsamples/sec. You must provide the coefficients for the filter as a vector size equal to the number of taps at the w input of the block. The parallel FIR cannot internally register or store the coefficients.

Parallel FIR Flow Control

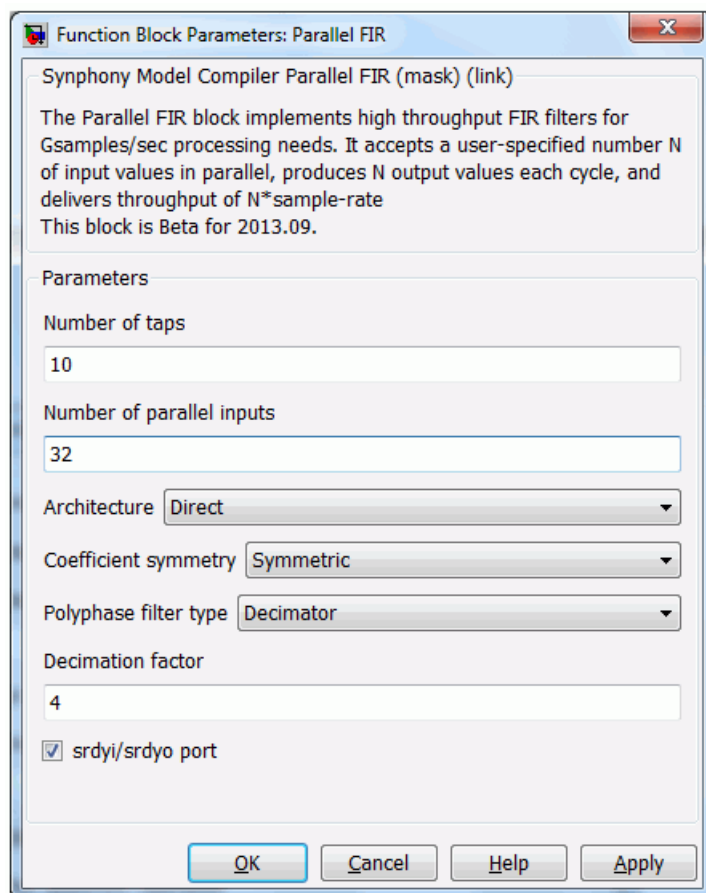
The Parallel FIR block provides the following optional flow control ports:

srdyi	The srdyi (source ready) input port determines whether the input data in the current sample period is valid. An invalid input sample is indicated by srdyi going low.
srdyo	The srdyo (source ready) output port determines whether the current output sample is valid. An invalid output sample is indicated by srdyo going low.

Latency

The latency for the Parallel FIR block is annotated on the mask, and depends on the options you select for the mask as well as the number of taps.

Parallel FIR Parameters



Number of taps

Specifies the number of taps for the FIR filter. The vector size of the w input must be equal to the number of taps.

Number of parallel inputs

Number of parallel data inputs. The vector size of the x input must be equal to the number of parallel inputs.

Architecture

Specifies the FIR architecture. You can select either the Direct or Systolic option. If you select the Systolic architecture, the coefficient symmetry options are not available. The filter is always implemented as an asymmetric filter.

Coefficient symmetry

Use this option only if you select the Direct architecture. You can specify the type of symmetry for the filter as follows:

- None specifies that symmetry is not inferred and the dimension of the w port is same as the number of taps.
- Symmetric infers pre-adders in the implemented FIR structure, such that the second half of the coefficients is same as the first half. For this option, the w input has a dimension equal to the $\text{ceil}(\text{Number of taps}/2)$.
- Antisymmetric infers pre-subtractors in the implemented FIR structure, such that the second half of the coefficients is the negative of the first half. For this option, the w input has a dimension equal to the $\text{ceil}(\text{Number of taps}/2)$.

Polyphase filter type

The following options are available:

- None implements a single-phase parallel FIR. For this option, the output dimension is the same as the input dimension.
- Decimator implements a polyphase decimator. The output has the same clock as the input and the output dimension is equal to $\text{ceil}(\text{number of parallel inputs} / \text{decimation factor})$. Decimators support the following decimation factors:
 - Number of parallel inputs / Decimation factor is an integer
 - Decimation factor / Number of parallel inputs is an integer

Decimation factor

Specifies the decimation factor for the polyphase decimators.

srnyi/srdyo port

Determines if the flow control ports are available.

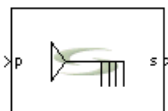
SMC Parallel to Serial

Implements a data packet splitter that divides the parallel data word at the input into small serial data packets in the order specified.

Library

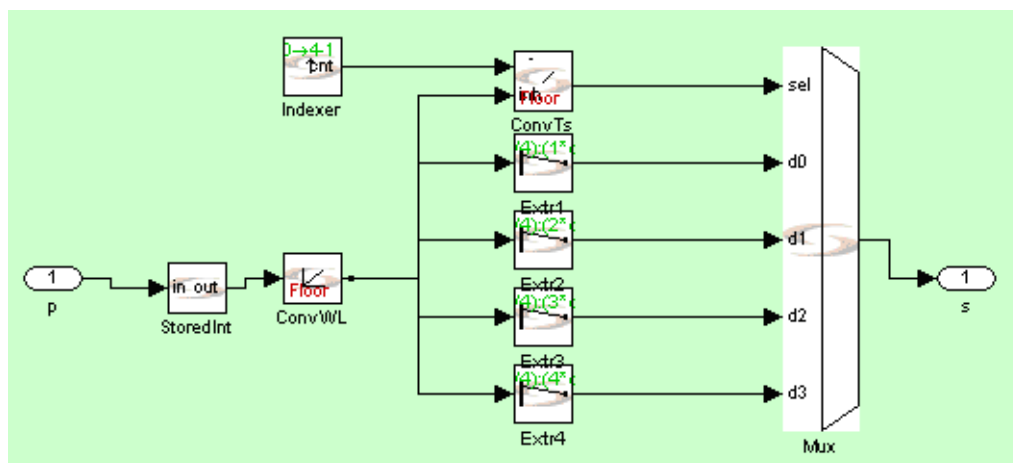
Synphony Model Compiler [Signal Operations](#)

Description



The Parallel to Serial block splits parallel input data into serial data packets for the output. You can specify the number of packets and the order of the packets at the output. As this block splits each input into multiple packets, the sampling rate at the output increases.

This block is a custom block. (See [Primitives and Custom Blocks](#), on [page 800](#) for a definition.) The following figure shows how the block is modeled:



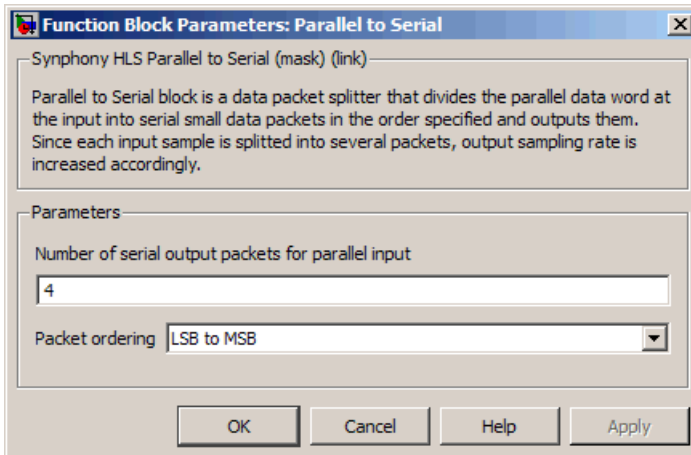
Icon Annotations

The icon for this block displays the following information:

Latency

Zero latency

Parallel to Serial Parameters



Number of serial output packets for parallel input

Specifies the number of serial output packets. As each input is being split into multiple packets, the output sampling rate increases.

Packet ordering

Determines the order of the data packets at the output.

- MSB to LSB sets the output order from the most significant to the least significant bit.
- LSB to MSB sets the output order from the least significant to the most significant bit.

Data format

This block produces output data as an unsigned integer with word length equal to packet size. If the most significant packet data is shorter than the serial packet size, this block 0-extends it from the left. If the total number of bits spanned by the output serial packets is shorter than the input word length, the block crops the excess bits at the input.

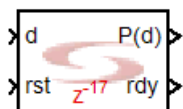
SMC Permutation

Shuffles the incoming data according to a specified permutation vector.

Library

Symphony Model Compiler [Memories](#)

Description



The Symphony Model Compiler Permutation block shuffles the incoming data according to a specified permutation vector, and produces a streaming output with a delay equal to $1 + \text{<minimum possible delay>}$. The software sets the order of the outgoing data stream by presenting a permutation of the incoming data stream $d[i]$.

When rst is 1, the software clears the buffer and resets the frame boundary. When it is used to perform blockwise permutation of a streaming signal, rst must be only applied to the first block. The rdy signal is the rst signal delayed by the input-to-output latency. You can use the rdy signal for synchronization and/or latency measurement.

Constant Propagation

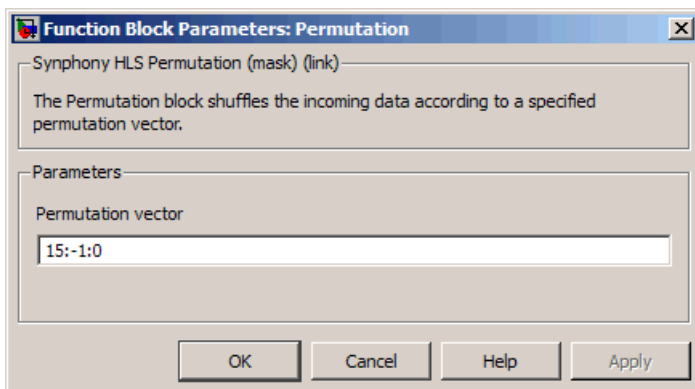
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Icon Annotations

The icon for this block displays the following information:

Latency For permutation $(P_0 P_1 \dots P_n)$, the latency is $\max(P_i - i) + 2$.

Permutation Parameters



Permutation vector

Resets the order of the incoming data stream samples by presenting a permutation vector of the data stream slot numbers 0 to N.

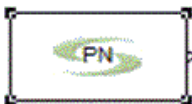
SMC PN Sequence Generator

Generates a sequence of pseudorandom (PN) binary numbers using a linear-feedback shift register (LFSR).

Library

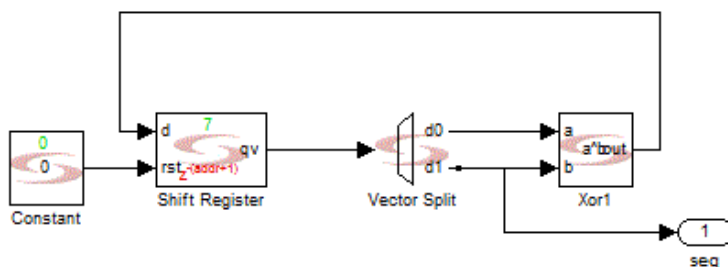
Synphony [Communications](#)

Description



The PN Sequence Generator block generates a sequence of pseudorandom (PN) binary numbers using a linear-feedback shift register (LFSR). The Generator polynomial parameter configures the LFSR.

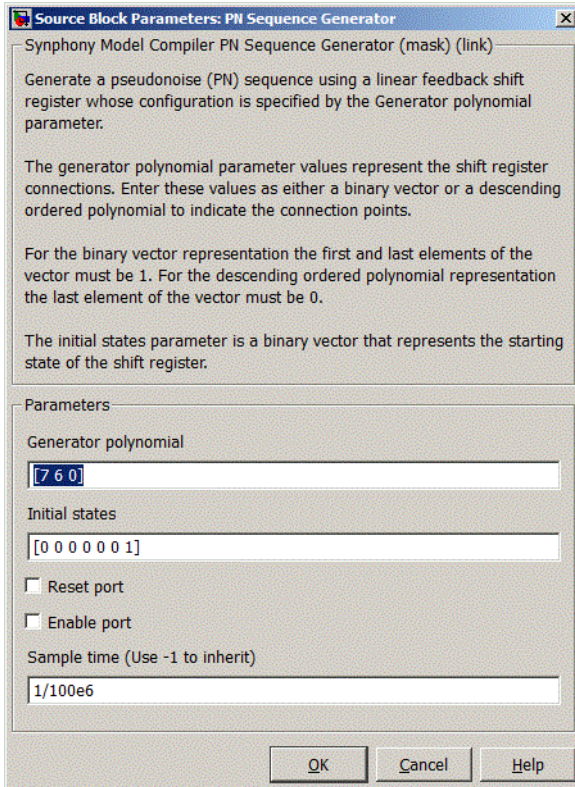
The PN Sequence Generator block implements LFSR using a simple shift register generator (SSRG, or Fibonacci) configuration. As the generator polynomial is a primitive polynomial the constant and the leading terms in the polynomial must be 1. This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition). The following figure shows the internal modeling:



Latency

This block has no latency.

PN Sequence Generator Parameters



Generator polynomial

Represent the shift register connections. Use either of the following to specify them:

- A binary vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. The length of this vector is one more than the degree of the generator polynomial, and the entry is 1 if there is a connection tap for the corresponding power and zero otherwise.

- A vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be 0.

For example, [1 1 0 0 0 0 1] and [7 6 0] represent the same polynomial, $p(z) = z^7 + z^6 + 1$.

Initial states

Specifies the initial value of the registers. It is a binary vector and must satisfy the following criteria:

- The length of the Initial states vector must equal the degree of the generator polynomial.
- At least one element of the Initial states vector must be 1 in order to generate a non-zero sequence.

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

With a high reset signal, there are no valid outputs for clock cycles. Hence, there is a one cycle discontinuity in the output when reset is applied.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you specify reset or enable ports.

SMC Port In

Defines inputs for the DSP design to be implemented in RTL.

Library

Synphony Model Compiler [Ports & Subsystems](#)

Description



The Synphony Model Compiler Port In block defines an input to the design to be implemented in RTL. Each input to the subsystem implemented by the Synphony Model Compiler blockset must be defined with a Port In block. This block quantizes the input data by the specified sample clock and word precision. It can also capture data passing through it and store it for the RTL testbench.

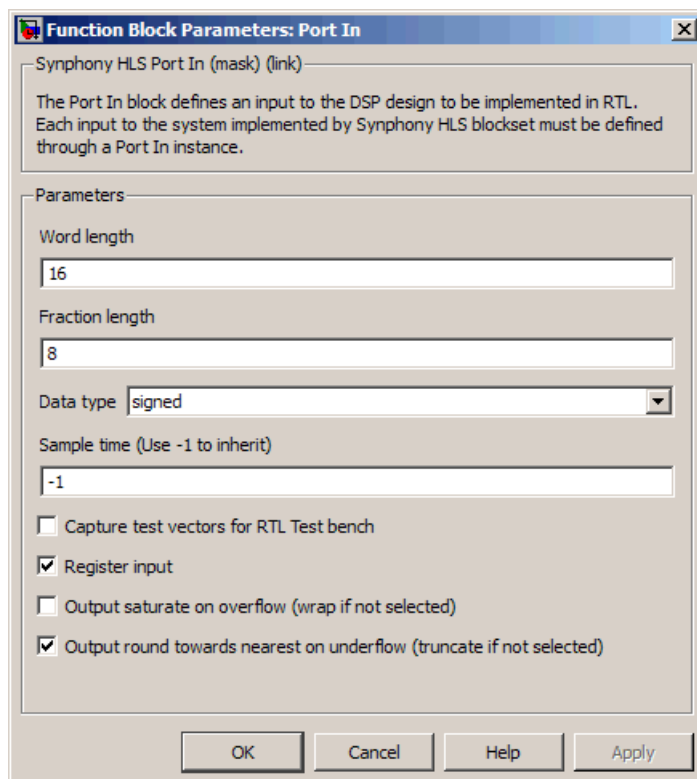
If you have matrix input, the generated RTL includes an input port for each element of the matrix, using the following standard nomenclature:

`<Port_In_Name>_e_<Row_Index>_<Column_Index>`

All matrix elements are written to one file in the same order as the ports defined. For each port, matrix elements are written row-wise.

Do not use this block to define the boundaries of a subsystem. Use the Synphony Model Compiler In block instead (see [SMC In, on page 345](#)).

Port In Parameters



The dialog box titled "Function Block Parameters: Port In" contains the following information:

Synphony HLS Port In (mask) (link)

The Port In block defines an input to the DSP design to be implemented in RTL. Each input to the system implemented by Synphony HLS blockset must be defined through a Port In instance.

Parameters

Word length: 16

Fraction length: 8

Data type: signed

Sample time (Use -1 to inherit): -1

☐ Capture test vectors for RTL Test bench

☒ Register input

☐ Output saturate on overflow (wrap if not selected)

☒ Output round towards nearest on underflow (truncate if not selected)

Buttons: OK, Cancel, Help, Apply

Word length

Sets the total width when the software converts the analog input to a Fixed Point data type.

Fraction length

Determines the position of the binary point when the software converts the analog input to a Fixed Point data type.

Data type

The data type can be either signed (two's complement) or unsigned.

- signed specifies Two's complement signed representation, and sets the sign bit to the MSB. This format specifies that an n -bit binary number be interpreted as a value in the range $[-2^{(n-1)}, (2^{(n-1)})-1]$. Numbers with their most significant bit equal to 1 indicate a negative value,

which is obtained by subtracting 2^n from the unsigned value of the number. For example, if a is a signed 3-bit binary number, $a=110$ means $6 - 2^3 = -2$.

- unsigned specifies that an n -bit binary number be interpreted as a value in the range $[0, (2^n)-1]$. If a is an unsigned 3-bit binary number, $a=110$ means $1*2^2 + 1*2^1 + 0*2^0 = 6$.

Sample time

Defines the sample period of the input signal.

Capture test vectors

When enabled, each input captures the test vectors on the sample clock and saves them in a file. The software can then use this file when it generates RTL to create stimuli for the RTL design. The .dat files for the test vectors are stored in the `<modelFileDir>/test_vectors` directory.

Register input

When enabled, registers the input. With registered input, the block has a latency of 1. The registers are generated with an attached `syn_keep` directive, to instruct the Synplify Pro or Synplify Premier synthesis tools not to move these registers during retiming.

Output saturate on overflow, Output round towards nearest on underflow

Determine how output overflow and underflow are treated.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round towards nearest on underflow	If the option is enabled, the tool rounds the output to the closest upper value which is representable with the data type defined for the Port In block. If it is disabled, the tool truncates the output.

The following table shows the results of some saturation and rounding options.

	Input Data		Output Data		
	Decimal	Binary	WL_FL	Binary	Decimal
Rounding Off (truncating) Saturation Off (wrapping)	2.375	010.011	sfix5_En2	010.01	2.25
	128	010000000	sfix8_En0	10000000	-128
Rounding On Saturation Off	3.875	011.111	sfix5_En2	100.00	-4
	-2.875	101.001	sfix5_En2	101.01	-2.75
Rounding Off Saturation On	128	010000000	sfix8_En0	01111111	127
	2.375	010.011	sfix4_En2	01.11	1.75
Rounding On Saturation On	128.375	010000000.011	sfix9_En1	01111111.1	127.5
	3.875	011.111	sfix5_En2	011.11	3.75

SMC Port Out

Defines outputs for the DSP design to be implemented in RTL.

Library

Synphony Model Compiler [Ports & Subsystems](#)

Description



The Synphony Model Compiler Port Out block defines an output of the design to be implemented in RTL. Each output to the subsystem implemented by the Synphony Model Compiler blockset must be defined with a Port Out block. It can also capture data passing through it and store it for the RTL test bench.

If you have matrix output, the generated RTL includes an output port for each element of the matrix, using the following standard nomenclature:

`<Port_Out_Name>_e_<Row_Index>_<Column_Index>`

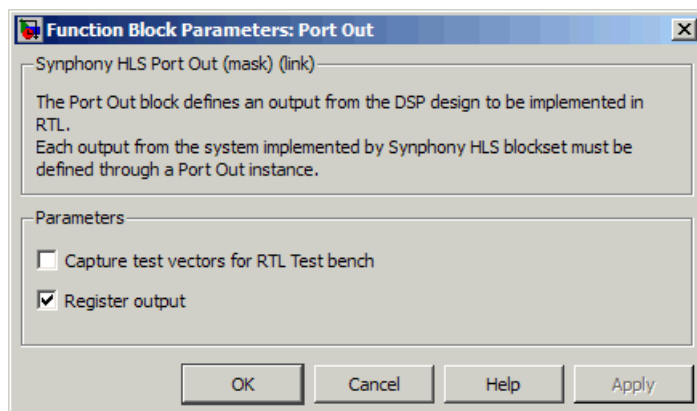
All matrix elements are written to one file in the same order as the ports defined. For each port, matrix elements are written row-wise.

Do not use this block to define the boundaries of a subsystem. Use the Synphony Model Compiler Out block instead (see [SMC Out, on page 388](#)).

Latency

If the block output is registered, it has a latency of one sample time.

Port Out Parameters



Capture test vectors

When enabled, each output captures the test vectors on the sample clock and saves them in a file. The software can then use this file when it generates RTL, to create expected results for the RTL design. The .dat files for the test vectors are stored in the <modelFileDir>/test_vectors directory.

Register output

When enabled, registers the output. With registered output, the block has a latency of 1. The registers are generated with an attached `syn_keep` directive, to instruct the Synplify Pro or Synplify Premier synthesis tools not to move these registers during retiming.

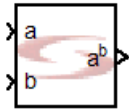
SMC Pow

Raises a value to the power of another value.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Pow block raises a value to the power of another value.

Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Modes of Operation

The Pow block has three modes of operation. The block icon reflects the operation mode.

Variable base and variable exponent a^b



The base (a) and exponent (b) of power operation are taken from the input ports of the block. In this mode, the tool can calculate integer powers of a fractional number. If the exponent has a fractional parts, fraction bits are ignored.

a: variable base, can be fractional

b: variable exponent, integer (fraction bits are ignored)

**Variable base
and constant
exponent
 a^{constant}**


The base of power operation (**a**) is taken from the input port and the exponent value (**constant**) is taken from the mask parameters of the block. In this mode the constant exponent must be an integer, but base can also take fractional values.

- **a**: variable base, can be fractional
- **constant**: constant exponent, integer

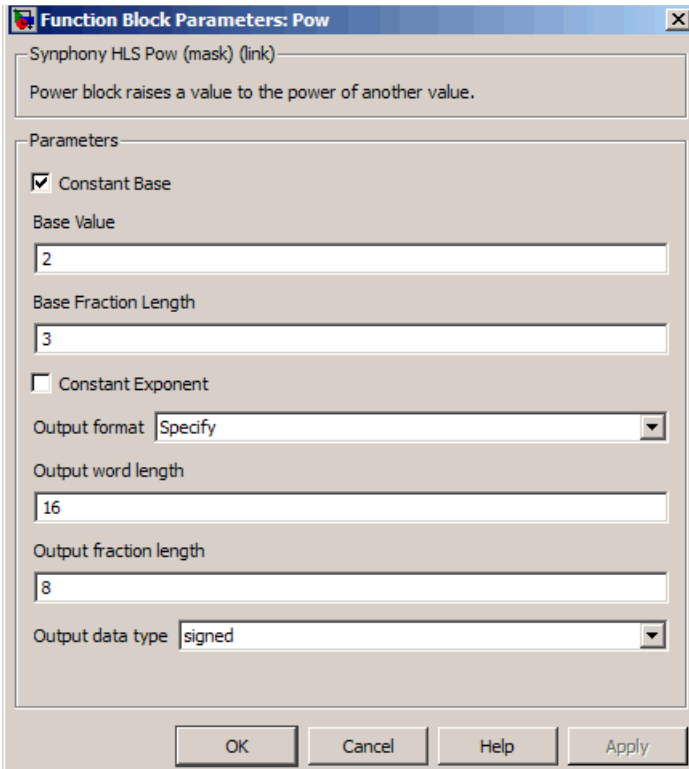
**Constant base
and variable
exponent
 constant^b**


The exponent of power operation (**b**) is taken from the input port and the base value is taken from the mask parameters of the block. Both the exponent and the base of power operation can be fractional numbers.

Since the number of output fraction bits can be infinitely long, the bit width of the output must be specified with the **Specify** option.

- **constant**: constant base, can be fractional
- **b**: variable exponent, can be fractional

Pow Parameters



The image shows a dialog box titled "Function Block Parameters: Pow". It contains a description of the block's function and a set of parameters. The description states: "Power block raises a value to the power of another value." The parameters section includes a checked checkbox for "Constant Base", a "Base Value" text field with the value "2", a "Base Fraction Length" text field with the value "3", an unchecked checkbox for "Constant Exponent", an "Output format" dropdown menu set to "Specify", an "Output word length" text field with the value "16", an "Output fraction length" text field with the value "8", and an "Output data type" dropdown menu set to "signed". At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

Function Block Parameters: Pow

Symphony HLS Pow (mask) (link)

Power block raises a value to the power of another value.

Parameters

☒ Constant Base

Base Value

2

Base Fraction Length

3

☐ Constant Exponent

Output format Specify

Output word length

16

Output fraction length

8

Output data type signed

OK Cancel Help Apply

Constant Base

When enabled, it lets you specify a base value and fraction length. See [Modes of Operation, on page 405](#) for descriptions of the operating modes.

Base Value

Specifies the constant base value. This is available when you enable Constant Base. See [Modes of Operation, on page 405](#) for descriptions of the operating modes.

Base Fraction Length

Specifies the fraction length for the constant base. This is available when you enable Constant Base. See [Modes of Operation, on page 405](#) for descriptions of the operating modes.

Constant Exponent

When enabled, it lets you specify an exponent value. See [Modes of Operation, on page 405](#) for descriptions of the operating modes.

Exponent Value

Specifies an exponent value for use by the operation.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

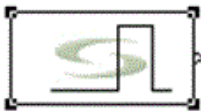
SMC Pulse Generator

Generates a single pulse.

Library

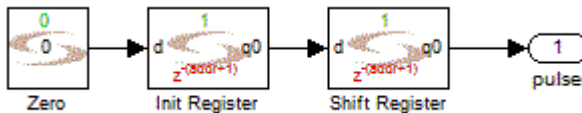
Synphony [Sources](#)

Description



The Pulse Generator block generates a single pulse. The generated pulse may be an impulse or a step (0 to 1 or 1 to 0).

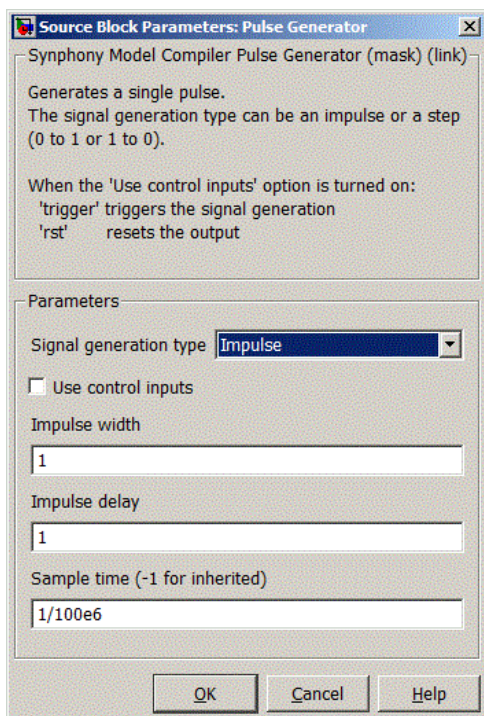
This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition). The following figure shows the internal modeling when signal generated is an impulse with use control inputs unchecked:



Latency

This block has zero latency.

Pulse Generator Parameters



Signal generation type

Specifies whether the output signal is an impulse or a unit step (0 to 1) or (1 to 0).

Use control inputs

When enabled, two additional input ports are available to the Pulse Generator block to perform the following functions:

Triggers	Triggers the signal generation
rst	Resets the output to zero till the next trigger arrives.

When both the signals occur simultaneously, the rst signal overrides the Trigger signal.

Impulse width

Specifies the width of the generated impulse in multiples of sample period when Signal generation type is set to impulse.

Impulse delay

Specifies the delay of the generated impulse in multiples of sample period when Signal generation type is set to impulse.

Step delay

Determines the delay of the generated impulse in multiples of sample period when Signal generation type is set to step.

Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you check Use control inputs.

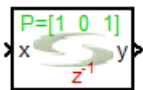
SMC Puncture

Removes user-specified bits from the input data stream.

Library

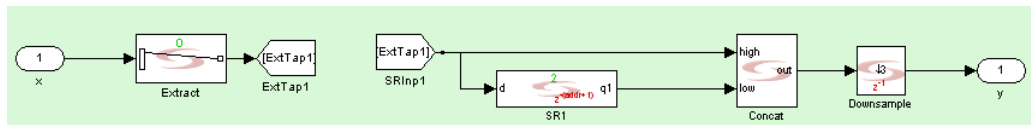
Symphony Model Compiler [Communications](#)

Description



The Puncture block removes the set of bits you specify from the input data stream. This block is commonly used in conjunction with a convolutional encoder ([SMC Convolutional Encoder, on page 106](#)) to implement punctured convolutional codes.

This block is a custom block. (See [Primitives and Custom Blocks, on page 800](#) for a definition.) The following figure shows how the block is modeled:

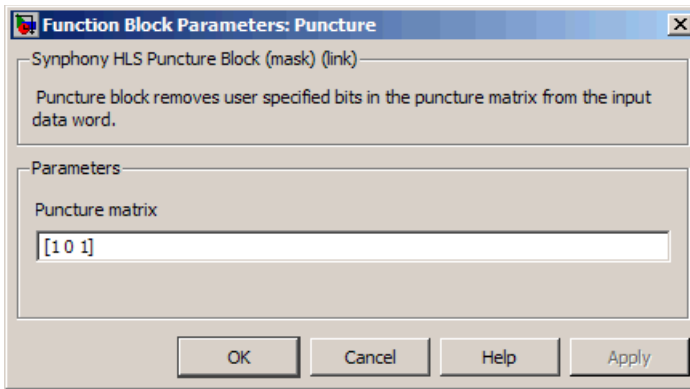


Icon Annotations

The icon for this block displays the following information:

Top Annotation	The upper annotation shows the puncture pattern set for the block.
Latency	This block has a fixed latency of 1.

Puncture Parameters



Puncture matrix

Determines the pattern of bits to be removed from the input data stream. Each row of the puncture matrix operates on a different bit in the input data word with last row corresponding to the LSB of the input data word.

Each 0 indicates a bit to be removed. For example, an input of UFix_3_0 and a puncture pattern of [1 0 1] results in the center bit being removed from the LSB of the input stream and a 2-bit punctured output of UFix_2_0. As no puncture patterns are specified for the remaining bits of input stream, the output stream does not convey any bit from unspecified bits of input data word.

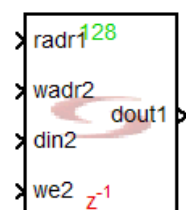
SMC RAM

Stores signals in an array with configurable read and write access ports.

Library

Synphony [Memories](#)

Description



The RAM (Random Access Memory) block implements a memory function through a storage array that has read and write access through ports. The ports can be configured for read, write, and read/write. For further details about RAMs, see [RAMs, on page 733](#).

The Synphony Model Compiler tool creates RAM memories that are fully synchronous, with write-first access mode. For information about data format, see [RAM Data Format, on page 415](#).

Automatic Scalar Expansion

Each individual RAM input can be either vector or scalar. If one input is vector, the tool automatically expands the other scalar inputs to the vector size. If more than one input is a vector, the vectors must be the same size.

Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

RAM Data Format

You must follow these data format rules for a RAM:

- The data type of the address must be an unsigned integer.
- All data inputs must have the same data type.
- The data type of each RAM output must be the data type of the inputs.
- All address input signals must have the same data type.
- Sample times within a port group (i.e. wadr, radr, wdin, we) must be the same.

RAM Initialization

You must initialize the RAM block. If you do not, you will get warning messages like the following when you run simulation:

```
Warning: block 'test_PPF_4x/PPF/RAM_chain/sampleRAM': Content of
address 511 unknown!
```

To suppress RAM warnings, specify the appropriate command at the MATLAB command line:

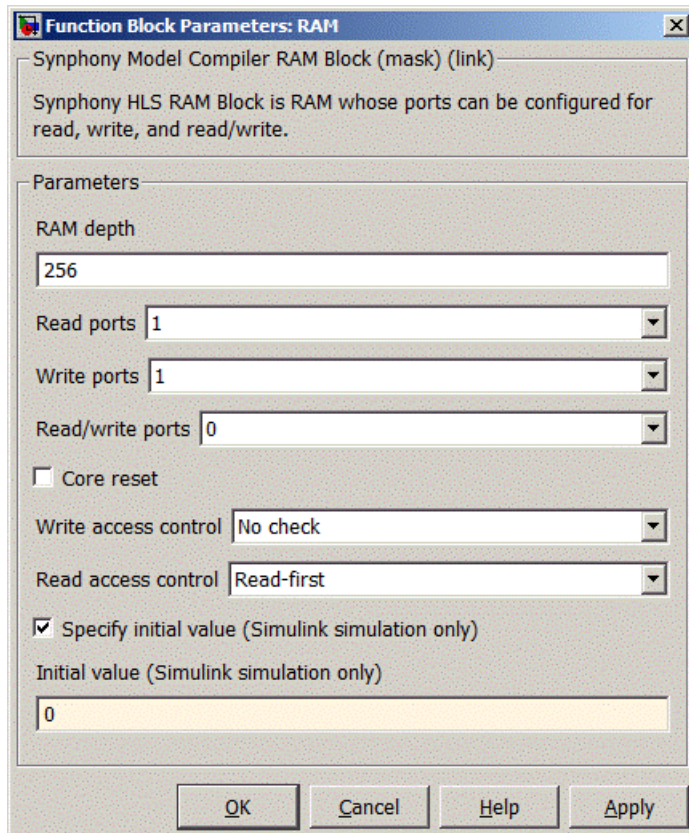
For unknown address warnings	<code>warning('off', 'SynHLS:Content_Unknown_RAM');</code>
For warnings about write clashes	<code>warning('off', 'SynHLS:Write_Clash_RAM');</code>

Icon Annotations

The icon for this block displays the following information:

Top Annotation	The annotation at the top of the RAM instance indicates the depth of the RAM in words.
Latency Annotation	The read and write of the RAM take one cycle (synchronous function).

RAM Parameters



The image shows a dialog box titled "Function Block Parameters: RAM". It contains a description of the block and a list of parameters. The parameters are: RAM depth (256), Read ports (1), Write ports (1), Read/write ports (0), Core reset (unchecked), Write access control (No check), Read access control (Read-first), Specify initial value (checked), and Initial value (0). The dialog box has OK, Cancel, Help, and Apply buttons at the bottom.

Function Block Parameters: RAM

Synphony Model Compiler RAM Block (mask) (link)

Synphony HLS RAM Block is RAM whose ports can be configured for read, write, and read/write.

Parameters

RAM depth
256

Read ports 1

Write ports 1

Read/write ports 0

☐ Core reset

Write access control No check

Read access control Read-first

☒ Specify initial value (Simulink simulation only)

Initial value (Simulink simulation only)
0

OK Cancel Help Apply

RAM depth

Sets the size of the RAM, in words. This value is annotated on the icon for this block.

Read ports

Sets the number of read ports. For additional information about the settings, see [Port Use in Different RAM Configurations, on page 737](#).

Write ports

Sets the number of read ports. For additional information about the settings, see [Port Use in Different RAM Configurations, on page 737](#).

Read/write ports

Sets the number of read ports. For additional information about the settings, see [Port Use in Different RAM Configurations, on page 737](#).

Core reset

When enabled, it initializes the RAM to all zeroes. When disabled, it does not initialize the RAM.

Reset port

This option becomes available when you enable Core Reset. When enabled, the block is implemented with a reset pin.

Enable port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal RAM. When the enable is low, the RAM output value and contents do not change.

Write access control

Determines the kind of logic for the write signal. You can set it to either of the following:

- No check
- Write prioritization
See [Write Access Control, on page 736](#) for details.

Read access control

Determines the kind of logic for the read signal.

- Read first
All read/write first operations operate as read-first.
- Read-write port, write first
Read/write first operate in write-first mode, while the read ports are read-first.
- Cross-port write first
All read and read/write ports operate as write-first. Read ports are sensitive to all the write ports at the same clock.

See [Read Access Control, on page 736](#) for details.

Specify initial value (Simulink simulation only)

When enabled, displays the Initial value field, where you can specify an initial value for the RAM.

Initial value (Simulink simulation only)

Specifies the initial value of the RAM block for Simulink simulation.

If you specify a single scalar value, it applies to all RAM locations. If you specify a vector initial value, ensure that the number of elements matches the depth of the RAM.

The initial value specified here is only used for Simulink simulation, and not in the generated RTL. To initialize RAMs for the generated RTL, you must explicitly load the initial values, according to the mechanism your environment supports. If you specify an initial value for simulation here, you could get warning messages during RTL generation if there are mismatches between Simulink and RTL simulation results.

SMC Ramp

Creates a ramp, based on increments derived from a port or a parameter.

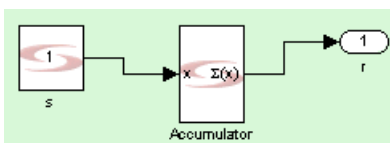
Library

Synphony Model Compiler [Sources](#)

Description



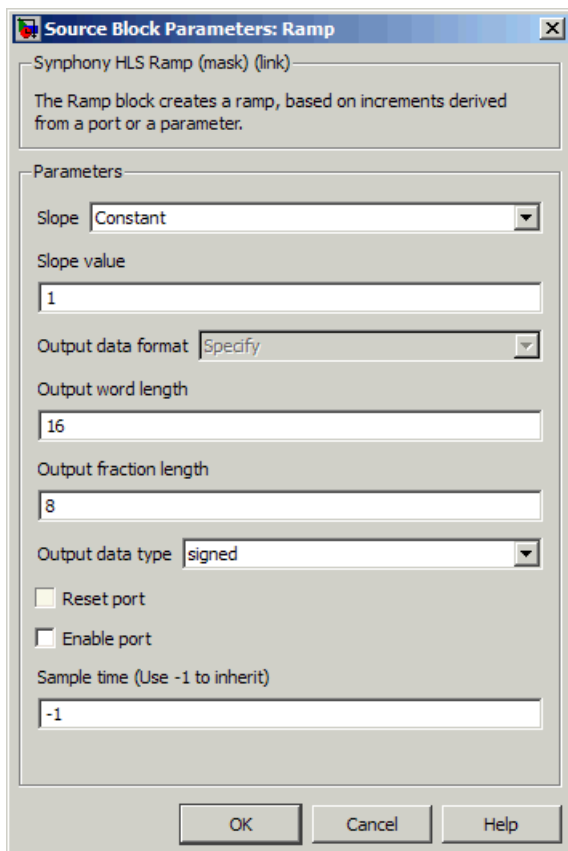
This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) generates a ramp signal using an accumulator with input increment that is determined by the value of the Slope option. The accumulator value wraps when overflow occurs.



Latency

This block has no latency.

Ramp Parameters



The dialog box titled "Source Block Parameters: Ramp" contains the following elements:

- A link: "Synphony HLS Ramp (mask) (link)".
- A description: "The Ramp block creates a ramp, based on increments derived from a port or a parameter."
- A "Parameters" section with the following controls:
 - "Slope" dropdown menu set to "Constant".
 - "Slope value" text input field containing "1".
 - "Output data format" dropdown menu set to "Specify".
 - "Output word length" text input field containing "16".
 - "Output fraction length" text input field containing "8".
 - "Output data type" dropdown menu set to "signed".
 - Two unchecked checkboxes: "Reset port" and "Enable port".
 - "Sample time (Use -1 to inherit)" text input field containing "-1".
- Buttons at the bottom: "OK", "Cancel", and "Help".

Slope

Determines whether the slope is derived from a port or a constant.

- Constant is a hard-coded slope value which is cast into the number format that you specify in other options in the dialog box.
- Port lets you specify a slope value dynamically via an input port. Selecting this option enables you to choose an automatic number format (Data format) that is inherited from the input port.

Slope value

Determines the rate of change for the generated signal, when you set Slope to Constant. The default value is 1.

Output data format

Determines the word size and data type of the output. Available options are determined by the value of Slope.

- Automatic calculates the output based on the input. The Ramp block uses the same size and type on the output as that driven on the input. This option is only available when Slope is set to Port.
- Specify lets you specify the size and data type using the Word Length, Fraction Length, and Data Type parameters.

Output word length, Output fraction length, and Output data type

For descriptions of these parameters, see the following:

Word length	Output Word Length, on page 584
Fraction length	Output Fraction Length, on page 584
Data type	Output Data Type, on page 584

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

Sample Time

Determines sample time when you set Slope to Constant and Reset Port and Enable Port are disabled. Use -1 to inherit. This option is not available if you specify reset or enable ports.

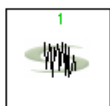
SMC Random

Creates a random integer of the requested word length.

Library

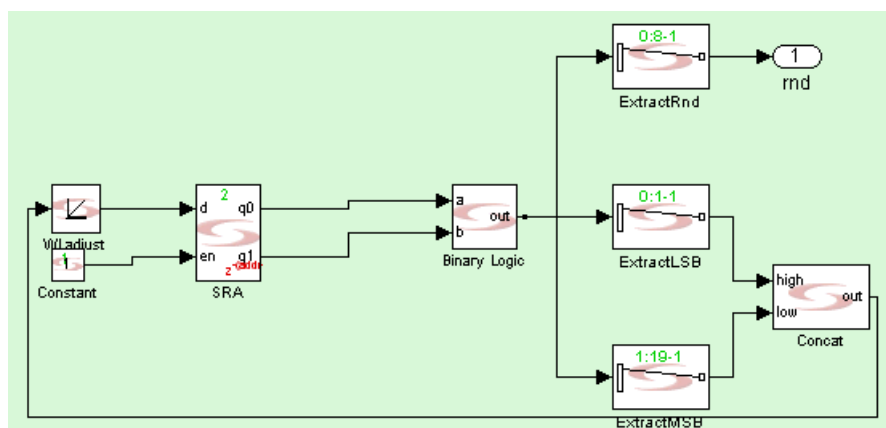
Symphony Model Compiler [Sources](#)

Description



This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) creates a random integer of the specified word length.

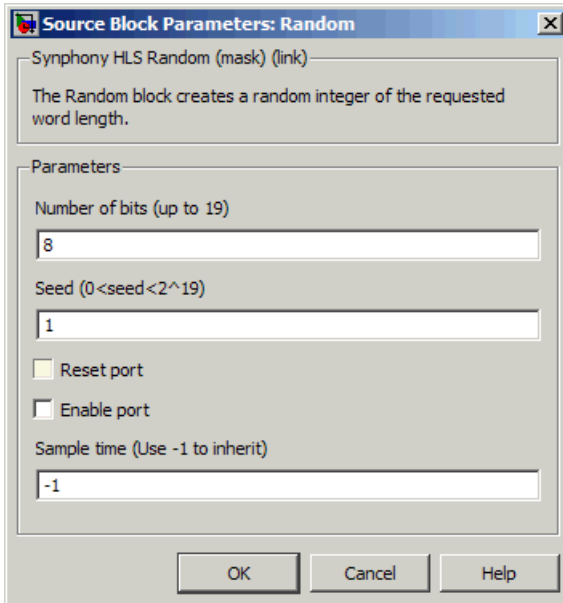
The following figure shows the internal construction of this block, without reset or enable ports:



Latency

This block has no latency.

Random Parameters



Number of bits

Specifies the length of the word, which in turn determines the size of the random integer. The maximum number of bits you can specify is 19.

Seed

Specifies the initial seed value for the random number generator. The format is an unsigned integer up to $2^{\text{(Number of Bits)}}$.

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you specify reset or enable ports.

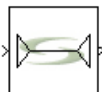
SMC Recast

Generates an output value, based on the requested data type you cast at the output.

Library

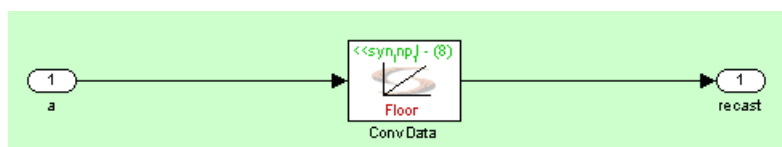
Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Recast block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for an explanation) for recasting the output value. The Recast block casts the input data to the specified output type. The block truncates or extends MSB bits if the specified output width is different than the input width. If the output is signed and you select a signed output data type, the block uses sign extension; otherwise the block extends the MSBs with zeroes. The Recast block can also use an inherit port. The inherited data format and/or input port data format can be used in arithmetic expressions when you specify the recast for the output

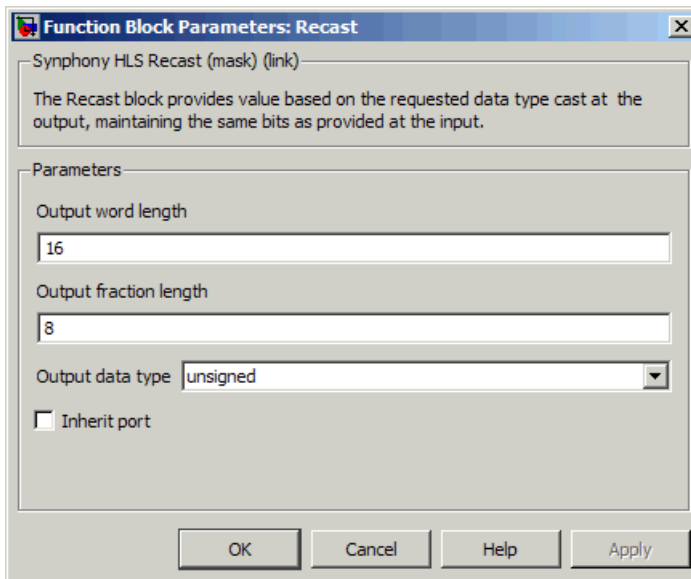
The following figure shows the internal construction of the Recast block:



Latency

This block has no latency.

Recast Parameters



Output word length

Determines the word length of the output in bits. This parameter is used together with Output fraction length. Given a word length WL, and a fraction length FL:

- The word bits go from WL-1 to 0
- The fraction bits go from FL-1 to 0
- Bit position WL-1 corresponds to the MSB.
- Bit position 0 corresponds to the LSB.

You can also specify the output word length in terms of the following variables `syn_inp_wl`, `syn_inp_fl`, and `syn_inp_dt`. If Inherit port is enabled, you can also use the `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` variables. The variables are described in [Recast Output Variables, on page 427](#).

If you change the word length, the block recasts the output as follows:

- If you shorten the word length, the block recasts the output by truncating the most significant bits as needed.
- If you increase the word length, the block extends the most significant bits as needed.

Output fraction length

Sets the fraction length of the output in bits. It is used with Output Word Length, as described above. You can also specify the output fraction length in terms of the variables `syn_inp_wl`, `syn_inp_fl`, and `syn_inp_dt`. If Inherit port is enabled, you can also use the `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` variables. The variables are described in [Recast Output Variables, on page 427](#).

Data Type

Determines the data type for the output.

- `signed` specifies Two's complement signed representation, and sets the sign bit to the MSB. This format specifies that an n -bit binary number be interpreted as a value in the range $[-2^{(n-1)}, (2^{(n-1)})-1]$. Numbers with their most significant bit equal to 1 indicate a negative value, which is obtained by subtracting 2^n from the unsigned value of the number. For example, if a is a signed 3-bit binary number, $a=110$ means $6 - 2^3 = -2$.
- `unsigned` specifies that an n -bit binary number be interpreted as a value in the range $[0, (2^n)-1]$. If a is an unsigned 3-bit binary number, $a=110$ means $1*2^2 + 1*2^1 + 0*2^0 = 6$.
- `preserve` preserves the input data type. If the input is signed, the output is also signed. If the input is unsigned, the output is also unsigned.
- `inherit` inherits the input data type from the inherit port. This option is only available when you enable Inherit port. See [Inherit port, on page 426](#) for information about this port.

Inherit port

When you enable this option, the tool creates an inherit port. This port does not convey data, but is used to specify the data type. Enabling this option allows you to do the following:

- Use the variables `syn_inh_wl`, `syn_inh_fl`, and `syn_inh_dt` to specify Output word length, Output fraction length, and Number of shift bits. See [Recast Output Variables, on page 427](#) for information about these variables.
- Use the `inherit` option to specify the Output data type. See [Data Type, on page 426](#) for a description of the option.

Recast Output Variables

You can use these variables to specify values for Output word length, Output fraction length, and Number of shift bits.

Variable	Description
<code>syn_inh_dt = 1 2</code>	Holds the data type for the input data of the inherit port 1 indicates signed input, and 2 indicates unsigned input.
<code>syn_inp_dt = 1 2</code>	Holds the data type for the input data. 1 indicates signed input, and 2 indicates unsigned input.
<code>syn_inh_fl</code>	Holds the input fraction length of the inherit port
<code>syn_inp_fl</code>	Holds the input fraction length
<code>syn_inh_wl</code>	Holds the input word length of the inherit port
<code>syn_inp_wl</code>	Holds the input word length

For example, if you specify an Output word length of $2 * \text{syn_inp_wl}$, the tool creates an output word length that is twice the input word length.

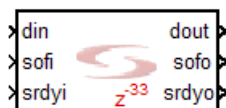
SMC Reed-Solomon Decoder

Decodes the encoded signal using Reed-Solomon error-correcting codes.

Library

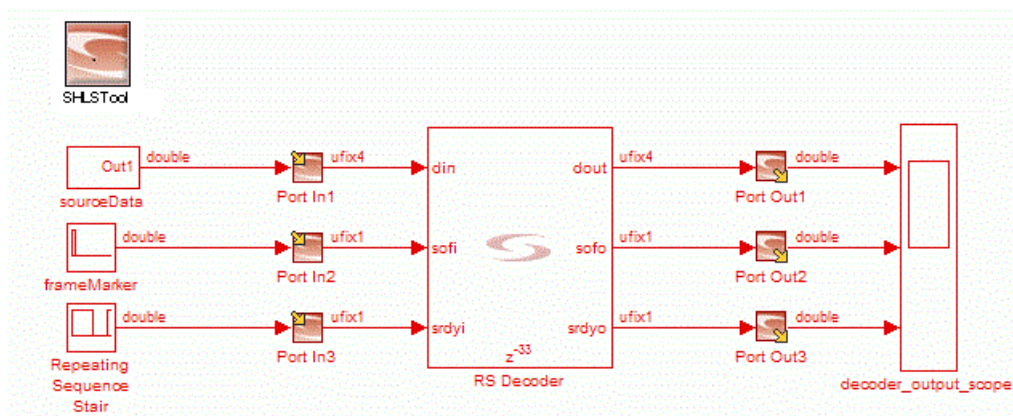
Synphony Model Compiler [Communications](#)

Description



The Synphony Model Compiler Reed-Solomon Decoder decodes the data block encoded by the Synphony Model Compiler Reed-Solomon Encoder (see [SMC Reed-Solomon Encoder, on page 435](#)). It processes each block and attempts to correct errors and recover the original data. See [Reed-Solomon Coding and Decoding, on page 436](#) for more information about Reed-Solomon encoding and decoding.

The following is an example of the Reed-Solomon Decoder block:



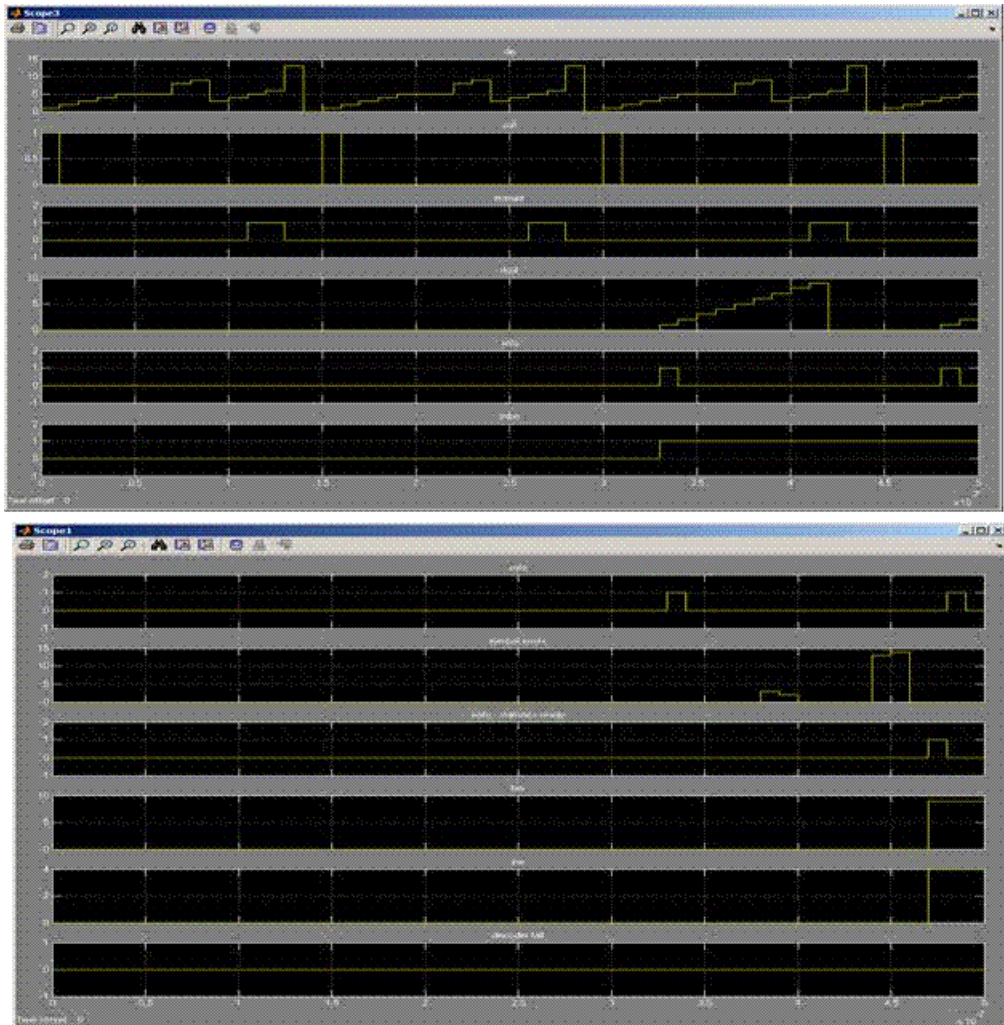
Input and Output Pins

The following table describes the standard and optional pins on the block.

sofi Start of frame input	When this pin is enabled, the block assumes start of frame data at input.
srdyi Source ready input	When this pin is enabled, the block assumes that a valid input signal is given.
sofo Start of frame output	When this pin is enabled, the block marks the start of frame data at output.
srdyo Source ready output	When this pin is enabled, the block generates a valid output signal.
krdyo Sink ready output	When this pin is enabled, the block is ready to accept new input data
eofi End of frame output	This is an optional pin. See Statistics Ready/End of Frame Output, on page 434 for a description.
eri Erasure put	This is an optional pin. See Erasure Input, on page 433 for a description.
symerr Symbol error output	This is an optional pin. See Symbol Error Output, on page 433 for a description.
ber Bit error count output	This is an optional pin. See Bit Error Count Output, on page 434 for a description.
ser Symbol error count output	This is an optional pin. See Symbol Error Count Output, on page 434 for a description.
decfail Decoder failure output	This is an optional pin. See Decoder Failure Output, on page 434 for a description.

Reed-Solomon Decoder Timing Diagram

The following figures show the Reed-Solomon Decoder block timing:



Constant Propagation

The tool propagates constants for this block. See [Constant Propagation](#), on page 731 for a description.

Latency

The latency for this block is calculated as follows:

Codeword length (N) * 2 - Message length(K) + Bitwidth (m) + 8.

Reed-Solomon Decoder Parameters

The screenshot shows the 'Synplify DSP Reed Solomon Decoder' configuration window. The title bar includes the Synplify logo and the text 'Synplify DSP Reed Solomon Decoder'. The window is divided into several sections:

- Synplify DSP RS Decoder**: A description box stating 'This block decodes the RS encoded signal'.
- Parameters**: A section containing input fields for:
 - Bitwidth m: 4
 - Codeword length N: 15
 - Message length K: 9
 - ☐ Specify primitive polynomial: 25
 - ☐ Specify generator polynomial: A dropdown menu with the text 'Specify parameters for the generator polynomial'.
 - Generator polynomial coefficients: [1 2 14 12 14 2 1]
 - Starting Power: 5
 - Root spacing: 1
- Optional Input/Output Selections**: A section with several checkboxes:
 - ☐ Erasure Input
 - ☐ Symbol Error Output
 - ☐ Statistics Ready / End of Frame Output
 - ☐ Bit Error Count Output
 - ☐ Symbol Error Count Output
 - ☐ Decoder Failure Output

At the bottom of the window are four buttons: OK, Cancel, Help, and Apply.

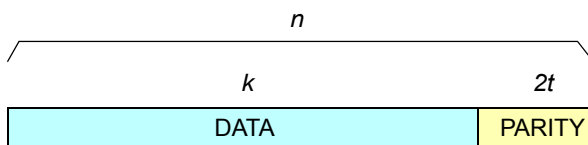
Bit width M

Specifies the bits per symbol for the message symbols and the parity symbols. The Symphony Reed-Solomon Decoder supports short codes for larger values. For example, you can specify a bitwidth (m) of 4, a codeword length (N) of 7 and a message length (K) of 3.

Code Word Length N

Specifies the number of symbols in the code. The value you specify determines the number of parity symbols added:

Codeword length N - Message Length K = Parity



See [Reed-Solomon Coding and Decoding, on page 436](#) for details.

Message Length K

Specifies the number of symbols in the message. The encoder takes the number of data symbols specified in this field that are of the bit width specified in Bitwidth M, and adds parity symbols to make a symbol codeword that is the size specified in Codeword length N. The message length also determines the number of parity symbols:

Codeword length N - Message Length K = Parity

Specify primitive polynomial

Specifies the primitive polynomial for the Galois field. You can specify it in either decimal or vector form:

$D^4 + D^3 + 1 \Rightarrow [1\ 1\ 0\ 0\ 1]$ or 25

Specify generator polynomial

Specifies the polynomial used to generate the codeword. This polynomial is used for oversampling the data that is encoded. All valid code words are exactly divisible by the generator polynomial.

You can define the generator polynomial in either of the following ways:

- Specify coefficients for the generator polynomial
Defines the polynomial coefficients as vectors. When you select this option, the Generator polynomial coefficients field becomes available.
- Specify parameters for the generator polynomial
Specifies the polynomial coefficients using first root and spacing. When you specify this option, the Starting power and Root spacing fields become available. For example:

$$(X-\alpha^{fr}) * (X-\alpha^{fr+sp}) * \dots * (X-\alpha^{fr+(N-K-1)*sp})$$

Generator polynomial coefficients

Defines the coefficients for the generator polynomial as vectors.

Starting power

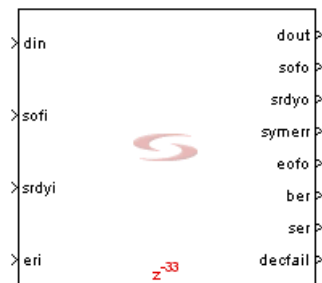
Defines the first root when you specify the coefficients for the generator polynomial using first root and spacing.

Root spacing

Defines root spacing when you specify the coefficients for the generator polynomial using spacing and first root. Root spacing can be greater or equal to 1.

Erasure Input

Adds an optional erasure input signal (eri) to the block. The following figure shows all the optional inputs and outputs to the block.



Symbol Error Output

Adds an optional symbol error output signal (symerr) to the block, which shows the location and value of the error.

Statistics Ready/End of Frame Output

Adds an optional output signal (eoff) for monitoring end of frame or statistics. This signal is high when statistics are available for the decoded data.

Bit Error Count Output

Adds an optional output signal (ber) for counting the total number of bit errors.

Symbol Error Count Output

Adds an optional output signal (ser) for counting the total number of symbol errors.

Decoder Failure Output

Adds an optional output signal (decfail) for decoder failures, which signals when the decoded message is not correct.

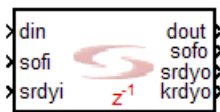
SMC Reed-Solomon Encoder

Generates an encoded signal, using Reed-Solomon error-correction codes.

Library

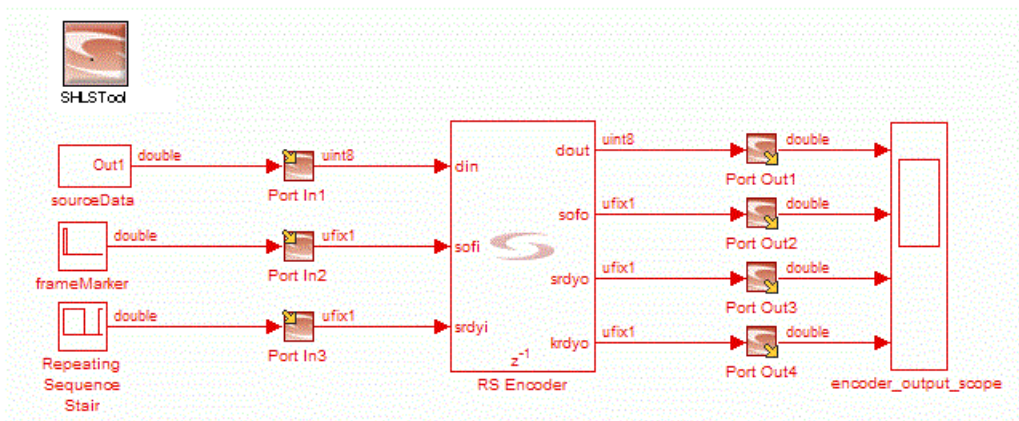
Synphony Model Compiler [Communications](#)

Description



The Reed-Solomon Encoder takes a block of digital data and adds extra parity bits for error handling to the data stream before transmitting it over a communications channel. See [Reed-Solomon Coding and Decoding](#), on page 436 for more information about Reed-Solomon encoding.

The following is an example of the Reed-Solomon Encoder block:

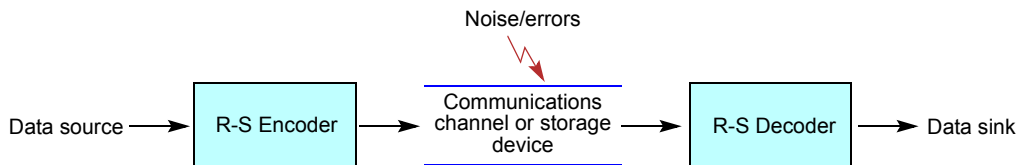


The number of errors that the Reed-Solomon Encoder block can correct is determined by the definition of the encoder. This applies to shortened codes too. For example, the shortened code for RS(255,223) is RS(200,168), but the number of parity symbols is still 32. Thus, the number of correctable errors is $32/2=16$.

You can use the Reed-Solomon commands available in the MATLAB Communications Toolbox to generate polynomials for your design. You can also use the MATLAB commands to validate the results from the Symphony Model Compiler Reed-Solomon blocks.

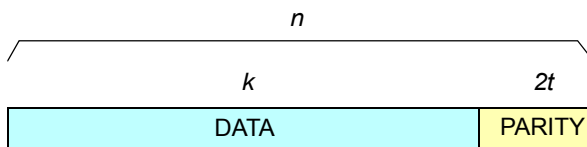
Reed-Solomon Coding and Decoding

Reed-Solomon coder/decoders (CODECs) are widely used for error detection and correction in DSP applications that deal with storage, retrieval, and transmission of data. The Reed-Solomon Encoder takes a block of digital data and adds extra parity bits to the data stream for error handling, before transmitting the data over a communications channel. The Reed-Solomon Decoder processes each block, determines if there are any errors, and corrects them if possible. The errors are transmission or storage errors, like those caused by noise or interference, or by scratches on a CD. Reed-Solomon codes are special linear block codes for correcting errors.



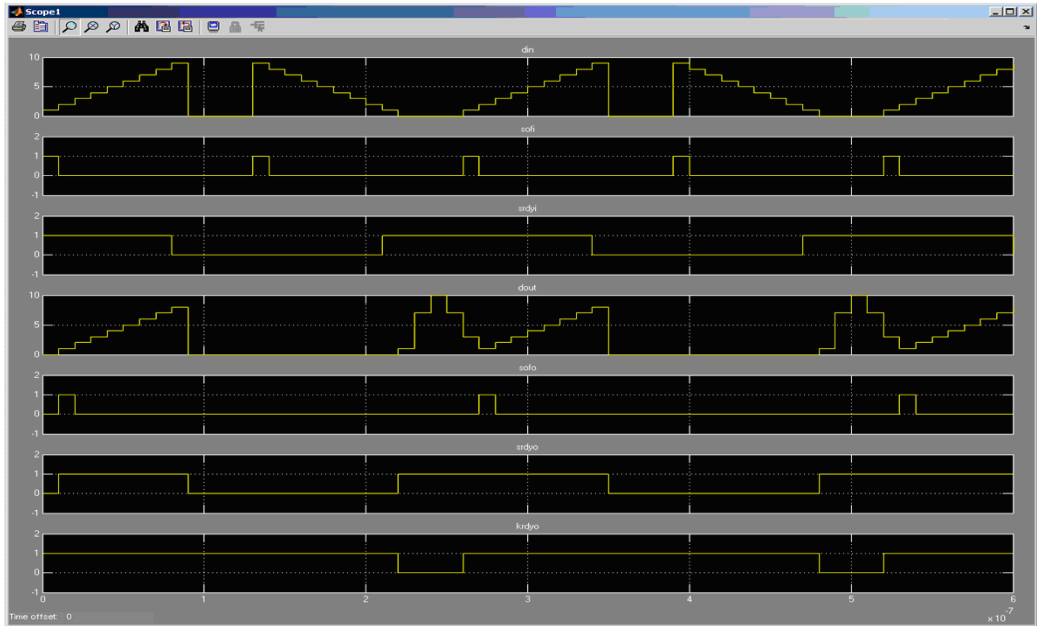
The information to be encoded consists of message symbols and the code that is produced after encoding consists of codewords. Each block of k message symbols is encoded into a codeword that consists of n message symbols. K is called the message length, n is called the codeword length, and the code is called an $[n,k]$ code. A Reed-Solomon code is specified as $RS(n,k)$ with m -bit symbols.

This means that the encoder takes k data symbols of m bits each and adds parity symbols to make an n symbol codeword. There are $n-k$ parity symbols of m bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n-k$. The following diagram shows a typical Reed-Solomon codeword. Note that the data is left unchanged and the parity symbols are appended:



Reed-Solomon Encoder Timing Diagram

The following figure shows the Reed-Solomon Encoder block timing:



Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

The latency of this block is $1(z^{-1})$.

Reed-Solomon Encoder Parameters

Synplify DSP RS Encoder
This block generates the RS encoded signal

Parameters

Bitwidth m

Codeword length N

Message length K

☐ Specify primitive polynomial

☐ Specify generator polynomial

Generator polynomial coefficients

Starting Power

Root spacing

OK Cancel Help Apply

Bit width m

Specifies the bits per symbol for the message symbols and the parity symbols. The Synphony Model Compiler Reed-Solomon encoder supports short codes for larger values. For example, you can specify a bitwidth (m) of 4, a codeword length (N) of 7 and a message length (K) of 3.

Codeword Length N

Specifies the number of symbols in the code. The value you specify determines the number of parity symbols added:

$$\text{Codeword length } N - \text{Message Length } K = \text{Parity}$$

Message Length K

Specifies the number of symbols in the message. The encoder takes the number of data symbols specified in this field that are of the bit width specified in Bitwidth M, and adds parity symbols to make a symbol codeword that is the size specified in Codeword length N. The message length also determines the number of parity symbols:

$$\text{Codeword length } N - \text{Message Length } K = \text{Parity}$$

Specify primitive polynomial

Specifies the primitive polynomial for the Galois field. You can specify it in either decimal or vector form:

$$D^4 + D^3 + 1 \Rightarrow [1 \ 1 \ 0 \ 0 \ 1] \text{ or } 25$$

Specify generator polynomial

Specifies the polynomial used to generate the codeword. This polynomial is used for oversampling the data that is encoded. All valid codewords are exactly divisible by the generator polynomial.

You can define the generator polynomial in either of the following ways:

- Specify coefficients for the generator polynomial lets you specify vectors for the coefficient of the polynomial. When you select this option, the Generator polynomial coefficients field becomes available.
- Specify parameters for the generator polynomial lets you specify the polynomial coefficients using first root and spacing. When you specify this option, the Starting power and Root spacing fields become available.

For example:

$$(X - \alpha^{\text{fr}}) * (X - \alpha^{\text{fr} + \text{sp}}) * \dots * (X - \alpha^{\text{fr} + (N - K - 1) * \text{sp}})$$

Generator polynomial coefficients

Defines the coefficients for the generator polynomial as vectors.

Starting power

Defines the first root when you specify the coefficient for the generator polynomial using first root and spacing.

Root spacing

Defines root spacing when you specify the coefficient for the generator polynomial using spacing and first root. Root spacing can be greater or equal to 1.

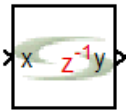
SMC Register

Inserts a delay, with optional reset and enable ports.

Library

Synphony Model Compiler [Memories](#)

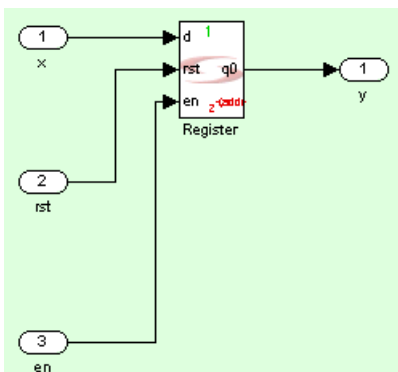
Description



The Synphony Model Compiler Register block is a custom block (see [Primitives and Custom Blocks](#), on page 800 for an explanation) that specifies a delay and optional enable and reset ports. Use this block when you need to put an enable or reset on a delay element.

For best results use the Delay block ([SMC Delay](#), on page 169) instead of the Register block whenever possible, because some retiming optimizations cannot be implemented with the Register block.

The following figure shows the internal construction of the Register block with optional reset and enable ports:



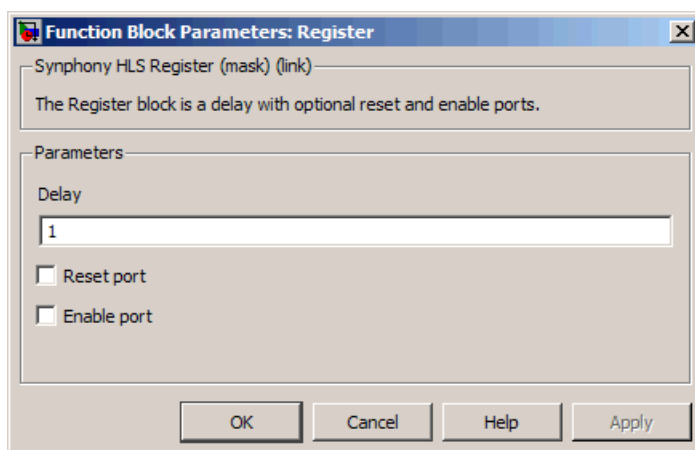
Automatic Scalar Expansion

If the data input is a vector and the reset or enable port is scalar, the tool expands the scalar reset or enable port to the size of the data input vector. The reset and enable can be either vector or scalar.

Latency

The latency of this block is determined by the delay you set.

Register Parameters



Delay

Specifies the delay for the block, in nanoseconds.

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

SMC Reshape

Takes an input matrix or vector and changes its dimensions to the specified values.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Reshape block transforms the dimensions of the input to the new dimensions you specify. For example, if the input is a 3x4 matrix and you specify output dimensions of [2,6], this block outputs a 2x6 matrix. The tool first flattens the matrix to a vector according to the input order and then arranges the vector back to a matrix according to the specified output order. The output dimensions can also be derived from an inherit port. See [Example: 2-D DCT Using Matrix Data Types, on page 698](#) for an example that uses the Reshape block to transpose matrix order.

The input and output elements are equal because this block only changes the dimensions of the input signal; it does not affect the number of elements. The number of input elements is equal to the product of its dimensions.

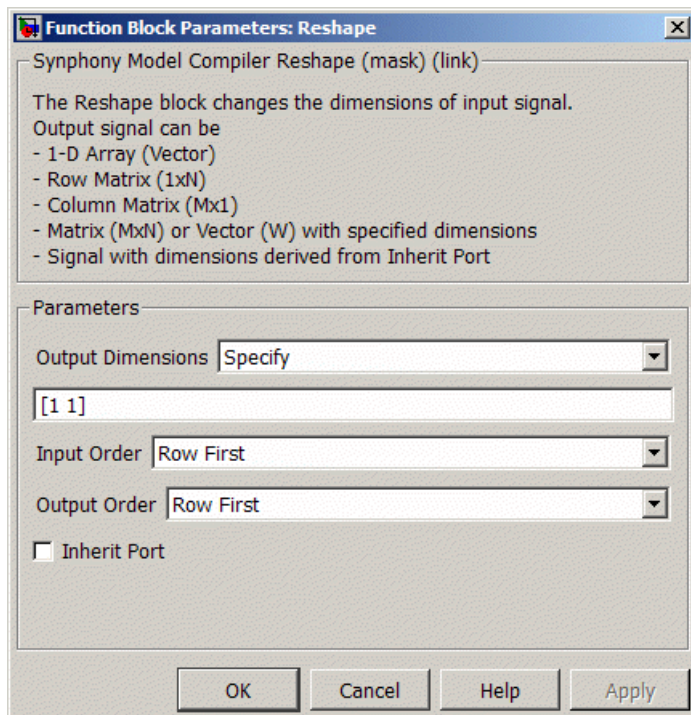
This table shows the supported input and the output choices for the block. Any accepted input can be converted to any of the output choices:

Accepted Input	Output Choices
Vector	Vector (1-D array)
Row matrix (1xN)	Row matrix (1xN)
Column matrix (Mx1)	Column matrix (Mx1)
Matrix (MxN)	Matrix (MxN) or vector (W) with specified dimensions
	Signal with dimensions derived from inherit port

Latency

This block has no latency.

Reshape Parameters



Output Dimensions

Specifies the dimensions of the output signal. You have four choices for the output signal dimensions:

1-D Array	Output signal is a vector that contains the same number of elements as the input signal.
Row Matrix	Output signal is a matrix consisting of a single row, with the same number of columns as the number of input signal elements.
Column Matrix	Output signal is a matrix consisting of a single column, with the same number of rows as the number of input signal elements.
Specify	Lets you specify the output dimensions in the field that becomes available. If you specify a single value [W], you get vector output. If you specify a pair of values [M N], the output is a matrix with M rows and N columns.

When you set Output Dimensions to Specify, another field becomes available to set the output dimensions. You can specify the output dimensions in the following ways:

- A single value [W] or a vector. In either case, it must equal the number of input signal elements.
- A pair of values [M N], where the product of M and N must equal the number of input elements.
- Special variables `syn_mat_rows` or `syn_mat_columns` where `syn_mat_rows` represents the number of rows of the input signal and `syn_mat_columns` represents the number of columns of the input signal.

Input order

Specifies the order in which the input signal is accessed for reshaping. Input order does not matter when the input is scalar or vector.

- Row First accesses the input by row first. This is the default.
- Column First accesses the input by column first.

Output order

Specifies the order in which the output signal is accessed for reshaping, when you set Output Dimensions to Specify. Output order does not matter when the output is scalar or vector.

- Row First uses a row-wise arrangement for the output matrix.
- Column First uses a column-wise arrangement for the output matrix.

Inherit port

Determines whether the tool creates an inherit port. This port does not convey data.

When enabled, the tool creates an inherit port and inherits the dimensions from the inherit port directly. The output dimensions are the same as those of the inherit port. When you enable Inherit Port, it overrides any other dimensions you specified.

Example

The table below shows the output signal values with different parameter settings, assuming that the input is the following 3x4 matrix:

$$\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

Parameter Settings**Output Signal**

Output Dimensions: Specify [2 6]

Input Order: Column First

Output Order: Row First

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}$$
Output Dimensionality: Specify
[syn_mat_columns syn_mat_rows]

Input Order: Row First

Output Order: Column First

Transpose of input signal

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Output Dimensionality: Specify [12]

Input Order: Row First

Output Order: Row First

$$[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12]$$

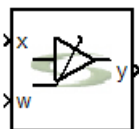
SMC RFIR

Implements a reloadable finite impulse response (FIR) filter with a coefficient load register.

Library

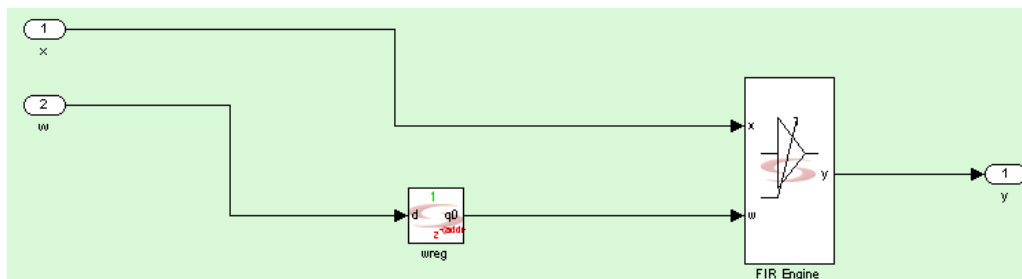
Synphony Model Compiler [Filtering](#)

Description



The Synphony Model Compiler RFIR block is a custom block (see [Primitives and Custom Blocks](#), on page 800 for an explanation) that implements an FIR filter with reloadable coefficients. It combines the FIR Engine block with a loadable coefficient register to allow changes to the filter response by writing different values into the register. As a custom block, it serves as a reference and a good starting point for using the FIR Engine block to create reloadable or adaptive coefficient logic.

You can apply it to programmable filters and adaptive filtering applications. The following figure shows the internal structure, which uses the FIR Engine block and custom block methodology to implement your own customized FIR coefficient update logic.

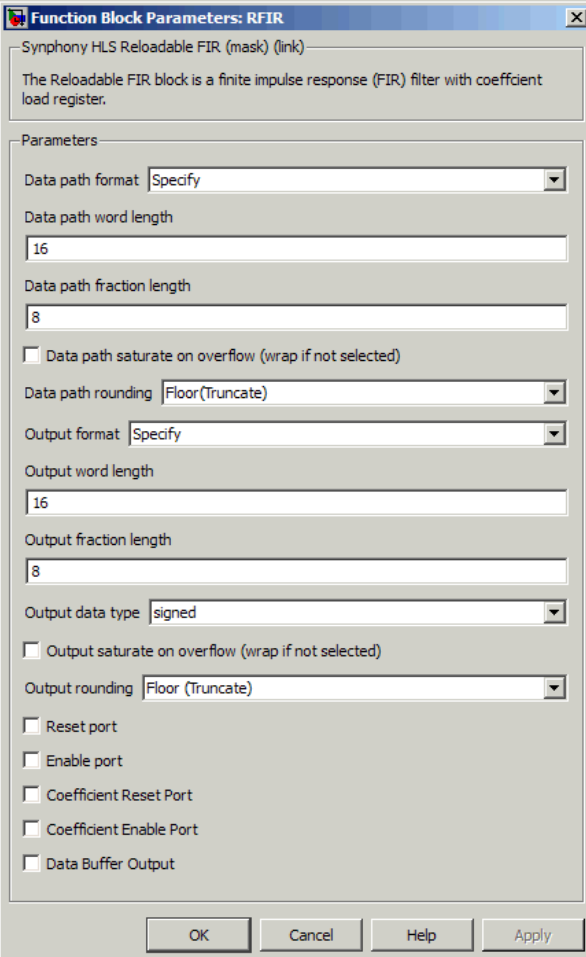


Latency

This block does not introduce latency, but the coefficient load register introduces a latency of one sample time. The icon annotation shows zero latency.

- Latency from input X to Y is zero (same as FIR Engine).
- Latency from W to Y is 1.

RFIR Parameters



The dialog box titled "Function Block Parameters: RFIR" contains the following information:

Synphony HLS Reloadable FIR (mask) (link)

The Reloadable FIR block is a finite impulse response (FIR) filter with coefficient load register.

Parameters

Data path format: Specify

Data path word length: 16

Data path fraction length: 8

☐ Data path saturate on overflow (wrap if not selected)

Data path rounding: Floor (Truncate)

Output format: Specify

Output word length: 16

Output fraction length: 8

Output data type: signed

☐ Output saturate on overflow (wrap if not selected)

Output rounding: Floor (Truncate)

☐ Reset port

☐ Enable port

☐ Coefficient Reset Port

☐ Coefficient Enable Port

☐ Data Buffer Output

Buttons: OK, Cancel, Help, Apply

Data Path Format

Determines data path format. You can set one of these options:

- Automatic sets the data path format to one that uses the maximum of input and output fractions, and the smallest bit width that guarantees no overflow.
- Full Precision uses the smallest bit width that guarantees no overflow, and no truncation is used internally.
- Specify uses the user-defined data type to cast the adder and multiplier outputs for internal calculations. It makes the Data Path Word Length and Data Path Fraction Length options available.

Data Path Word Length

Determines the word length of the data path in bits. It only becomes available when you set Data Path Format to Specify. You can specify it as a value or in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, `syn_coef_dt`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

Data Path Fraction Length

Sets the fraction length of the data path in bits. It only becomes available when you set Data Path Format to Specify. You can specify it as a value or in terms of the `syn_inp_wl`, `syn_inp_fl`, `syn_inp_dt`, `syn_coef_wl`, `syn_coef_fl`, `syn_coef_dt`, and `syn_guard_bit` variables, which are described in [Special Variables, on page 588](#).

Data path saturate on overflow

Determines how the data path overflow value is handled. See [Overflow Saturation Options, on page 585](#) for details.

Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 585](#) for details. This option is not available if Data path format is set to Full Precision.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584 You can specify it as a value or in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , <code>syn_coef_dt</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output fraction length	Output Fraction Length, on page 584 You can specify it as a value or in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , <code>syn_coef_dt</code> , and <code>syn_guard_bit</code> variables, which are described in Special Variables, on page 588 .
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output rounding

Determine how output overflow and underflow are treated. These options are only available when Output format is set to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output rounding	Specifies the algorithm to use to round the output underflow. See Underflow Rounding Options, on page 585 for details of the algorithms.

Reset Port

When enabled, the RFIR is implemented with a reset pin for resetting the FIR filter. The block icon reflects the change.

Enable Port

When enabled, the RFIR is implemented with an enable pin for enabling or disabling the filter. The block icon reflects the change.

Coefficient Reset Port

When enabled, the RFIR is implemented with a coefficient reset pin (`wrst`) for resetting coefficient registers. The block icon reflects the change. The

coefficient reset signal is single bit input, connected to the reset signals of the filter tap size. Enabling this option makes the Coefficient Reset Value option available.

Coefficient Reset Value

Specify one of these two options for reset values:

- All zeroes
- Specify makes the Coefficient Reset Vector option available. When specified, the block accepts a cell array of reset values.

Reset values have the same data format and data type as the FIR coefficients. See [FIR Parameters, on page 229](#) for details.

Coefficient Reset Vector

Specifies vectors for the coefficient reset ports. The reset values are specified as a cell array.

Coefficient Enable Port

When enabled, the RFIR is implemented with a coefficient enable pin (wen) for controlling coefficient updates. The block icon reflects the change. The coefficient enable signal is single-bit input, connected to the enable signals of the filter tap size.

Data Buffer Output

When enabled, the RFIR is implemented with a data buffer output pin (xvec) to feed into adaptive logic. The block icon reflects the change. The data buffer output is a vector output, with a size that is the same as the specified tap length.

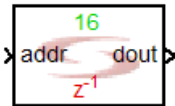
SMC ROM

Models a read-only memory (ROM) with a latency of one sample.

Library

Synphony Model Compiler [Memories](#)

Description



The Synphony Model Compiler ROM block models a ROM with a latency of one sample. If the addr input value exceeds the size of the ROM, the output is 0.

Resets on ROM Output Registers

The SMC tool generates ROM blocks with output registers with a reset signal. However, not all target technologies support this architecture in both synchronous and asynchronous versions.

Automatic Scalar Expansion

If the address input of the ROM is a vector and the enable port is scalar, the enable port is expanded according to the size of the address vector.

Latency

This block has a latency of one sample.

ROM Parameters

Function Block Parameters: ROM

Symphony Model Compiler ROM (mask) (link)

The ROM block is a one-sample-latency ROM.

Parameters

ROM data

35:55

☐ Automatically infer ROM data word length

ROM data word length

7

ROM data fraction length

0

ROM data type signed

☒ ROM data saturate on overflow (wrap if not selected)

☒ ROM data round towards nearest on underflow (truncate if not selected)

☐ Enable Port

☐ Dual Port ROM

OK Cancel Help Apply

ROM data

Sets the ROM depth and valid input values in one of these ways:

- Type in the ROM vectors. If you enter [10 20] as the vectors, the software generates a ROM with a depth of 2. The first value is 10 and the second value is 20.

The block inputs and resulting outputs are shown in this table:

Input Value	Output Value
0	10
1	20
Any other integer	0

- Use the `syn_read_hex` function. See [syn_read_hex](#), on page 610 for the syntax and [Specifying ROM Data with syn_read_hex](#), on page 776 for information on using this function.
- If the ROM input is vectorized, specify a matrix for the ROM values. Each row vector contains ROM values for one channel of input. The number of rows in the matrix must equal the input vector size.

ROM data word length, ROM data fraction length, and ROM data type

For descriptions of these parameters, see the following:

ROM data word length	Output Word Length , on page 584
ROM data fraction length	Output Fraction Length , on page 584
ROM data type	Output Data Type , on page 584

ROM data saturate on overflow, ROM data round towards nearest on underflow

Determine how overflow and underflow are treated. For descriptions of these parameters, see the following:

ROM data saturate on overflow	Saturates or wraps the overflow; see Overflow Saturation Options , on page 585.
ROM data round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see Underflow Rounding Options , on page 585.

Enable port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal ROM. When enable is low, the output as well as the contents of the ROM does not change. If Dual Port ROM is selected, both outputs are held at the previous value.

Dual Port ROM

When enabled, the block is implemented with two independent address and data output ports that share the same ROM data. To use dual-port ROM, make sure that both address ports have the same data type and the same sample times.

For most FPGA targets, this kind of ROM maps to a single dual-port block RAM macro.

SMC RTL Encapsulation

Lets you embed large third-party IP in a Symphony design and simulate it in Simulink without using external RTL simulators.

Library

Symphony [Ports & Subsystems](#)

Description



The RTL Encapsulation block lets you embed third-party blocks in an SMC Symphony design, and use fast C-based simulation within the Simulink environment. It points to the third-party RTL. RTL encapsulation uses C-model generation technology to provide high-performance, bit-accurate and cycle-accurate behavior, so this block does not require any special Simulink features or external simulators. The RTL Encapsulation block is the preferred method to embed third-party blocks in the SMC tool.

The RTL Encapsulation block is preferred over Smart Black Box, because the simulations run significantly faster and do not require access to an external RTL simulator. If you have IP with no access to the RTL code at all, you cannot use either RTL Encapsulation or Smart Black Box, and you must use the Black Box block ([SMC Black Box, on page 56](#)) instead.

To use this block, place it in the SMC model and point to the RTL files. Use normal Simulink simulation to verify the model. For the implementation, the tool instantiates source RTL in the generated RTL output.

Use this tool to easily do the following:

- Integrate your large third-party or legacy RTL IP into your SMC model, with fast C-based simulation
- Add standard and/or custom interface blocks to your model
- Reduce RTL integration effort in the downstream implementation flow

- Add hand-designed control logic blocks, state machines, and/or cycle accurate blocks to your model

Prerequisites

Before using the RTL Encapsulation block, the compiler must be configured in MATLAB to compile s-functions. You usually do this during installation with the `setup` script. If you did not do it then, type `mex -setup` in the MATLAB console to configure the compiler. The *Release Notes* list compilers supported for RTL encapsulation.

Latency

The contents of the third-party HDL source determines the latency.

RTL Encapsulation Parameters

Symphony Model Compiler RTL Encapsulation Block

This block will let you add custom RTL block into the Symphony Model Compiler flow

RTL Definition: Import File List

Import File List: .rtl_encapsulation_ddc_filelist.txt

Entity/Module Name: rx_filter

Include Directories:

- .rtl

RTL Parameters/Generics

	Parameter Name	Parameter Type	Parameter Value
1	BITWIDTH	integer	16
2			
3			
4			
5			
6			
7			

Status: Block is up-to-date.

Edit Port Configuration Ok Cancel Help

RTL Definition

Specifies how the third-party IP is defined, as a single file or multiple files

- Single HDL file
Specifies the single .v (Verilog) or .vhd (VHDL) file that defines the IP.

- **Import File List**
Specifies the text file that contains a list of the HDL files for the design. Use this setting if the IP is defined in multiple .v (Verilog) or .vhd (VHDL) files.

HDL File

Specifies the path to the RTL file. This can be an absolute or relative path to the model file. This option is only available when RTL definition parameter is set to Single HDL File.

Import File List

Specifies the path to a text file that lists all the HDL files to be included. The list must contain paths to the files which are either absolute or relative to the model file. The definition file extensions in the list must be .v or .vhd. For example, if your RTL is defined in four files named myiplib1.vhd (library definition file with myiplib being the library), myip2.v, myip3.v and myip4.vhd, create and save a text file (myipfilelist.txt) that lists the paths to the RTL definition files as follows:

```
-L myiplib C:\myips\myiplib1.vhd  
C:\myips\myip2.v  
C:\myips\myip3.v  
C:\myips\myip4.vhd
```

The files are compiled in the order specified.

Entity/Model Name

Specifies the topmost entity or model name for the RTL. This name becomes the instance name for the RTL Encapsulation block and the name of the instantiated entity or model.

Include Directories

Specifies the include directories for the Verilog include files.

- Click the Add Path button to browse to the directory and include it.
- Select the directory from the list and click Remove Path to exclude a directory.

RTL Parameters/Generics

Lets you define or override generics and parameters in the RTL file.

- **Parameter Name** specifies the parameter you wish to override.

- Parameter Type defines the type for the parameter. The supported types are integer and string.
- Parameter Value specifies a new value for the parameter.

Generate/Edit Port Configuration

Opens a tabular interface where you can generate, view, or edit the port configuration. Refer to [Port Configuration Window for RTL Encapsulation, on page 461](#) for a description of the interface.

The tool automatically regenerates the port configuration information if you make changes to the RTL files, any of the other files in the list, the top module name, or the include directories; or if you use the parameter overrides.

Encapsulation Status

The current status is printed at the bottom of the dialog box, if you already have it open. If the block is initialized from the Simulink window, a progress bar displays the progress.

Port Configuration Window for RTL Encapsulation

RTL Encapsulation - rtl_encapsulation_ddc/RTLE_ddc

RTL Encapsulation - Port Configuration

The interface will let you view and update the port configurations.
Check and update editable columns (*) in below tables.

Input ports

	Port Name	Sample time	Bit Width	Port Type (*)	Reset polarity (*)
1	GlobalReset	Inherited	1	global reset	active high
2	GlobalEnable2	Inherited	1	global enable	NA
3	GlobalEnable1	Inherited	1	global enable	NA
4	GlobalEnable4	Inherited	1	global enable	NA
5	FIRinQ	Inherited	18	input	NA
6	FIRinI	Inherited	18	input	NA

Output ports

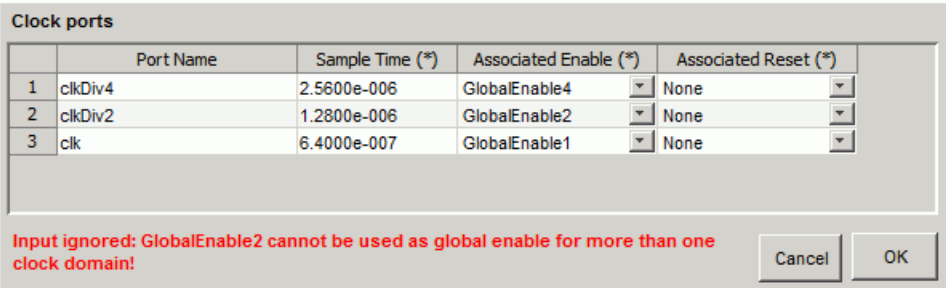
	Port Name	Sample Time (*)	Bit Width	Fraction Length (*)	Signed (*)
1	dummyClk	1.2800e-006	1	0	<input type="checkbox"/>
2	FIRoutQ	2.5600e-006	16	0	<input checked="" type="checkbox"/>
3	FIRoutI	2.5600e-006	16	0	<input checked="" type="checkbox"/>

Clock ports

	Port Name	Sample Time (*)	Associated Enable (*)	Associated Reset (*)
1	clkDiv4	2.5600e-006	GlobalEnable4	None
2	clkDiv2	1.2800e-006	GlobalEnable2	None
3	clk	6.4000e-007	GlobalEnable1	None

Only the columns marked with an asterisk (*) are editable. The tool validates the information entered. If you enter an invalid value, the tool ignores it and maintains the previous values.

If an invalid value is ignored, the tool prints the reason at the bottom of the configuration window as shown in the following figure:



Input Ports

Name	Description
Port Name	Specifies the input port name.
Sample Time	Specifies the input port sample time in seconds.
Bit Width	Specifies the bit width for the input port.
Port Type	<p>Specifies the port type. Set it to input, global reset, or global enable. The tool infers the global reset port if it is asynchronous, but it does not infer synchronous global resets.</p> <p>Global reset and global enable are implicit ports for Simulink simulation and do not appear as ports on the block. For RTL generation, these ports are connected to global reset and global enable signals respectively in the SMC generated RTL.</p>
Reset polarity	<p>Sets reset polarity. You must specify reset polarity for global reset ports. The default setting is active high.</p> <p>If the port is not a global reset, this option does not apply.</p>

Output Ports

Name	Description
Port Name	Specifies the output port name.
Sample time	For multirate designs, you must specify the sample time in seconds. For single rate designs, set this to Auto . With this setting, the tool uses the input port sample time as the sample time for the output ports. Auto is the default setting for single-rate designs. You can also specify sample time as a workspace variable or an expression.
Bit Width	Specifies the bit width for the output port.
Fraction length	The default fraction length is zero. If your design requires a different sample time for output ports, specify it here.
Signed	The default assumes that single-bit ports are unsigned and multi-bit ports are signed. If you require a different setting, specify it here..

Clock Ports

Name	Description
Port Name	Specifies the output port name.
Sample time	Specifies the sample time in seconds.
Associated Enable	Select one of the global enable input ports listed here to be the enable associated with the specified clock. This is mandatory for multirate designs.
Associated Reset	Select one of the global reset input ports listed here to be the reset associated with the specified clock. This is mandatory for multirate designs. If there is more than one reset in a design, there should be one reset per clock domain.

For multirate designs, the tool issues an error while editing port information if the following conditions are not met:

- Each clock must have an associated enable.
- If there is more than one reset in a design, there should be one reset per clock domain.

Limitations of the RTL Encapsulation Block

There are some limitations to using the RTL Encapsulation block:

- The RTL cannot have a clock port of a sequential element that is fed by internally generated clocks or gated clocks.
- The active clock edge must be posedge.
- You cannot use the RTL Encapsulation block inside an HLS Subsystem block.
- The RTL cannot have latches.
- The RTL cannot have combinational loops.
- The RTL cannot have instantiations of any FPGA device-specific primitives like DCM, RAMB16, or DSP48.
- The RTL cannot have black box modules.
- The RTL cannot be encapsulated if it has an input port with paths to multiple output ports, some of which are purely combinational and others registered. This is a Simulink s-function framework limitation. If you have such a design, the tool issues an error message.

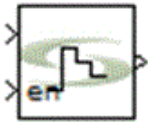
SMC Sample and Hold

Samples and holds the input signal.

Library

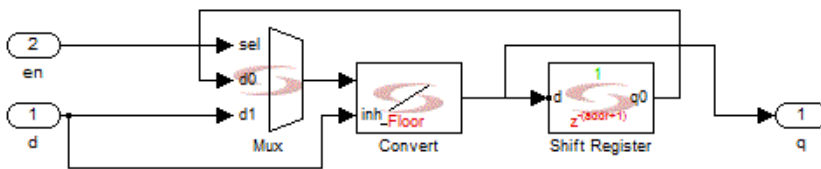
Synphony [Signal Operations](#)

Description



The Sample and Hold block samples and holds the input signal. A new sample is loaded into the memory when the enable (en) port is high; otherwise it retains the old value.

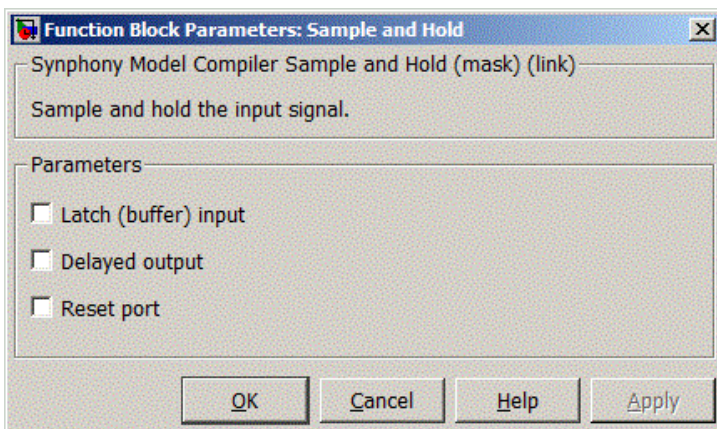
This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for definition). The following figure shows the internal modeling when delayed output is not checked:



Latency

This block has a latency of 1 if Delayed Output is checked.

Sample and Hold Parameters



Latch (buffer) input

When enabled, the Sample and Hold block outputs the input value right from the clock cycle till the next triggering event occurs. Checking this parameter enables the block to be used in a loop.

Delayed Output

When enabled, the block adds one cycle of latency from the input to the output. If a delay can be tolerated in the design, the delayed version will have better performance in the hardware.

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

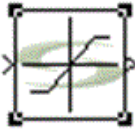
SMC Saturate

Saturates the input signal to the values specified in the positive and negative saturation value fields.

Library

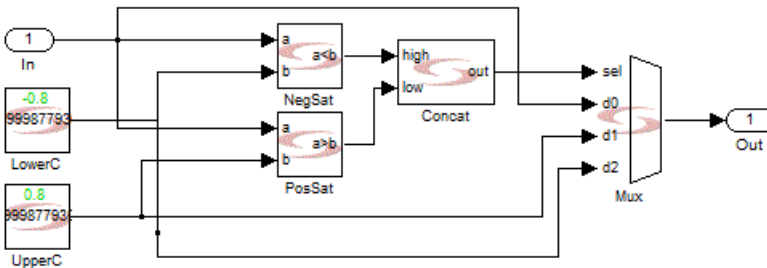
Synphony [Signal Operations](#)

Description



The Saturate block saturates the input to the values specified in the positive and negative saturation value fields.

This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition). The following figure shows the internal modeling when saturation mode is constant:



Latency

This block has zero latency.

Saturate Parameters

Function Block Parameters: Saturate

Synphony Model Compiler Saturate (mask) (link)

Saturate the input to the values specified in the positive and negative saturation value fields.

Parameters

Saturation mode: Constant

Positive saturation value: 0.8

Negative saturation value: -0.8

Saturation value fraction length: 15

☒ Saturation value round towards nearest on underflow (truncate if not selected)

Output format: Same as Input

OK Cancel Help Apply

Saturation mode

Specifies if the positive and negative saturation threshold values of the input signal are provided as constants through mask parameters, or as variable through input ports.

Positive saturation value

Specifies the maximum positive value of the input signal beyond which the input saturates to this set value.

Negative saturation value

Specifies the minimum negative value of the input signal beyond which the input saturates to this set value.

Gain round towards nearest on underflow

Determines how the underflow for the gain is treated. When enabled, the option rounds the underflow using Nearest algorithm, When disabled, the option rounds the overflow with the Floor (truncate) algorithms. See [Underflow Rounding Options, on page 585](#) for details.

Output Format Parameters

For descriptions of these parameters, see the following:

Output quantization rule	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round on underflow

Determine how overflow and underflow are treated. These options are only available when Output format is set to Specify.

Output saturate on overflow	When enabled it saturates the overflow; when disabled, it wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

SMC Sequence

Repeats a sequence of specified data.

Library

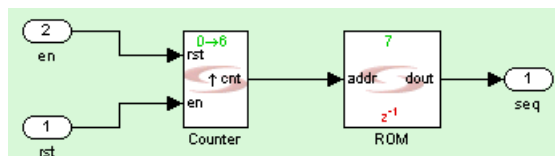
Symphony Model Compiler [Sources](#)

Description



This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) repeats a sequence of specified data.

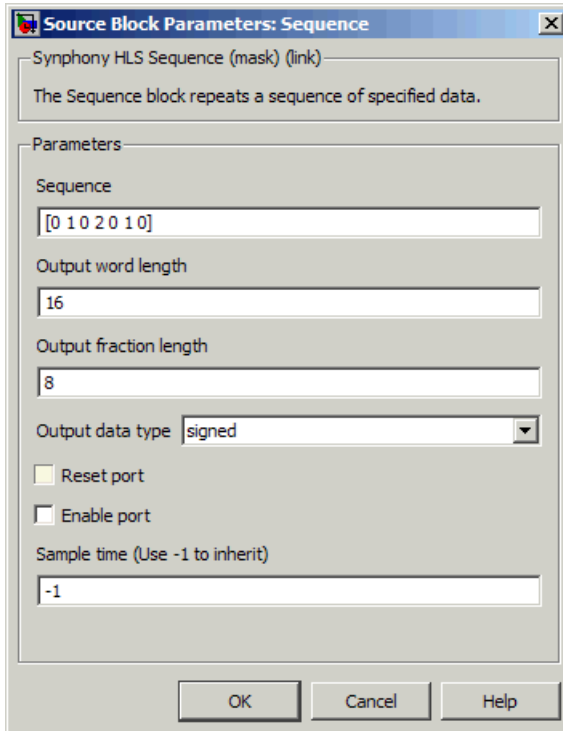
The following figure shows the internal construction of this block, with reset and enable ports:



Latency

The latency of the Sequence block is 1.

Sequence Parameters



Sequence

Specifies the sequence to be repeated. The data is cast into the number format specified by the Word Length, Fraction Length, and Data Type options.

Output word length, Output fraction length, and Output data type

For descriptions of these parameters, see the following:

Word length	Output Word Length, on page 584
Fraction length	Output Fraction Length, on page 584
Data type	Output Data Type, on page 584

Reset Port

When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

Enable Port

When enabled, the block is implemented with an enable pin. The enable port is connected to the enable signal of the internal shift register.

Sample Time

Determines sample time. Use -1 to inherit. This option is not available if you specify reset or enable ports.

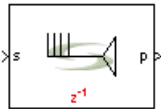
SMC Serial to Parallel

Implements a data packet combiner that collects serial data packets at the input and merges them into a parallel data word at the output.

Library

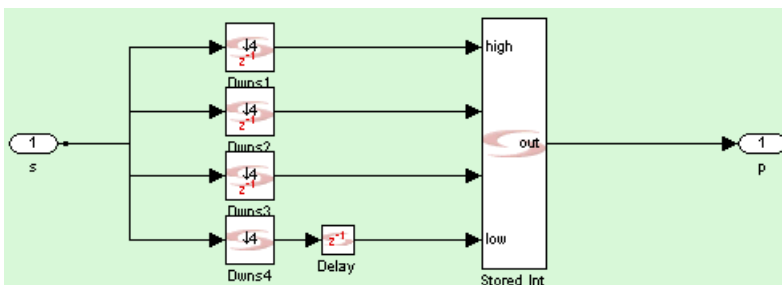
Synphony Model Compiler [Signal Operations](#)

Description



The Serial to Parallel block combines serial data packets from the input and merges them into a parallel word for the output. You can specify the order in which the serial inputs are combined. As this block combines several input samples into a word, the sampling rate at the output decreases.

This block is a custom block. (See [Primitives and Custom Blocks](#), on [page 800](#) for a definition.) The following figure shows how the block is modeled:

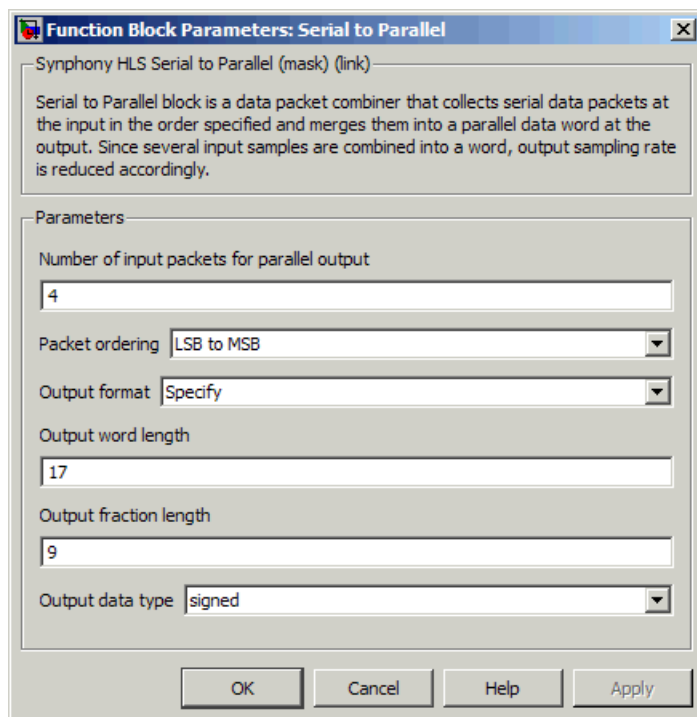


Icon Annotations

The icon for this block displays the following information:

Latency One sample latency with respect to the clock domain.

Serial to Parallel Parameters



The dialog box titled "Function Block Parameters: Serial to Parallel" contains the following information:

Synphony HLS Serial to Parallel (mask) (link)

Serial to Parallel block is a data packet combiner that collects serial data packets at the input in the order specified and merges them into a parallel data word at the output. Since several input samples are combined into a word, output sampling rate is reduced accordingly.

Parameters

Number of input packets for parallel output: 4

Packet ordering: LSB to MSB

Output format: Specify

Output word length: 17

Output fraction length: 9

Output data type: signed

Buttons: OK, Cancel, Help, Apply

Number of input packets for parallel output

Specifies the number of serial output packets. As the block combines many input packets into one output word, the output sampling rate decreases.

Packet ordering

Determines how the serial input packets are combined for output.

- MSB to LSB combines the serial input packets from the most significant to the least significant bit.

- LSB to MSB combines the serial input packets from the least significant to the most significant bit.

Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round on underflow

Determine how output overflow and underflow are treated. These options are only available when Output format is set to Specify.

Output saturate on overflow	When enabled, saturates the overflow; when disabled, wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Specifies which algorithm is used to round the output underflow. See Underflow Rounding Options, on page 585 for details of the algorithms.

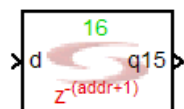
SMC Shift Register

Implements a delay line with dynamic or static access to intermediate taps.

Library

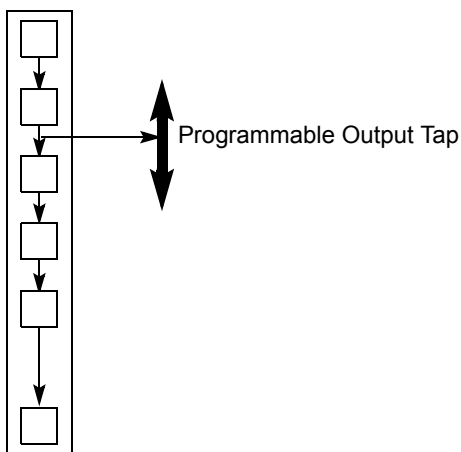
Synphony Model Compiler [Memories](#)

Description



The Synphony Model Compiler Shift Register block implements a static or dynamic shift register. Many applications use a delay line, and this block offers a component that can delay a signal by a certain number of samples.

You can also implement a delay line with the Delay block, but the Shift Register block allows you to tap into the delay line at a fixed or addressable location and get the output. You can use this for linear feedback shift register applications like pseudo-random noise generation, stream encryption/decryption algorithms, and for serial-to-parallel conversion.



Outputs inherit the input data type. Potential inputs must have the same data type.

For vector or matrix input, the tool infers multichannel shift registers. Separate shift registers are inferred for each element in the input signal.

Automatic Scalar Expansion

If the data input is a vector/matrix and a reset or enable port is scalar, the reset and enable ports are expanded according to the size of the data vector/matrix.

Constant Propagation

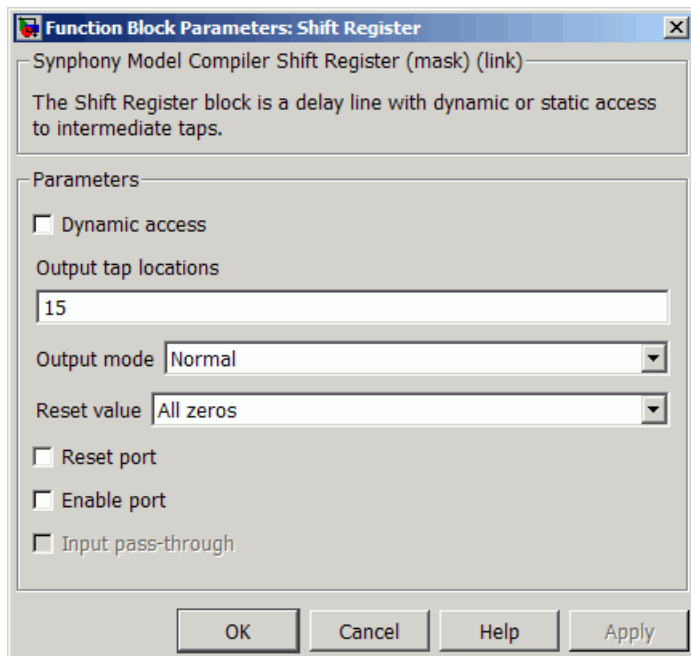
The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Icon Annotations

The icon for this block displays the following information:

Note (green)	Indicates the number of delay elements in the shift register.
Latency (red)	Indicates the latency of tap location <code>addr</code> with regard to input. For example: $z^{-(addr+1)}$.

Shift Register Parameters



Dynamic Access

When this option is enabled, the software implements a single output that taps into the register (dynamic shift register), and lets you specify the length of the delay line in Shift Register Length.

When this option is disabled, the software implements a static shift register, and allows you to set the tap locations in Output Tap Locations and specify the Output Mode.

Shift Register Length

Sets an integer value that specifies the length of the delay line. This parameter is only available when Dynamic Access is enabled.

Output Tap Locations

Specifies a vector of integer values. The number of delays for each output tap location is the corresponding integer value specified plus one. This parameter is only available when Dynamic Access is disabled.

Output Mode

Specifies the mode for the output. You can set it to one of the following:

- Normal
All tap outputs are exposed as output ports.
- Merge to vector
For scalar input, the block output is a vector. The first tap of the shift register becomes the first element of the vector, and the last tap becomes the last element.

You cannot have vector or matrix input signals in this mode.

- Merge to vector in reverse order
For scalar input, the block output is a vector. The last tap of the shift register becomes the first element of the vector, and the first tap becomes the last element.

You cannot have vector or matrix input signals in this mode.

- Merge to matrix/vector
For scalar input, the behavior is the same as with Merge to vector mode.

For vector input, the block output is a matrix. The number of columns in the output matrix is the same as the number of elements in the vector input. The number of rows in the output matrix is equal to the number of taps in the shift register. Each column is the merged output of each shift register inferred for each input element. The first tap of each shift register becomes the element in the first row of each column and the last tap becomes the element in the last row of each column.

You cannot have matrix input signals in this mode.

Reset Value

Determines the reset value. You can set this to one of the following:

- All zeros
- Specify lets you specify the reset value in the Reset Vector field that becomes available.

Reset Vector

This field only becomes available when you set **Reset** value to **Specify**. The number or elements in the reset value vector must be the same as the register depth.

Reset Port

When enabled, this clears the contents of the delay line. The software does a local block reset and combines it (OR) with a system reset. If the target architecture does not provide well-defined power-up behavior, the software generates an explicit system-level reset for the RTL implementation.

When disabled, the software uses the implicit system implementation reset where it is relevant, but does not reset the block implementation. The contents of the delay line are determined by the shift (**Enable**) operation.

The following table summarizes the effects of the **Enable** and **Reset Pin** settings. Note that the **Reset Pin** setting takes priority.

Enable Pin	Reset Pin	Functionality Implemented
Off	Off	No shifting, outputs maintained
Off	On	Reset delay line to all zeroes
On	Off	Enable delay line by making the shift register active
On	On	Reset delay line to all zeroes

Enable Port

When enabled, the **Enable** port controls the delay line and the sample clock for asynchronous buffering. The **Enable** pin can be interpreted as a shift, and the registers can shift with the sample clock. If the option is disabled, the delay line is always enabled.

The **Enable Pin** option is used with **Reset Pin**, as described in the previous table.

Input Pass-through

When enabled for static shift registers, directly passes the input to the output of tap 0. This option is only available for static access shift registers when one of the merge output modes is selected. You cannot use this in **Normal** mode.

Input pass-through operation is different from normal operation. Normally, in Merge to Vector mode for example, the first element in the vector output is the output from the first tap, the second is the output from the second tap, and so on. If the tap location is 0, output is the input delayed by one cycle; if the tap location is 1, the output is the input delayed by two cycles. With this option enabled, the input is directly passed to the output of tap 0: the output of tap location 0 is the input, the output of tap location 1 is the input delayed by one cycle, and so on. This option is useful for creating vectors of the current input sample and delayed input samples.

Examples of Shift Register Settings and Implementations

The following table shows the settings required for some implementations:

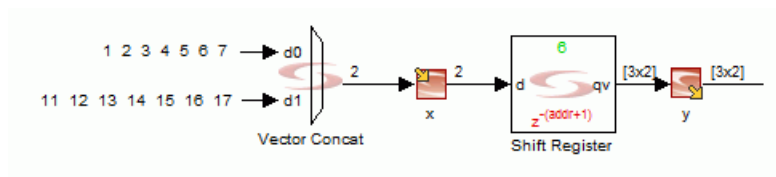
Standard Delay	Reset = off Enable = off Dynamic Access = off Tap Locations = [0]
Delay z^{-N}	Reset = off Enable = off Dynamic Access = off Tap Locations = [N-1]
Static Shift Register	Reset = off Enable = off Dynamic Access = off Tap Locations = [M-1, N-1]. This corresponds to the M and N delays from the shift register input
Dynamic Shift Register	Reset = off Enable = off Dynamic Access = on Length = N

Merge Output Mode Examples

The following examples illustrate the merge output mode.

Merge Output Mode with Scalar Input

The following figure shows a shift register with scalar input. The annotations shown are with Input pass-through disabled.

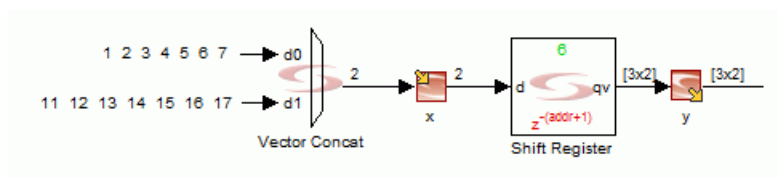


The table shows the output for different combinations of tap locations mode and other settings.

Taps	Output Mode	Input pass-through	Output Vector y
[0 2 5]	Merge to vector	Off	[6 4 1]
	Merge to matrix/vector		[1 4 6]
	Merge to vector in reverse order	On	[7 5 2]
	Merge to vector in reverse order		[2 5 7]
[1 3 5]	Merge to vector	Off	[5 3 1]
	Merge to matrix/vector		[1 3 5]
	Merge to vector in reverse order	On	[6 4 2]
	Merge to vector in reverse order		[2 4 6]

Merge Output Mode with Vector Input

The following figure shows a shift register with vector input and a Merge to matrix/vector output mode. The annotations shown are with Input pass-through disabled.



The table shows the output for different combinations of tap locations and input pass-through settings.

Tap locations	Input pass-through	Output matrix y
[0 2 5]	Off	$\begin{bmatrix} 6 & 16 \\ 4 & 14 \\ 1 & 11 \end{bmatrix}$
	On	$\begin{bmatrix} 7 & 17 \\ 5 & 15 \\ 2 & 12 \end{bmatrix}$
[1 3 5]	Off	$\begin{bmatrix} 5 & 15 \\ 3 & 13 \\ 1 & 11 \end{bmatrix}$
	On	$\begin{bmatrix} 6 & 16 \\ 4 & 14 \\ 2 & 12 \end{bmatrix}$

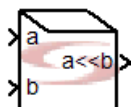
SMC Shifter

Performs a variable left or right shift on the input signal.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Shifter block performs a variable left or right shift on the input signal. The second operand (b) determines the shift amount. Negative shift values reverse the direction of the shift. Fractions are ignored. To implement constant shifts, use the [SMC Convert](#) block.

The type of architecture is automatically derived from the input type. For the most optimized solution or if you do not expect negative values, use unsigned data for the input.

- If the input has signed data and the value is negative, the Shifter block reverses the shift register and implements a larger shifter.
- If you have an unsigned number at the input, the software implements an optimized solution and creates a smaller architecture.

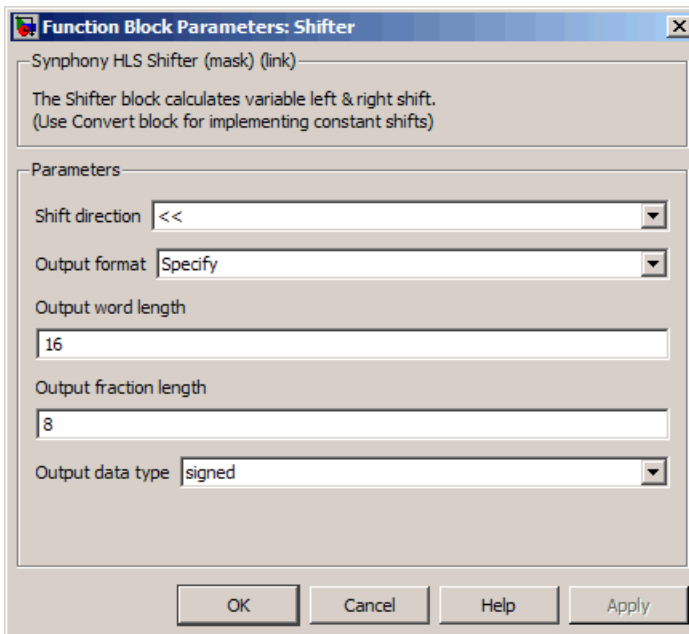
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has no latency.

Shifter Parameters



Shift direction

Sets the shift direction.

- << implements a left shift. The software does not do any overflow checks for left shifts.
- >> implements a right shift.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

SMC SHLSTool

Opens the SHLSTool interface where you can set system-level optimizations and set parameters for generating RTL code.

Library

Synphony Model Compiler, top level library

Description

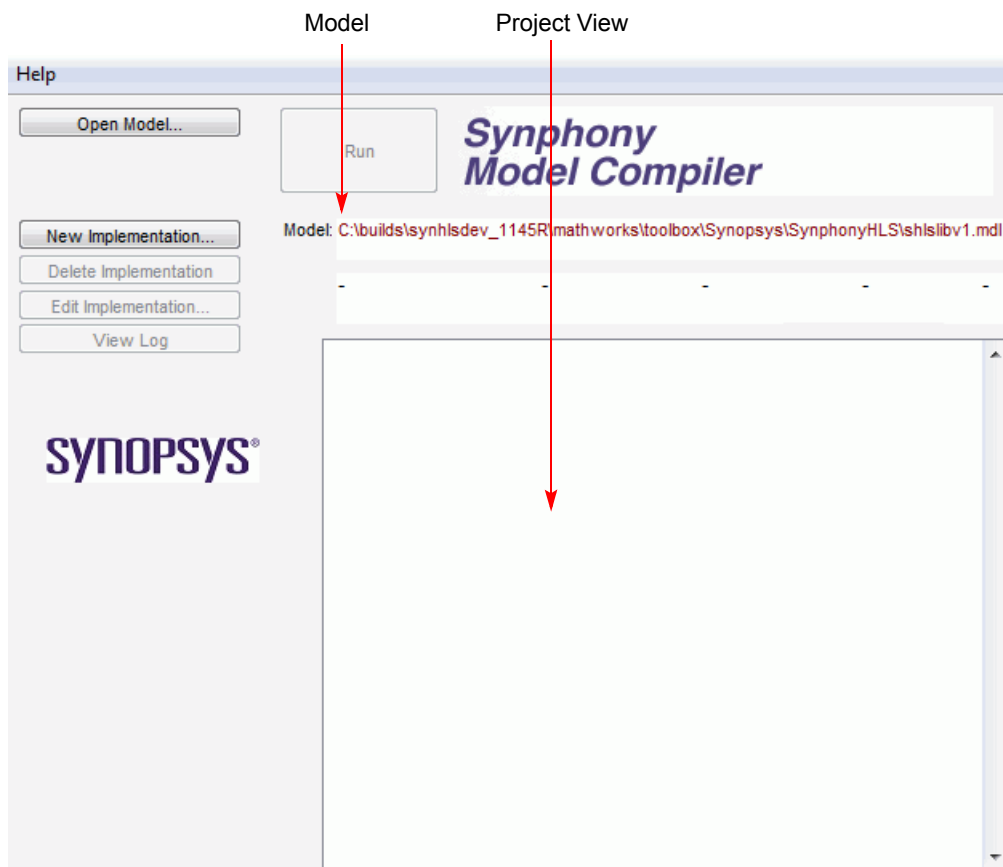


The SHLSTool toolbox provides an interface where you can specify system-level optimizations and set parameters for generating RTL code. Users of Synopsys FPGA synthesis products will find the interface intuitive, as it has the same look and feel as the synthesis tools. See [Running Synthesis with SHLSTool, on page 677](#) for information about using this toolbox.

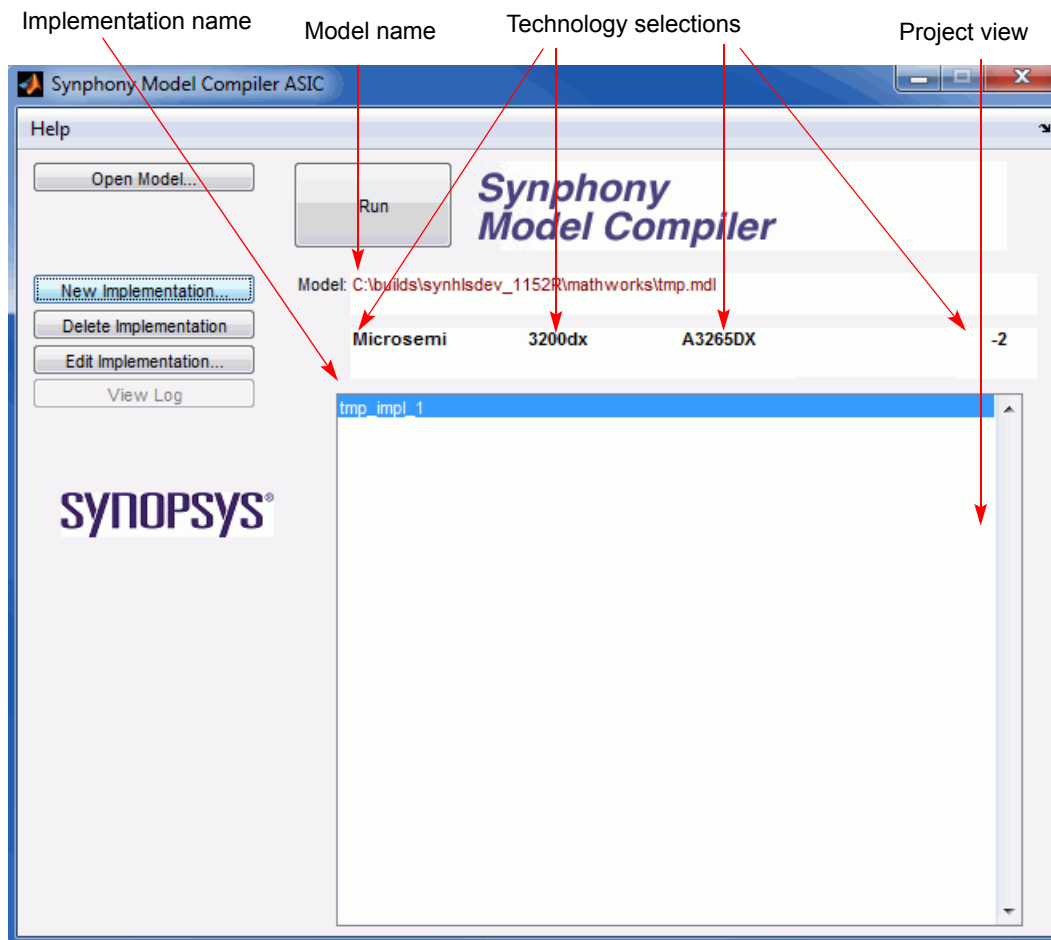
You must have a SHLSTool block in every Simulink® model that contains any element from the Synphony Model Compiler blockset.

SHLSTool Toolbox Interface

The following figure shows the Symphony toolbox interface as it appears before you create an implementation:



The next figure shows the interface once you have created an implementation. The interface displays information about the implementation, and the buttons on the left side are enabled.



Open Model

Opens a dialog box where you can select the model file you want to open. The current model file appears in the Model field.

New Implementation

Opens a dialog box where you can specify the name for a new implementation and set other options. See [Implementation Options Dialog Box, on page 490](#) for details.

Delete Implementation

Deletes the selected implementation.

Edit Implementation

Opens a dialog box where you can set various optimization and synthesis target options and generate RTL code. See [Implementation Options Dialog Box, on page 490](#) for details. When you set a target technology in this box, the results are reflected in the toolbox.

View Log

Opens the log file.

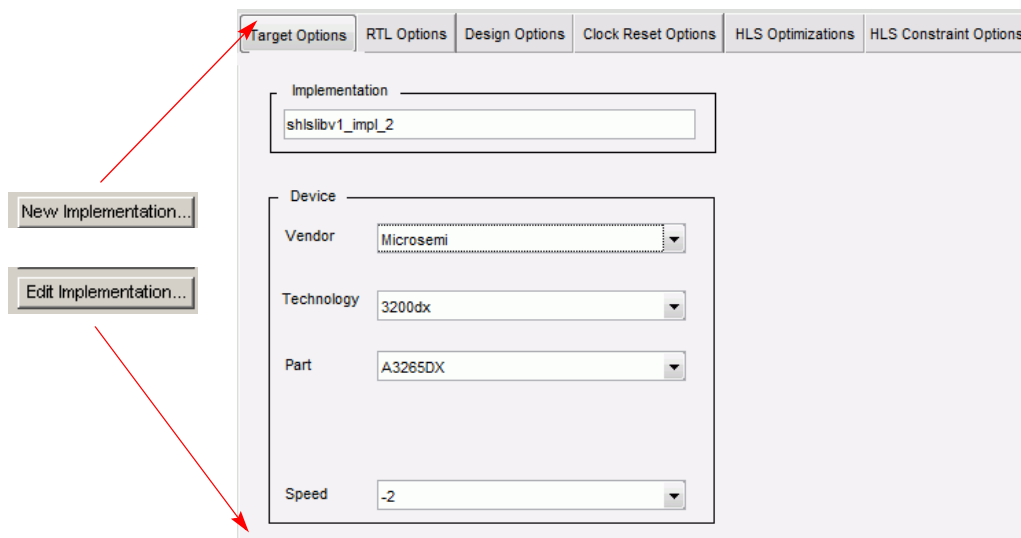
Run

Optimizes the model according to the options you set and generates VHDL source files, a Synplify Pro project file, a constraint file for synthesis, and an optional test bench. The files are written to the design directory.

If you set options and do not click Run, the options are saved for the selected implementation, but no RTL code or test benches are generated. You can do this to run a Simulink simulation with the RTL generator block added to your model.

Implementation Options Dialog Box

The Implementation Options dialog box opens when you click the New Implementation or Edit Implementation buttons in the Symphony Model Compiler toolbox (see [SMC SHLSTool, on page 486](#)). The implementation options vary, depending on the technology you select.



The following describe the options. See [Setting up Implementations, on page 644](#) for details about using the options.

- [Target Options, on page 491](#)
- [RTL Options, on page 492](#)
- [Design Options, on page 495](#)
- [Clock Reset Options, on page 497](#)
- [HLS Optimizations, on page 501](#)
- [HLS Constraint Options, on page 505](#)

Target Options

The options vary according to the technology you select in the Vendor option.

The screenshot shows the 'Implementation Options' dialog box with the 'Target Options' tab selected. The 'Implementation' field is set to 'shlslibv1_impl_2'. The 'Device' section is expanded, showing the following options:

- Vendor:** Microsemi
- Technology:** 3200dx
- Part:** A3265DX
- Speed:** -2

Implementation

Lets you name or rename the implementation.

Vendor

For FPGA designs, select the vendor you want to target. Setting this option determines the choices available for Technology, Part, Package, and Speed.

Technology

Selects the technology.

- For FPGA devices, this selects a target technology. The information is used to generate the project file for synthesis, and for retiming. You can use the Synplify Pro tool to port the design to other devices that are not listed here.

Part

Selects a target part for the FPGA technology device you selected. You can port the design to other available parts that are not listed here, by changing the target part in the Synplify Pro interface.

Package

Selects a target package for the FPGA technology device you selected. You can port the design to other available packages that are not listed here, by changing it in the Synplify Pro interface.

Speed

Selects a target speed grade for the FPGA technology device you selected. You can port the design to other speed grades that are not listed here, by changing it in the Synplify Pro interface.

When disabled (the default value), the Synopsys FPGA synthesis tools insert bypass logic to prevent mismatches when the logic specifies simultaneous reads and writes to the same RAM location.

RTL Options

The following figure shows the RTL Options tab of the Implementation Options dialog box. Some options are vendor-dependent.

The screenshot shows the 'RTL Options' tab of the Implementation Options dialog box. The tab is selected among others: Target Options, RTL Options, Design Options, Clock Reset Options, HLS Optimizations, and HLS Constraint Options. The RTL Options section is divided into three main areas:

- RTL**: Contains two checked options: ☒ Generate VHDL and ☒ Generate Verilog.
- Test Bench**: Contains one unchecked option: ☐ Generate RTL test bench.
- C output**: Contains one unchecked option: ☐ Generate C Code.

There is also an **RTL Preferences** section on the right side of the RTL Options area, which contains one unchecked option: ☐ Limit the length of names in output RTL. Below this option is a text field labeled 'Maximum Name Length' with the value '80' entered.

Generate VHDL

When enabled, this option generates a VHDL design that you can use as input for synthesis with the Synplify Pro tool. The supported format is VHDL 93.

Generate Verilog

When enabled, this option generates a Verilog design that you can use as input for synthesis with the Synplify Pro tool. The supported format is Verilog 2001.

Generate RTL Test Bench

When enabled, this option creates an HDL test bench for pre-synthesis functional verification with an HDL simulator. You use the test bench along with the HDL files created by the RTL Generator and the test vectors captured during Simulink simulation. The test bench instantiates the top-level module of the design, drives it with input test vectors, reads the output, and compares it with the output test vectors. The software handles the extra latency introduced by retiming by treating it as one of the inputs when it generates the test bench.

For information about using this option, see [Verifying the RTL with a Test Bench, on page 853](#).

Generate C Code

This option is only available when you enable Generate RTL test bench. To use this option you must have the appropriate license. If you choose to generate C code, the tool generates output in C format that you can then use. See [Chapter 13, Working with C Output](#) for details.

Limit the length of names in output RTL

When enabled, sets a limit on the length of names in the output RTL code generated. This limit applies to the names of entities, modules, signals and block names coming from blocks, subsystems and signal names in the Simulink model, when they are written out in RTL.

The shortened name follows this format: <Shortened hierarchy name>_<6 digit value><Shortened block/entity/module/signal name>_<6 digit value>.

<i><Shortened hierarchy name></i>	Hierarchy information that is included in the RTL name when the tool generates a flattened RTL for folding. The hierarchy name is shortened if it is more than three-quarters of the value you set in Maximum Name Length .
<i><6 digit value></i>	6 digit value appended to shortened identifiers to make the names unique after shortening.
<i><Shortened block/entity/module/signal name></i>	Shortened name for the block, entity, module or signal name. The name is shortened if it exceeds one-quarter of the value in Maximum Name Length .

The tool conforms to these rules:

- When the original identifier string is longer than the value you specify in **Maximum Name Length**, and both names (hierarchy name and block/entity/module/signal name) are longer than the pre-assigned lengths described below ($\frac{3}{4}$ and $\frac{1}{4}$ of desired maximum name length respectively), the tool shortens the names according to the format.
- The tool does not shorten hierarchy and/or block/entity/module signal names if the name length is less than the three-quarter or one-quarter of the maximum name length limit, respectively.
- If one of the names does not exceed the limit and is not to be shortened, the hierarchy or block/entity/module signal name can be longer than the three-quarter or one-quarter maximum name length limit.

The name length limit might not affect some cases:

- Some names might still be longer than the limit you set, because of auto-generated strings added to identifier names like `_block`, `N_`, `my`, etc. Set a lower maximum name length value to get the desired output.
- Some auto-generated instances and signals in the RTL implementation are not affected by this setting; for example, identifiers not defined in user model. Examples are low level primitives in some IPs, or the adder in an FIR implementation.

Maximum Name Length

Sets the maximum name length for names in the output RTL. The value you set here affects how long names are truncated in the output. See

[Limit the length of names in output RTL, on page 493](#) for details. The minimum value you can set is 32.

Design Options

The following figure shows the Design Options tab of the Implementation Options dialog box. Use these options to define global resets. See [Defining Reset Signals, on page 758](#) for a step-by-step procedure.

The screenshot shows the 'Design Options' tab of the Implementation Options dialog box. At the top, there are five tabs: 'Target Options', 'RTL Options', 'Design Options' (selected), 'Clock Reset Options', and 'HLS Optimizations'. Below the tabs, there are two main sections. The first section contains two checkboxes: 'Generate Global Enable' (unchecked) and 'Generate Global Reset' (checked). The second section, titled 'Design Parameters', contains three dropdown menus: 'Flip Flop Reset Polarity' set to 'Active High', 'Flip Flop Reset Sensitivity' set to 'Asynchronous', and 'Reset Option (Verilog only)' set to 'Automatic'. The third section, titled 'Reset Interface', contains one checkbox: 'Generate Separate Reset per Clock Domain' (unchecked).

Generate Global Enable

When enabled, creates an internal global enable signal for the design. No global enable ports are created for any of the clock domains. The default setting is disabled.

A global enable signal can be used to stall the design based on user input. You do not need to enable this option unless you need control of the global enable signal in your design. The tool achieves better area and timing if this option is disabled.

Generate Global Reset

This option is currently not available for Microsemi designs.

Flip Flop Reset Polarity

Sets the global reset polarity for the design. You can set it to Active High or Active Low.

Flip Flop Reset Sensitivity

Assigns a global type for the resets in the design.

- Synchronous codes the registered elements with synchronous global resets.
- Asynchronous codes the registered elements with asynchronous global resets.
- Automatic assigns the global reset according to the vendor selected.

For details of how the Symphony tool implements resets in the design, see [Synchronous and Asynchronous Resets, on page 684](#).

Reset Option (Verilog Only)

Creates resets for all registers in Verilog designs.

- Automatic lets the tool decide when to insert resets for registers. Some registers might not have resets. This is the default for FPGA, and is the recommended setting for FPGA designs.
- All registers with reset enforces resets for all registers.

Generate Separate Reset per Clock Domain

When enabled, the tool generates separate resets for each clock domain.

Clock Reset Options

The following figure shows the Clock Reset Options tab of the Implementation Options dialog box. Use these options to generate a top-level clock_reset module that contains the input clock and reset information for the design.

Target Options RTL Options Design Options **Clock Reset Options** HLS Optimizations HLS Constraint Options

☒ Generate Clock-Reset Circuitry

Reset Module Parameters

Power On Reset

Power On Reset Polarity

Active High

☒ User Reset

User Reset Polarity

Active High

User Reset Sensitivity

Synchronous

Clock Module Parameters

Set of Clock Sources (MHz)
(Leave blank for default oscillator value)

Clocking Scheme

Dedicated Clocks

Clock Circuit Type

Synthesizable Dividers

☒ Reset Deassertion Synchronization

Generate Clock-Reset Circuitry

When the option is disabled, only the files that correspond to the Simulink design are generated. This is the default.

When enabled, a top-level module called `clock_reset` is inserted in the top-level Symphony design in addition to the automatically generated RTL code. See [Clock and Reset Management, on page 686](#) for details. The tool also generates additional files for the `clock_reset` module; the files are listed in [Clock/Reset Circuitry Files, on page 690](#).

If this option is enabled, the log file contains this entry:

```
@N: Generation of Clock-reset circuitry enabled.
```

Enabling this option makes other parameters available, like the clocking scheme.

Power On Reset Polarity

Defines the power on reset polarity that is applied to the design externally (g_porst pin). You can set it to Active High or Active Low. The pin is assumed to be asynchronous, and you cannot adjust the sensitivity. For additional information, see [Reset Functionality with the Clock_reset Module, on page 689](#).

User Reset Polarity

Defines the user reset polarity that is externally applied to the design (g_urst pin). You can set it Active High or Active Low. For additional information, see [Reset Functionality with the Clock_reset Module, on page 689](#).

User Reset Sensitivity

Defines the reset type for the user reset input signal. You can set this to Synchronous or Asynchronous. For additional information, see [Reset Functionality with the Clock_reset Module, on page 689](#).

Set of Clock Sources

Specifies the available oscillator frequencies for the design. There must be a rational ratio between the design clocks and the oscillator frequency values entered here.

You specify the value as a row or column vector. For example:

Single rate designs	Enter the frequency with or without square brackets. For example: [100] or 100.
Multi rate designs	Enclose the frequency values in square brackets, and separate the values with spaces, commas, or semicolons. To specify 100 MHz, 200 MHz, and 300 MHz, enter one of the following: [100 200 300] [100, 200, 300] [100; 200; 300]

If you leave this field blank, the tool looks at the sample rates of the Simulink design and automatically takes the least common multiple frequency of clock signals.

It also records it in the log file:

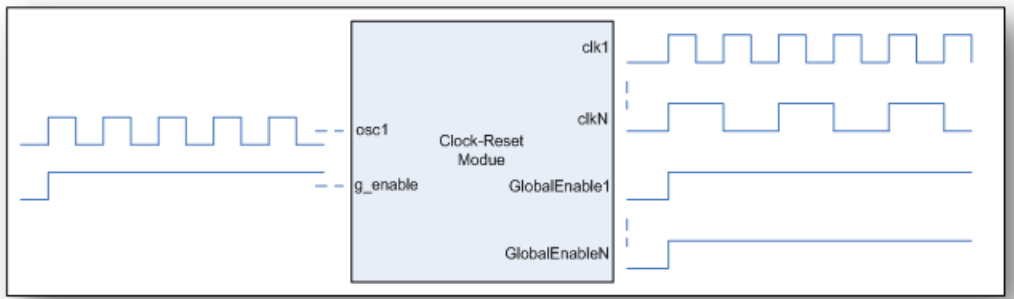
```
@N: Oscillator source file not specified. Using least common
multiple frequency value.
```

See [Clock_reset Module Limitations, on page 690](#) and [Log File Messages for the Clock_reset Module, on page 691](#) for more information about limitations to using clock sources and the listing of clock sources in the log file.

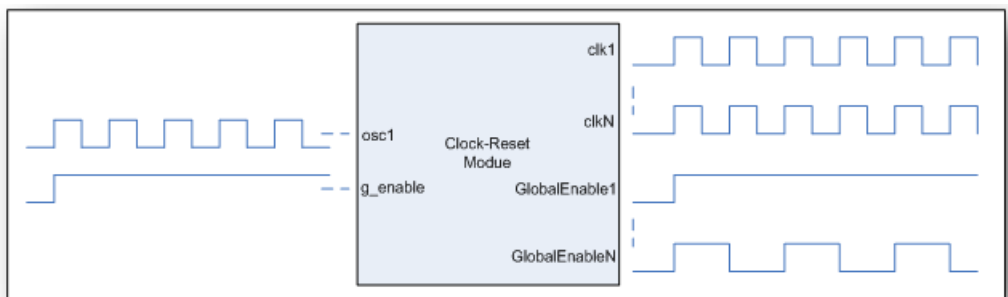
Clocking Scheme

Specifies the clocking strategy for the core design. Set one of the following:

- **Dedicated Clocks** assumes that each design clock is supplied to the design separately. Reset de-assertion is synchronized with the clock, and the tool generates logic for this de-assertion. You can set additional option details in Clock Circuit Type. Use this option for 1/N ratios.



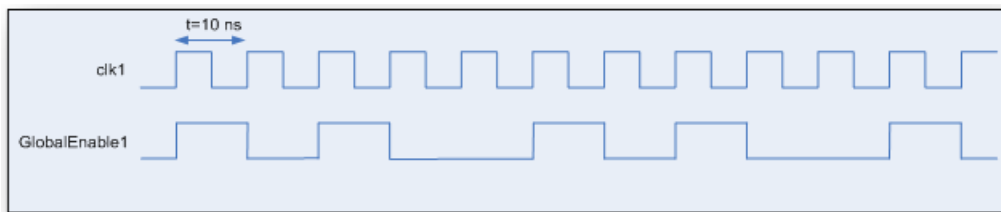
- **Enabled Clocks** uses a single clock source. Each design clock is fed with this fast clock and global enable signals are supplied according to the required clock division ratio. Use this option for 1/N or M/N ratios.



When you set this option for M/N ratios, the tool uses global enable signals to determine the ratio. In the following example, the waveforms show a 40 MHz design clock generated from an oscillator

with 100 MHz frequency. Note that GlobalEnable1 is active in two cycles for every five clk1 cycles.

$$M/N = 2/5$$

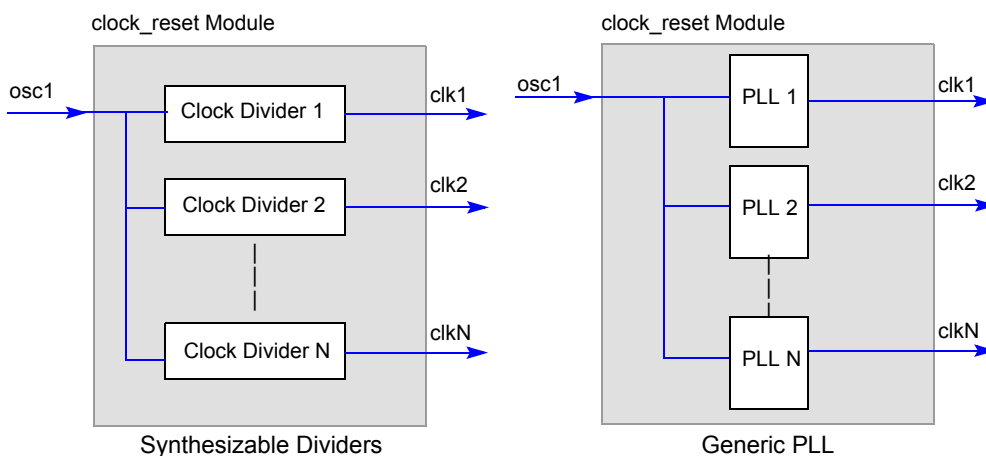


By default, the tool does not generate logic to synchronize the reset with the clock when it is deasserted. For de-assertion synchronization, you must enable Reset Deassertion Synchronization.

See [Clock_reset Module Limitations, on page 690](#) and [Log File Messages for the Clock_reset Module, on page 691](#) for additional information.

Clock Circuit Type

Determines the logical structure to be used while generating design clocks from the available oscillator frequencies. It determines the internal structure of the clock reset module. These options are only available when you set Clocking Scheme to Dedicated Clocks.



- **Synthesizable Dividers** uses clock divider logic to generate design clocks from the oscillator input to the design. Use this to implement 1/N division ratios. For M/N ratios, set Clocking Scheme to Enabled Clocks.

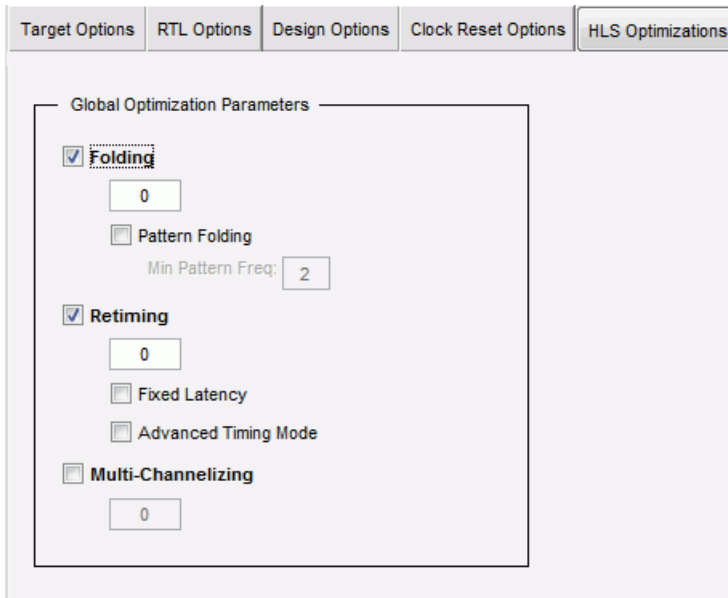
- Generic PLL uses PLL-like placeholder code to represent common PLL structures. You must replace these placeholders with the actual PLL or clock manager logic for the target FPGA.

Reset Deassertion Synchronization

When enabled, it generates logic to synchronize resets with the clock when resets are deasserted. This option is only available when Clocking Scheme is set to Enabled Clocks. When disabled, the clocking scheme does not use reset deassertion synchronization.

HLS Optimizations

The following figure shows the HLS Optimizations tab of the Implementation Options dialog box.



Folding

Performs time-multiplexed resource sharing during area/speed trade-offs within a single-channel system. When you enable Folding, it makes a box available where you can set the folding value, and automatically enables Pattern Folding and Retiming.

The folding value sets a minimum for the number of system clocks per output sample. 0 disables folding. A positive value sets a minimum that is used as a guide for the number of system clocks per sample. For information about the use of this option, see [Optimizing with Folding, on page 662](#).

Pattern Folding

When enabled, runs the pattern folding optimization on the design to identify recurring patterns and share resources. See [Using Pattern Folding, on page 665](#) for details about pattern folding. The tool reports the number of distinct patterns it identified in the log file. Enabling this option also makes the Min Pattern Freq option available.

Min Pattern Freq

Sets a value for the pattern folding algorithm. The algorithm does not identify any patterns that occur less frequently than the number you specify. The default value is 2. You can significantly reduce the computational complexity of pattern identification by judiciously selecting a value that allows larger patterns to be identified.

Retiming

Enables retiming. Retiming rearranges delays so as to optimize speed, while preserving functionality. Retiming cannot move Register block instances, as explained in [Retiming Register and Delay blocks, on page 659](#). When retiming is enabled, a box opens where you can set the number of extra latencies (delays) available for retiming. If you specify very fast sample rates, retiming can use these extra latencies to meet the timing requirements.

The default value of 0 retimes the design by moving existing delays. For details about using this option, see [Optimizing with Retiming, on page 655](#).

Enabling retiming also enables the Advanced Timing Mode option.

Advanced Timing Mode

Determines which timing mode is used: advanced timing mode or estimation mode.

- Enable this option to use the advanced timing mode for timing estimates. Use this mode for optimal results. In this mode, the tool uses Synplify Pro target-specific timing data to produce more accurate results. Greater accuracy means better architectural choices in the RTL. The first run with this option enabled takes longer than estimation mode. However, subsequent runs are as fast as the estimation mode, because the tool caches most of the timing characterization data and does not have to generate it.
- Disable this option to use estimation mode for timing estimates. Use estimation mode in the early stages of design, as the results are less accurate. In this mode, the tool uses simpler, latency-based device characterizations as a basis for optimizations.

The tool defaults to estimation mode if it cannot find Synplify Pro or if problems occur. The log file reports blocks that met timing, blocks that did not meet timing, and blocks in timing loops.

ATM is recommended for achieving optimal results. Only use estimation mode for the early stages of the design.

Fixed latency

Adds latency stages. This option is only available when you enable Retiming. When you specify this mode, the retiming engine retimes the design and then pads the outputs with the remaining delays so as to always maintain the specified latency. It adds the number of latency stages equal to the value you specified for the Retiming option. If you specify more latency stages than are needed for pipelining, the remainder of the latency stages pad the I/O.

Multi-channelizing

Generates a multi-channel system from a single-channel specification. Enabling this option makes a box available where you can set the number of channels. This number also sets the number of system clocks per output for each channel. If you set this option to 2, each channel operates at 2 clocks per sample and 2 channels share each computational resource. When you use the Multi-channelizing option, you cannot use Folding, which is an alternative mechanism to trade speed for resources.

For information about using this option, see [Optimizing with Multichannelization, on page 674](#).

HLS Constraint Options

This figure shows the HLS Constraint Options tab of the Implementation Options dialog box. Use these options to apply constraints from a constraint file.

The screenshot shows the 'HLS Constraint Options' tab selected in the Implementation Options dialog box. The tab bar at the top includes 'Target Options', 'RTL Options', 'Design Options', 'Clock Reset Options', 'HLS Optimizations', and 'HLS Constraint Options'. The main content area has two sections:

- Constraint File**: Contains a checked checkbox labeled 'Enable Constraint File'. Below it is a text input field and two buttons: 'Browse...' and 'Edit'.
- RAM Constraint**: Contains an unchecked checkbox labeled 'Use ReadWrite conflict logic attribute for RAM'.

Enable Constraint File

Lets you specify or edit a Tcl file that contains constraints to be applied to the design, For example, you can set `shls_retiming_lock` constraints and apply them to the design, as described in [HLS Constraints File, on page 620](#).

- If it is not checked (default for new designs), the tool does not use a constraint file when it runs synthesis.
- When enabled, specify the name of the Tcl file to be used in the associated field. You can use the Browse button to locate the file. The tool applies the constraints to the design when it synthesizes it. This setting becomes the default for the design if you previously enabled this option. The log file reports details about the success or failure of the application of the specified constraints.

Edit

Lets you edit the contents of the specified Tcl file or create a new one if it does not exist.

Use ReadWrite Conflict Logic Attribute for RAM

Guides how RAM read-write conflicts are handled during synthesis with the Synopsys FPGA tools. This option is only available when Vendor is set to an FPGA target.

When enabled, the software adds the following attribute to the fdc constraint file generated after DSP synthesis. This attribute specifies that the FPGA synthesis tool not insert bypass logic to resolve simultaneous reads and writes to the same RAM location.

```
define_global_attribute syn_ramstyle (no_rw_check)
```

When disabled (the default value), the Synopsys FPGA synthesis tools insert bypass logic to prevent mismatches when the logic specifies simultaneous reads and writes to the same RAM location.

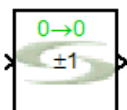
SMC Sign

Provides the 2-bit sign value (=1 or -1) for the input.

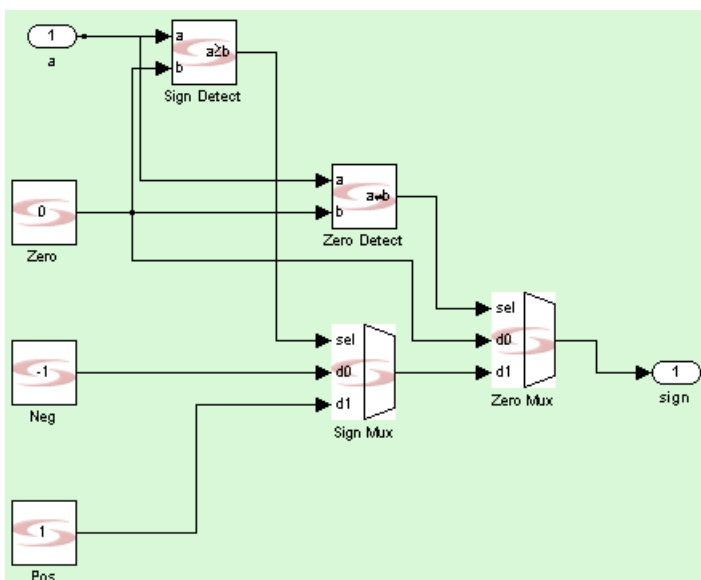
Library

Synphony Model Compiler [Math Functions](#)

Description



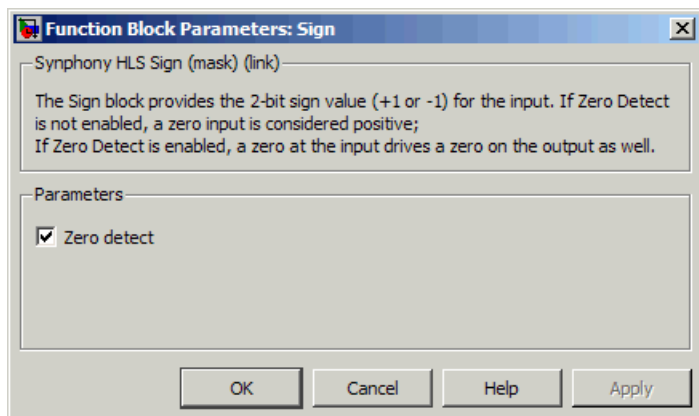
This custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition) provides the 2-bit sign value (=1 or -1) for the input.



Latency

This block has no latency.

Sign Parameters



Zero Detect

Determines what operation to perform when the input is 0. If you enable the option, a 0 input drives a 0 on the output. If you disable the option, the software treats the 0 input as a positive number, and outputs +1.

SMC Signal Update

Updates the specified elements of a vector or matrix input signal using a given update signal.

Library

Synphony Model Compiler [Signal Operations](#)

Description



This block updates a set of elements for input vector/matrix data. The block uses the following inputs:

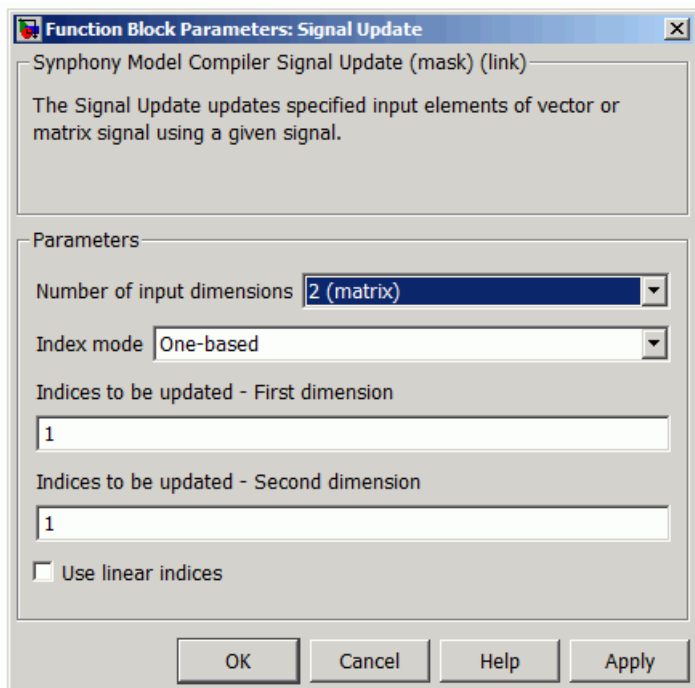
- Vector or matrix signal
- Data to be updated – You must use the same dimension as the index specification for the mask parameters.

The block outputs a vector/matrix signal with same dimension as the input signal containing the updated data.

Latency

This block has no latency.

Signal Update Parameters



Number of input dimensions

Specifies that either the input signal has 1 dimension (vector) or 2 dimensions (matrix). The default is 1 (vector).

Index mode

Specifies whether the index to be updated starts from zero (Zero-based) or one (One-based). The default is One-based.

Indices to be updated – First dimension

Specifies dimension 1 indices to be updated. You must specify the indices as a vector. For example:

[1 3 5] – Rows 1, 3, and 5

2:6 – Rows 2 through 6

Indices to be updated – Second dimension

Specifies dimension 2 indices to be updated. You must specify the indices as a vector. For example:

[2 4 6] – Columns 2, 4, and 6

Use linear indices

Use this option when it is useful to address matrix data using a single dimension. Linear indices are written column-wise. See the signal update examples below.

Example 1 – Signal Update Matrix

Parameters

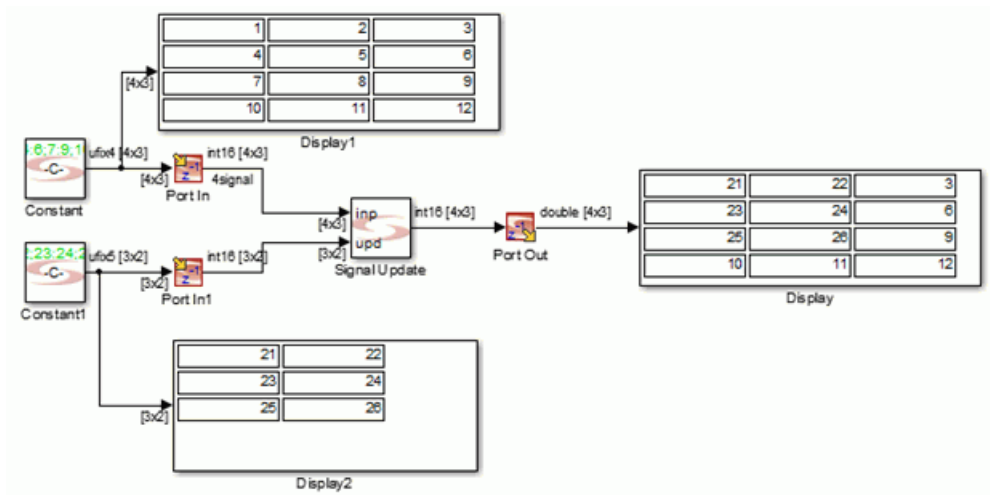
Number of input dimensions 2 (matrix)

Index mode One-based

Indices to be updated - First dimension
[1:3]

Indices to be updated - Second dimension
[1:2]

☐ Use linear indices



Example 2 – Signal Update Matrix Using Linear Indices

Parameters

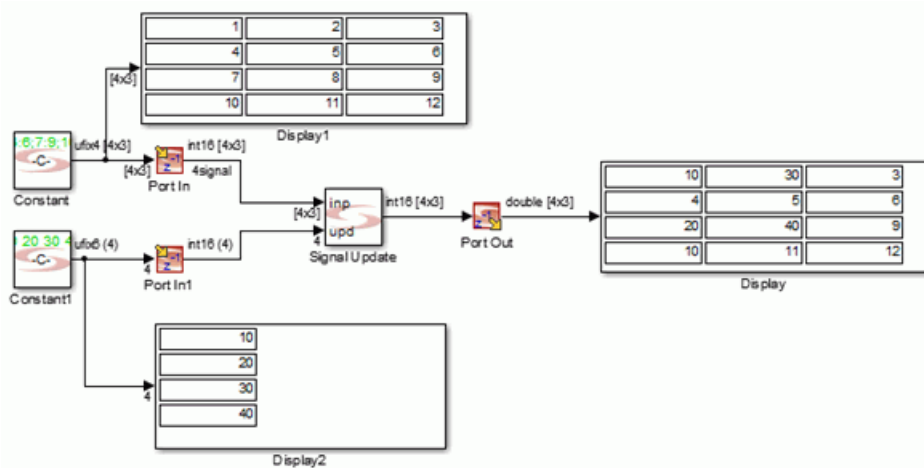
Number of input dimensions **2 (matrix)**

Index mode **One-based**

☒ Use linear indices

Linear indices

[1 3 5 7]



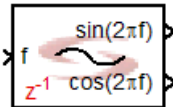
SMC SinCos

Calculates $A \sin(2\pi f)$ or $A \cos(2\pi f)$ for the input f , where A is the amplitude parameter.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler SinCos block calculates $A \sin(2\pi f)$ and/or $A \cos(2\pi f)$ for a scalar input. This implementation of sin/cos is based on a look-up table, the size of which is determined by the input fraction length and output word length.

This block requires a minimum of four fractional bits at the frequency input port. It only considers the fraction portion of the input. Given the scaling with 2π on the input, any integer portion corresponds to a full revolution of the trigonometric function, and therefore can be ignored while calculating the output value.

To keep the hardware implementation reasonable, the software only considers up to 18 fraction bits. If the input has more than 18 fraction bits, only the first 18 determine the value of the output. While determining the output for a given input, the block refers to a quadrant quantized in a look up table of size 2^{n+1} , and exploits quarter wave symmetry to produce outputs for other quadrants, where n is $\min\{16, \text{Input fraction length}-2\}$. This table shows examples for look up table entries:

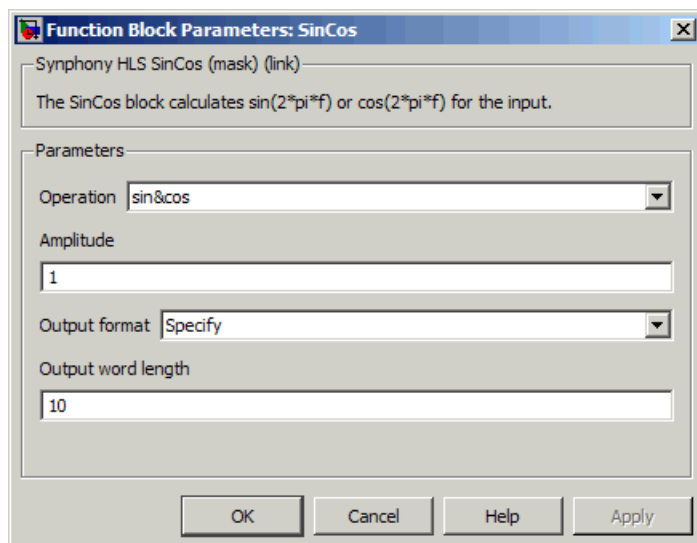
Input	Corresponding Degrees	Sin	Cos
1/12	30	0.5	0.866 ($\sqrt{3}/2$)
1/8	45	0.7071	0.7071 ($\sqrt{2}/2$)
1/4	90	1	0

Given the restriction on the input, the output accuracy is limited to 53 bits. Any request for a larger fraction on the output results in the same accuracy of 53 bits, zero-extended at the LSB side.

Latency

The latency of the SinCos block is 1.

SinCos Parameters



Function

Selects the operation to be performed:

- sin calculates $A \cdot \sin(2\pi f)$ for the input.
- cos calculates $A \cdot \cos(2\pi f)$ for the input.
- sin&cos calculates $A \cdot \sin(2\pi f)$ & $A \cdot \cos(2\pi f)$ for the input.

Amplitude

Determines the scaling of the output.

Output Format

Determines the word size and data type of the output. You can select one of the following settings for the output format:

- Automatic determines the output data format such that the output fraction length is equal to the input fraction length, and the output integer length is a minimum, causing no overflow for $[A, -A]$ range.
- Specify determines output data format such that output integer length is a minimum causing no overflow for $[A, -A]$ range, and the remaining space from the specified output word length becomes the output fraction length.

Output word length

Determines the word length of the output in bits.

SMC SinCos2

Creates sin and cos waveforms based on the input phase and amplitude values. The SMC SinCos2 block incorporates additional architectural and feature optimizations compared with the SMC SinCos block.

Library

Symphony Model Compiler [Math Functions](#)

Description



This block creates sin and cos waveforms based on the input phase and amplitude values. Phase precision, amplitude precision, and output precision can be specified independently. By selecting phase dithering, you can also flatten spurious noise components caused by the input phase quantization.

SinCos2 Multichannel Designs

To generate a multichannel SinCos2 block, set the number of channels to be greater than 1 and enable the Fold across channel option. The phase dither generation logic and sin-cos generation block, either CORDIC or LUT-based, are shared across all channels. When Fold across channel is enabled, the output is multiplexed by the specified folding factor.

For example, suppose you specify 8 channels with a folding factor of 4, then the output vector size is 2. The first element of the vector outputs channels [1, 2, 3, 4], time-multiplexed in this order and the second element outputs channels [5, 6, 7, 8], time-multiplexed similarly. The output sample time is $1/4^{\text{th}}$ the sample time value provided on the mask parameter or input ports.

For multichannel designs, the dimensions of `ssync` or `srnyi` ports must be the same as the number of channels. When the same `ssync` or `srnyi` input is provided to all channels, connect a Vector Expand block that expands the dimensions of the `ssync` or `srnyi` ports to match the number of channels.

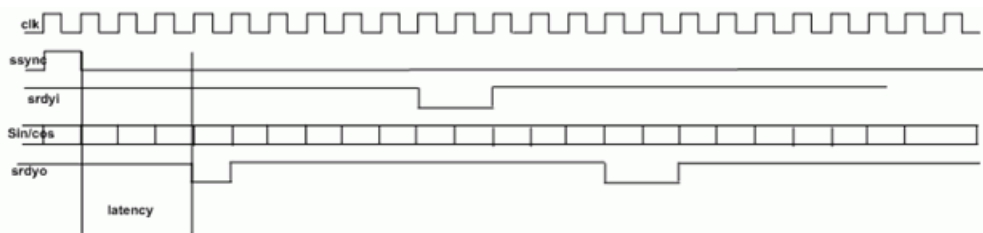
SinCos2 Flow Control

The SinCos2 block supports forward flow control through the optional `srdyi` and `ssync` input ports and the output `srdyo` port. The following table describes the ports:

<code>srdyi</code>	Qualifies whether or not the input is valid.
<code>ssync</code>	When the Latch inputs only on <code>ssync</code> option is enabled, the input port values are registered only when <code>ssync</code> is high; otherwise, any change to the port is ignored. If the corresponding value is a constant on the mask parameter, <code>ssync</code> is not affected as well.
<code>srdyo</code>	Qualifies whether or not the input is valid.

For multiple channels, you can provide independent `srdyi` and `ssync` ports for each channel by specifying a vector equal to the number of channels. When the Fold across channel option is enabled, the `srdyo` is multiplexed like the sin/cos output and is synchronous to the output. The sample time is the same as the `srdyo` output.

The following timing diagram illustrates the flow control operation:



Icon Annotations

The icon for this block displays the following information:

Top Annotation	The green annotation specifies the type of sin-cos generation, the folding factor, and the number of channels.
Latency Annotation	The red annotation at the bottom of the block specifies the latency value.

SinCos2 Parameters

The parameters for this block are displayed on following tabs:

- [Main Tab](#)
- [Optional Port Tab](#)
- [Data Types Tab](#)
- [Hardware Tab](#)

Main Tab

The Main dialog box displays general settings.

The screenshot shows the 'Main' tab of the SinCos2 block configuration dialog. It contains several settings: 'Function' is set to 'sin&cos', 'Method' is 'CORDIC', 'Phase angle input' is 'constant', 'Phase value (normalized to 2pi)' is '1/8', 'Phase dither' is checked, 'Phase dither word length' is '3', 'Amplitude' is 'constant', 'Amplitude value' is '3', and 'Number of channels' is '1'.

Tab	Function	Method	Phase angle input	Phase value (normalized to 2pi)	Phase dither	Phase dither word length	Amplitude	Amplitude value	Number of channels
Main	sin&cos	CORDIC	constant	1/8	<input checked="" type="checkbox"/>	3	constant	3	1

Function

Specifies the function for which the DDS2 block does calculations.
Choose one of the following functions:

- Sin

- Cos
- sin&cos

Method

Specifies the method used to generate the waveforms:

- LUT uses a lookup table containing the DDS output values to generate the waveforms.
- CORDIC uses CORDIC algorithms to generate the waveforms.

Phase Angle Input

Determines how to set the phase normalized to 2π .

- Constant sets the phase to the hard-coded value specified for the Phase Value.
- Port sets the frequency dynamically to the frequency of the input port.

Phase Value

Sets a constant value for the phase normalized to 2π . You must set Phase Angle Input to Constant, for this option to be available.

Amplitude

Specifies how to set the amplitude for the sin-cos waveform.

- Constant uses the hard-coded value specified for the Amplitude Value.
- Port uses the amplitude value set by the input port to modulate the frequency.

Phase Dither

Determines whether to improve the DDS spurious free dynamic range, using phase dithering. When this option is enabled, the software spreads the spurs through the available bandwidth to prevent phase error being introduced by the quantizer. The dithering sequence is added before quantization and uses the quantized value to index into the sine/cosine lookup table or CORDIC algorithm, mapping the phase space to time.

When disabled, the tool does not dither the signal.

Phase Dither Bits

When Phase Dither is enabled, you can determine how the precision of the dither output is computed.

- Automatic lets the tool automatically compute the difference between the waveform frequency word length and the waveform phase word length, where its value can range between 2 and 19.
- Specify lets you choose the phase dither word length.

Phase Dither Word Length

When Phase Dither Bits is set to Specify, you can set the word length for the dither generator output. If the dither word length is outside the range of [2, 19], then the tool automatically sets the word length to 2 and 19, respectively.

For both the automatic and specify Phase Dither Bits modes, if the tool needs to limit this value to 2 or 19:

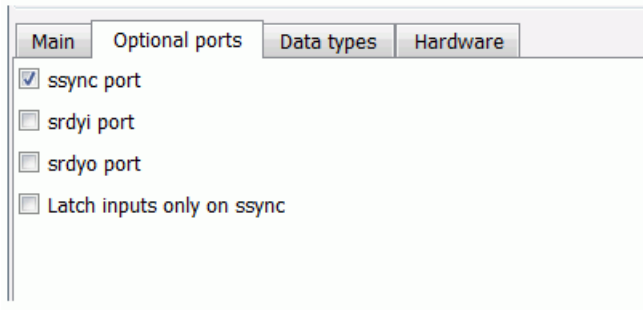
- The value you specified for waveform phase word length may be ignored and is automatically set to the frequency word length -2 or frequency word length -19.
- When the value you set leads to a zero or negative value of the phase word length, then the value you specified is retained.

Number of Channels

Specifies the number of output channels required.

Optional Port Tab

The ports on the Optional Port dialog box provide flow controls for the block. See [DDS2 Flow Control, on page 151](#) for additional information about these ports.

**ssync port**

When enabled, the block includes the ssync input port. This also makes the corresponding srdyo output port available.

srdyi port

When enabled, the block includes the srdyi input port. This also makes the corresponding srdyo output port available.

srdyo port

When enabled, the srdyo output port is available for the block. If either the ssync or srdyi port is enabled, then the srdyo port is always available at the output.

Latch inputs only on ssync

When enabled, the tool accepts and registers the input at the frequency, phase modulation, or frequency modulation port when ssync is high. This option becomes available when the ssync port is enabled and at least one of the inputs is through a port.

Data Types Tab

The screenshot shows the 'Data types' tab of the SMC SinCos2 block configuration. It contains several input fields and dropdown menus for configuring the data types and output format. The 'Phase fraction length' is set to 17, 'Amplitude fraction length' is 20, 'Output format' is 'Specify', 'Output word length' is 22, 'Output fraction length' is 20, 'Output data type' is 'signed', and 'Output round on underflow' is 'Floor (Truncate)'.

Parameter	Value
Phase fraction length	17
Amplitude fraction length	20
Output format	Specify
Output word length	22
Output fraction length	20
Output data type	signed
Output round on underflow	Floor (Truncate)

Phase Fraction Length

Specifies the fraction length of the phase. The word length can be computed from the fraction length when any phase greater than 2π is wrapped around.

Amplitude Fraction Length

Specifies the fraction length of the amplitude. The word length can be computed from the fraction length if the amplitude input is constant.

Amplitude Word Length

Specifies the word length of the amplitude, only when the amplitude is available through a port.

Output format, Output word length, and Output fraction length

For a description of the available output formats, output word length, and output fraction length for this block, see the SMC Convert block [Output format, Output word length, and Output fraction length](#), on page 101.

Hardware Tab

The screenshot shows the Hardware Tab of the SMC SinCos2 block configuration. The 'Data types' tab is selected, and the 'Hardware' tab is also visible. The configuration options are as follows:

- ☐ Fold across channels
- ☒ Pipeline sin-cos LUT
- ☒ Split phase LUT compression
- Target device:
- Optimization target:

The 'Data types' tab is selected, and the 'Hardware' tab is also visible. The configuration options are as follows:

- ☐ Fold across channels
- CORDIC parameters:
- Number of stages:
- CORDIC Latency:
- Stage output rounding:

Fold across channels

When enabled, specifies that channels are time division multiplexed. This option requires that the number of channels is greater than 1.

Folding Factor

Specifies the time division multiplexing factor across channels. This option requires that you enable Fold across channels.

The number of channels must be an integer multiple of the folding factor. Otherwise, the tool performs zero padding at the input to increase the number of channels to be a multiple of the folding factor. The zeros are not removed at the output.

Sample Time

Specifies sample time. Use -1 to inherit. If you specify any option that results in an input port being available, then this option is not available.

Available options vary, depending on whether you choose CORDIC or LUT as the Method. If LUT is selected, the following options are available.

Pipeline sin-cos LUT

This option automatically becomes available, when you enable the Fold across channels option. You must specify LUT as the Method to use this option. When enabled, the tool internally uses a fully pipelined optimized Quarter-wave LUT architecture that significantly increases the maximum achievable clock frequency. Turn on this option to optimally map the FPGA, unless the DDS2 block is used in a feedback path such as a Costas loop.

LUT Compression

When enabled, the LUT size reduces from 2^{phaseWL} to $3 \cdot 2^{(\text{phaseWL}/2)}$ at the cost of an extra complex multiplier. As the phase word length increases, this becomes a valuable option that lowers the block RAM utilization on the FPGA. If Pipeline sin-cos LUT and LUT compression mode are both enabled, the following options are additionally available:

- Target device

Specifies that the generated RTL is optimized for Virtex 5, Virtex 6/7 or Stratix 5.

- Optimization target

Specifies whether the generated RTL is optimized for Speed or Area.

If CORDIC is selected, the following options are available.

CORDIC parameters

If Automatic mode is set for this option, the tool automatically selects the optimal options depending on the other parameters specified. When this is set to Specify, following options are additionally available:

- Number of stages

Specifies the number of stages in the CORDIC.

- CORDIC Latency

Specifies the latency of the CORDIC.

- Stage output rounding

Specifies the rounding mode to use on the x, y and angle output for each stage of the CORDIC. The supported options are the following: Floor(Truncate), Nearest, Convergent, Fix, Ceil, and Round.

SMC Single Clock Downsample

Provides variable rate and single clock downsample operations.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Single Clock Downsample custom block complements the features of the SMC Downsample block with the following functionality:

- Variable rate and variable offset downsample
- Single clock downsample, where sample and hold are at the same clock as the input

Latency

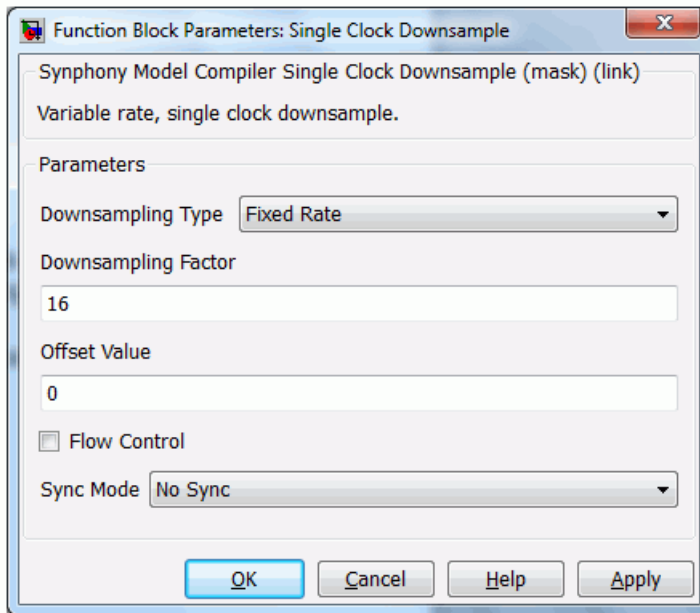
The block has a latency of 1 to the output sample domain.

Single Clock Downsample Flow Control

The Single Clock Downsample block provides the following optional flow control ports:

srnyi	The srnyi (source ready) input port determines whether the input data in the current sample period is valid. An invalid input sample is indicated by srnyi going low.
srnyo	The srnyo (source ready) output port determines whether the current output sample is valid. An invalid output sample is indicated by srnyo going low.

Single Clock Downsample Parameters



Downsampling Type

Specifies whether to perform fixed rate or variable rate downsampling. For both options, the output is in the same clock domain as the input.

Downsampling Factor

When fixed rate downsampling is selected, you can use this option to specify the constant downsampling factor.

Offset Port

When variable rate downsampling is selected, determines whether the offset port is available at the input to dynamically vary the sample offset.

Offset Value

If fixed rate downsampling or variable rate downsampling (when the Offset Port option is disabled) is selected, then you can specify the constant sample offset value for performing downsampling.

Flow Control

Specifies whether the `srdyi/srdo` ports are available on the block interface.

Sync Mode

Specifies the synchronization mode for the output. Do not use this option when Flow Control is enabled, which is the preferred implementation. This option is mainly used to provide compatibility for the SMC Downsample legacy block. When the clock counter reaches the position you specified with these options, the synchronized output produces 1. The output synchronization can be one of the following modes:

Mode	Description
No Sync	The output is not synchronized.
Aligned with Offset	The output is synchronized with the offset.
Right before Offset	The output is synchronized with one sample before the offset.

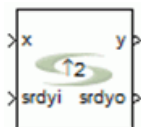
SMC Single Clock Upsample

Provides variable rate and single clock upsample operations.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Single Clock Upsample custom block complements the features of the SMC Upsample block with the following functionality:

- Variable rate and variable offset upsample
- Single clock upsample, where sample and hold are at the same clock as the input

Latency

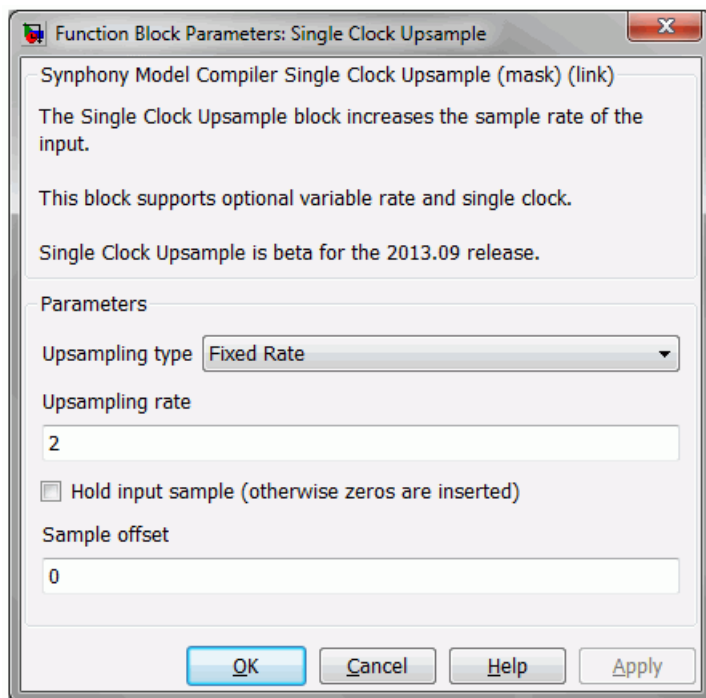
This block has no latency.

Single Clock Upsample Flow Control

The Single Clock Upsample block provides the following mandatory flow control ports:

srdyi	<p>The srdyi (source ready) input port determines whether the input data in the current sample period is valid. An invalid input sample is indicated by srdyi going low.</p> <p>The srdyi cannot be active for more than 1 cycle of every L, where L is the upsampling rate. Otherwise, the behavior is undefined for this block.</p>
srdyo	<p>The srdyo (source ready) output port determines whether the current output sample is valid. An invalid output sample is indicated by srdyo going low.</p> <p>The srdyo is automatically pulse stretched by L, where L is the upsampling rate.</p>

Single Clock Upsample Parameters



Upsampling Type

Specifies whether to perform fixed rate or variable rate upsampling. For both options, the output is in the same clock domain as the input.

Upsampling Rate

When the fixed rate option is selected, you can specify a value that the input sample rate is multiplied by to get the output sample rate.

For a single clock block, the actual sample rate does not change. The rate change is identified through the flow control signals, `srnyi` and `srnyo`.

- `srnyi` cannot be active for more than 1 cycle of L , where L is the upsampling rate. The behavior is undefined if this constraint is not met for this block.
- `srnyo` is pulse stretched by L cycles, which indicates the new data rate for the output of the block.

Hold Input Sample

When enabled, this option holds the input sample. The block copies the input as the first sample of every output frame (L samples) and holds the sample value for the other samples in the output frame.

When hold input sample is enabled, sample offsets are not supported.

Sample Offset Port

When variable rate upsampling is selected, determines whether the offset port is available at the input to dynamically vary the sample offset.

Sample Offset

Specifies the sample offset value for performing upsampling. When the Sample offset port option is enabled, this option is not available.

SMC Smart Black Box

Lets you embed third-party IP in a Symphony Model Compiler design and automatically cosimulate it.

Library

Symphony Model Compiler [Ports & Subsystems](#)

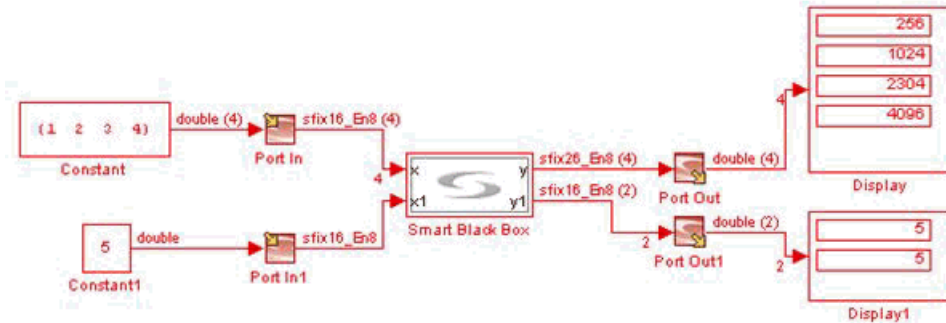
Description



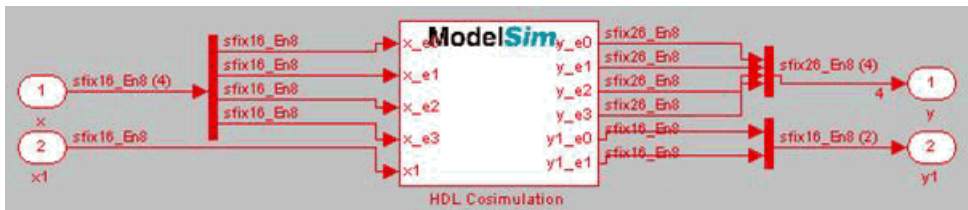
The Symphony Model Compiler Smart Black Box lets you embed third-party blocks for which you have access to the RTL code. If you have IP with no access to the RTL code, use the Black Box block ([SMC Black Box, on page 56](#)) instead.

The Smart Black Box requires that you have a license for EDA Simulator Link MQ® (formerly Link for ModelSim), which is a cosimulation interface between Simulink and the ModelSim HDL simulator. The cosimulation interface must be configured using the SynCoSimTool block ([SMC SynCoSimTool, on page 550](#)). Symphony Model Compiler uses EDA Simulator Link MQ to ls and simulate the embedded RTL-level models. The Simulink simulation is transparent, but the RTL generated by Symphony Model Compiler treats the IP as a black box. See [Using Smart Black Boxes for Cosimulation, on page 837](#) for details.

The Smart Black Box block supports vector inputs. The length of the vector should be specified in the configuration file (see [Configuration File For Smart Black Box, on page 535](#)). The following figure illustrates how vector input and output are handled in a Smart Black Box block.



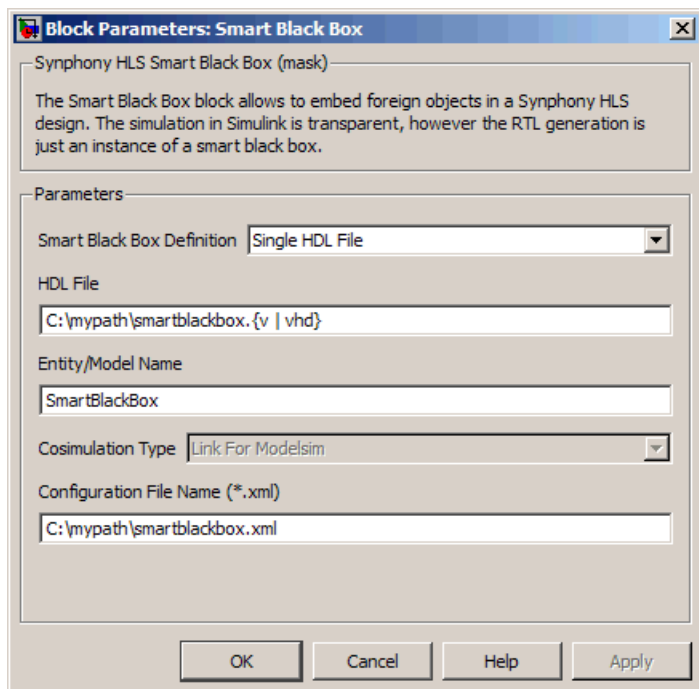
When the input to the Smart Black Box is a vector, the tool demultiplexes the vector input inside, and then transfers it to the ModelSim HDL Cosimulation block. It then multiplexes the output of ModelSim HDL Cosimulation block, and the output of the Smart Black Box again becomes a vector.



Latency

Latency is determined by the contents of the third-party HDL source, plus one more from the cosimulation infrastructure.

Smart Black Box Parameters



Smart Black Box Definition

Specifies how the third-party IP is defined. You can choose one of the following:

- Single HDL file
Use this if the IP is defined in a single `.v` (Verilog) or `.vhd` (VHDL) file. Specify the path and file name in HDL File.
- Import File list
Use this if the IP is defined in multiple `.v` (Verilog) or `.vhd` (VHDL) files. Specify the text file that contains a list of the HDL files in Black Box File List.

HDL File

Specifies the absolute path to the file that contain the smart black box definition. This file is added to the logic synthesis project file and to the simulator `.do` files. This option is only available when Smart Black Box Definition is set to Single HDL File.

HDL File or Black Box File List

Specifies the absolute path to a single text file that lists all the HDL files that define the smart black box. This option is only available when Smart Black Box Definition is set to Import File List.

The list must contain absolute paths to the files. The definition file extensions in the list must be `.v` or `.vhd`. For example, if your smart black box is defined in four files called `sbb1lib1.vhd` (library definition file with `sbb1lib` being the library name) and the other files are `sbb2.v`, `sbb3.v` and `sbb4.vhd`, create and save a text file (`sbblist.txt`) that lists the absolute paths to the smart black box definition files as follows:

```
-L sbb1lib C:\mypath\sbb1lib1.vhd
C:\mypath\sbb2.v
C:\mypath\sbb3.v
C:\mypath\sbb4.vhd
```

Entity/Model Name

Specifies the top-most entity or model name for the smart black box. This name becomes the instance name for the smart black box and the name of the instantiated entity or model.

Cosimulation Type

Specifies the tool used for cosimulation. Currently, the only choice is EDA Simulator Link MQ.

Configuration File

Specifies an `.xml` configuration file that describes the top-most entity or model ports, clock properties and global reset and enables. See [Creating Smart Black Box Configuration Files, on page 841](#) for information about creating this file, and [Configuration File For Smart Black Box, on page 535](#) for file format details.

Configuration File For Smart Black Box

The configuration file is an `.xml` file that contains port, clock, global enable and global reset information. See the following for syntax details:

- [Ports, on page 536](#)
- [Clocks, on page 537](#)
- [Global Enables, on page 537](#)

- [Global Resets, on page 537](#)
- [Sample Configuration File, on page 538](#)

Ports

The following illustrates how ports are described. Note the following terms:

PortModes	Specifies port mode, which can be IN or OUT.
PortPaths	Contains Port name.
PortTimes	Specifies the sampling time of the output port. If PortModes is IN (input port), this is meaningless and can be discarded. Otherwise, it must be a number.
PortSigns	Specifies the data type of the output port. If PortModes is IN (input port), this is meaningless and can be discarded. It can be “Inherit”, “Unsigned” or “Signed”
PortFractionLengths	Specifies the data fraction length of the output port. If PortModes is IN (input port), this is meaningless and can be discarded.
PortWidth	Specifies the width of the port. If the input or output port is a vector, this specifies the length of the vector.

```

<Ports>
  <Port>
    <PortModes>IN</PortModes>
    <PortPaths>In_PortName </PortPaths>
    <PortWidth>1</PortWidth>
  </Port>|
  <Port>
    <PortModes>OUT</PortModes>
    <PortPaths>Out_PortName</PortPaths>
    <PortTimes>5e-9</PortTimes>
    <PortSigns>Signed</PortSigns>
    <PortFracLengths>8</PortFracLengths>
    <PortWidth>1</PortWidth>
  </Port>
</Ports>

```

Clocks

The following shows how the sample clocks are described. The example uses the following terms:

ClockPath	Clock name.
-----------	-------------

ClockMode	Defines the system clock edge. It can be Falling or Rising.
-----------	---

ClockTime	Specifies the system clock period.
-----------	------------------------------------

```
<Clocks>
  <Clock>
    <ClockPath>ClockName</ClockPath>
    <ClockMode>Rising</ClockMode>
    <ClockTime>5e-9</ClockTime>
  </Clock>
</Clocks>
```

Global Enables

The following shows how the global enables are described. The example uses this term:

GlobalEnablePath	Global enable name.
------------------	---------------------

```
<GlobalEnables>
  <GlobalEnable>
    <GlobalEnablePath>GlobalEnableName</GlobalEnablePath>
  </GlobalEnable>
</GlobalEnables>
```

Global Resets

The following shows how the global enables are described. The example uses this term:

GlobalResetPath	Global reset name.
-----------------	--------------------

```
<GlobalResets>
  <GlobalReset>
    <GlobalResetPath>GlobalResetName</GlobalResetPath>
  </GlobalReset>
</GlobalResets>
```

Sample Configuration File

```
<SBBParams>
<Ports>
  <Port>
    <PortModes>IN</PortModes>
    <PortPaths>x_in</PortPaths>
    <PortWidth>1</PortWidth>
  </Port>
  <Port>
    <PortModes>OUT</PortModes>
    <PortPaths>y</PortPaths>
    <PortTimes>5e-9</PortTimes>
    <PortSigns>Signed</PortSigns>
    <PortFracLengths>0</PortFracLengths>
    <PortWidth>1</PortWidth>
  </Port>
</Ports>
<Clocks>
  <Clock>
    <ClockPath>clk</ClockPath>
    <ClockMode>Rising</ClockMode>
    <ClockTime>5e-9</ClockTime>
  </Clock>
</Clocks>
<GlobalEnables>
  <GlobalEnable>
    <GlobalEnablePath>GlobalEnable1</GlobalEnablePath>
  </GlobalEnable>
</GlobalEnables>
<GlobalResets>
  <GlobalReset>
    <GlobalResetPath>GlobalReset</GlobalResetPath>
  </GlobalReset>
</GlobalResets>
</SBBParams>
```

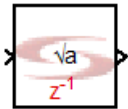
SMC Sqrt

Calculates the square root of the input.

Library

Synphony Model Compiler [Math Functions](#)

Description



The Synphony Model Compiler Sqrt block calculates the square root of the input. The implementation of the square root is based on a look-up table. See [Implementation Details, on page 540](#), for more information. This block also supports vector input.

The output word length is half the input word length, and the output fraction word length is half the input fraction word length. For odd input word length and input fraction length values, the output bit and fraction word lengths are rounded upwards. For signed input, the sign bit is discarded and the input is treated as unsigned. For a 9-bit signed input with 3 fraction bits, the Sqrt block sets the output length to 5 and the number of output fraction bits to 2. The following table illustrates:

	Input Parameters	Sqrt Block Output
Unsigned input	Input word length = 9 Input fraction length = 3	Output word length = $(9+1)/2+1 = 6$ Output fraction length = $(3+1)/2 = 2$
Signed input	Input word length = 9 Input fraction length = 3	Sign bit is discarded and the input is treated as unsigned Output word length = $(8+1)/2+1 = 5$ Output fraction length = $(3+1)/2 = 2$

Implementation Details

The Symphony Model Compiler software uses normalization to improve precision. The Symphony Model Compiler implementation of the Sqrt block uses a look-up-table of 768 entries, containing the square roots of integers from 256 to 1024. The input number is first normalized into this range by left or right shifts of an even count. Then, the Symphony Model Compiler software accesses the look-up-table using the integer part of this normalized number as the index. Finally, it shifts this table lookup result by half the normalization shifts.

To take a specific example, an input x is first converted into the form $x = 2^{(2N)} \cdot x_n$. The normalized x is $256 \leq x_n < 1024$. Then the square root is calculated as follows: $\text{sqrt}(x) = 2^N \cdot \text{sqrt_table}(\text{int}(x_n) - 256)$.

This method improves output precision for smaller numbers, because it puts an upper bound on the percentage error. It also prevents the excessive use of memory for the computation of square roots of very large numbers. If you need high-precision square roots of large numbers, use the CORDIC Sqrt block ([SMC CORDIC Sqrt, on page 126](#)).

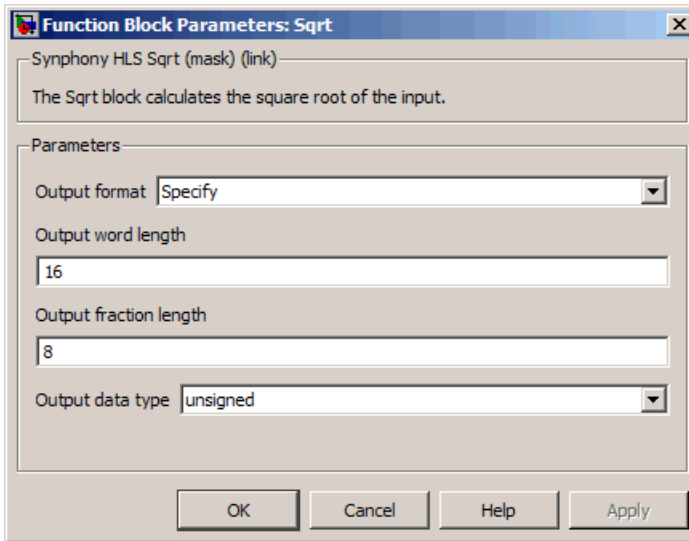
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

This block has a latency of 1.

Sqrt Parameters

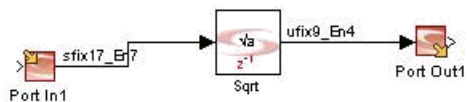
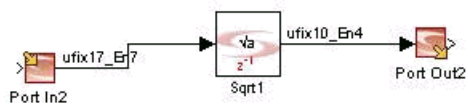


Output format, Output word length, Output fraction length, and Output data Type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583 If Output format is Automatic, the input sign bit is discarded and the output is unsigned. If Output format is Specify, the word length, fraction length, and data type are as specified.
Output word length	Output Word Length, on page 584 $\text{Output word length} = \text{input word length} / 2 + 1$
Output fraction length	Output Fraction Length, on page 584 If the fraction length of the input is odd, the tool adds another 0-valued fraction bit to the input. $\text{Output fraction length} = \text{input fraction length} / 2$
Output data type	Output Data Type, on page 584

Sqrt Block Examples



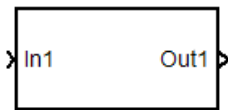
SMC Subsystem

Allows you to add a subsystem to a Symphony Model Compiler design.

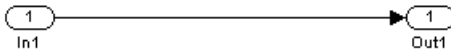
Library

Symphony Model Compiler [Ports & Subsystems](#)

Description



The Symphony Model Compiler Subsystem block provides a template for a subsystem. It consists of an input and an output block, to which you can add other blocks:



For more information about this block, refer to the Simulink documentation. For information on using it, refer to [Managing Subsystems and Hierarchy, on page 786](#).

Latency

The latency of this block is determined by its contents.

SMC Sum of Products

Multiplies inputs with gain values and calculates the sum of the computed products to provide a scalar output.

Library

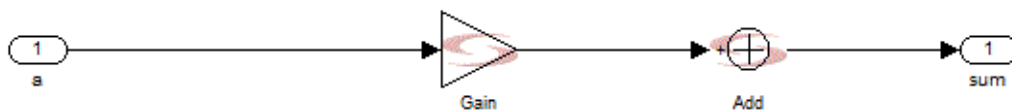
Synphony Model Compiler [Math Functions](#)

Description



The Sum of Products block multiplies inputs with *Gain* values and calculates the sum of the computed products to provide a scalar output. You can specify Gain values as constants, or provide them through the *Gain* port. If *Gain* is constant, either specify it as a scalar, where the same value is applied to all inputs, or as a vector, where the dimensions have to be the same as the number of elements.

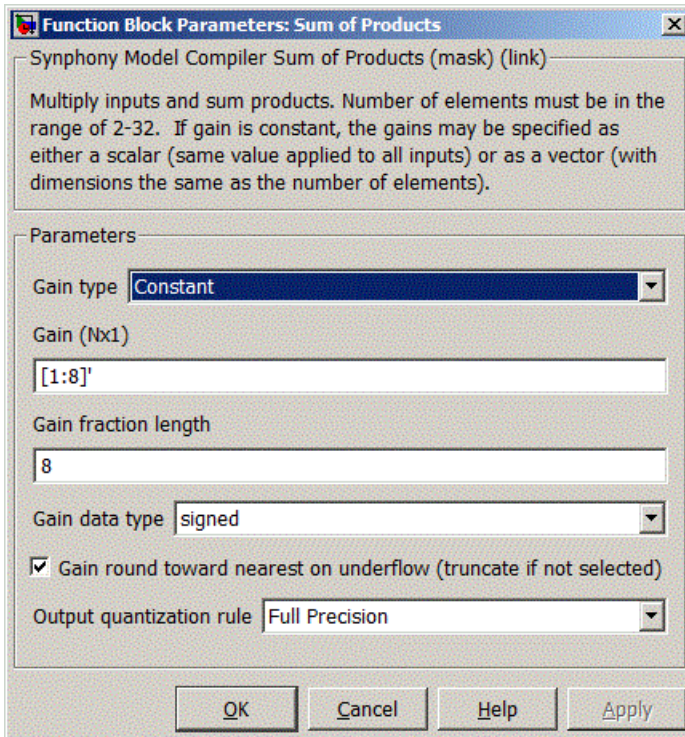
This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition). The following figure shows the internal modeling when *Gain* is a constant:



Latency

This block has no latency.

Sum of Products Parameters



The dialog box titled "Function Block Parameters: Sum of Products" contains the following information:

Symphony Model Compiler Sum of Products (mask) (link)

Multiply inputs and sum products. Number of elements must be in the range of 2-32. If gain is constant, the gains may be specified as either a scalar (same value applied to all inputs) or as a vector (with dimensions the same as the number of elements).

Parameters

Gain type: Constant

Gain (Nx1): [1:8]

Gain fraction length: 8

Gain data type: signed

☒ Gain round toward nearest on underflow (truncate if not selected)

Output quantization rule: Full Precision

Buttons: OK, Cancel, Help, Apply

Gain type

Specifies the mode of entering the gain values. Select one of the following settings:

- Constant
The gain value is constant, and is defined in the Gain (NX1) parameter.
- Port
The gain value is provided through an input port.

Gain (NX1)

Specifies the gain value. This field is available only if Gain type is set to Constant. Specify the value as a scalar if you want the same gain for all the inputs. Alternatively, specify it as a vector, with the length equal to the length of the input vector.

Gain fraction length

Specifies the accuracy of the fraction requested for the coefficient value. The software infers the total word length of the coefficient automatically from the specified value.

Gain data type

Determines the data type for the gain value (specified in the Gain option) for the block. You can set it to signed or unsigned value.

Gain round towards nearest on underflow

Determines how the underflow for the gain is treated. When it is enabled, the tool rounds the underflow using the Nearest algorithm.

When the option is disabled, the tool rounds the overflow with the Floor (truncate) algorithms. See [Underflow Rounding Options, on page 585](#) for details.

Output quantization rule, Output word length, Output fraction length, and Output data type

For descriptions of these parameters, see the following:

Output quantization rule	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

Output saturate on overflow, Output round on underflow

Determine how overflow and underflow are treated. These options are only available when Output format is set to Specify.

Output saturate on overflow	When enabled it saturates the overflow; when disabled, it wraps the overflow. See Overflow Saturation Options, on page 585 for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see Underflow Rounding Options, on page 585 for descriptions of the algorithms.

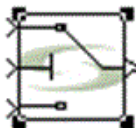
SMC Switch

Pass through input 1 when input 2 satisfies the selected criterion; otherwise, pass through input 3.

Library

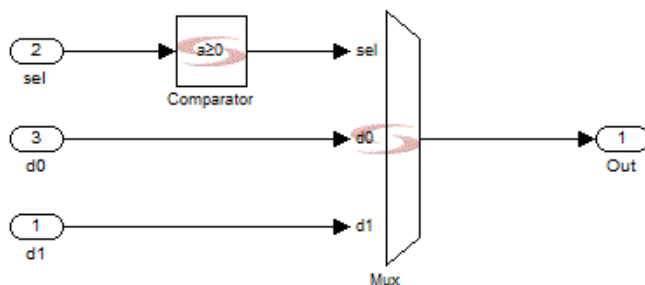
Symphony [Signal Operations](#)

Description



The Switch block passes through input 1 when input 2 satisfies the selected criterion; otherwise, passes through input 3. The inputs are numbered top to bottom (or left to right). The first and third input ports are data ports, and the second input port is the control port.

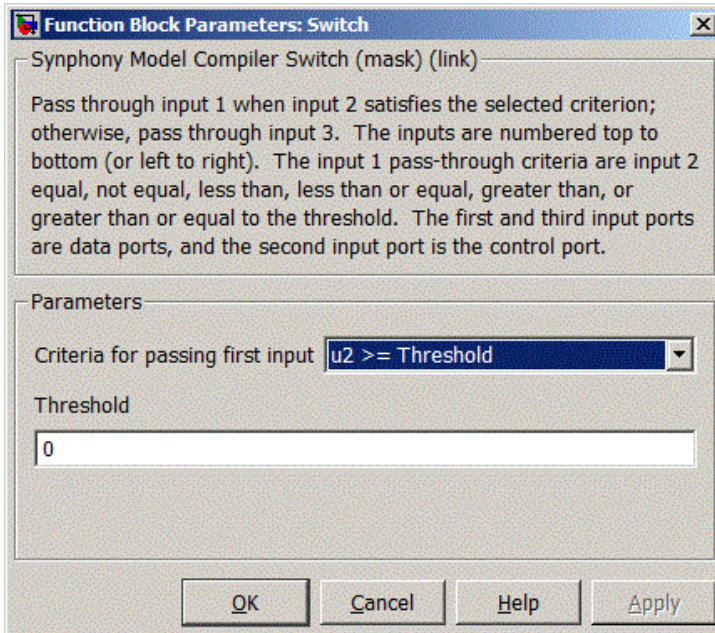
This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition). The following figure shows the internal modeling of default mask parameter values:



Latency

This block has zero latency.

Switch Parameters



Criteria for passing first input

Specifies one of the several criteria for passing signal through input port with respect to control port, and the pass-through criteria are as follows:

- input 2 = threshold
- input 2 \neq threshold
- input 2 < threshold
- input 2 \leq threshold
- input 2 > threshold
- input 2 \geq threshold

Threshold

Specifies the comparison threshold of control port to select either the input or the data port.

SMC SynCoSimTool

Manages communication between smart black boxes and the RTL cosimulation interface.

Library

Synphony Model Compiler, top level library

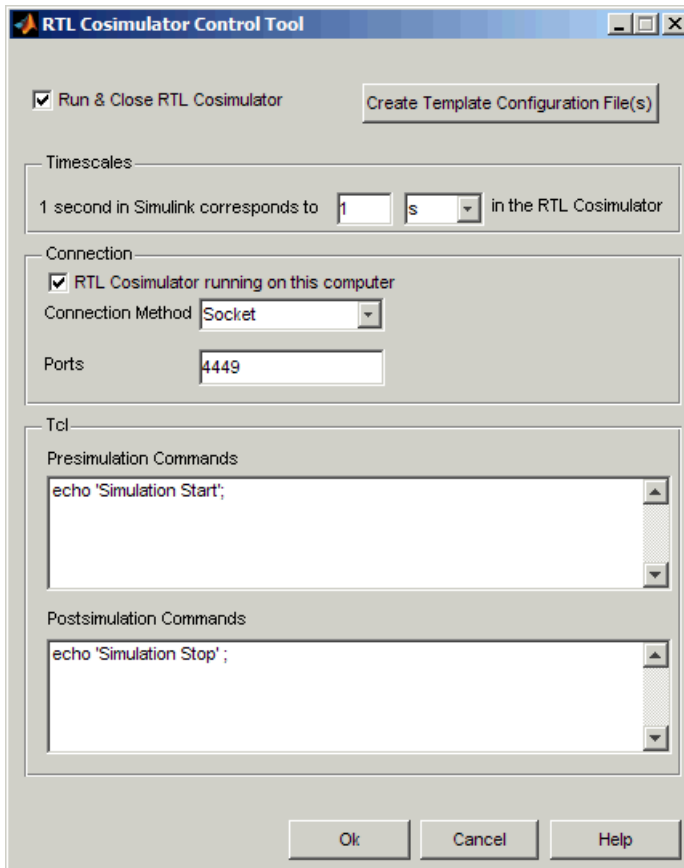
Description



The SynCoSimTool block controls the interaction between a smart black box (see [SMC Smart Black Box, on page 532](#)) and the RTL simulator. It must be configured with proper communication parameters. Once it has been configured, communication is established with MATLAB and you can run RTL cosimulation automatically.

For further information about using this tool and setting up the cosimulation interface, see [Using Smart Black Boxes for Cosimulation, on page 837](#).

SynCoSimTool Parameters



Run and Close RTL Cosimulator

Specifies options for running the cosimulator.

- If the box is enabled, the RTL cosimulator runs automatically and closes when it is done.
- If the box is disabled, the RTL cosimulator runs automatically for the first Simulink simulation, but it does not close when cosimulation is complete.

Enabling this option means that the cosimulator starts every run from an initial state, clearing its state elements. Initialization times could impact run times. On the other hand, disabling this option means that

the RTL cosimulator is only initialized the first time and continues from its last state at every subsequent simulation run. Therefore, if the external RTL source does not depend on initial state values, leave this box unchecked to get better run times.

Timescales

Lets you specify the timing relationship between Simulink and the RTL cosimulator. The value you enter and the selected timescale (Tick, fs, ps, ns, us, ms, or s) correspond to one second in Simulink. With the default setting, 1 second in Simulink corresponds to 1 s in RTL.

RTL Cosimulator running on this computer

Determines whether the RTL cosimulator is located on the same computer. Depending on the setting, other options available that determine the mode of connection. By default, this option is enabled.

Connection Method

Specifies the connection method to the RTL cosimulator. This option is only available when RTL Cosimulator running on this computer is enabled.

- Socket
Connects Simulink and the RTL cosimulator through the socket connection specified in Port. This is the default setting.
- Shared Memory
Connects Simulink to the RTL cosimulator using shared memory. When selected, the RTL cosimulator does not close after a run, even if Run and Close RTL Cosimulator is enabled.

For additional information, see [Configuring the Cosimulation Interface, on page 839](#).

Ports

Specifies the socket connection port. The port value you enter is used as a TCP/IP connection port. The registered port numbers for general use are from 1024 to 49151. If the design contains one smart black box, the port value is set to 4449 by default. If there are additional smart black boxes, the tool starts with 4449 and increments this value by one for each black box. If there are two smart black boxes, the ports are set to 4449 and 4450, respectively.

Host Name

Specifies the host name of the computer where the cosimulator is located. The RTL cosimulator must be running before you start Simulink.

Apply and create do file

Creates a .do file for simulation. Clicking this button saves all data from the SynCoSimTool block and creates the appropriate .do file in ../model-path/synwork. The RTL cosimulator file is called synSBB.

Presimulation Commands

Lets you specify tcl commands to be run before simulation. The commands you enter are executed on the RTL cosimulator before the HDL code is simulated. The tcl commands must be written with one command per line, or must be separated by semi semicolons(;).

Postsimulation Commands

Lets you specify tcl commands to be run after simulation. The commands you enter are executed after the HDL code is simulated. The tcl commands must be written with one command per line, or must be separated by semi semicolons(;).

Create Template Configuration File

Creates a template configuration file based on the options you set. The file is saved in the ../modelpath directory with SBB block names. You must manually modify the files with the correct port, clock, global enable and global reset settings. For information about working with this file and the file format, see [Creating Smart Black Box Configuration Files, on page 841](#) and [Configuration File For Smart Black Box, on page 535](#).

SMC SynFixPtTool

Opens the Simulink Fixed-Point interface.

Library

Synphony Model Compiler, top level library

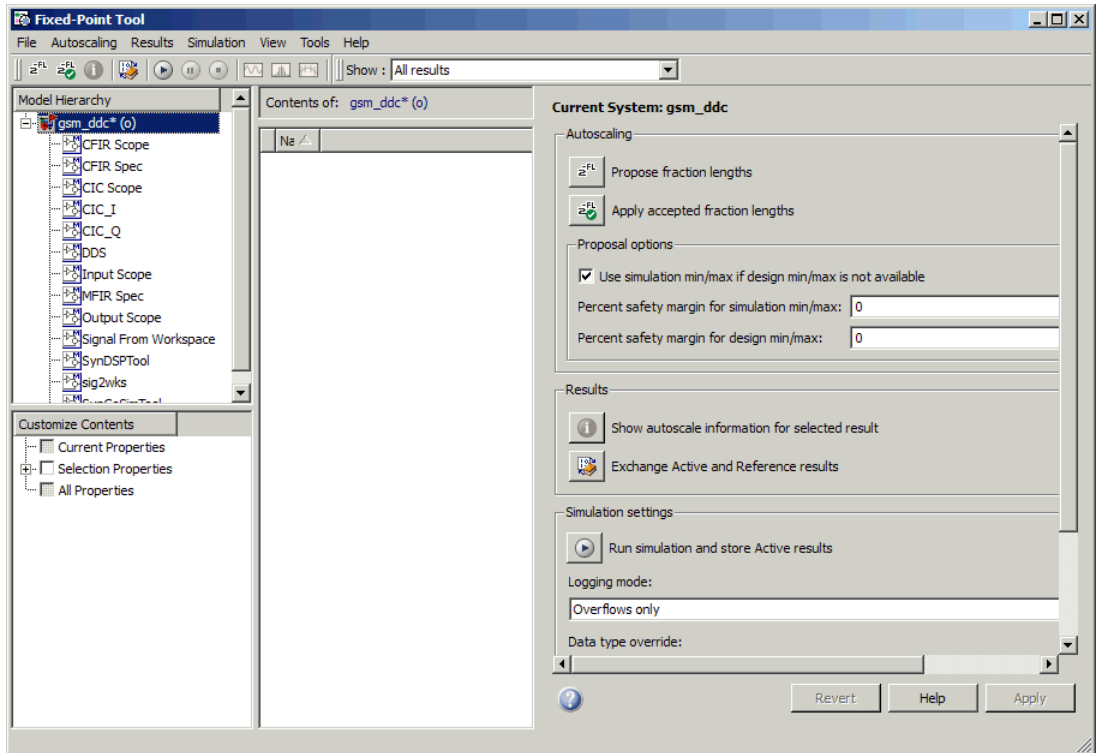
Description



The Synphony Model Compiler SynFixPtTool toolbox opens the Simulink Fixed-Point Settings interface where you can conveniently access global data type overrides and logging settings, the logged data, the automatic scaling script, and the Plot System interface.

The Fixed-Point Settings GUI is an optional Simulink package, and the Synphony Model Compiler SynFixPtTool toolbox will not function properly unless it is installed. Note that the appearance of the Simulink Fixed-Point Settings interface varies, depending on the MATLAB version you have installed.

For detailed information about the Simulink interface, type `doc fxptdlg` at the MATLAB prompt. For information about using the fixed-point data type, see [Using Quantization Analysis Tools, on page 832](#).



SMC Test Vector Capture

Toggles between setting or resetting Port In and Port Out Capture Test Vector mode for all Symphony Model Compiler ports.

Library

Symphony Model Compiler [Ports & Subsystems](#)

Description



The Symphony Test Vector Capture toggles between setting or resetting Port In and Port Out Capture Test Vector mode for all Symphony Model Compiler ports. This block is a custom block (see [Primitives and Custom Blocks, on page 800](#) for a definition).

Latency

This block has no latency.

Test vector Capture Parameters

There are no user visible parameters for this block. Clicking on the block changes the state from OFF to ON and vice versa.

This block has no input and output ports and should be instantiated in the model at the top level diagram.

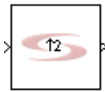
SMC Upsample

Increases the sample rate of the input by inserting zeroes.

Library

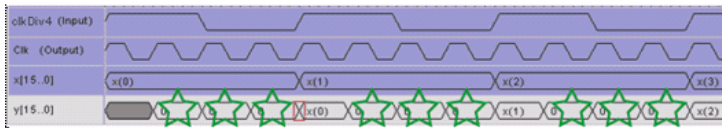
Synphony Model Compiler [Signal Operations](#)

Description

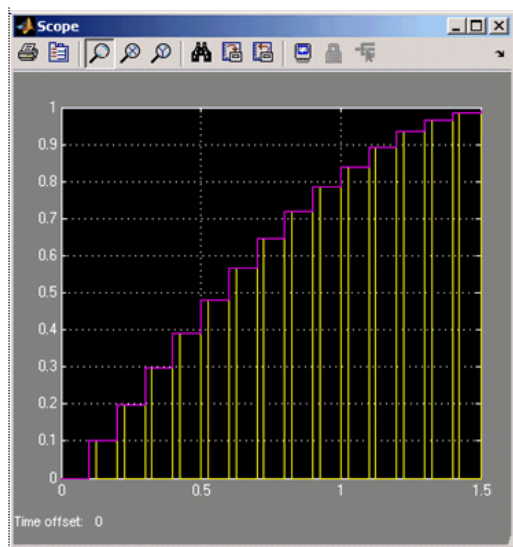


The Synphony Model Compiler Upsample block upsamples the sample rate of the input by adding samples. With an upsampling rate L , for every sample at the input, the software inserts $L-1$ samples at the output. This means that the sample rate at the output is the input sample rate multiplied by the upsampling rate, L . From a hardware implementation point of view, the easiest way to insert zeroes is to clock the input signal with the higher clock, and reset the flip-flop for the remaining $L-1$ clock cycles.

This figure shows the corresponding signal manipulation, with implementation clock and signal dependencies:



The following figure shows a practical simulation result for the implementation:



The software uses a delay at the input (based on the input sample rate) followed by a standard upsample operation, where it copies the input as the first sample of every output frame (L samples), and inserts $L-1$ zeroes for the other samples in the output frame.

For information about using the Upsample block in multi-rate designs, see [Multi-Rate Design, on page 717](#).

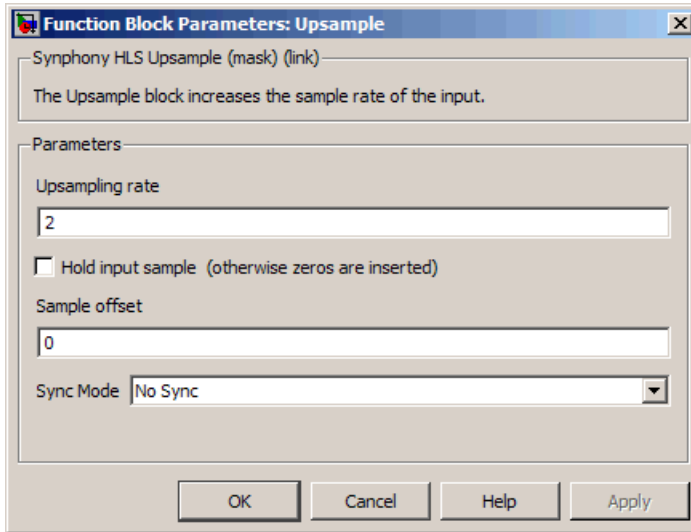
Constant Propagation

The tool propagates constants for this block. See [Constant Propagation, on page 731](#) for a description.

Latency

The latency of the Upsample block is at the output, and is equal to the sample offset.

Upsample Parameters



Upsampling rate

Specifies the value by which the input sample rate is multiplied to get the output sample rate.

Hold input sample

When enabled, this option holds the input sample. The software copies the input as the first sample of every output frame (L samples), and holds the sample value for the other samples in the output frame.

If disabled, the tool inserts zeroes. It copies the input as the first sample of every output frame (L samples), and inserts L-1 zeroes for the other samples in the output frame. When it is unchecked, Sample Offset becomes available, where you can specify a delay which gets added to the output.

Sample Offset

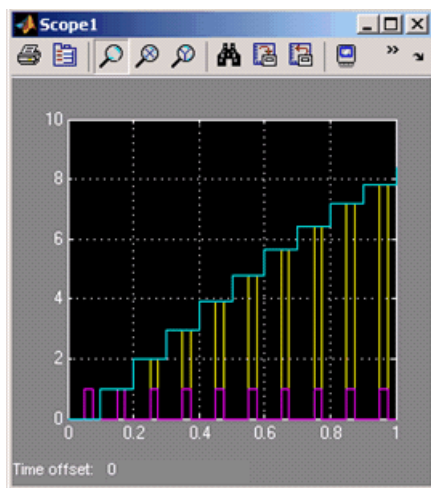
Specifies an offset for the sample rate. The delay you specify here gets added to the output. For a description of sample rates and multi-rate design, see [Multi-Rate Design, on page 717](#). This option is available when Hold input sample is disabled.

Sync Mode

Specifies the synchronization mode for the output. When the clock counter reaches the position you specify in this options, the synchronized output produces 1. You can choose one of the following positions:

Mode	Description
No Sync	There is no synchronized output.
When input changes	The sync output is synchronized with the input and produces 1 when the input changes.
Aligned with offset	The sync output is synchronized with the offset.
Right before offset	The sync output is synchronized with one sample before the offset

The following figure show the synchronization output of an Upsample block with an upsample rate of 4 and a sample offset of 2.



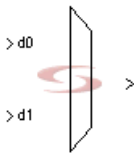
SMC Vector Concat

Constructs vectors by bundling up to 2048 inputs together.

Library

Synphony Model Compiler [Signal Operations](#)

Description



The Synphony Model Compiler Vector Concat block provides a concise way of drawing a Simulink model that performs the same operations on many scalar data streams in parallel. It constructs vectors by multiplexing up to 2048 inputs. The Vector Concat block multiplexes its inputs to a single data type, which is represented and interpreted as vectors by the Simulink tool. (See [Signal Dimensions, on page 561](#) for an explanation of vectors and matrices.) You can adjust the precision of the data type. You can feed the output to other blocks, as described in [Block Connections, on page 561](#).

Signal Dimensions

Different blocks accept or output signals of varying dimensions. A one-dimensional (1-D) signal consists of a stream of one-dimensional arrays output at a frequency of one array (vector) per simulation time step. A two-dimensional (2-D) signal consists of a stream of two-dimensional arrays emitted at a frequency of one 2-D array (matrix) per block sample time. Generally, the 1-D signals are called **vectors** and the 2-D signals are called **matrices**. Currently, the Synphony Model Compiler tool does not support matrix signals.

Block Connections

You can use the vectored data output to feed other blocks, like Abs, Accumulator, Add, Comparator, Constant, Counter, Downsample, FFT, FIR, Gain, IIR, Mult, Mux, Port In, Port Out, RAM, ROM, Shift Register, Shifter, SinCos, and Upsample. The Gain,

Constant, FIR, IIR and ROM blocks can have vectorized coefficients represented as rows of the coefficients. For Gain and Constant, vectorized parameters are column vectors, with each row element corresponding to one channel. For FIR and IIR blocks, each row of the coefficient matrices is for the corresponding channel of the vector data.

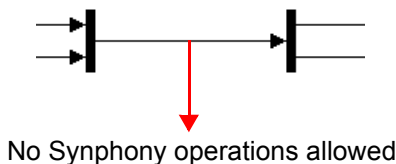
If you connect the input of a block that does not support vector signals, such as IIR, to a vector signal, you get a Simulink error like this one:

```
Error in port widths or dimensions. Output port of  
sinewave_irr.mdl/PortIn is a one-dimensional vector with 4  
elements.
```

Any Symphony Model Compiler block that accepts vector signals as input and output must perform its operation on all elements of its vector inputs at each simulation time step. The Symphony Model Compiler Verilog or VHDL hardware description written out for such a block operates on all elements of the vector inputs in parallel, thus duplicating the block behavior in Simulink. This is similar in effect to the Symphony multi-channelization feature, but at a block level. Symphony multi-channelization works globally on the entire Simulink model to produce a multi-channel hardware implementation. By using the Symphony Model Compiler Vector Concat block, part of the model can be single channel (scalar signals), and other parts of the model can have multi-channel values (different dimensions of vector signals).

Using Simulink Mux-Demux Pairs for Generating Vectors

You can use Simulink Mux and Demux blocks in Symphony Model Compiler designs to reduce link clutter from the model, group signals into one line, carry signals to another location and expand them for operations, and aid in visualization. However, you cannot use the native Simulink Mux and Demux blocks for vector operations. When you require vector operations, use the Symphony Model Compiler Vector Concat and Split blocks to create and split vectors.

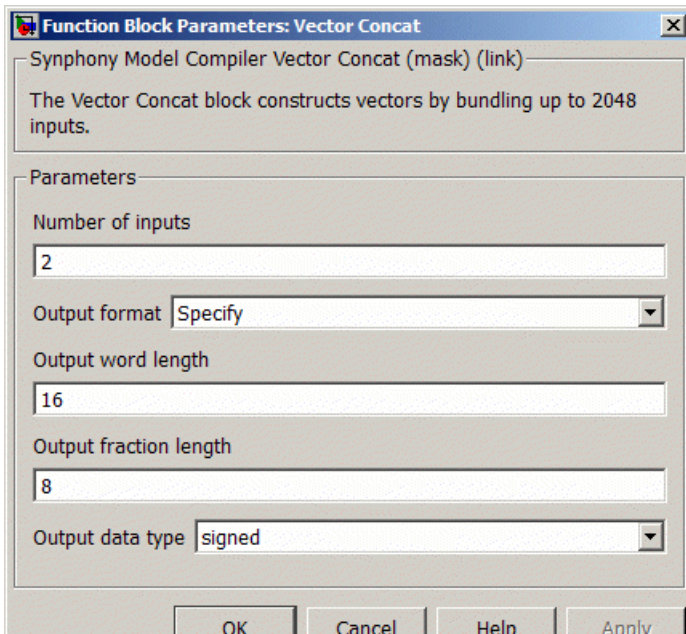


Furthermore, Symphony Model Compiler only supports Simulink Mux-Demux blocks in mux-demux pairs. Having Demux-Mux pairs might create problems in signal routing and decomposition.

Latency

This block has no latency.

Vector Concat Parameters



Number of inputs

Sets the number of inputs that are to be multiplexed to vectors. You can specify up to 2048 inputs.

Output format, Output word length, Output fraction length, and Output Data type

For descriptions of these parameters, see the following:

Output format	Output Format, on page 583
Output word length	Output Word Length, on page 584
Output fraction length	Output Fraction Length, on page 584
Output data type	Output Data Type, on page 584

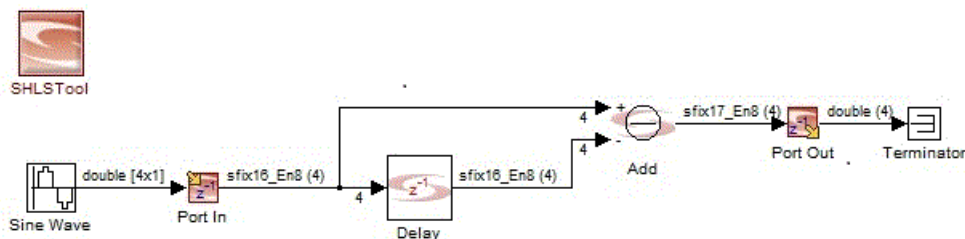
Vector Signal Examples

In this example, four signal streams are input simultaneously and the design requires that you calculate the difference between the current sample and the previous sample for each of the four input streams. Note that while this example is implemented using the Vector Concat block, it could also have been implemented as a single-channel implementation with scalar signals and converted to a 4 channel implementation with the Multichannelization option.

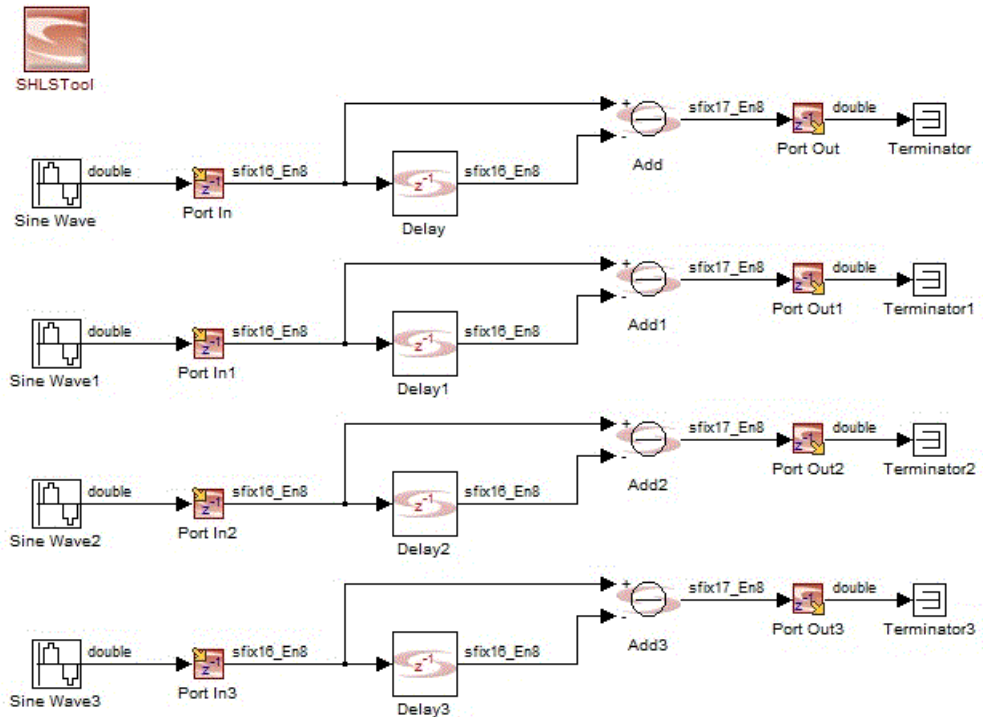
The example shows the vector signals highlighted with bold lines and also displays the vector dimensions.

- To turn on this highlighting, select Format->Port/Signal Displays, and enable the Wide Nonscalar Lines option.
- To display vector dimensions, select Format->Port/Signal Displays, and enable Signal Dimensions.

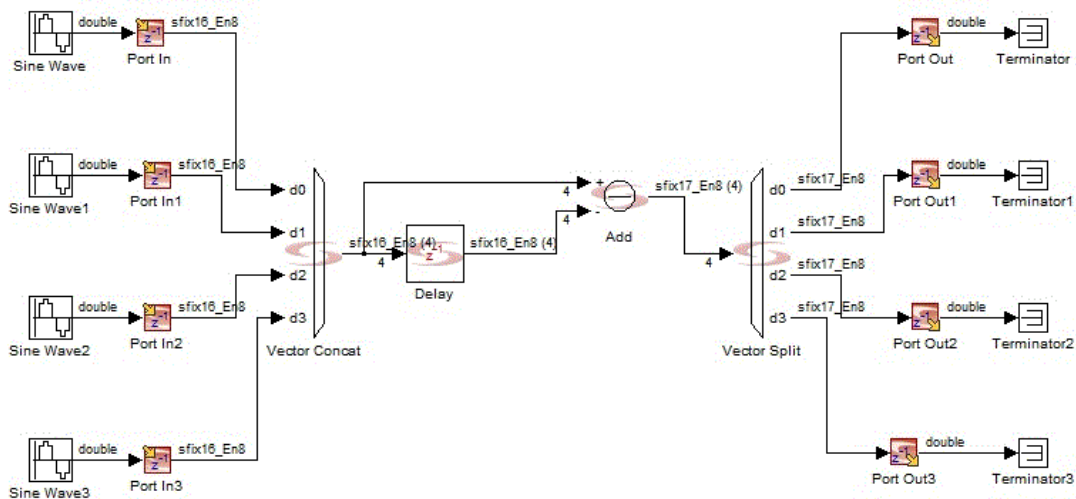
The Sine Wave signal source has been configured to provide a vector signal output. The Port In block inherits the dimensions of this vector signal as do the other Symphony blocks in the model.



The vector signal model above is essentially equivalent to the scalar signal model shown below, where the sources for the Sine Wave block are configured to output scalar signals. It is obvious that the vector signal model illustrated above is much more concise than the scalar signal model.



The following simple example illustrates the use of the Vector Concat and Vector Split blocks. The scalar outputs of the four Sine Wave sources are vectorized so that the delay and subtract operations can be concisely described with a couple of blocks. To illustrate the use of the Vector Split block, the vector signal is decomposed into 4 scalar signals and driven to 4 individual scalar output ports.



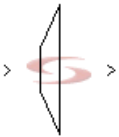
SMC Vector Expand

Converts scalar or vector input to vector output.

Library

Synphony Model Compiler [Signal Operations](#)

Description

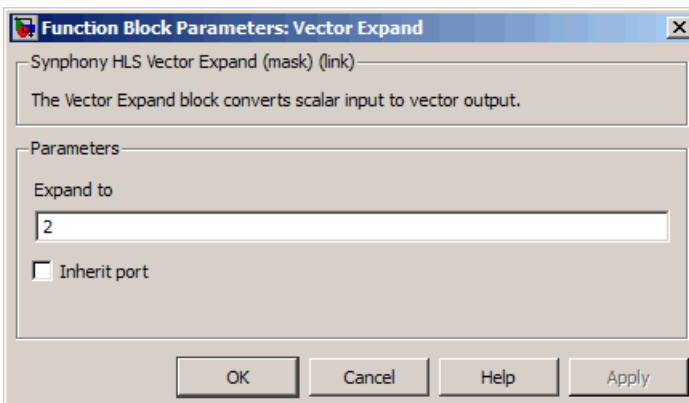


The Synphony Model Compiler Vector Expand block takes the scalar or vector input to the block and converts it to vector output. If the input is scalar, the tool repeats the scalar input to obtain the output vector of the specified size. If the input is vector, the tool cyclically repeats the input vector to obtain the specified size for the output vector.

Latency

This block has no latency.

Vector Expand Parameters



Expand to

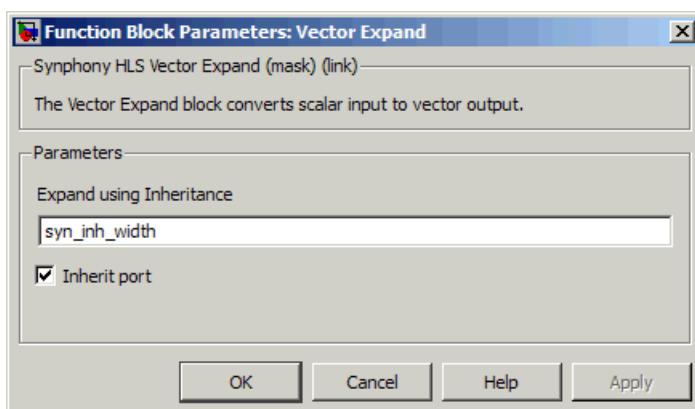
Specifies the output vector size for the block. The output vector size must be an integral multiple of the input vector size.

Inherit port

Determines whether the tool creates an inherit port. The tool creates an inherit port when you enable the option. This port does not convey data. Use the variable `syn_inh_width` to specify the output port dimension. See [Special Variables, on page 588](#) for a description of this variable.

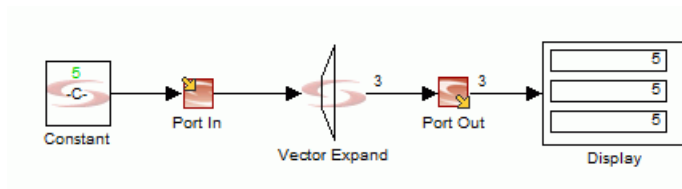
Expand using Inheritance

Lets you specify the output vector dimension. This option only becomes available when you specify Inherit Port. The default value for this option is `syn_inh_width`. You can use it in any regular expression to specify the output vector dimension, as shown in the following figure:

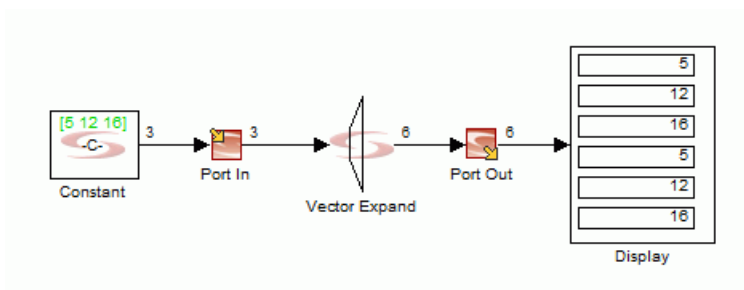


Vector Expand Examples

This example shows scalar input expanded to vector output of size 3:



The next example shows vector input of 3 expanded to vector output of size 6:



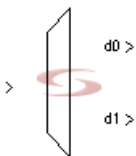
SMC Vector Extract

Extracts selected ports from input vectors, and outputs up to a maximum of 2048 output ports.

Library

Synphony Model Compiler [Signal Operations](#)

Description

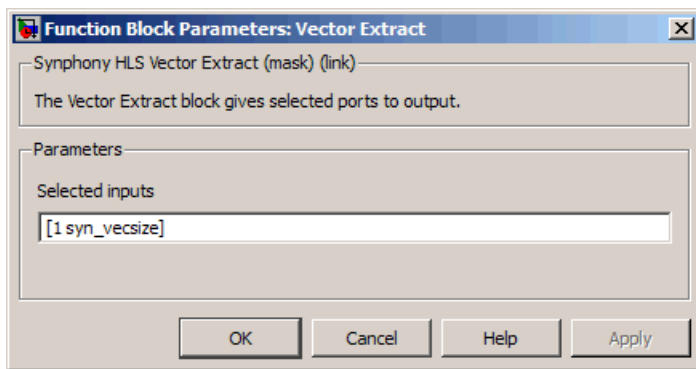


The Synphony Model Compiler Vector Extract extracts the selected inputs and outputs, up to 2048 output ports. See [SMC Vector Concat, on page 561](#) for a more detailed description. and [Vector Signal Examples, on page 564](#) for an example.

Latency

This block has no latency.

Vector Extract Parameters



Selected inputs

Specifies the inputs for extraction. It determines which inputs and the order in which they are routed to the output. You can specify up to 2048 output ports. Port numbering starts from 1.

The default setting outputs the first and last elements of the input vector. `syn_vecsize` is the input length. The number of output ports does not depend on the size of the input vector, but depends on the length of the value in this field. You can not extract negative selected input indices.

Examples

<code>[1 syn_vecsize]</code>	Outputs first and last elements of the input vector.
<code>[syn_vecsize : -1 : syn_vecsize -3]</code>	Outputs the last 4 elements of the vector input in descending order.

SMC Vector Split

Forms signals from vector inputs.

Library

Symphony Model Compiler [Signal Operations](#)

Description



The Symphony Model Compiler Vector Split block is a vector de-multiplexer of up to 2048 outputs. If the number of outputs is less than the vector size, the vectors are split equally. For example, with an input vector size of 8, the outputs are split as shown in the following table.

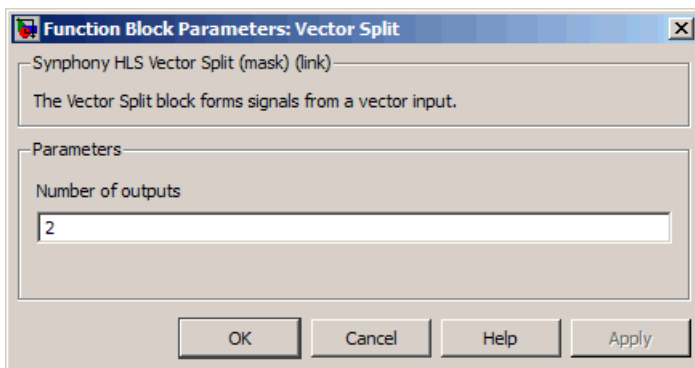
No. of Outputs	Vector Size of Outputs
8	1,1,1,1,1,1,1,1
4	2,2,2,2
3	3,3,2

See[SMC Vector Concat, on page 561](#) for a more detailed description and [Vector Signal Examples, on page 564](#) for an example of its use.

Latency

This block has no latency.

Vector Split Parameters



Number of outputs

Determines the number of outputs required. You can specify up to 2048 outputs.

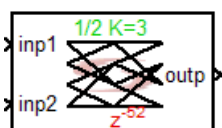
SMC Viterbi Decoder

Decodes convolutionally encoded input data.

Library

Symphony Model Compiler [Communications](#)

Description



The Symphony Model Compiler Viterbi Decoder decodes convolutionally encoded input data and outputs single bit decoded data.¹ It implements a fully parallel ACS (add-compare-select) operation, suitable for high speed applications. This block determines the decoded output using RAM-based traceback, and allows you to monitor the bit error rate (BER) and state metric normalization rate. To decode punctured encoded data, you can feed external erasure signals into the decoder.

The Viterbi Decoder uses a retimed version of the classic ACS unit for speed optimization with an area cost. It uses a modular normalization technique for normalization of state metrics. For further details, see the explanation of state metrics ([State Metric Word Length, on page 575](#)) and block parameter options ([Viterbi Decoder Parameters, on page 577](#)).

This block supports code rates from 1/2 up to 1/7. It allows a maximum constraint length of 8.

Currently, the Viterbi Decoder block can significantly increase DSP synthesis runtime when larger constraints are specified. As constraint lengths increase, synthesis times begin to last considerably longer for folded and retimed designs. For example, with a constraint length of 7, synthesis could take about an hour.

1. G. David Forney Jr, "The Viterbi Algorithm", Proceeding IEEE, vol61, pp 268 - 278, March 1973.

Viterbi Decoder Decoding

The following describe aspects of decoding in more detail:

Puncturing

The Viterbi Decoder block replaces punctured input data with the average of least confident 0 and 1 metrics. This results in better BER performance compared to data replacement with alternating least confident 0 and 1 metrics. All internal branch metrics are represented in offset binary format, whether or not the input data soft data type is offset binary.

State Metric Word Length

State metric word length is chosen to represent twice the dynamic range of the possible cumulated state metric differences so that the tool can perform modulo arithmetic or normalization on state metrics. The automatically computed state metric word length is $\text{ceil}(\log_2((M+1).B))+1$, where M is the number of states and B is the maximum branch metric value. This computation is valid for non-negative branch metrics, like the Viterbi Decoder block.¹

Viterbi Decoder Traceback Algorithm

The traceback method uses a RAM based k-pointer even algorithm (with $k=3$).² If you specify an odd value, the algorithm adjusts the traceback depth to an even value. In this method, there are six RAM banks of length $\text{ceil}(\text{traceback depth}/2)$ which store state transition decisions previously computed by add-compare-select modules for each state.

The traceback operation starts from state 0 and retrieves a survivor path for $2 \times \text{ceil}(\text{traceback depth}/2)$ decisions. The algorithm then assumes the path has converged into a proper state to start decoding for $\text{ceil}(\text{traceback depth}/2)$ decisions. In other words, the algorithm uses the survivor path of $2 \times \text{ceil}(\text{traceback depth}/2)$ decisions starting from fixed state 0 to decode $\text{ceil}(\text{traceback depth}/2)$ of bits. A new traceback starts every at $\text{ceil}(\text{traceback depth}/2)$ decisions on a new RAM bank.

1. C. Shung, G. Ungerboeck, P. Siegel, and H. Thapar, "VLSI architectures for metric normalization in the Viterbi algorithm," in Proc. 1990 Int. Conf Commun. (Atlanta, CA), Apr. 1990, pp. 1723-1728.

2. Gennady Feygin and P.G. Gulak, 'Architectural Tradeoffs for Survivor Sequence Memory Management in Viterbi Decoders', IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. 41, NO 3. MARCH 19

There are some current limitations to the traceback algorithm:

- Traceback architecture requires a LIFO buffer for bit order reversal.
- The RAM-based traceback algorithm is fundamentally different from the register exchange algorithm. Therefore, the BER performance may not be the same. The register exchange method generates the survivor sequence for every output bit usually starting from the state with the best state metric, while the RAM-based traceback algorithm uses the same survivor sequence for $\text{ceil}(\text{traceback depth}/2)$ decisions, as described above.
- The traceback algorithm has a higher latency than a register exchange algorithm.
- With fixed state decoding (the traceback survivor sequence generation always starts from state 0), you might get degraded BER performance for some generator polynomial and constraint length values. See [Trellis Termination, on page 576](#) for a technique to overcome this limitation.
- The tool does not support a traceback depth of 1. For proper trellis survivor sequence generation, $\text{ceil}(\text{traceback depth}/2)$ should be greater or equal to $\text{ceil}(\text{constraint length}/3)$.
- The tool does not support generator polynomials as a feedback type. The traceback architecture assumes feedforward state transitions and generates decoded output bits according to these assumed state transitions.

Trellis Termination

The Symphony Model Compiler traceback operation always starts from fixed state 0 to retrieve survivor sequence. This may lead to a loss in BER performance when compared to cases that use best state trellis initialization, or where the survivor sequence has not converged to a proper state after traceback depth of decisions have been processed.

One technique to overcome this performance issue is to start and end trellis with known state values. For each frame or block of data, reset the convolutional encoder before the first input. This ensures that the trellis starts from state 0. Similarly, you can append N bits to source frame or block data to force trellis termination at state 0. (N is constraint length – 1, i.e. the number of registers in the convolutional encoder). These N number of zero bits are usually called tail bits.

Icon Annotations

The icon for this block displays the following information:

Note	Indicates the code rate and constraint length. For example, $1/2 K = 3$ indicates a $1/2$ code rate, with a constraint length of $K=3$.
Latency Annotation	The latency of the block is $6 \times \text{ceil}(\text{Traceback depth} / 2) + 4$

Viterbi Decoder Parameters

Function Block Parameters: Viterbi Decoder

Symphony Model Compiler Viterbi Decoder (mask) (link)

The Viterbi decoder block decodes convolutionally encoded input data.

Parameters

Constraint length

Generator polynomial (octal)

Traceback depth

Decision type

Input data format

Number of soft bits

☐ External depuncturing

☒ BER port

Number of samples for BER calculation

BER word length

☐ BER ready port

☐ Normalization port

☐ Reset port

☐ Enable port

OK Cancel Help Apply

Constraint length

Is the length of the register used in convolutional encoder plus 1. You must use the same value that was defined for the Convolutional Encoder block. The number of states used in decoding is $2^{(\text{constraint length}-1)}$.

Generator polynomial

Specifies the code (an octal value) to be used for convolutionally encoding the input data. You must use the same value that was defined for the Convolutional Encoder block. The length of the generator polynomial defines the number of inputs to the Viterbi decoder.

Traceback depth

Specifies the length of the Viterbi trellis used in the traceback operation to determine the optimal path for a decoded output bit. Traceback depth is usually as follows:

Non-punctured data	5 times the length
Punctured data	10 times the constraint length

The traceback method uses a RAM based k-pointer even algorithm (with $k=3$). If you specify an odd value, the algorithm adjusts the traceback depth to an even value. See [Viterbi Decoder Traceback Algorithm, on page 575](#) for details.

To avoid loss of decoding performance with respect to best state decisions, use a larger traceback depth.

Decision type

You have two choices:

- Hard decision
The input data is represented by a single bit (0 or 1). The input bit length is 1.
- Soft decision
The input data is represented by more than one bit. When you select this option, two other options become available: Number of soft bits and Input data format, where you can specify the number of input data bits, and the data input format respectively. The extra bits in representation allow the decoder to use a confidence measure on the demodulation operation during channel transmission.

Soft decision coding improves the coding gain with respect to hard decision coding. See [BER Example, on page 582](#) for the effects of using soft decision data.

Input data format

Sets the input data format when the Decision type is set to Soft. You can choose one of the following formats for the input data:

- Sign-magnitude
- Offset binary
- 2's complement signed integer

The Viterbi decoder assumes antipodal demodulation operation where 0 is transmitted as a positive voltage and 1 is transmitted as a negative voltage. The following table shows the confidence measures for different input data formats when the Soft decision type is specified with 3 bits:

Confidence Measure	Offset Binary	Sign Magnitude	2's Complement Signed Integer
Most confident 1	111	111	100
	110	110	101
	101	101	110
Least confident 1	100	100	111
Least confident 0	011	000	000
	010	001	001
	001	010	010
Most confident 0	000	011	011

Number of soft bits

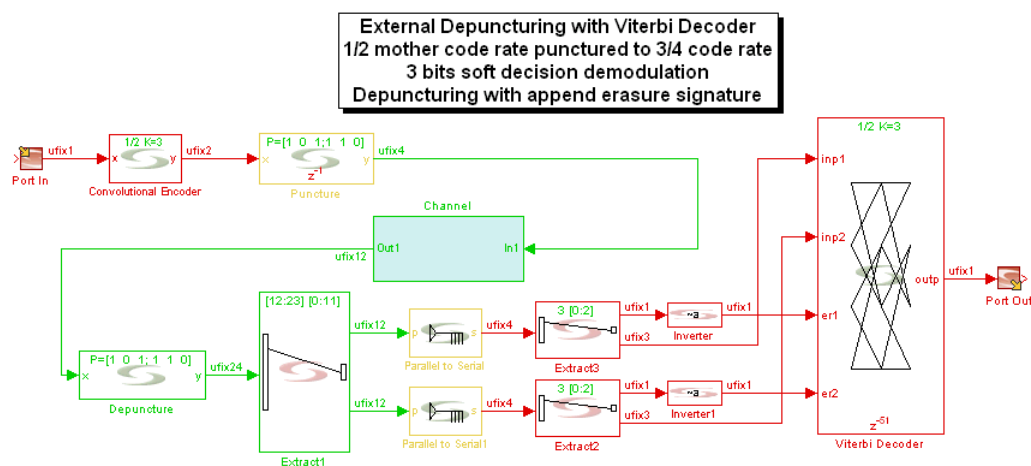
Specifies the bit length of inputs for Soft decision type. For Hard, the bit length of inputs is 1.

External depuncturing

When selected, the Viterbi decoder assumes a punctured convolutional encoder output and applies external depuncturing to the input stream before decoding.

To define whether incoming input data will be erased (punctured) or not, the block uses erasure ports. Each input has a corresponding erasure port with the same port number. An erasure signal of 1 means that the

input data will be erased. Alternatively, the block uses least confident measure levels for 0 and 1 for erasure. The Symphony Model Compiler Depuncture block lets you append erasure signature to output data and work on soft decision inputs.



See [BER Example, on page 582](#) for an example of the effect of puncturing.

BER port

When you select this option, the decoded output is re-encoded using the same decoding parameters (constraint length and generator polynomial) and compared to the delayed input data. Selecting this option makes the Number of samples for BER calculation, BER word length, and BER ready port options available.

The BER output register is reset to 0 after a specified number of samples for comparison are done. You specify the number of samples in Number of samples for BER calculation. For Soft decision inputs, the MSB of the input is used for comparisons (antipodal demodulator operation is assumed). The BER output word length is specified in BER word length. The BER output register wraps on overflow.

The BER output can be used to monitor the error rate on the transmission channel. Together with Normalization Port, the BER port option can be used to detect and correct synchronization errors in the Viterbi decoder.

Number of samples for BER calculation

Specifies the number of comparisons between decoded output and input for BER monitoring. This is only available when you select BER port.

BER word length

Specifies the length of the output BER register. This option is only available when you select BER port.

BER ready port

When selected, it creates a BER ready port, which outputs a ready pulse when the comparisons between input and output data (specified in Number of samples for BER calculation) are done. This option is only available when you select BER port.

Normalization port

When enabled, it creates a normalization port for the block. Together with the BER port, you can use this port to detect and correct Viterbi decoder synchronization errors.

The Viterbi decoder implementation uses minimum cost for state metric updates. When there are errors during channel transmission, state metrics tend to grow slower and there may be overflows in state metric updates. The normalization port gives an instant rough measure for the error rate on the channel by outputting the number of normalizations (overflows) that occurred when state metrics were updated. Low normalization rates can indicate a loss of synchronization. You can then correct the input data order to synchronize the Viterbi decoder with the input stream.

You can compare the normalization rates and BER output to pre-computed threshold values (depending on channel and decoder parameters) to detect synchronization losses.

Reset port

When enabled, it creates a local reset port for Viterbi decoder. When the Viterbi decoder is reset, state metrics, traceback trellis stage and BER register are all reset to their default 0 values.

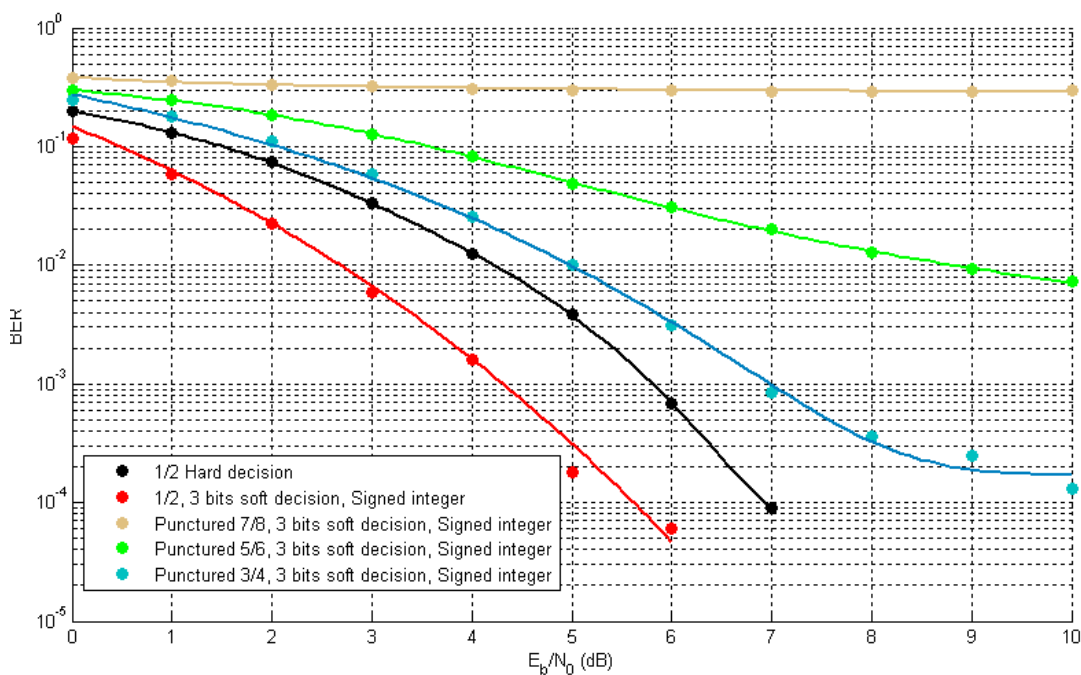
Enable port

When enabled, it creates an enable port for Viterbi decoder. When the Viterbi decoder is not enabled, state metrics, traceback trellis stage and BER register are maintained at their last state.

BER Example

The following BER performance plots show the effect of using soft decision and punctured encoding. The following settings were used:

Constraint length	3
Generator polynomial	[7 5] (1/2 mother code rate)
Traceback depth	15
Puncture matrices, 3/4 code rate	[1 0 1; 1 1 0]
Puncture matrices, 5/6 code rate	[1 0 1 0 1; 1 1 0 1 0]
Puncture matrices, 7/8 code rate	[1 1 1 1 0 1 0; 1 0 0 0 1 0 1]



Common Parameters

This section describes parameters for defining output data type and handling overflow and underflow, that are common to many of the Symphony Model Compiler blocks. The following parameters are defined here:

- [Output Format Options, on page 583](#)
- [Overflow Saturation Options, on page 585](#)
- [Underflow Rounding Options, on page 585](#)
- [Special Variables, on page 588](#)

Output Format Options

The following options for specifying the output are described here:

- [Output Format, on page 583](#)
- [Output Word Length, on page 584](#)
- [Output Fraction Length, on page 584](#)
- [Output Data Type, on page 584](#)

Output Format

Determines the word size and data type of the output. You can select one of the following settings for the output format:

- Automatic calculates the output based on the input. The block uses at least the same size and type on the output as that driven on the input, and guarantees no overflow.
- Full Precision uses the smallest bit width that guarantees no overflow, together with full precision fraction length without internal truncation. For blocks with separate data path specifications like Transform and the filter blocks, this option directly reflects the specified data path format in the output.
- Specify lets you specify the size and data by making the Output Word Length, Output Fraction Length, and Output Data Type parameters available. For certain blocks, it also lets you specify saturation and rounding options.

Output Word Length

Determines the word length of the output in bits. It only becomes available when you set Output Format to Specify. This parameter is used together with Output Fraction Length. Given a word length WL, and a fraction length FL:

- The word bits go from WL-1 to 0
- The fraction bits go from FL-1 to 0
- Bit position WL-1 corresponds to the MSB.
- Bit position 0 corresponds to the LSB.

Output Fraction Length

Sets the fraction length of the output in bits. It only becomes available when you set Output Format to Specify. It is used along with Output Word Length, as described above.

Output Data Type

Determines the data type for the output, and is only available when you set Output Format to Specify.

- **signed** specifies Two's complement signed representation, and sets the sign bit to the MSB. This format specifies that an n-bit binary number be interpreted as a value in the range $[-2^{(n-1)}, (2^{(n-1)})-1]$. Numbers with their most significant bit equal to 1 indicate a negative value, which is obtained by subtracting 2^n from the unsigned value of the number. For example, if a is a signed 3-bit binary number, a=110 means $6 - 2^3 = -2$.
- **unsigned** specifies that an n-bit binary number be interpreted as a value in the range $[0, (2^n)-1]$. If a is an unsigned 3-bit binary number, a=110 means $1*2^2 + 1*2^1 + 0*2^0 = 6$.

Overflow Saturation Options

Determines how the overflow is treated for the block. If the option is enabled, the output is saturated. When a number exceeds the data-range limit for that data type, it saturates to the largest number in the data-range limit. If you disable this option, the tool wraps the overflow.

If the calculated output value is 128 with 8-bit signed integer format (-128, 127), you could get the following results for the block output:

Option Enabled (Saturated)	Option Disabled (Wrapped)
127	-128

Underflow Rounding Options

Determines how underflow is treated for the block. For different blocks, the following options are available. Note that Nearest, Convergent, and Round are very similar; the only difference is in their treatment of cases where there are two valid values for underflow rounding.

- **Floor (Truncate)**
Rounds the underflow down to the first valid quantized value. This operation is equivalent to truncation for both signed and unsigned values, and there is no hardware cost.
- **Nearest**
Rounds the underflow to the nearest valid quantized value. If there are two valid quantized values, it rounds to the larger value. For example, 2.5 with no fractional part can be rounded to 2 or 3, and selecting Nearest rounds it to 3. This operation is equivalent to adding half of quantization step and then doing a Floor on the result.
- **Convergent**
Rounds the underflow to the nearest valid quantized value. If there are two valid quantized values, it rounds to the even value. For example, 2.5 with no fractional part can be rounded to 2 or 3, and selecting Convergent rounds it to 2.
- **Fix**
Rounds the underflow towards zero. For positive values, this is the same as Floor; for negative values, it is the same as Ceil.

- **Ceil**
Rounds the underflow up to the first valid quantized value.
- **Round**
Rounds the underflow to the nearest valid quantized value. If there are two valid quantized values, it rounds up for positive values, and rounds down for negative values. For example, 2.5 with no fractional part can be rounded to 2 or 3, and selecting Round rounds it to 3. This operation functions just like the MATLAB Round operation.

Examples of Rounding with no Fraction Length

The following table shows how the results of applying the different rounding options, when there is no fraction length specified.

Value	Floor	Nearest	Convergent	Fix	Ceil	Round
-1.75	-2	-2	-2	-1	-1	-2
-1.50	-2	-1	-2	-1	-1	-2
-1.25	-2	-1	-1	-1	-1	-1
-1.00	-1	-1	-1	-1	-1	-1
-0.75	-1	-1	-1	0	0	-1
-0.50	-1	0	0	0	0	-1
-0.25	-1	0	0	0	0	0
0.00	0	0	0	0	0	0
0.25	0	0	0	0	1	0
0.50	0	1	0	0	1	1
0.75	0	1	1	0	1	1
1.00	1	1	1	1	1	1

Value	Floor	Nearest	Convergent	Fix	Ceil	Round
1.25	1	1	1	1	2	1
1.50	1	2	2	1	2	2
1.75	1	2	2	1	2	2

Bitwise Rounding Examples

The following table shows examples of bitwise rounding from sfix7.5 to sfix4.2 and sfix9.4 to sfix6.1.

Bitwise Value	Floor	Nearest	Convergent	Fix	Ceil	Round
01.10010	01.10	01.10	01.10	01.10	01.11	01.10
10.01110	10.01	10.10	10.10	10.10	10.10	10.10
01011.1100	01011.1	01100.0	01100.0	01011.1	01100.0	01100.0
11010.0100	11010.0	11010.1	11010.0	11010.1	11010.1	11010.0

Decimal Rounding Examples

The following table shows examples of decimal rounding from sfix7.5 to sfix4.2 and sfix9.4 to sfix6.1.

Decimal Value	Floor	Nearest	Convergent	Fix	Ceil	Round
1.5625	1.5	1.5	1.5	1.5	1.75	1.5
-1.5625	-1.75	-1.5	-1.5	-1.5	-1.5	-1.5
11.75	11.5	12	12	11.5	12	12
-11.75	-12	-11.5	-12	-11.5	-11.5	-12

Special Variables

You can use the variables described below to specify values for Data path word length, Data path fraction length, Coefficient fraction length, Output word length, Output fraction length, Output vector dimension, Number of shift bits, and Inherit Port when applicable. You can apply any mathematical operation on these variables. For example, if you specify an Output word length of $2 \times \text{syn_inp_wl}$, the software creates an output word length that is twice the input word length.

Variable	Description
<code>syn_coef_dt = 1 0</code>	Holds the data type of the coefficients: 1 - signed input 0 - unsigned input
<code>syn_coef_fl</code>	Holds the coefficient fraction length.
<code>syn_coef_wl</code>	Holds the coefficient word length.
<code>syn_guard_bit</code>	Holds the internally calculated bit growth value (for no overflow) for the selected data path, and output data formats of the associated filtering block.
<code>syn_inh_dt = 1 0</code>	Holds the data type of the inherit port: 1 - signed input 0 - unsigned input
<code>syn_inh_fl</code>	Holds the fraction length of the inherit port.
<code>syn_inh_width</code>	Holds the vector dimension of the inherit port.
<code>syn_inh_wl</code>	Holds the word length of the inherit port.
<code>syn_inp_dt = 1 0</code>	Holds the data type of the input data: 1 - signed input 0 - unsigned input
<code>syn_inp_fl</code>	Holds the input fraction length
<code>syn_inp_wl</code>	Holds the input word length.
<code>syn_mat_columns</code>	Holds the number of matrix columns of the input port.
<code>syn_mat_rows</code>	Holds the number of matrix rows of the input port.

CHAPTER 4

SMC Functions

This chapter describes the Symphony Model Compiler functions in alphabetical order.

shls_bitrev	shls_convert	shlsdemo
shlsdoc	shlslib	shlsroot
shlstool	shlsver	syn_get_coefs
syn_get_datatype	syn_get_dspstartup	syn_get_wordlength
syn_read_hex	syn_set_atm	syn_set_dspstartup
syn_set_portcapture	syn_set_portregister	syn_unlink
syn_write_wave		

shls_bitrev

M Control function that reverses the order of bits in an unsigned integer.

Syntax

shls_bitrev (<x>, <w>)

<x> is an unsigned integer for which you want to reverse the order of bits. You can specify a variable for <x>.

<w> specifies the width in bits of the data <x> to be bit-reversed. This argument must be a positive, non-zero integer value. The maximum allowed value for this argument is 52 bits. If you use a variable for <w>, it must evaluate to a constant at compile time for synthesis.

You must specify the bit width (<w>) for simulation because <x> will be represented by a double in the M code (even if quantize has been applied) and the simulation version of shls_bitrev needs to know what bit-width to use when reversing the bits.

Description

The shls_bitrev function is used by the M Control block. It reverses the order of bits in an unsigned integer, using the specified word width. Before performing bit-reversal, the function casts the specified integer to a large, unsigned integer. It discards any fractional portion of the integer that is to be bit-reversed. The bit-reversed result is also unsigned.

Errors and Warnings

The following cases result in simulation warnings and errors:

- If <w> is not a non-zero, positive integer value. Note that <w> can be of type double for simulation, but the value it contains must have a zero fractional portion (e.g., “3.0” is valid).
- If the value of <w> exceeds 52 bits, because the Symphony software uses the maximum width limit of 52 bits instead of the actual value of <w>.
- If the <x> value contains a non-zero fractional part. This fractional part is discarded and the integer portion is bit-reversed.

The following cases result in M-Control synthesis warnings and errors:

- If `<w>` is not a non-zero, positive integer-valued constant. Note that a constant of 3.0 does not cause an error because the fractional portion is zero.
- If the propagated fixed-point type for `<x>` contains any fractional part. This fractional part is ignored and only the integer portion to the left of the binary point is bit-reversed.
- If the fixed-point type for `<x>` is signed. The software converts `<x>` to unsigned and bit-reverses it, producing an unsigned type of width `<w>` or 52 bits, whichever is smaller.
- If the type-propagated width for the integer part of `<x>` (to the left of the binary point) does not match the value of `<w>` argument. The tool uses the value of `<w>` or 52 bits, whichever is smaller, in the bit-reversal. This is true regardless of whether `<w>` is larger or smaller than the type-propagated width. If `<w>` is smaller than the propagated width, some bits will be lost. If `<w>` is larger than the propagated width, there will be zero-valued low-order bits in the bit-reversed result.

shls_convert

Applies the specified quantization rules to the input data.

Syntax

```
shls_convert (<input_data> [, 'format', <output_widths>] [, '<sign_mode>']  
[, '<overflow_mode>'] [, '<round_mode>'])
```

Description

Use this function to directly replace the MATLAB quantize function. Unlike quantize for which you must define a quantizer object, shls_convert takes its quantization parameters in a single line.

Syntax Description

You can specify the parameters in any order, as long as input_data is the first argument to the function, and the format keyword is followed by the output word length and fraction length row vector.

input_data is the data to be cast and it must be the first argument to the function. You can set the input data to any of the following:

Scalar	shls_convert (1.4, 'format', [7 6], 'ufixed', 'saturate')
Vector	shls_convert ([4.4 9.2], 'Format', [33 32], 'wrap', 'nearest')
Matrix	shls_convert ([1.8 2.5 3.3; 8.48 7.56 3.299; 5.5 6.6 7.7], 'format', [4 2], 'ceil')
Variable	shls_convert (var_a, 'Format', [12 4], 'fixed', 'saturate', 'convergent')
Complex	shls_convert (2.1+2.9i, 'Format', [15 3], 'ufixed', 'saturate', 'fix') shls_convert ([5i 2.4; 2.2 4.3i], 'format', [16 15], 'fixed', 'wrap', 'ceil') shls_convert ([1+2i; 2+3i; 4+5i], 'format', [8 7], 'saturate', 'convergent')

format/Format is the keyword for specifying the output data format. It must be followed by the output data format specification. The first letter of this keyword can be either lowercase or uppercase.

output_widths is a row vector with its elements being the output word length and fraction length respectively. If not specified, the default for these parameters is [16 8]. You can set it to either a scalar or a variable:

Scalar `shls_convert (1.4, 'format', [16 15], 'fixed', 'wrap')`

Variable `shls_convert (3.299, 'Format', [out_wl out_fl], 'saturate', 'fix')`

sign_mode defines the signedness of the output data. It can be either **ufixed** or **fixed**. If you do not specify this parameter, the function uses **fixed** by default.

overflow_mode determines how overflow at the output is treated. This value can be **overflow** or **wrap**. If this parameter is not specified, **wrap** is the default.

round_mode determines how underflow at the output is treated. The default is **nearest**, but you can set it to any of the following values:

round	Rounds toward nearest quantized value (symmetric)
floor	Truncate
nearest	Rounds toward nearest quantized value (asymmetric)
convergent	Rounds toward nearest even value
fix	Rounds toward zero
ceil	Rounds up

You can specify the arguments for the **shls_convert** function in any order. The following are all valid, and define the same quantization operation:

```
shls_convert (127.4863, 'format', [16 8], 'fixed', 'wrap', 'nearest')
shls_convert (127.4863, 'Format', [16 8], 'wrap', 'nearest', 'fixed')
shls_convert (127.4863, 'wrap', 'format', [16 8], 'nearest')
shls_convert (127.4863, 'nearest', 'wrap', 'format', [16 8])
shls_convert (127.4863, 'fixed', 'wrap', 'nearest')
shls_convert (127.4863, 'nearest')
shls_convert (127.4863)
```

shlsdemo

Runs the Symphony demos.

Syntax

shlsdemo [('<demo>')]

The <demo> argument specifies a particular demo to run. The argument is optional and, when included, must be enclosed in single quotes. Any of the following keywords can be entered for <demo>:

dctexample	2-D Discrete Cosine Transform design
dynFFT	FFT example design with reconfigurable transform size
gsm_ddc	Digital Down Converter
im_histogram	Histogram computation with the Symphony Model Compiler M Control and RAM blocks
medianfiltering	Median filtering, using an image distorted by noise
noisecanceller_lms	Noise cancellation with the LMS algorithm
noisecanceller_signdatalms	Noise cancellation with the Signed Data LMS algorithm
qam16	QAM 16 modem with a Viterbi decoder black box
qam16withviterbi	QAM 16 modem with the Symphony Model Compiler Viterbi Decoder block
qam16withviterbiwithpunc	QAM 16 modem with the Symphony Model Compiler Viterbi Decoder using 7/8 puncturing
resettableserialtoparallel	Resettable serial to parallel example
rsdecexample	Reed Solomon decoder design
rsencexample	QAM 16 modem with Reed Solomon encoder
smartblackbox	Smart black box example
sobelfiltering	Sobel filtering example with vector operations
stateflow	State flow design with M Control
tut_fir	FIR filter

Description

The `shlstdemo` function locates and executes all preparations for the demos that are bundled with the Symphony Simulink interface. A demo opens a model window, and manages the MathWorks Help Browser to provide extra information.

If you specify the function without any arguments, the command opens the Help browser to the main demo window, from where you can specify a demo. Alternatively, specify the demo at the command line, as described above.

Examples

```
shlstdemo
```

```
shlstdemo ('gam16')
```

shlsdoc

Shows the documentation for blocks.

Syntax

shlsdoc('<blockName>')

The <blockName> argument must be the exact name of the block and must be enclosed in single quotes. For example:

```
shlsdoc('Gain')
```

Description

The shlsdoc function opens the Help browser with detailed information on the specified block. You must specify the exact name of the block.

See Also

[shlslib](#), [shlsver](#)

shllib

Manages the Symphony Model Compiler blockset library.

Syntax

shllib [[['<action>'],<version>]]

The <action> argument specifies what to do with the blockset. The argument is optional and, when included, must be enclosed in single quotes. Any of the keywords specified below can be entered for <action>. If you do not specify an argument, the function defaults to **open**.

info	Returns information about the blockset, without loading or opening the blockset. The information reported is [model, version, path].
load	Loads the blockset in memory and returns the model, version, and path of the blockset.
open	Opens the blockset in a design window and returns the model, version and path of the blockset. If you do not specify <action>, the function defaults to open .

The <version> argument specifies which version of the blockset to manage. The argument is optional and, when included, must be enclosed in single quotes. If you do not explicitly specify a version, the function uses the latest version.

Older versions are available for converting legacy designs, but these libraries are not functional.



Description

The shlslib function opens the Synphony Model Compiler blockset. Use <version> if you want to open a particular blockset. The version number for the current blockset is 8.

Examples

```
model=shlslib('info');  
[model,version]=shlslib('open');  
[model,version,path]=shlslib('open',1);
```

See Also

[shlsdemo](#), [shlsdoc](#), [shlsver](#)

To access other information about the Synphony Model Compiler product, use the following MathWorks commands:

To view the table of contents	help SynphonyHLS or help ('SynphonyHLS')
To view the Release Notes	info SynphonyHLS or info ('SynphonyHLS')
To view the online documentation	doc SynphonyHLS or doc ('SynphonyHLS')

shlsroot

Returns the location where Synphony was installed.

Syntax

shlsroot

Description

The shlsroot function stores the location of the Synphony Model Compiler installation directory when you are setting up the Synphony MATLAB interface. It is used as a reference point to find files in the Synphony Model Compiler tree.

See Also

[shlsver](#)

shlstool

Manages the Symphony synthesis application.

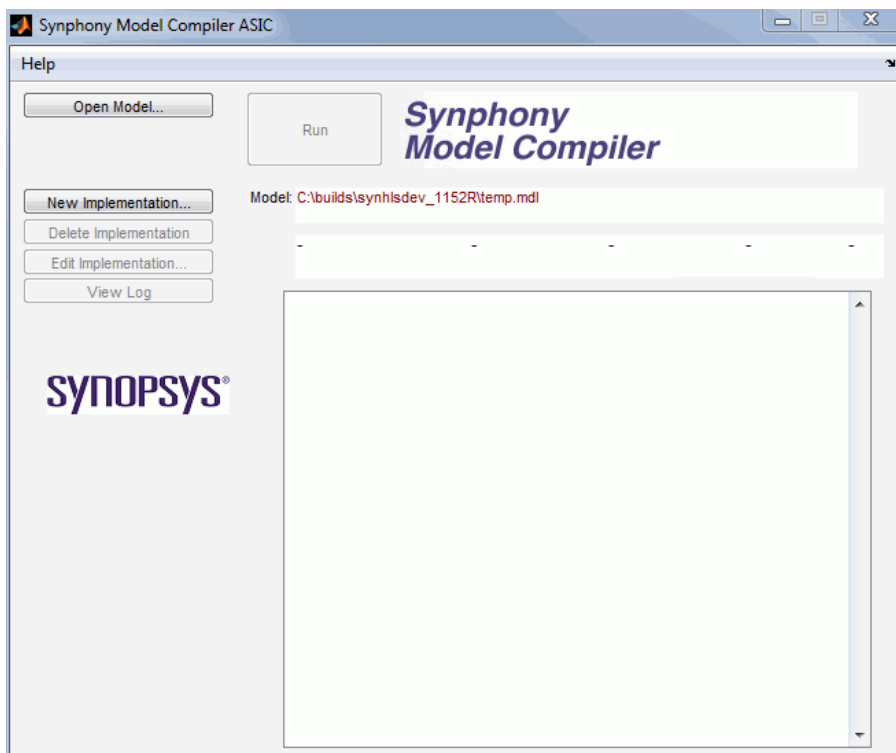
Syntax

```
shlstool [('Model', '<modelName>')]
```

The optional Model argument specifies a design to open in the application and must be enclosed in single quotes.

Description

The shlstool function opens the Symphony application. It brings up a toolbox interface where you can set optimization options and generate RTL code.



Examples

```
shlstool;  
shlstool('Model','design');  
shlstool('Model','design.mdl');  
shlstool('Model','C:\Temp\design.mdl');
```

shlsver

Displays version information for the current MATLAB, Simulink and Symphony installations.

Syntax

shlsver [*('<mode>')*]

The *<mode>* argument determines the amount of information displayed. The argument is optional and, when included, must be enclosed in single quotes. Any of the following keywords can be entered for *<mode>*.

all	Displays information about the software version and the blockset executable builds. It also displays the currently selected simulation and synthesis licenses.
silent	Suppresses the display of results. Use this argument to assign information to output variables.

Description

The **shlsver** function displays the standard version information for the Symphony Model Compiler, MATLAB, and Simulink software.

The function returns up to three outputs:

- The software versions
- The version of the Symphony toolbox executable
- The versions of the Symphony blockset executables

Examples

```
[v,l,b]=shlsver('silent')
```

```
[v,l,b]=shlsver
```

```
[v,l,b]=shlsver('all')
```

The output of this function looks like this:

```
v =  
Name: 'Symphony'  
Version: 'D-2009.12'  
Release: ' (PRODUCTION) '  
Date: '04-Nov-2009'  
l =  
Expiration: '31-dec-2009'  
Id: 'NE577658084841671'  
Vendor: 'Microsemi'  
Features: 'batch,cout,msynth,csim'  
b =  
Shlslib: [67x1 struct]  
Shlstool: 'symphony_hls.exe Nov  4 2009 08:51:25'
```

See Also

[shlsroot](#), [shlslib](#)

syn_get_coefs

Gets the filter coefficients from a Synphony FDATool instance.

Syntax

```
syn_get_coefs [('<instance>', '<type>')] [, <indexList>]]
```

The <instance> argument is the name of the instance in a currently selected system (model) and corresponds to the name of the FDATool instance where the filter is specified. The argument is optional and, when used, must be enclosed in single quotes. The default instance is FDATool.

The <type> argument defines the type of coefficients that are returned. The argument is optional and, when used, must be enclosed in single quotes. Either of the following keywords can be entered for <type>:

forward	Selects the coefficients of the nominator of the filter transfer function (the default).
feedback	Selects the coefficients of the denominator of the filter transfer function. For FIR functions, this is an empty array.

For an FIR filter, the default type forward gets all the relevant coefficients for the filter. For an IIR filter, the default type forward returns the forward coefficients for the filter corresponding to the numerator of the transfer function. The feedback coefficients must be requested explicitly with the feedback keyword.

The <indexList> is a row of integers that picks a specific coefficient. If this is not specified, all coefficients are returned.

The order of the <type> and <indexList> options is not relevant, so you have some flexibility.

Description

The `syn_get_coefs` function returns coefficient information from an FDATool instance. It can return all the forward or feedback coefficients, or one specific coefficient.

Examples

The following examples return the corresponding coefficients from the numerator of the filter (FIR or IIR) defined by the FDATool instance:

```
syn_get_coefs
syn_get_coefs('FDATool', 2)
syn_get_coefs('FDATool', [1 3 5])
syn_get_coefs('FDATool', 'forward', 2)
syn_get_coefs('FDATool', 2, 'forward')
syn_get_coefs('FDATool', 1:2:length(syn_get_coefs('FDATool')))
```

syn_get_datatype

Converts a Simulink data type into Synphony information.

Syntax

syn_get_datatype('<datatype_string>')

The <datatype_string> argument is a string, enclosed in single quotes, that represents a legal Simulink data type, including floating point overwrite.

Description

Often in custom libraries or other applications, you need to get compiled data type information from Simulink and convert it into Synphony information like word length, fraction length, or data type.

Examples

```
syn_get_datatype('uint13')
[wl fl dt]=syn_get_datatype('sfix13_En3')
```

syn_get_dspstartup

Checks the Simulink configuration of a system.

Syntax

```
syn_get_dspstartup(['<system>'])
```

The <system> argument is optional and, when used, must be enclosed in single quotes. If you do not specify <system>, the function uses the top of the current system.

Description

This function takes the system and analyzes the Simulink configuration settings for it. If the system does not have the optimal configuration of settings for DSP simulation, the function returns a warning. The function returns the following values to reflect the status of the configuration settings:

Status	Description
0	Recommended configuration
1	Configuration problem
2	Error in reading system configuration

The default settings for new designs can be set with the Simulink dspstartup command, which sets the default settings to those that are optimal for DSP designs. It is recommended that you put this command in your MathWorks startup file. If you check settings with the syn_get_dspstartup function and find the settings are not optimal, you can enforce the settings with syn_set_dspstartup (see [syn_set_dspstartup](#), on page 614 for details).

The following table lists the optimal configuration settings. If you specify the FixedStepDiscrete or VariableStepDiscrete solver, the function honors it. If you specify any other solver, the function resets it to FixedStepDiscrete. The FixedStepDiscrete setting is recommended for designs that use Simulink source blocks with continuous sample times, and the VariableStepDiscrete setting is recommended for multirate designs.

Setting	Value
Solver	FixedStepDiscrete VariableStepDiscrete
SolverMode	SingleTasking
FixedStep	Auto
SaveTime	Off
SaveOutput	Off
AlgebraicLoopMsgError	
InvariantConstants	On
SignalLogging	Off

Examples

```
syn_get_dspstartup  
syn_get_dspstartup('topLevel')
```

syn_get_wordlength

Calculates the word length required to represent a given set of values.

Syntax

```
syn_get_wordlength (<value>[, '<option>'])
```

The <value> argument is required. If you do not specify <value>, the function runs based on a value of 0.

The <option> argument is optional and, when used, must be enclosed in single quotes. You can specify multiple options, in any order. The <option> arguments can affect the value. Any of the following values can be entered for <option>:

fl <i><floatingLength></i>	Limits the number of fraction bits used to represent the value, and affects the returned value. If not specified, the function uses 0.
---	--

fixed ufixed	Determines the data type used to represent the value. It affects the word length required. If not specified, the function uses the fixed option by default. If you specify a negative value for a ufixed data type, you get a warning message. The value is adjusted to preserve the stored integer.
------------------------------	--

round floor nearest fixed ceil convergent	Affects the value because it affects the underflow behavior to represent the value. If it is not specified, the function uses the nearest option.
---	---

Description

This function returns the word length required to represent the given value without overflow. Depending on the options, it can return an adjusted value cast to represent underflow, because some options can affect the underflow. When <value> is a vector, the `syn_get_wordlength` function returns the maximum word length required to represent all elements of <value> for the given options.

Examples

```
syn_get_wordlength(1)
syn_get_wordlength (3.14)
syn_get_wordlength (pi)
[w,v]=syn_get_wordlength (pi,'ufixed','fl',2,'nearest')
[w,v]=syn_get_wordlength ({exp(1) pi},'fl',4)
```

syn_read_hex

Reads a file with hex-encoded ROM data and returns a vector.

Syntax

```
syn_read_hex [(['<filename>'], 'checksum')]
```

Description

The `syn_read_hex` function reads a file and scans it for ROM data encoded in hex format. If you specify a filename, it must be enclosed in single quotes. If you do not specify a filename, the function searches for a file called `rom.hex`. If you specify more than one file, the function only reads the last file specified. The optional checksum argument causes the function to validate the checksum for each hex record and, when used, must be enclosed in single quotes. The `syn_read_hex` function returns the data as a vector. See [Specifying ROM Data with `syn_read_hex`, on page 776](#) for information on using this function.

Hex Format

A hex record consists of the following:

```
:LLAAAATT[DD. . .]CC
```

The six fields in a hex record are described in this table:

Field	Characters	Description
:	1	Start code. An ASCII colon, ":".
LL	2	Byte count. The count of the character pairs in the data field.
AAAA	4	Address. The 2-byte address at which the data field is to be loaded into memory.

Field	Characters	Description
TT	2	Type. This can be 00, 01, 02, or 04. <ul style="list-style-type: none"> • 00 - Data record. A record containing data and the 2-byte address for the data to reside. • 01 - End-of-file record. A termination record for a file of hex records. Only one termination record is allowed per file and it must be the last line of the file. There is no data field. • 02 - Extended segment address record. • 04 - Extended linear address record.
DD...	0-2n	Data. From 0 to n bytes of executable code, or memory-loadable data. n is normally 20 hex (32 decimal) or less.
CC	2	Checksum. The function calculates the checksum of the record by summing the values of hexadecimal digit pairs in the record, module 256, and taking the two's complement.

Example of Hex-Encoded ROM Data

This is a sample of hex-encoded ROM data:

```
:01000000AA55
:01000100AB53
:01000200AC51
:01000300AD4F
:01000400AE4D
:00000001FF
```

You can decompose the first line as follows:

```
||||| CC->Checksum ('h55=>85)
||||| DD->Data ('hAA)
||||| TT->Record Type (00 : Data Record)
|| AAAA->Address ('h0000=>0)
| LL->Record Length ('h01 =>1 byte)
:->Start Code
```

Examples of the Function

```
syn_read_hex
syn_read_hex('checksum')
syn_read_hex('data.hex')
syn_read_hex('data.hex', 'checksum')
```

syn_set_atm

Allows you to configure Symphony timing modes.

Syntax

```
syn_set_atm
```

Description

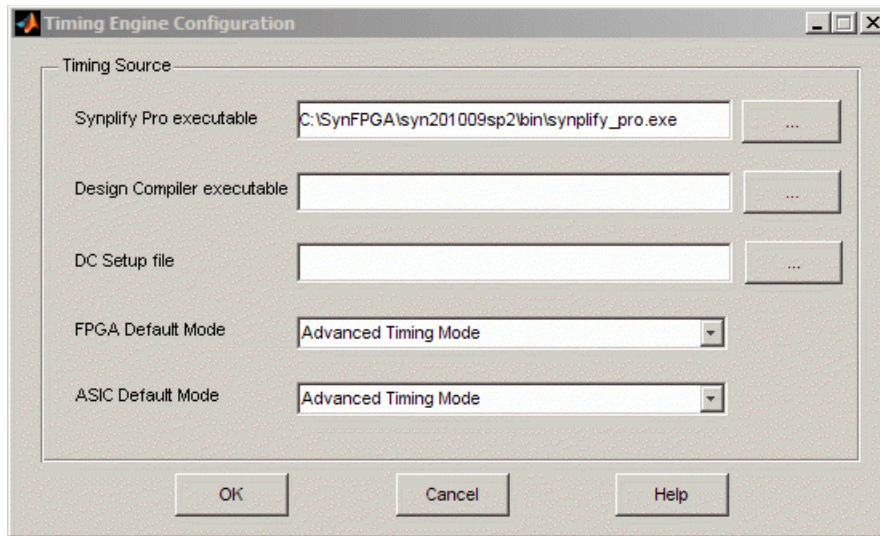
The `syn_set_atm` function opens a dialog box that lets you set the timing mode. See [Timing Engine Configuration Dialog Box, on page 612](#) for details.

Example

```
syn_set_atm
```

Timing Engine Configuration Dialog Box

This dialog box lets you configure the timing mode you want to use as a basis for optimizations. The dialog box opens automatically during installation, but you can open it at any time by typing `syn_set_atm` at the MATLAB command prompt. See [Configuring SMC Timing Modes for FPGAs, on page 638](#) for details about setting timing modes.



Synplify Pro Path	Specifies the path to the Synplify Pro executable. You must specify this path for Advanced Timing Mode, because it uses the Synplify Pro timing engine.
FPGA Default Mode	<p>Sets the default timing mode used to calculate timing parameters for all DSP synthesis runs. You must specify this for FPGA designs.</p> <ul style="list-style-type: none"> • Advanced Timing Mode Uses the Synplify Pro timing engine and target-specific timing data to produce more accurate results. • Estimation mode Uses simpler, latency-based device characterizations as a basis for optimizations. Estimation mode is faster, but the results are less accurate.

syn_set_dspstartup

Sets the Simulink configuration for a system.

Syntax

```
syn_set_dspstartup [('<system>')]
```

The <system> argument is optional and, when included, must be enclosed in single quotes. If you do not specify <system> explicitly, the function uses the top level of the current system.

Description

This function takes the system and forces the Simulink configuration settings to an optimal configuration for DSP simulation. Before using this command, check the optimization settings with `syn_get_dspstartup`. For more information on configuration settings, see [Configuring Settings for Simulink Simulation, on page 638](#).

Typically, you set default settings for new designs with the Simulink `dspstartup` command, which ensures that the settings are the best for DSP designs. It is recommended that you put this command in your MathWorks startup file. To check your current settings, use the `syn_get_dspstartup` function ([syn_get_dspstartup, on page 606](#)).

Examples

```
syn_set_dspstartup  
syn_set_dspstartup('topLevel')
```

syn_set_portcapture

Lets you determine data capture parameters for the input and/or output ports.

Syntax

syn_set_portcapture [(['<port_type>'], '<capture_status>')]

The <port_type> argument specifies the kind of port. The argument is optional and, when included, must be enclosed in single quotes. Any of the following values can be entered for <port_type>:

all	This is default. It applies the capture criteria to all the ports.
in	Applies the capture criteria to all input ports
out	Applies the capture criteria to all output ports.

The <capture_status> argument specifies the capture parameters for the ports. The argument is optional and, when included, must be enclosed in single quotes. Any of the following values can be entered for <capture_status>:

toggle	This is default. It toggles the capture status for the specified ports.
set	Sets all specified ports to capture data.
clear	Clears all specified ports so that they do not capture data.

Examples

This example toggles the capture parameters of all ports:

```
syn_set_portcapture;
```

The following function clears the capture parameters of all input port:

```
syn_set_portcapture('in', 'clear');
```

This function specifies that all ports capture data:

```
syn_set_portcapture('set');
```

syn_set_portregister

Determines the register (extra latency) parameters of the input and/or output ports.

Syntax

syn_set_portregister [[('<port_type>'], '<register_status>')]

The <port_type> argument specifies the kind of port. The argument is optional and, when included, must be enclosed in single quotes. Any of the following values can be entered for <port_type>:

all	This is default. It applies the parameters to all the ports.
in	Applies the parameters to all input ports
out	Applies the parameters to all output ports.

The <register_status> argument specifies the latency parameters for the registers. The argument is optional and, when included, must be enclosed in single quotes. Any of the following values can be entered for <register_status>:

toggle	This is default. It toggles the capture status for the specified ports.
set	Sets all specified ports to capture data.
clear	Clears all specified ports so that they do not capture data.

Examples

This example toggles the register parameters of all ports:

```
syn_set_portregister;
```

The following function clears the register input parameters of all input ports:

```
syn_set_portregister('in', 'clear');
```

This function sets register parameters for all ports:

```
syn_set_portregister('set');
```

syn_unlink

Unlinks any instance of a Symphony Model Compiler custom block.

Syntax

```
syn_unlink [('<system>')]
```

The <system> argument specifies the name of a system or a subsystem. The argument is optional and, when included, must be enclosed in single quotes. If you do not provide an argument, the function uses the selected system.

Description

The `syn_unlink` function goes through the hierarchy, and unlinks any instance that has a Symphony Model Compiler custom block as a reference. The function returns the names of all blocks that have been unlinked.

Examples

```
syn_unlink  
syn_unlink('topLevel')  
syn_unlink('topLevel/Hierarchy')
```

syn_write_wave

Writes a Value Change Dump (VCD) file based on a set of Simulink logged signals.

Syntax

```
syn_write_wave [(['<system>'])]
```

The *<system>* argument specifies the model for which the VCD file is generated. This optional must be enclosed in single quotes or be a variable that specifies a string representing the model name; if it is included. When the argument is not included, the default is `gcs` that gets the current Simulink system.

Description

To use `syn_write_wave`:

1. Enable signal logging for the model. To do this, select Configuration Parameters->Data Import/Export for the model.
2. Log signals in your model. To do this, right-click on a signal and select Signal Properties.
3. Run simulation to generate the signal log.
4. Run `syn_write_wave`.
5. Open the generated VCD file in any waveform viewer of your choice.

For details see, [Viewing Simulink Signals in a Waveform Viewer, on page 846](#).

CHAPTER 5

Constraints

The following sections describe constraints you specify, constraints that the tool infers, the constraints file, and forward-annotation:

- [HLS Constraints File](#), on page 620
- [Synphony Model Compiler Constraints](#), on page 622
- [Multicycle Path Constraints](#), on page 632
- [Forward-Annotation](#), on page 636

HLS Constraints File

The HLS constraints file is a Tcl file that is used to set implementation constraints on blocks and subsystems in the model hierarchy. You can use it to specify constraints for pattern annotation, retiming, or FIR architecture.

If you specify conflicting constraints, the tool uses the lowest-level constraint in terms of hierarchy, or the most specific constraint.

Each line in the Tcl file consists of a `define_attribute` command that specifies a constraint that is applicable to one or more blocks or subsystems. Blank lines and lines beginning with the hash (#) character are ignored.

For information about adding a constraint file to an implementation, see [Using Constraints, on page 653](#).

Constraint Command Syntax

This is the general command syntax for the constraints in the file.

```
define_attribute [-r] [block_name] attribute_name [attribute_parameters]
```

<code>-r</code>	Optional. Applies the constraint to the entire hierarchy inside a subsystem.
-----------------	--

<i>block_name</i>	<p>Required only if the constraint is block-specific; otherwise it applies to the whole model.</p> <p>Block name or path to the block. It is a hierarchical expression that matches the hierarchical names of one or more blocks and subsystems in the model. The expression always begins with <code>top/</code>, where <code>top</code> is the name of the model, and uses the forward slash (/) as the hierarchy separator.</p> <p>For the last character, you can use an asterisk (*) as a wildcard that matches zero or more characters.</p> <p>Currently the tool treats the expressions <code>top/SubSystemA/*</code> and <code>top/SubSystemA</code> as equivalent, but it deprecates the syntax of the second expression. It is recommended that you update the existing files to make the trailing wildcard explicit. See Hierarchical Block Name Examples, on page 621 for examples.</p> <p>An expression that names a single block or subsystem (without wildcards) is same as the expression returned by the MATLAB <code>gcb</code> command for that block.</p>
--------------------------	---

<i>attribute_name</i>	Name of the attribute to be applied to the specified blocks. See pattern_annotation Constraint, on page 626 , shls_retiming_lock Constraint, on page 169 , and Constraints for FIR Architecture, on page 228 for specific information about individual constraints and their syntax.
<i>attribute_parameters</i>	A string containing parameters specific to the attribute being applied. If it contains spaces, the string must be enclosed in quotes.

Hierarchical Block Name Examples

The following table shows some examples of block names:

toplevel/SubsystemA/BlockB	Matches the BlockB block inside the SubsystemA subsystem in the toplevel design.
toplevel/SubsystemA	Matches the SubsystemA subsystem in the toplevel design.
toplevel/SubsystemA/*	Treated as equivalent to the previous example.
-r toplevel/SubsystemA	Matches the hierarchy inside the SubsystemA subsystem.
toplevel/SubsystemA/B*	Matches all blocks and subsystems whose names start with B that are immediately inside the SubsystemA subsystem.

Symphony Model Compiler Constraints

This section describes the Symphony Model Compiler constraints, in alphabetical order. All these constraints can be specified in the HLS constraints file, and follow the general command syntax conventions described in [HLS Constraints File](#), on page 620.

- [add_register_and_balance_parallel_paths](#), on page 622
- [areabased_fir_arch_selection_atm Constraint](#), on page 623
- [fir_architecture Constraint](#), on page 623
- [multi_cycle_path Constraint](#), on page 624
- [pattern_annotation Constraint](#), on page 626
- [retime_across_blackbox](#), on page 627
- [retiming_scale_factor Constraint](#), on page 628
- [shls_retiming_lock Constraint](#), on page 628

add_register_and_balance_parallel_paths

Inserts a register in a specified location in the RTL generated by the tool.

`define_attribute block_name add_register_and_balance_parallel_paths`

<i>block_name</i>	The hierarchical block after which you want to insert a register
-------------------	--

This register balancing constraint works for single-rate and multirate designs. The tool automatically inserts a register after the specified block and balances all parallel paths in the generated RTL.

For this constraint to work, you must select the Retiming option with a non-zero latency. During register balancing, the tool does not move the registers in your design, but uses the additional latencies specified by the Retiming option to balance the registers. The tool ignores the constraint if you select Folding or Multichannelization.

To insert registers in multiple locations, specify one Tcl constraint per location. The tool uses the minimum number of registers possible to satisfy all the register balancing constraints. If required, the tool reuses additional registers placed in parallel paths during the retiming of those paths.

areabased_fir_arch_selection_atm Constraint

Disables area-based selection of FIR architecture for the FIR block.

By default, the tool includes area estimates in the calculation if automatic timing mode is on. You can disable this behavior and disregard area estimates by specifying the following Tcl constraint:

```
define_attribute [-r] block_name areabased_fir_arch_selection_atm "disable"
```

-r block_name Refer to [Constraint Command Syntax, on page 620](#) for information about the syntax for specifying a block name and path with `define_attribute`.

Note that `block_name` can point to the entire model, a subsystem, or a specific instance. When applied to an entire model or subsystem, this constraint only applies to the FIR blocks within that model or subsystem.

disable	Overrides the default and does not take area estimates from the synthesis tools into account when determining the FIR architecture.
---------	---

fir_architecture Constraint

Overrides the FIR architecture decision for the baseline implementation by specifying an architecture for the FIR block.

This constraint is valid only when applied to an FIR block instance. Use the syntax shown below:

```
define_attribute [-r] block_name fir_architecture "fir_architecture_name [-timeout]"
```

<code>-r block_name</code>	Refer to Constraint Command Syntax, on page 620 for information about the syntax for specifying a block name and path with <code>define_attribute</code> .
<code>fir_architecture_name</code>	Must be either <code>direct</code> , <code>transpose</code> , or <code>mcm</code> . The architecture must be enclosed in quotes and is case-insensitive. If Folding is enabled, the tool always uses the <code>transpose</code> architecture, regardless of which option you specify. A corresponding warning message is printed in the log file.
<code>-notimeout</code>	Only applies to the MCM architecture. It forces MCM architecture to be selected even if the size and bit-width of the filter are beyond the usual scope of MCM (up to 200 24-bit coefficients). Note that using this option might result in a longer runtime.

multi_cycle_path Constraint

Lets you specify multicycle paths on Subsystem blocks at the Simulink level.

For an *n*-cycle path, valid output is available once every *n* cycles. Multicycle path constraints relax retiming on the Subsystem block, by setting the target clock period for those paths to *<number of cycles>* x actual clock period. This is the syntax for the command:

define_attribute *<subsystem_name>* **multi_cycle_path** *<cycles>*

<i>subsystem_name</i>	Refer to Constraint Command Syntax, on page 620 for information about the syntax for specifying a block name and path with <code>define_attribute</code> .
<i>cycles</i>	The number of cycles in the multicycle path. The value must be greater than 1.

For details about specifying multicycle path constraints, see [Specifying Multi-cycle Path Constraints, on page 632](#).

When a multicycle path constraint is applied to a subsystem, all combinational paths between enabled registers that have the same enable signal are considered multicycle paths. For subsystems, the constraint applies to the entire hierarchy inside the subsystem. The tool relaxes the constraints on these paths, and reports the applied constraints in the log file:

@N: MCP *<cycles>* is applied to the *<subsystem_name>*.

The tool automatically infers multicycle paths in certain situations and you do not have to explicitly apply a constraint. See [Automatically Inferring Multicycle Path Constraints, on page 633](#) for details.

Multicycle Constraint Limitations

The tool does not apply the constraint in certain circumstances:

- You cannot use the constraint if folding is applied to the top level design.
- The tool ignores a multicycle path inside a multicycle path subsystem if the path contains rate-changing blocks, the RTL Encapsulation or HLS Subsystem blocks, or the Black Box or Smart Black Box blocks.
- The input and output registers of the multicycle path must be enabled by the same enable signal. If they are not, the tool ignores the constraint and issues this warning message:

```
@W: Multi-cycle path constraint on sub-system <sub-system name> is
ignored as enabled registers are not connected to the same enable
signal.
```

Multicycle Constraint Priority

If there is a conflict between multiple constraints, the constraint that is specified first has priority. All ignored constraints are reported in the log file.

The following describe some constraint scenarios:

- If there are multiple constraints on the same subsystem, the tool honors the first one specified.
- If you first specified a multicycle constraint on subsystem A, and then added another on subsystem A/b of A, the tool ignores the constraint on A/b. If you first specified the multicycle constraint on A/b, the tool ignores the constraint on A and uses the constraint specified on A/b.

Forward-Annotation

Multicycle constraints are passed on to the logic synthesis tools. They are forward-annotated to the Synopsys synthesis tools as follows:

Synplify tools	define_scope_collection <collection> {expand -hier -seq -from <enable net name>} define_multicycle_path -from <collection> -to \$<collection> <no of cycles>
DC	set <collection> [get_cells -of_objects [all_fanout -flat -from [get_pins <enable name>] -endpoints_only]] set_multicycle_path -from <collection> -to <collection> <cycles>

pattern_annotation Constraint

Specifies patterns for folding in the constraints file. Alternatively, you can specify pattern annotation constraints in the Tag property on Simulink blocks, as described in [Setting Folding Annotations in the Simulink GUI, on page 668](#).

This is the Tcl syntax to specify pattern annotation in the constraints file. Use the same keywords in the Tag property.

```
define_attribute block_name pattern_annotation "pattern p instance i"
define_attribute [-r] block_name pattern_annotation exclude
```

<i>block_name</i>	See Constraint Command Syntax, on page 620 for information about hierarchical block names and paths.
	<ul style="list-style-type: none"> List each instance on a separate line with a unique instance number. Do not use the -r recursive argument for folding annotations; you can only use it with the exclude keyword. You can only use wildcards in block names if you are using the exclude keyword. If an annotated instance contains another annotated instance, the inner instance is ignored.

pattern <i>p</i> instance <i>i</i>	<p>Overrides the Synphony Model Compiler automatic pattern detection, and marks instances to be folded. Enclose pattern instance annotation in quotes to protect the spaces.</p> <p>The pattern annotation is a case-insensitive string and indicates that a given subsystem is an instance numbered <i>i</i> of a pattern numbered <i>p</i>, where <i>i</i> and <i>p</i> have the following requirements:</p> <ul style="list-style-type: none"> • <i>i</i> and <i>p</i> are non-negative integers. • All instances of one pattern must have the same value for <i>p</i>, which should be different for different patterns. • Every instance for the same pattern must have a unique value for <i>i</i>.
exclude	<p>Marks individual blocks to exclude them from folding. Such blocks are not considered during plain multiplier folding, and are not included in patterns. Note that this annotation is currently ignored if it is applied to a subsystem.</p> <p>If an instance of a user-annotated pattern includes a block that is marked for exclusion, all the instances of that pattern are dropped.</p>

Examples

```
define_attribute toplevel/SubsystemA/BlockB pattern_annotation exclude
```

This excludes BlockB in SubsystemA from folding.

```
define_attribute toplevel/SubsystemA/SubsystemC pattern_annotation "pattern 0 instance 1"
```

This marks SubsystemC as an instance numbered 1 for the pattern numbered 0.

```
define_attribute toplevel/SubsystemA/SubsystemC pattern_annotation "exclude"
```

This exclude constraint has no effect.

retime_across_blackbox

Overrides the default and enables retiming across the specified black box or RTL Encapsulation block.

In general, it is not safe to retime across a black box or RTL Encapsulation block because it may change the functionality of the design. By default, the tool does not do such retiming.

If you specify this constraint, it is up to you to ensure that the design functionality is not changed. Typically, if the block has a source in it, it is not safe to retime across it. A source is any logic whose output does not fully depend on its input values, like a constant, or a counter with an initial value. For the constraint to take effect, the Retiming option must be enabled.

This is the syntax for the constraint:

```
define_attribute <RTL Encapsulation or black box name> retime_across_blackbox
```

retiming_scale_factor Constraint

Accounts for routing delay by specifying a scale factor for how aggressive the Synphony tool should be during retiming.

The tool multiplies the clock frequencies by the scale factor for retiming. If you specify a scale factor of 1.5 in a multirate design with two clocks, one 100 MHz and the other 300 MHz, the SMC tool retimes the design using 150 MHz and 450 MHz respectively as the clock frequencies.

This is the constraint syntax:

```
define_attribute retiming_scale_factor <scale factor> [forward_annotate]
```

<i>scale factor</i>	A fractional number greater than or less than 1. This value is written to the log file for reference.
<i>forward_annotate</i>	Optional. If it is not specified, the SMC tool passes the original clock frequencies to the downstream tools. When specified, the tool also forward-annotates the scale factor to the Synplify synthesis tools. For example: <pre>define_attribute retiming_scale_factor 1.25</pre>

shls_retiming_lock Constraint

Specifies whether retiming affects Delay blocks. This is the syntax:

```
define_attribute [-r] {block_name} shls_retiming_lock {lock_only | none}
```

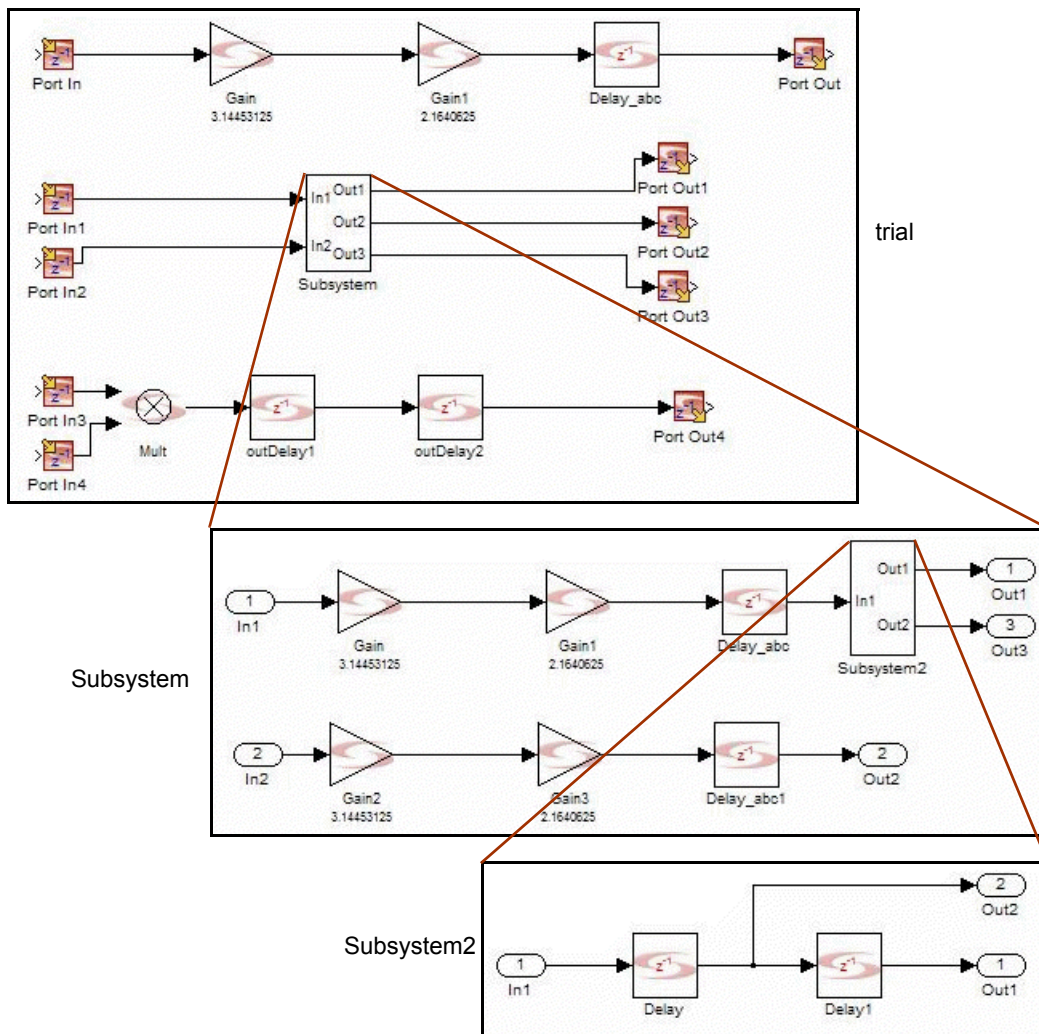
<code>-r block_name</code>	Refer to HLS Constraints File, on page 620 for information about the syntax for specifying a block name and path with <code>define_attribute</code> .
<code>shls_retiming_lock</code>	Must be <code>lock_only</code> or <code>none</code> . <ul style="list-style-type: none">• <code>lock_only</code> prevents the <code>Delay</code> block from being moved during retiming.• <code>none</code> allows the <code>Delay</code> block to be moved during retiming. It also removes a previously applied lock.

The Synphony Model Compiler tool forward-annotates the constraint to the logic synthesis tool, unless folding is enabled. In it is enabled, it prints this warning:

```
@W: Folding is turned on; retiming lock constraints are not
forward-annotated to Synplify.
```

If a retiming constraint is applied to a subsystem, the tool applies the constraint to every delay immediately under that subsystem. In other words, `toplevel/SubsystemA` is considered equivalent to the `toplevel/SubsystemA/*`. The former behavior is deprecated, and applying a retiming constraint to a subsystem will have no effect. Update any existing HLS constraints files that use the previous expression to use the new expression.

The following shows the top level of a simple design called `trial`, and the internals of the `Subsystem` block. `Subsystem2` includes two `Delay` blocks as shown below, so the design contains `Delay` blocks at the `trial` level, the `Subsystem` level, and the `Subsystem2` level.



You can define retiming constraints for the Delay blocks at all three levels, as shown below.

Constraint	Applies to...	shls.log Results
define_attribute {trial/Delay_abc} shls_retiming_lock {lock_only}	trial/Delay_abc	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: Delay_abc : {lock_only}
define_attribute {trial/Subsystem/*} shls_retiming_lock {lock_only}	All Delay blocks under trial/Subsystem	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: Subsystem.Delay_abc1 : {lock_only} Subsystem.Delay_abc : {lock_only}
define_attribute -r {trial/Subsystem/*} shls_retiming_lock {lock_only}	All Delay blocks under trial/Subsystem and lower hierarchies recursively	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: Subsystem.Subsystem2.Delay1 : {lock_only} Subsystem.Subsystem2.Delay : {lock_only} Subsystem.Delay_abc1 : {lock_only} Subsystem.Delay_abc : {lock_only}
define_attribute {trial/outDel*} shls_retiming_lock {lock_only}	All blocks beginning with outDel under trial	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: outDelay2 : {lock_only} outDelay1 : {lock_only}
define_attribute {trial/*} shls_retiming_lock {lock_only}	All Delay blocks under trial	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: outDelay2 : {lock_only} outDelay1 : {lock_only} Delay_abc : {lock_only}

Constraint	Applies to...	shls.log Results
define_attribute {trial/*} shls_retiming_lock {lock_only} define_attribute {trial/outDelay1} shls_retiming_lock {none}	All Delay blocks under trial except outDelay1	@N: The retiming constraints were successfully applied for the following blocks or hierarchies in the following order: outDelay2 : {lock_only} outDelay1 : {lock_only} Delay_abc : {lock_only} outDelay1 : {none}

Multicycle Path Constraints

You can specify multicycle constraints explicitly, but the Symphony Model Compiler tool also automatically infers multicycle path constraints in certain situations. For details, see the following:

- [Specifying Multicycle Path Constraints](#), on page 632
- [Automatically Inferring Multicycle Path Constraints](#), on page 633

Specifying Multicycle Path Constraints

There are two typical cases where you can specify the multicycle constraint on a subsystem:

- **Loops with enabled registers**
If you have a loop that fails to meet timing, retiming it will not add extra latency in the loop. If you already know that the enable signal to the registers in the loop comes every *n* clock cycles, you can isolate the loop in a subsystem, and attach an *n*-cycle constraint to it. The multicycle constraint helps to meet timing by relaxing the timing constraint.
- **Enabled data paths**
If you have a data path where the registers are enabled only once every *n* clock cycles, that path does not have to meet timing according to the clock constraints. If you specify a multicycle constraint on this path, you might be able to meet timing without inserting any extra latency.

The following procedure shows you how to explicitly specify multicycle path constraints for a subsystem.

1. Use the SMC library blocks to create your Simulink model design, and validate it.
2. Specify a multicycle path constraint on the subsystem using a `multi_cycle_path` constraint.

All combinational paths between enabled registers that have the same enable signal are considered multicycle paths. For subsystems, the constraint applies to the entire hierarchy inside the subsystem. For example, to apply a constraint of 16 on subsystem `xyz` in the top level of Simulink model `abc`, specify the following constraint:

```
define_attribute abc/xyz multi_cycle_path 16
```

Details about this constraint are described in [multi_cycle_path Constraint, on page 624](#).

3. Ensure that the Tcl constraint file is specified and enabled in Implementation Options->HLS Constraint Options->Enable Constraint File.
4. Run SMC synthesis to create the implementation of the Simulink model.

Specified multicycle constraints are not applied in some situations, which are described in [Multicycle Constraint Limitations, on page 625](#). If multiple constraints cause a conflict, the constraint that is specified first has priority. All ignored constraints are reported in the log file. See [Multicycle Constraint Priority, on page 625](#) for details.

The tool forward annotates multicycle constraints in the generated synthesis scripts, as described in [Forward-Annotation, on page 636](#).

5. Make sure to source the SMC-generated synthesis scripts when you use the downstream synthesis tools for RTL synthesis.

Automatically Inferring Multicycle Path Constraints

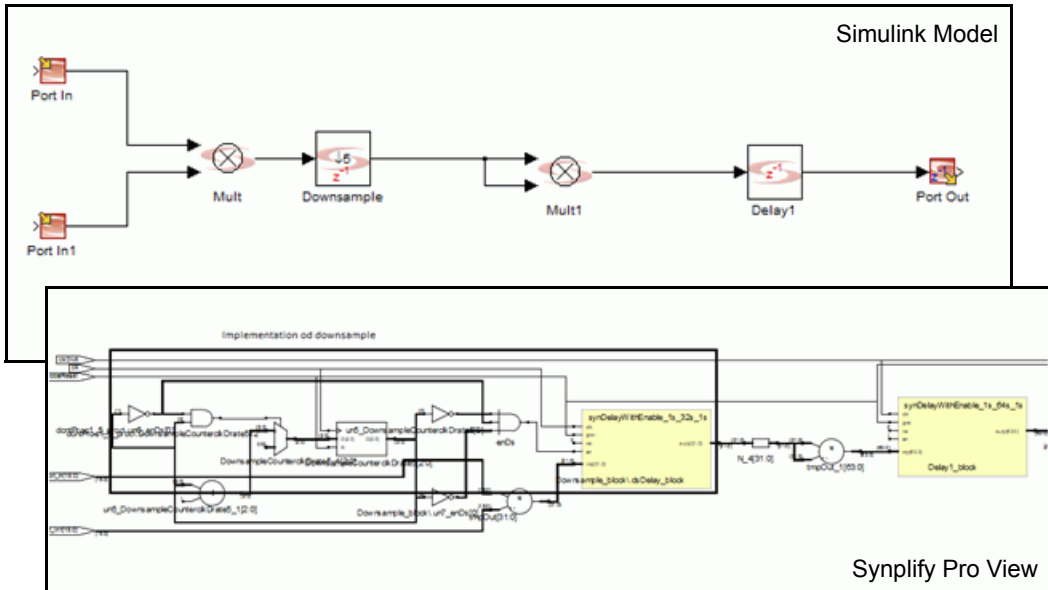
The Symphony Model Compiler tool automatically infers multicycle path constraints in downsampling and upsampling situations, as described below.

Downsampling and Automatic Inference of Multicycle Paths

Paths from a Downsample block output to the registers that immediately follow can be designated multicycle paths. Define a multicycle path with the value set to the downsample rate -1 in either of the following cases:

- When the sample offset is zero and downsample rate is greater than 2.
- When multichannelization is on, but not defined for multi-rate folding.

The following figure shows a model with a Downsample block with a downsample rate of 5 and an offset of 0, and the corresponding Synplify Pro implementation.



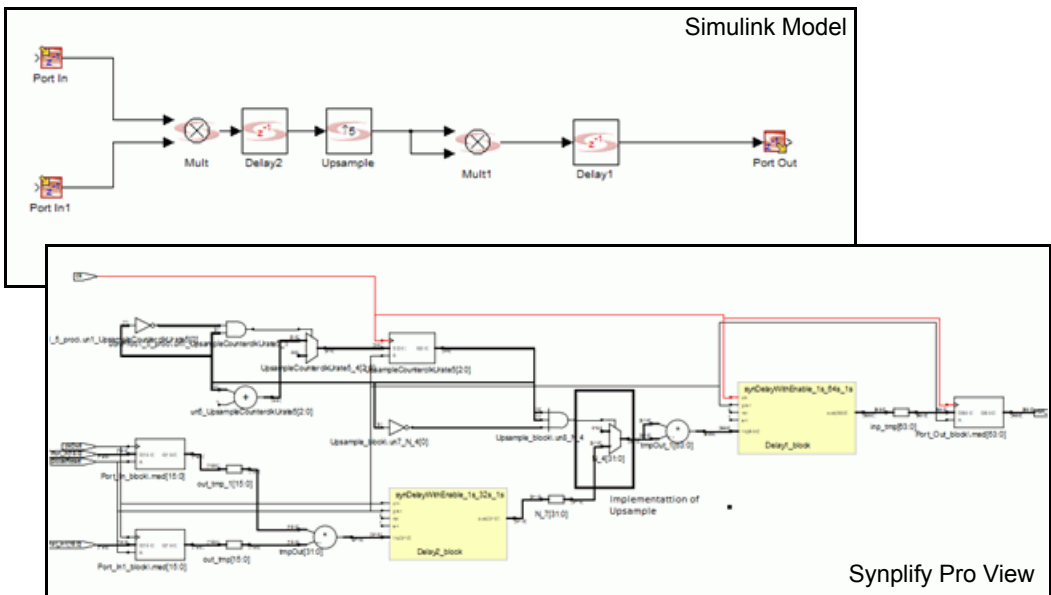
The dsDelay_block inside Downsample is clocked with clk and Delay1_block is clocked with clkDiv5. Delay1_block is updated once every fifth clock and the downsample input data is available at dsDelay_block after the first clock. So, the path between dsDelay_block and Delay2_block can be treated as a multicycle path of 4 of the clock domain clk.

When the offset is greater than 0, the delay of the downsample is 1 (in clkdiv5). A slower clock domain delay is already available inside the downsample, so the path following the downsample is not a multicycle path.

Upsampling and Automatic Inference of Multicycle Paths

With Upsample blocks, define multicycle path constraints when the sample offset is greater than zero, and folding and multichannelization are off. In this case, paths going through the upsampler are considered multicycle paths. Set the multicycle value to sample offset + 1.

The following figure shows a model with an Upsample block and the corresponding Synplify Pro implementation. The upsampler is implemented as a multiplexer. The select line of the mux is `counter == offset`. Delay1_block and the counter are clocked with `clk` and Delay2_block is clocked by `clkDiv5`. The offset is 3. In this case, the Delay2_block data passes the mux in the third clock. This means that the data1_block register is updated with the data from Delay2_block in the fourth clk (3+1). Consequently, you can specify a multicycle path constraint of 4 on the path between Delay2_block and Delay1_block (clock domain `clk`).



Forward-Annotation

The tool forward-annotates inferred and specified constraints to the Synopsys logic synthesis tools as follows:

- Clock frequency is forward-annotated.
- Multicycle path constraints are forward-annotated.

The `define_scope_collection` command in this example creates a collection of sequential elements in the module instance `myI_1` driven by the `enable` signal `en_1` in to a variable called `smc_reg1`. The multicycle constraint specifies a value of 8 on all the paths starting from and ending at this collection.

```
define_scope_collection smc_regs1 {expand -hier -seq -from {myI_1.p:en_1}}  
set_multicycle_path {8} -from {$smc_regs1} -to {$smc_regs1}
```

- Path delays are forward-annotated.
- FPGA synthesis attributes attached to subsystems are forward-annotated. See [Tagging Subsystems with FPGA Synthesis Attributes, on page 796](#) for details.
- The tool converts `shls_retiming_lock` constraints to Synplify `syn_allow_retiming` attributes with a value of 0, and forward-annotates them.
- The Synplify `syn_ramstyle` attribute is forward-annotated if you set it in the SMC Implementation Options dialog box.
- For FPGA targets, the SMC tool sets the Synplify `syn_useioff` attribute, to prevent the packing of registers in I/O pads.

CHAPTER 6

Synthesizing the Design

This chapter contains step-by-step procedures that describe how to set up the Symphony Model Compiler software and run typical design flow tasks:

- [Configuring Symphony Model Compiler, on page 638](#)
- [Basic Procedures, on page 641](#)
- [Setting Options for an Implementation, on page 644](#)
- [Running Synthesis with SHLSTool, on page 677](#)
- [Synthesizing with a Host Interface Block, on page 678](#)

The design flow is described in [Symphony Model Compiler Design Flows, on page 20](#).

Configuring Symphony Model Compiler

The following describe how to configure settings so that you can use the Symphony Model Compiler tool effectively:

- [Configuring Settings for Simulink Simulation, on page 638](#)
- [Configuring SMC Timing Modes for FPGAs, on page 638](#)
- [Setting Default Display Modes, on page 640](#)

Configuring Settings for Simulink Simulation

The Simulink simulator can be optimized for discrete-time fixed-point designs. This improves simulation run time and behavior with the Symphony Model Compiler blockset.

1. Type `syn_set_dspstartup` at the MATLAB command line for the optimal configuration.

This function automatically tunes the settings for the model. It ensures that you have the best Simulink settings for discrete-time DSP design. See [syn_get_dspstartup, on page 606](#) for the syntax and the settings.

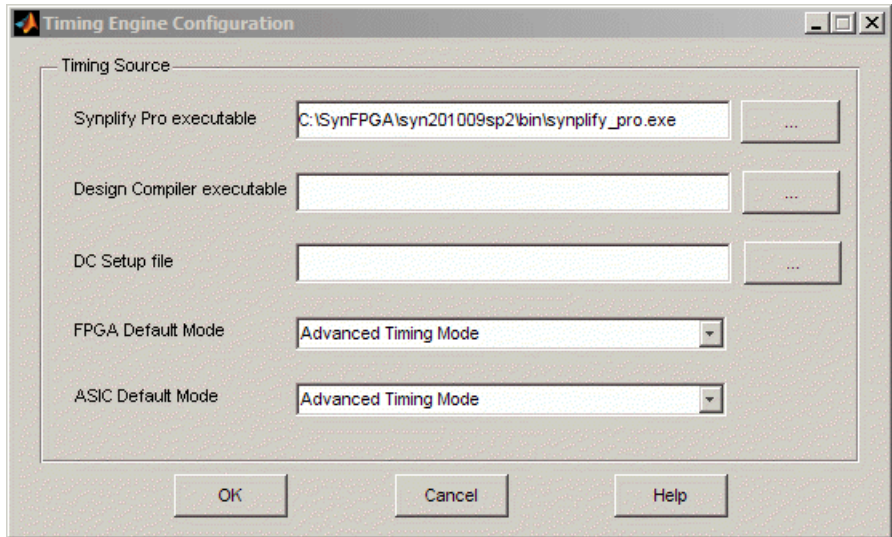
2. To view the current settings, select Simulation->Configuration Parameters from the model window. This opens a dialog box where you can view the current settings.

Configuring SMC Timing Modes for FPGAs

The Symphony Model Compiler tool offers two timing modes for FPGAs: estimation mode and advanced timing mode. The latter uses Advanced Timing Mode, which produces more accurate results by running the Synplify Pro timing engine.

1. Make sure you have the Synplify Pro software installed and accessible.
2. To set the default timing mode for all synthesis runs, do the following:
 - Open the Timing Engine Configuration dialog box. You can either do this when the dialog box opens as part of the installation process, or open

it later, by typing `syn_set_atm` ([syn_set_atm](#), on page 612) at the MATLAB command prompt.



- Set FPGA Default Mode to the mode you want to use, either advanced timing mode or estimation mode. See [Timing Engine Configuration Dialog Box](#), on page 612 for descriptions of this dialog box.
 - If you set the default to Advanced Timing Mode, you must also set Synplify Pro Path to point to the Synplify Pro executable. This is because advanced timing mode uses the Synplify Pro timing engine to estimate timing. The tool defaults to estimation mode if it cannot find Synplify Pro or if problems occur.
 - Click OK.
 - When you run DSP synthesis, make sure to enable Retiming and Advanced Timing Mode in the SHLSTool UI. To use estimation mode, disable Advanced Timing Mode.
3. To override the default and set the timing mode for the current implementation, do the following.
- To specify advanced timing mode for the current implementation, first type `syn_set_atm` at the MATLAB command prompt. In the dialog box, specify the path to the Synplify pro executable, and click OK.
 - Double-click the SHLSTool toolbox.

- To use advanced timing mode, enable Retiming and Advanced Timing Mode in the toolbox UI. To use estimation mode, make sure Advanced Timing Mode is disabled.

4. Run DSP synthesis.

The tool uses the timing mode you specified to run synthesis. The log file reports the estimated frequency of the overall design, as calculated by the ATM engine. If the design does not meet the frequency you requested, the log file lists possible reasons and solutions for fixing this deficiency, like the following:

- Blocks that did not meet timing
- Loops that prevent retiming
- Extra latency required to achieve the target frequency

Setting Default Display Modes

When you work with Symphony Model Compiler designs in Simulink, it is useful to have certain display settings. The following shows you how to configure the recommended settings.

1. In the Simulink model window, enable the following from the Format->Port/Signal Display menu:
 - Sample Time Colors
 - Port Data Types
 - Signal Dimensions

Basic Procedures

Symphony Model Compiler runs under the MATLAB and Simulink interface, and most of what is described here should be familiar to all MATLAB users.

- [Starting a Symphony Model Compiler Design, on page 641](#)
- [Working with Symphony Model Compiler Blocks, on page 642](#)

For an overview of the design flow, see [Symphony Model Compiler Design Flows, on page 20](#).

Starting a Symphony Model Compiler Design

The following describes how to start up Symphony Model Compiler and set up a model window for your design.

1. Start MATLAB and make sure you are in your design directory. Click the Simulink icon and open Simulink.



2. Set up the model window.
 - Open a design or create a new one. For details about the interface, see the Simulink documentation.

For a new design....	Select File->New->Model or click the icon. An empty model window opens.
-----------------------------	---

For an existing design...	Select File->Open and specify the model you want to open. The model window opens with your design.
----------------------------------	--

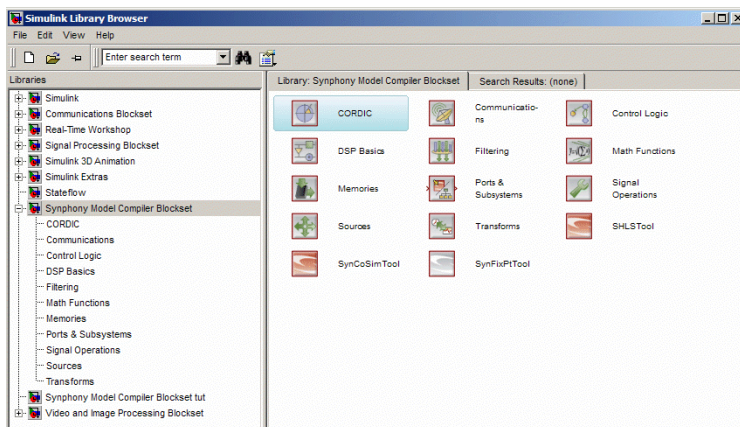
You can now add blocks to your design as described in [Working with Symphony Model Compiler Blocks, on page 642](#).

Working with Symphony Model Compiler Blocks

This basic procedure shows you how to work with Symphony Model Compiler blocks in your design.

1. From the Simulink library browser, double-click Symphony Model Compiler blockset.

The toolbox blocks are at the top level, and the other blocks are organized into libraries.



2. To add a block, do the following:
 - If needed, double-click the library with the block.
 - Select the block from the list in the library.
 - Drag it into your model window.
3. Build your design.
 - Make sure that all design inputs and outputs that you want implemented in RTL are defined with Port In and Port Out blocks from the Symphony blockset. A Symphony Model Compiler design must be bounded by these blocks.
 - Build your circuit using the Symphony blockset. The software only generates RTL for Symphony blocks. You can use Simulink blocks for analysis and stimuli, but they will not be reflected in the generated RTL.

- Connect the blocks as required by your design. A quick way to connect blocks is to select the starting block, press Ctrl, and then select the block or point to which you want to connect.
- Set block parameters by double-clicking a block in the model window and setting the options specific to that block in the dialog box that opens.
- Instantiate the SHLSTool toolbox so that you can run DSP synthesis and generate RTL for the design. For information about using this toolbox, see [Running Synthesis with SHLSTool, on page 677](#).

Setting Options for an Implementation

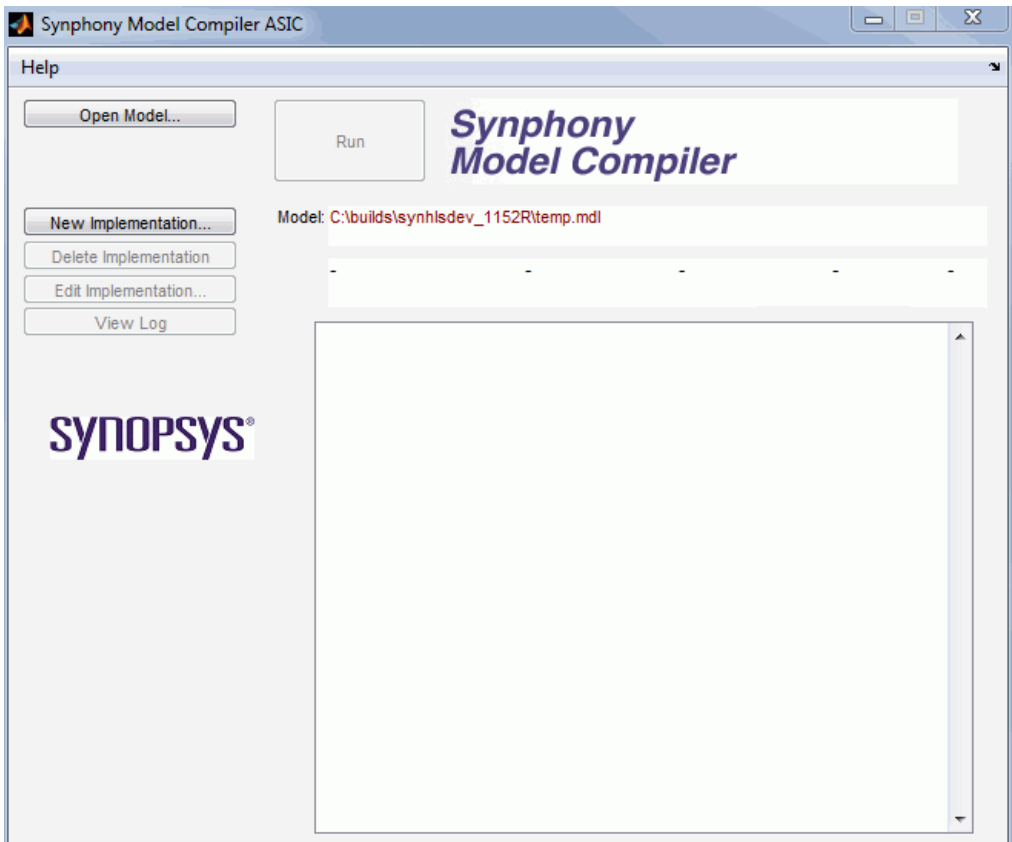
The following describe how to set up an implementation and specify various options for DSP synthesis. For information about running DSP synthesis or optimizing results, see [Running Synthesis with SHLSTool, on page 677](#),

- [Setting up Implementations, on page 644](#)
- [Resolving Read/Write Conflicts in FPGA RAMs, on page 647](#)
- [Including Comments in the Generated RTL, on page 649](#)
- [Keeping Signal Names in Generated RTL, on page 650](#)

Setting up Implementations

Implementations let you run the same design with different optimizations or target technologies so that you can evaluate the results.

1. Include the SHLSTool block in your design.
 - Add the SHLSTool block to your Simulink schematic design. When you add a SHLSTool instance anywhere in the hierarchy, it controls the complete system you have captured, Symphony Model Compiler optimizations and the generation of RTL code in particular.
 - After you have finished your design and verified it, double-click the SHLSTool block in the schematic. The SHLSTool window opens.



2. Set up the implementation.

- In this toolbox window, set Model to the appropriate model file, if necessary. By default, it shows the current file.
- You must create a new implementation if this is the first time you have opened the Symphony Model Compiler window. To create a new implementation, click New Implementation, and specify a name for the implementation in the Implementation Options dialog box. The default name is <design_name>_impl_1.
- To open an existing implementation, double-click the implementation name in the Symphony Model Compiler window, and click Edit Implementation. This opens the Implementation Options dialog box.

3. For an FPGA target, set the following:

- Select the vendor, technology, part, package, and speed you want on the Target Options tab.

The screenshot shows the 'Target Options' tab selected in a software interface. The 'Implementation' field is set to 'shlslibv1_impl_2'. Under the 'Device' section, the 'Vendor' is set to 'Microsemi', 'Technology' is '3200dx', 'Part' is 'A3265DX', and 'Speed' is '-2'. Each of these fields is a dropdown menu.

- If you want your design to use registers with only asynchronous global resets, set Flip Flop Reset Sensitivity to Asynchronous in the Design Options tab.
 - On the HLS Constraint Options tab, specify how you want logic synthesis to handle possible RAM read/write conflicts. See [Resolving Read/Write Conflicts in FPGA RAMs, on page 647](#) for details.
4. Set options for generating output files:
- To generate an RTL file in VHDL format, enable Generate VHDL. The tool supports the VHDL 93 format. See [Considerations for Generating RTL Output Files, on page 647](#).
 - To generate an RTL file in Verilog format, enable Generate Verilog. The tool supports the Verilog 2001 and Verilog 93 formats. See [Considerations for Generating RTL Output Files, on page 647](#).
 - To generate a testbench, follow the procedure described in [Verifying the RTL with a Test Bench, on page 853](#).
5. Click OK.

The Symphony Model Compiler window reflects the technology choices you made.

You can now run the implementation by clicking the Run button in the Symphony Model Compiler window. Alternatively, you can set other optimization and output options before clicking Run. For information about the optimizations, see [Optimizing with Folding, on page 662](#), [Optimizing with Retiming, on page 655](#), and [Optimizing with Multichannelization, on page 674](#). For additional information about using the output files, see [Working with the Output for FPGA Designs, on page 856](#).

Considerations for Generating RTL Output Files

Here are some considerations for generating RTL output files in Verilog or VHDL formats. The procedure for generating the files is described in [Setting up Implementations, on page 644](#).

- **Mixed output (Verilog and VHDL)**
If you generate both Verilog and VHDL output files, the files are written into separate subdirectories under the implementation directory: <implementation>/verilog or <implementation>/vhdl.
- **Verilog 2001**
The Verilog output files use Verilog 2001 statements including generate statements, ANSI C-style port declarations, and wild-carded sensitivity lists. Any simulators you use must be able to handle this format.
- **VHDL 93**
The VHDL output files use the VHDL 93 format, so any simulators you use must be able to handle this format.
- **Keywords**
Make sure that your Simulink design does not use Verilog or VHDL keywords to name ports and instances.

Resolving Read/Write Conflicts in FPGA RAMs

For FPGA targets, the Symphony Model Compiler software automatically extracts RAMs from the Symphony Model Compiler blocks in the design and writes them out to the generated RTL. You can control the way in which the FPGA logic synthesis tool handles read/write conflicts in the RAM by setting the Use ReadWrite conflict logic attribute for RAM option.

1. Open the Implementation Options dialog box. See [Setting up Implementations, on page 644](#) for details.
2. On the Target Options tab, do the following:
 - Set the target to an FPGA device.
 - To have the FPGA logic synthesis tool insert bypass logic around RAM instances to resolve conflicts, make sure the Use ReadWrite conflict logic attribute for RAM option is not checked (the default). When the logic synthesis tool encounters a mismatch where there is a simultaneous read and write to the same RAM location, it inserts bypass logic to handle the conflict.

The screenshot shows the 'Target Options' tab of the Implementation Options dialog. The 'Implementation' field contains the text 'shlsilbv1_impl_2'. Below this, the 'Device' section is expanded, showing four dropdown menus: 'Vendor' set to 'Microsemi', 'Technology' set to '3200dx', 'Part' set to 'A3265DX', and 'Speed' set to '-2'.

- If you do not want the FPGA logic synthesis tool to insert bypass logic to resolve conflicts, enable Use ReadWrite conflict logic attribute for RAM. The Symphony Model Compiler software sets an attribute in the output fdc file:

```
define_global_attribute syn_ramstyle {no_rw_check}
```

This attribute directs the FPGA synthesis tool not to insert bypass logic to handle mismatches. Use this setting with caution, if you need to minimize overhead logic and are sure there are no mismatches.

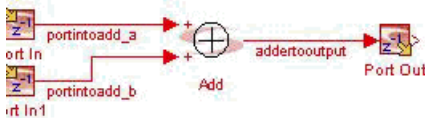
Including Comments in the Generated RTL

You can ensure that the Symphony Model Compiler includes any block comments when it generates the RTL code. Use the following block-tagging procedure to ensure that any comments that are written to the block in the Simulink environment are retained in the generated HDL code.

1. To tag a block, use `/*<BlockName>*/ <Comment to be transferred to HDL>`.

This example has the following added to the Adder block:

`/*Add*/ And the comment goes to the Adder`



`/*Add*/ And the comment goes to the adder`

The resulting RTL includes the comment:

```
Verilog    endgenerate
            /*Add: And the comment goes to the Adder */
            generate
            begin: Add_block
                wire enab;
                wire [16:0] tmpOut;
                wire [16:0] tmp_portintoadd_a_0;
                wire [16:0] tmp_portintoadd_b_1;
            ...
```

```
VHDL      end Block;
            -- Add: And the comment goes to the Adder
            Add_block: Block
                signal enab : std_logic;
                signal tmpOut : std_logic_vector (16 downto 0);
                signal tmpOutPre : std_logic_vector (16 downto 0);
                begin
                    enab <= GlobalEnable1;
            ...
```

2. To tag a block which is inside a subsystem from outside the subsystem, use the following syntax:

```
/*<Subsystem_name.lower_blockname> */ <Comment to be transferred to HDL>
```

Note the following about block tagging:

- White spaces, illegal characters and multiline characters are all transferred to HDL.
- Comments for subsystems and blocks inside the subsystems are also transferred.
- If you have different comments for the same block, they are all transferred.
- If you have comments for blocks outside the Symphony boundary, they are not transferred to the RTL. The log file records this as follows: Cannot find the comment tagged block:
- If you fold two multipliers and each of them have different comments, these comments appear in the HDL for the folded multiplier.
- When the Symphony Register block is tagged, the comment is not transferred to the HDL code if the name of the block is Register. This is because register is an HDL keyword. To make the comment transfer to the HDL code, rename the block.
- For FPGA targets, you cannot use block tagging for the following blocks: Vector Concat, Vector Split, Vector Extract, and Vector Expand.
- If a Delay block is replaced by the retiming algorithm, the comment attached to this block might not be transferred to the generated RTL.

Keeping Signal Names in Generated RTL

You can ensure that the Symphony Model Compiler software keeps the signal names from the Simulink environment when it generates RTL.

1. To ensure that the signal names from the Simulink environment are maintained, make sure to name the signals in the Simulink environment.

Naming the signals ensures that the names are traced and transferred to the generated code.

2. Note the following about signal tracing:

- Do not use numbers, keywords or illegal characters in the signal name, because the compilers might not accept the code.
- If a name contains a white space, it will be converted to an underscore.
- The tool does not generate code for a signal or block that cannot reach the output port. Signals that do not reach the output port cannot be traced and transferred to the HDL code.

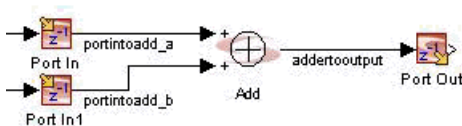
3. Run DSP synthesis.

The Simulink signal names appear in the RTL as follows:

- Signal names inside a subsystem are prefixed by <Subsystemname>_.
- If a signal is vectorized, each vector element is suffixed by _e<number from zero to vector size>.
- White spaces in the signal name are converted to underscores.
- Signal names are protected during retiming and the folding optimizations. If a delay is inserted on the signal during retiming, the signal going to the Delay block retains the original name. The signal name from the Delay block is named <signalname>_w<delaynumber>.

Example

The following shows a design with named signals and the code that is generated for it.



The resulting RTL shows that the Simulink signal names (portintoadd_a, portintoadd_b, and addertooutput in the examples below) are retained as signal names in the generated HDL code.

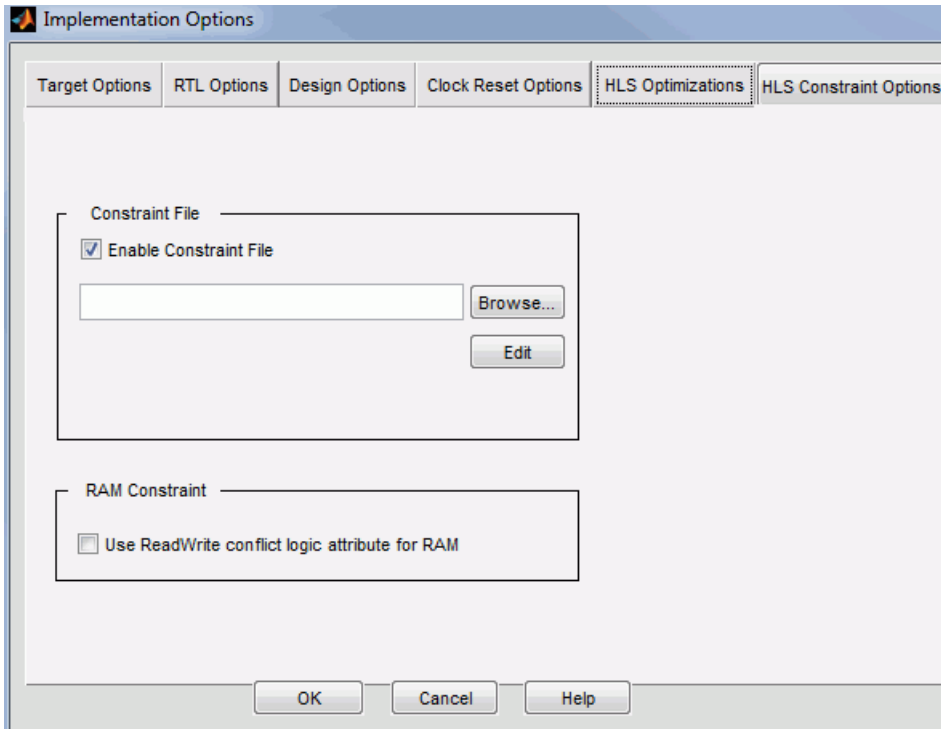
Verilog wire [15:0] portintoadd_a /*synthesis syn_keep=1 */;
 wire [15:0] portintoadd_b /*synthesis syn_keep=1 */;
 wire [15:0] addertooutput /*synthesis syn_keep=1 */;
 wire [0:0] GlobalEnableSignal1 /*synthesis syn_keep=1 */;
 wire GlobalResetSel;
 ...

VHDL attribute syn_keep : boolean;
 signal portintoadd_a : std_logic_vector (15 downto 0);
 attribute syn_keep of portintoadd_a : signal is true;
 signal portintoadd_b : std_logic_vector (15 downto 0);
 attribute syn_keep of portintoadd_b : signal is true;
 signal addertooutput : std_logic_vector (15 downto 0);
 attribute syn_keep of addertooutput : signal is true;
 signal GlobalEnableSignal1 : std_logic_vector (0 downto 0);
 attribute syn_keep of GlobalEnableSignal1 : signal is true;
 begin
 GlobalEnableSignal1 (0) <= GlobalEnable1;
 ...

Using Constraints

You can specify a Tcl file that contains constraints, using the following procedure.

1. Add the SHLSTool toolbox to your design and double-click the block.
2. Create a constraints file by typing in the constraints directly or through the GUI as described below:
 - Click New Implementation or Edit Implementation to open the Implementation Options dialog box for your synthesis run.
 - Click the HLS Constraint Options tab.
 - Click Enable Constraint File, type a name for the file, and click Edit. A window opens where you can enter the `define_attribute` constraints. See [HLS Constraints File, on page 620](#) for information about the file and the constraint syntax.
3. If you have already created a constraints file, do the following to use the constraints when you run the tool:
 - Click New Implementation or Edit Implementation in the SHLSTool toolbox to open the Implementation Options dialog box for your synthesis run.
 - Click the HLS Constraint Options tab.
 - Click Enable Constraint File, and specify the file you want to use.



4. Synthesize the design.

The tool applies the constraints you specified when it synthesizes the design. The log file reports how the constraints were applied, with messages like the following:

```
@N:The retiming constraints are successfully applied for the
following blocks or hierarchies in the following order:
```

```
  Subsystem.Delay_abc : {lock_only}
  Subsystem.Delay_abcl : {lock_only}
```

```
@W: "shls_retiming_lock {lock_only}" constraint is already applied
for the Delay_abcl block.
```

Using Retiming

The following provide more information about using the retiming optimization effectively:

- [Optimizing with Retiming, on page 655](#)
- [Using Automatic Gate-level Retiming, on page 660](#)

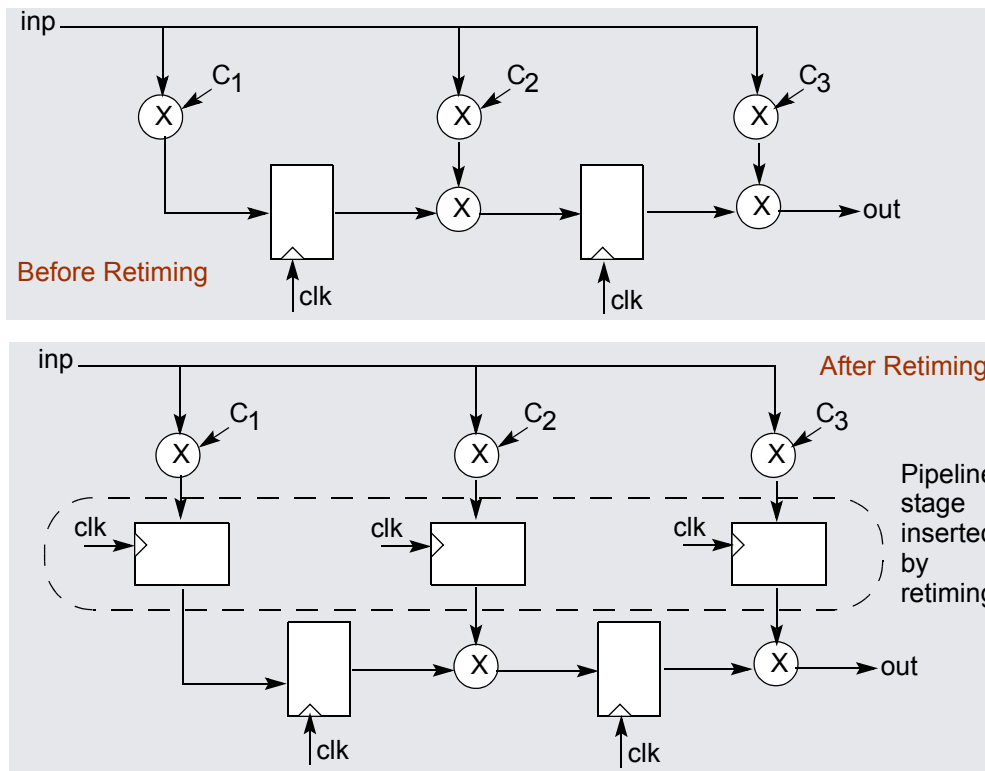
Optimizing with Retiming

The retiming optimization is described in more detail in the following sections:

- [The Retiming Optimization, on page 655](#)
- [Specifying Retiming, on page 657](#)
- [Specifying a Scale Factor Constraint for Retiming, on page 658](#)
- [Running Retiming Multiple Times, on page 658](#)
- [Retiming Register and Delay blocks, on page 659](#)
- [Retiming Tight Loops, on page 659](#)

The Retiming Optimization

Retiming is a performance optimization that improves speed by rearranging delays or adding extra delays if you specified them.



Retiming Limitation

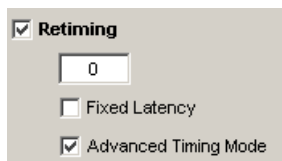
Currently, retiming has a limitation: if the ratio between design clocks is larger than $2e7$ in a multirate design, retiming becomes infeasible. When it encounters this situation, the tool does not generate RTL code but issues the following error message:

```
@E: Infeasible clock ratios. Ratios between implementation clock
frequencies must be lower than 2e7.
```

Specifying Retiming

Use the following procedure to use the retiming optimization in your design:

1. If you want to use advanced timing mode for retiming calculation, make sure you have selected this timing mode. See [Configuring SMC Timing Modes for FPGAs, on page 638](#) for details.
2. After setting up the implementation and target technology (see [Setting up Implementations, on page 644](#)), click Retiming in the SHLSTool window.



3. Set a value in the field that becomes available.
 - To retime your design without adding any extra latency to the design, set the value to 0. The Symphony Model Compiler tool rearranges the existing delays to improve performance, but does not add any latency. Retiming with a value of 0 maintains the original latency of the design.
 - To specify extra latency, set a positive value. For example, if you set it to 3, the Symphony Model Compiler tool has up to 3 extra pipeline stages available. The tool only uses as many stages as it needs. It might insert extra latency to meet aggressive timing goals. The latency of the design increases, depending on the number of pipeline stages actually used for retiming.
 - To specify a fixed amount of latency, enable the Fixed Latency option. When you enable this mode, the retiming engine retimes the design and then pads the outputs with the remaining delays so as to always maintain the specified latency. For example, if you specify more latency stages than needed for pipelining, the remainder of the latency stages pad the I/O.

If you have a multirate design, the tool may not insert all the extra latencies specified from the GUI. In these cases, the tool does not

generate RTL, but prints a warning message about the extra latency being infeasible:

```
@E: Infeasible latency value for fixed latency operation
    Use <x> or multiples for extra latency value
```

4. If needed, set retiming constraints on Delay blocks.

See [shls_retiming_lock Constraint, on page 169](#) and [Using Constraints, on page 653](#) for more information.

5. Set any other optimizations you want, and click Run.

6. Click View Log.

A window displays the log file, which records the specified number of optional cycles, and the number of cycles actually used to meet timing. It reports any increase in system latency because of the extra delays.

See [Running Retiming Multiple Times, on page 658](#) for information on using multiple retiming runs.

Specifying a Scale Factor Constraint for Retiming

You can account for the routing delay between the SMC timing estimate and the estimates of the logic synthesis tools by using the `retiming_scale_factor` constraint to specify how aggressive the Symphony tool should be during retiming. See [retiming_scale_factor Constraint, on page 628](#) for details.

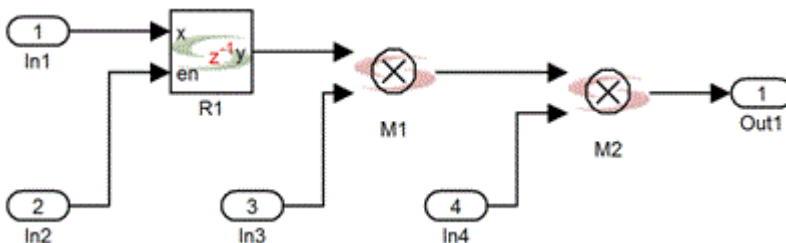
Running Retiming Multiple Times

The following procedure shows you how to use multiple retiming runs to get the best results:

1. Enable retiming and set a large retiming value.
2. Run DSP synthesis.
3. Check the `shls.log` file for the latency value used by the retiming algorithm to meet timing requirements.
4. Set this retiming value and enable the Fixed Latency option.
5. Rerun DSP synthesis and use the automatically generated RTL of this DSP synthesis.

Retiming Register and Delay blocks

Retiming can move Delay blocks but not Register blocks, because the latter have reset/enable signals on the registers. These ports make it difficult if not impossible to ensure functional correctness after retiming. Consider the following example where register R1 precedes a long combinational path. The register has an enable port that is driven by an independent input. The circuit must be retimed to break the combinational path.

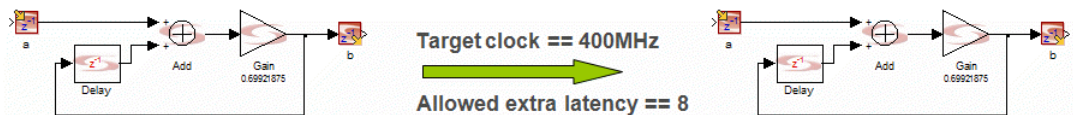


If no extra latencies are specified, retiming must use existing latencies. For this example, R1 must be placed between M1 and M2. But depending on the state of the enable signal, R1 holds the value of its input for some intervals of time, and these values are provided as input to the multiplier M1. If R1 is removed from the input of the multiplier, then it is impossible to preserve the stream of values produced by M1, thus breaking functional correctness.

The tool therefore excludes the Register block from retiming. However if the Register block does not have reset/enable ports, the tool treats it like a Delay block and moves it freely during retiming. You can set a constraint so that a Delay block is not optimized by retiming (see [shls_retiming_lock Constraint, on page 169](#)) but you cannot set this constraint on a Register block.

Retiming Tight Loops

Retiming does not automatically add delays to tight loops because that changes the functionality of the design, but you can manually insert delays.



If you enable Retiming and have tight loops, you see a report like the following:

TIMING INFORMATION

@W: Estimated system clock frequency does not meet target frequency.

Following loop prevents retiming to achieve target clock

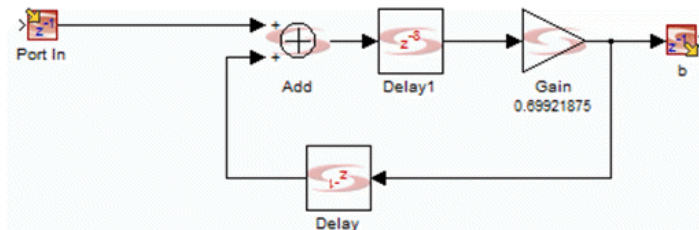
Gain

Add

ATM estimated clock(s):

Estimated clk: 138.888889 MHz

You can manually insert delays inside the loop as shown in the following figure to fix the tight loops and generate the results you want:



TIMING INFORMATION

@N: System meets timing requirements.

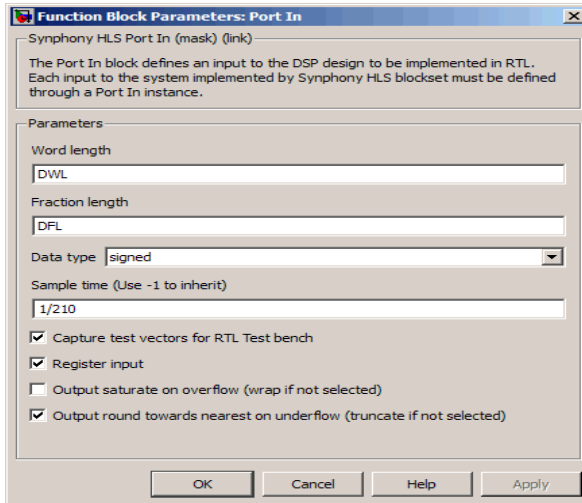
ATM estimated clock(s):

Estimated clk: 555.555556 MHz

Using Automatic Gate-level Retiming

To further increase performance, you can increase the sample rate for further gate-level retiming. The following procedure shows you this technique on an existing design.

1. In the model window, double-click the input port: to open the Function Block Parameters: Port In dialog box. Increase the sample rate and click OK.



2. Select the implementation and run DSP synthesis.
3. Click the Run button to execute the optimization and generate the RTL again. Click View Log to see the results:

When you check the log file, it shows that extra cycles have been inserted in the architecture.

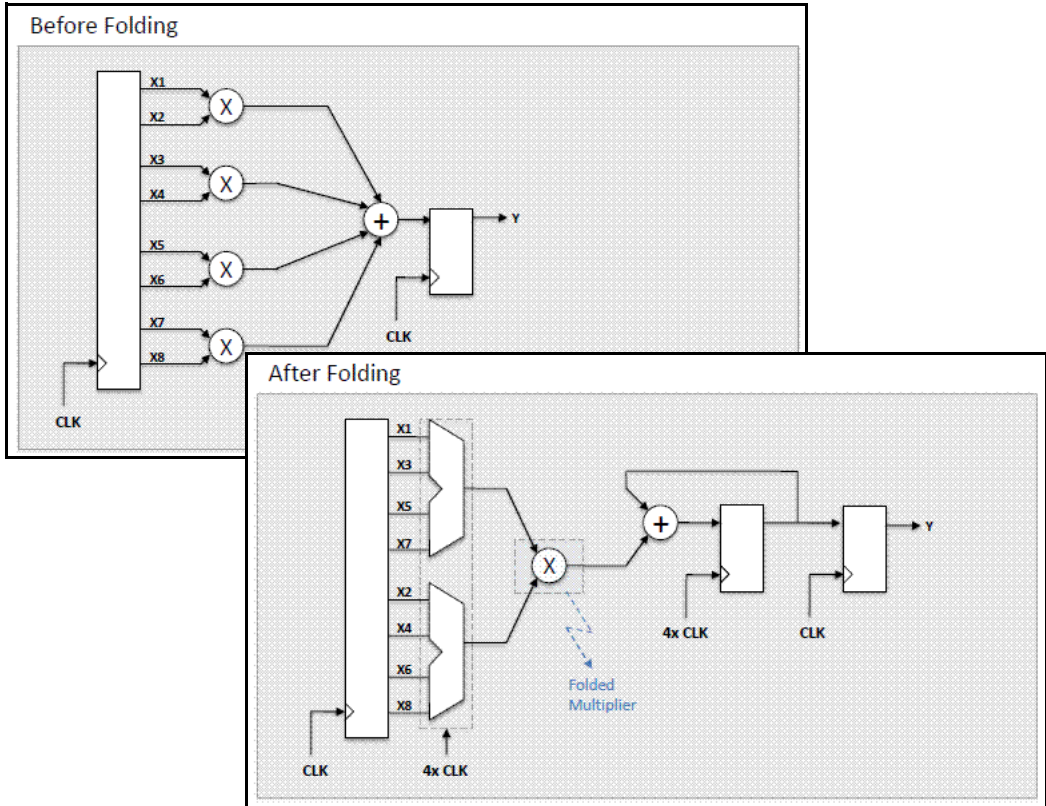
4. Start Synplify Pro, and run logic synthesis on the design.
When you examine the architecture with the HDL Analyst tool, you see that the outputs of the multipliers are double-registered. The first set of registers after the multiplier migrate inside the multipliers and create two pipeline stages. The second set of registers stays at the output of the multipliers to terminate the second pipeline stage. This optimized architecture improves target performance.

Using Folding

- [Optimizing with Folding, on page 662](#)
- [Using Pattern Folding, on page 665](#)
- [Using Annotations for Folding, on page 668](#)

Optimizing with Folding

Folding is an area optimization that shares resources. The Symphony Model Compiler tool folds the algorithm by reusing the same resources over multiple physical clock cycles. This helps the designer reduce the number of expensive functions like multipliers that take up a lot of silicon. The Symphony Model Compiler tool automatically puts in control logic near the multipliers so that they can be multiplexed.



When you use the Folding option, you cannot use Multi-channelize, which is an alternative mechanism to trade speed for resources. When the folding factor is set to 1 in multirate designs, the tool automatically folds the slow clock domain to the clock decimation amount.

1. In the Simulink window, double-click the SHLSTool block, and set up the implementation and target technology (see [Setting up Implementations, on page 644](#)).
2. Click Folding in the SHLSTool window. This automatically enables the Retiming and Pattern Folding options.
3. Set the Folding, Pattern Folding, and Retiming values:
 - For the Folding option, specify a minimum value for the number of system clocks you want to spend to compute one sample. If your

design has different sample rates, the Folding value applies to the fastest rate.

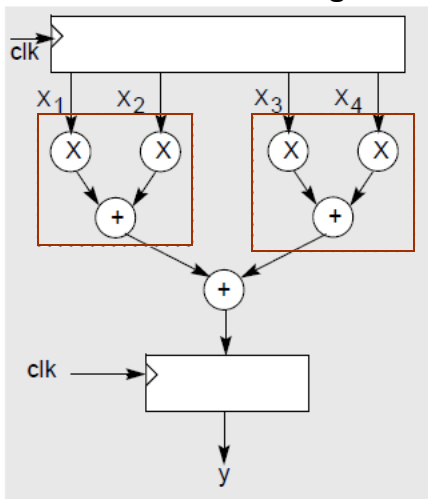
- To identify repeating patterns in the design for resource sharing, enable **Pattern Folding**. Then set a value in **Min Pattern Freq** as a guide, so that the software can ignore patterns that occur less frequently than this number. For full descriptions of these options, see [SHLSTool Toolbox Interface, on page 487](#). For more information about pattern folding, see [Using Pattern Folding, on page 665](#).
 - For **Retiming**, specify the extra latency you want to add, to be used for retiming. If you set this value to 0, the tool does not add any extra latency, but moves the existing delays for retiming. When you specify extra latency, the tool only uses as many pipeline stages as needed.
4. Set any other optimizations you want.
 5. Click **Run**.
 6. Click **View Log**.

A window displays the log file. The log file reports the actual number of system clocks used to compute a sample. It also shows the optional latency specified and the extra latency actually used for retiming. If you specified pattern-folding, it also reports pattern usage statistics. See [Pattern Usage Reports, on page 673](#) for details.

Using Pattern Folding

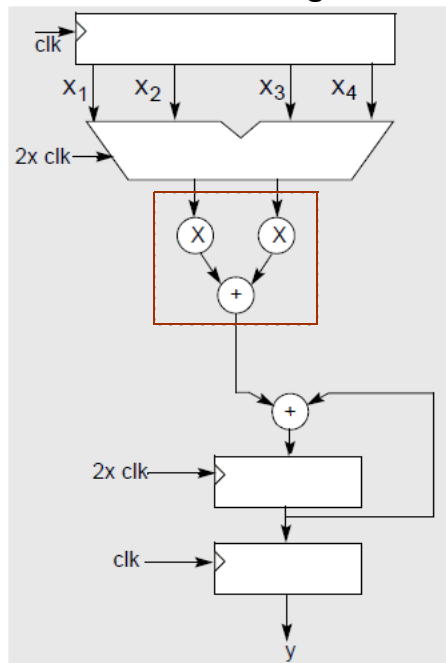
Pattern folding is an optional preprocessing step for folding, where the tool identifies and marks repeating patterns in the design, for time-multiplexed resource sharing when the folding algorithm is run. A *pattern*, in this context, is a group of Symphony Model Compiler block instances and the interconnect between these instances. Rate-changing blocks and I/O blocks are not identified as part of a pattern.

Before Pattern Folding



 Pattern

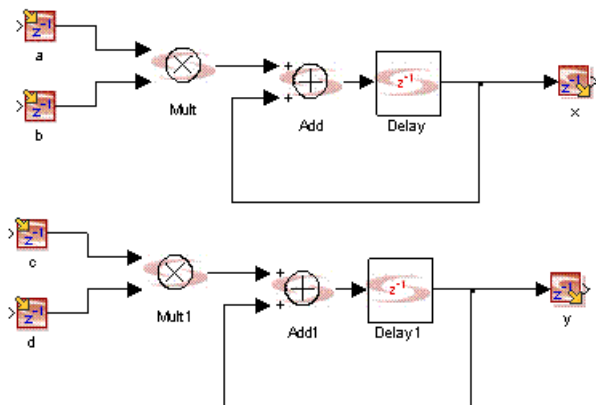
After Pattern Folding



Benefits of Pattern Folding

One of the key components of folding is time-multiplexed resource sharing. Each shared resource will have multiplexers at the inputs, which can be costly in FPGA designs. Excessive multiplexing can increase area significantly and slow circuit speeds. Pattern folding is an effective mechanism to reduce the multiplexing overhead, because as the pattern grows larger, the ratio of the multiplexing overhead gets smaller.

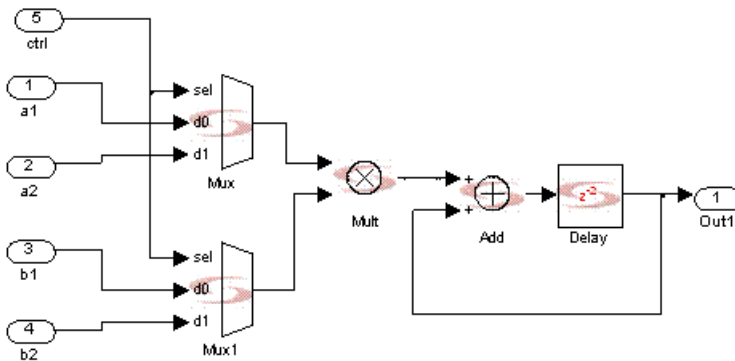
In the following design, folding by 2 results in a multiplexing overhead of 4 multiplexers. Using independent resource-sharing for the multipliers and adders would require 2 multiplexers for the multiplier inputs and another 2 for the adder inputs, resulting in an overhead total of 4.



However, if you identify the Multiply-And-Accumulate (MAC) pattern in this design, and resource-share this pattern, you only need two multiplexers (for the two inputs of the MAC pattern). This reduces the total multiplexing overhead. The following figure shows the resource-shared MAC pattern. Note that the delay is now 2, instead of the original value of 1. This is because we need to multichannelize the shared pattern by the folding factor (which is 2 in this example) for functional correctness.

Enabling Pattern Folding

Pattern folding is an optional step, so if you want to use it you must enable Folding and Pattern Folding on the SHLSTool toolbox, and supply a minimum threshold value for pattern repetition in Min Pattern Freq (see the procedure described in [Optimizing with Folding, on page 662](#)).



You can also manually identify and annotate patterns for pattern folding as described in [Setting Folding Annotations in the Simulink GUI, on page 668](#).

Current Limitations of Pattern Folding

Currently, pattern folding has the following limitations:

- Instances with the same pattern must have block parameters and I/O widths that match perfectly.
- There is a limit on the maximum number of blocks within a pattern. This is due to computational complexity. Pattern identification starts with small patterns, and iteratively grows patterns by combining smaller frequent patterns into larger candidate patterns, and then eliminates candidate patterns which are not frequently used in the design. If there is an exponential growth of candidate patterns, the tool terminates pattern identification.
- Identification of vectorized sections as patterns is limited.
- Estimates of pattern area are rudimentary. (This estimate is used to determine whether a pattern is large enough to overcome multiplexing overhead. Patterns that are found to be too small are not considered for resource sharing.)

Using Annotations for Folding

You can specify `pattern_annotation` constraints to guide folding. User-annotated patterns take precedence over automatic pattern detection. The tool always honors and folds annotated patterns, even when the minimum pattern frequency is set to such a high value that no pattern in your design would satisfy the requirement.

Manually annotating patterns does not preclude the pattern folding engine from identifying patterns; it still searches for other patterns in your design, in addition to the ones you annotate. Conversely, you cannot augment pattern annotation with pattern identification, so you must ensure that you have considered all instances to get the full benefits of pattern folding.

Setting Folding Annotations in the Simulink GUI

The following procedure shows you how to manually identify patterns that can be used for pattern folding, using the Simulink Tag property. Alternatively, you can set a `pattern_annotation` constraint in the Symphony Model Compiler constraint file, using the syntax described in [pattern_annotation Constraint, on page 626](#). See [HLS Constraints File, on page 620](#) for information about this file.

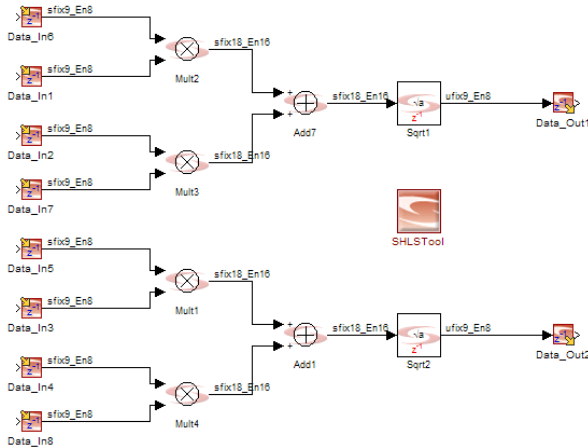
1. Ensure that all blocks that are part of the pattern instances you want to annotate are identical in the following respects:
 - Port enumerations
 - Propagated data types (fixed-point, sample rate, signal dimensions)
 - Blocks that make up the pattern (incoming data types, block mask specifications)
 - Block interconnections

If the annotated instances of a pattern are not identical, the pattern folding engine ignores all corresponding annotations and does its own pattern search. It is recommended that you first create one instance of a pattern and then create other instances from copies of the original instance.

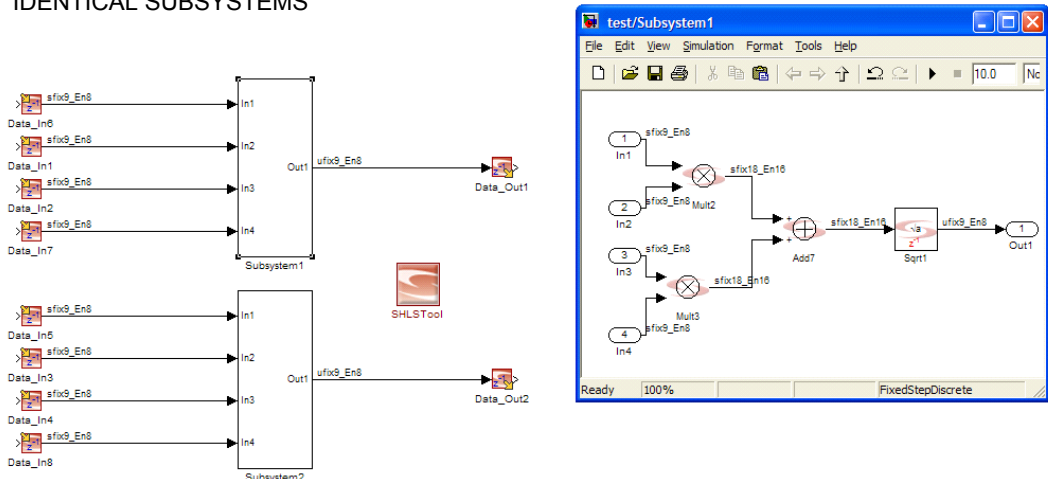
2. Make sure that all pattern instances are single rate.

3. Encapsulate all blocks that are part of pattern instances into Simulink subsystems. The following figures show a design before and after encapsulation.

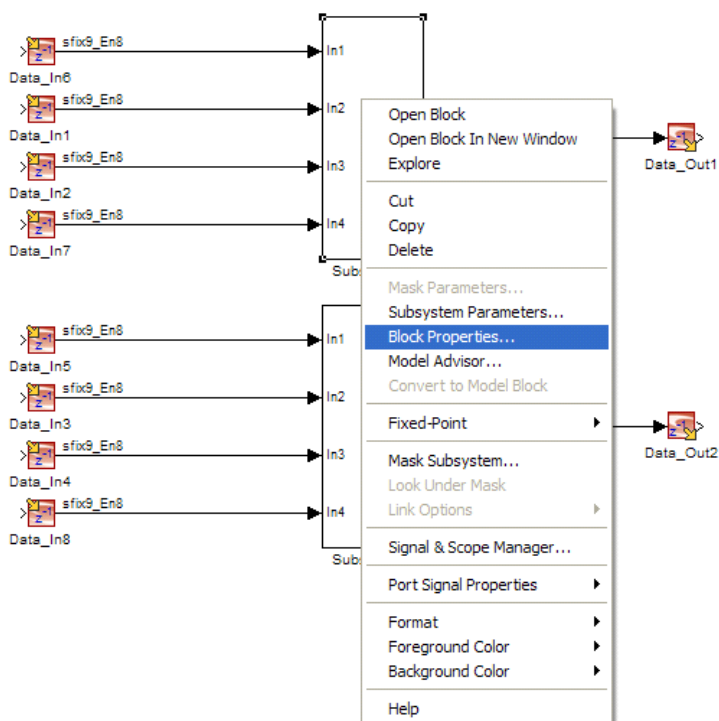
ORIGINAL DESIGN BEFORE ENCAPSULATION



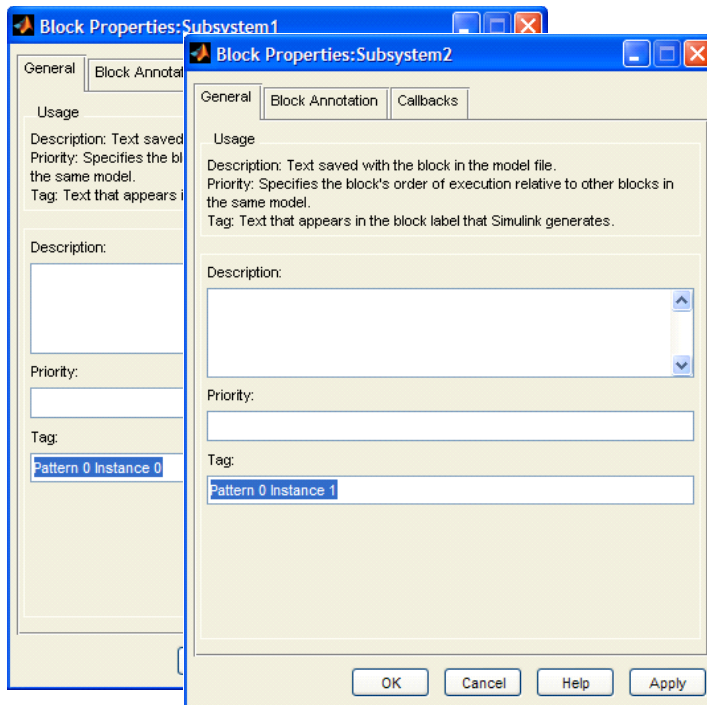
DESIGN ENCAPSULATED IN TWO IDENTICAL SUBSYSTEMS



4. Tag each instance that is part of the pattern with a unique pattern-instance ID combination.
 - Right-click the encapsulating subsystem, and select Block Properties.

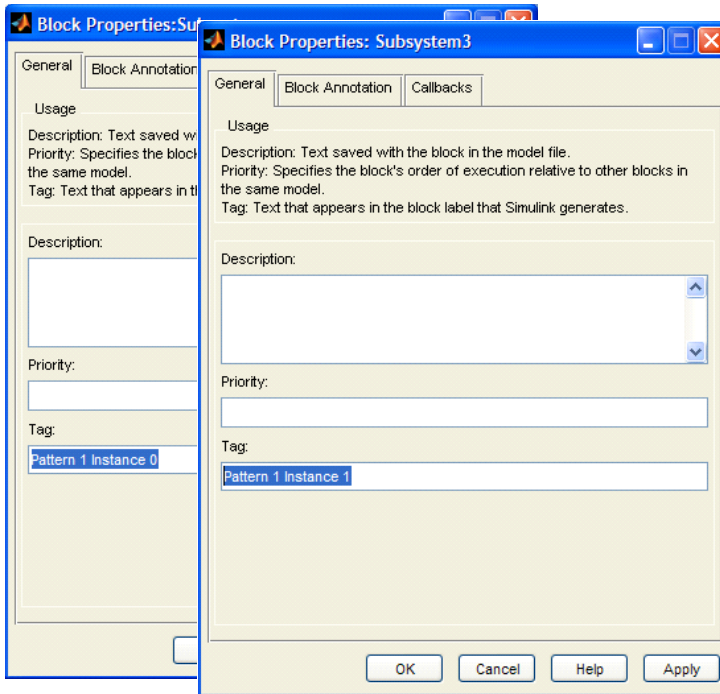


- In the dialog box that opens, type a unique pattern-instance ID string in the Tag field. Use natural numbers for the IDs. The following figure shows unique values set for two identical subsystems.



Alternatively, you can set pattern-instance IDs from the command line, as described in step 5, below.

- If you have other patterns, annotate the pattern instances in the same way, using unique tags:



5. Alternatively, you can set pattern-instance IDs from the MATLAB command line by typing this command:

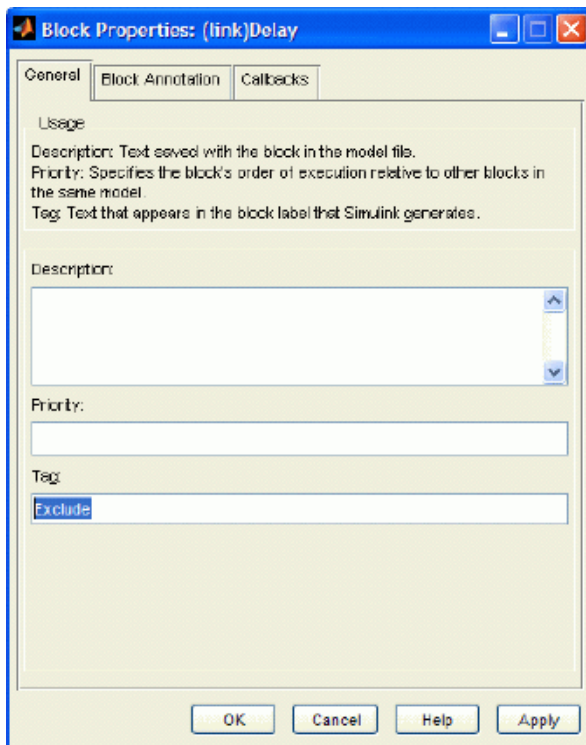
set_param(<path_to_instance_subsystem>, 'Tag', <annotation_string>)

<path_to_instance_subsystem> is the string returned by the MATLAB `gcb` command for the subsystem you are identifying for pattern annotation.

<annotation_string> is the unique pattern and instance ID. Remember that the numbers must be natural numbers.

For example: `set_param(myModel/Subsystem1', 'Tag' , 'Pattern 0 Instance 0')`

6. To exclude a block from all identified patterns, including the tool pattern search, do the following:
 - Right-click the encapsulating subsystem, and select Block Properties.
 - Type Exclude in the Tag field.



Pattern Usage Reports

The Pattern Usage Statistics section of the Symphony log file summarizes the statistics for each pattern identified in the user design. For each pattern, it lists three statistics:

- Number of primitive blocks in the pattern netlist
- Number of times the pattern occurs in the original netlist
- Instantiated number of pattern in the generated RTL (after folding)

Here is an example of pattern usage statistics:

```
PATTERN USAGE STATISTICS
*****

2 distinct patterns are used in the system
-----

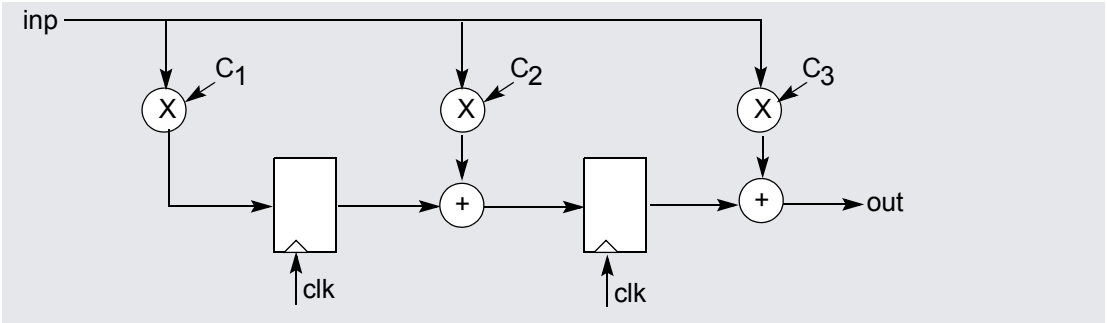
Pattern 1
-----
@N: Number of blocks in the pattern: 3
@N: Occurrence in the original netlist: 10
@N: Number of instantiated devices: 2
-----

Pattern 2
-----
@N: Number of blocks in the pattern: 4
@N: Occurrence in the original netlist: 15
@N: Number of instantiated devices: 3
-----
```

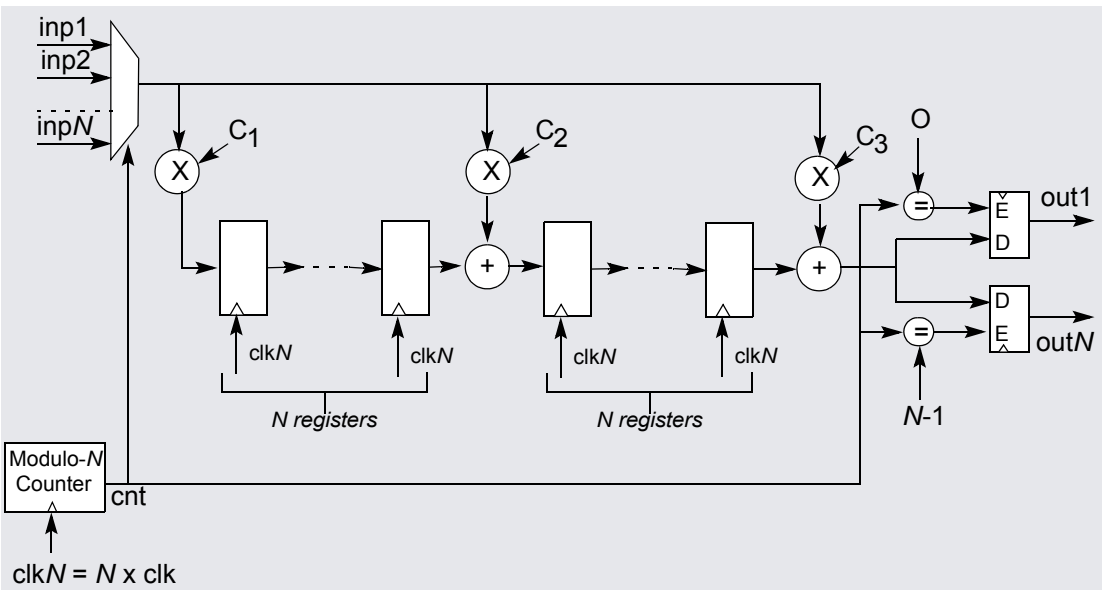
Optimizing with Multichannelization

Multichannelization is another optimization that trades speed for resources. It helps you minimize hardware by sharing the hardware over multiple channels. You can use different implementations to explore different channel widths and analyze multi-thread capacity. Multichannelization and folding (see [Optimizing with Folding, on page 662](#)) are mutually exclusive.

Before Multichannelization



After Multichannelization



1. In the Simulink window, double-click SHLSTool, and set up the implementation and target technology (see [Setting up Implementations, on page 644](#)).
2. Click Multi-channelize in the SHLSTool window.
3. Set the maximum number of channels in the text box. For example, if you set it to 3, the tool can create three channels.

4. Set any other optimizations you want, and click Run. When you set Multi-channelize, you cannot also use Folding.
5. Click View Log. A window displays the log file. The log file reports the number of system clocks used to compute a sample. It also shows the number of delays specified and the number of registers actually used for retiming.

Running Synthesis with SHLSTool

The SHLSTool toolbox lets you specify different optimizations and generate RTL code. After setting the implementation and optimization options described in [Running Synthesis with SHLSTool, on page 677](#), do the following to run DSP synthesis and generate the output files.

1. Set the optimizations you want, and click Run in the Symphony Model Compiler window.

The tool runs with the targets you set and generates the output files you specified. It generates a log file called shls.log after synthesis.

You can only use the Synplify Pro or Synplify Premier tools for FPGA logic synthesis. Symphony Model Compiler generates the following files for logic synthesis. The files are in the model directory, under the VHDL and Verilog subdirectories.

File	Description
.prj	Project file for synthesis.
.vhd .v	VHDL/Verilog netlist for synthesis. Each format has a separate subdirectory under the implementation directory.
.fdc	Constraint file for synthesis
..vhd	Optional testbench for simulation

The software also generates a testbench for verification. The files are in the model directory.

Synthesizing with a Host Interface Block

The Host Interface block provides a slave interface to a simple interconnect protocol, which lets the design interface with the host processor and load the memory-mapped configuration registers required by the SMC design.

1. If you have not already done so, configure the compiler in MATLAB to compile s-functions.

This is usually done during installation with the `setup` script. If you did not do this at that time, run `mex -setup` in the MATLAB console to configure the compiler. Check the Release Notes for a list of supported compilers.

2. Add the SMC Host Interface block to your design.
3. Double-click the Host Interface block and specify the following parameters:
 - On the Bus interface tab, specify the bus protocol to use and set the parameters for the protocol. For details about the bus protocols, see [Bus Protocols, on page 738](#).
 - On the Memory map tab, add as many registers as are needed and specify the parameters for each one. See [SMC Host Interface, on page 326](#) for complete descriptions of all the Host Interface block parameters.
 - You can save the settings to a csv file or an IP-XACT-compatible xml file.
 - Click OK.

The tool generates the Verilog RTL files that implement the behavior specified by the block parameters, and also generates C code for the block from the RTL implementation. Finally, it creates a Simulink wrapper that you can use to simulate the behavior of the block implementation in Simulink.

4. Run high-level synthesis on your entire design, which includes the Host Interface block.

The SMC tool instantiates the Host Interface block in the final RTL, along with its connectivity to the SMC design as set up in the Simulink model. The SMC tool includes the host interface RTL files in the simulation and synthesis scripts it creates.

Take note of the following points when simulating and synthesizing a model that includes the Host Interface block:

- Never register the input and output bus protocol interface signals in the SMC design. This is because all the protocols have a handshake mechanism which would no longer be compliant with the protocol if the bus interface signals are delayed. This leads to unexpected results.
- You cannot use multichannelization during synthesis. Multichannelizing the block would mean replicated memory-mapped registers with the same address, which is not supported.
- You cannot use folding. Folding creates registers on the ports and typically introduces delay on the bus interface signals, which leads to the failure to meet bus protocol specifications.
- You cannot use retiming, because this optimization tends to create registers on the bus protocol signal outputs from the Host Interface block. The registers violate the handshake mechanism between the Host Interface block and the bus master.
- Reset sensitivity and polarity must be asynchronous and active low. All supported bus protocols mandate this.

CHAPTER 7

Underlying DSP Fundamentals

The following sections describe how the Symphony Model Compiler software interprets and handles some basic issues of DSP design:

- [Clock Domains, on page 682](#)
- [Resets in the SMC Tool, on page 683](#)
- [Clock and Reset Management, on page 686](#)
- [Data Types, on page 695](#)
- [CORDIC Algorithms, on page 701](#)
- [Multi-Rate Design, on page 717](#)
- [Hierarchy Preservation, on page 728](#)
- [Subsystem Consolidation, on page 729](#)
- [Block Consolidation, on page 730](#)
- [Constant Propagation, on page 731](#)
- [RAMs, on page 733](#)
- [Bus Protocols, on page 738](#)

Clock Domains

The Symphony Model Compiler tool derives the signal clocks and implementation clocks required by optimization algorithms automatically from the Simulink design. The software uses MATLAB sample domains to derive physical clock domains. Alternatively, you can generate a top-level module that contains the clock and reset information for the design, as described in [Specifying a clock_reset Module, on page 755](#).

This section describes how the Symphony Model Compiler tool handles signal and implementation clocks. The tool automatically derives signal and implementation clocks.

Derived Signal Clocks

Signal clocks (sample rate) are defined or derived at the Port In and Port Out blocks, where the sample rate of the signal is specified. The tool uses the standard notation [*<sampleTime.time>* *<sampleTime.offset>*] with the different notation permutations on the input ports to sample signals. See the Simulink documentation on *Modeling and Simulating Discrete Systems* for details of the permutations.

If the sample definition is inherited (*<sampleTime.time>* == -1), the sample rate is propagated from the encapsulating block.

The software treats all ports with the same *<sampleTime>* as belonging to the same clock domain. The Upsample and Downsample blocks create separate sample domains of the design, within a different but related derived clock.

Derived Implementation Clocks

The Symphony Model Compiler optimization algorithms require an implementation clock (clock period) for each clock domain derived from the design. The tool automatically derives the clock domains, defined by a set of input port blocks and the Upsample and Downsample blocks.

Resets in the SMC Tool

Sequential logic circuitry requires a reset signal to start and/or return to a known state. The following describe how the Symphony Model Compiler tool handles resets in the design.

- [Global and Local Resets, on page 683](#)
- [Synchronous and Asynchronous Resets, on page 684](#)
- [Reset Implementation in RTL Code, on page 685](#)
- [Resets and RTL Testbenches, on page 686](#)

For step-by-step procedures to define resets, see [Defining Clocks and Resets, on page 754](#).

Global and Local Resets

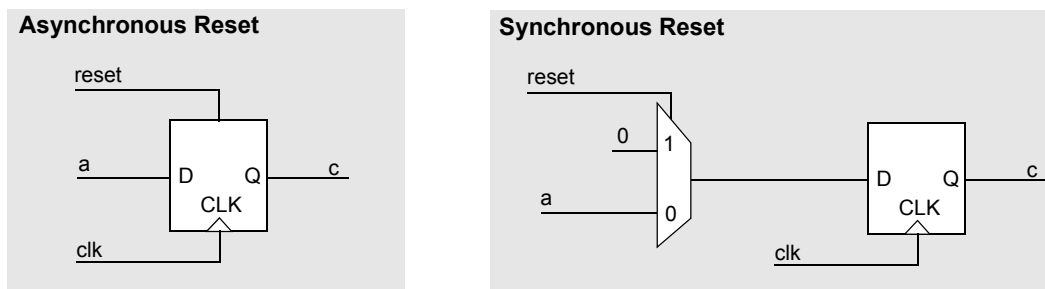
The circuitry produced by the Symphony Model Compiler tool has two kinds of resets that can be used together to bring all the flip flops in the design to a known state:

- An optional local reset, which is visible at the Simulink level, and is under the designer's control. It always executes a synchronous reset.
- A global reset that can be set to either synchronous or asynchronous in the Design Options tab of the Implementation Options dialog box. If you want to set the global reset automatically for your target FPGA, set Flip Flop Reset Sensitivity to Automatic.

You can also set the polarity for the global reset signal in the Flip Flop Reset Polarity option. See [Defining Reset Signals, on page 758](#) for a step-by-step procedure.

Synchronous and Asynchronous Resets

With a synchronous reset input, circuitry operates only on the positive edge of the clock and the tool treats the reset input as another, albeit highest priority, input which clears the flip-flop when asserted. Asynchronous reset inputs are built into the design of the flip-flop itself and bypass the clock to clear the output of the flip-flop right away.



The synchronous reset approach adds additional logic (a multiplexer) to the data path compared to an asynchronous reset. If gate count is an issue, the asynchronous reset flip-flop is more complex, and any reduction in the number of gates because of the multiplexer may be offset by the increase in the number of gates in the flip-flop. With the current die sizes, this reduction is probably not significant and the decision should be based on other factors depending on the design.

For FPGA implementations, basic cells support asynchronous resets by default, and they turn off this feature to implement synchronous resets. For some architectures, moving from synchronous to asynchronous resets can mean shaving off a multiplexer from the data path. This leads to higher clock speeds and reduced consumption of the logic resources on the FPGA. For example, the Symphony Model Compiler tool reduces logic consumption in FPGAs that have only asynchronously resettable flip-flops, by using asynchronous resets instead of synchronous resets. For architectures that have configurable flip-flops, using asynchronous resets does not result in reduced resource consumption.

Reset Implementation in RTL Code

You can implement asynchronous resets globally in your Symphony Model Compiler design by setting the option described previously. The resulting RTL code does not change in function, i.e. all the signals and statements remain as they were, but the sensitivity list of all process (VHDL) or always (Verilog) statements includes a reset input along with the clock.

- For synchronous resets, the sensitivity list for the process or always statements contain just the clock signal.

VHDL Synchronous Reset

```
process(clk)
begin
  if (rising edge (clk)) then
    if (rst=1') then
      out signal <= reset_value;
    elsif (en=1') then
      out signal <= logic(inputs);
    endif;
  endif;
end process;
```

Verilog Synchronous Reset

```
always @(posedge clk)
begin
  if (rst)
    myreg <= reset_value;
  else if(en) //enable
    myreg <= logic (inputs);
end
```

- For asynchronous resets, the statements also contain the reset signal as a trigger because any change in the reset signal cause the statement to execute.

VHDL Asynchronous Reset

```
process(clk, rst)
begin
  if (rst=1') then
    out signal <= reset_value;
  elsif (rising_edge (clk)) then
    if (en=1') then
      out signal <= logic(inputs);
    endif;
  endif;
end process;
```

Verilog Asynchronous Reset

```
always @(posedge clk or posedge rst)
begin
  if (rst)
    myreg <= reset_value;
  else if(en) //enable
    myreg <= logic (inputs);
end
```

Resets and RTL Testbenches

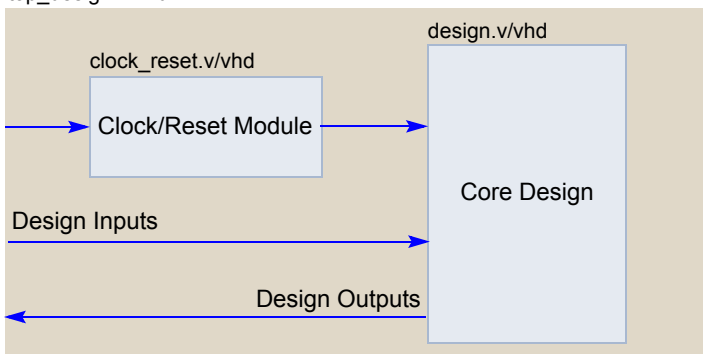
The testbenches generated by the Symphony tool apply a global reset to the RTL code once, at the start of the simulation, to bring the RTL simulation to a known state. This ensures that the Simulink model and the RTL model have the same initial state.

- For a global synchronous reset, the tool first asserts the reset signal (i.e. logic 0). Then it forces all clocks in the design to a positive edge transition, and resets the RTL code synchronously. After a while, the reset is released (t0), and all the clocks start ticking with their respective periods after this time, t0.
- For asynchronous resets, the RTL test bench code generated by the Symphony tool applies a reset with no clock activity. The reset signal is included in the sensitivity list so that the RTL simulation is reset with the application of the reset signal in the absence of a positive edge clock transition. The Simulink model and the RTL testbench functional inputs are the same, which means that the output results are functionally the same as those produced with a global synchronous reset option. The functionality remains unchanged.

Clock and Reset Management

Instead of specifying signal and implementation clocks in the RTL code directly or implementing clock generation logic around the core design, you can specify clocks and resets as a top-level module in the Symphony Model Compiler design. This circuitry not only generates design clocks from the input oscillators but also generates global reset signals from input reset signals, using proper timing. The following figure shows the top-level connections for this structure.

top_design.v/vhd



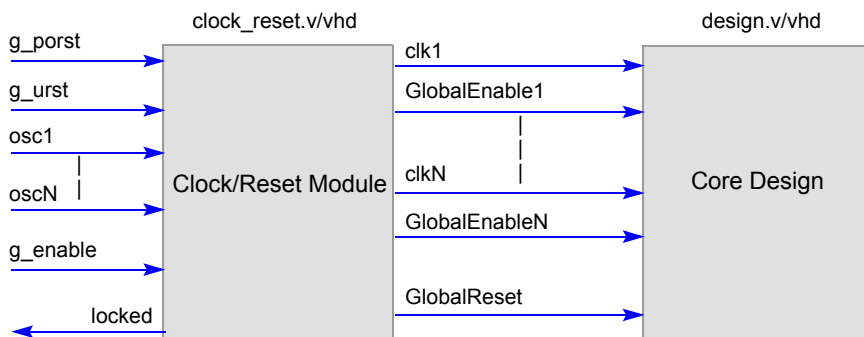
For more information, see the following:

- [Clock_reset Module Interface](#), on page 688
- [Reset Functionality with the Clock_reset Module](#), on page 689
- [Clock Functionality with the Clock_reset Module](#), on page 689
- [Clock/Reset Circuitry Files](#), on page 690
- [Clock_reset Module Limitations](#), on page 690
- [Log File Messages for the Clock_reset Module](#), on page 691

For a step-by-step procedure, see [Specifying a clock_reset Module](#), on page 755.

Clock_reset Module Interface

The following shows the signal interface between the clock_reset module and the core design:



g_porst	Power on reset signal. Input to the top-level design. See Reset Functionality with the Clock_reset Module, on page 689 for more information.
g_urst	User reset signal. Input to the top-level design. See Reset Functionality with the Clock_reset Module, on page 689 for more information.
osc1...oscN	Oscillator input(s) to the top-level design.
g_enable	Top-level global enable input signal. Polarity of the signal is active high. GlobalEnable signals follow g_enable. When g_enable is held low, all GlobalEnable signals are held low.
locked	PLL locked signal output. Used only for test bench purposes. Left unconnected in the top-level design.
clk1..clkN	Clock outputs of the clock_reset module. Clock signals for each of the clock domain of the core design.
GlobalEnable1... GlobalEnableN	Global Enable signals (clock enables) associated with each clock domain of the core design.
GlobalReset	Reset output of the clock_reset module. Global reset input of the core design.

Reset Functionality with the Clock_reset Module

The clock_reset module applies the GlobalReset signal with proper polarity and timing to the core design according to the input reset signals (g_porst, g_urst). Note that there can be two reset pins added to the top-level design:

- The power on reset pin, g_porst, which is enabled by default.
- The user reset pin, g_urst, which is optional.

Each of the reset input signals has the same priority, so when the g_porst or g_urst signal is asserted, the GlobalReset signal is asserted and the core design is held in reset state. At the same time, the clock generation circuitry inside the clock_reset module is held at reset state. Because of the synchronization circuits, the input reset signals must be held asserted for a minimum of two clock cycles (in terms of the slowest oscillator period input to the design). When the g_porst/g_urst signal is de-asserted, the clock generation logic inside the clock_reset module starts functioning. When the clocks become stable, the GlobalReset signal is de-asserted and the core design also starts functioning.

Clock Functionality with the Clock_reset Module

You can set various options for the clock_reset module from the Clock Reset Options of the Implementation Options dialog box. For details about these options, see [Clock Reset Options, on page 497](#).

Clocking Scheme • For separate design clocks, set this option to **Dedicated Clocking Scheme**.
 • To use the same clock for each domain in the core design, select **Enabled Clocking Scheme**. With this setting, the tool achieves required clock frequencies by connecting the appropriate clock enable signals.

Clock Circuit Type This defines the internal structure of the clock reset module when it generates design clocks from the oscillator input to the top-level design.

- **Synthesizable Dividers**
 For synthesizable dividers, the tool uses clock divider logic while generating design clocks. This clock divider logic can be used for implementing 1/N division ratios. For M/N ratios, you must use the Enabled Clocking Scheme.
- **Generic PLL**
 The tool generates design clocks using placeholder code to represent common PLL structures inside FPGAs. This code should be filled or changed with the actual PLL or clock manager logic by the user for the specific FPGA chosen.

Set of Clock Sources	<p>This specifies available oscillator frequencies as a row or column vector.</p> <ul style="list-style-type: none"> • If this field is left blank, the tool automatically takes the least common multiple frequency of clock signals by looking at the sample rates of the Simulink design. • If a value or values are specified, the tool uses that value as the oscillator frequency.
----------------------	--

Clock/Reset Circuitry Files

When clock-reset circuitry generation is enabled, the following files are created under implementation folders in addition to the other design files.

New Files	Description
clock_reset.v/vhd	The HDL file that describes the clock reset module
top_<model_name>.v/vhd	Top-level design file that is a wrapper for the clock reset module and core design
top_<model_name>.sdc	Top-level constraints file
top_<model_name>.prj	Top-level synthesis tool project file
top_<model_name>_Test.v/vhd	Top-level test bench file that instantiates top_<model>.v/vhd as the design under test*
top_<model_name>_activehdl.do	Top-level Active HDL simulator do file*
top_<model_name>_affirma.do	Top-level Affirma simulator do file*
top_<model_name>_modelsim.do	Top-level Modelsim simulator do file*
top_<model_name>_vcs.do	Top-level VCS MX simulator do file *

* This file is generated when you select **Generate RTL test bench**.

Clock_reset Module Limitations

Currently clock reset management feature has the following rules and limitations:

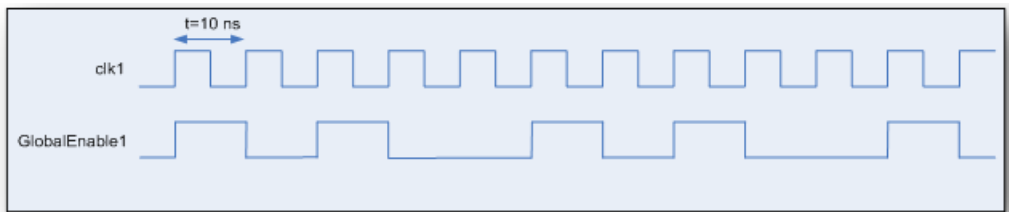
- Because of reset synchronization logic, the external reset signals `g_porst` and `g_urst` must be applied for a minimum of two clock cycles (in terms

of the slowest oscillator period input to the design) to ensure proper reset assertion.

- There must be a rational ratio between the design clocks and the oscillator frequency values entered in Set of Clock Sources field. Do not enter 13.45 MHz as the frequency value for a 10 MHz design clock. If you do, you get a warning message.
- Both the Dedicated Clocks and Enabled Clocking schemes can handle $1/N$ (N = integer) clock ratios between design clocks and oscillator frequencies. For M/N (M, N = integer) clock ratios use Enabled Clocking mode, because the implementation uses global enable signals to generate M/N ratio.

In the following example, the waveforms are generated for a 40 MHz design clock from an oscillator with 100 MHz frequency. Note that GlobalEnable1 is active in two cycles for every five clk1 cycles.

$$M/N = 2/5$$



- When you select Enabled Clocking as the clock scheme, you cannot set Clock Circuit Type. This is because all design clocks are connected to a common fast clock in enabled clocking, and there no clocks are generated in this process. Global enable signals are generated accordingly.

Log File Messages for the Clock_reset Module

The Clock Relationships section of the Symphony log file reports the oscillator sources specified in Set of Clock Sources. The following table shows the log file clock reporting for a multirate design with three different clock domains of 62.5 MHz, 125 MHz, and 500 MHz:

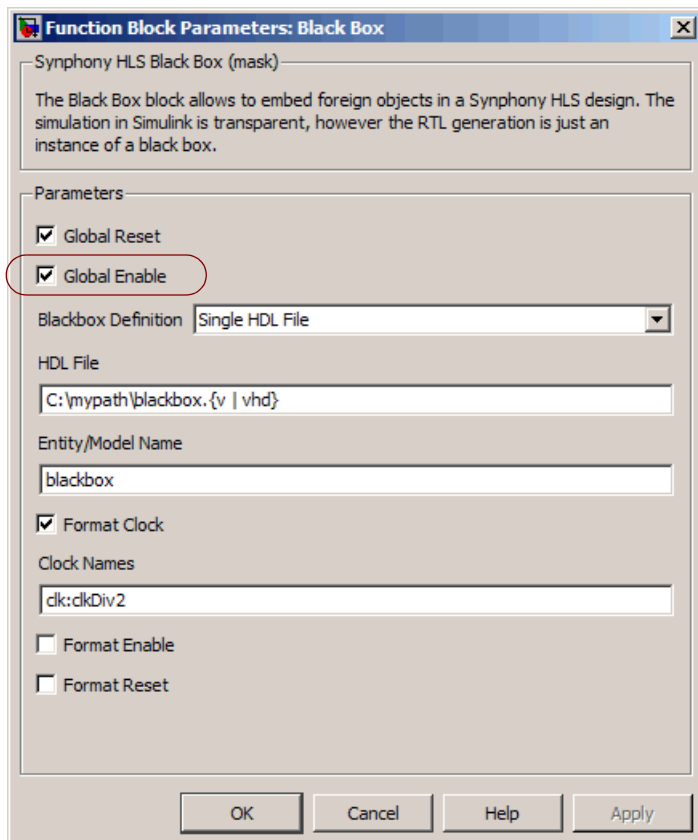
No sources are specified	<p>The input oscillator frequency used is 500 MHz and the log file reports the following:</p> <pre>Oscillator Inputs for Clocking Circuitry ----- osc1 500.000000 MHz.</pre>
Sources are specified as [33.3;100;125;500]	<p>The log file reports the following:</p> <pre>Oscillator Inputs for Clocking Circuitry ----- osc1 33.300000 MHz. osc2 100.000000 MHz. osc3 125.000000 MHz. osc4 500.000000 MHz.</pre>

The Synphony Model Compiler tool can also report notes, errors, and warning messages for the clock_reset module. The following table lists the messages and the situation that causes the message to be generated.

Message	Cause
@E: Use of rate multiplier clock implementation requires Enabled Clocking scheme.	<p>You see this message if you selected Dedicated clocking as the clocking scheme, but when there is no simple ration (1/N) between the design clocks and the oscillator frequencies. To avoid this message, select Enabled Clocks as the clocking scheme.</p>
@E: Clock frequency XXX MHz in the design cannot be generated from the given oscillator(s).	<p>The design clock frequency is not rationally related to oscillator frequencies. In most cases, the design frequency is faster than the supplied oscillator frequencies.</p> <p>Make sure that for all sample times in the design the ratio of the clock frequency to at least one oscillator frequency is one of the following:</p> <ul style="list-style-type: none"> • An integer reciprocal (results in a regular clock divider) • A rational number smaller than 1 (results in a rate multiplier)

Message	Cause
@E: Cannot generate clocking circuitry for Enabled Clocking scheme with rate multipliers if there are multi-rate blocks (FIFO, RAM, BLACKBOX) in the design. You can avoid rate multipliers by changing oscillator frequencies.	If you select an enabled clocking scheme, the tool does not generate RTL for designs that contain multi-rate blocks (FIFOs, RAMs, black boxes).
@E: Clock division ratio limit exceeded in rate multiplier implementation. For clock division ratios of M/N, the denominator(N) value cannot be larger than 1024.	There is a limit of 1024 on the denominator value (N) in rate multiplier implementations. If the design frequency is 3 MHz and the available oscillator frequency is 1025 MHz, the clock ratio (M/N) is 3/1025. N is therefore greater than the current limit of 1024.
@N: Generation of Clock-reset circuitry enabled.	The Generate Clock-Reset Circuitry option was selected.
@N: Oscillator source file not specified. Using least common multiple frequency value.	No clock source was specified.
@W: Rate multipliers were used for clock synthesis of some clocks.	There is no simple ratio (1/N) between the design clocks and the oscillator frequencies. A clock is synthesized from a regular clock divider if its frequency is a divisor of oscillator frequencies. Otherwise, you need rate multipliers if any design clock frequency is rationally related to and slower than the oscillator frequencies.
@E: Black box blocks should have GlobalEnable connections for the enabled clocking scheme	If you select Enabled Clocking as the clocking scheme, all Synphony blocks must have their Global Enable mask parameters selected. If you have Enabled Clocking selected in a design with black boxes, the tool checks if the Global Enable parameter is in place for all Synphony Black Box blocks. See the following figure.

This figure shows the Global Enable parameter set for a black box:



Data Types

The term **data type** refers to the way in which a computer represents numbers in memory. In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type. The data type determines the amount of storage allocated to a number, the method used to encode the number's value as a pattern of binary digits, and the operations available for manipulating the type. Different data types have specific advantages in the areas of precision, dynamic range, performance, and memory usage.

This section describes the following topics:

- [Fixed-Point and Floating-Point Representation, on page 695](#)
- [Symphony Model Compiler Data Type Implementation, on page 696](#)
- [Fixed-Point Data Type, on page 696](#)
- [Data Type Casting: Setting the Output Data Type, on page 697](#)
- [Matrix Data Types, on page 698](#)

Fixed-Point and Floating-Point Representation

Numbers are represented as either fixed-point or floating-point data types.

- The fixed-point data type is characterized by the word length in bits, the binary point, and whether it is signed or unsigned. The binary point position defines the scaling of fixed-point values. A common representation of a binary fixed-point number (either signed or unsigned) is shown below.



See [Fixed-Point Data Type, on page 696](#) for additional information.

- Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field.



Synphony Model Compiler Data Type Implementation

The Synphony Model Compiler software uses signed/unsigned fixed-point representation for the data types, because it offers advantages in terms of power consumption, size, memory usage, speed, and cost of the final product, compared to the floating-point representation. Synphony Model Compiler uses the new fixed-point data type that was added to the Simulink framework. The fixed-point data type is available in the Signal Processing Toolbox in MATLAB 7 and the DSP blockset in MATLAB 6.5p1.

Fixed-Point Data Type

The fixed-point data type supports integers, fractionals, and generalized fixed-point numbers. The main difference between these data types is the location of the binary point:

Integers	The binary point for signed and unsigned values is assumed to be just to the right of the LSB.
Fractionals	The binary point for unsigned fractional values is just to the left of the MSB, while for signed fractionals the binary point is just to the right of the MSB.
Generalized fixed-point numbers	The binary point can be anywhere in the word.

Simulink supports fixed-point word lengths up to 128 bits.

For information about setting the fixed-point data type in the Synphony tool, see [Using Quantization Analysis Tools, on page 832](#).

To display the data types of ports in your model, select **Port data types** from the Simulink Format menu. The port display for fixed-point signals consists of three parts: the data type, the number of bits, and the scaling.

Data type	Reflects the value of the block's Output data type parameter, or the data type that is inherited from the driving block or through back-propagation.
Number of bits	Reflects the value of the block's Output word length parameter, or the word length that is inherited from the driving block or through back-propagation.
Scaling	Reflects the value of the block's Output fraction length parameter, or the scaling that is inherited from the driving block or through back-propagation.

Whenever you start a simulation, enable display of port data types, or refresh the port data type display, Simulink performs a processing step called data type propagation. This step determines the data type for signals whose type is not otherwise specified, and checks that the data types of signals and input ports do not conflict. If there is a type conflict, Simulink displays an error dialog that specifies the signal and port whose data types conflict. Simulink also highlights the signal path that creates the type conflict.

The components of the Symphony Model Compiler blockset always align at the binary point. The Simulink simulation models and hardware RTL generated from this automatically take care of any required scaling to make this happen.

Data Type Casting: Setting the Output Data Type

Data type casting is the last operation that the software performs. Data type casting occurs when you set a block output data type to cast the output to a data type that is different from the input data type, or when the output data type differs from the input data type after an operation. The software performs data type casting last; the operations follow this order:

- Operation
- Resizing
- Casting

Matrix Data Types

Several DSP algorithm applications use two-dimensional processing, where the natural specification is represented in 2-D matrix arithmetic. Some examples of this are wireless MIMO, multi-channel filtering, signal transforms, adaptive filtering, and video and image processing. Support for the matrix data type raises the level of abstraction in capturing such designs. In addition, matrix notation support helps to compactly represent any scenario where data can be logically arranged in two dimensions.

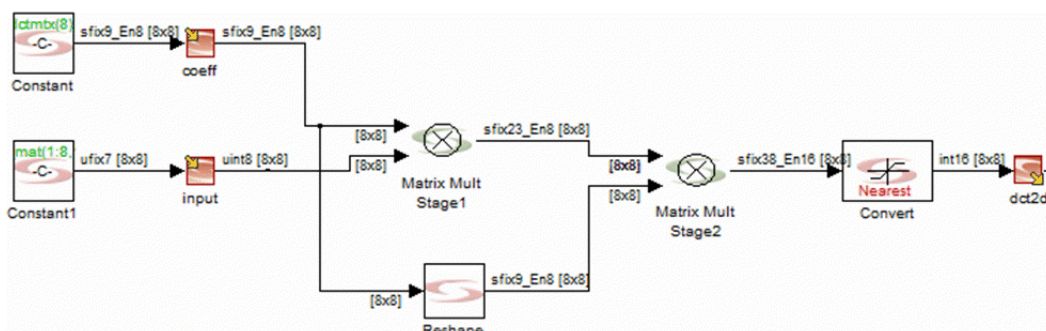
The Symphony Model Compiler tool includes matrix support for the most widely used DSP blocks, like Matrix Mult, Port In, Port Out, Convert, Recast, Reshape, and so on. For a full list, refer to the table in [Appendix A, Blockset Summary](#).

Example: 2-D DCT Using Matrix Data Types

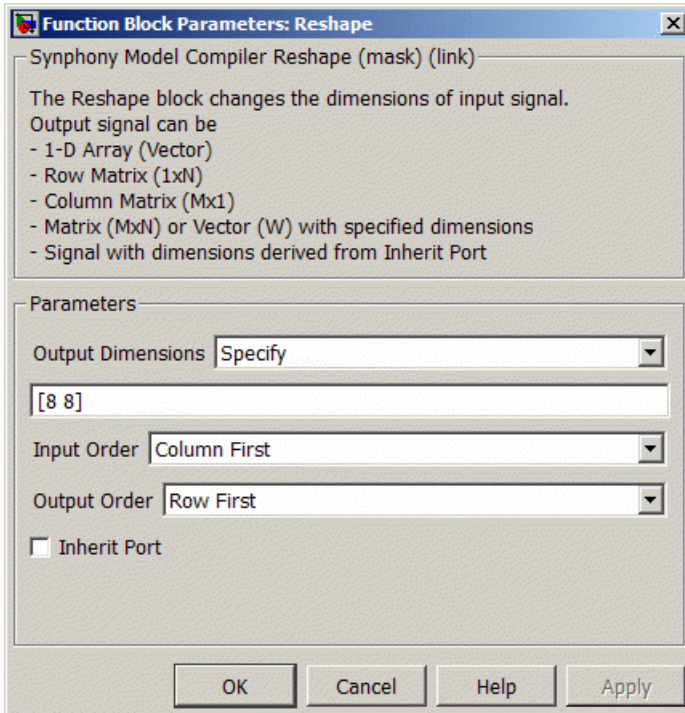
The following demo design shows how you can use matrix notation to capture a design at a high level of abstraction. To open the demo design, go to the MATLAB console and type this command:

```
shlsdemo('dct2d_usingmmult')
```

The example shows the 2-D direct cosine transform (DCT) of an 8x8 block is a two-stage calculation in the SMC tool. The first stage does a matrix multiplication of the DCT coefficient matrix (Constant) and the input matrix (Constant1). The second stage does a matrix multiplication of the output of the first stage and the transposed DCT coefficient matrix to obtain the final result. A Matrix Mult block is used to implement each stage.



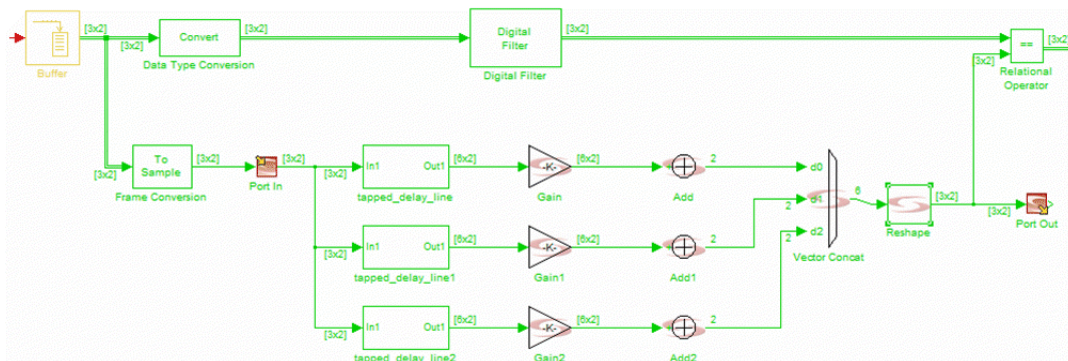
The design also shows the Reshape block being used to get the transpose of a matrix. The transpose is achieved by setting the Input Order and Output Order block parameters for Reshape, as shown below:



Example: Multichannel Filter Using Matrix Data Types

The following example shows how to use the matrix data type to capture a design without frame-based processing. It implements a direct form FIR filter, with 6 taps for a multichannel frame-based input signal. The frame size is 3 and there are 2 channels. Each channel is filtered by same filter with coefficients from the FDA Tool. You can open the demo for this design by going to the MATLAB console and typing the following command:

```
type shlsdemo('framebasedfiltering')
```



The tool first converts the frame-based signal into a sample-based signal for RTL implementation. The sample-based signal is a 3x2 matrix, where each column of the matrix corresponds to a channel. So in each simulation step for each channel, there are 3 input samples that must produce 3 output samples. For this purpose, the design is captured as three parallel processing units that calculate the outputs for both channels. The tapped delay lines maintain the past input samples. Additionally, you can fold the pattern (Gain followed by Add) for area savings.

CORDIC Algorithms

CORDIC is an acronym for COrdinate Rotation DIgital Computer. CORDIC algorithms are a set of shift-add algorithms for rotating vectors in a plane. These algorithms were originally developed to digitally solve real-time navigation problems, but vector rotation is useful in many DSP applications as well. CORDIC algorithms offer a hardware-efficient alternative to traditional DSP design.

Some functions can be computed through vector rotations. In addition to trivial applications like polar to rectangular conversion and rectangular to polar conversion, vector rotations can be used for more sophisticated trigonometric and other mathematical functions.

The base trigonometric algorithm CORDIC was first described by Volder¹. Andraka also has a good overview². The extension towards hyperbolic algorithms was first introduced by Walther³, with a comprehensive overview by Dawid/Meyer⁴. For an overview of the Symphony Model Compiler implementation, see [Circular, Linear, and Hyperbolic Coordinate Systems, on page 117](#). Symphony Model Compiler provides a unified CORDIC algorithm, combining the three coordinate systems in a single block.

The following sections explain CORDIC terms and underlying algorithms, and describe applications of unified CORDIC algorithms.

- [CORDIC Definitions, on page 702](#)
- [Unified CORDIC Applications, on page 711](#)

1. Jack Volder. Binary computation algorithms for coordinate rotation and function generation. Convair Report IAR-1 148 Aeroelectronics Group. June 1956.
2. Ray Andraka. A survey of CORDIC algorithms for FPGA. 1998. <http://www.andraka.com/files/crdcsrvy.pdf>
3. John S. Walther. A unified algorithm for elementary functions/. Spring Joint Computer Conf. 1971 (pp.379-385).
4. Herbert Dawid, Heinrich Meyr. CORDIC Algorithms and Architectures. Digital Signal Processing for Multimedia Systems, 1999, pp.623-655. http://www.eecs.berkeley.edu/~newton/Courses/EE290sp99/lectures/ee290aSp99_1/cordic_chap24.pdf

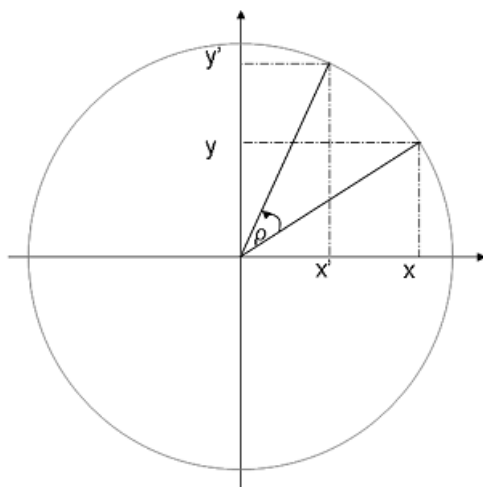
CORDIC Definitions

This section describes the following, listed here in alphabetical order:

- [CORDIC Algorithm, on page 705](#)
- [CORDIC Angle, on page 703](#)
- [CORDIC Gain, on page 708](#)
- [CORDIC Mode, on page 709](#)
- [CORDIC Range, on page 707](#)
- [CORDIC Rotation, on page 704](#)
- [CORDIC Rotator, on page 706](#)
- [Rotation Transform, on page 703](#)
- [Unified CORDIC, on page 710](#)
- [Vector Rotation, on page 702](#)

Vector Rotation

Vector rotation takes a vector (x,y) and rotates it over an angle ρ to a new position (x',y') , while maintaining the magnitude.



Rotation Transform

A vector rotation can be mathematically expressed with the basic rotation transform:

$$x' = x \cdot \cos \rho - y \cdot \sin \rho$$

$$y' = x \cdot \sin \rho + y \cdot \cos \rho$$

When you rearrange this to matrix notation and factor out $\cos \rho$, you get the following:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos \rho \begin{bmatrix} 1 & -\tan \rho \\ \tan \rho & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

CORDIC Angle

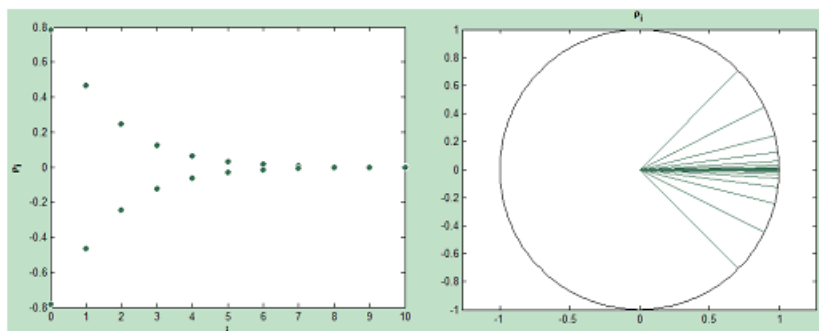
Restrict the rotation angle ρ to satisfy (integer $i \geq 0$):

$$\tan \rho_i = \pm \frac{1}{2^i} = \delta_i 2^{-i}$$

$$\cos \rho_i = \cos(-\rho_i) = \frac{1}{\sqrt{1 + 2^{-2i}}} = K_i$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = K_i \begin{bmatrix} 1 & -\delta_i 2^{-i} \\ \delta_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

This corresponds to a discrete set of rotation angles within the $[-\pi/4, \pi/4]$ range. The positive angles correspond to $\delta_i = 1$, and the negative angles correspond to $\delta_i = -1$.

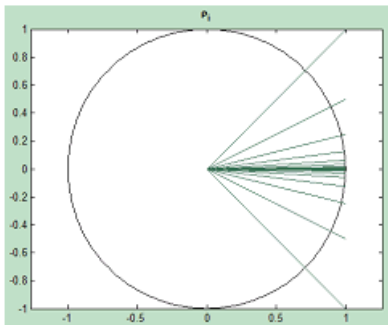


Index	$\tan \rho_i$	ρ_i (rad)	ρ_i (deg)
0	1	.785 ($\pi/4$)	45.0
1	1/2	.464	26.6
2	1/4	.245	14.0
3	1/8	.124	7.13
4	1/16	.062	3.58
5	1/32	.031	1.79
6	1/64	.016	.90
7	1/128	.008	.45

CORDIC Rotation

If you drop the K_i factor from the initial rotation equation, the angle is still pursued, but the magnitude changes by a factor $1/K_i$. This is also known as pseudo-rotation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & -\delta_i 2^{-i} \\ \delta_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



Index	K_i	$1/K_i$
0	.7071	1.4142
1	.8944	1.1180
2	.9701	1.0308
3	.9981	1.0078
4	.9995	1.0020
5	.9999	1.0005
6	1.0000	1.0001
7	1.0000	1.0000

The calculations for a pseudo-rotation by a CORDIC angle are reduced to a shift (division by 2) and add operation.

CORDIC Algorithm

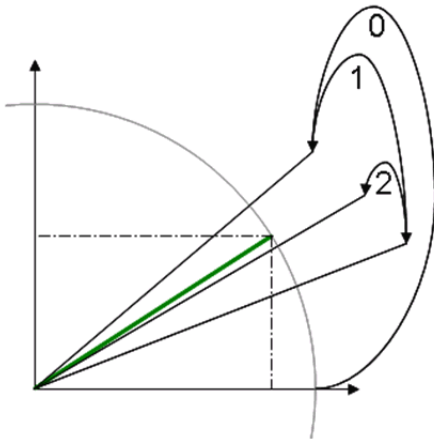
You can obtain an arbitrary rotation angle through a sequence of CORDIC rotations of successively declining CORDIC angles, with variable rotation direction. The decision to rotate clockwise $\delta_i = 1$ or counter clockwise $\delta_i = -1$ decomposes the desired rotation angle ρ into microrotations ρ_i :

$$\rho = \sum_{i=0}^{n-1} \delta_i \cdot \rho_i \quad \left| \begin{array}{l} z_0 = 0 \\ z_{i+1} = z_i - \delta_i \cdot \rho_i \\ \rho = z_n \end{array} \right.$$

Using the iterative form of this rotation decomposition, there are associated coordinate transformations $(x_0, y_0) \rightarrow (x_n, y_n)$, which results in the following iterative formula:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -\delta_i 2^{-i} \\ \delta_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

$$z_{i+1} = z_i - \delta_i \rho_i$$



CORDIC Rotator

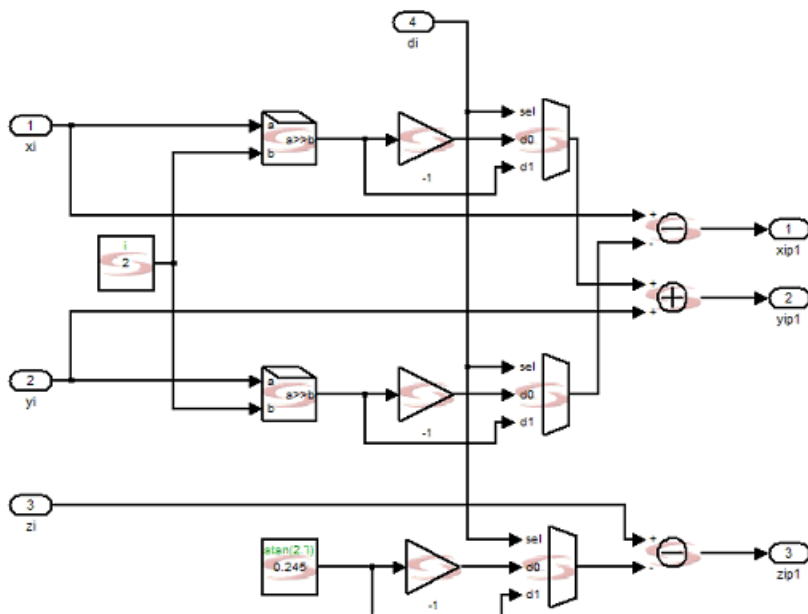
The CORDIC algorithm is an iterative manipulation of 3 simple equations, calculating a vector (x_i, y_i) and an angle accumulator z_i . A device capable of doing so is a CORDIC rotator or CORDIC processor.

$$x_{i+1} = x_i - y_i \cdot \delta_i \cdot 2^{-i}$$

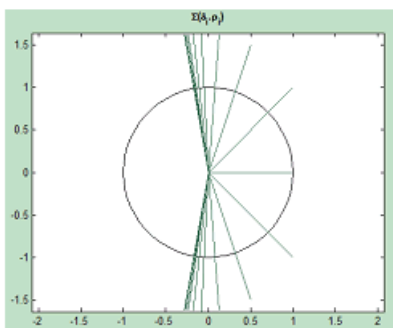
$$y_{i+1} = y_i - x_i \cdot \delta_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - \delta_i \cdot \tan^{-1} \cdot 2^{-i}$$

Note that the CORDIC algorithm or sequence of CORDIC rotations is uniquely defined by the sequence δ_i , which is called the decision vector.



The range of the CORDIC rotations is determined by continuously rotating in the same direction. This limits CORDIC rotation to just a little beyond the first ($\delta_i == 1$) and last ($\delta_i == -1$) quadrant:



© 2013 Synopsys, Inc.
707

coordinates outside this boundary to equivalent coordinates in the supported range. What constitutes equivalency depends on the application or function calculated with the CORDIC algorithm.

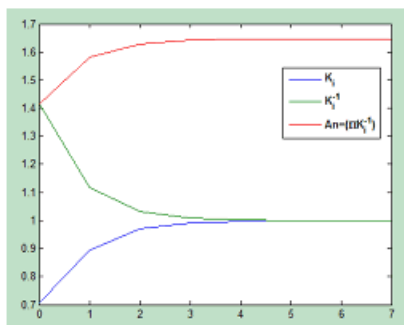
CORDIC Gain

Typically, the sequence of CORDIC rotations is accomplished with CORDIC pseudo-rotations. This means that the overall rotation provides a gain, depending on the number of desired rotations:

$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

Index	A_n
0	1.4142
1	1.5811
2	1.6298
3	1.6425
4	1.6457
5	1.6465
6	1.6467
7	1.6467

The local factor K_i and therefore the local pseudo-rotation gain $1/K_i$, quickly goes to 1 with increasing index; the cumulative gain converges to approximately 1.647.



CORDIC Mode

The CORDIC rotator can be operated in two different modes, rotation or vectoring.

Rotation

This mode rotates the input vector by the specified angle ρ , calculating the resulting coordinates x_n and y_n . You can do this with the CORDIC Rotator block by specifying $z_0=\rho$, and drive the decision vector to make the angle accumulator 0.

$$z_0 = \rho$$

$$\delta_i = -1(z_i < 0)$$

$$\delta_i = 1(z_i \geq 0)$$

When the CORDIC algorithm reaches $z_n=0$, the result is

$$x_n = A_n(x_0 \cdot \cos \rho - y_0 \cdot \sin \rho)$$

$$y_n = -A_n(y_0 \cdot \cos \rho + x_0 \cdot \sin \rho)$$

$$z_n = 0$$

Vectoring

This mode rotates the input vector to the x-axis and calculates the required angle ρ . You can do this with the CORDIC Rotator block by driving the decision vector to make the y_n coordinate 0.

$$y_0$$

$$\delta_i = 1(y_i > 0)$$

$$\delta_i = -1(y_i \geq 0)$$

When the CORDIC algorithm reaches $y_n=0$, the result is

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$

$$y_n = 0$$

$$z_n = z_0 + \tan^{-1} \frac{y_0}{x_0}$$

Unified CORDIC

The iteration equations described above are for trigonometric or circular systems. Similar equations can be derived for hyperbolic systems and linear systems; they can be unified ³ with an m-factor, defined as shown in the following table. For a description of the different systems, see [Circular, Linear, and Hyperbolic Coordinate Systems](#), on page 117.

System	m Value
Circular	1
Hyperbolic	-1
Linear	0

$$x_{i+1} = x_i - m \cdot y_i \cdot \delta_i \cdot 2^{-i}$$

$$y_{i+1} = y_i + x_i \cdot \delta_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - \delta_i \cdot \varepsilon_i$$

$$\varepsilon_i = \tan^{-1} 2^{-i} \quad m = 1$$

$$\varepsilon_i = \tanh^{-1} 2^{-i} \quad m = -1$$

The iteration index starts at 0 for circular systems, and 1 for linear and hyperbolic systems. The following table gives you an overview for the different operation modes. Note that the hyperbolic algorithm only converges if certain iterations are repeated ($i=4,13,40,121,k,3k+1,\dots$). The Symphony CORDIC Rotator block does this automatically. The gain of .8282 takes this into account.

	Rotation	Vectoring
Circular	$x_n = A_n (x_0 \cdot \cos \rho - y_0 \cdot \sin \rho)$ $y_n = A_n (y_0 \cdot \cos \rho + x_0 \cdot \sin \rho)$ $z_n = 0$ $A_n = \prod_n \sqrt{1 + 2^{-2i}} \approx 1.647$	$x_n = A_n \sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = z_0 + \tan^{-1} \frac{y_0}{x_0}$ $A_n = \prod_n \sqrt{1 + 2^{-2i}} \approx 1.647$
Linear	$x_n = x_0$ $y_n = y_0 + x_0 \cdot z_0$ $z_n = 0$	$x_n = x_0$ $y_n = 0$ $z_n = z_0 + \frac{y_0}{x_0}$
Hyperbolic	$x_n = A_n (x_0 \cdot \cosh \rho - y_0 \cdot \sinh \rho)$ $y_n = A_n (y_0 \cdot \cosh \rho + x_0 \cdot \sinh \rho)$ $z_n = 0$ $A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx .8282$	$x_n = A_n \sqrt{x_0^2 - y_0^2}$ $y_n = 0$ $z_n = z_0 + \tanh^{-1} \frac{y_0}{x_0}$ $A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx .8282$

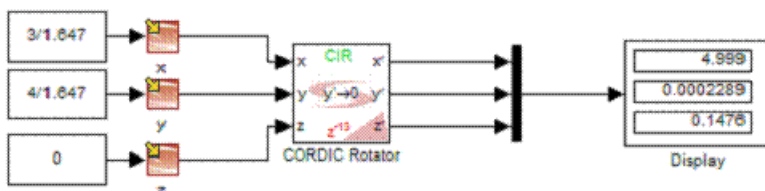
Unified CORDIC Applications

The Unified CORDIC engine can be used to calculate a variety of mathematical functions. The following examples show how you can manipulate the engine to calculate the desired function. Note that the convergence criteria are not discussed.

Rectangular-Polar Conversion

Use the CORDIC engine in vectoring mode, in a circular coordinate system. Apply the vector coordinates to x and y. Make z = 0.

$$\left. \begin{aligned} x &= \frac{x_0}{A_n} \\ y &= \frac{y_0}{A_n} \\ z &= 0 \end{aligned} \right| \begin{aligned} x' &= M \\ y' &= 0 \\ z' &= \rho \end{aligned}$$



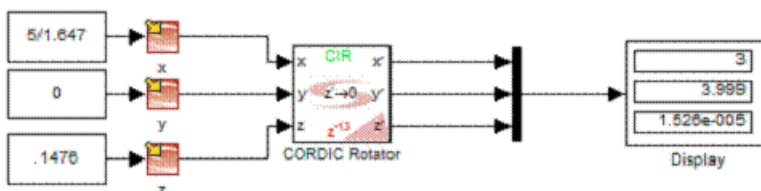
$$\sqrt{3^2 + 4^2} = 5$$

$$\tan^{-1} \frac{4}{3} / 2\pi = .1476$$

Polar-Rectangular Conversion

Use the CORDIC engine in rotation mode, in a circular coordinate system. Apply the magnitude to x, and make y=0 and z=p:

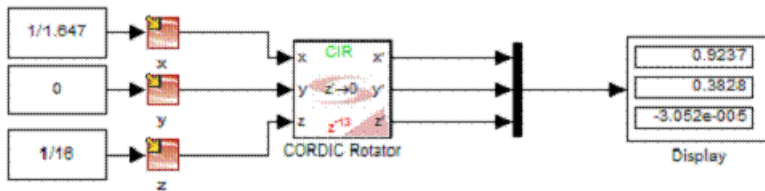
$$\left. \begin{aligned} x &= \frac{M}{A_n} \\ y &= 0 \\ z &= \frac{\rho}{2\pi} \end{aligned} \right| \begin{aligned} x' &= M \cos \rho \\ y' &= M \sin \rho \\ z' &= 0 \end{aligned}$$



Cosine and Sine

Use the CORDIC engine in rotation mode, in a circular coordinate system. Apply the unit vector to the X-axis and make $z=p$:

$$\left. \begin{aligned} x &= \frac{1}{A_n} \\ y &= 0 \\ z &= \frac{\rho}{2\pi} \end{aligned} \right| \begin{aligned} x' &= \cos \rho \\ y' &= \sin \rho \\ z' &= 0 \end{aligned}$$



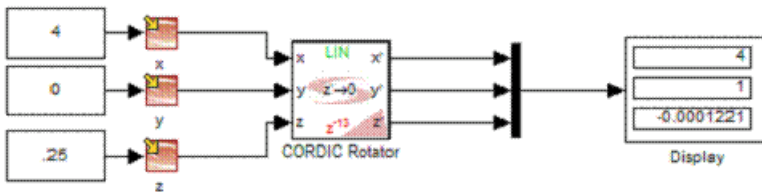
$$\cos \frac{\pi}{8} = .9239$$

$$\sin \frac{\pi}{8} = .3827$$

Multiplication

Use the CORDIC engine in rotation mode, in a linear coordinate system. Apply the first operand to x and the second operand to z . Make $z=0$.

$$\left. \begin{aligned} x &= a \\ y &= 0 \\ z &= b \end{aligned} \right| \begin{aligned} x' &= a \\ y' &= a \cdot b \\ z' &= 0 \end{aligned}$$

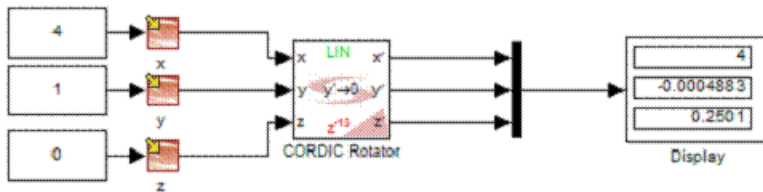


This multiplication is similar to a regular shift-add implementation, but there are more efficient architectures available.

Division

Use the CORDIC engine in rotation mode, in a linear coordinate system. Apply the vector coordinates to (a,b) and make z=0.

$$\begin{array}{l|l} x = a & x' = a \\ y = b & y' = 0 \\ z = 0 & z' = \frac{b}{a} \end{array}$$



Square Root

$$\sqrt{(a+b)^2 - (a-b)^2} = \sqrt{4ab}$$

Given this formula, if $b=1/4$, then the formula corresponds to \sqrt{a} . Use the CORDIC engine in vectoring mode, in a hyperbolic coordinate system. Apply the vector coordinates $(a + .25, a - .25)$ and make $z=0$.

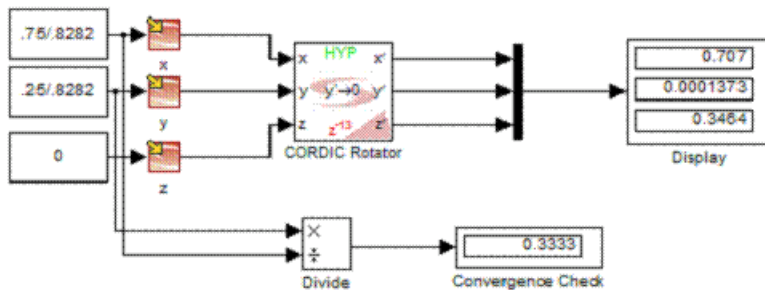
$$\begin{array}{l|l} x = \frac{(a + .25)}{A_n} & x' = \sqrt{a} \\ y = \frac{(a - .25)}{A_n} & y' = 0 \\ z = 0 & z' = \rho \end{array}$$

Note that the hyperbolic convergence puts a limitation on the input:

$$\left| \frac{y}{x} \right| < .81$$

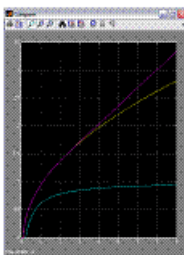
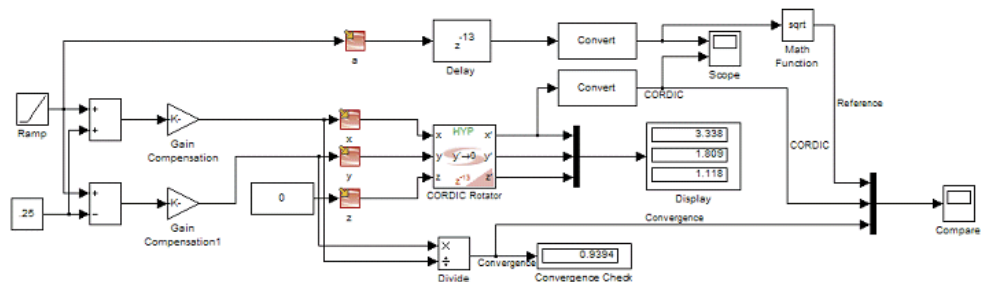
$$\frac{(a-1)/4}{(a+1)/4} < .81$$

$$a \leq 2.38$$



$$\sqrt{5} = .7071$$

The following more elaborate design shows that the CORDIC implementation of the square root diverges for inputs outside the calculated range [0,1.75].



Convergence Transformations

The nature of the CORDIC algorithm poses some limitations on the inputs to assure convergence. This means that for some applications, the arguments have to be pre-processed to fall in the supported range, and post-processed to derive the actual function value for the original input.

Quadrant Folding

To calculate the sine function of an angle ρ , the circular CORDIC methods only converge in vectoring mode if this angle is within the convergence domain. Practically, this is done by

- Preprocessing: folding the angle to the supported range $[-\pi/2, \pi/2]$
- Postprocessing: adjusting the sign of the output

$\sin \rho = \sin(\pi - \rho) \quad \frac{\pi}{2} < \rho \leq \pi$ $z_0 = \pi - \rho$ $x' = x_n$	$0 < \rho \leq \frac{\pi}{2}$ $z_0 = \rho$ $x' = x_n$
$\sin \rho = \sin(\rho - \pi) \quad \pi < \rho \leq \frac{3\pi}{2}$ $z_0 = \rho - \pi$ $x' = x_n$	$\sin \rho = \sin(\rho - 2\pi) \quad \frac{3\pi}{2} < \rho \leq 2\pi$ $z_0 = \rho - 2\pi$ $x' = x_n$

Shifting

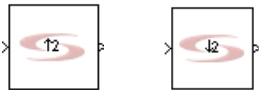
Hyperbolic CORDIC methods are very sensitive to the input range. Depending on the function, sometimes shift helps keep the inputs in this range:

- Preprocessing: shift the data by $2n$ to the desired range
- Postprocessing: shift the output by n to get the result

$$\sqrt{a} = 2_n \sqrt{\frac{a}{2^{2n}}}$$

Multi-Rate Design

The sample rate of a signal is determined by the sample rate propagated from input signals. Every block in the Symphony Model Compiler blockset propagates the sample rate of the driving signal to the output signals. You can change the sample rate of a signal with the Upsample or Downsample blocks (Symphony Model Compiler Signal Operations library). You can also use the Upsample and Downsample blocks in a multirate design that has inputs with different sample rates.



This section discusses the following:

- [Sample Rate Terminology, on page 717](#)
- [Clock Generation and Clock Reset, on page 721](#)
- [Polyphase Filtering, on page 724](#)

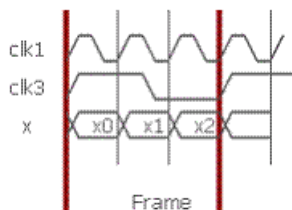
Sample Rate Terminology

The following table defines various terms used in this discussion on sample rates.

Term	Description
Decimation	Decrease of sample rate by throwing samples away: <pre>function y = down(x, M, offset) for k = 1:floor((length(x) - offset)/M) y(k) = x(M*k + offset); end;</pre>
Delay	Register used to move a signal to the next slot in a frame. In the context of a rate change block, the delay is applied at the clock of the highest rate, to manage the offset

Term	Description
------	-------------

Frame	Collection of samples of a higher rate signal, contained within the boundary of the rising edges of the rising edge of the lower rate clock in a a rate changer:
-------	--



Interpolation	Increase of sample rate by inserting extra samples:
---------------	---

```
function y = up(x, L, offset)
for o= 1:offset
    y(o)=0;
end

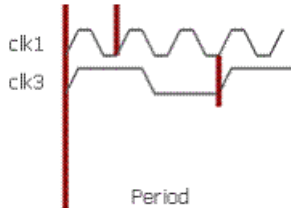
for k = 1:length(x)
    y((k-1)*L+1+offset) = x(k);
    for ll=2:L-1
        y((k-1)*L+ll+offset) = 0;
    end
end;
```

Latency	Difference between the output frame number and input frame number of a function:
---------	--

- For any offset in an upsample rate change, the latency is 0.
- For an offset of 0 in a downsample rate change, the latency is 0.
- For any other offset in a downsample rate change, the latency is 1.

Term	Description
------	-------------

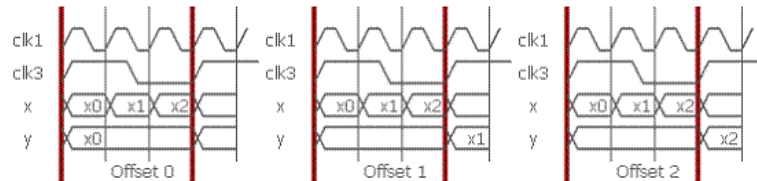
Multirate Design	Design that uses multiple sample rates
------------------	--



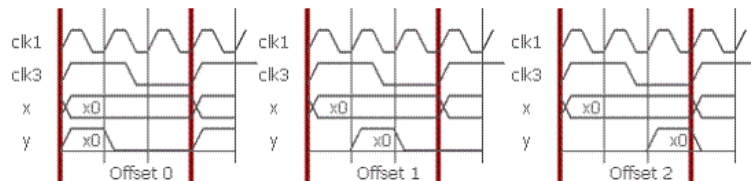
Simulink can introduce clocks with different periods in these ways:

- Source: sample period definition
- Data type conversion: samples a continuous signal
- Upsample/Downsample/Resample blocks

Offset (downsample rate change)	Slot required to populate the frame at the lower rate.
---------------------------------	--

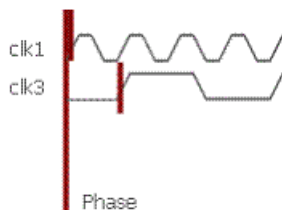


Offset (upsample rate change)	Slot to be populated in the frame at the higher rate.
-------------------------------	---



Term	Description
------	-------------

Phase	Delay of the rising edge of a clock, relative to time zero
-------	--



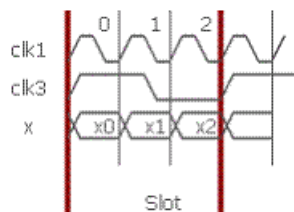
Simulink supports sample phase for single rate systems, but does not support this when applying rate changes. The Symphony Model Compiler tool does not support sample phase for any signal; it assumes all clocks have phase zero.

Resampling	Combination of decimation and interpolation to change the sample rate with a fractional value.
------------	--

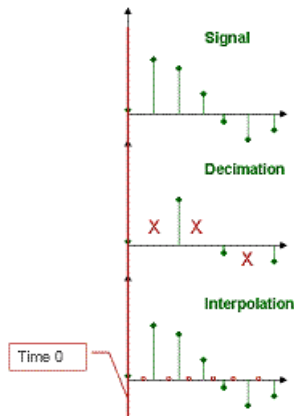
Sample Period	Period of the sample clock
---------------	----------------------------

Sample Rate	Frequency of the sample clock
-------------	-------------------------------

Slot	Different boundaries defined by the rising edges of the higher rate clock within a frame defined by a rate changer
------	--



Term	Description
Time 0	Reference provided by the first sample in the digital domain. Without offset, this first sample is maintained through either decimation or interpolation.

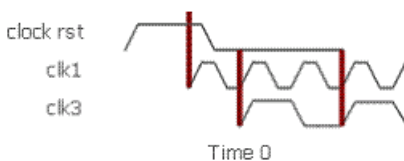


Clock Generation and Clock Reset

The reset (main reset) provides, by definition, a synchronization point for the complete design. The Symphony Model Compiler tool needs a Time 0 point. The relative location of this point to the clock reset is important, because of the following:

- It determines the functionality of simple single-rate functions
- It determines the shared samples of multi-rate functions.

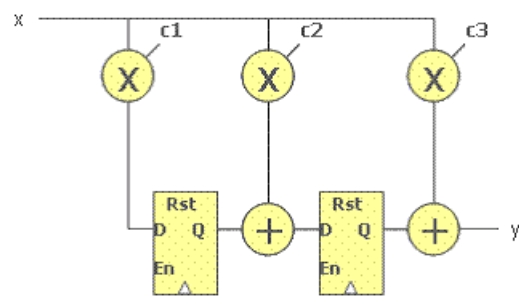
Most clock reset circuitry aligns the rising edges of derived clocks with the rising edge of the highest clock right after clock reset (or generates a corresponding enable at that location).



Alternatively, you can specify clocks and resets using a clock reset module. For a detailed description of the module and its limitations, see [Clock and Reset Management](#), on page 686.

Single-Rate Analysis

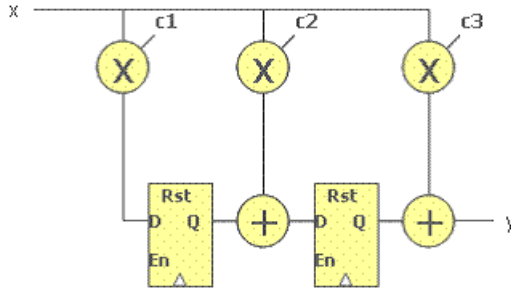
For a single-rate system, the main reset pulse just indicates the position of the first active edge of the clock. To look at the impact of a single-rate function, consider a 3-tap FIR; The input $x[0]$ is provided at the first rising edge of a clock after a main reset-pulse. The desired (DSP mathematics) functionality is captured in the following table:



Time

-1	U	$C1*U$	0	$C2*U$	0	$C3*U$
0	$X[0]$	$C1*X[0]$	0	$C2*X[0]$	0	$C3*X[0]$
1	$X[1]$	$C1*X[1]$	$C1*X[0]$	$C2*X[1]+C1*X[0]$	$C2*X[0]$	$C3*X[1]+C2*X[0]$
2	$X[2]$	$C1*X[2]$	$C1*X[1]$	$C2*X[2]+C1*X[1]$	$C2*X[1]+C1*X[0]$	$C3*X[2]+C2*X[1]+C1*X[0]$
3	$X[3]$	$C1*X[3]$	$C1*X[2]$	$C2*X[3]+C1*X[2]$	$C2*X[2]+C1*X[1]$	$C3*X[3]+C2*X[2]+C1*X[1]$

The desired behavior requires that the DFF elements of the FIR not be governed by the main reset. There would be a problem at time zero; the DFFs would also update and destroy the initialization value. This is illustrated in the next table:



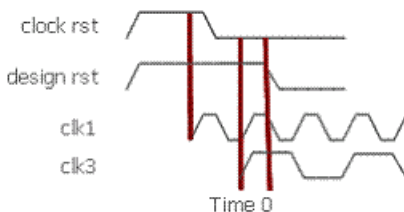
Time

-1	U	$C1*U$	0	$C2*U$	0	$C3*U$
0	$X[0]$	$C1*X[0]$	$C1*U$	$C2*X[0] + C1*U$	$C2*U$	$C3*X[0] + C2*U$

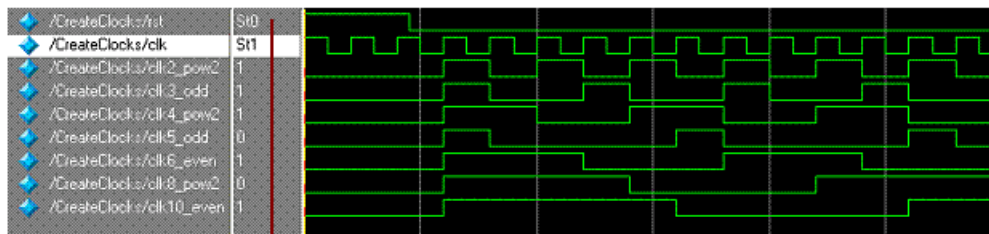
For single rate designs, it is essential that the design reset makes sure that the registers do no update at time zero. This means that the design reset needs to be different from the main reset. The design reset needs to be delayed by one clock cycle.

Multi-Rate Analysis

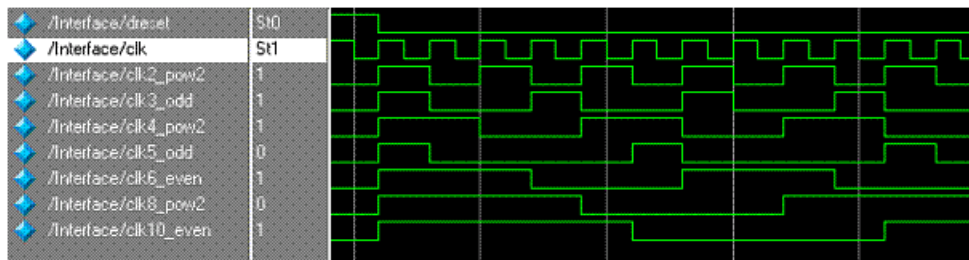
For a multi-rate system, assume that the clock reset circuitry generates the rising edges of all clocks aligned with the first rising edge of the fastest clock right after clock reset (Time 0). For any clock domain, the first rising edge of the respective domain can not trigger an update of the design registers (see [Single-Rate Analysis, on page 722](#)). This means that the design reset can be created in the same way as for single-rate designs:



However, the clock counters must be initialized so that they create a rising edge after the main reset. This can only happen if the counters are controlled by the main reset. The following waveforms show the aligned edges after aligning them with reset, and defining time zero and the beginning of the first frame:



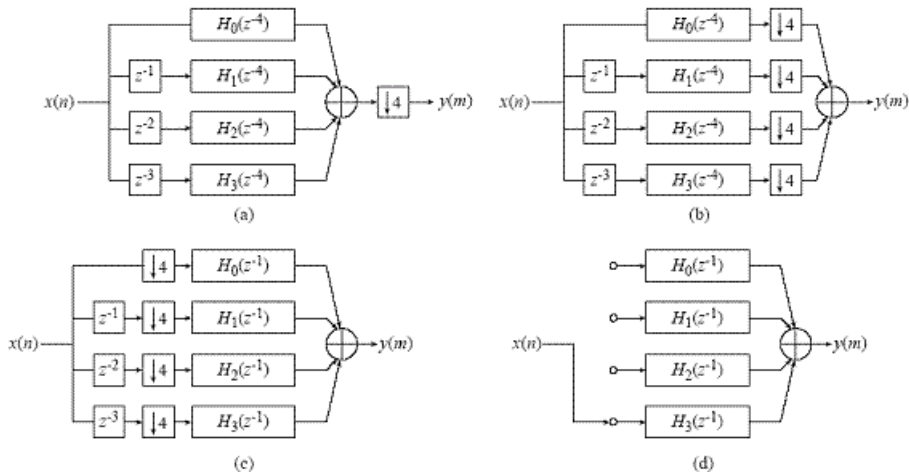
The combination of a design reset and the clocks shows a typical input for a Symphony system:



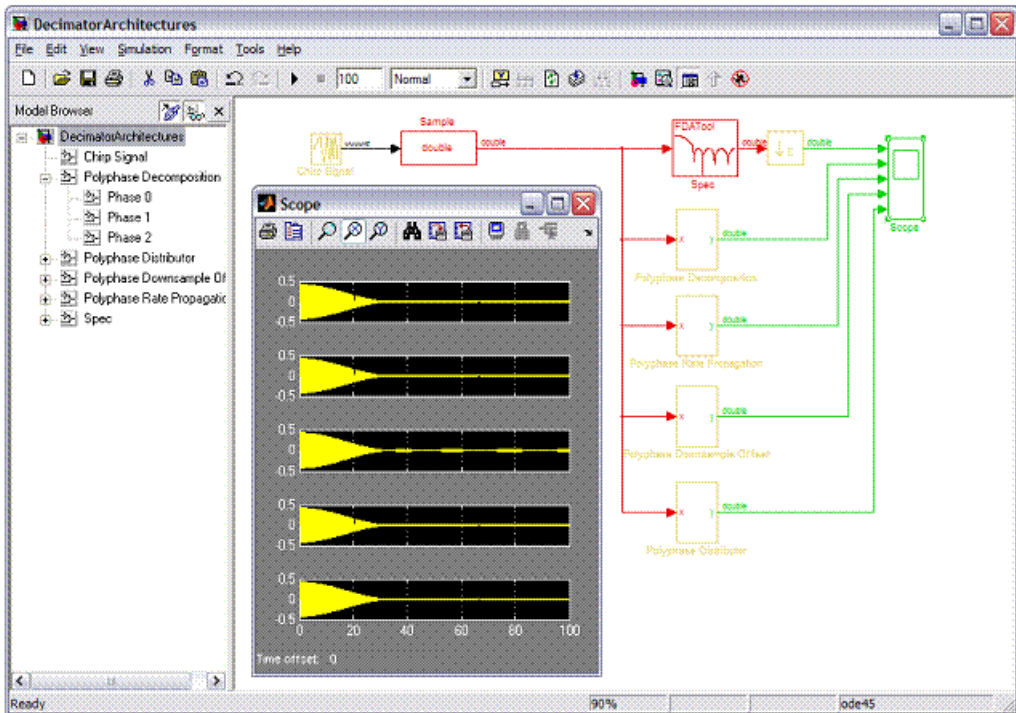
Polyphase Filtering

The use of upsample and downsample rate changers is determined by the requirements to do polyphase filtering. This section describes downsampling.

In a typical low-pass/downsample rate conversion, the different phases correspond to the different offset selections possible: the zero offset requires zero latency, while the other offset is aligned with the next rising edge of the frame, introducing a latency of one:



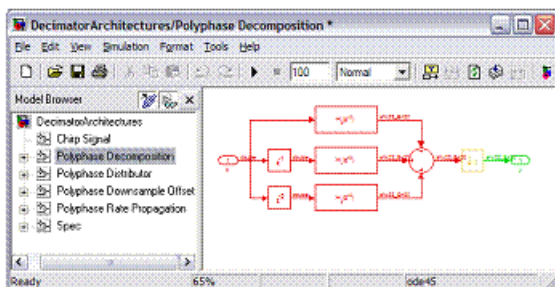
You can validate this in Simulink, using Simulink primitives.



The Spec block provides a fully specified FIR filter, and is followed by a Downsample block. It is the reference for the decimation functionality. The functionality goes through these transformational phases:

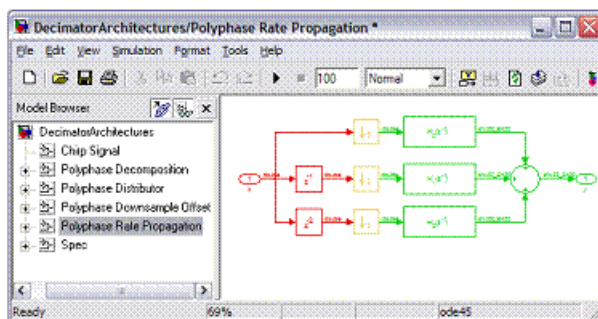
- Polyphase Decomposition

The first transformation re-organizes the FIR transfer function. It creates FIR structures with z^{-3} delays.



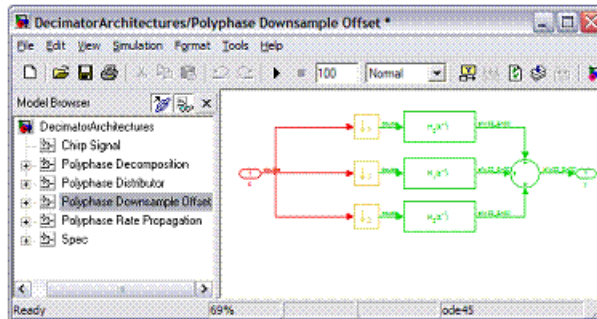
- Polyphase Rate Propagation

The second transformation moves the Downsample block across functions and delays. This turns the z^{-3} delays into z^{-1} delays.



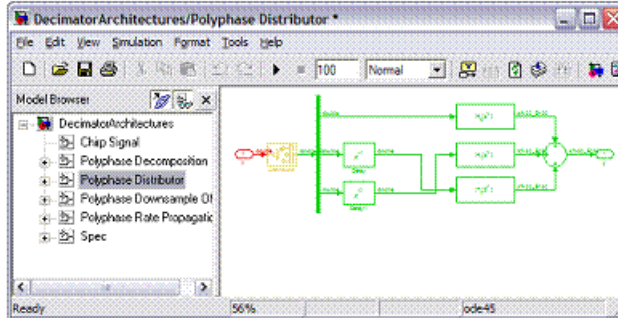
- Polyphase Downsample Offset

The third transformation moves the delays into the downsample blocks as an offset. This lets you confirm the functionality of offset in the Downsample block.



- Polyphase Downsample Offset

The final transformation replaces the downsample blocks by a distributor (). This allows to confirm the functionality of a distributor: the order at the outputs 2:N needs to be reversed to hook up to the polyphase branches. In a distributor, all outputs have a delay, whereas in a polyphase decimator there is no delay for phase 0. To maintain functionality, the 2:N outputs need to be delayed to sync up with the requirements for phase 0.



Hierarchy Preservation

If you define subsystems in the Simulink model, the Symphony tool retains these hierarchy levels, unless you have enabled folding. If you have enabled folding, the subsystems might be optimized. The following design has two subsystems defined:



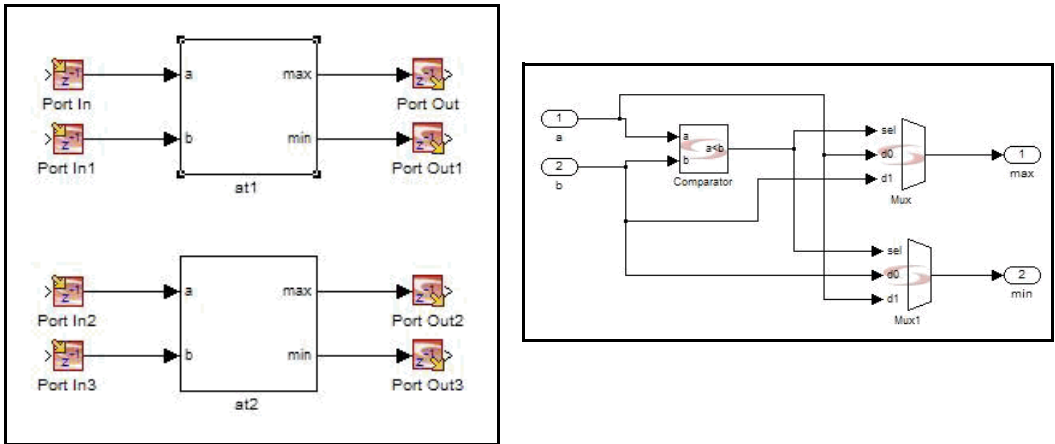
The tool maintains the hierarchy and generates a two-level design, with Subsystem1 and Subsystem2 instantiated in the top-level RTL. The log file reports the number of top level-subsystems found:

```
2 top level subsystems found:
  Subsystem1
  Subsystem2
```

Hierarchy preservation works together with subsystem consolidation (see [Subsystem Consolidation, on page 729](#)). If there are duplicates of a subsystem, only one module/entity is generated.

Subsystem Consolidation

The tool consolidates subsystems in model files if they have the same content and port order. Subsystems running at different rates are not consolidated. In the following example, subsystems at1 and at2 have the same content:



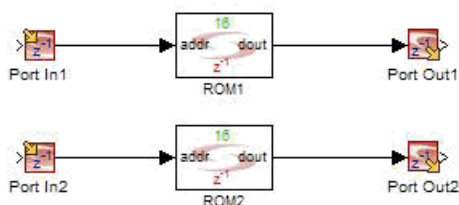
The tool consolidates them and generates only one module/entity (at1) and instantiates it for both subsystems instances:

```
.
instance1:at1
  port map (...);
.
.
instance2:at1
  port map (...);
.
.
```

Block Consolidation

For blocks that are not instantiated from the Symphony Model Compiler library, the tool consolidates the RTL descriptions of these blocks and implements only one module for blocks that have the same options and input/output bit widths. It does not consolidate blocks running at different rates.

The following design shows two ROM blocks that implement the same feature.



The tool consolidates the blocks and generates one module/ entity (ROM2), which it instantiates for both ROMs:

```
.
instance1:ROM1
  port map (...);
.
.
instance2:ROM1
  port map (...);
.
.
```

Constant Propagation

The Symphony Model Compiler tool propagates constant signals through the design and optimizes away blocks whose outputs are constant during runtime. The tool replaces the optimized blocks with proper constants in the generated RTL.

The log file notes which primitive blocks have been optimized away; it does not list custom blocks that have been optimized away. This is because custom blocks are first flattened before the constants are propagated.

Constant propagation improves performance with the advanced timing module (ATM) in the following ways:

- It enables the Synplify Pro synthesis tool to carry out constant optimizations, because the blocks with constant inputs in the design are queried with the same constants connected in the RTL. This improves the ATM estimations for individual blocks.
- It provides the retiming engine with a more realistic design topology, by removing blocks with constant inputs from the beginning.
- It reduces the ATM execution time by eliminating blocks with constant inputs from consideration. This means that the advanced timing module only queries blocks that appear in the generated RTL.

Constant propagation also offers the following advantages:

- It simplifies generated RTL for blocks like M-Control, and thus increases RTL readability.
- Optimized blocks have constant inputs, and this lets you detect design errors.

The log file notes primitive blocks that are optimized away. It does not list custom blocks that are optimized away. Constant propagation affects the primitive blocks listed in the following table.

Library	Block
Communications	Reed-Solomon Decoder Reed-Solomon Encoder
DSP Basics	Adder Gain
Filtering	FIR FIR Engine IIR
Math Functions	Abs Accumulator Binary Logic Comparator DivMod Inverter Mult Negate Pow Shifter Sqrt
Memories	Delay FIFO Permutation RAM Shift Register
Signal Operations	Concat Convert Demux Downsample Extract Mux Upsample
Sources	Counter
Transforms	FFT

RAMs

This section contains information about RAMs and RAM configurations in the Symphony Model Compiler software.

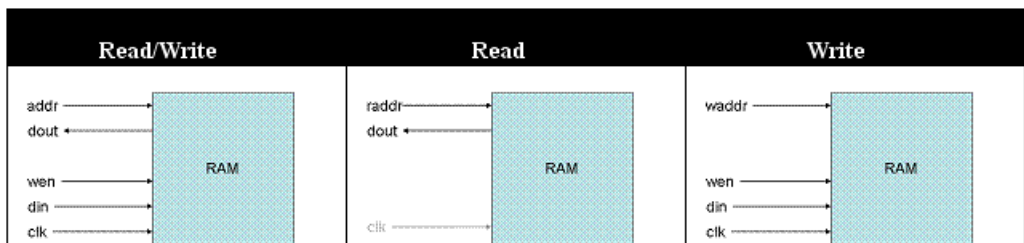
- [RAM Definitions, on page 733](#)
- [RAM Access Control, on page 736](#)
- [Port Use in Different RAM Configurations, on page 737](#)

RAM Definitions

This section describes commonly-used terms.

Port

Ports typically control access to a RAM, and combine different signals: clock (clk), write enable (we), address (addr), write data (din) and read data (dout). Depending on the actual signals present, the port can be a read/write port, read port, or write port.



Synchronous RAM/Asynchronous RAM

The write access to a RAM is always clocked (synchronous), so the overall operation of the RAM is determined by the read access. When clocked (either address line or data output), the RAM is called synchronous. For unclocked access, the RAM is called asynchronous.

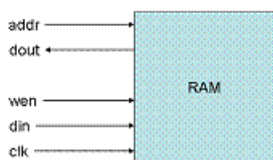
Write Mode

If a read and a write to the RAM core happen to the same location, there are different strategies:

WRITE-FIRST	The result of the read is the data written to that location.
READ-FIRST	The result of the read is the old data from that location.
NO-CHANGE	The output is held to the previous value.

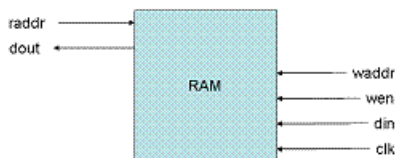
Single Port RAM

Read and write access to the RAM go through a single port with a shared address bus. A Single Port RAM does not allow for a simultaneous read and write to different locations in the storage array.



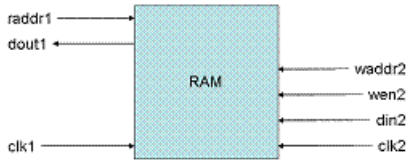
Two Port RAM

Read and write access to the RAM go through two ports on the RAM, one dedicated to read access and one dedicated to write access; the clocks for read and write are the same (or the read doesn't use a clock). The dedicated read and write address allow for a simultaneous read and write with different locations in the storage array.



(Simple) Dual Port RAM

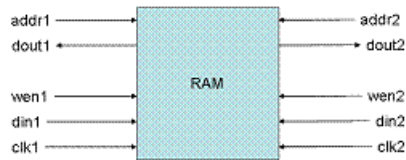
Read and write access to the RAM go through two ports on the RAM, one dedicated to read access and one dedicated to write access; the clocks for read and write are independent.



True Dual Port RAM

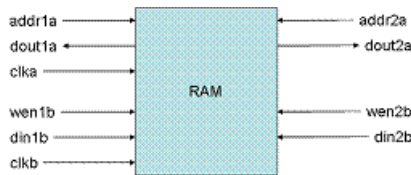
Read and write access to the RAM go through two ports on the RAM, where both ports can be used in either read or write mode; the clocks can be independent. This allows for the following permutations:

- Two simultaneous reads.
- Two simultaneous writes.
- Simultaneous read and write.



Quad Port RAM

Read and write access to the RAM go through four ports on the RAM, two dedicated to read access and two dedicated to write access; the clocks for read and write are independent.



RAM Access Control

There are potential clashes in the following situations, and you must specify how the RAM is to be accessed.

- Write operations to the same location
Control this through the write access control mode, as described in [Write Access Control, on page 736](#).
- Read and write operation to same location
Control this through the read access control mode, as described in [Read Access Control, on page 736](#).

Write Access Control

You can use the following options to control write operations to the same location:

No check	With this setting, the tool does not check simultaneous write operations to the same location. The output is undefined for clash situations.
Write prioritization	With this setting, simultaneous write operations to the same location are resolved through priority encoding (the lower port number has higher priority). For different read and write port clock frequencies, the behavior is undefined. If you select this option for a multirate case like this, the tool does not generate RTL for the block.

Read Access Control

If you have a read and a write to the same RAM location, there are different strategies. See the following table:

WRITE-FIRST	The result of the read is the data written to that location.
READ-FIRST	The result of the read is the old data from that location.

You can specify the following read access control options:

- Read-first
All read and write ports operate in read-first mode

- **Read-write port write first**
All the read ports operate in read-first mode. Read-write ports are switched to write-first mode, where the corresponding output is equal to the input data when write enable is high.
- **Cross-port write first**
All read and read-write ports operate cross-port write first. The output for a read or read-write port is set to the corresponding data input when other write port(s) perform writes to the same location. If more than one write is to be performed, write-prioritization is employed to select a read value. Therefore this option is only valid for write-prioritization write access. Note that this mode only considers the write ports at the same clock as a read port.

RAM Access Message

During Simulink simulation, if the address of a RAM block is read before it is written, you see the following warning message in the MATLAB command window:

```
Warning: block 'test/RAM': Contents of address 93 unknown!
```

Port Use in Different RAM Configurations

	Read Ports	Write Ports	Read/Write Ports
Single Port	0	0	1
Two Port (address has same sample rate)	1	1	0
(Simple) Dual Port (address has different sample rate)	1	1	0
True Dual Port (address can have different sample rates)	0	0	2
Quad Port read ports share sample rate; write ports share sample rate)	2	2	0

Bus Protocols

The SMC tool, through the Host Interface block, supports bus protocols that simplify the configuration of your design. For information about the Host Interface block and how to use it, see [SMC Host Interface, on page 326](#) and [Synthesizing with a Host Interface Block, on page 678](#), respectively.

The bus protocols are described here:

- [AXI4-Lite Protocol, on page 738](#)
- [APB Protocol, on page 743](#)
- [AVLON-MM Protocol, on page 745](#)
- [Generic Interface Protocol, on page 748](#)

AXI4-Lite Protocol

AMBA AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the ARM AMBA interface specification. AXI4-Lite is a simplified version of the AXI4 protocol and is suitable for simpler control register-style interfaces that do not require the full functionality of the AXI4. The SMC Host Interface block implements the AXI4-Lite protocol specification as described in the revision D of the *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite* document from ARM.

The AXI4-Lite is a five-channel interface. Each channel is an independent parallel connection between the source and the slave. Each channel offers a two-way flow control, using the valid-ready handshake signals. The corresponding interface signals for each channel are shown in the table below, which is extracted from the protocol specification:

Channels					
Global	Write Address	Write Data	Write Response	Read Add	Read Data
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	-AWADDR	WDATA	BRESP	ARADDR	RDATA
-	-AWPROT	WSTRB	-	ARPROT	RRESP

The Host Interface block does not implement the functionality of the signal defined by AWPROT and ARPROT that define the protection type of the transaction.

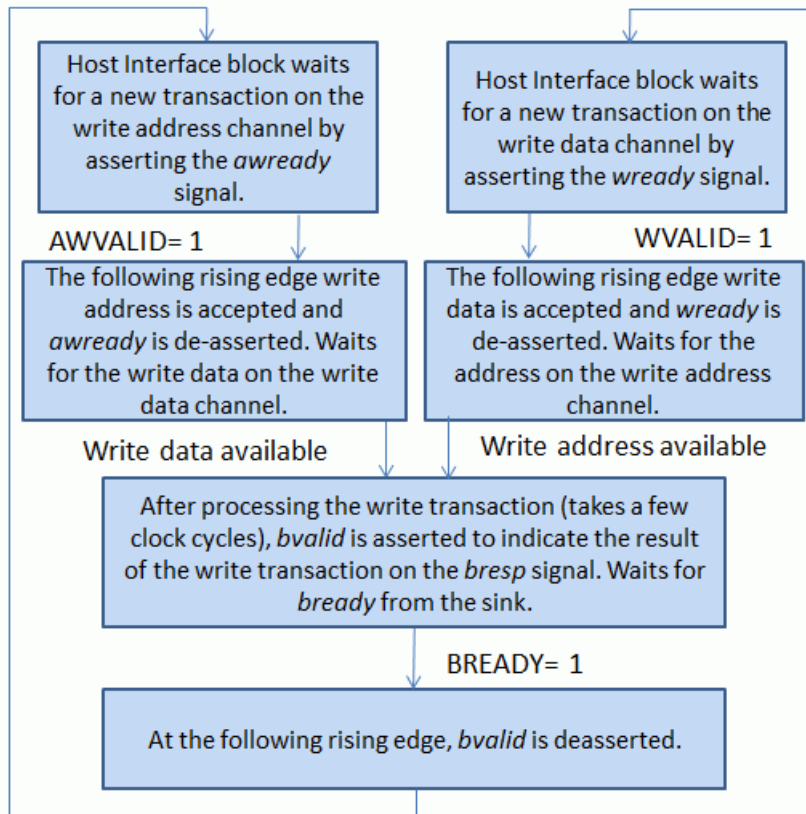
Handshake Process

The slave acts as a sink for the write address, write data, and read address channels. It acts as a source for the write response and read data channels. The source of a channel asserts VALID to indicate a transfer on that channel. The source then waits for the READY from the sink. On the edge following the READY assertion by the sink, the source disables the VALID and removes the data from the channel. The transfer occurs on the rising edge of clock, when both the VALID and READY of that channel are asserted.

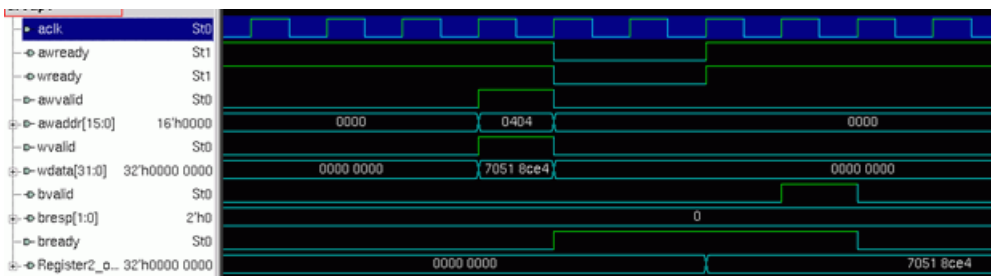
The source is not permitted to wait until READY is asserted before asserting the VALID. Once VALID is asserted, it must remain asserted until the handshake occurs, at a rising clock edge where VALID and READY are both asserted. The slave can choose to keep the READY asserted in its IDLE state or wait for VALID. The SMC Host Interface block does not wait for the VALID before asserting the READY signal.

Write Transactions

A complete write transaction requires the source to send the write address and data using the write address and write data channels and for the Host Interface block to respond back with the result through the write response channel. The following flow chart indicates the sequence of operations that takes place in a write transaction.



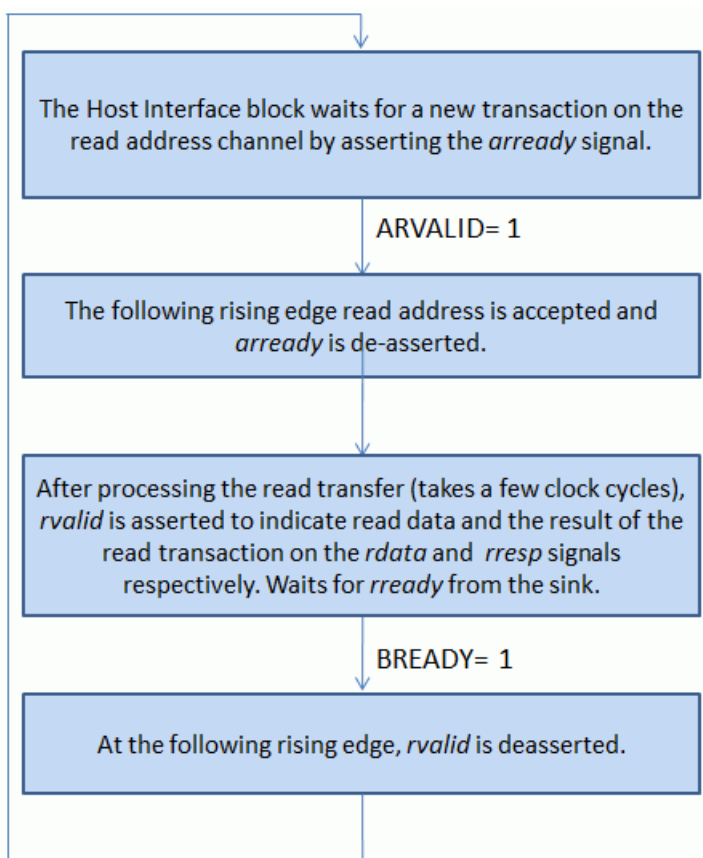
A snapshot of the simulation waveform for a typical write transaction is shown below.



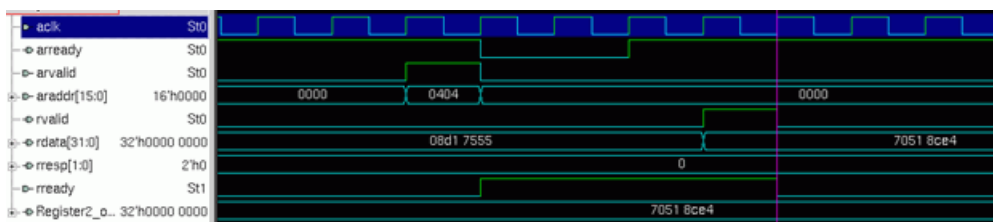
In this simulation, the source initiates the transaction by asserting **AWVALID** and **WVALID** along with the write address **16'h0404** and write data **32'h70518CE4**. As the slave (the Host Interface block) already has **AWREADY** and **WREADY** asserted, the source deasserts **AWVALID** and **VALID** on the following rising edge. The block then takes a few cycles to write the data in to the register that corresponds to the address (the signal shown in the waveform). The block then initiates the write response channel by asserting **BVALID** and driving the **BRESP** signal with a value of 0, which indicates that the transaction was successful. Since the source already has **BREADY** asserted, the Host Interface block deasserts **BVALID** on the following clock cycle. For details refer to the protocol specs.

Read Transactions

A complete read transaction requires the source to send the read address using the read address channel, and the Host Interface block to respond back on the read response channel with the read data and the result of the read transaction. The following flow chart indicates the sequence of operations that take place in a read transaction.



A snapshot of the simulation waveform for a typical read transaction is shown below:



In this simulation, the source initiates the transaction by asserting ARVALID along with the write address 16'h0404. Since the slave (the Host Interface block) already has ARREADY asserted, the source deasserts ARVALID on the following

rising edge. The block then takes few cycles to read the data (32'h70518CE4) from the register that corresponds to the address (the signal shown in the waveform). The block then initiates the read data channel by asserting RVALID, along with the read data on RDATA and the read response on RRESP. RDATA shows the data available in the register and RRESP indicates a value of 0, which means the transaction was successful). As the source already has RREADY asserted, the Host Interface block deasserts RVALID on the following clock cycle.

APB Protocol

APB (Advanced Peripheral Bus) is a part of the AMBA protocol family. APB is a low-cost interface that is optimized for minimal power consumption and reduced interface complexity. The Host Interface block implements the protocol as specified in revision C of the *AMBA APB Protocol v2.0* specification from ARM.

This table lists the bus interface I/O pins for the Host Interface block with the APB protocol:

Signal	Direction	Description
paddr	Input	Address input.
psel	Input	Select input. Indicates that the slave is selected.
penable	Input	Enable. Indicates the second or subsequent cycle of the transaction.
pwrite	Input	When asserted, indicates a write access. When de-asserted, indicates a read access.
pwwdata	Input	Write data.
pstrb	Input	Write strobes. This port is only created when the Support write strobes option is enabled for the Host Interface block.
pready	Output	Used by the slave to force the master to hold the transaction.
prdata	Output	Reads data from the slave.
pslverr	Output	Indicates a transaction failure.

The PPROT (protection type) signal defined in the APB specification is not implemented in the Host Interface block.

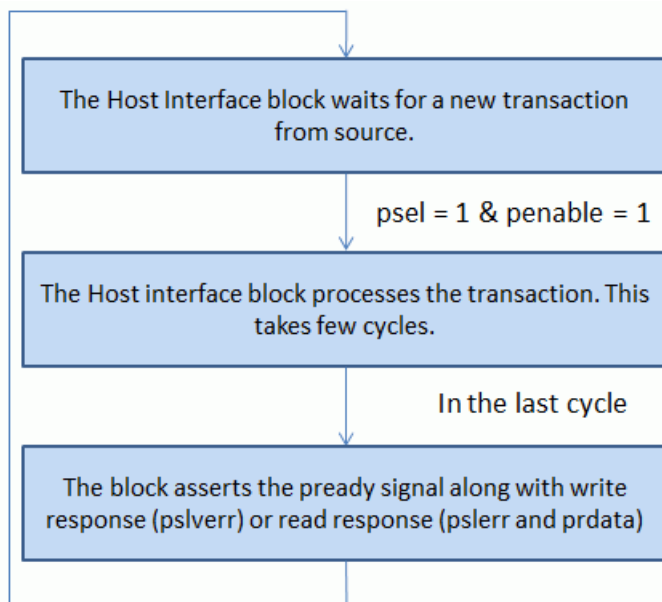
Handshake Process

An APB source first selects the APB slave by asserting the PSEL signal of that slave. The source can then initiate a transaction by asserting PENABLE and driving the rest of the signal (PWRITE, PWDATA, PADDR, and PSTRB) with their appropriate values. The source holds the transaction until the slave asserts PREADY. Once the slave asserts PREADY along with the response data (PRDATA in case of read transactions, and PSLVERR), the source releases the transaction on the following edge. If it can accept and respond to the transaction in a single cycle, the slave can keep PREADY asserted by default. Alternatively, it can wait for PENABLE and PSEL from the source before asserting PREADY.

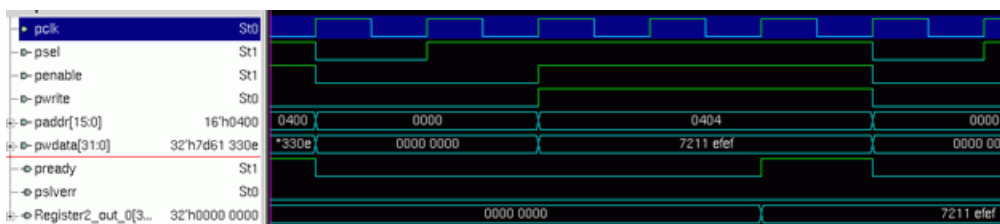
The Host Interface block does not assert the PREADY signal by default.

APB Transactions

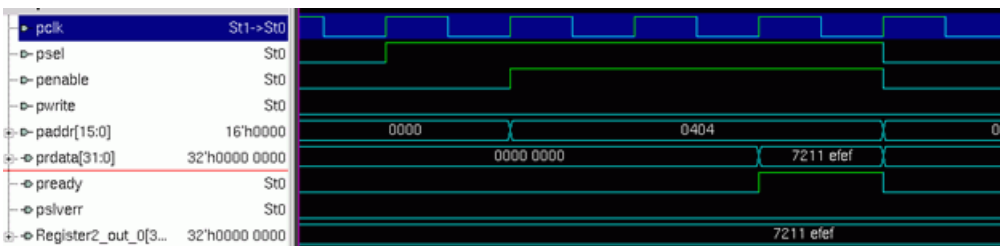
The following flow chart shows how the block responds to APB transaction requests.



The following snapshot shows a simulation waveform during a typical write transfer:



This snapshot shows the simulation waveform during a read transfer:



AVLON-MM Protocol

AVLON-MM is a memory-mapped version of the AVLON interfaces from Altera. It allows you to easily connect components in an Altera FPGA.

The table lists the Host Interface bus interface signals for the AVLON-MM protocol. The rest of the signals defined by the AVLON-MM specification are not implemented in the Host Interface block because they are redundant for a simple register configuration interface.

Signal	Direction	Description
address	Input	Address input.
begintransfer	Input	Asserted by a source to the slave on the first cycle of each transfer.
byteenable	Input	Enables specific byte lanes during transfers, for data widths greater than 8 bits. This port gets created only when the Support write strobes option is enabled.

Signal	Direction	Description
read	Input	Indicates a read transaction.
readdata	Output	Reads data output from the slave.
write	Input	Indicates a write transaction.
writedata	Input	Data to be written into the slave memory-mapped location.
waitrequest	Output	Used by the slave to force the master to hold the transaction.

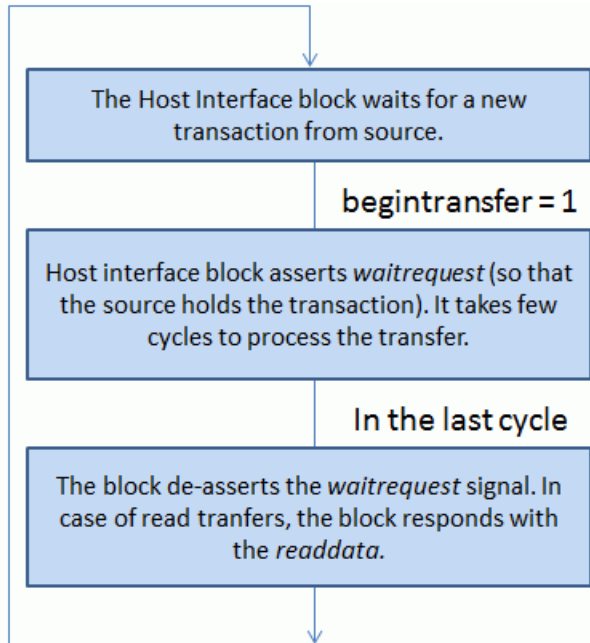
Handshake Process

The AVLON source initiates the transfer by asserting the BEGINTRANSFER signal. If the slave wants to extend the request, it asserts WAITREQUEST immediately. The slave can also choose to assert WAITREQUEST without waiting for BEGINTRANSFER. The source is required to hold the request until WAITREQUEST is de-asserted.

The Host Interface block asserts WAITREQUEST without waiting for BEGINTRANSFER.

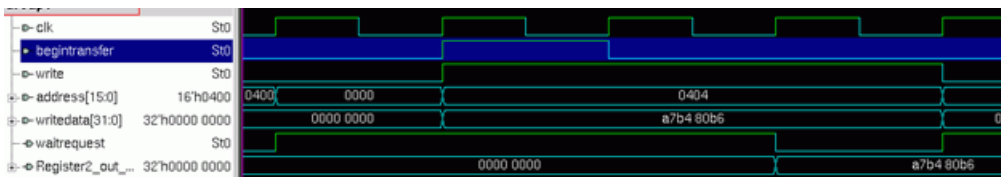
AVLON-MM Transactions

This flow chart shows how the Host Interface block responds to transaction requests:



AVLON-MM protocol does not define a way for the slave to communicate to the source if there is any error in the transaction.

The following figure shows the simulation waveform during a write transfer:



This is a snapshot of the simulation waveform during a read transfer:



Generic Interface Protocol

The generic interface for the Host Interface block simplifies the connection of third-party or custom bus protocols to SMC designs. The generic interface consists of two independent channels for request and response. Each channel offers a two-way flow control similar to the valid-ready handshake of the AXI4-Lite protocol described in [AXI4-Lite Protocol, on page 738](#).

The generic interface request channel includes the following bus interface signals:

Signal	Direction	Description
maddr	Input	Address input. Width is defined by the Address width option to the Host Interface block. See SMC Host Interface, on page 326 .
mread	Input	Single-bit signal that indicates a read transaction.
mwrite	Input	Single-bit signal that indicates a write transaction.
mdata	Input	Data to be written during the write transaction. Width is defined by the Data width option to the Host Interface block.
mwstrb	Input	Write strobes indicating valid bytes in the data to be written. This signal exists only if Support write strobes is enabled for the Host Interface block. Its width is equal to the number of bytes in the data width. The LSB bit corresponds to the LSB byte of the data bus, and the MSB bit corresponds to the MSB byte of the data bus.
saccept	Output	Single-bit signal that indicates that the slave is ready to accept transactions on the request channel.

The generic interface response channel includes these signals:

Signal	Direction	Description
svalid	Output	Single-bit signal asserted by slave to initiate a response transaction.
sdata	Output	Reads data from the slave during read response transaction. The width is defined by Data width option to the Host Interface block.
sresp	Output	Two-bit signal which is the response from the slave indicating the result of the transaction. <ul style="list-style-type: none"> • h'0 -> OK_R: Read transaction finished successfully • h'1 -> OK_W: Write transaction finished successfully • h'2 -> SLVERR_R: Slave-generated read access error • h'3 -> SLVERR_W: Slave generated write access error
mready	Input	Single-bit signal that indicates that the master is ready to accept response requests from the slave.

Handshake Process

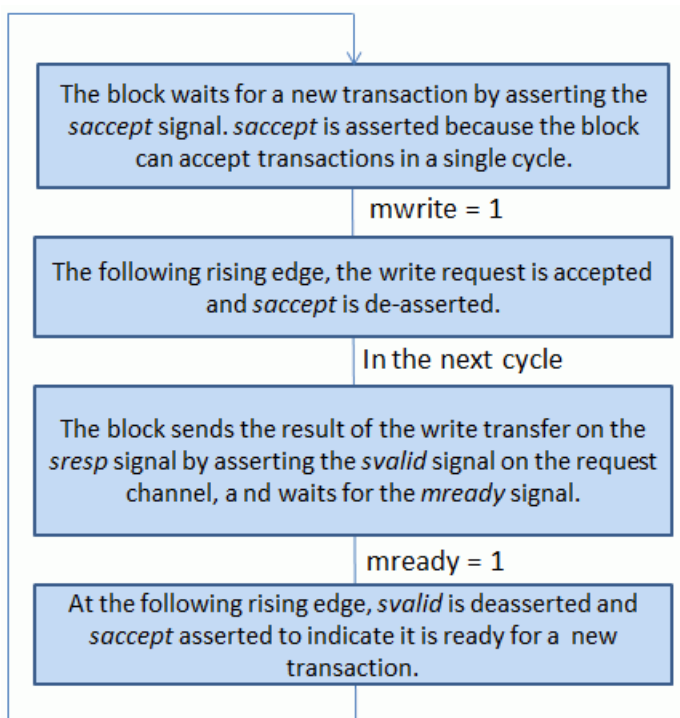
The source initiates a transaction on the request channel by driving either MWRITE or MREAD (for write and read transfers respectively) high along with appropriate values on the other signals, and waits for SACCEPT to be asserted by the slave. When the slave asserts SACCEPT, on the following rising edge, the source releases the request channel by deasserting MWRITE/MREAD. The slave can then communicate the result of the transaction on the response channel by asserting SVALID and waiting for MREADY. When the source asserts MREADY, the following cycle slave can release the response channel (SVALID = low).

The Host Interface block keeps SACCEPT asserted in its idle state, because it can accept and process the transaction in a single cycle. The block also asserts SVALID on the following rising edge after MREAD/MWRITE go high.

Write Transactions

In a write transaction, the source transfers the address and the write data on the request channel and the slave responds back with the result of the transaction on the response channel.

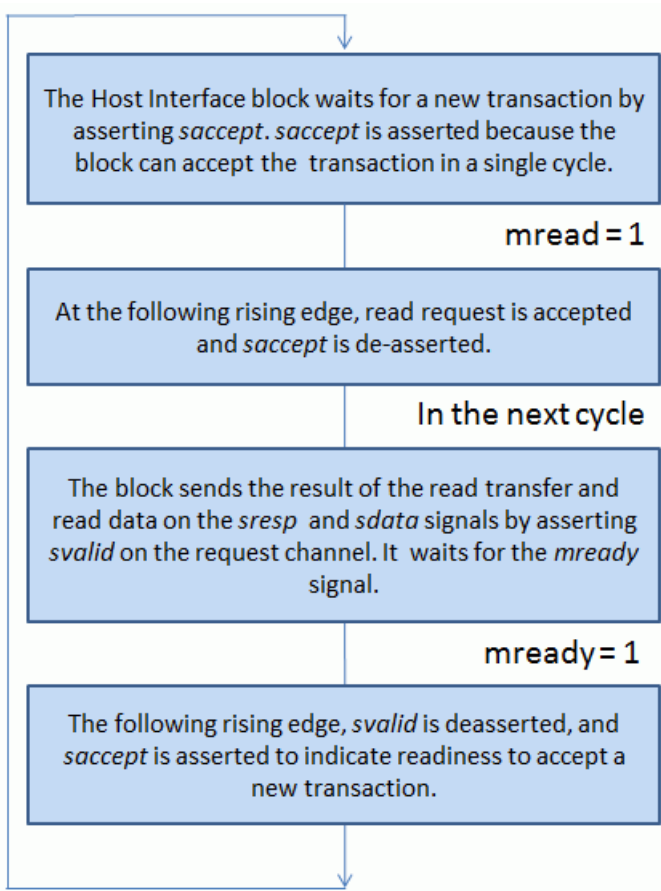
The following sequence explains how the block works in a write transaction.



Read Transaction

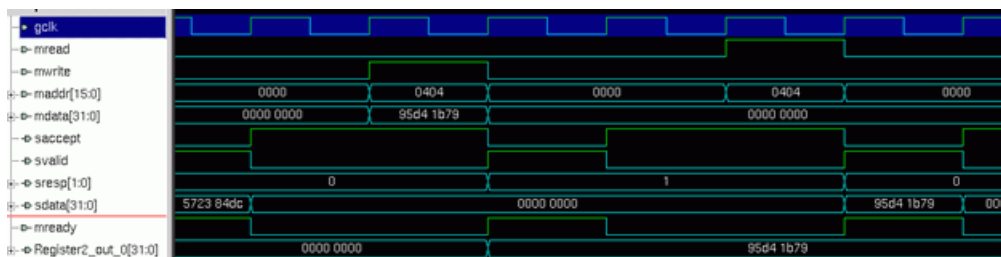
In a read transaction, the source transfers the address on the request channel and the slave responds back with the data and the result of the transaction on the response channel.

The following sequence explains how the block works in a read transaction.



Read and Write Simulation Waveform

This snapshot of the simulation waveform explains both write and read transactions.



The waveform shows that the source first initiates a write transaction by asserting MWRITE along with the address (16'h0404) and data (32'h95D41B79). As the SACCEPT signal of the Host Interface block is already asserted, the request channel is released on the following rising edge (MWRITE is low, MADDR and MDATA are released).

The Host Interface block processes the transaction within a cycle and hence it asserts SVALID in the same rising edge, along with the result of the write transaction in SRESP. SRESP = 1 indicates a successful write transaction. In the same rising edge, notice that the block writes the data on the register (the last signal in the waveform). As MREADY from the source is high, the Host Interface block releases the response channel on the following rising edge.

The source then follows another read transaction from the same address location. Notice the handshake process that took place on the request and response channels for the read transaction.

CHAPTER 8

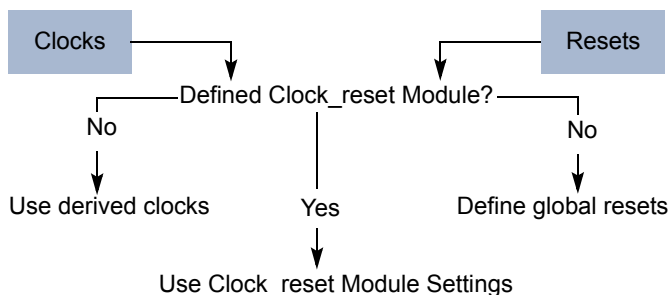
Designing with the SMC Tool

This chapter describes how to use the Symphony Model Compiler tool to define specific architectures or design features:

- [Defining Clocks and Resets, on page 754](#)
- [Designing Filters, on page 760](#)
- [Working with Vectors, on page 773](#)
- [Using Black Boxes and Third-Party IP, on page 777](#)
- [Specifying ROM Data with `syn_read_hex`, on page 776](#)
- [Managing Subsystems and Hierarchy, on page 786](#)

Defining Clocks and Resets

The following figure summarizes how clocks and resets are defined for the Symphony Model Compiler tool:



See the following for details:

- [Specifying a clock_reset Module, on page 755](#)
- [Defining Reset Signals, on page 758](#)
- [Clock Domains, on page 682](#), for information about derived clocks
- [Resets in the SMC Tool, on page 683](#), for information about resets

Do not use the following naming conventions for the Port In and Port Out blocks, to avoid conflicts with port names for inferred clocks:

- The name clk
- Name that begins with ClkDiv
- Name that begins with GlobalReset
- Name that begins with GlobalEnable

For more information, see [SMC Port In, on page 399](#) and [SMC Port Out, on page 403](#).

Specifying a clock_reset Module

The following procedure shows you how to define a `clock_reset` module that specifies the clocks and resets for your design. This module provides an easy way to define the inputs. For details about the interface between the module and the design, see [Clock_reset Module Interface, on page 688](#).

1. Open the Implementation Options dialog box and do the following:
 - Go to the Clock Reset Options tab.
 - Enable Generate Clock-Reset Circuitry. This ensures that the tool inserts a `clock_reset` module in the top-level Symphony Model Compiler design. The log file reports the creation of the module.

The screenshot shows the 'Clock Reset Options' tab of the Implementation Options dialog. The 'Generate Clock-Reset Circuitry' checkbox is checked. The 'Reset Module Parameters' section contains 'Power On Reset' (Active High) and 'User Reset' (Active High, Synchronous). The 'Clock Module Parameters' section contains 'Set of Clock Sources (MHz)' (empty), 'Clocking Scheme' (Dedicated Clocks), 'Clock Circuit Type' (Synthesizable Dividers), and 'Reset Deassertion Synchronization' (checked).

2. Define the clocks.
 - Specify the oscillator frequencies in Set of Clock Sources.

Single rate designs	Enter the frequency with or without square brackets. For example: [100] or 100.
Multi rate designs	Enclose the frequency values in square brackets, and separate the values with spaces, commas, or semicolons. For example: [100 200 300] [100, 200, 300] [100; 200; 300]

- To specify a single clock source, set Clocking Scheme to Enabled Clocks. The tool feeds each design clock with this fast clock and supplies global enable signals according to the required clock division ratio. By default, reset deassertion is not synchronized with the clock, but you can specify synchronization by enabling Reset Deassertion Synchronization.

- To specify separate design clocks, set Clocking Scheme to Dedicated Clocks. With this selection, reset deassertion is automatically synchronized with the clock, and the tool generates the required logic. Do not use this clocking scheme for M/N division ratios.

- If you specified separate design clocks, specify the logical structure for the design clocks in Clock Circuit Type.

For 1/N division ratios, or to use clock divider logic, set this option to Synthesizable Dividers.

If you want to take advantage of PLLs, set the option to Generic PLL. When you use this option, you must replace the placeholder code for PLL structures that this option generates with the actual code appropriate for the target FPGA.

For additional information about defining the clocks, see [Clock and Reset Management, on page 686](#).

3. Define the resets. See [Clock_reset Module Interface, on page 688](#) for a description of the signals.
 - To define the g_porst reset signal, set the polarity for the signal in Power On Reset Polarity. This signal is the power on reset signal for the design.
 - To define the optional g_urst user reset signal, enable User Reset. This signal is an optional user reset input for the design. Set the polarity

for the signal in User Reset Polarity. Set the sensitivity for the signal in User Reset Sensitivity.

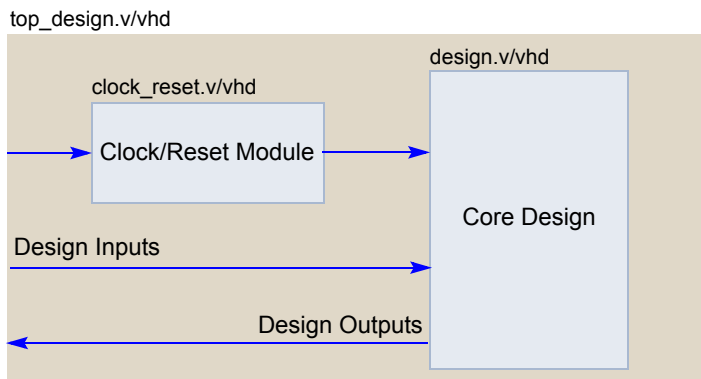
The reset signals have the same priority, so when `g_porst` or `g_urst` is asserted, the GlobalReset signal is asserted and the core design is held in reset state. At the same time, the clock generation circuitry inside the `clock_reset` module is in the reset state. Because of the synchronization circuits, the input reset signals must be held asserted for a minimum of two clock cycles (in terms of the slowest clock period in the design). When the `g_porst/g_urst` signal is deasserted, the clock generation logic inside the `clock_reset` module starts functioning. When the clocks become stable, the GlobalReset signal is deasserted and the core design also starts functioning.

If you selected a clocking scheme of Dedicated Clocks or Enabled Clocks with the Reset Deassertion Synchronization option enabled, the tool generates the logic required to synchronize resets with the clock when the reset is de-asserted. If you specify Enabled Clocks with the Reset Deassertion Synchronization option disabled, the tool does not synchronize the resets or generate the logic.

For information about limits to reset definition, see [Clock_reset Module Limitations, on page 690](#).

4. Run DSP synthesis.

The tool generates the following structure:



It also generates the corresponding files for the clock-reset circuitry. See [Clock/Reset Circuitry Files, on page 690](#) for details.

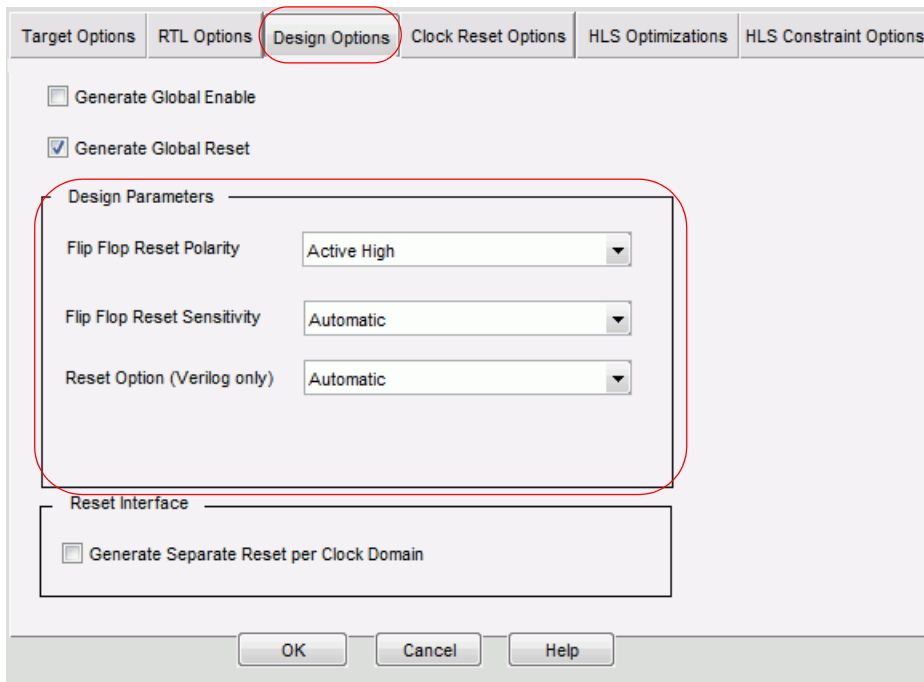
Defining Reset Signals

There are two ways to define reset signals for your design: by using a `clock_reset` module to define the circuitry, or by specifying global reset options.

1. To define resets in a `clock_reset` module, set up the module as described in [Specifying a clock_reset Module, on page 755](#).

If you choose not to create this module, the tool uses the settings in the Implementation Options dialog box. See the next step for details.

2. To define global reset signals without using the `clock_reset` module, do the following:
 - Open the Implementation Options dialog box and click Design Options.



- Set the polarity for the signals in Flip Flop Reset Polarity.
- To specify asynchronous or synchronous global reset signals, set Flip Flop Reset Sensitivity to Asynchronous or Synchronous, respectively. See [Synchronous and Asynchronous Resets, on page 684](#) for information about how the tool handles these settings.

- To automatically select the setting for your target vendor, set Flip Flop Reset Sensitivity to Automatic.
- For Verilog designs, use Reset Option to specify whether you want all registers to have resets. The recommended setting for FPGA designs is Automatic.

3. Complete the rest of the design and run DSP synthesis.

The RTL generated after synthesis includes reset inputs, as described in [Reset Implementation in RTL Code, on page 685](#). The global resets are also used to initialize the design for RTL simulation. See [Resets and RTL Testbenches, on page 686](#) for more information.

Designing Filters

This section shows you how to design FIR and IIR filters with Symphony Model Compiler. The Symphony Model Compiler FDATool block provides an interface to the MathWorks Filter Design and Analysis Tool, a part of the Signal Processing toolbox. The FDATool automatically generates coefficients for many styles of filters with different characteristics. You can use this tool to define filter coefficients. Symphony Model Compiler provides a function to automatically incorporate the coefficients generated by the FDATool.

The following provide more detail:

- [Implementing FIR Filters with the FIR2 Block, on page 760](#)
- [Implementing FIR Filters with the FIR Block, on page 764](#)
- [Implementing Polyphase FIR Filters, on page 767](#)
- [Defining FIR Filter Coefficients with FDATool, on page 768](#)
- [Implementing IIR Filters, on page 769](#)
- [Defining IIR Filter Coefficients with FDATool, on page 771](#)

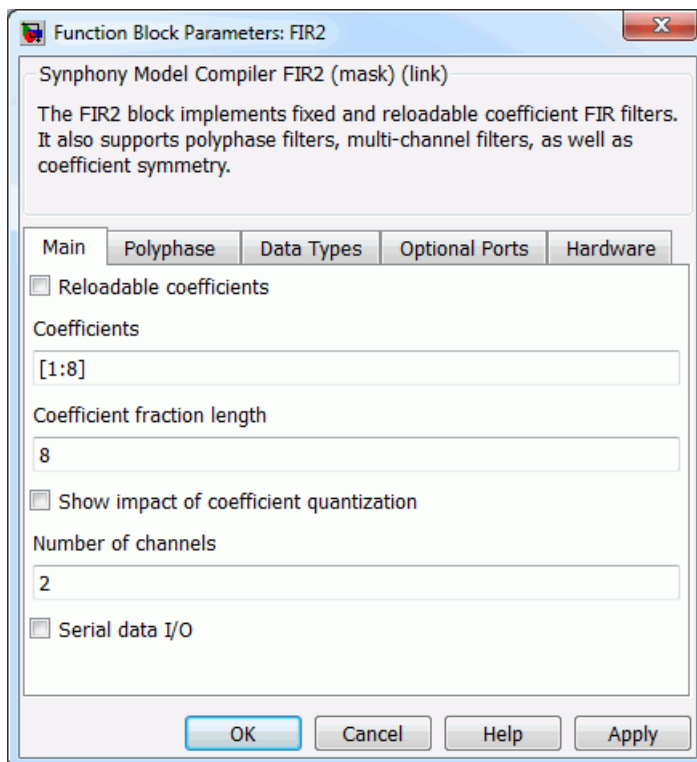
For some information about using adaptive filters, see the example in [Using Math Operations on Vector Signals, on page 774](#) and the LMS demo example.

Implementing FIR Filters with the FIR2 Block

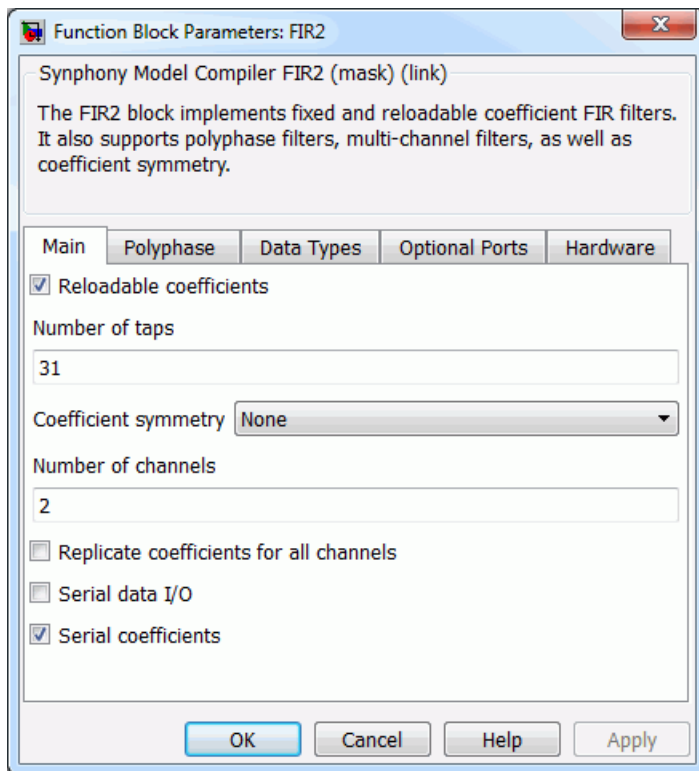
The FIR2 block lets you implement a number of different filters, including fixed coefficient filters, reloadable coefficient filters, single-rate filters, polyphase decimating filters, polyphase interpolating filters, single-channel filters, multi-channel filters, symmetric filters, antisymmetric filters, and half-band filters.

For a detailed description of the FIR2 block, refer to [SMC FIR2, on page 246](#).

1. Add the Symphony Model Compiler FIR2 block to your design.
2. Double-click the block and set the coefficient parameters for your FIR on the initial tab:
 - For a fixed coefficient filter, disable Reloadable coefficients, and enter a coefficient matrix in the Coefficients parameter.



- For a reloadable coefficient filter, enable **Reloadable coefficients**, and enter a value in **Number of taps**. The icon for the block reflects the addition of a coefficient port, **w**. You must provide the coefficients as an input vector or matrix through the **w** port.



3. Set other parameters, according to the type of filter you want to create.

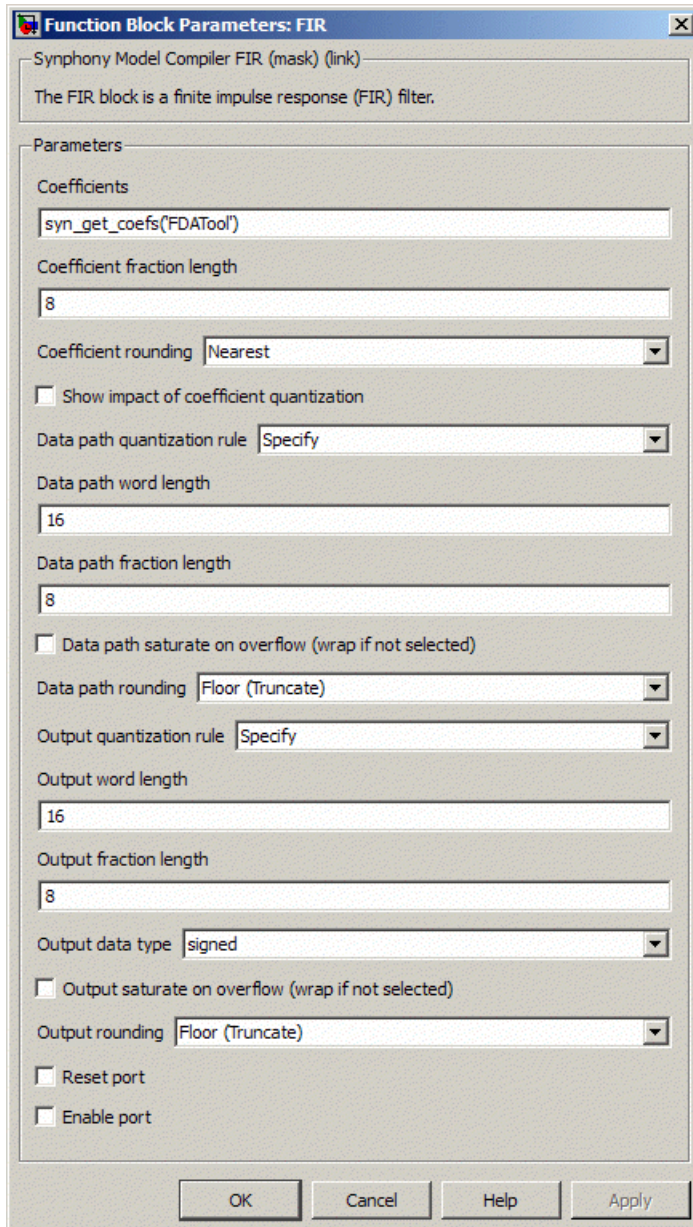
The following table describes the settings for some of these filters:

Filter Type	Setting Description
Single rate	<p>Polyphase tab-> Polyphase = None.</p> <p>For a folded implementation with folding within the filter taps, set Hardware->Hardware oversampling factor within phases/filters = >1.</p>
Polyphase decimator	<p>Polyphase tab-> Polyphase = Decimator.</p> <p>Set a value in Polyphase tab -> Decimation factor.</p> <p>To fold across phases, set Hardware->Hardware oversampling factor across phases = >1.</p>
Polyphase interpolator	<p>Polyphase tab-> Polyphase = Interpolator.</p> <p>Set a value in Polyphase tab -> Interpolation factor.</p> <p>To fold across phases, set Hardware->Hardware oversampling factor across phases = >1.</p>
Multichannel	<p>Main tab -> Number of channels = >1.</p> <p>Specify coefficients as a matrix:</p> <ul style="list-style-type: none"> • Reloadable coefficient filters Define the coefficient input to the w port as a matrix, with the number of rows equal to Main tab- Number of channels, and the number of columns equal to Main tab->Number of taps. • Constant coefficient filters Define Main tab->Coefficients as a matrix, with the number of rows equal to Main tab->Number of channels, and number of columns equal to number of taps. <p>To fold across channels, set Hardware->Hardware oversampling factor across channels = >1.</p>
Symmetric Antisymmetric Half band	<ul style="list-style-type: none"> • Reloadable coefficients Select the appropriate option from Main tab->Coefficient Symmetry, or specify unique coefficients through the w port. • Constant coefficients Specify the entire coefficient set in Main tab->Coefficients. The tool automatically infers the appropriate filter.

Implementing FIR Filters with the FIR Block

This procedure shows you how to implement FIR filters with the FIR block, but it is recommended that you use the FIR2 block, as described in [Implementing FIR Filters with the FIR2 Block, on page 760](#).

1. Add the Symphony Model Compiler FIR block to your design.
2. Define the filter coefficients in one of these ways:
 - Double-click the FIR block and specify MATLAB vector variables to define the coefficients in the Coefficients field.
 - Add the Symphony Model Compiler FDATool block and define the filter coefficients. See [Defining FIR Filter Coefficients with FDATool, on page 768](#), which describes this procedure in detail.
3. Set FIR block parameters.
 - Double-click the FIR block to open the block parameters dialog box.



The dialog box is titled "Function Block Parameters: FIR". It contains a description: "The FIR block is a finite impulse response (FIR) filter." Below this is a "Parameters" section with various settings:

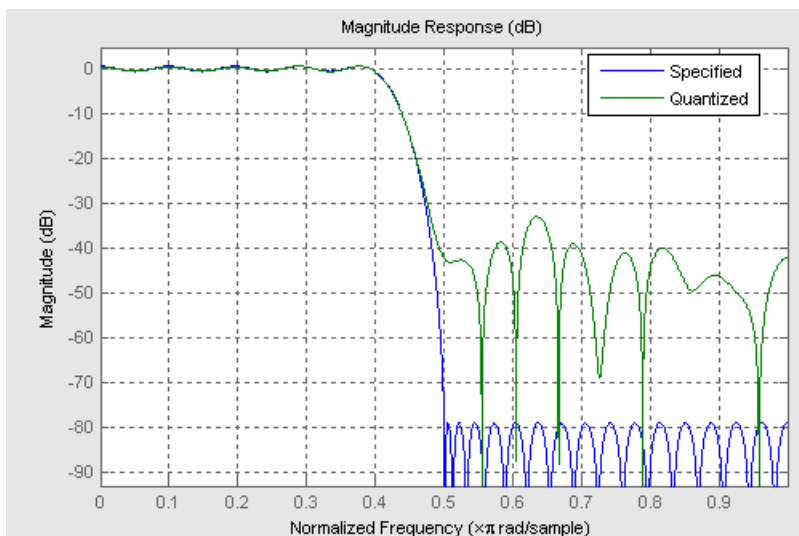
- Coefficients:** A text field containing `syn_get_coefs('FDATool')`.
- Coefficient fraction length:** A text field containing `8`.
- Coefficient rounding:** A dropdown menu set to `Nearest`.
- ☐ **Show impact of coefficient quantization**
- Data path quantization rule:** A dropdown menu set to `Specify`.
- Data path word length:** A text field containing `16`.
- Data path fraction length:** A text field containing `8`.
- ☐ **Data path saturate on overflow (wrap if not selected)**
- Data path rounding:** A dropdown menu set to `Floor (Truncate)`.
- Output quantization rule:** A dropdown menu set to `Specify`.
- Output word length:** A text field containing `16`.
- Output fraction length:** A text field containing `8`.
- Output data type:** A dropdown menu set to `signed`.
- ☐ **Output saturate on overflow (wrap if not selected)**
- Output rounding:** A dropdown menu set to `Floor (Truncate)`.
- ☐ **Reset port**
- ☐ **Enable port**

At the bottom are four buttons: **OK**, **Cancel**, **Help**, and **Apply**.

- If you used the FDATool block to define coefficients, set Coefficients to `syn_get_coefs('FDATool')`, making sure to use the correct quote

characters. The `syn_get_coefs` function imports the coefficients you defined in step 2.

- Fine tune quantization settings. The FIR block coefficients are quantized, based on the precision fraction bit length you specify in Coefficient fraction length. To view the impact of quantization, enable Show Impact of Quantization in the FIR block parameters dialog box. This automatically displays the effects of quantization.



- Optionally, set the precision of the internal format in Data path quantization rule and Output quantization rule.
- Set any other options you want in the dialog box.
- Click OK.

The software implements an FIR filter according to the criteria you specified.

Implementing Polyphase FIR Filters

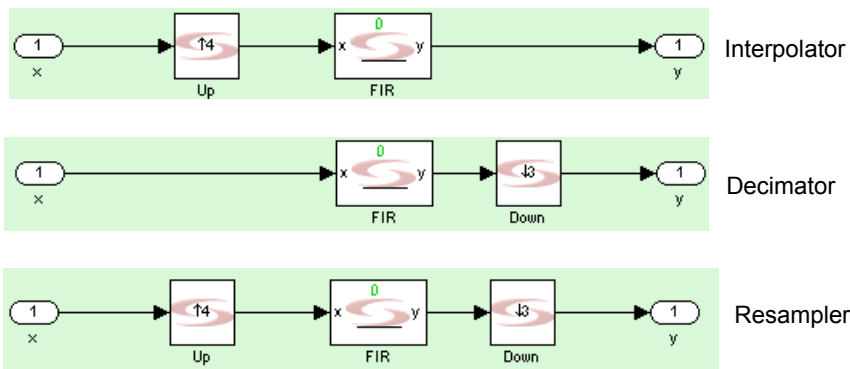
This procedure shows you how to implement polyphase FIR filters. You can use this method to implement interpolators, decimators, or resamplers. You can also implement polyphase decimating filters with the FIR2 block, as described in [Implementing FIR Filters with the FIR2 Block, on page 760](#).

1. Add the Symphony Model Compiler FIR Rate Converter block to your design.
2. Double-click the block to set the block parameters:
 - To implement an interpolator, set Filter type to Interpolator.
 - To implement a decimator, set Filter type to Decimator.
 - To implement a resampler, set Filter type to Resampler.

For details of the block parameters, see [FIR Rate Converter Parameters, on page 242](#).

3. Set the other options:
 - Set upsample and/or downsample rates.
 - Set the coefficients. See [Defining FIR Filter Coefficients with FDATA Tool, on page 768](#).
 - Set the data path and output formats.
4. Click OK.

The software implements a polyphase FIR filter according to the criteria you specified.

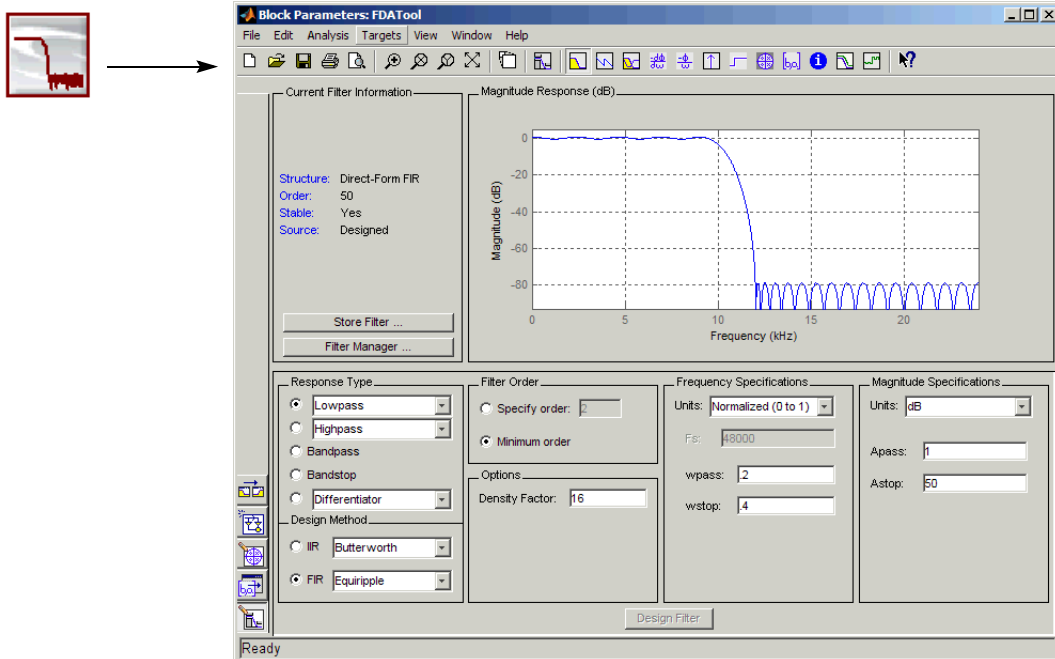


Defining FIR Filter Coefficients with FDATool

The following describes how to define coefficients for a Symphony Model Compiler FIR filter.

1. Add an FIR filter block from the Symphony Model Compiler blockset to your design.
2. Add the Symphony Model Compiler FDATool block. Double-click this block in the Simulink window.

A window opens with the MathWorks Filter Design and Analysis tool.



3. Specify the filter in the FDATool window
 - Set frequency and magnitude specifications for your filter.
 - From the FDATool menu bar, select Analysis->Filter Coefficients and verify the coefficients.
 - Close the FDATool window by clicking the X button.
4. In the Simulink schematic window, double-click the filter block.

5. Do the following in the parameters dialog box that opens:

- Type the `syn_get_coefs` function in the Coefficients field. For the complete syntax, refer to [syn_get_coefs, on page 604](#). The following table lists some typical ways to specify this function:

<code>syn_get_coefs</code>	Looks for the default instance 'FDATool'
<code>syn_get_coefs('Spec')</code>	Looks for the instance 'Spec'
<code>syn_get_coefs('Spec', 'forward')</code>	Takes all forward coefficients for the instance 'Spec'
<code>syn_get_coefs('Spec', 1:4:length(syn_get_coefs('Spec')))</code>	Picks the polyphase coefficient for the instance 'Spec'

- Click OK.

This updates the filter block with the coefficients you defined in the FDATool window. The tool updates the block icon in the Simulink schematic window to reflect the new coefficients.

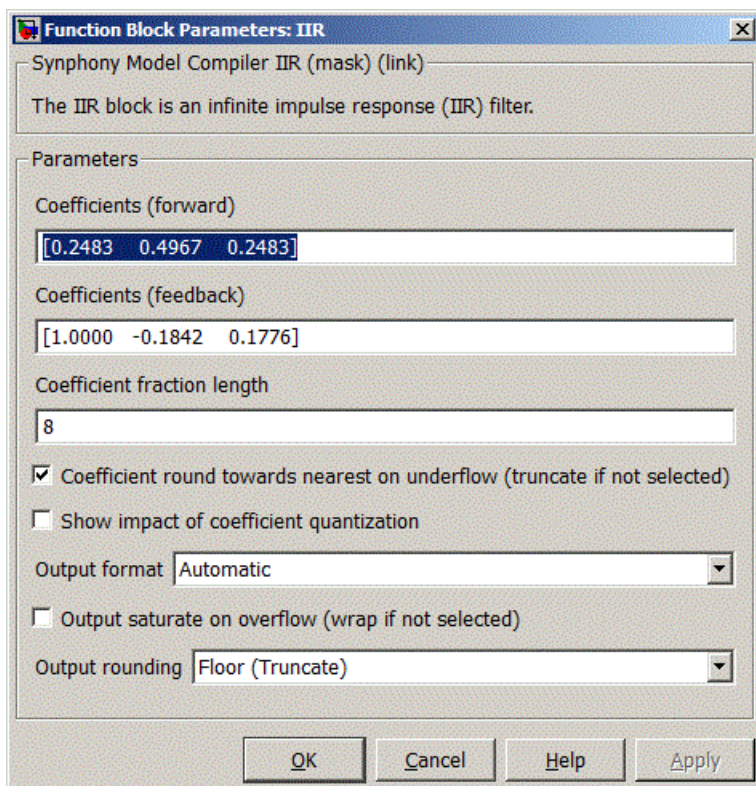
Implementing IIR Filters

This procedure shows you how to implement IIR filters.

1. Add the Symphony Model Compiler IIR block to your design.
2. Define the filter coefficients in one of these ways:
 - Double-click the IIR block to open the block parameters dialog box, and use MATLAB vector variables to define the forward and feedback coefficients.
 - Add the Symphony Model Compiler FDATool block and define the forward and feedback coefficients. See [Defining IIR Filter Coefficients with FDATool, on page 771](#), which describes this procedure in detail.
3. Set IIR block parameters.
 - If you have not done so, double-click the FIR block to open the block parameters dialog box.
 - If you used the FDATool block to define coefficients, set the two Coefficients fields to `syn_get_coefs('FDATool')`, making sure to use

the correct quote characters. The `syn_get_coefs` function imports the coefficients you defined in step 2.

- Optionally, set the precision of the internal format in Data path format and Output format.
- Set any other options you want in the dialog box.
- Click OK.



The software implements an IIR filter according to the criteria you specified.

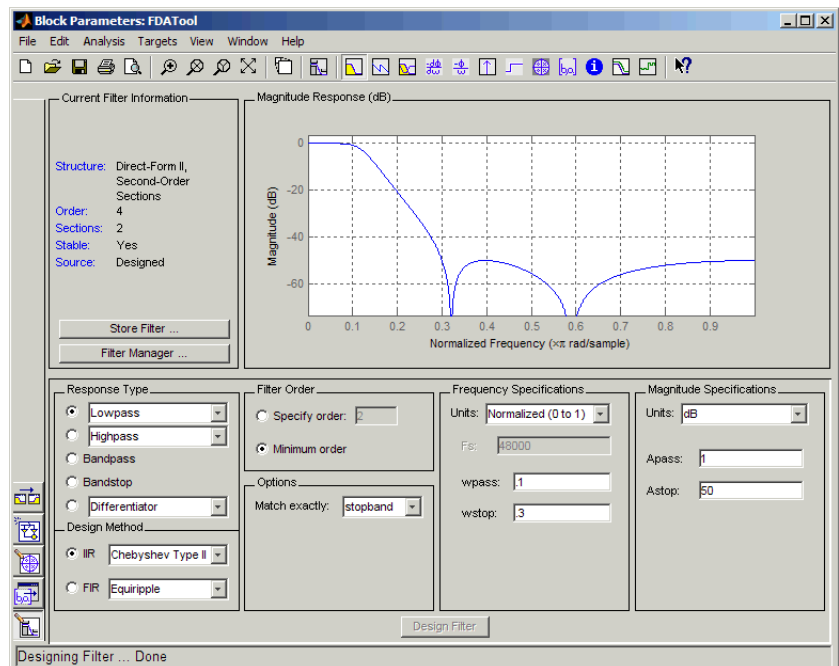
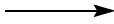
Defining IIR Filter Coefficients with FDATool

The following describes how to define coefficients for a Symphony Model Compiler IIR filter.

1. Add an IIR filter block from the Symphony Model Compiler blockset to your design.
2. Add the Symphony Model Compiler FDATool block. Double-click this block in the Simulink window.

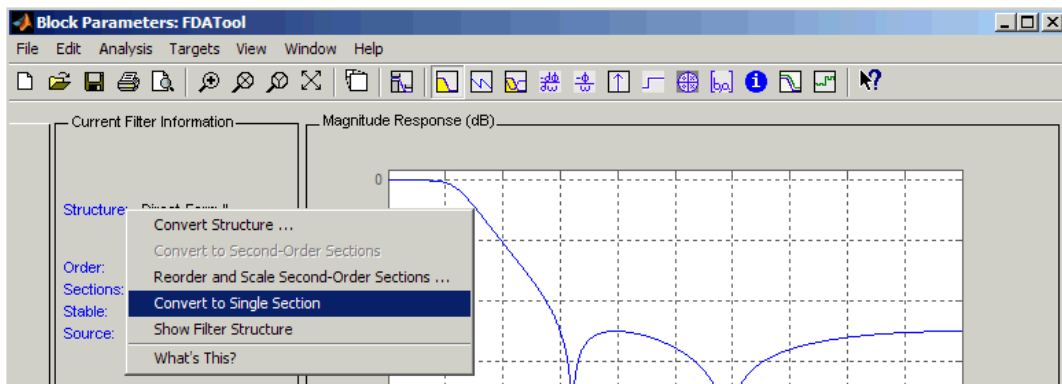
A window opens with the MathWorks Filter Design and Analysis tool.

3. Specify the filter in the FDATool window
 - Set frequency and magnitude specifications for your filter.
 - From the FDATool menu bar, select Analysis->Filter Coefficients and verify the coefficients.



4. Convert the filter structure to a single section.

- Go to the Current Filter Information section of the FDATool window, and right-click the word Structure.
- Select Convert to Single Section. If the filter design changes, make sure that the filter structure is still a single section, re-converting if necessary before attempting to extract its coefficients.



- Close the FDATool window by clicking the X button.
5. In the Simulink schematic window, double-click the filter block.
 6. Do the following in the parameters dialog box that opens:
 - Type the following in the respective Coefficients fields:
`syn_get_coefs('<instance>', 'forward')` and `syn_get_coefs('<instance>', 'feedback')`. If you do not specify an instance name, the function searches for an instance called FDATool. See [syn_get_coefs](#), on page 604 for the complete syntax for this function.
 - Set any other parameters and click OK.

This updates the filter block with the coefficients you defined in the FDATool window. The tool updates the block icon in the Simulink schematic window to reflect the new coefficients.

Working with Vectors

Many Symphony Model Compiler blocks accept vector signal inputs and adjust the operation to process the vector elements. For a quick summary of vector support and automatic scalar expansion on a per-block basis, see [Blockset Summary, on page 945](#).

This section describes the following:

- [Creating Vector Signals, on page 773](#)
- [Using Math Operations on Vector Signals, on page 774](#)

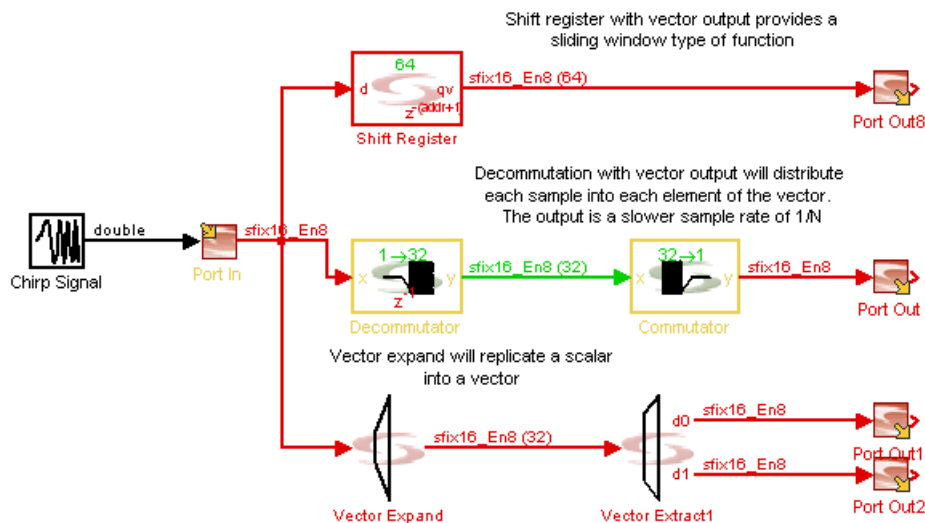
Creating Vector Signals

1. To bring in a vector signal from Simulink, use the Symphony Model Compiler Port In block.

Similarly, you can use the Port Out block to export vector signals to Simulink.

2. To create vectors from streaming scalar input, use the Symphony Model Compiler Decommutator and Shift Register blocks.
3. To merge or manipulate vectors, use the following blocks.

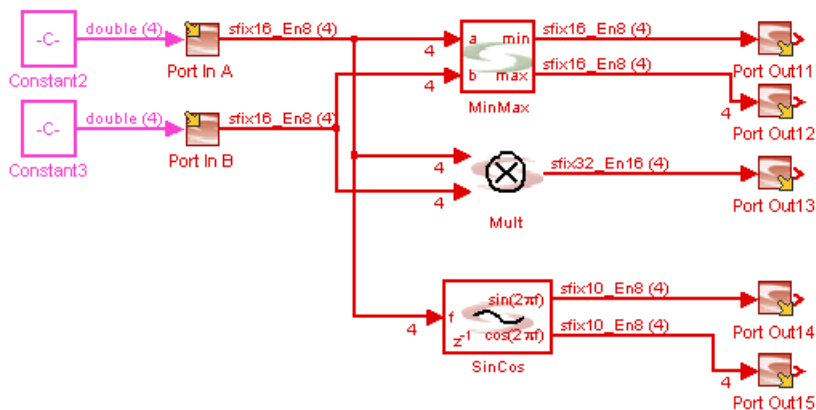
To...	Use...
Replicate a scalar input and create vector output	Vector Expand
Concatenate vector and scalar inputs to a single vector	Vector Concat
Generate scalar data from vector input	Vector Extract
Split vector input	Vector Split



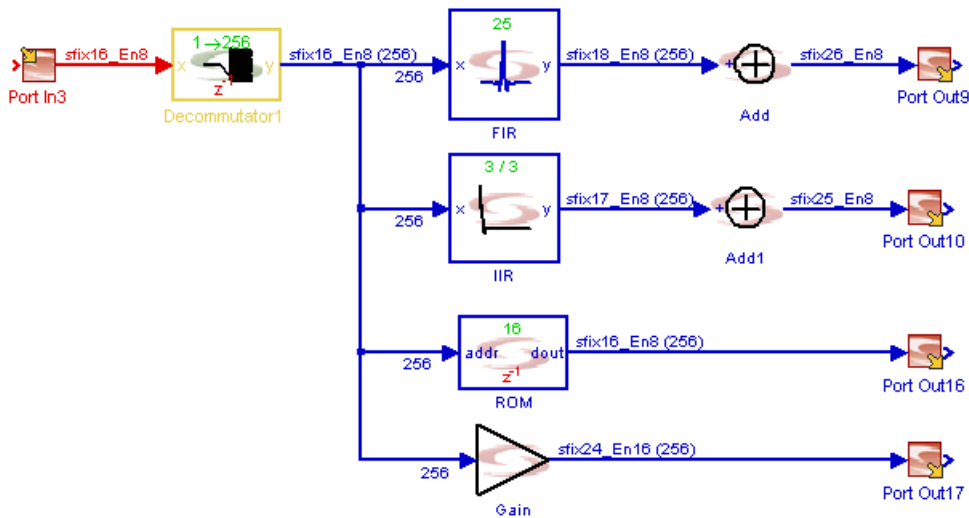
Using Math Operations on Vector Signals

Many blocks from the Symphony Model Compiler Math Functions library support vectors. For details, see [Blockset Summary, on page 945](#).

Some blocks have special capabilities for vector operation. The following table highlights them.



Sum the vector elements	Use the Add block and specify a single + operation. See the design example above.
Specify different values for each gain vector	Use the Gain block and specify a vector for the coefficients. See the example above.
Specify multichannel FIRs or IIRs	Use a matrix to specify different coefficients for each channel in the FIR and IIR implementations. See the following design example, which shows multichannel filters with vector inputs.
Specify multichannel memory	Define a matrix for the ROM block where each row specifies contents for each element. For vector RAM, the tool implements a RAM for each element.
Create adaptive filters with vectors	Use the FIR Engine and Reloadable FIR blocks, and use vectors to specify the coefficients. For an example, see the LMS adaptive filter demo.



Specifying ROM Data with syn_read_hex

As an alternative to typing in ROM vectors, you can use the `syn_read_hex` function to specify ROM data that is encoded in hex format in a file. Use the following procedure.

1. Ensure that you have a file with the ROM data in hex format.
2. Add the Symphony ROM block to your design.
3. Double-click the block in the model window to open the Block Parameters: ROM dialog box.
4. Set the parameters:

- In the ROM vector field, type the following:

`syn_read_hex <filename>`

See [syn_read_hex, on page 610](#) for the full syntax for this function.

- Set any other parameters needed.
- Click OK.

The function reads the hex-encoded data in the specified ROM file and converts it into vectors.

Using Black Boxes and Third-Party IP

A black box is a specialized Symphony Model Compiler block that lets you incorporate foreign IP into your Symphony Model Compiler design. Symphony Model Compiler offers two black box blocks, a simple black box and a smart RTL black box. Unlike a smart black box, a simple black box does not have accessible RTL code. This section describes the implementation of a general black box using the Black Box block. For information about using smart RTL black boxes, see [Using Smart Black Boxes for Cosimulation, on page 837](#).

For details about the Black Box block, see [SMC Black Box, on page 56](#). For an example, see `<install_dir>\mathworks\toolbox\`Synopsys\demoss\examples`. This section discusses the following:

- [Integrating Black Boxes in the Design, on page 777](#)
- [Setting Black Box Parameters, on page 780](#)
- [Configuring a Black Box - Example, on page 782](#)
- [Using Optimizations with Black Boxes, on page 784](#)

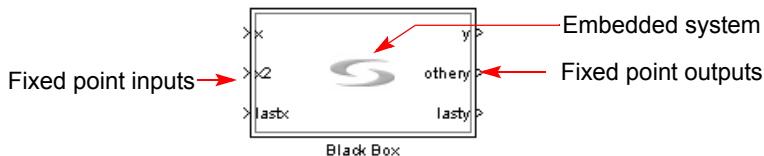
Integrating Black Boxes in the Design

By incorporating black boxes into your design you can build designs that include existing IP in another format, such as RTL or VHDL from a third-party IP provider. The Symphony Model Compiler Black Box block instantiates a wrapper in the RTL implementation, into which you can plug the RTL for the foreign IP. It lets you manage the simulation model and the interfaces of the foreign IP. If you want to simulate accurately with Simulink, you must provide the underlying simulation model for the IP. You can also use black boxes to implement a control function in RTL to drive a DSP function in the Symphony Model Compiler tool.

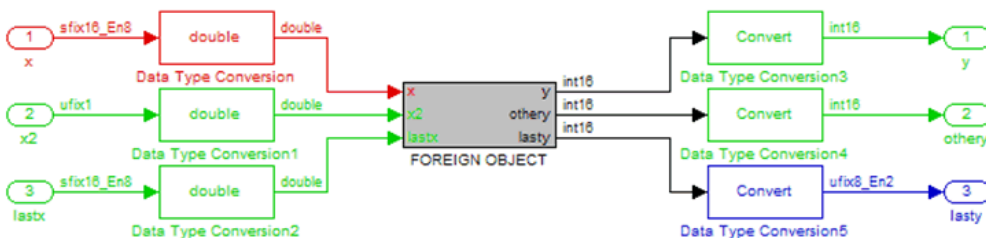
To incorporate a black box in your design, use the following procedure:

1. Select the Black Box block from the Symphony Model Compiler Ports & Subsystems library, and add it to your design.

This block provides the interface to the embedded block.



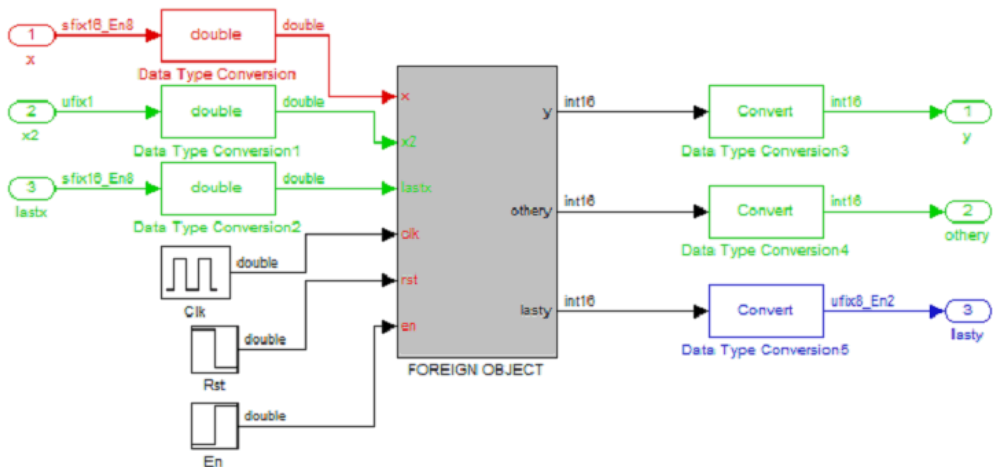
2. Double-click the black box, select Mask Parameters, and configure the black box to match your black box parameters. For an example, see [Configuring a Black Box - Example, on page 782](#).
3. Complete the internal black box design.
 - Right-click the black box and select Look under mask.
 - In the new view that opens, add the blocks you need between the input and output ports provided. See [Configuring a Black Box - Example, on page 782](#) for an example.
4. Make sure that the embedded system meets the following criteria:
 - The input ports must have a fixed point data type. They always inherit the sample period from Symphony Model Compiler blocks.



- The block must have a discrete sample time.
- The embedded object must connect to each black box output port through a Symphony Model Compiler Convert block, which adjusts the data type if needed.

5. Define the port interface of the embedded object.

- Align the bit positions of the input ports with the driver of these ports.
- Align the bit positions of the output ports with the sinks of these ports.
- Add hidden signals like clock, reset, and enable, to the instance ports, based on the sample periods of the signals going in and out of the block. This is sufficient to cosimulate with Symphony Model Compiler blocks.
- For complete simulation, you might need to manage the clock, reset and enable signals explicitly, using the appropriate Simulink sources, as shown in the following figure.



6. Provide a simulation model that represents the underlying RTL. There are several ways to do this:

- Use an RTL cosimulation tool like EDA Simulator Link MQ (formerly Link for ModelSim) or any other RTL simulator interface. For information about cosimulation, see [Using Smart Black Boxes for Cosimulation, on page 837](#).
- Contact your IP provider and obtain a Simulink model directly from them.
- Obtain C models from your IP vendor. You can then port these models to Simulink S-function models. For details, refer to the documentation from your IP vendor and Simulink.

7. Run Symphony Model Compiler synthesis.

- Set the Symphony Model Compiler optimizations you want for the design. For details, see [Using Optimizations with Black Boxes, on page 784](#).
- Synthesize the design and generate RTL.

When you generate RTL for the completed design, it includes an instance for the black box. The rest of the design is hooked up to the ports of the black box, with the appropriate connections for global enables, reset, and black box clocks you specified. Timing arcs stop at the input ports of the black box and resume from the output of the black box; they do not include the timing through the black box.

Setting Black Box Parameters

This procedure describes how to set parameters for a black box.

1. Double-click the black box.

The Function Block Parameters: Black Box dialog box opens, where you can set the parameters.

2. Specify the files that define the black box:

Black box defined in...	Specify these options...
Single Verilog or VHDL file	<ul style="list-style-type: none"> • Set Black Box Definition to Single HDL File. • In HDL File, specify the absolute path to the Verilog/VHDL definition file. • Specify the name of the top level entity in Entity/Model Name.
Single EDIF file, as with soft cores purchased from a third party	<ul style="list-style-type: none"> • Set Black Box Definition to Single EDIF File. • Specify the absolute path to the EDIF definition file in EDIF File. • In Simulation File, specify the absolute path to the simulation behavioral model file (Verilog or VHDL). • Specify the name of the top level entity in Entity/Model Name.

Black box defined in...	Specify these options...
Multiple Verilog, VHDL, or EDIF files	<ul style="list-style-type: none"> • Create a text file that lists the absolute paths to each Verilog, VHDL, or EDIF definition file and behavioral simulation file. • Set Black Box Definition to Import File List. • In Black Box File List, specify the absolute path to the text file you created. • Specify the name of the top level entity in Entity/Model Name.
Another black box block, or one for which you have no definition	<ul style="list-style-type: none"> • Set Black Box Definition to Undefined. • Specify the name of the top level entity in Entity/Model Name.

3. Specify a global reset port by doing the following:

- Enable Global Reset. When you write out RTL, the black box has a global reset port called GlobalResetSel. If you want the port to be called something else, go to the next step. If you do not want to generate a global reset port, do not enable the Global Reset option.
- For a global reset port with a name other than the default, enable Format Reset. Then type the name for the reset port in Reset Name.

Note that the global reset port you specify is not displayed in the Symphony Model Compiler design. It is only specified in the RTL generated after Symphony Model Compiler synthesis, and is hooked up to the global reset for the design.

4. Specify global enables by doing the following:

- Enable Global Enable. When you write out RTL, the black box has global enable ports with the default Symphony Model Compiler names. If you want the ports to be called something else, go to the next step.
- For global enable ports with names other than the default, enable Format Enable. In Enable Names, type the names of the enable signals, beginning with the fastest domain enable signal and separating signals with colons. For example: ce_sg:ce_2_sg.

Note that the global enable ports you specify are not displayed in the Symphony Model Compiler design. They are only specified in the RTL generated after Symphony Model Compiler synthesis, and are hooked up to the appropriate global enables.

5. To specify clock names other than the default, do the following:

- Enable Format Clock.
- In Clock Names, enter the clock names, beginning with the fastest clock and separating names with colons. For example: `clk_sg:clk_2_sg` specifies two clocks for your black box.

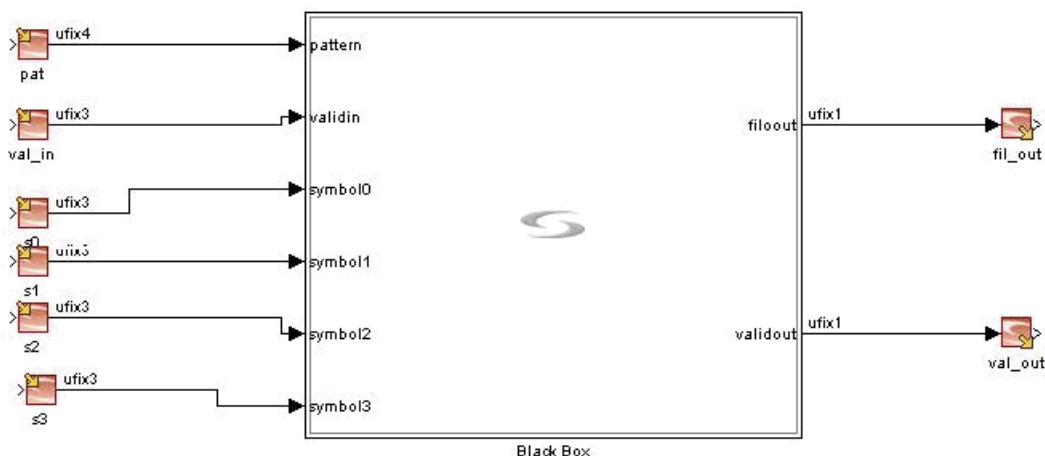
If you do not enable Format Clock, the black box uses the default naming convention for the clocks, with the fastest clock being `clk`, and N-reduced frequency clocks called `clkDivN`.

Configuring a Black Box - Example

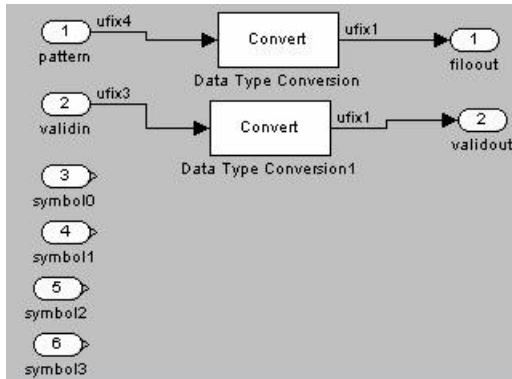
The following procedure illustrates how to configure a black box, using a Viterbi decoder as an example. For this example, the decoder is defined in a Verilog file, `C:\myblackboxes\viterbidecoder.v`.

1. Add the Symphony Model Compiler Black Box to the design and set it up to match the topmost interface specified. In this case, it is as follows:

```
module decoder(mclk, rst, valid_in, symbol0, symbol1, symbol2,
              symbol3, pattern, filo_out, valid_out);
  input mclk, rst, validin;
  input[2:0] symbol0, symbol1, symbol2, symbol3;
  input[3:0] pattern;
  output filoout, validout;
```



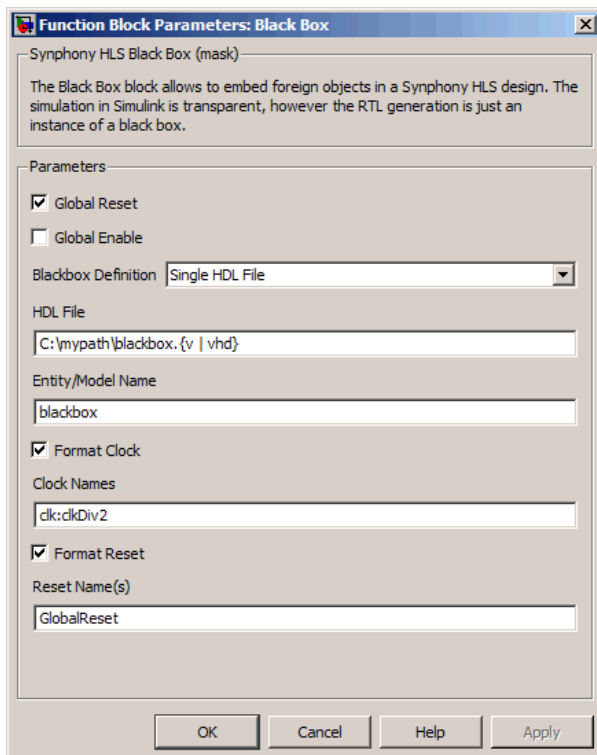
2. Specify the internal design by right-clicking the black box block and selecting Look under mask. For this example, add Convert blocks to adjust the output data type and convert ufix4 and ufix3 to ufix1.



3. Configure black box parameters by double-clicking the black box.

A dialog box opens. For detailed information about using various options on this dialog box, see [Setting Black Box Parameters, on page 780](#). For this example, set the following parameters:

- Specify the definition file. Leave Black Box Definition set to Single HDL File, and specify the absolute path to the Verilog file in the HDL File field: C:\myblackboxes\viterbidecoder.v.
- Specify the top entity. In Entity/Model Name, type decoder.
- Specify the global reset to match the decoder name. Enable Global Reset and then enable Format Reset. Type rst in Reset Name.
- Specify the clock to match the decoder name. Enable Format Clock and then type mclk as the clock name in Clock Names.
- Click OK.



You have now configured the black box. Note that with this example, you do not have a simulation model, so the RTL testbench generated after DSP synthesis will fail simulation. However, you can run RTL cosimulation, as described in [Using Smart Black Boxes for Cosimulation](#), on page 837.

Using Optimizations with Black Boxes

You can use the retiming, folding, and multichannelization optimizations in designs with both simple and smart black boxes. Note that the synthesis optimizations do not apply to the RTL inside the black box. However, the rest of the design is retimed, folded, or multichannelized, as long as the design follows the guidelines listed below.

Requirements for Retiming with Black Boxes

You can take advantage of retiming if your black box design meets the criteria below.

1. Make sure the design is one of the following:
 - Single rate black box in a single-rate design
 - Single rate black box in multi-rate design
 - Multi-rate black box
2. Make sure that the black box has registered inputs and outputs.

This is one of the assumptions that the tool makes for retiming. It retimes around a black box. It disables retiming into and across black boxes. If you do not have registered inputs and outputs, the tool generates sub-optimal retiming results, but maintains functionality.

Requirements for Folding with Black Boxes

The tool can integrate a black box into a folded design if your design is one of the following:

- A single rate black box in a single rate design
- A single rate black box in a multi-rate design
- A multi-rate black box in a multi-rate design. However in some cases, the tool might not generate code for multi-rate folded black boxes because of insufficient latency before or after the black box. If this happens, insert some extra delays in the design after and/or before the black box.

Requirements for Multi-channelizing Black Boxes

The Symphony tool can multi-channelize a design containing a black box. The tool instantiates a black box for each channel.

Managing Subsystems and Hierarchy

You can use subsystems to manage data type settings for a group of blocks collectively. The following procedures show you how to use the Subsystem and HLS Subsystem blocks. The Subsystem block lets you create hierarchical models using other blocks. The HLS Subsystem block lets you create a hierarchical model that uses external models, and which includes high-level optimizations like retiming, folding, and multichannelization.

- [Using the HLS Subsystem Block, on page 786](#)
- [Using the Symphony Subsystem Block, on page 792](#)
- [Tagging Subsystems with FPGA Synthesis Attributes, on page 796](#)

Using the HLS Subsystem Block

Use the HLS Subsystem block in a bottom-up design flow, where you implement independent modules and optimize them separately, and then assemble them at the top level.

For a hierarchical model that uses other blocks, use the Subsystem block or create a custom block, as described in [Using the Symphony Subsystem Block, on page 792](#) and [Working with Custom Blocks, on page 799](#).

The following sections describe how to use the HLS Subsystem block:

- [Creating an HLS Subsystem Block, on page 786](#)
- [Accessing User Variables, on page 792](#)
- [Locking HLS Subsystem Blocks, on page 792](#)

Creating an HLS Subsystem Block

This is the basic procedure for creating and using a HLS Subsystem block:

1. Make sure you have followed the prerequisites:
 - Specify a compatible C compiler during the setup process. The tool requires the C compiler to generate the C-model for the block.
 - Make sure your design matches the limits described in [Limitations to Using the HLS Subsystem Block, on page 322](#).

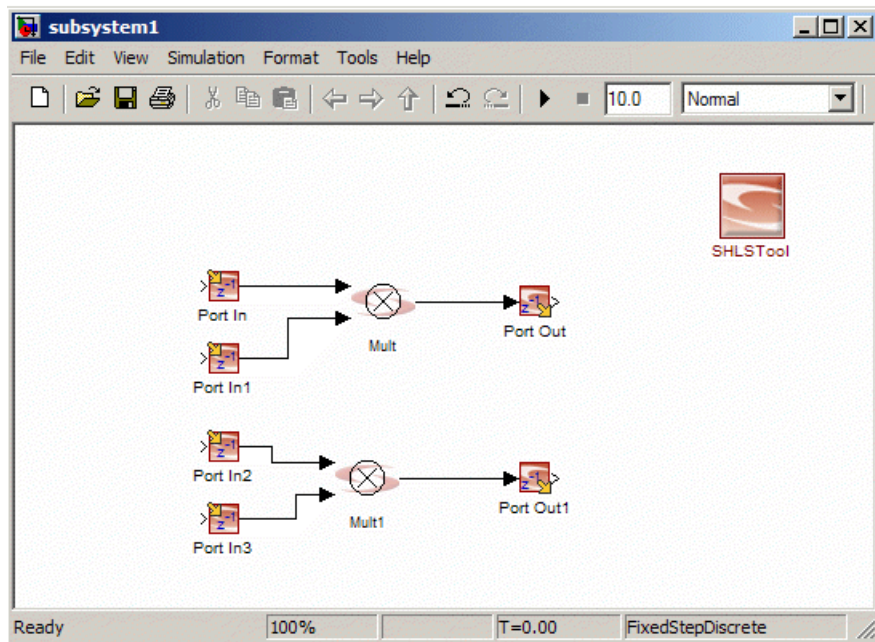
2. Create a subsystem.

- Capture the functionality of the subsystem in a Simulink model.
- Create an implementation for the subsystem and, set optimization options. If you select Retiming as an optimization for the subsystem, make sure to also select the Fixed Latency option.
- For reference subsystems, register the input and output ports as shown here:

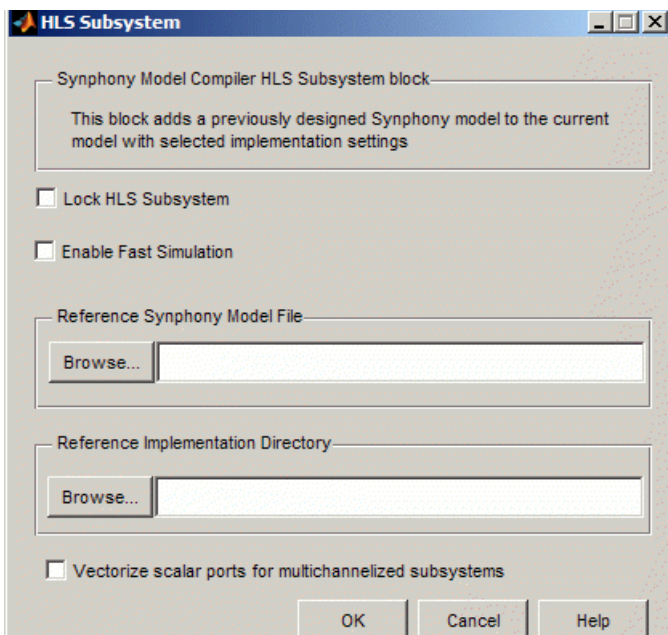
Implementation	Reference Subsystem Ports
Baseline	Need not be registered.
Retiming	
Folding	Must be registered.
Multichannel	Input ports of the subsystem must be registered.

- Generate RTL for the subsystem and check that the result is what you need for your design.

The following figure shows a simple subsystem design.



3. Set up the top-level design.
4. Add the subsystem to the top-level design by doing the following:
 - Add an HLS Subsystem block to the top-level design.
 - Double-click the block and set parameters for the subsystem. For details about the block parameters, see [SMC HLS Subsystem, on page 319](#).



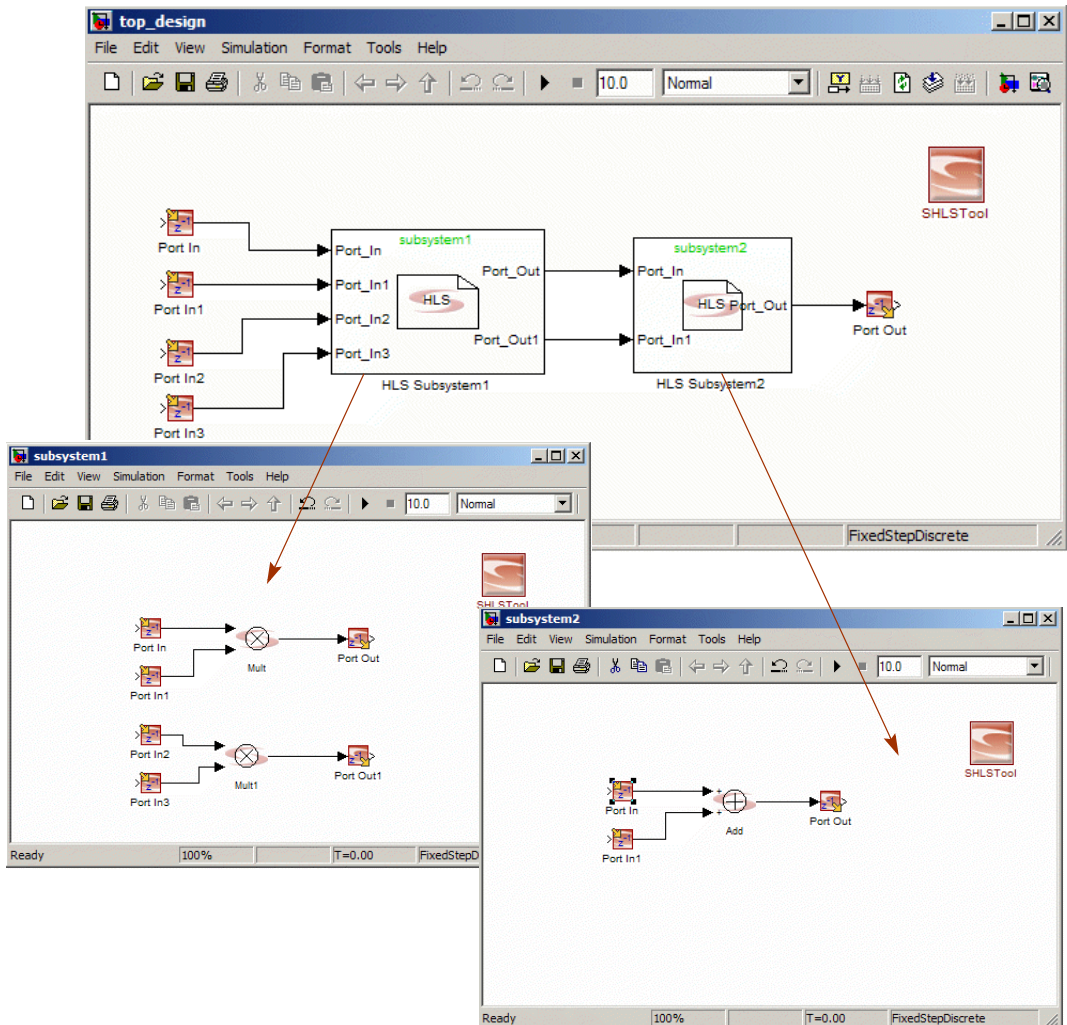
- Set Synphony model file to the mdl file you generated in Step 2. Set Implementation directory to the implementation you want, that you created in Step 2.
- If you want to preserve the exact behavior of the subsystem and its associated RTL as created in Step 1, check Lock HLS Subsystem. If you do not check Lock HLS Subsystem, the tool treats the subsystem as a separate Synphony model, and might override some subsystem implementation parameters with settings from the top level.
- Click OK.

The HLS Subsystem block is added to the top-level design.

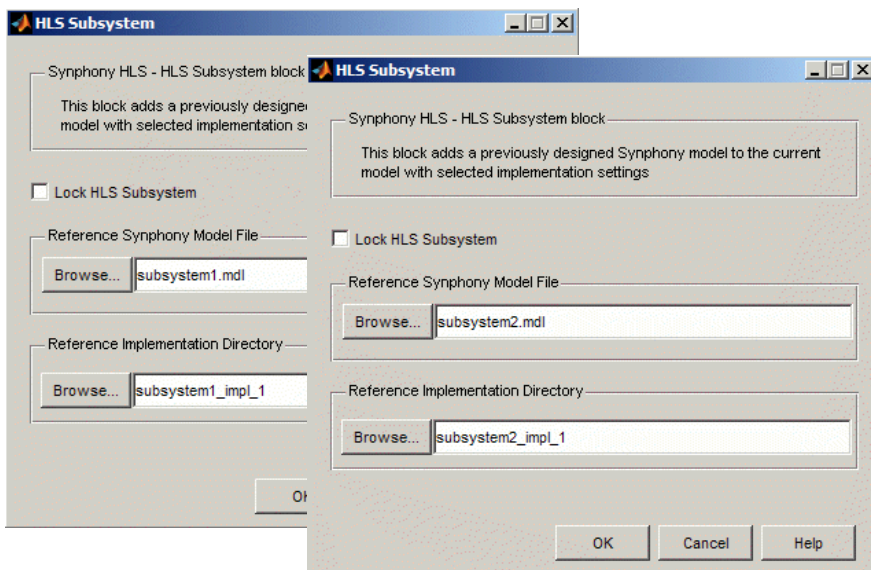
5. Repeat the previous steps for each HLS Subsystem block to be added to the top-level Simulink design.

Note that you can set different optimization parameters for each subsystem, but the synthesization parameters must match the top-level settings. If there is a mismatch in synthesization settings, the tool uses the settings from the top level and ignores the subsystem settings.

The following figure shows a top-level two mult-add design that includes subsystem1 (from Step 1) and subsystem2. The tool annotates the subsystem model names on each HLS Subsystem block icon. If you right-click a subsystem block and select Look under mask, you see the design of the underlying model.



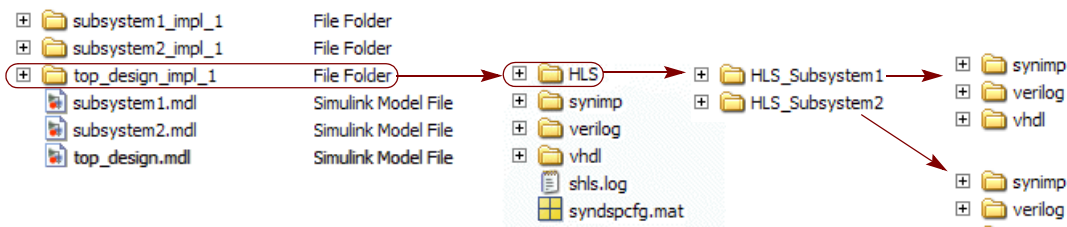
If you double-click a subsystem block, you see the parameters for that block:



At this stage, you see the tool has generated the following directory structure for your design:

+ subsystem1_impl_1	File Folder
+ subsystem2_impl_1	File Folder
subsystem1.mdl	Simulink Model File
subsystem2.mdl	Simulink Model File
top_design.mdl	Simulink Model File

When you generate an implementation, the tool creates a directory called HLS, with subdirectories for each HLS Subsystem block in the top-level design, as shown below:



6. Set top-level options and synthesize the design.

Note that you cannot currently set any retiming, multichannelization, or folding optimizations at the top level. If you have blocks at the top level that you want to optimize, put them in another subsystem (HLS Subsystem block) and optimize them in that way.

The tool runs through the phases described in [Tool Simulation Process for HLS Subsystem Blocks, on page 320](#) and [Tool Synthesis Process for HLS Subsystem Blocks, on page 321](#).

The shls.log file contains a separate section for each HLS subsystem block. The following is an example:

```
-----HLS Subsystem2-----
@N: Found 4 recognized primitive blocks other than the RTL
Generator.
@N: Advanced timing mode is off.
@N: Subsystem model name: "C:\WORK\HLS_Subsystem\subsystem2.mdl".
@N: Subsystem reference implementation directory:
"C:\WORK\HLS_Subsystem\subsystem2_impl_1".
@N: Subsystem is not locked.
@N: Subsystem latency: 0.

HIERARCHY INFORMATION
*****
Design has no subsystem.
-----

-----HLS Subsystem1-----
@N: Found 8 recognized primitive blocks other than the RTL
Generator.
@N: Advanced timing mode is off.
@N: Subsystem model name: "C:\WORK\HLS_Subsystem\subsystem1.mdl".
@N: Subsystem reference implementation directory:
"C:\WORK\HLS_Subsystem\subsystem1_impl_1".
@N: Subsystem is not locked.
@N: Subsystem latency: 0.

HIERARCHY INFORMATION
*****
Design has no subsystem.
-----
```

Accessing User Variables

HLS Subsystem blocks cannot access user variables created in the MATLAB base workspace. These variables can only be used by the top-level model.

1. Use the `PreLoadFcn` or `InitFcn` callbacks of the HLS Subsystem model to define and use variables specific to the HLS Subsystem block.

These subsystem variables are only visible to the HLS Subsystem block and cannot be accessed from the MATLAB base workspace.

Locking HLS Subsystem Blocks

When you use a HLS Subsystem block in your design, make sure that you do one run with it unlocked, so that you can generate RTL for it. If you do not do this, you can run into situations like the following:

- If you add the HLS Subsystem block to a model, pick a reference model and reference implementation directory and lock it before clicking OK, your implementation will have a single input-output block where the output is connected to the input internally.
- If you simulate the block and lock the subsystem before generating RTL, you get the following error message:

```
@E:Missing RTL for locked HLS Subsystem <Subsystem_Name>. RTL  
must be generated at least once without the lock.
```

Using the Symphony Subsystem Block

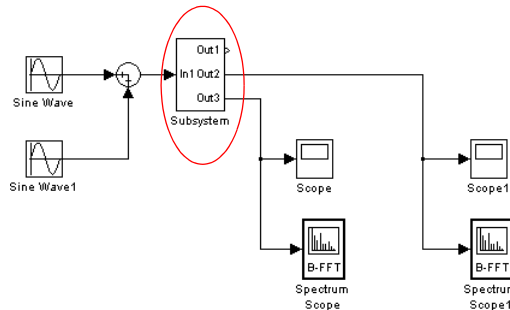
You use the Subsystem block to create a hierarchical model that uses other blocks. You can also use a custom block to do this. If you want to create a hierarchical model that uses external models, you must use the HLS Subsystem block, as described in [Using the HLS Subsystem Block, on page 786](#).

The following procedure uses a Subsystem block to manage fixed-point settings collectively for a group of blocks. For a custom block example that uses subsystems and hierarchy, see [Working with Custom Blocks, on page 799](#).

1. Create a subsystem. Creating a subsystem bundles the selected blocks together and creates an extra level of hierarchy.

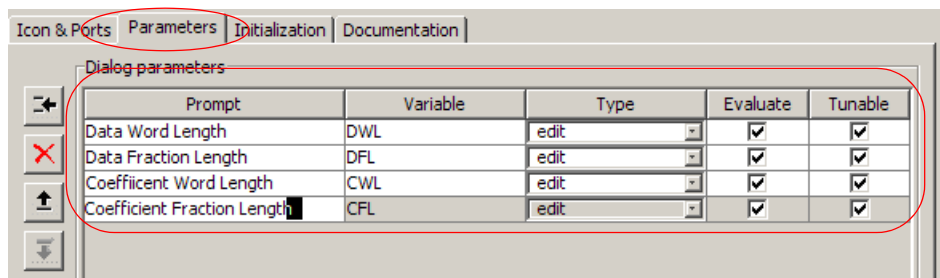
- Instantiate the Subsystem block from the Ports & Subsystems library and add the blocks you want to group together.
- Alternatively, draw a box around the blocks you want to group, right-click, and select Create Subsystem.

The blocks are grouped together and only the Subsystem block appears at the top level.



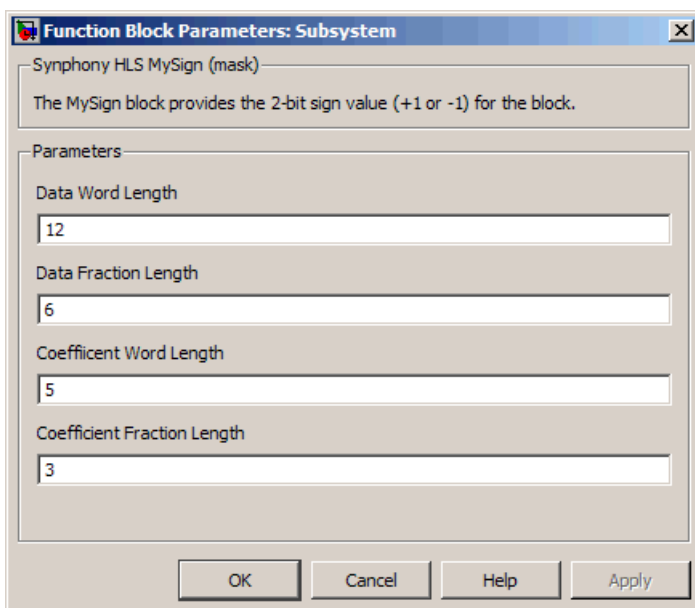
If you double-click the subsystem, another window shows you the internal hierarchy of the subsystem with the individual blocks you grouped.

- Set up a mask to manage the fixed-point settings:
 - In the schematic window, right-click the subsystem and select Mask System or Edit Mask to open the Mask editor window.
 - Click the Parameters tab, which is relevant for fixed-point conversion.
 - Specify variables to manage the fixed-point architecture:



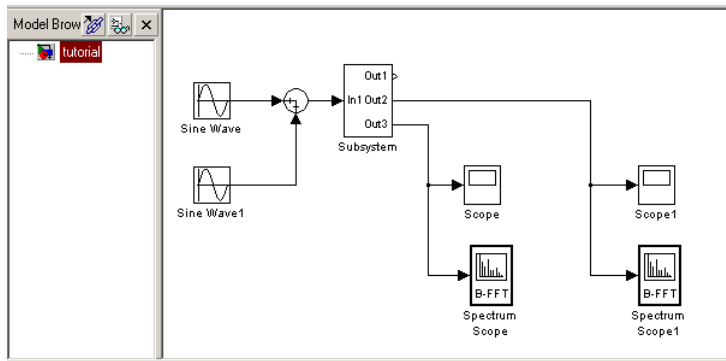
- Assign values to the variables.

- Double-click the subsystem block in the schematic window. The subsystem hierarchy underneath is no longer revealed, and the Function Block Parameters: Subsystem dialog box opens.
- Set the parameter values and click OK.

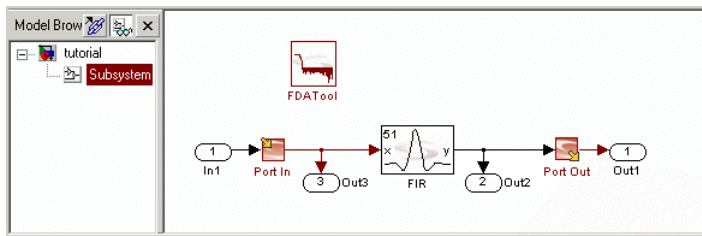


The tool applies these values to the subsystem. You can use any expression of the variables if you use the Specify option for the output format of individual blocks.

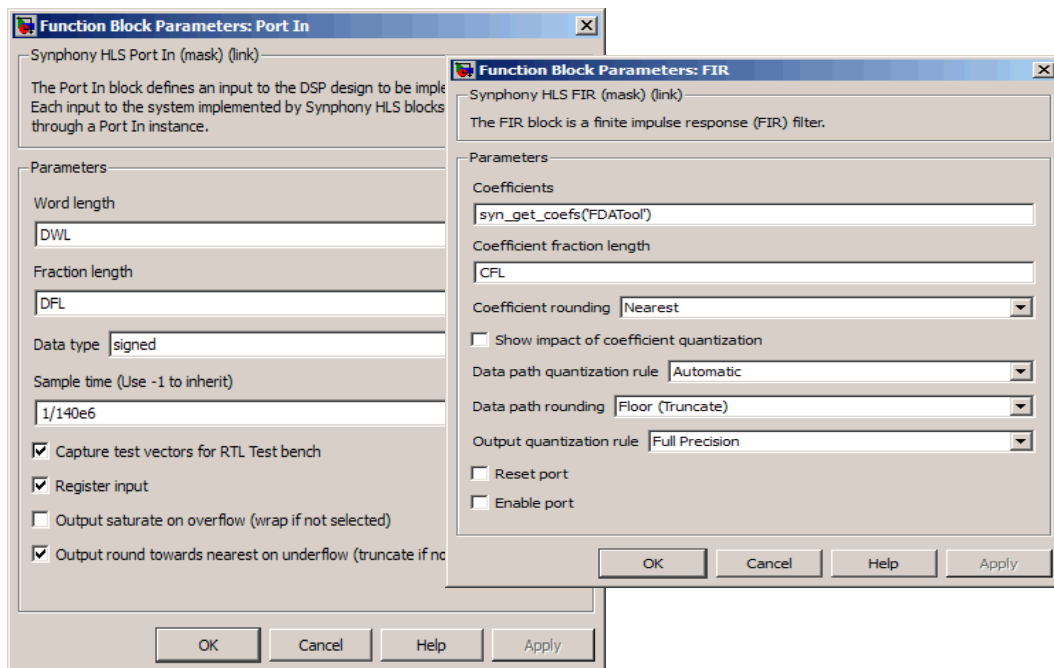
4. View the model hierarchy.
 - In the schematic toolbar, select View->Model Browser Options->Model Browser. This shows a tree view of the hierarchy.



- In the schematic toolbar, select View->Model Browser Options->Show Masked Subsystems. This adds a Subsystem entry to the left panel, which allows you to select the masked subsystem in the tree view.
- To view the original design which is in the subsystem, select Subsystem in the left panel tree view.



5. Set the block parameters to the design variables. You can take advantage of the mask parameters and automatic data type overwrite to determine the appropriate settings. Do the following on a per-block basis:
 - In the schematic window for the subsystem, double-click the block to redisplay the parameters dialog box.
 - Set the appropriate options like Word Length, Fraction Length, Coefficient Length, etc. to the variables you defined in step 2.
 - Click OK.



- If you need to control the rate for the subsystem, specify the number of cycles with a multicycle path constraint on the subsystem:

```
define_attribute <subsystem> multi_cycle_path <cycles>
```

Tagging Subsystems with FPGA Synthesis Attributes

The Synphony Model Compiler tool provides an infrastructure that enables you to apply Synplify Pro or Synplify Premier synthesis pragmas or attributes by specifying them as block tags in the Simulink mdl. The SMC tool simply passes on the pragma, but does not validate that it is valid.

There are three scenarios where Synplify synthesis attributes are applied to SMC designs:

- If the attribute can be applied on a module, you can specify the attribute through a block tag on a subsystem in your SMC design, as described in the procedure below.

- If you tag a Simulink connection wire (signal), the SMC tool automatically adds a `syn_keep` attribute to the corresponding wire declaration in the output RTL.
- You can use the relevant SMC Tcl command to set retiming on specified registers, and the SMC tool automatically translates this to the corresponding synthesis retiming attribute.

You can tag Simulink subsystem blocks with Synplify Pro or Synplify Premier synthesis pragmas or attributes by following these steps:

1. Select a Simulink subsystem block, right-click, and select **Block Properties**.

You can only set pragmas on Simulink subsystem blocks because they become modules in the generated RTL.

You can encapsulate your block in a subsystem. For example, to specify `syn_srlstyle=SRL32` on a shift register, first encapsulate the shift register in a Simulink subsystem by right-clicking and selecting **Create subsystem**. Then specify the attribute as described in step 2 below.

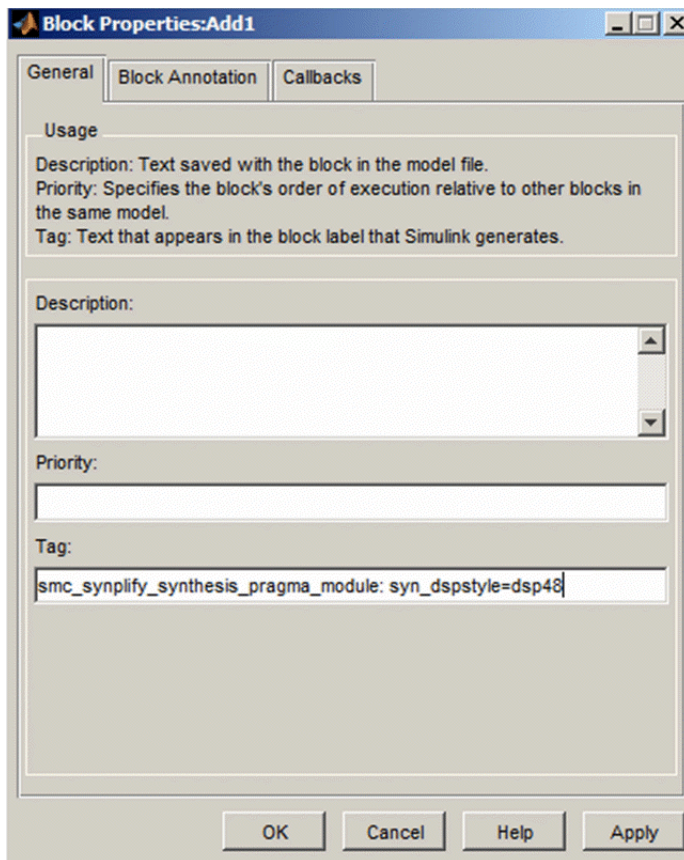
2. Specify the attributes you want in the **Tag** field using the syntax below. Separate multiple synthesis primitives with commas.

```
smc_synplify_synthesis_pragma_module=<FPGA synthesis attribute>
```

For example:

```
smc_synplify_synthesis_pragma_module:syn_srlstyle=SRL32
```

The figure shows the `syn_dspstyle` attribute set to `dsp48`. You can specify any synthesis directive that applies to a module with `smc_synplify_synthesis_pragma_module`. See the Synplify documentation for information about directives that apply to modules.



3. Validate the implementation in the synthesis tool.

The SMC tool simply passes on the specified attribute, and does not validate that the pragma is valid. It is up to you to validate all such pragmas. In particular, the Synplify synthesis tools might map large shift registers into block RAMs because it is more efficient, and ignore the `syn_srstyle` pragma that was specified on the shift register block.

CHAPTER 9

Working with Custom Blocks

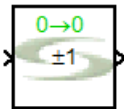
This tutorial describes how to generate a custom block:

- [Primitives and Custom Blocks, on page 800](#)
- [Design Flow for Building Custom Blocks, on page 804](#)
- [Set up a Custom Library, on page 805](#)
- [Create a Custom Block, on page 806](#)
- [Define Basic Content for Custom Blocks, on page 812](#)
- [Define Content for Parameterized Blocks, on page 816](#)
- [Define Content for Reconfigurable Blocks, on page 820](#)
- [Designing with Custom Blocks, on page 823](#)
- [Maintaining Custom Libraries, on page 824](#)
- [The MySign M-Generator, on page 826](#)

Primitives and Custom Blocks

The Symphony Model Compiler software includes two kinds of blocks:

- Primitive blocks are basic functions. They can be used to build a custom block.
- Custom blocks are more complex blocks for custom IP or higher-level functions. The Symphony Model Compiler tool includes some custom blocks as part of the blockset. In addition, you can create your own custom blocks (see [Design Flow for Building Custom Blocks, on page 804](#)). Custom blocks are distinguished from primitive blocks by their icon; the Symphony Model Compiler custom blocks have a green Synopsys S logo instead of a red one.



Custom block with green S logo



Primitive block with red S logo

When you right-click on a custom block, you can see its mask parameters. With a primitive block, you cannot see the mask parameters.

There are two advantages to using custom blocks:

- You can generate your own custom IP, or higher-level functions using the Symphony Model Compiler primitive blocks and other custom blocks. The Symphony Model Compiler tool will generate RTL for these blocks as it will for the primitives.
- The Symphony Model Compiler custom blocks can be optimized like the primitive blocks.

List of Custom Blocks

The following table lists the Symphony Model Compiler custom blocks:

SMC Block Deinterleaver	Shuffles a fixed number of interleaved input symbols to obtain the original sequence.
SMC Block Interleaver	Shuffles a fixed number of input symbols to a new permutation.
SMC CIC	Implements a CIC filter.
SMC CIC2	Implements a CIC filter with additional enhancements compared to the CIC block.
SMC Commutator	Sequentially switches the data from the specified number of input ports to a single output port.
SMC Convolutional Deinterleaver	Reshuffles streaming input symbols according to a predefined mapping scheme.
SMC Convolutional Encoder	Corrects feed-forward errors using k/n convolutional codes.
SMC Convolutional Interleaver	Shuffles streaming input symbols to a new permutation, using a predefined mapping scheme.
SMC CORDIC2	Implements a circular CORDIC (Coordinate Digital Rotation Computer).
SMC DDS	Creates a direct digital synthesizer, with sin and cos waves based on frequency, phase settings, and modulations.
SMC DDS2	Creates a direct digital synthesizer with sin and cos waves based on frequency, phase settings, and modulations. This block provides additional functionality and QoR improvements compared with the DDS block.
SMC Decommutator	Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly.
SMC Depuncture	Removes specified bits from the input data stream and replaces them with zeroes.
SMC Differentiator	Performs a discrete time differentiation of the input signal.
SMC Extract	Provides an n-bit integer based on the value of the bits extracted from specified positions at the input.

SMC FIR2	Implements fixed and reloadable coefficient FIR filters, including polyphase filters, multichannel filters, and symmetric coefficient filters.
SMC FIR Rate Converter	Implements a polyphase FIR filter.
SMC Integrator	Performs a discrete time integration of the input signal.
SMC Leading Zero Counter	Computes the number of leading zeros for an unsigned input.
SMC MinMax	Calculates the minimum, maximum, or minimum and maximum of two inputs.
SMC Moving Average Filter	Implements a hardware efficient moving average filter.
SMC Parallel FIR	Implements a parallel input FIR filter.
SMC Parallel to Serial	Implements a data packet splitter that divides the parallel data word at the input into small serial data packets in the order specified.
SMC Puncture	Removes user-specified bits from the input data stream
SMC Pulse Generator	Generates a single pulse.
SMC Ramp	Creates a ramp based on increments derived from a port or parameter
SMC Random	Creates a random integer of the requested word length
SMC Recast	Provides a value, based on the requested data type cast at the output and maintaining the same bits as provided at the input
SMC Register	Inserts a delay.
SMC RFIR	Custom block that implements a reloadable finite impulse response FIR filter.
SMC Sequence	Repeats a sequence of specified data
SMC Serial to Parallel	Implements a data packet combiner that collects serial data packets at the input and merges them into a parallel data word at the output.

SMC Sign	Provides the 2-bit sign value (+1 or -1) for the input.
SMC Single Clock Downsample	Provides variable rate and single clock downsample operations.
SMC Single Clock Upsample	Provides variable rate and single clock upsample operations.

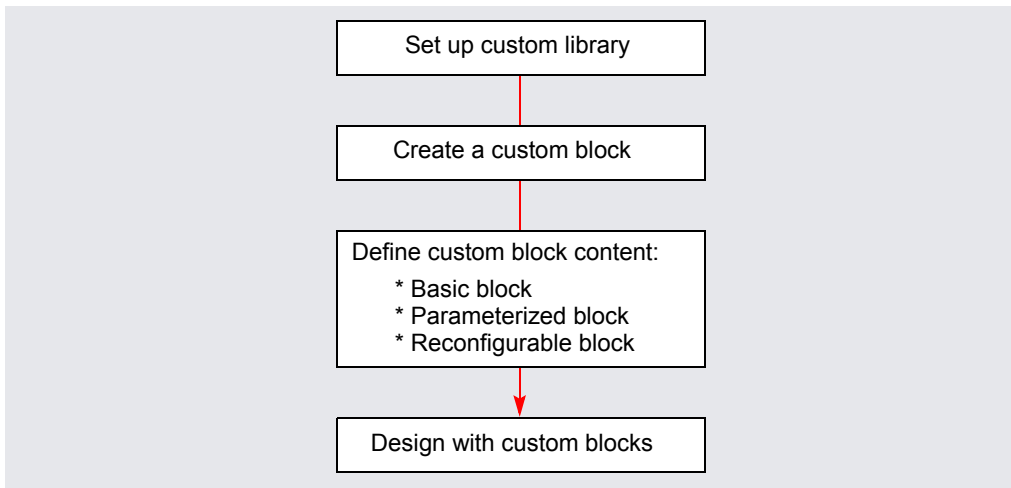
Library of Custom Block Examples

The tool `<install_dir>\mathworks\toolbox\Synopsys\SynphonyHLS\demos\APPEX` directory includes examples of custom blocks. For the most current set of custom block examples, refer to SolvNet article 030247, *"Synphony Model Compiler Custom Library Examples."*

Design Flow for Building Custom Blocks

This is a mini-tutorial that runs through an example to show you how to create custom blocks using Symphony Model Compiler blocks. For a definition of custom blocks, see [Primitives and Custom Blocks, on page 800](#).

The following figure illustrates the steps in the tutorial and shows the different kinds of custom blocks you can create.



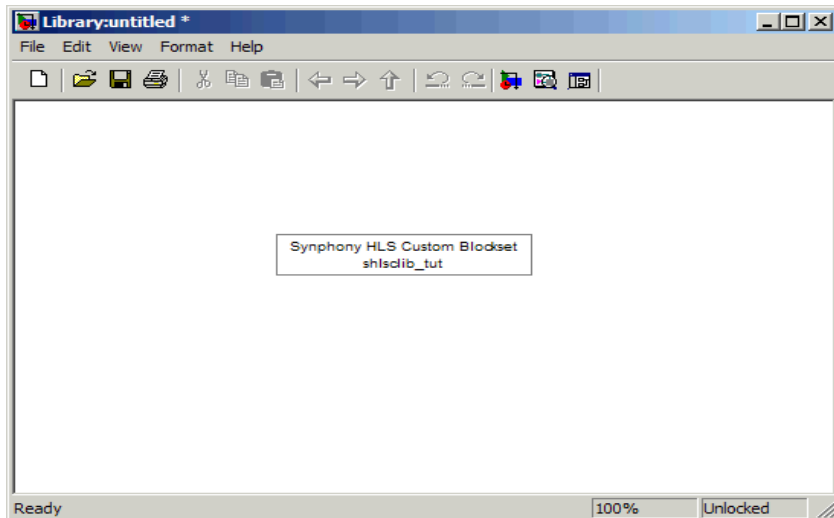
The design flow above is described in the following sections:

- [Set up a Custom Library, on page 805](#)
- [Create a Custom Block, on page 806](#)
- [Define Basic Content for Custom Blocks, on page 812](#)
- [Define Content for Parameterized Blocks, on page 816](#)
- [Define Content for Reconfigurable Blocks, on page 820](#)
- [Designing with Custom Blocks, on page 823](#)
- [The MySign M-Generator, on page 826](#)

Set up a Custom Library

The first step is to set up a custom library where you can group your custom blocks. The following procedure illustrates the steps.

1. Open the Simulink Library Browser, and select File->New->Library. This opens a schematic window that you use to capture the library elements.
2. In the library schematic window, do the following:
 - Double-click in the window. In the resulting text box, type a description for the custom library you are creating.



- Select File->Save, and name the library shlsclib_tut.mdl. All custom libraries must be named shlsclib<string>.mdl. Do not name it shlslib, as this name is reserved for the main Synphony Model Compiler library.
- Save the library file to a location in your Synphony installation hierarchy.
- If necessary, select Edit->Unlock Library. When you first save or open a Simulink library, the lock protects it from accidental changes. If you intend to modify the file, you must unlock the library.

3. At the MATLAB prompt, type `rehash toolboxcache`

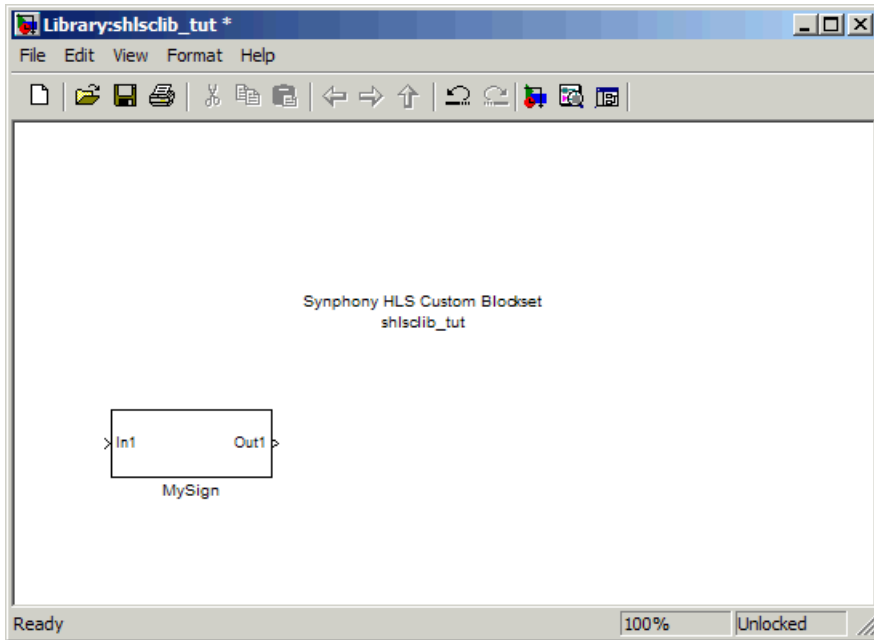
This command rescans all the toolbox directories for new files and updates the MATLAB cache file. You only need to run the `rehash` command once. At subsequent sessions you do not need the command, but can open the library by just typing its name at the MATLAB prompt. You can now create a custom block as described in [Create a Custom Block, on page 806](#).

Once you have set up a custom library, you can ensure that you can use it with different software versions by following the techniques described in [Maintaining Custom Libraries, on page 824](#).

Create a Custom Block

This step-by-step procedure shows you how to create a subsystem and mask for a custom MySign block. You mask the block to make it suitable for inclusion in a library. Masking the block ensures that you can treat the block as a primitive when you use it in your design.

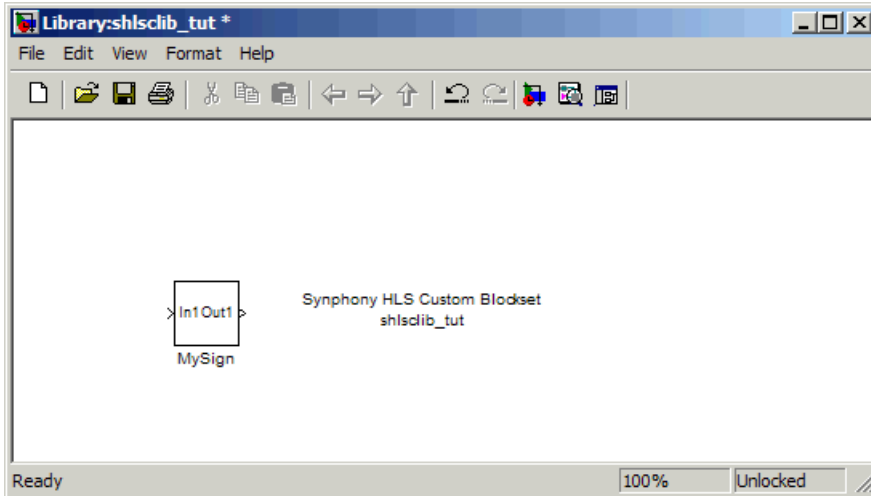
1. After you have created the custom library, create a subsystem for the custom block.
 - Drag a Subsystem block from the Simulink Simulink->Ports & Subsystems library into the library schematic window. The Subsystem block provides the starting point for the block.
 - Close the Simulink library window.
 - In the custom library window, rename the instance by double-clicking the name and typing a new name; in this case, type MySign.



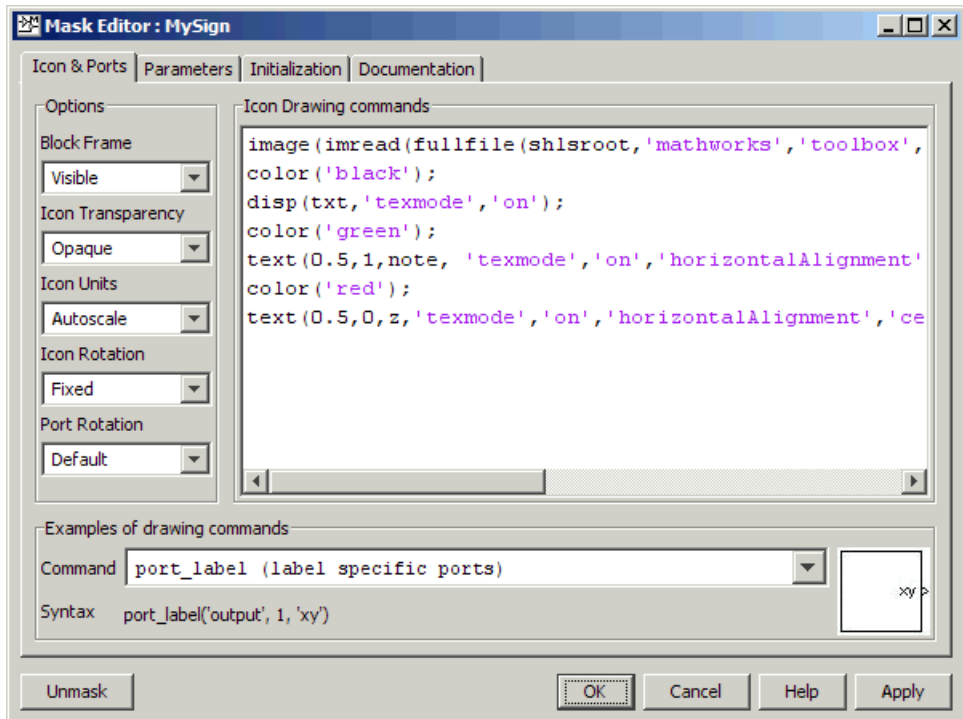
- Position and size the block by typing the following at the MATLAB command prompt:

```
set_param('shlsclib_tut/MySign','Position',[100 100 140 140])
```

This positions the instance at $x=100$, $y=100$, and sets the block to a standard size of 40 x 40 pixels. For positioning, it is a good practice to put the origins of different blocks on a grid of 100 x 100 pixels. The Synopsys blocks use a standard size of 40 x 40 pixels for a one input, one output block. For each additional port, add 20 pixels to the height. The width can remain 40 pixels unless the port names necessitate an increase in width. Thus, the standard width is 40 pixels, and the standard height is $\min(40, 20 * (\max(\text{inputs}, \text{outputs})))$ pixels.



2. Right-click MySign and select Mask subsystem from the popup menu. This opens the Mask editor window, where you can set up the mask.
 - On the Icon & Ports tab, set the display to be used for the icon by typing the commands in the Drawing commands area. You can see a list of available commands by clicking in the Command field. Consult the Simulink documentation for syntax details.



The following table shows the commands specified for the MySign block. Note that no port labels are defined because they are obvious.

Command	Effect
<code>image(imread(fullfile(shlsroot,'mathworks','toolbox', 'Synopsys','SynphonyHLS','icons','synplicity40_fg.jpg')), center');</code>	Puts the specified logo image on the icon.
<code>color('black'); disp(txt,'texmode','on');</code>	Sets the color of text in the txt variable. The variable itself is defined on the Initialization tab.
<code>color('green'); text(0.5,1,note, 'texmode','on','horizontalAlignment', 'center', 'verticalAlignment','top');</code>	Defines a placeholder for a note variable. The note is defined on the Initialization tab.

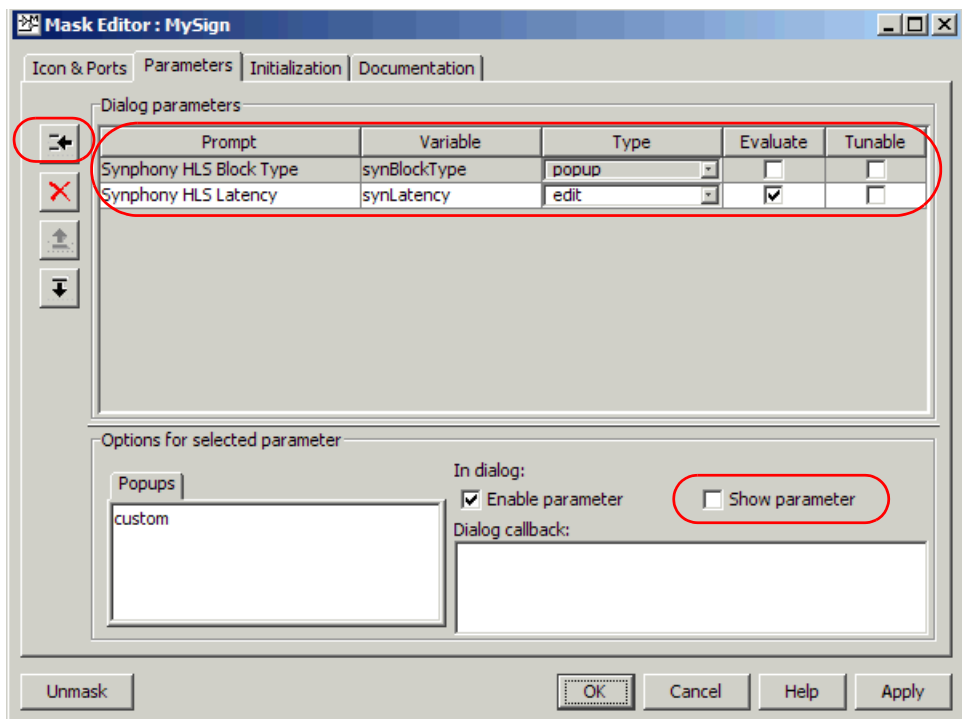
Command

```
color('red');
text(0.5,0,z,'texmode','on','horizontalAlignment',
    'center','verticalAlignment','bottom');
```

Effect

Defines a placeholder for the Z (latency) variable. The latency is defined on the Initialization tab. If the latency is 0, nothing is displayed.

3. Select the Parameters tab and define the synBlockType and synLatency parameters. For the MySign block, do the following:
 - Click the Add icon on the left to add a line in the Dialog parameters area.
 - On this line define the synBlockType parameter with Type set to popup and Evaluate and Tunable disabled. Type custom in the Popups area at the lower left, and disable Show Parameter.

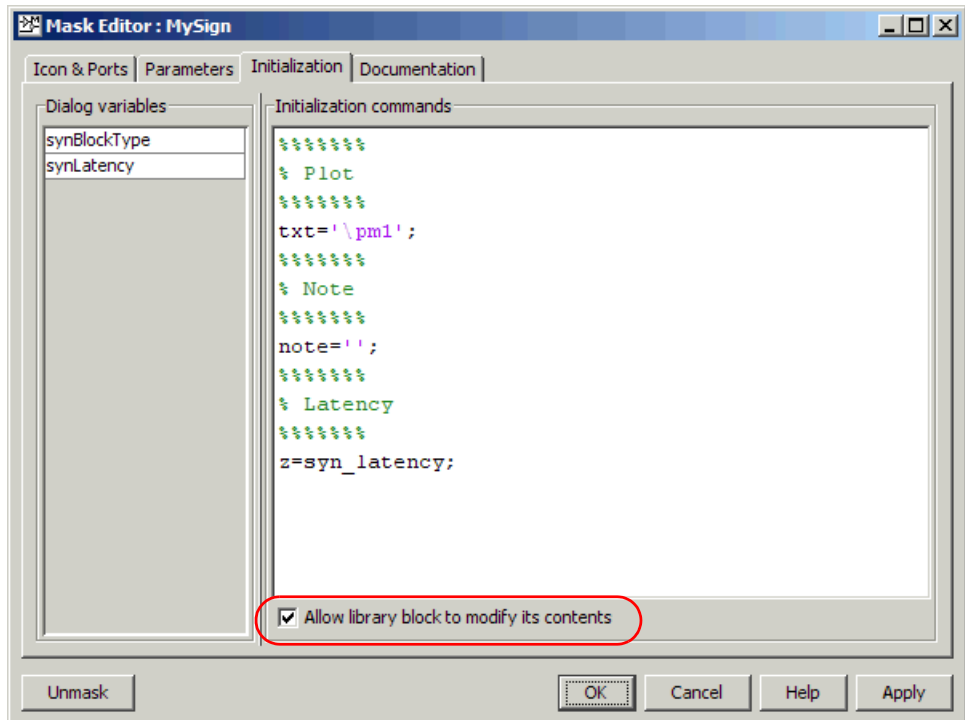


- Add another parameter line to define `synLatency`, to hold the potential latency of the block. Disable the Tunable and Show Parameters checkboxes, as shown in the figure

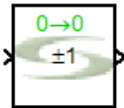
Setting parameters causes the underlying block schematics to be hidden from the designer in the Library Browser.

4. Select the Initialization tab and specify the initialization code for the variables defined on the Icon tab. For the MySign block, do the following:
 - In the Initialization commands area, define the `txt`, `note`, and `z` variables with the following commands:

```
txt='\pm1';  
note='';  
z=syn_latency;
```
 - Enable Allow library block to modify its contents.



5. Select the Documentation tab, and do the following:
 - Fill out Mask description, with a one-line description of the block functionality; for example, The MySign block provides the 2-bit sign value (+1 or -1) for the block. If you do not have parameters defined, you must fill out this field to ensure that the block appears as a non-hierarchical primitive in the Simulink library browser. If your custom block has parameters, you do not need to fill out the description, but it is a good practice.
 - Type a descriptive name in Mask Type. For example: Symphony HLS MySign.
6. Save the settings.
 - Click OK in the mask editor window.
 - Select File->Save in the library window. The icon in the window now reflects the changes you made.



You can now define the content for the blocks. You can create complex blocks like parameterized or reconfigurable blocks. For the purposes of this tutorial, create a basic block first ([Define Basic Content for Custom Blocks, on page 812](#)).

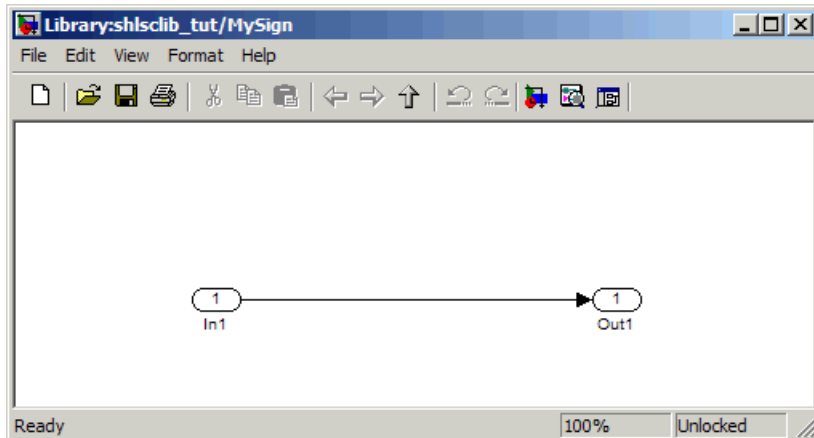
Define Basic Content for Custom Blocks

After creating a custom library and block (see [Set up a Custom Library, on page 805](#) and [Create a Custom Block, on page 806](#)), you still have to define the contents of the block. The following procedure shows you how to group Symphony Model Compiler primitives to create a higher-level function (MySign) as a library component.

1. Make the custom block editable. In the custom library window, right-click the custom block and select Look under mask.

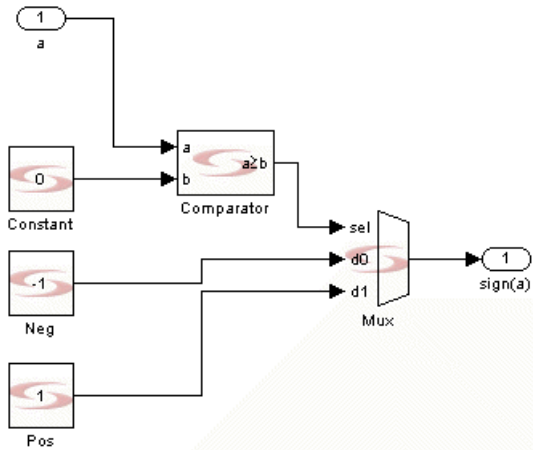
A window opens with the initial content of the masked block, which is one input port and one output port. If you do not use the Look under mask command, you will not be able to see or edit any content when you double-click the block because it has been masked.

2. In the window with the block contents, type new port names for the block. The following figure shows the default names changed to a and sign(a) for the MySign block.



3. Create the contents of the block using primitives from the Symphony library or with other Symphony custom blocks.

The following figure shows the MySign block defined with $\text{sfix2_En0}(+1)$ on the output for inputs larger than or equal to 0, and $\text{sfix2_En0}(-1)$ for inputs smaller than 0.

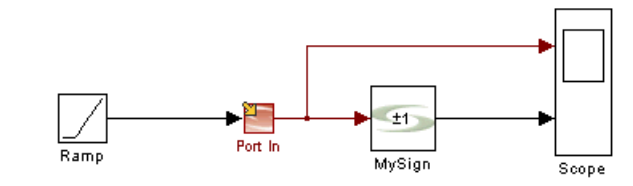


4. In the library window, select File->Save to save the contents of the block.

This procedure creates a static block where primitives are grouped together for a higher-level function. For information about more flexible blocks, see [Define Content for Parameterized Blocks, on page 816](#) and [Define Content for Reconfigurable Blocks, on page 820](#).

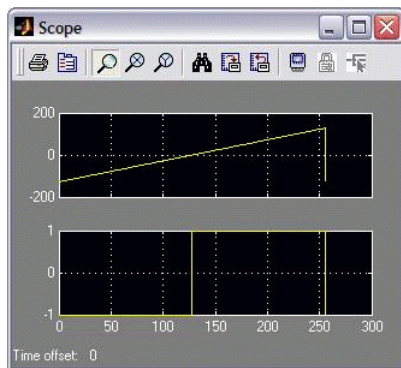
5. Test the block.

- Create a simple design that uses the block, and save it as MySign_test.



- Double-click the Ramp block and set Initial output to -2^7 . Click OK.
- Double-click Port In and set Sample time to 1. Click OK.
- Simulate the design for 2^8 cycles.

- Check the results. You should see the following:

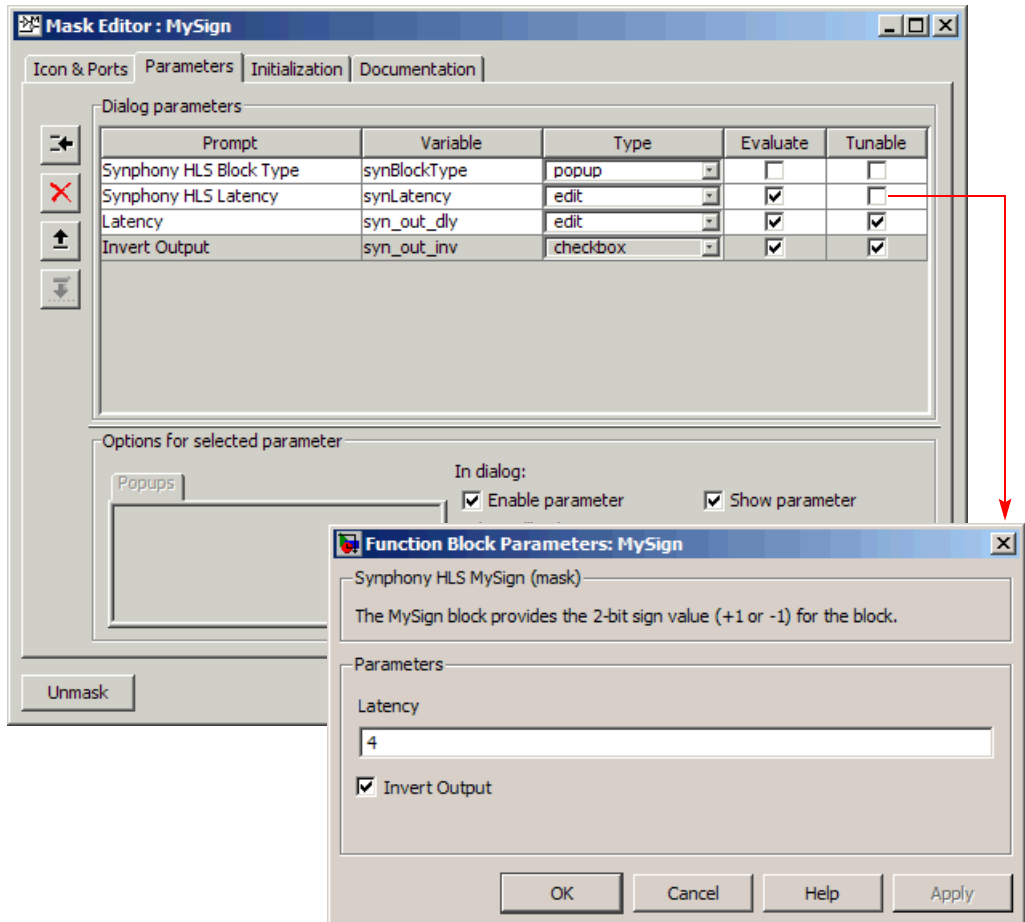


Define Content for Parameterized Blocks

Parameterized blocks include parameters in the block mask, which allow you to fine-tune some options. The following example starts with the MySign block you created earlier (see [Create a Custom Block, on page 806](#)), and adds parameters for specifying latency and reversing the output.

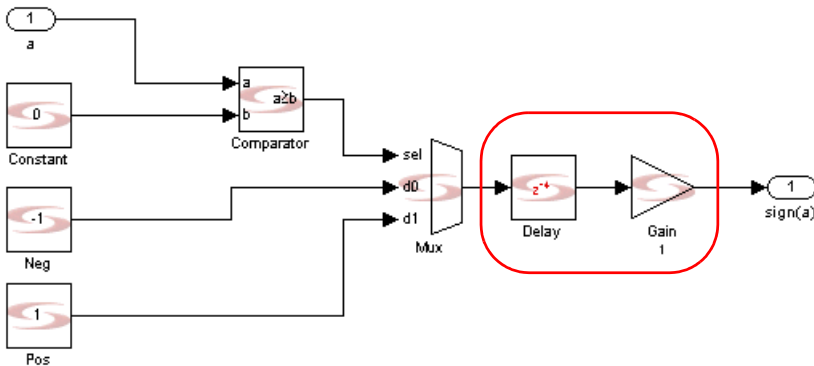
1. Open the block.
 - At the MATLAB prompt, type `shlsclib_tut` to open the window with the custom blockset.
 - From the library window, select Edit->Unlock Library. This allows you to edit the blockset.
 - Right-click the MySign block and select Edit Mask. This opens the mask editor.
2. Set mask parameters in the mask editor, as follows:
 - Click the Parameters tab and add the `syn_out_dly` parameter, specifying it as a text string. This parameter captures the desired latency for the custom block.
 - Add the `syn_out_inv` parameter, and specify it as a checkbox. This determines whether the output is inverted.
 - Make sure that the Show Parameter option is on for both these parameters.
 - Click OK in the mask editor.
3. Set parameter defaults.
 - In the library window, double-click the MySign block to open the Block Parameters: MySign dialog box. This box shows the parameters you defined in the previous step.
 - Enter 4 for Latency.
 - Enable the Invert Output option.
 - Click OK.

The changes are made inside the custom blockset, and these settings are inherited by all instances of the MySign block as defaults.



4. Edit block content to add the blocks required to support parameterization.
 - Right-click the MySign block and select Look under mask. This opens a window with the contents of the block.
 - Add a Delay block after the mux to provide the desired latency.
 - Double-click the Delay block. In the Delay field of the dialog box, enter `syn_out_dly`. Click OK. This applies the parameterized delay to the MySign block.
 - Add a Gain block after the Delay block to support inversion.

- Select File->Save and save the changes made to the block.



5. Program the response to the parameters. You need to do this so that the `syn_out_inv` parameter determines the gain of the Gain block.

- Go back to the `shlsclib_tut` window. Right-click the `MySign` block and select `Edit mask`. This opens the mask editor.
- Select the `Initialization` tab, and edit the code as follows:

```

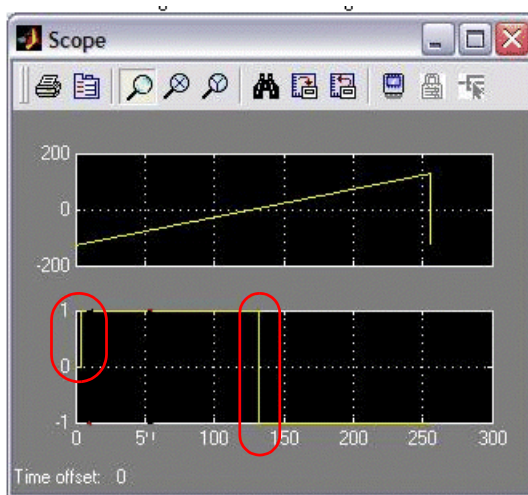
%%%%%%%%%
% Plot
%%%%%%%%%
txt='\pml';
%%%%%%%%%
% Note
%%%%%%%%%
% See Parameterization Section
%%%%%%%%%%
% Latency
%%%%%%%%%%
set_param(gcb,'synLatency',num2str(syn_out_dly));
z=syn_latency;
%%%%%%%%%%%%%%%%%%%%%%%%
% Parameterization
%%%%%%%%%%%%%%%%%%%%%%%%
gainBlock=[gcb '/Gain'];
switch syn_out_inv
case 0
    note='\rightarrow';

```

```
set_param(gainBlock, 'syn_gain_val', '1');  
case 1  
    note='\rightarrow\circ';  
    set_param(gainBlock, 'syn_gain_val', '-1');  
end
```

Note that the `syn_out_inv` parameter controls the `note` variable to differentiate the appearance of the MySign block. It is also used in the Parameterization section to determine the gain of the Gain block. The `syn_out_dly` parameter sets the `synLatency` parameter of the MySign block.

- Click OK. The icon for the MySign block reflects the changes you made.
6. Test your block.
- Open the design you created in [Define Basic Content for Custom Blocks, on page 812](#).
 - Select Edit->Update diagram to make sure that the design uses the updated parameterized block. The icon changes to reflect the parameters.
 - Simulate the design.
 - Check the results. It should show that the output is inverted and delayed by 4 samples.



Define Content for Reconfigurable Blocks

A reconfigurable block provides even more flexibility than a parameterized block (see [Define Content for Parameterized Blocks, on page 816](#)), by allowing different input or output permutations or making the content dependent on block parameters.

Reconfigurable blocks differ from basic blocks ([Define Basic Content for Custom Blocks, on page 812](#)) and parameterized blocks in the way that they are implemented. Reconfigurable blocks use the M-generator instead of the Icon tab on the mask editor to determine the display.

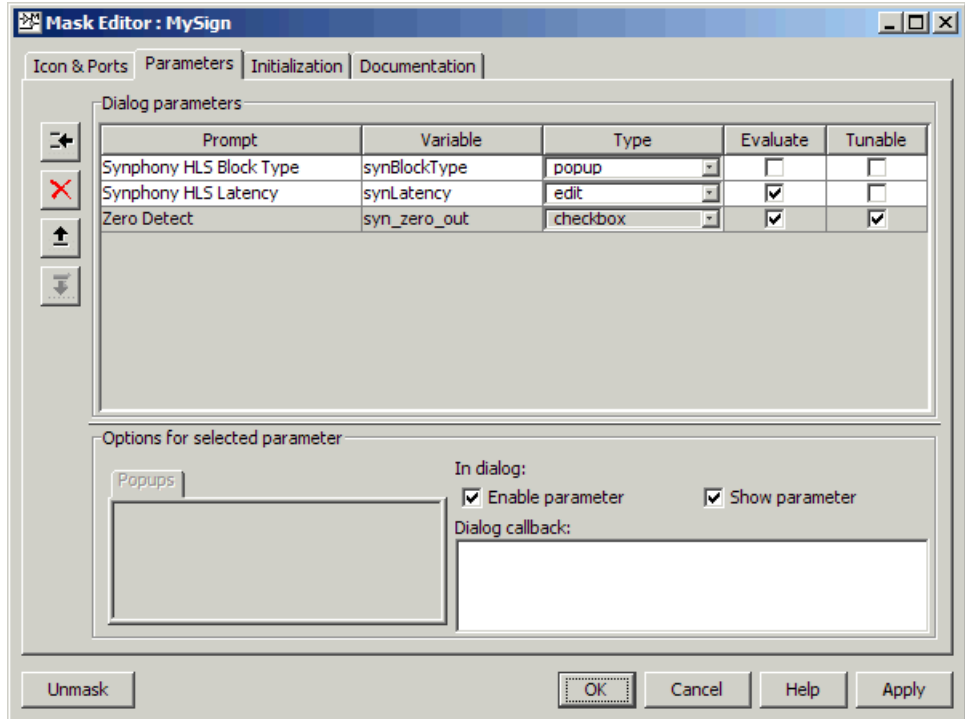
1. Set icon size.

- At the MATLAB prompt, type `shlsclib_tut` to open the window with the custom blockset.
- From the library window, select Edit->Unlock Library. This allows you to edit the blockset.
- At the MATLAB prompt, type the following command, which resizes the icon width to 60 pixels:

```
set_param('shlsclib_tut/MySign','Position',[100 100 160 140])
```

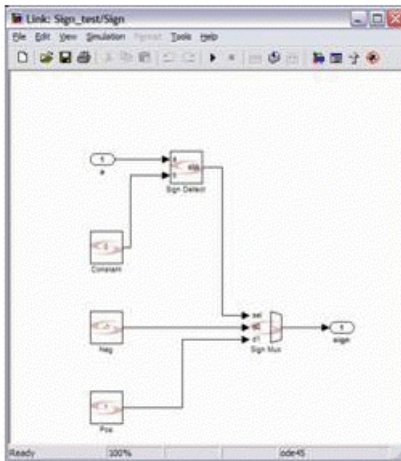
2. Provide an M-generator mask (see [The MySign M-Generator, on page 826](#) for an example of an M-generator for the MySign block). When you have an M-generator mask, the tool uses the M-generator to initialize the icon, instead of the settings on the Icon tab.

- Right-click the MySign block and select Edit Mask.
- On the Icon tab, delete all the drawing commands.
- On the Parameters tab, set the `syn_zero_port` variable, which reconfigures the optional zero port. You should have `synBlockType`, `synLatency` and `syn_zero_port`.

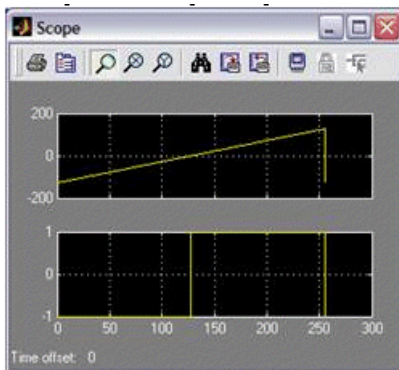


- On the Initialization tab, delete all the code and type `syn_mysign_init`; This is a call to the M-generator, which is called `syn_mysign_init`. See [The MySign M-Generator, on page 826](#) for the code.
 - Click OK.
 - Select Edit->Save in the library window to save the changes.
3. Put the M-generator in the path. See [The MySign M-Generator, on page 826](#) for the code.
- It is recommended that you put it in the same location as the custom blockset.
4. Test your block.
- Open the design you created in [Define Basic Content for Custom Blocks, on page 812](#).

- Select Edit->Update diagram to make sure that the design uses the updated reconfigurable block and save the design. The icon changes to reflect the parameters.
- Right-click the block and select Look under Mask. This shows the contents as determined by the M-generator.



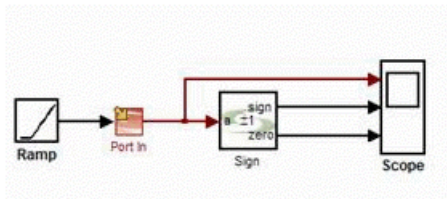
- Simulate the design and check the results. The results show basic sign detection.



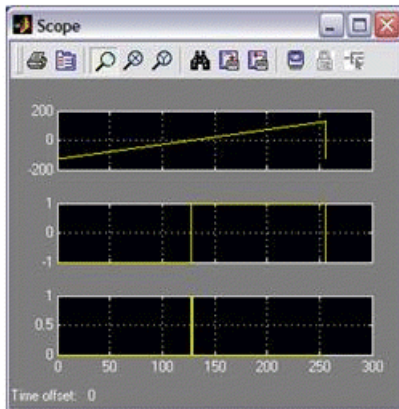
5. Run another test with zero detection.

- Double-click the MySign block to open the Block Parameters dialog box.

- Enable the Zero Detect option and click OK. The MySign instance now has an extra output port, zero.
- Edit the scope to track the extra output port as well.



- Simulate and check the results. The results show basic sign detection combined with zero detection.



Designing with Custom Blocks

After creating custom blocks and libraries as described in the previous section, incorporate them in your design using the following procedure.

1. The first time you install the library, make the library available for use by typing the following at the MATLAB command prompt:

```
rehash toolboxcache
```

On subsequent sessions, you can skip this step.

2. Load the custom library.

- Close any open Simulink library browsers and open an updated browser by typing `simulink` at the MATLAB command prompt. The browser now contains an entry for the new custom library: Symphony HLS Custom Blockset `shlsclib`.
- Double-click the entry to display the Max block in the right pane.

If your custom library is not listed, make sure that you saved the library to the specified directory using the appropriate naming convention. Repeat step 1 to update the Simulink library browser.

3. Follow the usual Symphony Model Compiler procedures when you instantiate and use the custom block in your design.

4. Generate RTL.

In the Symphony output files, the functionality of the custom block is automatically resolved to the main primitives, so that the design can be synthesized.

5. Synthesize your design as usual.

Maintaining Custom Libraries

This section describes the following techniques for maintaining custom libraries:

- [Maintaining Independent Custom Libraries](#), next
- [Converting Custom Libraries](#), on page 825

Maintaining Independent Custom Libraries

To ensure that your custom library remains independent of new versions of the software, maintain the blocks in a directory outside the software tree. The following procedure illustrates.

1. Create a directory outside the software tree with the custom libraries.

For example: `C:/Program Files/Synopsys/Symphony_lib/shlsclib*.mdl`

2. In the MathWorks release, add this directory to your path:

```
matlabroot/toolbox/local/startup.m:  
addpath(fullfile(shlsroot, '..', 'Synphony_lib'));
```

3. In the M-generator for a Synphony custom library block, use shlslib to determine the current release of the library:

```
syn_lib=shlslib('info');  
...  
add_block([syn_lib '/'<name_of_your_block>'],...);
```

This will make the custom libraries release-independent.

Converting Custom Libraries

This procedure shows you how to update a custom library for use with a newer version of the software. Use the following procedure to convert the shlslib_XXX.mdl libraries that are in MATLAB path.

1. Update blocks manually if needed.
 - If your custom block in the library uses an M script to initialize its contents, manually convert the static library references in the M script.
 - If your custom block in the library uses an M script to initialize its contents, manually convert the script to accommodate any Synphony Model Compiler blocks whose parameters have changed in the latest release. You can access the parameters from the mask editors for the blocks. You must do this because there could be block enhancements from release to release.
2. Run the syn_update_clib script.

When the Simulink library is launched, this script finds all the custom libraries in the MATLAB path for custom libraries, and converts them to be compatible with the current software version. It renames all legacy libraries OBSOLETE_libraryname. Automatic conversion only converts libraries starting with shlslib.

The MySign M-Generator

This is the code for the `syn_mysign_init.m` file.

```
% Store current configuration

syn_gcb=gcb;
syn_gcbh=gcbh;

% Erase content
% Erase all lines (delete_line works on array)
% Erase all blocks (delete_block does not work on array)
% The 'Parent' criterion is necessary to avoid deleting the
% current block.
% Ports are not erased to maintain existing connectivity if
% possible (erasing a port will disconnect any signal, even if the
% port is recreated).

delete_line(find_system(syn_gcbh,...
    'FollowLinks','on',...
    'LookUnderMasks','all',...
    'SearchDepth',1,...
    'FindAll','on',...
    'Type','line'));

syn_handles=find_system(syn_gcbh,...
    'RegExp','on',...
    'FollowLinks','on',...
    'LookUnderMasks','all',...
    'SearchDepth',1,...
    'Parent',syn_gcb);

for i=1:length(syn_handles)
    syn_handle=syn_handles(i);
    syn_bt=get_param(syn_handle,'BlockType');
    if strcmp(syn_bt,'Inport') || strcmp(syn_bt,'Outport')
        syn_name=get_param(syn_handle,'Name');

        % Create a variable to represent the existence of this port,
        % holding the handle to the port
        eval(['syn_' syn_name '_h=syn_handle;']);
    else
        delete_block(syn_handle);
    end
end
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize icon
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

licon=strcat('fullfile(shlsroot','mathworks','toolbox','...',
    'Synopsys','SynphonyHLS','icons','synplicity40_fg.jpg'));
synDisplay=sprintf('image(imread(%s),'center');',licon);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Input ports
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ipTot=1;
ipNames{1}='a';
if exist('syn_a_h','var')
    set_param(syn_a_h,...
        'Position',[100 103 130 117],...
        'Port',num2str(ipTot));
else
    add_block('built-in/Inport',[syn_gcb '/a'],...
        'Position',[100 103 130 117],...
        'Port',num2str(ipTot));
end

for n=1:ipTot
    synDisplay=strvcat(synDisplay,...
        strcat('port_label(''input'',',...
        sprintf('%d',''s'',n,ipNames{n})),...
        '','texmode',''on'');');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Output ports
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

opTot=1;
opNames{1}='sign';
if exist('syn_sign_h','var')
    set_param(syn_sign_h,...
        'Position',[400 313 430 327]);
else
    add_block('built-in/Outport',[syn_gcb '/sign'],...
        'Position',[400 313 430 327],...
        'Port',num2str(opTot));
end

```

```

if (syn_zero_port)
    opTot=opTot+1;
    opNames{opTot}='zero';
    if ~exist('syn_zero_h','var')
        % Create port
        add_block('built-in/Outport',[syn_gcb '/zero'],...
            'Position',[400 203 430 217],...
            'Port',num2str(opTot));
    else
        set_param(syn_zero_h,...
            'Position',[400 203 430 217],...
            'Port',num2str(opTot));
    end
else
    if exist('syn_zero_h','var')
        % Delete port
        delete_block(syn_zero_h);
    end
end

for n=1:opTot
    synDisplay=strvcat(synDisplay,...
        strcat('port_label(''output'',',...
        sprintf('%d','%s','',n,opNames{n}),...
        ','''texmode''',''on''');');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Content
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_block('shlslibv1/Sources/Constant',[syn_gcb '/Zero'],...
    'Position',[100 200 140 240],...
    'syn_cst_val','0',...
    'syn_cst_wl','2',...
    'syn_cst_fl','0',...
    'syn_cst_dt','signed');

add_block('shlslibv1/Sources/Constant',[syn_gcb '/Neg'],...
    'Position',[100 300 140 340],...
    'syn_cst_val','-1',...
    'syn_cst_wl','2',...
    'syn_cst_fl','0',...
    'syn_cst_dt','signed');

```



```

add_block('shlslibv1/Sources/Constant',[syn_gcb '/Pos'],...
    'Position',[100 400 140 440],...
    'syn_cst_val','1',...
    'syn_cst_wl','2',...
    'syn_cst_fl','0',...
    'syn_cst_dt','signed');

add_block('shlslibv1/Math Functions/Comparator',...
    [syn_gcb '/Sign Detect'],...
    'Position',[200 100 240 140],...
    'syn_comp_opr','a>=b');

add_line(syn_gcb,'a/1','Sign Detect/1','autorouting','on');
add_line(syn_gcb,[140 220; 150 220; 150 130; 200 130]);

add_block('shlslibv1/Signal Operations/Mux',...
    [syn_gcb '/Sign Mux'],...
    'Position',[300 290 340 350],...
    'syn_in_nb','2');

add_line(syn_gcb,'Sign Detect/1','Sign Mux/1','autorouting','on');
add_line(syn_gcb,'Neg/1','Sign Mux/2','autorouting','on');
add_line(syn_gcb,'Pos/1','Sign Mux/3','autorouting','on');
add_line(syn_gcb,'Sign Mux/1','sign/1','autorouting','on');

if (syn_zero_port)
    add_block('shlslibv1/Math Functions/Comparator',...
        [syn_gcb '/Zero Detect'],...
        'Position',[300 190 340 230],...
        'syn_comp_opr','a==b');
    add_line(syn_gcb,'a/1','Zero Detect/1','autorouting','on');
    add_line(syn_gcb,[150 220; 300 220]);
    add_line(syn_gcb,'Zero Detect/1','zero/1','autorouting','on');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

txt='\pml';
synDisplay=strvcat(synDisplay,...
    'color('black')';...
    'disp(txt','texmode','on');');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Note
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

note='';
synDisplay=strvcat(synDisplay,...
    'color('green');',...
    strcat('text(0.5,1,note,'texmode','on','',...
        ''horizontalAlignment','center','',...
        ''verticalAlignment','top');'));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Latency
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

z=syn_latency;
synDisplay=strvcat(synDisplay,...
    'color('red');',...
    strcat('text(0.5,0,z,'texmode','on','',...
        ''verticalAlignment','bottom');'));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Display Icon
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

set_param(syn_gcb,'MaskDisplay',synDisplay);

```

CHAPTER 10

Analyzing and Verifying the Design

This chapter describes how to analyze, simulate, and cosimulate your Symphony Model Compiler design to ensure it is correct:

- [Using Quantization Analysis Tools, on page 832](#)
- [Using Smart Black Boxes for Cosimulation, on page 837](#)
- [Simulating HLS Subsystem Blocks, on page 844](#)

Using Quantization Analysis Tools

The Symphony tool uses the Simulink fixed-point data type to represent the discrete amplitude of the signals in the Symphony blockset. For background information about this data type in the Symphony Model Compiler tool, see [Fixed-Point Data Type, on page 696](#). This section describes how to use the Symphony Model Compiler SynFixPtTool block and the Simulink Fixed-Point Tool interface:

- [Specifying Fixed-Point Options, on page 832](#)
- [Validating Algorithms with the Fixed-Point Toolbox, on page 834](#)
- [Using Plots, on page 835](#)

Specifying Fixed-Point Options

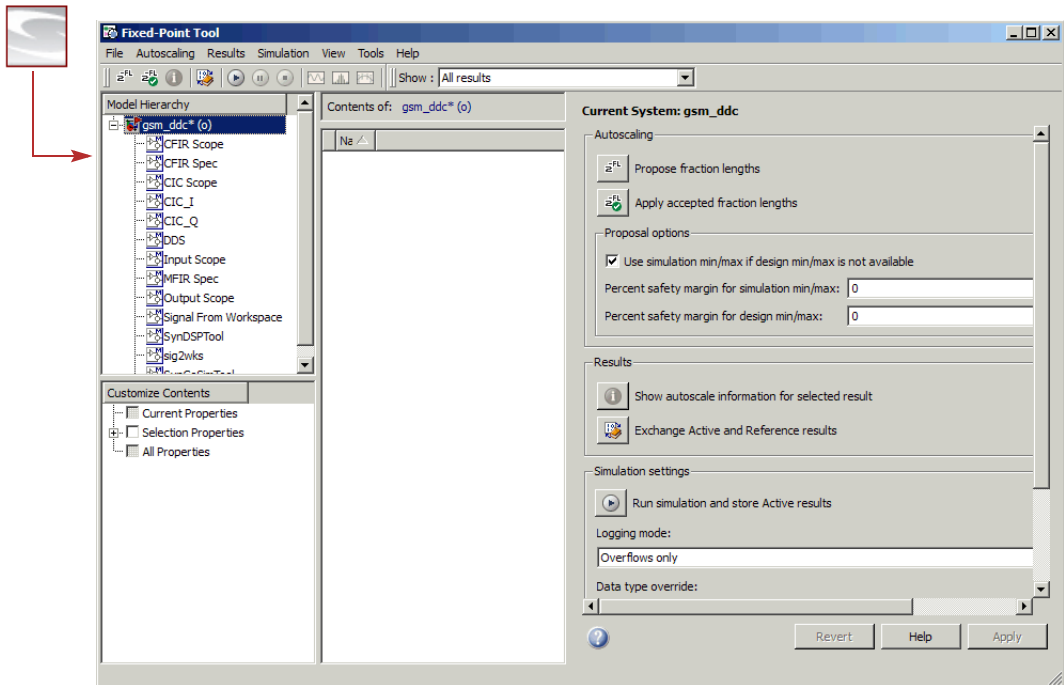
To use the fixed-point functionality for conversion before simulation, use the link to the Simulink fixed point functionality, as described below:

1. Add the Symphony SynFixPtTool block to your design.

This block leverages the existing Simulink fixed point functionality by providing a link to it.

2. Open the Simulink Fixed-Point Settings tool using one of these methods:
 - Double-click the SynFixPtTool block.
 - Right-click in the background of the Simulink schematic, and select Fixed-Point Settings from the menu.
 - From the Simulink schematic menu bar, select Tools->Fixed-Point Settings.

Any of these actions opens the Simulink Fixed-Point Tool interface, which provides convenient access to global data type overrides and logging settings. For information about this toolbox, type `doc fxptdlg` at the MATLAB prompt.



3. The Model Hierarchy pane displays a tree-structured view of the Simulink model hierarchy; the fixed-point tool controls the object selected in its Model Hierarchy pane.

You can use the Fixed-Point Tool interface for any system or subsystem.

4. Use the Simulation settings area of the settings pane to specify the fixed-point settings. For details, see [Validating Algorithms with the Fixed-Point Toolbox, on page 834](#) and [Using Plots, on page 835](#).

You can use all the toolbox features with the following exceptions, which the Symphony tool does not currently support:

- For the Logging mode option, you can only use Use local settings and Overflow Only. This option controls logging for the selected subsystems.
- For the Data type override option, the only valid choices are Use local settings and Scaled Doubles. This option controls data type overrides for the selected subsystems.

- You cannot currently use the Autoscale fixed-point blocks feature, which automatically changes the scaling for any block that does not have its scaling locked.

Validating Algorithms with the Fixed-Point Toolbox

Typically, you first simulate your design with full-accuracy calculations to validate the algorithm, and then simulate the fixed-point algorithm. The SMC tool lets you use some of the Simulink fixed point functionality. The following outlines the general procedure to use the Fixed-Point toolbox:

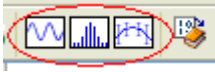
1. Set up your design.
 - Add the Symphony SynFixPtTool block to the design.
 - Add Simulink scopes to your design. To plot data, you must set up the Simulink time scopes to store data as described in [Using Plots, on page 835](#).
2. Double-click SynFixPtTool to open the Fixed-Point Tool interface.
3. To validate a full-accuracy algorithm, do the following:
 - Select the entire design or a particular block from the Model Hierarchy pane.
 - Set Logging mode to Overflow only in the Simulation settings area of the settings pane.
 - Set Data type override to Scaled Doubles in the Simulation settings area of the settings pane.
 - Simulate the design by clicking the Start button (right arrow icon). The software ignores the fixed-point settings and does a full-accuracy simulation.
4. Analyze the information in the scope windows to check the floating-point algorithm.
5. After validating the floating-point algorithm, validate the fixed-point algorithm by following these steps:
 - Select the entire design or a particular block from the Model Hierarchy pane.

- Set Logging mode to Overflow only in the Simulation settings area of the settings pane.
 - Enable finite word length effects by setting Data type override to Use Local Settings. This is the default mode where overflow is detected. Overflow is determined by the local fixed-point annotations.
 - Simulate the design by clicking the Start button (right arrow icon). The software simulates the design using the fixed-point settings and reports any overflow effects.
6. Analyze the information in the scope windows to check the fixed-point algorithm. You can now compare the data from the fixed-point and floating simulations, as described in [Using Plots, on page 835](#).

Using Plots

The Simulink Fixed-Point Tool interface has a plotting feature that you can use to compare the simulation results.

1. Before simulation, set up the Simulink time scopes to store data, as follows. Do this for all scopes that you want to plot:
 - Double-click on the scope to open the scope window.
 - Click the Parameters icon to open the Parameters window.
 - Click the Data History tab and disable Limit data points.
 - For all scopes that you want to plot, also enable Save Data to Workspace.
 - Click OK.
2. Validate your floating-point and fixed-point algorithms (see [Validating Algorithms with the Fixed-Point Toolbox, on page 834](#)).
3. Open the Simulink Fixed-Point Tool interface and select the scope with the data to be plotted from the Contents pane.
4. You can create any of the following types of plots using the Fixed-Point Tool interface:
 - Time-series plot
 - Histogram plot
 - Time-series difference (A -R) plot



For information about the plot interface, see *Plot Interface* on the Fixed-Point Tool help page.

5. To compare the full-accuracy simulation results with the quantized results, use a time-series difference (A -R) plot:
 - Select **Store All Active Results As Reference Results** to store the floating-point simulation results.
 - Use a time-series difference (A -R) plot to plot both the active and reference versions of a signal on the upper axes and to plot the difference between the active and reference versions of the same signal on the lower axes.

Using Smart Black Boxes for Cosimulation

This section describes how to use the Smart Black Box block for hardware cosimulation. A smart black box differs from a simple black box, because you have access to the RTL code for the IP. For information about implementing a simple black box, see [Using Black Boxes and Third-Party IP, on page 777](#).

This section describes the following:

- [Incorporating Smart Black Boxes in the Design, on page 837](#)
- [Configuring the Cosimulation Interface, on page 839](#)
- [Creating Smart Black Box Configuration Files, on page 841](#)
- [About Cosimulation with ModelSim, on page 842](#)

Incorporating Smart Black Boxes in the Design

1. Make sure you have EDA Simulator Link MQ (formerly Link for ModelSim)® installed and accessible.

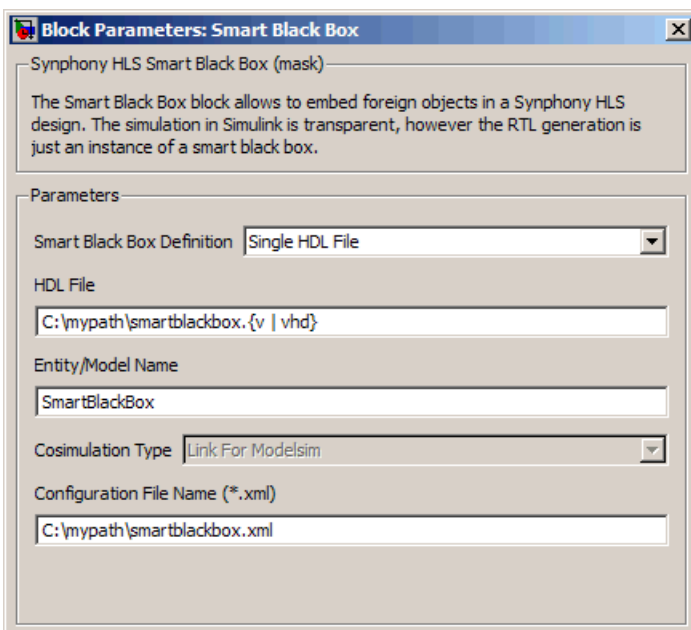
EDA Simulator Link MQ is a cosimulation interface between Simulink and the ModelSim® HDL simulator. Currently, Synphony Model Compiler uses this tool to verify and simulate the embedded RTL-level models. See [About Cosimulation with ModelSim, on page 842](#) for some background information.

2. Instantiate the SynCoSimTool block (top-level library) in your design and configure the cosimulation interface. See [Configuring the Cosimulation Interface, on page 839](#) for details.
3. Create a configuration file that contains port, clock, global enable and reset information for the smart black box. See [Creating Smart Black Box Configuration Files, on page 841](#) for details.
4. If the black box is defined in multiple files, create a text file that lists the absolute paths to all the HDL definition files. Skip this if you have a single-file definition.

This example creates a file called `sbblist.txt` that lists four black box definition files:

```
-I sbblib C:\mypath\sbblib1.vhd  
C:\mypath\sbb2.v  
C:\mypath\sbb3.v  
C:\mypath\sbb4.vhd
```

5. Instantiate the Smart Black Box block (Ports & Subsystems library) in your design.
6. Double-click the Smart Black Box block to open the parameters dialog box. See [SMC Smart Black Box, on page 532](#) for details about the block.
 - Specify the location of the black box definition file or files. If you have a single file, type the absolute path to it in Black box Definition. If you have multiple definition files, specify the absolute path to the text file you created (step 4) in Black box File List.
 - Specify the location of the configuration file you created (step 3) in Configuration File Name.
 - Type a name for the black box in Entity/Model Name. Click OK.



7. Run Symphony Model Compiler synthesis.

- Set the Symphony Model Compiler optimizations you want for the design. For details about the effect of different optimizations on smart black boxes, see [Using Optimizations with Black Boxes, on page 784](#).
- Synthesize the design and generate RTL.

The tool uses EDA Simulator Link MQ (formerly Link for ModelSim) to cosimulate and verify the embedded RTL-level models. The Simulink simulation is transparent.

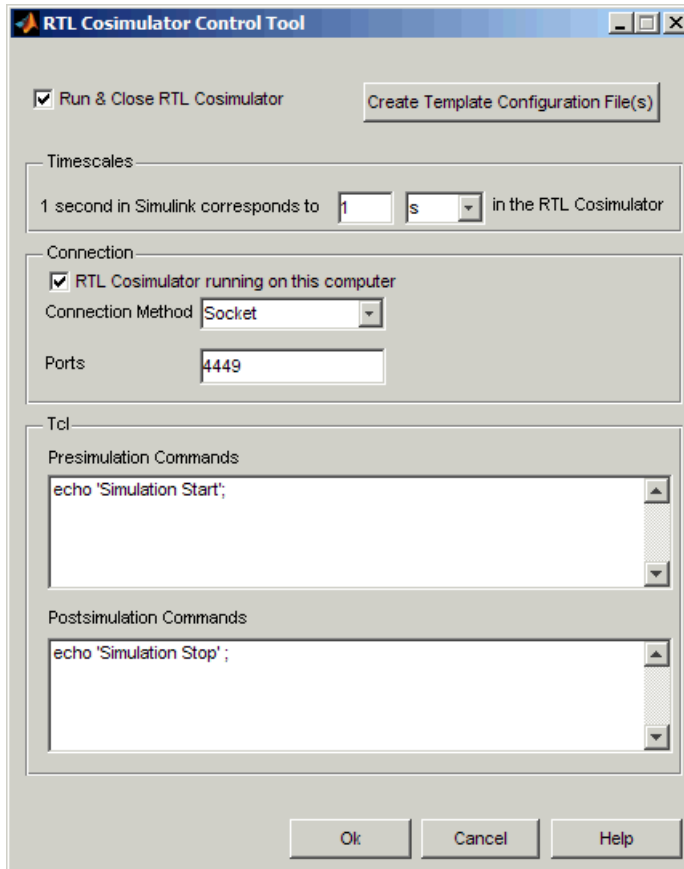
The generated RTL for the design includes an instance for the smart black box. The rest of the design is hooked up to the ports of the black box, with the appropriate connections for global enables, reset, and black box clocks you specified. Timing arcs stop at the input ports of the black box and resume from the output of the black box; they do not include the timing through the black box.

Configuring the Cosimulation Interface

This procedure shows you how to configure the cosimulation interface between EDA Simulator Link MQ and the DSP synthesis tool, so that you can use smart black boxes in your design, as described in [Incorporating Smart Black Boxes in the Design, on page 837](#).

1. Instantiate the SynCoSimTool block from the top-level Symphony Model Compiler library, and double-click it to open the parameters dialog box.

This dialog box lets you configure the cosimulation interface to EDA Simulator Link MQ. Do the next few steps in the dialog box. For detailed description of the dialog box options, refer to [SMC SynCoSimTool, on page 550](#).



2. Specify the location of the cosimulator and the connection method.
 - In the Connection section, specify the location of the cosimulator.
 - Specify the connection method.
 - Specify the socket connection port.
 - If you have more than one smart black box, set Connection Method to Socket and provide different port numbers for each smart black box.
3. Specify parameters for the run.
 - To have the cosimulator close down after a run and re-initialize every subsequent time it starts up, enable Run and Close RTL Cosimulator. The advantage to this setting is that the cosimulator is re-initialized at

every run; the disadvantage is longer run times because the cosimulator is initializing.

- To have the cosimulator remain open after a run, disable Run and Close RTL Cosimulator. Use this setting if your RTL source does not depend on initial state values because it offers the advantage of faster run times. The downside of this setting is that the cosimulator is only initialized once at the initial run. All subsequent runs use the same settings.

4. Optionally, specify pre-run and post-run Tcl commands.

These commands execute before or after the cosimulator run, as you specified.

5. Set any other options in the dialog box and click OK.

When you run DSP synthesis, the tool uses these configuration settings to run EDA Simulator Link MQ for cosimulation and verification of smart black boxes.

Creating Smart Black Box Configuration Files

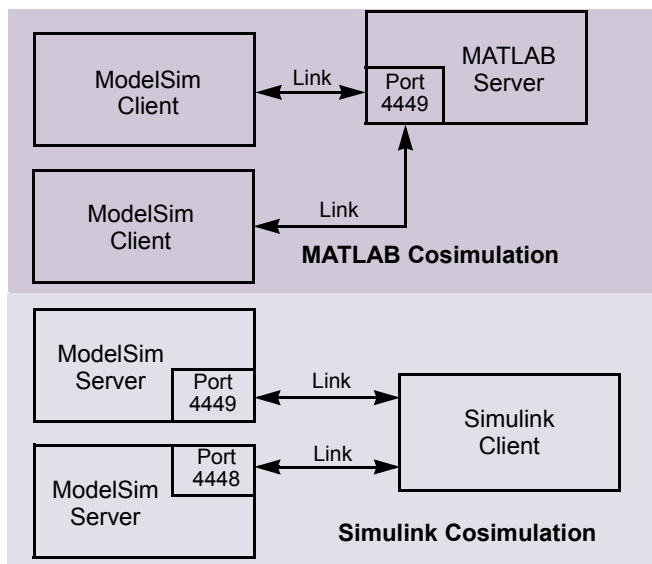
Smart black boxes require configuration files that define connection details like ports, clocks, global enables and resets. You must have this file to use smart black boxes in your design as described in [Incorporating Smart Black Boxes in the Design, on page 837](#).

You can create this xml file in one of two ways:

- Manually create the file in xml format, following the syntax and example described in [Configuration File For Smart Black Box, on page 535](#).
- Create and then edit a template file.
 - Instantiate and configure the SynCoSimTool block as described in [Configuring the Cosimulation Interface, on page 839](#).
 - Click Create Template Configuration File. This creates a template configuration file in the `../modelpath/synwork` directory with SBB block names.
 - Edit the xml template file to include port, clock, global reset and enable specifics for the smart black box. See [Configuration File For Smart Black Box, on page 535](#) for syntax.

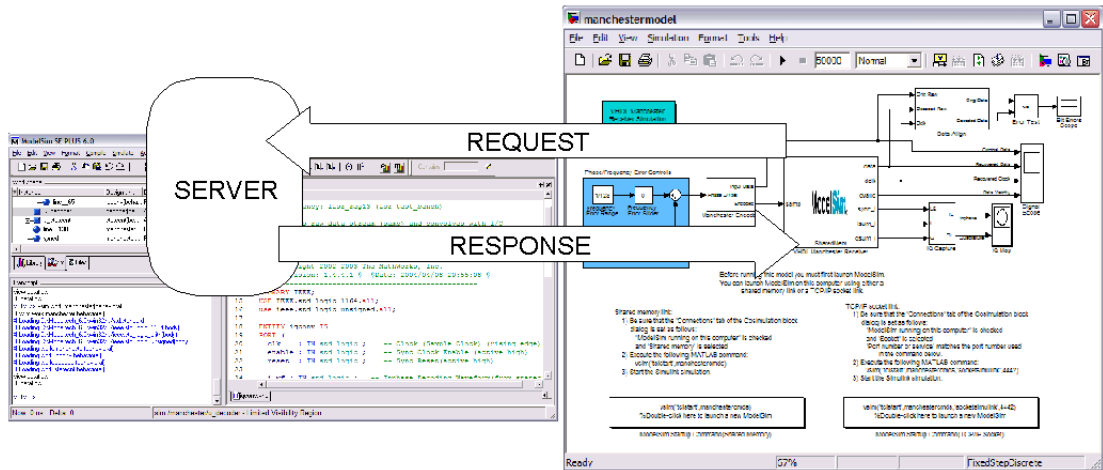
About Cosimulation with ModelSim

For cosimulation, the Symphony, Simulink (with EDA Simulator Link MQ), and ModelSim tools need to work together. You must have the EDA Simulator Link MQ interface software from MATLAB, which sets up the proper relationships between ModelSim and the MathWorks products.



This figure illustrates client-server relationships. The link between the simulators is either shared memory (optimal performance) or a TCP/IP socket (most versatile). One simulator is a server (responds to requests) and the other is a client (initiates simulation requests). Links between different MathWorks products differ. For Simulink cosimulation, ModelSim is the server and Simulink is the client.

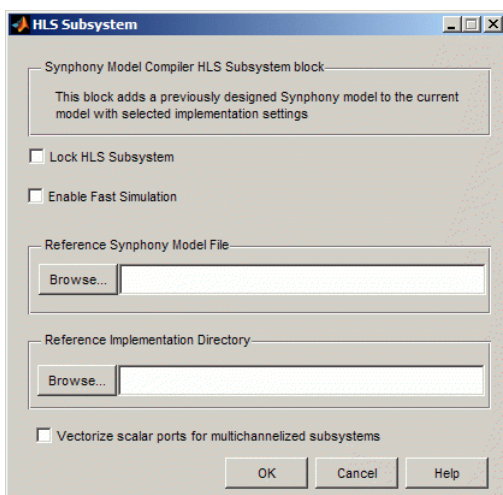
The following figure shows the EDA Simulator Link MQ communications interface, where a HDL Cosimulation block co-simulates a hardware component by applying input signals to and reading output signals from a VHDL model being simulated in ModelSim. The HDL Cosimulation block is the Simulink client to the ModelSim server.



Simulating HLS Subsystem Blocks

Model and verify the subsystem design before using it as a HLS Subsystem block in your design. The following procedure describe how to simulate the subsystem design, using normal mode and fast simulation model.

1. Set up the HLS Subsystem block, as described in [Creating an HLS Subsystem Block, on page 786](#).
2. To simulate the block in normal mode, do the following:
 - Make sure the Enable Fast Simulation option is not checked when you set up the block.



- Simulate the subsystem.

The tool uses the Simulink model for simulation. See [Enable Fast Simulation, on page 323](#) for details.

3. To simulate the block in Fast Simulation mode, do the following:
 - This mode requires a C compiler. Make sure you specified a compatible C compiler when you set up and installed the Synphony Model Compiler tool.
 - Make sure you have a C output license for each HLS Subsystem block you want to simulate. In Fast Simulation mode, each of these blocks requires its own C output license for simulation.

- Enable Enable Fast Simulation. The tool generates a C-model for the block and uses it for simulation.
- Simulate the subsystem. The tool uses the C-model an S-function wrapper to simulate the block. See [Enable Fast Simulation, on page 323](#) for details.

Use fast simulation when you are sure that the underlying block design is not going to change.

Viewing Simulink Signals in a Waveform Viewer

To debug signals within Simulink, you can use the built-in Scope or To Workspace blocks. However, using an HDL simulator waveform viewer is more convenient, especially when debugging control logic signals.

Synphony Model Compiler can export Simulink simulation signals to a waveform viewer environment, using a mechanism that generates a Value Change Dump (VCD) file.

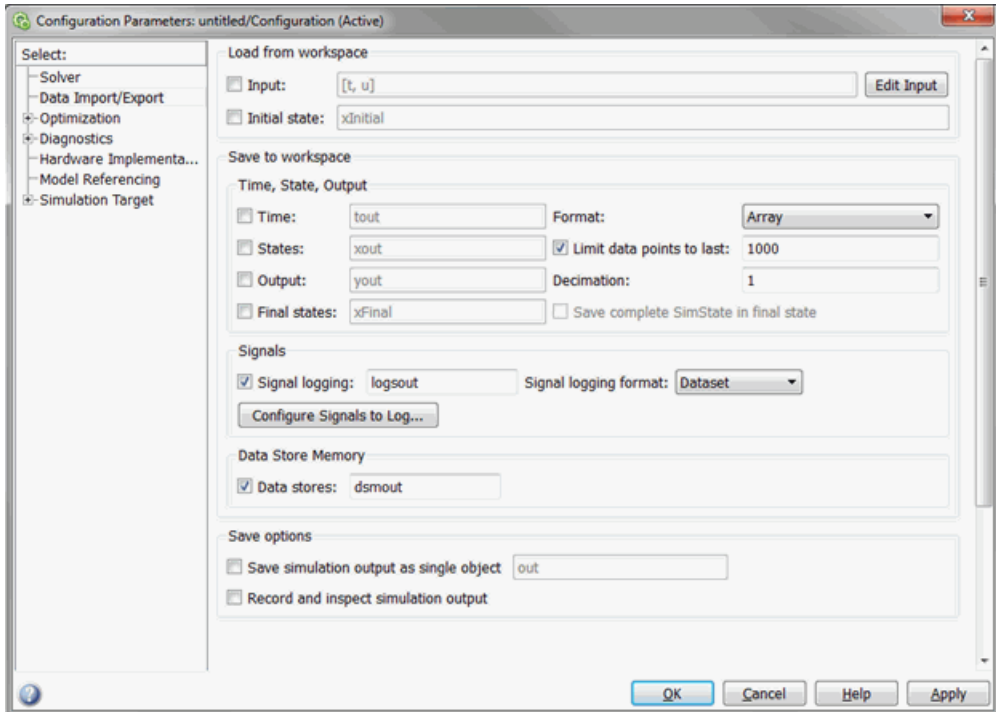
Enabling Signal Logging for the Model

To enable signal logging for a Simulink model:

1. Open the Configuration Parameters dialog box for your model.

From the menu choose Simulation, then Model Configuration Parameters.

2. Select the Data Import/Export tab.
3. Enable Signal logging and click OK.



Your model is configured to generate a signal logging variable that contains data for any individual signals in the model. You now have enabled signal logging.

Enabling Signal Logging for Individual Signals

To enable signal logging for a Simulink signal:

1. In the Simulink model, right-click on the signal.
2. From the context menu, select Signal Properties.
3. On the Signal Properties dialog box, select Log signal from the Logging and accessibility tab.
4. You should provide a meaningful name for the signal.

See the Simulink documentation for more information about signal logging.

Running the Simulation

To perform simulation:

1. After you have enabled signal logging for your model and selected signals to be logged, run simulation.
2. Simulink generates a MATLAB workspace variable containing all of your logged signals. The MATLAB workspace variable name is specified on the Data Import/Export tab of the Configuration Parameters dialog box.

Generating the VCD File

To generate the VCD file:

1. Use the script provided by SMC called `syn_write_wave`.
The script extracts all the logged signals from your model and writes them to a VCD file. Logged signals also include a clock for each sample rate found in the model. The VCD file has the same name as your model file.
2. Once the VCD file is created, open it in any waveform viewer of your choice.

CHAPTER 11

Working with SMC Output

This chapter describes how to verify the output generated by the Symphony Model Compiler tool and run logic synthesis on it:

- [Checking the Log File, on page 850](#)
- [Verifying the RTL with a Test Bench, on page 853](#)
- [Working with the Output for FPGA Designs, on page 856](#)

Checking the Log File

The shls.log file is generated after synthesis.

1. Check the log file for messages like the following to ensure that your constraints were applied:

```
@N:The retiming constraints are successfully applied for the
following blocks or hierarchies in the following order:
```

```
Subsystem.Delay_abc : {lock_only}
```

```
Subsystem.Delay_abcl : {lock_only}
```

```
@W: "shls_retiming_lock {lock_only}" constraint is already applied
for the Delay_abcl block.
```

2. Check the Hardware Resource Utilization section for estimates of resource usage.

See [Hardware Resource Utilization Section, on page 850](#) for a description.

3. Check the Pattern Usage section for a report of pattern usage.

See [Pattern Usage Report, on page 851](#) for a description.

Hardware Resource Utilization Section

The log file contains a section that estimates usage for multiplier, adder, register, shift register and RAM resources. This preliminary estimate provided by the Symphony Model Compiler tool saves you from running logic synthesis in order to get an estimate. This is particularly useful in the iterative design development stage, and can save a significant amount of time during the development cycle.

The resource estimate includes hierarchical blocks like custom blocks, M Control, and HLS Subsystem, but does not include black boxes.

An example of the resource usage section is shown below:

```
ESTIMATED HARDWARE RESOURCE UTILIZATION
*****
Number of Multipliers           : 10
14X18 bits: 2
18X8 bits: 4
16X8 bits: 4
```

```

Number of Adders                : 25
    49X49  bits:  20
    32X32  bits:   1
    25X1   bits:   4

Number of Registers/Delays      : 44 (99 register bits)
    48 bits   : 22
    32 bits   :  2
    19 bits(SR): 20

Number of RAMs                  : 17
    W: 64, D:  32 bits(SP):  6
    W:  1, D: 128 bits(SP):  3
    W:  1, D:  64 bits(SP):  2

-----
W  => Width
D  => Depth
SR => Shift Register
SP => Single Port
DP => Dual Port
-----

```

Pattern Usage Report

The Pattern Usage Statistics section of the Symphony log file summarizes the statistics for each pattern identified in the user design. For each pattern, it lists three statistics:

- Number of primitive blocks in the pattern netlist
- Number of times the pattern occurs in the original netlist
- Instantiated number of pattern in the generated RTL (after folding)

Here is an example of pattern usage statistics:

```

PATTERN USAGE STATISTICS
*****

2 distinct patterns are used in the system
-----

Pattern 1
-----
@N: Number of blocks in the pattern: 3
@N: Occurrence in the original netlist: 10
@N: Number of instantiated devices: 2
-----

```

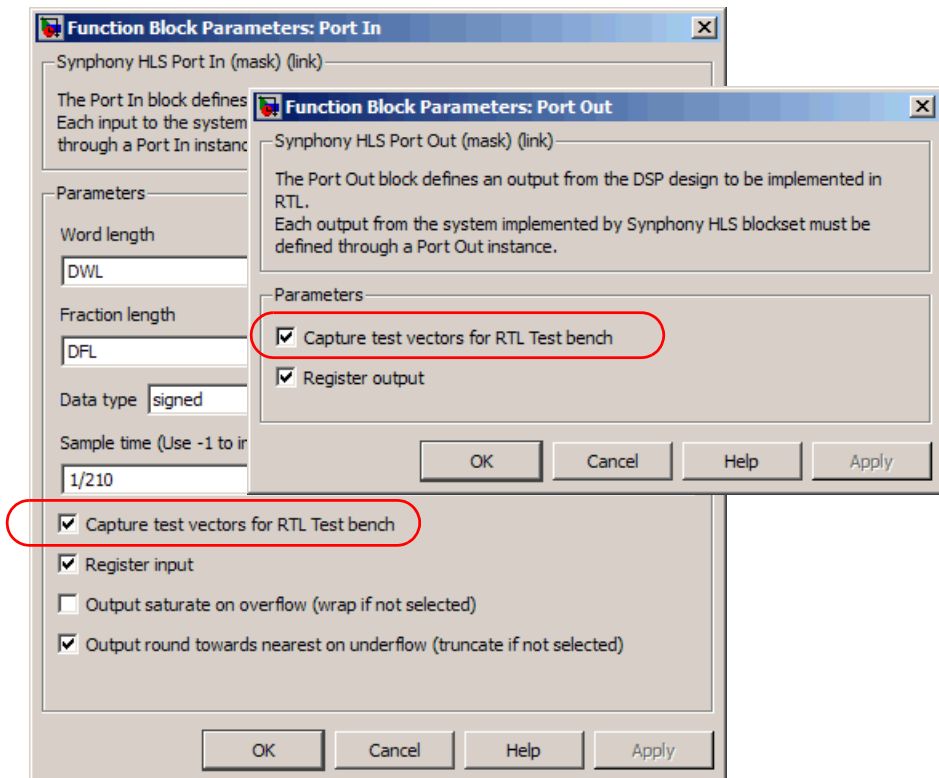
Pattern 2

@N: Number of blocks in the pattern: 4
@N: Occurrence in the original netlist: 15
@N: Number of instantiated devices: 3

Verifying the RTL with a Test Bench

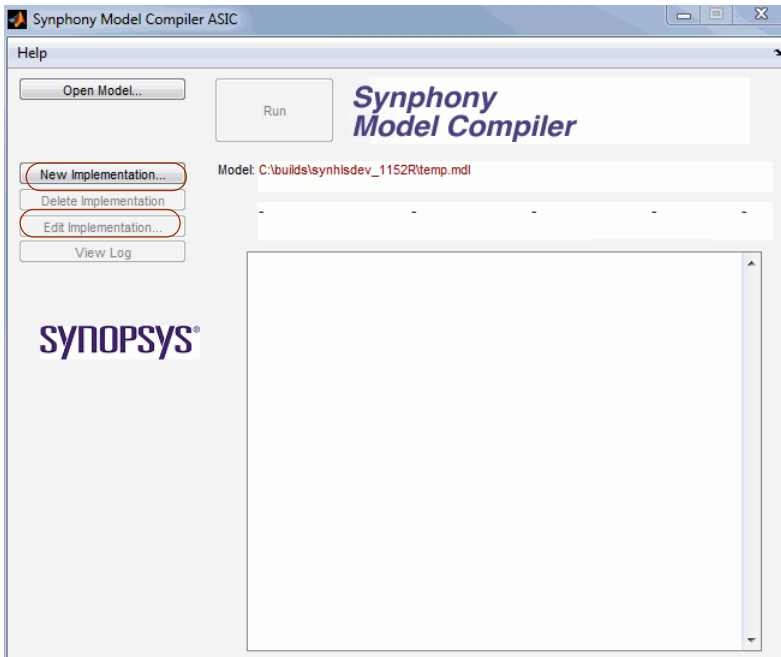
The following procedure shows you how to generate a test bench from the Symphony Model Compiler tool, and use it to verify your design.

1. In the Simulink schematic window, set parameters for the Port In and Port Out blocks:
 - For each Port In block in the design, double-click the block, and enable the Capture test vectors for RTL Test bench option. Click OK.
 - For each Port Out block in the design, double-click the block, and enable the Capture test vectors for RTL Test bench option. Click OK.



2. Simulate your design. This ensures that you have captured the stimuli and expected results for the design.

3. Double-click the SHLSTool block in the model window and then click New Implementation or Edit Implementation in the SHLSTool window.



- For a testbench with a VHDL netlist, click Generate VHDL to generate a VHDL design. Enable the Generate RTL test bench checkbox. This generates a .vhd netlist file and a .vhd testbench in the <implementation>/vhd directory.
 - For a testbench with a Verilog netlist, click Generate Verilog to generate a Verilog design. Enable the Generate RTL test bench checkbox. This generates a .v netlist file in the <implementation>/verilog directory, and a .v testbench in the <implementation>/verilog directory.
 - Set other target options as usual (see [Setting up Implementations, on page 644](#)).
 - Click OK.
4. Click Run in the Synphony Model Compiler window.

The Synphony Model Compiler tool generates a Verilog or VHDL netlist, as specified, along with a Verilog or VHDL test bench and .do files for supported simulators. The following files for simulation are in the direc-

tory where the .mdl file for the design is stored, under the vhd or verilog subdirectory for the implementation.

File	Description
Inport_<design>_<port>.dat	Each Port In instance has a file with stimuli for the test bench.
Output_<design>_<port>.dat	Each Port Out instance has a file with expected results for the test bench.
<design>.vhd or <design>.v	The RTL associated with the design.
<design>_Test.vhd or .v	The test bench wrapper for the design. It applies the stimuli, and compares the results with the expected results.
<design>_affirma.do	A simulation script for the Cadence NC simulator.
<design>_activehdl.do	A simulation script for the Aldec simulator.
<design>_modelsim.do	A simulation script for the ModelSim simulator.

The simulation scripts help to quickly verify the RTL-level representation of the DSP algorithm.

5. Simulate the test bench to verify your design.

- To run Aldec Active-HDL, type the following at the command prompt:
vsimsa <design>_activehdl.do
- For ModelSim, type the following at the command prompt:
vsim < simulate_modelsim.do

You can also use the Cadence IUS54 or Affirma simulators. The Verilog simulator you select must be able to handle Verilog 2001-style statements, and the VHDL simulator must handle VHDL 93.

Working with the Output for FPGA Designs

This procedure shows you how to use the Symphony Model Compiler output for logic synthesis with the Synopsys FPGA synthesis tools, Synplify Pro or Synplify Premier. These are the only supported FPGA synthesis tools.

1. Make sure you have access to a Synopsys FPGA synthesis tool, Synplify Pro or Synplify Premier.
2. Create your Symphony Model Compiler implementation and specify the following implementation options. See [Setting up Implementations, on page 644](#) for details.
 - Specify an FPGA target architecture.
 - Specify the format for the output netlist, and any optimizations (see [Optimizing with Retiming, on page 655](#), [Optimizing with Folding, on page 662](#), and [Optimizing with Multichannelization, on page 674](#)).
 - Click Run.

The Symphony Model Compiler software generates an optimized RTL netlist, a project file and other files. The netlist is vendor-independent and can be used as input for synthesis.

3. Optionally, verify your design (see [Verifying the RTL with a Test Bench, on page 853](#)).
4. Start Synplify Pro or Synplify Premier, and set up a project, using the project file, constraint file, and the RTL netlist generated by the Symphony Model Compiler tool.
5. Set synthesis options and constraints and synthesize your design.

In the logic synthesis tool, you can further explore device trade-offs by setting device-specific constraints. You can also use various vendor-specific synthesis optimizations to further refine your design before placing and routing it.

CHAPTER 12

Using M Code Blocks

The Synphony Model Compiler tool includes the M Control block, which provides a way to implement complex control-intensive functions using the MATLAB M language. The following provide more information about creating and using this block:

- [Using M Code Blocks, on page 858](#)
- [M Coding Style, on page 862](#)
- [M Language Support for M Code Blocks, on page 893](#)

Using M Code Blocks

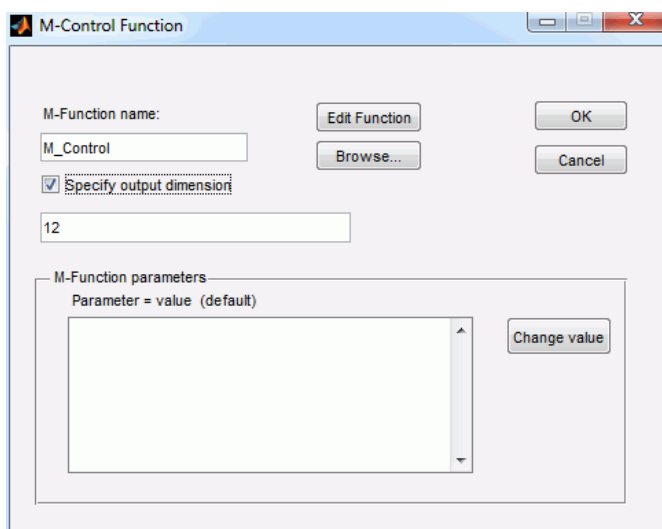
This section shows you how to incorporate an M Control block in your design and some tips on using M Control blocks.

- [Using M Code Blocks in SMC Designs, on page 858](#)
- [Coding for Synthesis with M Code Blocks, on page 860](#)

Using M Code Blocks in SMC Designs

Use the following procedure to incorporate M functions to your design.

1. Select the M Control block and instantiate it in your design.
See [SMC M Control, on page 356](#) for details. Use the M Control block for control logic.
2. Specify the function for the block.
 - Double-click the block to open the dialog box.



- Enter the name of the M function for the block. See step 3 for information about creating a new M function. The specified function must be in the MATLAB path.

- If you want to view or make changes to a function, click Edit Function to open the file in the MATLAB M file editor. Edit the function as needed.
 - If you want to override the currently-defined M-Function parameters, select the relevant parameters in the list and click Change Value. When you override a parameter with a new value, the default value is shown in parentheses near the new value. For more information about defining overridable parameters, see [Overridable Parameters, on page 881](#).
 - If the output port dimensions are not scalar, enable Specify output dimensions, and specify the dimensions.
 - Save the file and close the window.
3. To write a new M function, do the following:
- Open the dialog box for the M code block, and enter a name for the function as before.
 - Click Edit Function to open a MATLAB M file editor window.
 - Write the M function following the guidelines and caveats described in [Coding for Synthesis with M Code Blocks, on page 860](#), [M Coding Style, on page 862](#), and [M Language Support for M Code Blocks, on page 893](#).
 - Use the syntax highlighting feature built into the M editor to check syntax.
 - Save the file when you have finished.
4. Click OK in the dialog box when you have finished creating or editing the function file.
5. In the model window with the design, press Ctrl-d to update Simulink.

This propagates data types and sample rates through the input and output ports. At every simulation tick, the block converts the fixed-point data at the block inputs to double, executes the M control function on this double data, and then converts the output double data to fixed-point again for the rest of the model. If your M file followed the guidelines, the model updates successfully and you can simulate the design.

Coding for Synthesis with M Code Blocks

This section outlines the best practices for setting up your M code so that it generates the optimal hardware during M synthesis.

1. Make sure the design is appropriate for M synthesis:
 - M code blocks only support a single sample rate.
 - All inputs must be fixed point.
2. Write M code that is hardware-aware.
 - Define the states in the design (memories, registers, etc.).
 - Describe the conditional and unconditional data flow between the states during a clock cycle, using M language semantics like matrix variables, loops, and built-in functions.

See [Hardware-Aware M Code, on page 892](#) for an example.

3. Specify precision.
 - Specify the precision of persistent variables used in accumulation-type operations. See [Precision Bounds for Persistent Variables, on page 885](#) and [Counters, on page 878](#) for more information.
 - Data type propagation through the M code is full-precision up to a limit of 53 bits. Use MATLAB quantize commands to control the precision and, if necessary, to limit precision to 53 bits or less. See [Data Type Restrictions, on page 866](#) and [Controlling Precision and Signedness with Quantizers, on page 867](#).
 - M synthesis uses maximum precision to calculate the data type of constants; this means that constants with infinite precision could cause an increase in area. Use the SMC `shls_convert` function or the MATLAB quantizer object to quantize constants. For details, see [Defining Precision with shls_convert, on page 886](#) and [Defining Precision with the Quantizer Object, on page 887](#), respectively.
4. Model persistent variables accurately.
 - To create state-holding elements, use persistent variables that are initialized to zero with the `isempty` function. See [Using Persistent Variables, on page 883](#).

- Ensure that the access-update sequence for persistent variables is correct. M synthesis optimizes away persistent variables (registers) that are never used before they are updated. See [Access-Update Sequence for Persistent Variables, on page 888](#) for details.
 - For better performance, reduce the number of conditional assignments to persistent variables. Conditional assignments reduce the possibilities for mapping datapath logic to dedicated DSP FPGA resources. See [Conditional Assignments to Persistent Variables, on page 890](#) for details.
5. Specify combinatorial logic.
- Avoid incomplete assignments to outputs and program variables. For more information, refer to [Combinatorial Logic, on page 868](#).
6. Use the features built into the M editor like syntax highlighting to step through your M code and identify problems.
- For additional information about the extent of M syntax support, see [M Language Support for M Code Blocks, on page 893](#).

M Coding Style

Specifying block behavior with a high-level, sequential programming style makes it easier to develop and debug. The Symphony Model Compiler M code blocks provide a way to implement complex control-intensive and datapath functions using the MATLAB M language (see [SMC M Control, on page 356](#)).

The M function is ultimately used to infer synchronous digital hardware, so you must observe certain coding restrictions and idioms, which are described in the following sections:

- [Ports and Timing, on page 862](#)
- [M Code Block Data Types, on page 864](#)
- [Combinatorial Logic, on page 868](#)
- [Using Persistent Variables, on page 883](#)
- [Precision Bounds for Persistent Variables, on page 885](#)
- [Memories, on page 869](#)
- [State Machines, on page 870](#)
- [Counters, on page 878](#)
- [MATLAB Function that Evaluates to a Constant, on page 880](#)
- [User-Defined Functions for M Code Blocks, on page 880](#)
- [Overridable Parameters, on page 881](#)

Ports and Timing

You encapsulate the behavior of an M code block in a top-level function written in M. The named input parameters of this top-level function define the physical input ports of the synthesized SMC M code block, and the named return values of the function define the physical output ports of the block. A scalar or single-element vector return value defines a single output port in the hardware. A multi-element vector return value defines multiple output ports, one for each element of the returned vector.

This is a function header for 3-input, single-output design:

```
function z = checksum(a, b, c)
```

This is a function header for 3-input, 2-output design:

```
function [sum, cout] = adder(a, b, cin)
```

This top-level function can call other functions, like user-defined functions or some MATLAB built-in functions. See [Built-In Function Support, on page 895](#) for a list of supported MATLAB built-in functions and [User-Defined Functions for M Code Blocks, on page 880](#) for information about user functions.

Ports

As with all Symphony Model Compiler blocks, the inputs of an M code block receive signals which have a specific data type (word width and signedness) and a specific sample rate.

Sample rate	All inputs to an M code block must have the same sample rate; you can not have a multi-rate M code block.
Data type	<p>The data type of each input can be in either signed 2's complement or unsigned fixed-point format.</p> <p>Each input port can have its own fixed-point format. The fixed-point format allows a certain number of bits to the left and to the right of the digital point. The tool determines the data type of an input port from the system context in which the M code block is instantiated; it is not explicitly specified in the M function. The fixed-point format of each output port is determined by using the input port formats derived from the system context and the M language expressions used to compute output port values.</p>
Word width	The total word width of each input port signal must not exceed 53 bits.
Dimensions	The dimensions of the input ports are inherited. You must explicitly specify output port dimensions in the block parameters (SMC M Control, on page 356).

Timing

The simulation timing model for an M code block assumes that the top-level function for the block is called once for each input sample. This means that multiple references to an input parameter within the same function call all access the same input sample.

Since synthesis should match simulation, this also defines the timing of the synthesized hardware. The hardware must therefore operate with a maximum latency from inputs to outputs of one sample period.

M Code Block Data Types

Each input, output, and internal variable in the M function that defines an M code block must have a well-defined fixed-point data type. This data type specifies a digital hardware representation:

- The number of bits to the left of the binary point (integer portion)
- The number of bits to the right of the binary point (fractional portion)
- Whether the format is signed (2's complement) or unsigned

In addition, inputs and outputs must be scalar, vector, or matrix operations. Complex types and operations are not supported.

The data type of each input and variable affects the hardware synthesized for the operations performed on that input or variable. It is therefore important to understand how data types are assigned and propagated and how to control the precision and signedness of operations:

Data type assignment	Input Data Type Assignment, on page 864 Output Data Type Assignment, on page 865
Data type propagation through the M-function	Internal Data Type Propagation, on page 865 Constant Data Type Assignment, on page 866 Port Dimensions, on page 866 Data Type Restrictions, on page 866
Precision and signedness control	Controlling Precision and Signedness with Quantizers, on page 867

Input Data Type Assignment

The tool determines the data types of primary inputs from the Simulink block diagram in which the M code block is instantiated. Before compiling an M code block, the tool determines the fixed-point data type of the signal driving each block input by propagating the data type from the Simulink model

enclosing the block. This means that the data types for the input ports are determined outside the M function that defines the behavior of the M code block.

Internal Data Type Propagation

The fixed-point data types of variables and operations defined in the function for the M code block are determined by propagating data types from the primary inputs. This is the propagation of data types through a Simulink model. The data type resulting from an operation (such as add or multiply) is determined by considering the nature of the operation and the data types of its input operands.

Type propagation within an M code block tries to preserve the full precision of the result of each operation.

- Operations such as add and multiply increase precision. For example, the output of an addition performed on two N-bit inputs will generally have an N+1-bit result type, preserving any overflow in the N+1st bit. Similarly, multiplying an N-bit input by an M-bit input will usually generate an N+M-bit result.

It is not always possible to maintain the full precision result of an operation such as add or multiply, which increases required precision. Exceptions to the full precision model are discussed below in [Data Type Restrictions, on page 866](#).

- Other operations naturally maintain the precision of their inputs. For example, a bitwise AND of two N-bit inputs produces an N-bit result.
- The output precision of certain operations will be less than the input precision. For example, a compare-for-equality operation on two N-bit inputs produces a 1-bit output signifying the inputs are either equal or not equal.

Output Data Type Assignment

The fixed-point data type of each M code block output is determined using the same type propagation process described above for internal operations. The propagated data type of the operation driving an output becomes the data type of that output. The fixed-point formats of each output are communicated to the enclosing Simulink model and assigned to the signals driven by the outputs of the M code block.

Constant Data Type Assignment

The fixed-point format of constants in the M source code is determined by using the smallest amount of precision necessary to represent the constant value. This also applies to constants which are derived by evaluating constant-valued expressions (such as $1.5 + 6$) at compile time. The use of minimal precision to represent the constant assumes the constant value is exactly representable with finite precision in a binary representation.

Some constants and constant expressions cannot be exactly represented with finite precision in base-2; for example the constant expression $1/3$. For these constants, the maximum length of the fractional port is restricted to 32 bits. This level of precision can produce very large hardware elements which may not be required by your application. It is therefore a good practice to constrain the precision of constants in the M source code. See [Controlling Precision and Signedness with Quantizers, on page 867](#) for information on how to do this.

Port Dimensions

The tool determines the dimension of primary inputs from the Simulink block in which the M code block is instantiated. The dimensions of variables and operations defined in the function for the M code block are determined by propagating the dimension from the primary inputs, which is analogous to the procedure used to propagate dimensions through a Simulink model.

The dimension resulting from an operation, such as concatenation or extraction, is determined by considering the nature of the operation and the dimension of its input operands. The dimensions of the output ports must be explicitly specified in the Specify output dimension parameter of the M code block. ([SMC M Control, on page 356](#)).

Data Type Restrictions

In Simulink, the SMC M code block is simulated using an internally-generated S-Function which calls the original M function. By simulating your M code, you can interactively debug the source for the M code block. However, Simulink executes your M function using IEEE double-precision floating point types and arithmetic, rather than the fixed-point types that are ultimately synthesized.

Precision

As long as the synthesized fixed-point types and internal operations use a precision less than the precision of doubles (53 bits), the simulation results using double precision floating point will match the synthesized hardware. However, if an input or internal operation requires a precision greater than 53 bits, simulation is not guaranteed to match synthesis.

During internal type propagation, the Symphony Model Compiler M compiler checks for full precision result conditions that exceed 53 bits, and issues a warning for each such violation. It also trims the full-precision bit width of the offending operation to 53 bits by discarding enough least significant bits. However, this trimming does not guarantee that simulation will match synthesis, so it is a good practice to constrain the precision of internal operations as described in [Controlling Precision and Signedness with Quantizers, on page 867](#).

Precision of Input Port Widths

For the same simulation mismatch reasons, input port widths are restricted to 53 bits or less. Input port widths greater than 53 bits result in an error. To correct such an error, you must constrain the width of the signal driving the offending M code block input in the Simulink model. To do this, either use a Convert block before the M code block input or adjust the output data type of the block driving the signal.

Controlling Precision and Signedness with Quantizers

The SMC M code blocks support quantizers, which are available in the MATLAB Fixed-Point Toolbox. Quantizers are useful in situations where you need to explicitly define the precision and signedness of a result.

You can use quantizers to constrain internal operation widths and avoid simulation-synthesis mismatches, and in situations where the full precision of an operation, input, or constant is not required. You can use the `shls_convert` function instead of quantizers. The two methods result in equivalent behavior, but `shls_convert` has faster simulation times for M code blocks.

For detailed information about quantizers, refer to the MATLAB documentation for the Fixed-Point Toolbox. In brief, you define a quantizer object which specifies a fixed-point format and a set of modes to handle overflows and underflows when converting to this format. The type conversion specified by

the quantizer object is applied to an expression by invoking the `quantize()` function. For examples of the use of quantizers, see [Precision Bounds for Persistent Variables, on page 885](#).

The SMC M code blocks support a subset of the full quantizer functionality provided by the Fixed-Point Toolbox, as described in this table:

Quantizer Property Name	Quantizer Property Value	SMC Support
mode	'double'	No
	'float'	No
	'fixed'	Yes
	'ufixed'	Yes
	'single'	No
roundmode	'ceil'	Yes
	'convergent'	Yes
	'fix'	Yes
	'floor'	Yes
	'nearest'	Yes
overflowmode (fixed-point only)	'saturate'	Yes
	'wrap'	Yes
format	[wordlength fractionleng] 'fixed' or 'ufixed' only)	Yes
	[wordlength exponent length] ('float' mode)	No

Combinatorial Logic

Most useful control functions such as state machines and counters require state-holding elements (see [Using Persistent Variables, on page 883](#) and [Counters, on page 878](#) for details), but the simplest M code block is one that requires no state-holding elements. You implement such a block as purely

combinatorial digital logic with no delay elements or registers. An M function which does not use persistent variables to compute the values of its outputs is implemented with purely combinatorial logic.

The following example shows a block that is implemented as purely combinatorial logic, with no states:

```
function res = sum(a, b, c, d, sel)
    if(sel)
        res = a + b;
    else
        res = c + d;
    end
end
```

In a purely combinatorial block, each output must be assigned a value along some control path through the M function. Failure to do so may cause a mismatch between simulation in Simulink and the hardware implementation produced by the Symphony Model Compiler tool. The preceding code example illustrates this required practice; note that the output `res` is always assigned a value each time the function `sum` is called.

Persistent Variables

A variable that is declared as *persistent* retains its value across function calls when an M function is executed in the software. State-holding elements in hardware require this behavior, because they must retain their value across input sample times, so in the programming model for M code blocks, state-holding elements are inferred using persistent variables. See [Using Persistent Variables, on page 883](#) for details.

Memories

If you are using the M Control block, code persistent arrays as described in [Using Persistent Variables, on page 883](#). The tool infers memory from the persistent array.

State Machines

The M language makes it easy to define hardware state machines using the coding conventions described in [Using Persistent Variables, on page 883](#).

This section contains examples of Mealy and Moore state machines. It also has a stateflow example, that demonstrates how to implement simple state flow designs with the M Control block.

Mealy State Machine Example

A Mealy state machine is one in which the outputs are a function of both the current state and the state machine inputs.

The following `find_pattern_mealy` function implements a state machine for detecting the pattern 1001 in a serial bitstream, including overlapping instances of the pattern.

```
% Detect the bitpattern 1001, including overlapping instances such
% as in the sequence 1001001.

function detected = find_pattern_mealy(rst, x)

persistent state; % The state variable

if(isempty(state)) %Not a substitute for explicit local reset
input
    state = 0;
end

% Define named constants for states
RESET = 0;
GOT_FIRST = 1;
GOT_SECOND = 2;
GOT_THIRD = 3;
GOT_PATTERN = 4;

if(rst == 1)
    state = RESET; % Implements a local, synchronous reset
    detected = 0;
else
    switch(state)
        case RESET
            if(x)
                state = GOT_FIRST;
            end
            detected = 0;
        case GOT_FIRST
```

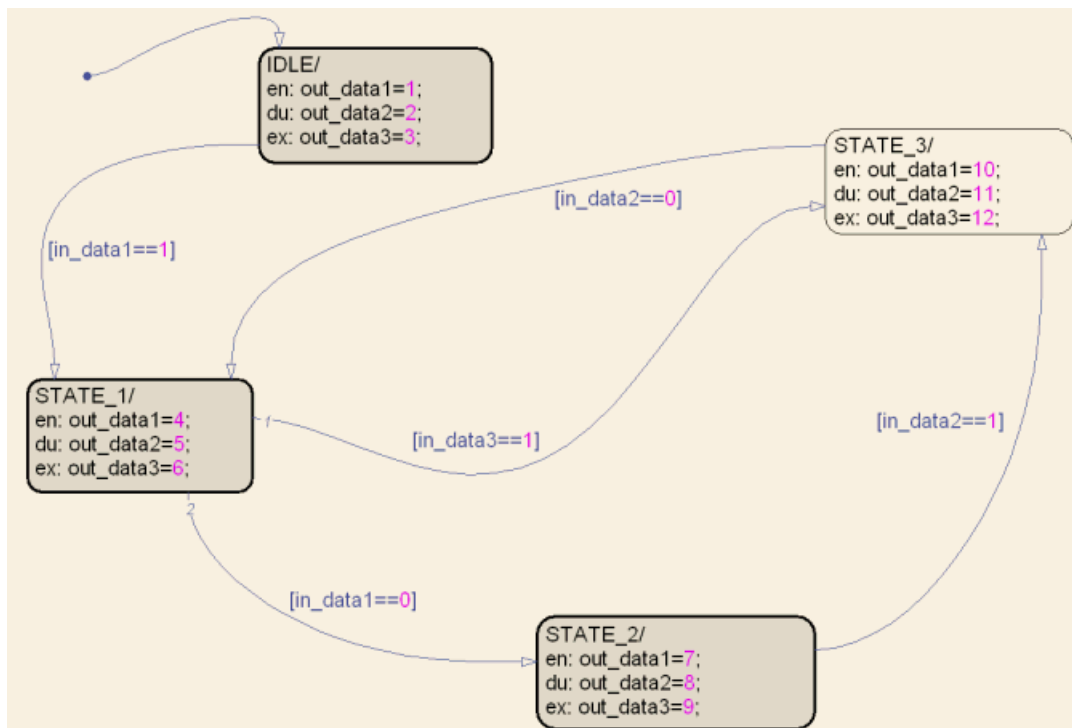
```

        if(x)
            state = RESET;
        else
            state = GOT_SECOND;
        end
        detected = 0;
    case GOT_SECOND
        if(x)
            state = RESET;
        else
            state = GOT_THIRD;
        end
        detected = 0;
    case GOT_THIRD
        if(x)
            state = GOT_PATTERN;
            detected = 1;
        else
            state = RESET;
            detected = 0;
        end
    case GOT_PATTERN
        if(x)
            state = GOT_FIRST;
        else
            state = GOT_SECOND; % Detect overlapping pattern
        end
        detected = 0;
    otherwise
        state = RESET;
        detected = 0;
    end
end
end

```

State Flow Example

This example provides guidelines for translating a stateflow design into M language.



```

function [outdata1, outdata2, outdata3] = example(indata1,indata2,
    indata3)

%%%% declaration of variables %%%%
persistent CurrentState
persistent NextState
persistent Poutdata1
persistent Poutdata2
persistent Poutdata3

RESET=0;
IDLE=1;
STATE1=2;
STATE2=3;
STATE3=4;

%%%% initialize persistent variables %%%%
if (isempty(CurrentState))
    CurrentState=0;
end
  
```

```
if(isempty(NextState))
    NextState=0;
end

if(isempty(Poutdata1))
    Poutdata1=0;
end

if(isempty(Poutdata2))
    Poutdata2=0;
end

if(isempty(Poutdata3))
    Poutdata3=0;
end

%%%% start of main body %%%%

if (CurrentState == RESET)
    NextState=IDLE;
    Poutdata1=1; % execute entry actions for state IDLE
else
    switch CurrentState
        case IDLE
            if(indata1==1) % execute exit actions for state IDLE
                Poutdata3=3;
                NextState=STATE1;
            else % execute during actions for state IDLE
                Poutdata2 =2;
            end
        case STATE1
            if(indata3==1) % execute exit actions for state STATE1
                NextState=STATE3;
                Poutdata3=6;
            elseif(indata1==0) % execute exit actions for state STATE1
                NextState=STATE2;
                Poutdata3=6;
            else % execute during actions for state STATE1
                Poutdata2 =5;
            end
        case STATE2
            if(indata2==1) % execute exit actions for state STATE2
                NextState=STATE3;
                Poutdata3=9;
            else % execute during actions for state STATE2
                Poutdata2 =8;
            end
        case STATE3
            if(indata2==0) % execute exit actions for state STATE3
```

```

        NextState=STATE1;
        Poutdata3=12;
    else % execute during actions for state STATE3
        Poutdata2 =11;
    end
end
end
end

%%%% executing entries if any %%%%
if (NextState~=CurrentState)
    switch NextState
        case IDLE
            Poutdata1 =1;
        case STATE1
            Poutdata1 =4;
        case STATE2
            Poutdata1 =7;
        case STATE3
            Poutdata1 =10;
    end
end
end
%%%%% end of main body %%%%

%%%%% update of persistent variables %%%%
CurrentState=NextState;
outdata1=Poutdata1 ;
outdata2=Poutdata2 ;
outdata3=Poutdata3 ;

```

State Encoding

In this example, the numeric encodings of the various states are given descriptive names by assigning the state encodings to variables.

The five states are modeled using a single persistent variable, `state`. The input `x` provides samples of the serial bitstream and the input `rst` places the state machine into a reset state (named `RESET`) when asserted. The output `detected` is asserted high whenever the target pattern is detected and is reset to low on the sample period following a successful detection. A `switch` statement whose cases correspond to legal operating states is used to compute next-state updates and to correctly set the detection signal.

Regardless of the state in which the system is operating, the `rst` input places it into the RESET state when asserted. To prioritize the effect of `rst` over the `x` input, the example has a top-level if-else statement which evaluates if `rst` is used. If `rst` evaluates to true, reset actions are performed. Otherwise, the switch statement is executed.

Local Resets

This coding style synthesizes a local, synchronous reset for `rst`. Updates to the state register are synchronized to a physical clock in the synthesized hardware. There is no way to code a local, asynchronous reset. However, the mandatory global reset modeled with `isempty` can be specified as either synchronous or asynchronous using the Symphony Model Compiler UI.

Although the `state` persistent variable must be initialized to zero using `isempty`, it is better to also have a local synchronous reset, such as `rst`. There are two reasons why a local reset for state machines is desirable in addition to the global reset.

- The global reset always sets the state registers to zero, which may not be the desired encoding for the reset or start-up state of the state machine, or zero might be an illegal state for that encoding.
 - Where a zero-encoded state (as in one-hot encoding) is not a legal operating state, code it as a separate case in the switch state or in the otherwise clause of the switch. This lets the state machine detect when a global reset forces it into the zero state and it can transition to the correct start-up state on the next sample period.

In one-hot encoding, all normal operating states have exactly one bit set in the state register, so setting all state bits to zero places the system in an illegal state.
 - When a zero-encoded state is a normal operating state but not the desired start-up state, define a local reset input using a top-level if-else and ensure that it is asserted as soon as possible after the global reset to place the state machine into the correct start-up state.
- A local reset is also useful because the global reset is generally only asserted once at system start-up. The local reset allows other processes to reset the state machine at any time during system execution.

If the start-up state of the state machine is encoded as zero and if the state machine will only be reset once, then the global reset will suffice for initializing the state machine and the local reset is unnecessary.

Assignment for Outputs and Non-Persistent Variables

Note that the detected output is implemented as combinatorial logic because it is not declared as a persistent variable. The detected signal is stateless, so it is crucial that you assign it a value along every control path in the `find_pattern_mealy` function. Failure to always assign a value to detected can result in an undefined signal during simulation and an unpredictable hardware implementation. You must assign a value to every non-persistent variable or output.

Assignment for Persistent Variables

You do not always need to assign a value to the persistent variable `state`, because it can retain its current state. In this example, the `switch` statement that handles the RESET case only assigns `state` if the `x` input is asserted. Otherwise, it retains the current RESET state and no explicit assignment is required.

Moore State Machine Example

In a Moore state machine, the outputs are a function of the current state alone, unlike a Mealy state machine where the outputs can change as soon as an input changes. If there is an input glitch on a Mealy state machine, the glitch can propagate to the state machine output.

You can easily convert a Mealy state machine to a Moore state machine. The following M code converts the previous example ([Mealy State Machine Example, on page 870](#)) to a Moore state machine. Note that the exact timing of when the detected output changes is different in the Mealy and Moore examples. This is because the output change is sensitive to the `x` input in the Mealy example, but not sensitive to it in the Moore example.

The `find_pattern_moore` function shown here factors the code for computing the detected output out of the original `switch` statement in `find_pattern_mealy` and no longer depends on the input variable `x`. It is solely a function of the state persistent variable.

```
% Detect the bitpattern 1001, including overlapping instances
% as in the sequence 1001001.

function detected = find_pattern_moore(rst, x)
persistent state; % The state variable
```



```
if(isempty(state)) % Not a substitute for explicit reset input
    state = 0;
end

% Define named constants for states
RESET = 0;
GOT_FIRST = 1;
GOT_SECOND = 2;
GOT_THIRD = 3;
GOT_PATTERN = 4;

% This implements the detected output, Moore-style
if(state == GOT_PATTERN)
    detected = 1;
else
    detected = 0;
end

% This if-else and switch implements the next-state update.
if(rst == 1)
    state = RESET; % Implements a local, synchronous reset
else
    switch(state)
        case RESET
            if(x)
                state = GOT_FIRST;
            end
        case GOT_FIRST
            if(x)
                state = RESET;
            else
                state = GOT_SECOND;
            end
        case GOT_SECOND
            if(x)
                state = RESET;
            else
                state = GOT_THIRD;
            end
        case GOT_THIRD
            if(x)
                state = GOT_PATTERN;
            else
                state = RESET;
            end
        case GOT_PATTERN
            if(x)
                state = GOT_FIRST;
            end
        end
    end
end
```

```

        else
            state = GOT_SECOND; % Detect overlapping pattern
        end
        otherwise
            state = RESET;
        end
    end
end
end

```

Counters

Counters are another commonly-used control function that are easily expressed in M. The following examples illustrate a counter and a counter with a local reset.

Counter

The following function implements a counter which continuously counts from 2 up to 10.

```

% Implements a counter which continuously counts from 2 up to 10.
function countOut = counter1()
    persistent count;
    if isempty(count)
        count = 0;
    end
    q = quantizer([4 0], 'wrap', 'ufixed');
    if ((count == 0) || (count == 10))
        count = 2;
    else
        count = quantize(q, count + 1);
    end
    countOut = count;
end

```

You implement the counter state with a single persistent variable, `count`. Because `count` is initialized to zero by the global reset, you must set it up so that this condition is detected and the counter can be given the correct

start-up value of 2. To do this, the if-else statement resets the count to 2 if the count is zero (due to global reset) or if the count has reached the maximum value of 10. Otherwise, the count is incremented.

The M compiler does not detect precision bounds for the counter, although this can be calculated by considering the sequence of states the counter moves through. Therefore, the M function quantizes the result of the increment to 4 bits, because this is the minimum precision required to represent the range of counter values.

Counter with Local Reset

A more practical counter might have a local reset as in the following code. The local reset is coded as for the state machine examples, using a top-level if-else statement. See [Mealy State Machine Example, on page 870](#) and [Moore State Machine Example, on page 876](#) for coding examples, and [Local Resets, on page 875](#) for a discussion on the use of local resets.

```
% Implements a counter which continuously counts from 2 up to 10.
% The counter has a local reset input.

function countOut = counter2(rst)

    persistent count;

    if isempty(count)
        count = 0;
    end

    q = quantizer([4 0], 'wrap', 'ufixed');

    if(rst)
        count = 2;
    else
        if((count == 0) || (count == 10))
            count = 2;
        else
            count = quantize(q, count + 1);
        end
    end

    countOut = count;

end
```

MATLAB Function that Evaluates to a Constant

Your M function can place a call to MATLAB functions that pass constant arguments and return a constant value. The M code blocks replace such function calls with a constant, just like constant propagation optimization. The returned constant from the function must be a scalar or vector; `struct` is not supported as a return type.

In the following `my_filter` function, the call to `get_filter_coeffs` is replaced by the filter coefficients obtained from `get_filter_coeffs`. No hardware is generated for the `get_filter_coeffs` function.

```
function [output] = my_filter(input)
...
    coefs = get_filter_coeffs();
...
...
end

function coefs = get_filter_coeffs()
    Fs = 150e6;
    mfir_N      = 15;                % Order
    mfir_Fpass  = 80e3/(Fs/128)*2;   % Passband Frequency
    mfir_Fstop  = 100e3/(Fs/128)*2; % Stopband Frequency
    mfir_Wpass  = 1;                % Passband Weight
    mfir_Wstop  = 1;                % Stopband Weight
    mfir_dens   = 16;                % Density Factor
    % Calculate the coefficients using the FIRPM function.
    coefs=firpm(mfir_N, [0 mfir_Fpass mfir_Fstop 1], [1 1 0 0], ...
        [mfir_Wpass mfir_Wstop], {mfir_dens});
end
```

If you are logged in with restricted access on a Windows machine, evaluation to constants might fail. M code blocks try to register the registry server where MATLAB is currently running. If this access is restricted, evaluation to constants will not work.

User-Defined Functions for M Code Blocks

You can define other M functions in addition to the top-level M function which defines the M code block. You can call these additional functions as needed within the body of the top-level function.

User-defined functions are useful for reusing M code and for improving the overall readability/maintainability of the M specification. For example, you can encapsulate frequent calls to `quantizer` and `quantize` in user-defined functions, as you use them multiple times to quantize results assigned to persistent variables.

The following example illustrates how user-defined functions can simplify the code for function `q_accum1` shown in [Defining Precision with the Quantizer Object, on page 887](#). The calls to `quantize` and `quantizer` are folded into the `my_quantize` user-defined function.

```
%Shows user-defined function for quantizing persistent variables
function res = q_accum1(x)

    persistent sum;

    if isempty(sum)
        sum = 0;
    end

    sum = sum + x;
    sum = my_quantize(sum, 8);
    res = sum;

end

function x_q = my_quantize(x, width)
    x_q = quantize(quantizer([width 0], 'wrap'), x);
end
```

Overridable Parameters

You can annotate simply assigned variables in the M-function so that they can be overridden for each block. The following steps describe the procedure.

1. Add a comment tag `% syn_parameter` to the end of the assignment. For example:

```
threshold = 5; % syn_parameter
```

When you open the dialog box of an M code block using this M-function, this parameter (and any others defined in the same way) are listed in the M function parameters list.

2. Open the dialog box for an M block using this function.

3. Select the parameter you want to override from the list in the M-Function parameters section, and click **Change Value**. Specify the override value you want to use.

The tool displays the original value in parentheses.

You can specify the override values as an expression that can be evaluated in the base workspace. Note that you cannot override parameters for variables that have the same name but which appear in different sub-functions of the M code.

Using Persistent Variables

The M programming language includes persistent variables. A variable that is declared as *persistent* retains its value across function calls when an M function is executed in the software. You use them to implement M code with state-holding elements. State-holding elements in hardware require this behavior, because they must retain their value across input sample times.

See the following for detailed information about coding with persistent variables:

- [M Code for Persistent Variables, on page 883](#)
- [Precision Bounds for Persistent Variables, on page 885](#)
- [Access-Update Sequence for Persistent Variables, on page 888](#)
- [Conditional Assignments to Persistent Variables, on page 890](#)

M Code for Persistent Variables

See the following topics for details:

- [M Language Example of Persistent Variable, on page 883](#)
- [Initialization with the isempty Construct, on page 884](#)
- [Persistent Variables and Inference of State Registers, on page 884](#)
- [Precision Bounds for Persistent Variables, on page 885](#)

M Language Example of Persistent Variable

The following M function describes an M code block which computes parity (exclusive OR) over a window of three consecutive input samples.

Persistent variables `x0` and `x1` save the state of the previous two input samples, and the current input sample is supplied by input `x2`. The M compiler infers registers for persistent variables `x0` and `x1`. In hardware, these registers form a two-element shift register with `x2` feeding the shift register input.

```
function p = parity(x2)
    persistent x0;
    persistent x1;

    if isempty(x0)
        x0 = 0;
    end
    if isempty(x1)
        x1 = 0;
    end

    p = bitxor(bitxor(x0,x1),x2);
    % next two assignments implement 2-element shift register
    x0 = x1;
    x1 = x2;
end
```

Initialization with the isempty Construct

The `isempty` function shown in the previous example initializes the two persistent variables to zero at the start of system simulation. In the synthesized hardware, the mandatory global reset performs this initialization. The global reset is connected to all state-holding elements in an SMC design. In order for simulation to correctly model the mandatory global reset, all persistent variables must be initialized to zero in the style shown above. Each persistent variable must have its own `isempty` construct containing a single assignment to zero. This is the only legal use of the `isempty` function in the SMC M code programming methodology.

Persistent Variables and Inference of State Registers

Using persistent variables does not guarantee that state registers will be inferred in the synthesized hardware. The synthesis software only infers state-holding elements when they are needed; that is, when a persistent variable can be referenced before it is assigned during the same call to the M function. In such cases, the referenced value is necessarily the state the variable had before the current function call and you need a register to hold this state in the corresponding hardware implementation. On the other hand, if a persistent variable is always assigned before it is referenced during each function call, then its previous state is never used, and state-holding hardware is unnecessary.

The following code illustrates a case where no register is inferred for a persistent variable:

```
function res = no_state(a, b)
    persistent x;

    if isempty(x)
        x = 0;
    end

    % x is assigned on every path through function, so
    % no state is inferred.
    if ((a == 0) && (b == 0))
        x = 3;
    elseif ((a == 0) && (b == 1))
        x = 2;
    elseif ((a == 1) && (b == 0))
        x = 1;
    else
        x = 0;
    end

    res = x;
end
```

Precision Bounds for Persistent Variables

To create hardware from the M-language description, the software must determine the width and signedness of the internal signals, operators, and output ports of the M code blocks. The M compiler uses the fixed-point format of the block inputs and the nature of the computations that are performed within the M function to compute the minimum required precision. The following cases describe how precision bounds are handled:

- [Precision for Persistent Variables, on page 886](#)
- [Defining Precision with `shls_convert`, on page 886](#)
- [Defining Precision with the Quantizer Object, on page 887](#)
- [Effect of Quantize Call Placement, on page 888](#)

Precision for Persistent Variables

In some cases, the software cannot statically compute the precision of persistent variables from the M function. Specifically, this happens when an assignment to a persistent variable may increase the required precision of the variable on each call of the M function. A common example of this is accumulation into a persistent variable, as shown in this code:

```
function res = accum(x)
    persistent sum;
    if isempty(sum)
        sum = 0;
    end
    sum = sum + x;
    res = sum;
end
```

In this example, the precision required to faithfully represent the state of `sum` is unbounded because you potentially require an extra bit of precision on each call to `accum`. In such a situation, the M compiler prompts you to set a bound on the precision of the persistent variable in question. You can do this with the MATLAB quantizer object as described below in [Defining Precision with the Quantizer Object](#), on page 887.

Defining Precision with `shls_convert`

You can use the SMC `shls_convert` function to define precision, instead of using the MATLAB quantizer object, as described in [Defining Precision with the Quantizer Object](#), on page 887. The advantage to using `shls_convert` is that it results in much faster simulation runtimes in MATLAB.

The following M code uses the `shls_convert` function to specify the quantization for the `cnt` counter.

```
if cnt == NoOfCoeff-1
    cnt = 0;
    out_reg = acc;    % final output
    delay_chain.shift(in);    % new sample shifted in
else
    cnt = shls_convert(cnt + 1, 'format', [ceil(log2(NoOfCoeff)) 0], 'ufixed'); % counter increment
end
```

Defining Precision with the Quantizer Object

The following shows one way to modify the persistent variable in the previous example so that it compiles. Here, the modified function `q_accum1` defines a quantizer object (`q`) and uses it to set the precision of `sum` to 8 bits (with no fractional portion).

Example 1

```
function res = q_accum1(x)

    persistent sum;
    q = quantizer([8 0], 'wrap'); % define quantizer object

    if isempty(sum)
        sum = 0;
    end

    sum = sum + x;
    sum = quantize(q, sum); % limit sum precision with quantizer q
    res = sum;

end
```

Example 2

You can also set precision bounds using other variations of the code shown above. For example:

```
function res = q_accum2(x)

    persistent sum;
    q = quantizer([8 0], 'wrap'); % define quantizer object

    sum = quantize(q, sum); % limit sum precision with quantizer q

    if isempty(sum)
        sum = 0;
    end

    sum = sum + x;
    res = sum;

end
```

Example 3

This code uses the quantizer object to specify the quantization for the cnt counter:

```
q_cnt = quantizer('ufixed','floor','wrap',[NoOfBitsPerPixel 0]);
if cnt == NoOfCoeff-1
    cnt = 0;
    out_reg = acc;    % final output
    delay_chain.shift(in);    % new sample shifted in
else
    cnt = quantize(q_cnt,cnt + 1); % counter increment
end
```

Effect of Quantize Call Placement

In general, any call to quantize that adequately constrains the precision of persistent variables allows compilation to complete. However, the results in the synthesized hardware may depend on where the call to quantize is inserted in the M function. You can see this if you compare the first two examples in [Defining Precision with the Quantizer Object, on page 887](#). While function q_accum2 in the second example also limits the precision of sum, it produces slightly different synthesis results than q_accum1.

- In q_accum1, the quantize call comes after the accumulation into sum and this quantized result is assigned to the output port res. This means that the output port res is 8 bits wide.
- In q_accum2, the quantize call occurs before the accumulation into sum and the accumulation increases the precision of sum to 9 bits. Therefore, output port res is 9 bits wide.

For more information about the quantize() and quantizer(), see [Controlling Precision and Signedness with Quantizers, on page 867](#).

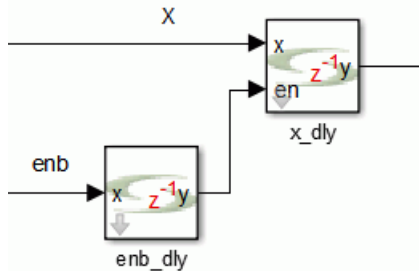
Access-Update Sequence for Persistent Variables

A persistent variable in M code can be implemented as an SMC Register component. However, it is not always implemented as a register; it might get optimized away, depending on how it is accessed. If the persistent variable is used in the M code before it is updated, it implies that the design accesses

the output of the register. If the persistent variable is used in the M code after it is updated (i.e. using the new value in the same cycle), it implies that the D input is accessed. If the variable is never used before being updated, the register inferred by the persistent variable will be optimized away.

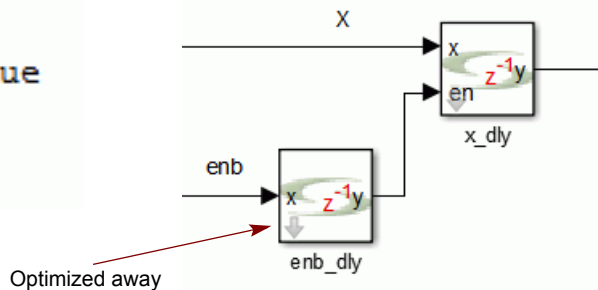
In the following example, M synthesis implements a register for the persistent variable `enb_dly`, because there is an access to its output:

```
if enb_dly == true
    x_dly = x;
end
enb_dly = enb;
```



The next example shows an input access to the persistent variable `enb_dly`. As the output of `enb_dly` is never accessed, the register is optimized away.

```
enb_dly = enb;
if enb_dly == true
    x_dly = x;
end
```

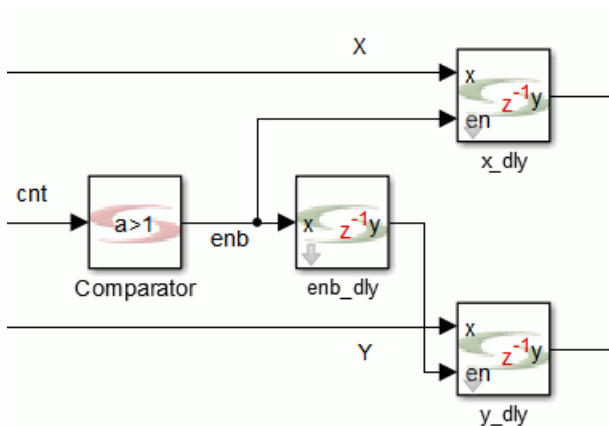


The following code shows both input and output access to `enb_dly`:

```

if enb_dly == true
    y_dly = y;
end
enb_dly = cnt > 1;
if enb_dly == true
    x_dly = x;
end

```



Conditional Assignments to Persistent Variables

Designs driven by local enables and resets tend to be written with conditional assignments to persistent variables. This practice reduces the possibilities for mapping datapath logic to DSPs for FPGA devices.

For such cases, separate datapath assignments, so that the core datapath processing registers are not enabled. Instead, use enabled registers on the periphery to control the data flow to and from the core. Such designs generally result in better QoR. The enable net ceases to be a high fanout net, and this simplifies implementation. The following code shows the separation of assignments, with the local if-else condition controlled by reset and enable:

```

%sequential assignments
temp = index;
if (reset || done_reg)
    index = 0;
    a_RowIndex = 0;
    start_held = false;
    srdyo_reg = false;
    done_reg = false;
elseif (start || start_held)
    start_held = true;
    if (index == 3) && (a_RowIndex > 0)
        dotProd = acc_reg;
        | srdyo_reg = true;
        if a_RowIndex == N
            done_reg = true;
        end
    else
        srdyo_reg = false;
    end
    if (index == N-1)
        index = 0;
        a_RowIndex = shls_convert(a_RowIndex + 1, 'format', [ceil(log2(N))+1 0], 'ufixed');
    else
        index = shls_convert(index + 1, 'format', [ceil(log2(N)) 0], 'ufixed');
    end
end
% datapath assignments have be moved out of reset and enable (start) controlled if-else
if temp == 3
    acc_reg = shls_convert(prod_reg, 'format', [(2*no_of_bits+ceil(log2(N))) 0], 'ufixed');
else
    acc_reg = shls_convert(prod_reg + acc_reg, 'format', [(2*no_of_bits+ceil(log2(N))) 0], 'ufixed');
end
prod_reg = B_row_mem*A_element_reg;
B_row_mem = Bin;
A_element_reg = ain;
end

```

M Code Examples

The following are examples of recommended coding in the M language.

- [Hardware-Aware M Code, on page 892](#)
- [Quantization of Constants, on page 893](#)

Hardware-Aware M Code

The following code is hardware-aware code for a multi-channel transposed FIR filter that can be synthesized by the M code blocks. It defines the states and describes the data flow, using the following sequence that you can use in your own designs:

- Constant definition
- State definition
- Combinational logic assignments
- Sequential logic assignments

```
function [ y] = myTransposeFIR( x,w )
    % Define constants
    no_of_channels = size(x,1);
    no_of_coeffs = size(w,2);

    out_wl = 16; % syn_parameter - output wordlength
    out_fl = 8; % syn_parameter - output fractional length

    % Define states
    persistent xs; %Declare state variable
    if isempty(xs)
        xs = zeros(no_of_channels,no_of_coeffs);
    end
    y = 0;

    % Define combinational assignments
    y=xs(:,1); % output assignment
    pp = zeros(no_of_channels, no_of_coeffs);
    for i = 1:no_of_channels
        pp(i,:) = x(i) .* w(i,:); %channel-wise partial product computation
    end
    sum_of_pp=[xs(:,2:noOfCoeffs) zeros(noOfChannels,1)]+pp; %sum of partial product
```



```
% sequential assignments
xs=shls_convert(sum_of_pp,'format',[out_wl,out_fl],'fixed','ceil','nearest');
end
```

Quantization of Constants

By default M synthesis uses maximum precision. You must specify the precision of constants to avoid generating designs that are area-inefficient. The following constant definition is inefficient because M synthesis uses the maximum word length for `coeffMatrix`, which has infinite precision values (1/3, 1/6) defined in the matrix:

```
coeffMatrix = [0 1 0 0; ...
               -1/3 -1/2 1 -1/6; ...
               1/2 -1 1/2 0; ...
               -1/6 1/2 -1/2 1/6];
```

Specifying the desired quantization in the M code will produce a more area-efficient hardware implementation. The following example shows the `shls_convert` function used for quantization:

```
coeffMatrix = shls_convert([0 1 0 0; ...
                           -1/3 -1/2 1 -1/6; ...
                           1/2 -1 1/2 0; ...
                           -1/6 1/2 -1/2 1/6], 'format', [8 6], 'fixed', 'nearest', 'saturate');
```

M Language Support for M Code Blocks

Various features that are available in the M language are supported in the Symphony Model Compiler tool:

- [Keywords, Variables, Functions, and Structures, on page 894](#)
- [Operator Support, on page 894](#)
- [Built-In Function Support, on page 895](#)
- [SMC Functions for M Code Blocks, on page 898](#)

- [M Language Limitations, on page 898](#)

Keywords, Variables, Functions, and Structures

- **Keywords**
The Symphony Model Compiler tool supports all M language keywords except for while and for; while and for loops are not supported. Each supported keyword has its usual semantics, except for try-catch-end, where the code between catch and end is never executed.
- **Variables**
Variables must be real-value, fixed-point types. The tool does not support variables and operations with complex values.
- **Functions**
Function arguments must be real scalars. The software does not support recursive function calls.
- **Structures** are not supported.

Operator Support

The following operators and special characters are supported for scalars, with their usual semantics except where noted.

M Operator	Symbol	Notes
Plus	+	
Unary plus	+	
Minus	-	
Unary minus	-	
Multiply	*	
Power	^	Only for both arguments a constant
Equal	==	
Not equal	~=	
Less than	<	

M Operator	Symbol	Notes
Greater than	>	
Less than or equal	<=	
Greater than or equal	>=	
Short-circuit logical AND	&&	
Short-circuit logical OR		
Logical NOT	~	
Decimal point	.	
Continuation	...	
Separator	,	
Semicolon	;	
Comment	%	
Assignment	=	
Quote	'	
Horizontal concatenation	[,]	Only to compose return argument list of top-level M functions.

Built-In Function Support

The Symphony Model Compiler software supports the following built-in functions and constants. They have their usual semantics, except that they are generally limited to scalar arguments. Exceptions are noted below. The hardware implied by all functions execute in one input sample period or less, in keeping with the timing model described in [Timing, on page 863](#).

Function Description

atan Supports scalar input, but not matrix or vector input.

bin2dec Argument must evaluate to a single string constant. For M synthesis, you can only use constant-valued string expressions as arguments. String variables are not allowed. Similarly, you can only use scalar-valued string expressions as arguments. Cell-arrays of strings are not allowed. However, you can use an expression that evaluates to a single string argument, as in the following example:

```
result = bin2dec(['101', '011']);
```

Here, the concatenation expression argument evaluates to the string '101011' and the function returns decimal value 43.

The SMC implementation follows the MATLAB implementation, and allows white spaces in the input string and a length restriction of 52 digits for the input string.

bitand

bitcmp The number of bits argument must be a constant.

bitget The selected bit argument must be a constant.

bitor

bitset The set bit argument must be a constant.

bitshift The shift value and number of bits to shift must be constants.

bitsll Input operand A can be a scalar, vector, or matrix.

bitsra Input operand A can be a scalar, vector, or matrix.

bitxor

ceil

convergent

cosh Supports scalar input, but not matrix or vector input.

exp Supports scalar input, but not matrix or vector input.

fix

floor

Function	Description
hex2dec	Argument must evaluate to a single string constant. For M Control, you can only use constant-valued string expressions as arguments. String variables are not allowed. Similarly, you can only use scalar-valued string expressions as arguments. Cell-arrays of strings are not allowed. However, you can use an expression that evaluates to a single string argument. The SMC implementation follows the MATLAB implementation, and does not allow white spaces in the input string and has no length restrictions for the input string.
isempty	May only be used to initialize persistent variables to zero.
length	The argument must be scalar or vector.
log	Supports scalar input, but not matrix or vector input.
log10	Supports scalar input, but not matrix or vector input.
log 2	Supports scalar input, but not matrix or vector input.
logical	
max	Supports scalar or vector input. This function only supports a single return code; [Y,I] = max(X) is not supported, only Y = max(X) is supported.
min	Supports scalar or vector input. This function only supports a single return code; [Y,I] =min(X) is not supported, only Y = min(X) is supported.
nearest	
ones	Only supports scalar/vector output. No support for matrix output.
pi	
power	Supports scalar input, but not matrix or vector input. The second argument y of the power (x, y) function must be an integer constant.
quantize	
quantizer	Only supports fixed, ufixed, wrap, saturate, floor, ceil, fix, round, convergent, and nearest. See Controlling Precision and Signedness with Quantizers , on page 867.
round	
sinh	Supports scalar input, but not matrix or vector input.

Function	Description
tan	Supports scalar input, but not matrix or vector input.
transpose	Supports vector and matrix inputs.
zeros	Only supports scalar/vector output. No support for matrix output.

SMC Functions for M Code Blocks

The Symphony Model Compiler tool includes the following functions for the M code blocks:

shls_bitrev	Reverses the bits of a specified integer (shls_bitrev , on page 590).
shls_convert	Applies the quantization rules to the input data (shls_convert , on page 592).

M Language Limitations

The following features are not supported:

- Vectors and matrices are not supported for all operations.
- Complex-valued variables, constants, operations, and related built-ins (e.g., `real`, `imag`, `angle`) are not supported. Using them results in the following error message:

```
"Complex numbers and operations not supported for M-Control synthesis."
```
- Function pointers
- Operator overloading
- Structures
- Cell arrays, except for case expression lists, which are supported. For example, `case {2, 3, 6}`.
- Logical indexing
- Function arguments `varargin` and `varargout`

- The M compiler does not support loop pipelining. All loops are unrolled by default.
- It does not support the automatic serialization of data from a parallel representation, like vectors for example. When a loop is unrolled, memory access inside a loop becomes concurrent access, so no memory component can be inferred for it. In such cases, you must serialize the data input to the M function or loop.

In addition, M language support has the following limitations:

- Function arguments (input ports) must be real scalars or vectors.
- Top-level output arguments can not have the same names as input arguments. The software renames top-level output arguments that have the same name as input arguments.
- Recursive functions are not supported.

CHAPTER 13

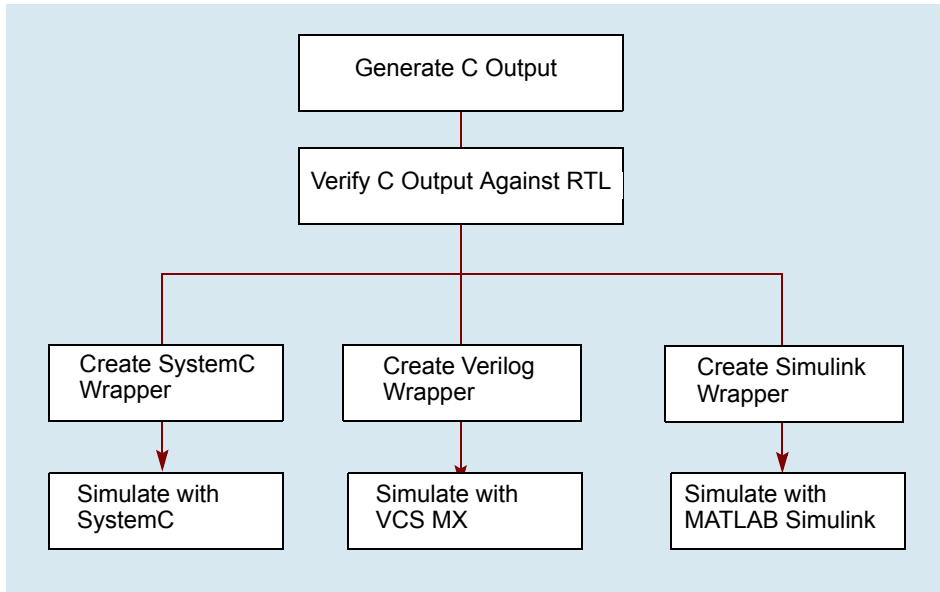
Working with C Output

This chapter describes how to generate and work with the C output from the Symphony tool.

- [Design Flow for Working with C Output, on page 902](#)
- [Generating C Output Data, on page 903](#)
- [Verifying C Output Against RTL, on page 905](#)
- [Simulating C Output, on page 906](#)
- [Supported APIs for C Output, on page 913](#)
- [C Model API Usage, on page 925](#)
- [Using C Output in Simulink, on page 927](#)
- [Using C Output with SystemC, on page 932](#)
- [Using C Output with Verilog-C Interfaces, on page 933](#)

Design Flow for Working with C Output

The following figure shows the high-level design flow for working with the C output generated by the Symphony tool:



See the following for more information:

- [Generating C Output Data, on page 903](#)
- [Verifying C Output Against RTL, on page 905](#)
- [Simulating C Output, on page 906](#)
- [Using C Output with SystemC, on page 932](#)
- [Using C Output with Verilog-C Interfaces, on page 933](#)

Generating C Output Data

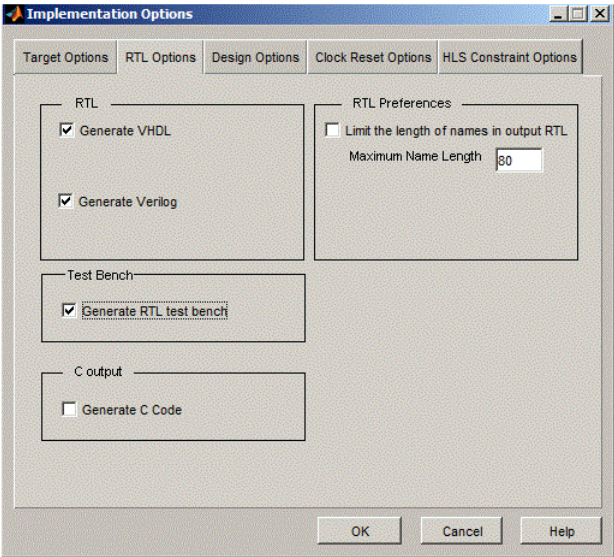
This section describe the following:

- [Generating C Output, on page 903](#)
- [Generating Output Data Files for C Output, on page 905](#)

Generating C Output

The Symphony tool lets you generate C output that is equivalent to the RTL. Use the following procedure:

1. Do the following in your design:
 - Double-click the input ports and make sure that Capture test vectors for RTL testbench is enabled.
 - Double-click the output ports and make sure that Capture test vectors for RTL testbench is enabled.
 - Double-click the SHLSTool block and open the Implementation Options dialog box by clicking New Implementation or Edit Implementation, as appropriate.
2. Do the following in the Implementation Options dialog box:
 - Go to the RTL Options tab.
 - In the RTL section, enable the language or languages you want for the RTL output. If you select one language, the tool uses that RTL (Verilog or VHDL) to generate the C output. If you specify both languages for a mixed design, the tool generates the C output based on the Verilog RTL.
 - Enable Generate RTL test bench. You must enable this option in order to generate C output.
 - Enable C output, and click OK.



3. Click Run in the SHLSTool interface to synthesize your design.

The tool generates a C-model of the RTL generated by the Symphony engine. This model is bit- and cycle-accurate at the inputs and outputs. Internally, it is optimized for higher speed simulation.

The tool generates a cout directory in the implementation directory. This contains the following files:

<code><design>.c</code>	ANSI-C source file for the Symphony design.
<code><design>_sim.c</code>	A wrapper that enables simulation of the same RTL testbench using the native environment. See Verifying C Output Against RTL, on page 905 for details.
Other .c files	Support files for building C-model executables.
*.h	Header files for building C-model executables.
Makefile	File for building the native simulation executable, <code><design>_sim</code> executable file.

Generating Output Data Files for C Output

You can also save the values at the output ports of the C model to a file. Do the following:

1. In the C model driver, set `WRITE_OUTPUT_DAT_FILE` to 1. Do this by uncommenting the following line in the C model driver file, `<design_name>_sim.c`:

```
// #define WRITE_OUTPUT_DAT_FILE 1
```

When you set this option, the tool generates an output file for each output port, using the following naming convention for the files:

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

Verifying C Output Against RTL

Once you have generated the C output, you can verify it using the GCC compiler and native simulation. Use the following procedure to verify that the C model behavior matches the behavior of the RTL testbench data.

1. Open the compiler window, and go to the implementation directory with the C output makefile.
2. Type `make` at the command line.

The compiler generates a *design_sim* executable file in the same directory. This is the C model simulation executable.

3. Run native simulation by typing *design_sim* at the command line.

The tool runs simulation and then reports the results in the `simlog.txt` file. Now that you have checked that the C output matches the behavior of the RTL output files, you can generate other wrappers and use the C output.

Simulating C Output

You can verify the generated C output by running simulation with a third-party tool. See the following for details:

- [Simulating C Output with GCC, on page 906](#)
- [Simulating C Output in Microsoft Visual Studio 2010, on page 906](#)

Simulating C Output with GCC

The following procedure shows you how to run simulation with GCC on a Linux platform. Alternatively, you can use Microsoft Visual Studio, as described in [Simulating C Output in Microsoft Visual Studio 2010, on page 906](#).

1. Install the GCC simulator. Check the release notes for recommended software versions.
2. Run synthesis and generate C output. The tool writes the C output to the cout directory.
3. To simulate the C output, do the following:
 - Go to the cout directory and type make to generate an executable file.
 - Run the executable.

Simulating C Output in Microsoft Visual Studio 2010

Use Visual Studio 2010 to simulate the C output on Windows and Linux platforms. The following describe two ways to simulate the C output using Visual Studio: with the generated project file, or by creating a new project. Make sure you have the following before you start:

- A working installation of Microsoft Visual Studio 2010
- A Windows installation of the Symphony tool

Method 1: Using the Generated Project File

1. Start Microsoft Visual Studio 2010 and open the `<model_name>.vcproj` file created by the Symphony tool.

This is a VC++ project file, and contains the project settings necessary to build the executable.

2. Build the executable (F7).

The software prompts to save the project as a solution. Click Yes.

3. Add the directory in which the C output files are generated as the working directory for the executable: Project Properties -> Configuration Properties -> Debugging -> Working Directory.

Note that the generated C output driver (i.e. `<model_name>_sim.c`) uses the DAT files in the verilog directory at the same level as the cout directory.

4. Run the executable (Ctrl+F5).

This method is straightforward for the C Output files generated by a Windows build of Symphony.

Method 2: Creating a New Project

1. Create an empty Win32 Project file for console application.

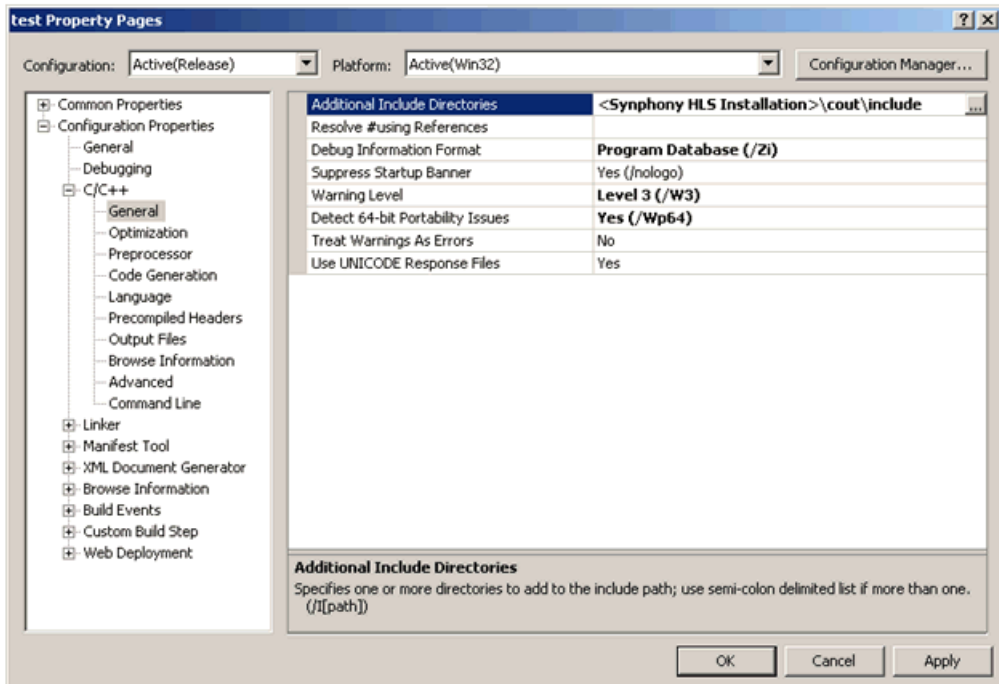
2. Add the following files to the project:

- `<model_name*>.c`
- `<model_name>_sim.c`

You can now update the project file properties as outlined in the remaining steps.

3. Select Project Properties->Configuration Properties. Then select C/C++ and do the following:

- Select General. In the Additional Include Directories field, add `<Symphony installation>\cout\include`.



- Select C/C++ -> Preprocessor from the menu on the left, and add the following definitions in the Preprocessor Definitions field:

```
CSIMOS=\win\  
TOOLDIR=\<Synphony Installation>\
```

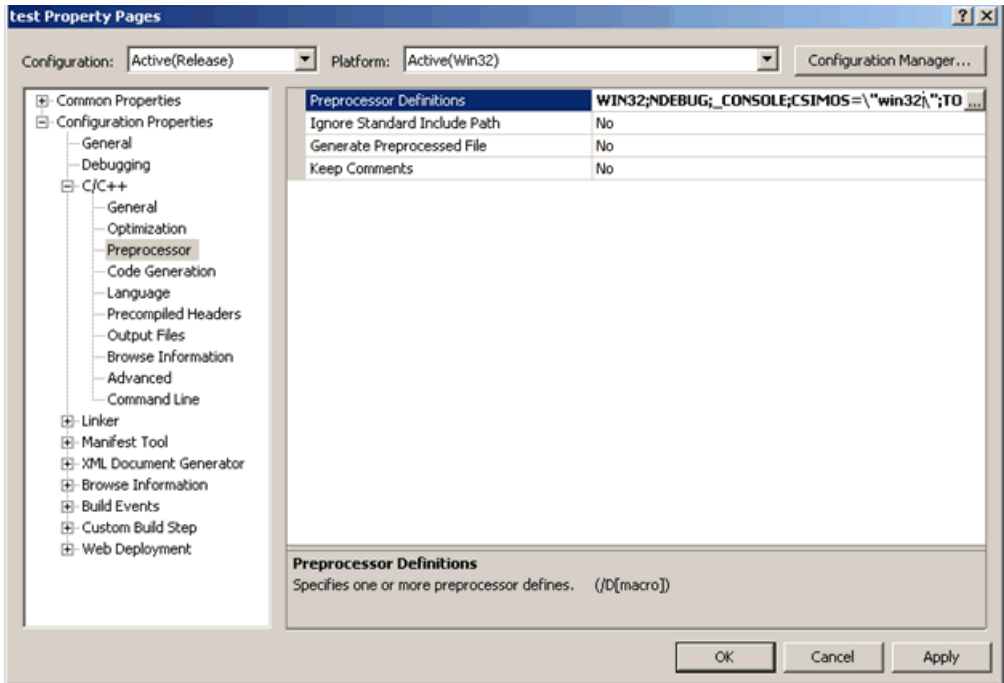
When you specify the path, use a backslash (\) before each backslash character used as a path separator to escape it. For example:

For this path...

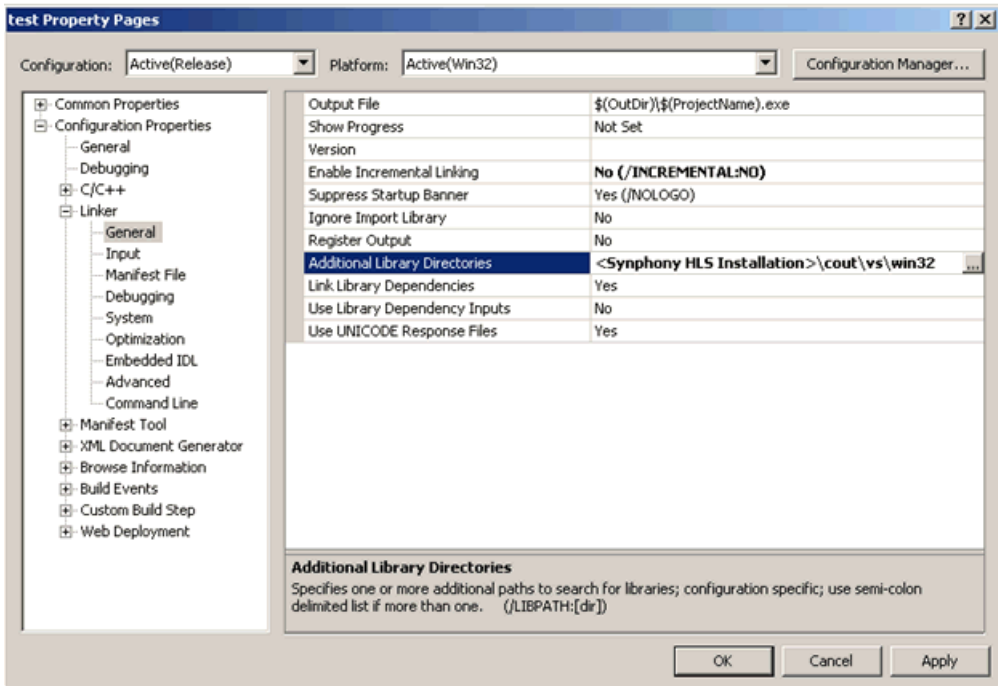
Enter this...

C:\Synopsys\SynphonyHLS200912

C:\\Synopsys\\SynphonyHLS200912

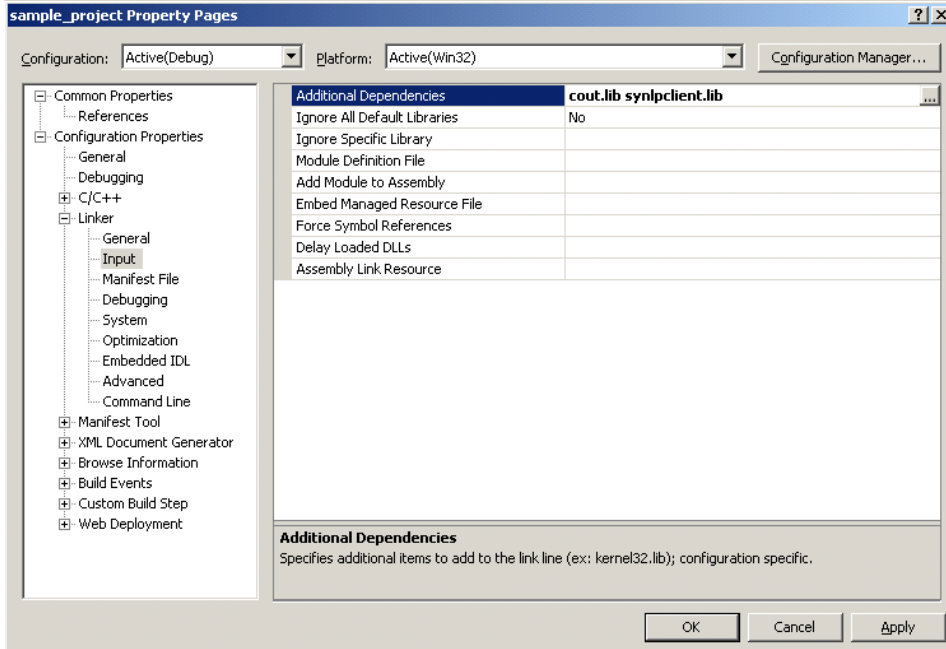


4. Select Linker in the menu on the left and do the following in this section:
 - Select General and add <Synphony Installation>\cout\vs\win32 in the Additional Library Directories field.



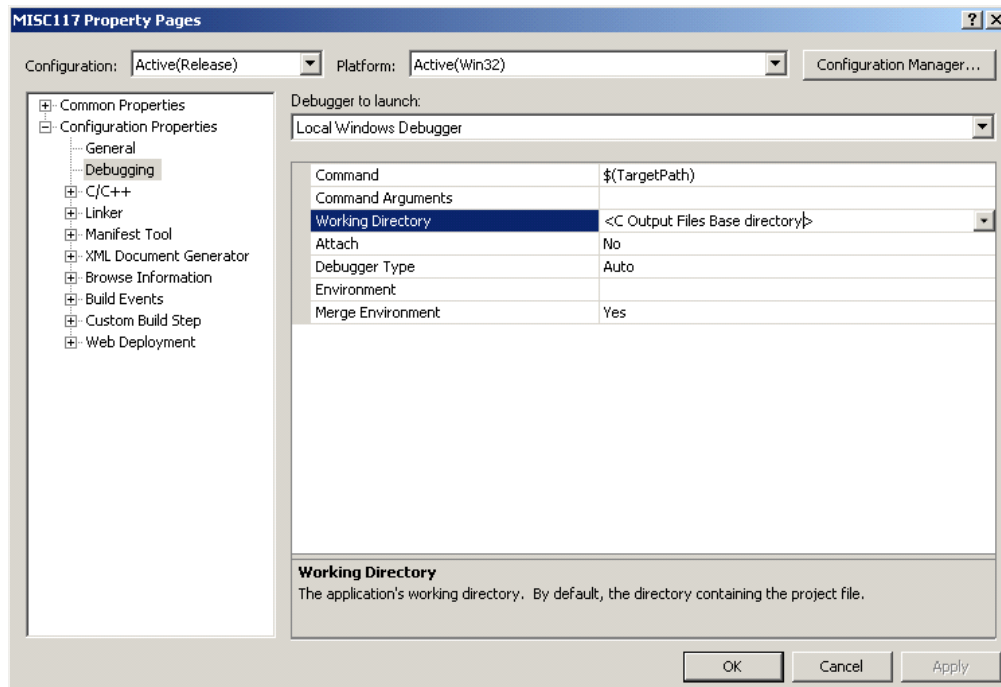
- Select Input on the left, and add the following libraries in the Additional Dependencies field:

```
cout.lib
synlpcclient.lib
```



5. Select Configuration Properties -> Debugging from the menu on the left. In the Working Directory field on the right, add the directory in which the C output files are generated.

Note that the generated C Output driver (<model_name>.sim.c) uses the DAT files in the verilog directory at the same level as the cout directory.



Supported APIs for C Output

You can access and use the C output generated by the Symphony tool with APIs. See the following for details about the supported APIs:

- [CEvent, on page 913](#)
- [int CModelDeleteEvent, on page 915](#)
- [REGISTER_DESIGN, on page 915](#)
- [void * CModelCreateInstance, on page 916](#)
- [int CModelDeleteInstance, on page 917](#)
- [int CModelSetInput, on page 918](#)
- [char * CModelGetOutput, on page 919](#)
- [int CModelEvalNext, on page 920](#)
- [CModelGetErrMsg\(\), on page 921](#)
- [int CSimGetLicense\(\), on page 923](#)
- [int CSimReleaseLicense\(\), on page 924](#)

CEvent

This API allows you to create different types of events, based on what you need.

Syntax

```
CEvent * CModelCreateEvent(char *event_name, char edge, char signal);
```

<i>event_name</i>	Pointer to the string that contains the name of the signal on which an event has to occur.
<i>edge</i>	Defines the edge for the event. It can be one of the following: <ul style="list-style-type: none">• f for falling edge• r for rising edge
<i>signal</i>	Indicates the type of signal. Currently, the tool only supports clock events as an external event, so this must be set to C, to indicate that the event is of clock type.

Return Values

Success	Valid CEvent pointer, with a reference to the created event.
Failure	NULL

Use

To create an event, first declare a pointer to the CEvent event structure. Then call this API and assign the return reference to the declared pointer.

Examples

```
CEvent *clock_rise_event;  
  
CEvent *clock_fall_event;  
  
clock_rise_event = CModelCreateEvent(clk1, 'r', 'c');  
  
clock_fall_event = CModelCreateEvent("clk2", 'f', 'c');
```

int CModelDeleteEvent

This API allows you to delete an event. When you delete an event, the event reference becomes invalid. The argument to the API is the reference to the event which should be deleted. This API frees all memory that was being used by the specified event.

Syntax

```
int CModelDeleteEvent(CEvent *);
```

Return Values

Success	0
Failure	Non-zero positive value

Example

```
CModelDeleteEvent (clock_rise_event); CModelDeleteEvent (clock_fall_event);
```

REGISTER_DESIGN

This API allows you to register the C Model with a name. You must call this API before creating any instances of the named C model.

Syntax

```
REGISTER_DESIGN(<design>)
```

Return Values

Success	0
Failure	Non-zero positive value

Use

You must first call this API once to register the <design> name. You must do this before creating an instance of <design>. If you call the API multiple times to register the same design, you get warning messages. Although these multiple calls do not affect functional simulation in any way, they do degrade performance.

void * CModelCreateInstance

This API allows you to create a C Model instance of a <design> and returns a reference to the created C Model instance.

Syntax

```
void * CModelCreateInstance (const char *design_name)
```

Return Values

Success	0
Failure	Non-zero positive value

Use

To create an instance of a C model, do the following:

1. Declare a void pointer.
2. Then call this API and assign the return reference to the declared pointer. In the SystemC wrapper, call this API in the constructor to create an instance of the C model.

Thereafter, you must use this reference to access the C Model instance.

3. Call the REGISTER_DESIGN API for the design before calling CModelCreateInstance. If not, you get a runtime error because the API tries to instantiate a model of a design that is not registered.

Example

```
void *my_model1, *my_model2, *my_model3;
REGISTER_DESIGN(<design>);           //Registers the design
my_model1= CModelCreateInstance (<design>);
my_model2= CModelCreateInstance (<design>);
my_model3= CModelCreateInstance (<design>);
```

int CModelDeleteInstance

This API allows you to free the instance of a model. After you do this, the instance reference becomes invalid. The argument to the API is the reference to the model to be freed. This API frees all the memory that was being used internally by the model.

Syntax

```
int CModelDeleteInstance (void *model);
```

Return Values

Success	0
Failure	Non-zero positive value

Use

In the SystemC wrapper, call this API in the destructor to free all memory allocated to the model instance.

Example

```
if (CModelDeleteInstance (my_model))
{ //Error Handling }
```

int CModelSetInput

This API lets you set the value `val` on the specified input port of the C model instance.

Syntax

```
int CModelSetInput(void *model, const char *port_name, const char *val);
```

<i>model</i>	The reference to the Model whose input is to be set.
<i>port_name</i>	Pointer to a string containing the name of the input port to which the value has to be set.
<i>val</i>	Pointer to a binary string containing the value to be set at input. The MSB of the value is the 0th element of the binary string.

Return Values

Success	0
Failure	Non-zero positive value

Use

To set values for different ports of the same model, call this API multiple times changing the `port_name` and `val` arguments as required.

Example

```
If(CModelSetInput (my_model,"Input1","10011"))
{ //Error Handling}
else
{
    If(CModelSetInput (my_model,"Input2","1101"))
    { //Error Handling}
    else
    {
        If(CModelSetInput (my_model," SynchronousReset ","1"))
        { //Error Handling}
    }
}
```

char * CModelGetOutput

This API allows you to read the value on the specified output port of the C model instance.

Syntax

```
char * CModelGetOutput (void *model, const char *port_name);
```

<i>model</i>	Reference to the model whose output is to be read.
<i>port_name</i>	Pointer to a string that contains the name of the output port from which the value is to be read.

Return Values

Success	A string pointer that has the value of that output port as a binary string. The 0th element of the binary string is the MSB of the output value.
Failure	NULL

Use

To get the values of multiple outputs, call this API multiple times with different *port_name*. If you do this, copy the return value to another place for future reference if needed. This is because the string pointer returned by the API is the pointer to an internal buffer that is updated by the C model each time this function is called.

Example

```
char *outval; // Temporary pointer to hold the output string
char output1_copy_fromModel[9];
char output2_copy_fromModel[32]
outval = CModelGetOutput(my_model,"output1")
if(outval == NULL)
    { //Error Handling}
strcpy(output1_copy_fromModel, outval );
outval = CModelGetOutput(my_model,"output2")
if(outval == NULL)
    { //Error Handling}
strcpy(output2_copy_fromModel, outval );
```

int CModelEvalNext

This API lets you advance the state of the model in response to the set of given events as the arguments.

Syntax

```
int CModelEvalNext(void *model, CEvent *events[], int numevents);
```

<i>model</i>	The reference to the model to be advanced.
--------------	--

<i>events</i>	A pointer to an array of events
---------------	---------------------------------

<i>numevents</i>	The number of events in the event array.
------------------	--

Return Values

Success	0
---------	---

Failure	Non-zero positive value
---------	-------------------------

Use

1. To apply a single event to the model, the arguments should be a pointer to an array containing just that single event alone and 1 as the value for numevents.
2. To apply multiple events at the same time to the model, first create an array of events that must be applied simultaneously. Then call this API with that array pointer as the events argument, and the number of events set in numevents. You can mix the rise events with fall events.

For example, you can apply 2X clock (200 MHz, for example) clocks rising edge and 1X clock's (100 MHz) falling edge at the same time to the model. You must use the CModelEvalNext API call if you want to apply both clock rising and clock falling edges to the C Model.

Example

```
CEvent* clk_rising_array[1];
CEvent* clk_falling_array[1];
clk_rising_array[0] = CModelCreateEvent("clk1",'r','c');
clk_falling_array[0] = CModelCreateEvent("clk2",'f','c');
for(clock_cnt=0; clock_cnt<10000; clock_cnt++) {
```

```

    if(!CModelEvalNext (my_model,clk_falling_array,1))
    {\Error Handling}
    If(!CModelEvalNext(my_model,clk_rising_array,1))
    {\Error Handling}
}

```

CModelGetErrMsg()

This API lets you retrieve the message when one of the other APIs signals an error by returning a non-zero code. It returns a pointer to the string containing the error message, which is stored in an internal buffer. This string only changes only when it encounters an error in the API calls. The buffer is updated as and when errors occur.

The following table shows the error messages that can occur for the APIs described in the preceding sections:

Error Message	Description
REGISTER_DESIGN	
Invalid Design Name	Returned when the C model for the design name passed is not available. This causes a compiler error.
void * CModelCreateInstance	
Design not registered	Returned in the following situations: <ul style="list-style-type: none"> When the CModelCreateInstance API is called before registering the design (before calling REGISTER_DESIGN) for the design. You type a wrong design name when you creating an instance of the model.
Design Name pointer Invalid	Returned when the pointer to the design_name string is an invalid pointer.
Out of Memory	Returned when there is not enough memory to create the model.
int CModelDeleteInstance	
Invalid Model pointer	Returned when an invalid model pointer is passed to the API.
Internal Error	Returned when there is an internal error in the C model instance.

Error Message	Description
int CModelSetInput	
Invalid Model pointer	Returned when an invalid model pointer is passed to the API.
Invalid Input port	Returned when the <code>port_name</code> string does not match any of the input ports of the C model.
Invalid Input port pointer	Returned when the pointer to the <code>port_name</code> string is an invalid pointer.
Invalid binary string pointer	Returned when the pointer to the binary string <code>val</code> is an invalid pointer.
Invalid binary string	Returned when the binary string <code>val</code> is not a valid binary string; i.e. the binary string has values other than 0 and 1.
Binary string size mismatch	Returned when the width of the binary string <code>val</code> is different from the width of the input port.
char * CModelGetOutput	
Invalid Model pointer	Returned when an invalid model pointer is passed to the API.
Invalid Output port	Returned when the output <code>port_name</code> string does not match any of the output ports of the C model.
Invalid Output port pointer	Returned when the pointer to the <code>port_name</code> string is an invalid pointer.
CEvent * CModelCreateEvent	
Invalid event_name pointer	Returned when the pointer to the <code>event_name</code> string is an invalid pointer.
Out of Memory	Returned when there is not enough memory in the stack to create the event.
Invalid Edge Info	Returned when the edge passed to the API is a character that is not <code>r</code> or <code>f</code> .
Invalid Signal Info	Returned when the signal passed to the API is a character that is not <code>c</code> .
int CModelDeleteEvent	
Invalid CEvent pointer	Returned when an invalid CEvent pointer is passed to the API.

Error Message	Description
<code>int CModelEvalNext</code>	
Invalid Model pointer	Returned when an invalid model pointer is passed to the API.
Invalid Events array pointer	Returned when an invalid pointer of CEvent array events is passed to the API.
Invalid numevents Info	Returned when the numevents passed to the API is less than or equal to 0.
Wrong Event	Returned when the event is applied on a wrong signal or on a signal that does not exist.

int CSimGetLicense()

This API lets you check out the simulation license so that the C simulation can run. You must call this API before any other APIs.

You use this API call in conjunction with other calls to provide the build information and the OS information to the C Model. The following are the other C simulation calls. You only need to call them once in the simulation.

- NCSetTooldir
- NCSetPlatform

They need to be called only once in the simulation.

The `int CSimGetLicense` API returns 0 on success, and a non-zero positive value if it fails to check out the license.

Syntax

```
CSimGetLicense(void);
```

Example

```
#ifdef TOOLDIR
// TOOLDIR is the define set in the Symphony HLS generated Makefile
// If not using the generated Makefile, define the TOOLDIR with the
// Symphony HLS build path

    NCSetTooldir(TOOLDIR);
#endif
```

```
#ifndef CSIMOS
// CSIMOS is the define set in the Symphony HLS generated Makefile
// If not using the generated Makefile, define the CSIMOS with the
// information of the OS in which the simulation is run
    NCSetPlatform(CSIMOS);
#endif

If (CSimGetLicense())
{
    //Error Handling
}
```

int CSimReleaseLicense()

This API lets you check in the simulation license for the C simulation you are using. This API checks in the license that was previously checked out with the CSimGetLicense API. Use the CSimReleaseLicense API when there are no more C Model API calls later in the simulation.

The CSimReleaseLicense API returns 0 on success and a non-zero positive value if it fails to check in the license.

Note that the tool automatically releases the C simulation license at the end of simulation, even if you do not specifically call this API. The CSimReleaseLicense API lets you release the license when none of the C Model APIs are used.

Syntax

```
CSimReleaseLicense(void);
```

Example

```
If (CSimReleaseLicense())
{
    //Error Handling
}
```


C Model API Usage

The following illustrates how to use the C Model APIs described in [Supported APIs for C Output, on page 913](#). For a complete working model, see the C Model test bench generated along with the C Model.

```

/*Include the necessary libraries */
/*Include the C Model specific libraries */
#include "ncsim.h"
#include "<design_name>.h"

Int main()
{
    /*Variable declarations*/

    /*Check out the license for the C Simulation*/
    #ifdef TOOLDIR
        NCSetTooldir(TOOLDIR);
    #endif

    #ifdef CSIMOS
        NCSetPlatform(CSIMOS);
    #endif

    If(CSimGetLicense())
        { //Error Handling}

    /*Register the design to be used*/
    REGISTER_DESIGN(<design_name>);

    /*Create an instance of design */
    <model pointer> =
        CModelCreateInstance("<design_name>");

    /*Create clock events */
    <CEvent Variable for rising edge> =
        CModelCreateEvent(<clk_name>, 'r','c');
    <CEvent Variable for falling edge> =
        CModelCreateEvent(<clk_name>, 'f','c');

    /*Set the input port of the C Model*/
    CModelSetInput(<model pointer>, "<input port_name>",
        "<Input value as binary string>");

    /*Evaluate the next state. Uses the pointer to the CEvent array
    that contains the events on different clocks at a particular
    instant.*/
    CModelEvalNext(<model pointer>, <CEvent array pointer>,
        <Number of events in the array>);

```

```
/*Get the value at the output port of the C Model */
    <output string variable > = CModelGetOutput(<model pointer>,
        "<output port name>");

/*After simulation is complete, you must delete the Model and
CEvent variables. */

/*Delete the clock Events*/
    CModelDeleteEvent(<CEvent pointer>);

/*Delete the C Model instance*/
    CModelDeleteInstance(<model pointer>);

/*Release the license if no further C Model API is used*/
    If(CSimReleaseLicense())
    {//Error Handling}
}
```

Using C Output in Simulink

The tool generates an S-function wrapper for the C model that allows you to run the C model in the Simulink environment. Using C output can significantly speed up verification times. See the following for details:

- [Using C Output to Speed up Simulink Simulations, on page 927](#)
- [Generating the Simulink C Output Wrapper, on page 928](#)

Using C Output to Speed up Simulink Simulations

You can speed up simulation runtime by using C output from the Symphony Model Compiler tool and a Simulink wrapper to significantly improve verification productivity.

Using the G-2012.09 software, tests have demonstrated runtime reductions of 3 to 10x. For the GSM DDC (Digital Down Converter) example, which you can access using the `shlsdemo` command, this methodology reduced run time from 915 seconds to 191 seconds, a 4.8x reduction in simulation runtime. For details about using C output with this example, refer to Solvnet article 036381, *Using C Output to Speed up Simulink Simulations*.

1. Capture and set up your design.
 - Capture the design using the SMC blockset.
 - Set up the implementation for your chosen hardware target as usual.
 - Enable C output generation on the RTL Options tab of the Implementation Options dialog box. See [Generating C Output, on page 903](#) for details.

2. Run SMC to synthesize the model and generate RTL.

The tool generates a C model at the same time that it creates the RTL. The C output is written to a `cout` directory under the implementation directory. The `cout` directory is parallel to the `verilog` and `vhdl` directories.

3. Run the MATLAB scripts in `<implementation_dir>/cout/simulink` to create a Simulink-wrapped C model.

First run `compile.m` and then `mdlgen.m`. For a detailed procedure, see [Generating the Simulink C Output Wrapper, on page 928](#).

4. Connect the interfaces of the subsystem to the top level ports of the design.

The interface signals must be of type double, as required by Simulink source and sink blocks. The SMC tool automatically inserts the appropriate type-casting to maintain the functionality of the C model.

Note that C output generation automatically includes the Port In and Port Out blocks, when it uses the saturation or rounding option and for registering input and output logic for the C model. When connecting the interfaces of the subsystem, you can exclude the Port In and Port Out blocks.

5. Simulate the new model, using your existing testbench.

If the simulation does not meet the required functionality and/or performance, modify the design and repeat the previous steps until the model meets your design requirements.

6. Continue with RTL synthesis and place-and-route for your hardware target.

Generating the Simulink C Output Wrapper

The tool generates Simulink wrapper files for the C output so that you can run the C model in the Simulink environment. Simulink wrapper is generated only when no optimization options are used in the implementation options. For a description of these files, see [Simulink Wrapper Files, on page 930](#). To use the wrapper files, follow this procedure:

1. In MATLAB, change the current directory to the directory with the Simulink wrapper files.

For the location of the Simulink wrapper files, see [Simulink Wrapper Files, on page 930](#).

2. Compile the S-function.
 - If you did not do this at startup, configure MATLAB to use Mex. You must do this to set the appropriate C/C++ compiler to be used by the `mex` command. Consult the Mathworks documentation (MEX-files Guide) for information about configuring the compilers. The recommended compilers are listed below:

Windows	Microsoft Visual C++ 2005 Express
Linux	Gcc 4.2.2

- Enter the following command in the MATLAB console:

```
compile
```

This command compiles a Simulink wrapper for the C model and creates a Mex file called `<design_name>_sunc.mex<platform>`. The `<platform>` is based on the platform where MATLAB is run:

Platform	<code><platform></code> Value
Windows 32-bit	w32
Linux 32-bit	glx
Linux 64-bit	a64

3. Create an mdl file to use the S-function, by typing the following command in the MATLAB console:

```
mdlgen
```

You must have an mdl file to use the S-function. The `mdlgen` command uses the `mdlgen.m` file (see [Simulink Wrapper Files, on page 930](#)) to create an mdl file called `<design_name>_subsystem.mdl`. This mdl file is a Simulink wrapper that uses the S-function and contains the required input port connections and conversion functions as a subsystem. You can use the `<design_name>_subsystem.mdl` file instead of the model compiler blocks in the Simulink model to speed up simulation time.

4. Optionally, verify the Simulink wrapper before you run it.

You verify the input and output DAT files generated by Simulink against the original model, by driving the input DAT files to the corresponding ports, capturing the data at the output ports, and then comparing the captured output DAT files with the original output DAT files in the `<implementation>/verilog` directory. The following describe this procedure:

- Convert the input DAT files into MAT files by typing the following command in the MATLAB console window:

```
dat2mat
```

See [Input and Output DAT Files, on page 930](#) for a description.

- Create an mdl file to test the Simulink wrapper by typing the following command in the MATLAB console window:

```
mdlgen_test
```

This command creates an mdl file called `<design_name>_inst_test.mdl`, which instantiates the Simulink wrapper mdl file. It also contains the components required to drive the input DAT files and capture the output DAT files.

- Use the `<design_name>_inst_test.mdl` file to simulate the mdl file with a time equal to the stop time of the original mdl. In the following example `StopTime` is equal to the stop time of the original mdl file. This command creates DAT files for each output port:

```
sim('<design_name>_inst_test', <StopTime>);
```

- Compare the captured output DAT files against the original output DAT files in the `<implementation>/verilog` directory to verify the functionality of the model.

Input and Output DAT Files

Note the following:

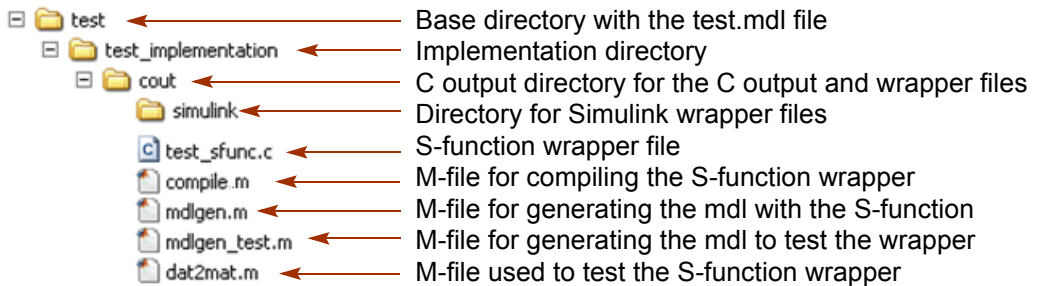
- The original output DAT files are located in the `<implementation>/verilog` directory.
- There is one DAT file for each input port. The input DAT files follow this naming convention: `Inport_<model_name>_<port_name>.dat`.
- There is one DAT file for each output port. The output DAT files follow this naming convention: `Outport_<model_name>_<port_name>.dat`.

Simulink Wrapper Files

This section describes the directory structure and the Simulink wrapper files generated for C output.

Directory Structure

The Symphony tool creates a simulink directory under the `cout` directory. This directory contains the files for working with C output in Simulink, as described in [Generating the Simulink C Output Wrapper, on page 928](#).



SMC Scripts for Simulink Wrapper

The Synphony Model Compiler tool generates the following Simulink wrapper files for use with the Synphony C output:

<code><design_name>_sfunc.c</code>	S-function wrapper for the C model generated by the Synphony tool.
<code>compile.m</code>	M file for compiling the S-function wrapper.
<code>mdlgen.m</code>	M-file that creates the mdl file for the S-function wrapper.

SMC Wrapper Verification Scripts

The SMC tool also generates the following files for testing the Simulink wrapper:

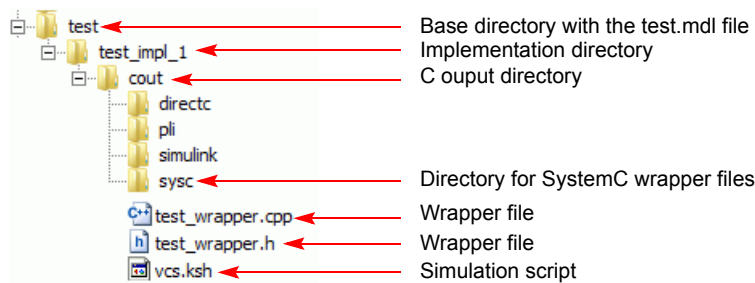
<code>mdlgen_test.m</code>	M-file that creates the mdl file that references the Simulink wrapper mdl file for testing.
<code>dat2mat.m</code>	M-file required to support Simulink wrapper testing. This file converts the input DAT files into MAT files, which is a MATLAB format.

Using C Output with SystemC

To use the C output with SystemC, you must use the SystemC wrapper generated by the Symphony Model Compiler tool. The wrapper lets you plug the C model into SystemC verification environments by providing a port interface for the C model.

1. Synthesize the SMC design as usual.

The tool automatically generates a SystemC wrapper for the C model output. The following figure shows the files generated:



The `<design>_wrapper.cpp` and `<design>_wrapper.h` files define the SystemC module interface and the code required to integrate it with the C model APIs. The wrapper is modeled using cycle accurate modeling constructs.

The tool also generates the `vcs.ksh` simulation script which simulates the C model and SystemC wrapper with a Verilog testbench that is created for the Verilog files. This script is for the VCS MX simulator, but you can use it as a reference to create scripts that work with other RTL simulators, including the OSCI simulator.

2. Simulate and verify your design.

Using C Output with Verilog-C Interfaces

The Synphony tool generates wrappers for various Verilog-C interfaces, so that you can use the Synphony C output with them. The wrappers let you plug the Synphony C model into a Verilog verification environment, because they provide a port interface for the C model. The supported Verilog-C interfaces are Verilog Procedural Interface (VPI), Programming Language Interface (PLI), and Direct C.

For details, see the following:

- [Simulating C Output with Verilog-C Interfaces, on page 933](#)
- [Verilog-C Interface Wrappers, on page 934](#)
- [Verilog-C Interface Wrapper Example, on page 936](#)
- [Verilog-C Interface Wrapper System Tasks, on page 938](#)

Simulating C Output with Verilog-C Interfaces

The following procedure shows you how to simulate the Synphony C output with Verilog-C interfaces:

1. Open the Verilog module for the Verilog-C interface you want to use.

The Verilog module and wrapper files are automatically generated by the Synphony tool. See [Verilog-C Interface Wrappers, on page 934](#) for a description of the directory structure and wrapper files. See [Verilog-C Interface Wrapper Example, on page 936](#) and [Verilog-C Interface Wrapper System Tasks, on page 938](#) for descriptions of the system tasks.

2. Check the following:
 - Check that the system tasks are defined for your needs.
 - Check that you have set the appropriate environment variables to run the script.
3. Simulate and verify your design, using the scripts generated by the Synphony tool.

The available simulation scripts vary, depending on which Verilog-C interface you are targeting and your working platform. See [Verilog-C Interface Wrappers, on page 934](#) for a description.

VPI	Linux: VCS MX Windows: Riviera Pro, ModelSim
PLI	Linux: VCS MX Windows: Riviera Pro, ModelSim
Direct C	Linux: VCS MX

Verilog-C Interface Wrappers

The following describe the wrappers generated by the Symphony tool for Verilog-C interfaces:

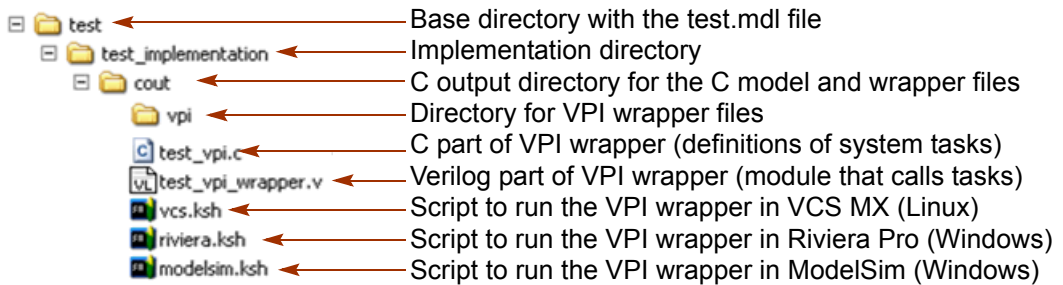
- [VPI Wrapper, on page 934](#)
- [PLI Wrapper, on page 935](#)
- [Direct C Wrapper, on page 936](#)

VPI Wrapper

The Symphony tool generates a VPI wrapper for the C model output, so that it can be used with the Verilog Procedural Interface (VPI). This wrapper consists of the following:

- A C code wrapper around the C model
This wrapper uses the required VPI routines for information exchange, as defined in the Verilog 2001 LRM, so that information can be passed between the Verilog and the C model. The C code functions in this wrapper are exported as system tasks.
- A Verilog module with the port interface for the C model
This Verilog module is provided for your convenience, and also serves as a guide to using the C model system tasks. The module calls the system tasks defined for the C model to transfer data from the port interface to the C model, and vice versa.

Directory for VPI Wrapper Files

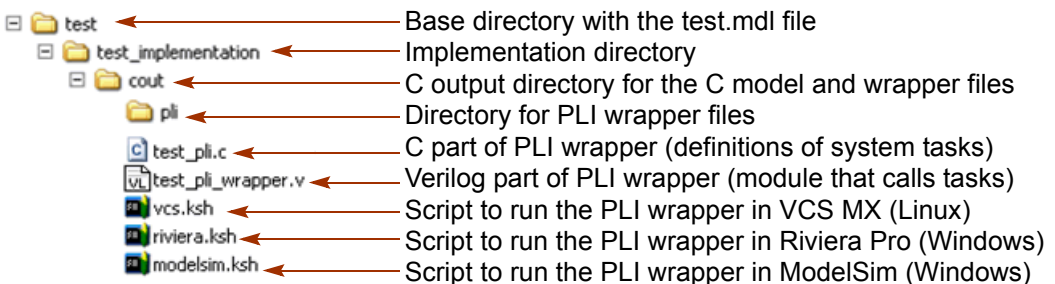


PLI Wrapper

The Symphony tool generates a PLI wrapper for the C model output, so that it can be used with the Verilog Procedural Interface (VPI). This wrapper consists of the following:

- A C code wrapper around the C model
This wrapper uses the required PLI routines for information exchange, as defined in the Verilog 2001 LRM, so that information can be passed between the Verilog and the C model. The C code functions in this wrapper are exported as system tasks.
- A Verilog module with the port interface for the C model
This Verilog module is provided for your convenience, and also serves as a guide to using the C model system tasks. The module calls the system tasks defined for the C model to transfer data from the port interface to the C model, and vice versa.

Directory for PLI Wrapper Files

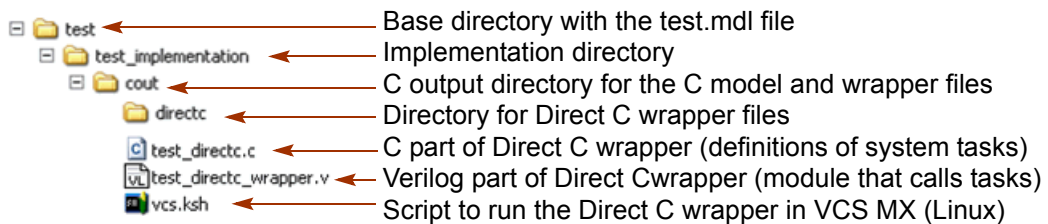


Direct C Wrapper

The Symphony tool generates a Direct C wrapper for the C model output, so that you can use C functions directly as Verilog tasks or functions. This wrapper consists of the following:

- A C code wrapper around the C model
This wrapper manages information exchange, so that information can be passed between Verilog and the C model. The C code functions in this wrapper are exported as Verilog system tasks.
- A Verilog module with the port interface for the C model
This Verilog module is provided for your convenience, and also serves as a guide to using the Direct C system tasks. The module calls the system tasks defined for the C model to transfer data from the port interface to the C model, and vice versa.

Directory for Direct C Wrapper Files



Verilog-C Interface Wrapper Example

The following is an example of a Verilog-C interface wrapper. The example uses VPI tasks, but you can extrapolate it for the other Verilog interface wrappers.

```

module top (...)
integer lic_check;
integer model_id;
reg [4:0] inport1;
reg[16:0] outport1;

...

initial
begin
#0.1;
```

```

//The first task is the Get License task. Call this to get the
//simulation license
$VPI_CSimGetLicense(lic_check);
if(lic_check)
begin
    $display("Unable to check-out Symphony C model
            license!");          $finish;
end

//Call this immediately after Get license, to register the design
$VPI_RegisterDesign("test");

//Call this to create an instance of the design. It is equivalent
//to the component instantiation of the C model
$VPI_CModelCreateInstance("test", model_id);

end

//Use this always block to set the C model input port value. This
//is like connecting the input port of the component to a signal.
//Assume that the input port name is IN1 and the signal to be
//connected to the input port is inport1.
always @(inport1) begin
    #0.2;
    $VPI_CModelSetInput(model_id, "IN1", inport1);
end

...

//This block gets the output port value from the C model.
//This is like connecting a signal to output port of the component.
// Assume that the output port name is OUT1, the signal to be
//connected to the input port is outport1, and the output port is
//synchronous to clk1.
always @(clk) begin
    #0.2;
    $VPI_CModelGetOutput(model_id, "OUT1", outport1);
end

//This always block connects the clock ports to the C model and
//evaluates the clocks. Assume that the clock port name is "clk"
//and the clock signal is clk.
always @(clk) begin
begin
    #0.3;
    $VPI_CModelSetInput(model_id, "clk", clk);
    #0.1;
    $VPI_CModelEvalNext(model_id);
end
end

```

```
//Do deleteInstance and release license just before the end of
//simulation.
initial begin
wait (endSimulation);
$VPI_CModelDeleteInstance(model_id);
$VPI_CSimReleaseLicense();
$finish;

end

...

endmodule
```

Verilog-C Interface Wrapper System Tasks

The Verilog-C interface wrappers described in [Verilog-C Interface Wrappers, on page 934](#) use system tasks to pass information between Verilog and the C model. For an example file that includes the system tasks, see [Verilog-C Interface Wrapper Example, on page 936](#).

The following describe the system tasks:

- [Get C Output Simulation License, on page 938](#)
- [Release C Output Simulation License, on page 939](#)
- [Register the Design, on page 940](#)
- [Create a Design Instance, on page 940](#)
- [Delete a Design Instance, on page 941](#)
- [Set Input Port Value, on page 942](#)
- [Get Output Port Value, on page 942](#)
- [Evaluate Clock Edge, on page 943](#)

Get C Output Simulation License

Checks out the simulation license for the C model. You must run this C model task before any other C model task.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CSimGetLicense</code>
PLI Task	<code>\$PLI_CSimGetLicense</code>
Direct C Task	<code>DIRECTC_CSimGetLicense</code>

The output of this task is an integer:

0 = Successful checkout of a license

Non-zero positive value: License checkout failed

Example

```
integer lic_check;
...
initial
begin
    $VPI_CSimGetLicense(lic_check);
    if(lic_check)
    begin
        $display("Unable to check-out Symphony C model
                license!") $finish;
    end
end
```

Release C Output Simulation License

Releases the simulation license. Use this at the end of simulation to release the simulation license checked out by the `*_CSimGetLicense` task ([Get C Output Simulation License, on page 938](#)).

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CSimReleaseLicense</code>
PLI Task	<code>\$PLI_CSimReleaseLicense</code>
Direct C Task	<code>DIRECTC_CSimReleaseLicense</code>

The output of this task is an integer:

0 = Successful release of the license

Non-zero positive value: License release failed

Example

```
integer lic_check;
...
initial
begin
...
    $VPI_CSimReleaseLicense(lic_check);
    //You can call this before the end of the simulation
    $finish;
end
```

Register the Design

Registers the C model. You must register the design before using the C model. Specify the name of the design as an argument to the system task.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_RegisterDesign <designName></code>
PLI Task	<code>\$PLI_RegisterDesign <designName></code>
Direct C Task	<code>DIRECTC_RegisterDesign <designName></code>

<designName> is the name of the design to be registered.

Example

```
$VPI_RegisterDesign("test"); // test is the name of the design
```

Create a Design Instance

Creates an instance of the C model. You must specify the name of the design as an argument to this task.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CModelCreateInstance <designName></code>
PLI Task	<code>\$PLI_CModelCreateInstance <designName></code>
Direct C Task	<code>DIRECTC_CModelCreateInstance <designName></code>

<designName> is the name of the design from which you want to create an instance.

The output of this task is an integer. The integer is a model reference number that you can use to refer to the created model.

Example

```
integer model_id;
...
$VPI_CModelCreateInstance("test", model_id);
```

Delete a Design Instance

Deletes a C model instance. You must specify the reference number of the instance to be deleted as an argument to this task.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CModelDeleteInstance <model_id></code>
PLI Task	<code>\$PLI_CModelDeleteInstance <model_id></code>
Direct C Task	<code>DIRECTC_CModelDeleteInstance <model_id></code>

<model_id> is an integer, the reference number of the instance to be deleted.

Example

```
integer model_id;
...
$VPI_CModelDeleteInstance("test", model_id);
```

```

$VPI_CModelDeleteInstance(model_id);
//Called at the end of simulation to free the memory
$VPI_CSimReleaseLicense();

$finish;

```

Set Input Port Value

Sets a value for an input port of the C model. You must specify the model reference number, the name of the port, and the value for it.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	\$VPI_CModelSetInput <model_Id> <portName> <value>
PLI Task	\$PLI_CModelSetInput <model_Id> <portName> <value>
Direct C Task	DIRECTC_CModelSetInput <model_Id> <portName> <value>

<model_id> is an integer, the model reference number for the design.

<portName> is a string, the name of the port to be updated.

<value> is the value to be assigned to the port, and can be either reg or wire.

Example

```

always @(GlobalEnable2)
begin|
    $VPI_CModelSetInput(model_id, "GlobalEnable2", GlobalEnable2);
end

```

Get Output Port Value

Gets the value from a C model output port. You must specify the model reference number and the name of the port for this system task.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CModelGetOutput <model_Id> <portName></code>
PLI Task	<code>\$PLI_CModelGetOutput <model_Id> <portName></code>
Direct C Task	<code>DIRECTC_CModelGetOutput <model_Id> <portName></code>

<model_id> is an integer, the model reference number for the design.

<portName> is a string, the name of the port to be evaluated.

The output of this task is of type `reg`. The task returns the value for the specified output port.

Example

```
always @(clk)
begin
    $VPI_CModelGetOutput(model_id, "xmitout_Q", xmitout_Q_int);
end
```

Evaluate Clock Edge

Applies clock events to the C model. You must call this system task for each clock transition.

Syntax

The syntax for this task varies, according to the Verilog interface:

VPI Task	<code>\$VPI_CModelEvalNext <model_Id></code>
PLI Task	<code>\$PLI_CModelEvalNext <model_Id></code>
Direct C Task	<code>DIRECTC_CModelEvalNext <model_Id></code>

<model_id> is an integer, the model reference number for the design.

Example

```
always @(clkDiv2,clk,clkDiv4)
begin
    $VPI_CModelEvalNext(model_id);
end
```


APPENDIX A

Blockset Summary

This chapter contains a handy summary of various block features.

SMC Block Summary

The table lists the blocks alphabetically and contains information about word length, vector, and saturation support for each block. Some blocks that support vectors also include automatic scalar expansion (ASE), which is the automatic expansion of scalar ports to match the vector ports. For full descriptions of the blocks, refer to [Chapter 2, SMC Blocks: Abs to Host Interface](#).

Block	Word Length	Vector / Matrix	Saturation	Complex Input
Abs	Input WL <= 128 bits	V	-	-
Accumulator	Input WL <= 128 bits	V	-	-
Add	Depends on the number of inputs. Sum of input WLs must be <= 128 bits.	V, M ASE	Yes	-
Binary Logic	Input WL <= 128 bits	V	-	-
Black Box	Input WL <= 128 bits	V	-	-
Block Deinterleaver	Input WL <= 128 bits	-	-	-
Interleaver	Input WL <= 128 bits	-	-	-
CIC	Output WL <= 128 bits. If R: Rate change, M: Differential delay, N: Number of stages, then output WL is calculated as follows: <ul style="list-style-type: none"> For decimator CIC: $\text{OutputWL} = \text{InputWL} + N \log_2(RM)$ For interpolator CIC: $\text{OutputWL} = \text{InputWL} + \log_2((RM)^{N/R})$ 	V ASE	-	-
Commutator	Input WL <= 128 bits	V	-	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
Comparator	Internal size must be less than or equal to 128. Find the internal size by aligning inputs with respect to the binary point. For example, with two inputs <code>sfix52_16</code> and <code>sfix40_21</code> , the internal size is $\text{MAX}(52-16, 40-21) + \text{MAX}(16, 21)$ which is 57.	V	-	-
Concat	Sum of input WLs ≤ 128 bits	V	-	-
Configurable FFT/IFFT	Depends on FFT size and scaling. FFT reports when internal WL exceeds 128.	V ASE	Yes	-
Constant	Output WL ≤ 128 bits	V, M	-	-
Convert	Input WL ≤ 128 bits	V, M	Yes	-
Convolutional Deinterleaver	Input WL ≤ 128 bits	V	-	-
Convolutional Encoder	Input WL ≤ 128 bits	-	-	-
Convolutional Interleaver	Input WL ≤ 128 bits	V	-	-
CORDIC Exp	Input WL ≤ 120 bits Mantissa ≤ 32 bits Exp ≤ 16 bits	-	-	-
CORDIC Log	Input WL ≤ 120 bits Output WL ≤ 54 bits	-	-	-
CORDIC Polar	Input WL ≤ 52 bits Output WL ≤ 53 bits	-	-	-
CORDIC Rotate	Input WL ≤ 120 bits Number of iterations ≤ 53	-	-	-
CORDIC SinCos	Input WL ≤ 120 bits Number of iterations ≤ 53 Number of iterations $<$ output WL	-	-	-
CORDIC Sqrt	Input WL ≤ 64 bits	-	-	-
Counter	Input WL ≤ 128 bits	V ASE	-	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
CRC Generator	Input/ Output WL = 1 bit unsigned	V	-	-
DDS	Output WL <= 128 bits. Waveform frequency precision, waveform phase precision, and waveform magnitude precision must be no less than 2^{-128} .	-	-	-
Decommutator	Input WL <= 128 bits	V		-
Delay	Input WL <= 128 bits	V, M	-	-
Demux	Input WL <= 128 bits	V	-	-
Depuncture	Input WL <= 128 bits	-	-	-
Differentiator	Input WL depends on output format: <ul style="list-style-type: none"> • For Full Precision, <= 127 bits • For Automatic, <= 128 bits 	V ASE	Yes	-
Divider	Input WL <= 128 bits	V	Yes	-
DivMod	Input WL <= 128 bits	V	-	-
Downsample	Input WL <= 128 bits	V	-	-
Edge Detector	Input WL <= 128 bits	-	-	-
Extract	Input WL <= 128 bits	V, M	-	-
FDATool	Not Applicable	N/A	N/A	-
FFT	Depends on FFT size and scaling. FFT reports when internal WL exceeds 128.	V ASE	-	-
FIFO	Input WL <= 128 bits	V ASE	-	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
FIR	Internal data path WL <= 128 bits. According to data path format (Automatic, Full precision, Specify), FIR output is calculated with maximum possible input data and should be less than or equal to 128. See SMC FIR, on page 226 .	V ASE	-	-
FIR Engine	Same as Synphony FIR	Coeffs only	-	-
FIR Rate Converter	Same as Synphony FIR	V	-	-
Gain	Input WL + Gain Fraction Length + Ceil(log2(Gain Value)) <= 128	V, M	-	-
Gold Sequence Generator	Output WL = 1 bit unsigned	-	-	-
HLS Subsystem	Input WL <= 128 bits	V, M	N/A	-
IIR	Internal WL <= 128. See the FIR entry above for details.	-	-	-
In	Input WL <= 128 bits	V, M	-	-
Integrator	Input WL <= 128 bits	V ASE	Yes	-
Inverter	Input WL <= 128 bits	V	-	-
Log	Input WL <= 128 bits	V	-	-
M Control	Input WL <= 128 bits	-	-	-
Matrix Mult	Output WL <= 128 bits	M	Yes	-
MinMax	Input WL <= 128 bits	V	-	-
Moore State Machine	Output WL <= 128 bits	-	-	-
Mult	(First input WL + second input WL) <= 128 bits	V, M ASE	Yes	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
Mux	Input WL <= 128 bits	V	-	-
Negate	Input WL <= 126 bits	V	-	-
Out	Input WL <= 128 bits	V, M	-	-
Parallel to Serial	Ceil(Input WL/Number of Packets * Number of Packets <= 128	-	-	-
Permutation	Input WL <= 128 bits	-	-	-
PN Sequence generator	Output WL = 1 bit unsigned	-	-	-
Port In	Input WL <= 128 bits	V, M	Yes	-
Port Out	Input WL <= 128 bits	V, M	Yes	-
Pow	Input WL <= 128 bits. Input word lengths must allow power calculation to fit into 128 bits without data loss. If BaseWL is base word length, ExpIL is exponent integer length, and SB is sign bit value (SB=1 if signed, SB=0 if unsigned), then the output is calculated as follows: $(\text{BaseWL} - \text{SB}) * (2^{\text{ExpIL}}) + \text{SB}$ Thus, if base is in <code>sfix6.4</code> and exponent is in <code>sfix5.2</code> , the output is in $5 * 7 + 1 = 36$ bits.	-	-	-
Pulse Generator	Output WL = 1 bit unsigned	-	-	-
Puncture	Input WL <= 128 bits	V	-	-
RAM	Input WL <= 128 bits	V ASE	-	-
Ramp	Input WL and Output WL <= 128 bits	-	-	-
Random	Output WL <= 19 bits	-	-	-
Recast	Input WL <= 128 bits	V, M	-	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
Reed-Solomon Decoder	Input WL Minimum 3 bits Maximum 12 bits	V	-	-
Reed-Solomon Encoder	Input WL Minimum 3 bits Maximum 12 bits	V	-	-
Register	Input WL <= 128 bits	V, M ASE	-	-
Reshape	Input WL <= 128 bits Output WL = same as Input WL	V, M	-	-
RFIR	Same as Symphony FIR Engine	Coeffs only	-	-
ROM	Input WL <= 128 bits	V	-	-
RTL Encapsulation	Input WL <= 128 bits	No	-	-
Sample and Hold	Input WL <= 128 bits	-	-	-
Saturate	Input WL <= 128 bits	V	Yes	-
Sequence	Output WL <= 128 bits	-	-	-
Serial to Parallel	Input WL * Number of packets <= 128 bits	-	-	-
Shift Register	Input WL <= 128 bits	V, M ASE	-	-
Shifter	Input WL <= 127 bits	-	-	-
SHLSTool	Not Applicable	N/A	N/A	-
Sign	Input WL <= 128 bits	-	-	-
SinCos	Input WL <= 128 bits Only 18 bits of the fraction are supported. Output is truncated to 54 bits.	V	-	-
Smart Black Box	Input WL <= 128 bits	V	-	-

Block	Word Length	Vector / Matrix	Saturation	Complex Input
Sqrt	Input WL <= 127 bits	V	-	-
Subsystem	Not Applicable	N/A	N/A	-
Sum of Products	Input WL + Gain Fraction Length + Ceil(log2(Gain Value)) <= 128	V	Yes	-
Switch	Input WL <= 128 bits	V	-	-
SynFixPtTool	Not Applicable	N/A	N/A	-
Test Vector Capture	N/A	N/A	N/A	N/A
Upsample	Input WL <= 128 bits	V	-	-
Vector Concat	Output WL <= 128 bits, being the max of Input WLs.	V	-	-
Vector Expand	Input WL <= 128 bits. Output WL is the same as Input WL.	V	-	-
Vector Extract	Input WL <= 128 bits. Output WL is the same as Input WL.	V	-	-
Vector Split	Input WL <= 128 bits. Output WL is the same as Input WL.	V	-	-
Viterbi Decoder	Input WL <= 128 bits	-	-	-

Index

A

- Abs block [42](#)
- absolute value
 - calculating [42](#)
- Accumulator block [44](#)
- Active-HDL [855](#)
- adaptive filters
 - using [760](#)
 - using vectors [775](#)
- adaptive FIR [448](#)
- Add block [48](#)
- adder
 - block [48](#)
- adders
 - resource usage [850](#)
- add_register_and_balance_parallel_paths [622](#)
- advanced timing engine *See* advanced timing mode
- advanced timing mode
 - FPGAs [638](#)
 - Synplify Pro executable [639](#)
- AMBA AXI4 [738](#)
- AND operation
 - Binary Logic block [54](#)
- antisymmetric coefficients [254](#)
- APB protocol [743](#)
- APIs for C output [913](#)
- areabased_fir_arch_selection_atm constraint [623](#)
- asynchronous resets
 - RTL code, Symphony [685](#)
 - testbenches, Symphony [686](#)
- asynchronous resets, Symphony [684](#)
- ATM
 - constant propagation [731](#)
- automatic scalar expansion [946](#)
- AVLON-MM protocol [745](#)
- AXI4-Lite [738](#)

B

- Binary Logic block [66](#)
- binary logic functions [53](#)
- binary point
 - interpretation of fixed-point numbers [99](#)
- bit growth, FFT2 [213](#)
- bit reversal
 - M compiler [590](#)
- Black Box block [56](#)
- black boxes
 - advantages [777](#)
 - folding [785](#)
 - multichannelization [785](#)
 - port interface [779](#)
 - retiming [785](#)
 - retiming constraint [627](#)
 - RTL [780](#)
 - using [777](#)
 - using variables for names [60](#)
- block [806](#)
- Block Deinterleaver block [62](#)
- Block Interleaver block [64](#)
- block parameters
 - Abs [43](#)
 - Accumulator [45](#)
 - Add [49](#)
 - Binary Logic [54](#)
 - Black Box [58](#)
 - Block Deinterleaver [63](#)
 - Block Interleaver [65](#)
 - CIC [67](#)
 - Commutator [78](#)
 - Comparator [84](#)
 - Concat [87](#)
 - Configurable FFT/IFFT [90](#)
 - Constant [95](#)
 - Convert [100](#)
 - Convolutional Deinterleaver [105](#)
 - Convolutional Encoder [108](#)
 - Convolutional Interleaver [110](#)
 - CORDIC Div [112](#)

CORDIC Log [114](#)
CORDIC Polar [116](#)
CORDIC Rotator [119](#)
Counter [133](#)
CRC Generator [141](#)
DDS [145](#)
DDS2 [152](#)
Decommutator [164](#)
Delay [170](#)
Demux [172](#)
Depuncture [174](#)
Differentiator [177](#)
Divider [180](#)
DivMod [184](#)
Downsampler [193](#)
Edge Detector [198](#)
Extract [201](#)
FFT [206](#)
FFT2 [215](#)
FIFO [222](#)
FIR [229](#)
FIR Engine [237](#)
FIR Rate Converter [242](#)
FIR2 [248](#)
Flow Control Buffer [279](#)
Gain [312](#)
Gold Sequence Generator [317](#)
HLS Subsystem [322](#)
IIR [342](#)
Integrator [347](#)
Inverter [351](#)
Log [354](#)
M Control [357](#)
Matrix Mult [361](#)
Mealy State Machine [365](#)
MinMax [368](#)
Moore State Machine [370](#)
Mult [379](#)
Mux [383](#)
Negate [387](#)
Parallel to Serial [393](#)
Permutation [395](#)
PN Sequence Generator [397](#)
Port In [400](#)
Port Out [404](#)
Pow [407](#)
Pulse Generator [410](#)
Puncture [413](#)
RAM [416](#)
Ramp [420](#)
Random [423](#)

Recast [425](#)
Reed-Solomon Encoder [438](#)
Register [442](#)
Reshape [444](#)
RFIR [449](#)
ROM [454](#)
RTL Encapsulation [458](#)
Sample and Hold [466](#)
Saturate [468](#)
Sequence [471](#)
Serial to Parallel [474](#)
Shift Register [478](#)
Shifter [485](#)
Sign [508](#)
SinCos [125](#), [514](#), [551](#)
Smart Black Box [534](#)
specifying [643](#)
Sqrt [541](#)
Sum of Products [545](#)
Switch [549](#)
Test Vector Capture [556](#)
Upsampler [559](#)
Vector Concat [563](#)
Vector Expand [567](#)
Vector Extract [570](#)
Vector Split [573](#)
Viterbi Decoder [577](#)
block parameters
 Host Interface [329](#)
block tagging [649](#)
 rules [650](#)
blocks
 Abs [42](#)
 Accumulator [44](#)
 Add [48](#)
 adding (Synphony) [642](#)
 alphabetical list of Synphony blocks [39](#)
 Binary Logic [53](#)
 Black Box [56](#)
 Block Deinterleaver [62](#)
 Block Interleaver [64](#)
 casting output data type [697](#)
 CIC [66](#)
 CIC2 [70](#)
 Commutator [77](#)
 Comparator [84](#)
 Concat [86](#)
 Configurable FFT/IFFT [88](#)
 consolidating in RTL [730](#)
 Constant [94](#)
 Convert [98](#)

- Convolutional Deinterleaver 104
- Convolutional encoder 106
- Convolutional Interleaver block 109
- CORDIC Exp 111
- CORDIC Log 113
- CORDIC Magnitude 115
- CORDIC Rotator 117
- CORDIC SinCos 124
- CORDIC Sqrt 126
- CORDIC2 127
- Counter 131
- CRC Generator 138
- Decommutator 163
- Delay 169
- Demux 171
- Depuncture 173
- Differentiator 176
- Divider 179
- DivMod 183
- Downsampler 191
- Edge Detector 197
- Extract 200
- FDATool 200, 203, 424
- FFT 204, 211
- FIFO 220
- FIR 226, 241
- FIR Engine 235
- FIR2 246
- Fixed to FP 295
- FP Add 286
- FP Compare 290
- FP Constant 292
- FP Fused Mult Add block 298
- FP Mult 301
- FP Port In 303
- FP Port Out 306
- FP to Fixed 309
- Gain 275, 311
- Gold Sequence Generator 315
- Host Interface 326
- IIR 340
- Integrator 346
- Inverter 350
- Leading Zero Counter 352
- Log 354
- M Control 356
- Matrix Mult 360
- Mealy State Machine 364
- MinMax 367
- Moore State Machine 369
- Moving Average Filter 372
- Mult 378
- Mux 381
- Negate 386
- Out 388
- Parallel FIR 389
- Parallel to Serial 392
- parameterized 816
- Permutation 394
- PN Sequence Generator 396
- Port In 399
- Pow 405
- Pulse Generator 409
- Puncture 412
- RAM 414
- Recast 424
- Reed-Solomon Decoder 428
- Reed-Solomon Encoder 435
- Register 441
- RFIR 448
- ROM 453
- RTL Encapsulation 456
- Sample and Hold 465
- Saturate 467
- Serial to Parallel 473
- Shift Register 476
- Shifter 484
- shlsdoc function 596
- SHLSTool 486
- Sign 507
- SimControl Tool 550
- SinCos 513
- SinCos2 516
- Single Clock Downsample 526
- Single Clock Upsample 529
- Smart Black Box 532
- Sqrt 539
- Sum of Products 544
- Switch 548
- SynFixPtTool 554
- Symphony libraries 28
- Test Vector Capture 556
- types 800
- Upsample 557
- Vector Concat 561
- Vector Expand 567
- Vector Extract 570
- Vector Split 572
- Viterbi Decoder 574
- blocksets
 - custom. *See* custom blocksets 804
 - shllib function 597

Symphony libraries [28](#)
bus protocols [738](#)

C

C output

- APIs [913](#)
- design flow [902](#)
- directory [904](#)
- files [904](#)
- generating [903](#)
- option for generating [493](#)
- output port values file [905](#)
- simlog.txt file [905](#)
- simulation speedup [927](#)
- Simulink wrapper [928](#)
- Simulink wrapper scripts [931](#)
- Simulink wrapper verification scripts [931](#)
- System C [932](#)
 - verifying against RTL [905](#)

Cadence IUS54 [855](#)

casting [697](#)

CEvent API [913](#)

channels

- creating multiple [674](#)

char * CModelGetOutput API [919](#)

CIC block [66](#)

CIC filters [66](#)

CIC2 block [70](#)

CIC2 filters [70](#)

clock alignment [721](#)

clock domains [682](#)

clock reset options [497](#)

clocks [682](#)

- dedicated clocking [499](#)
- multi-rate designs [724](#)
- single clock source [499](#)

CModelGetErrMsg() API [921](#)

coefficient variables [588](#)

coefficients

- syn_get_coefs [604](#)

command line commands

- Symphony information [598](#)

Commutator block [77](#)

Comparator block [84](#)

comparison operations [85](#)

Concat block [86](#)

Constant block [94](#)

constant propagation [731](#)

constraints [622](#)

- add_register_and_balance_parallel_paths [622](#)
- area_abased_fir_arch_selection_atm [623](#)
- file description [620](#)
- FIR architecture [228](#)
- fir_architecture [623](#)
- forward-annotation [636](#)
- mult_cycle_path [624](#)
- pattern_annotation [626](#)
- retimie_across_blackbox [627](#)
- retiming_scale_factor [628](#)
- ROM reset [453](#)
- shls_retiming_lock [169](#), [628](#)

constraints file

- description [620](#)

Control Logic library [30](#)

Convert block [98](#)

- examples [99](#)

Convolutional Deinterleaver block [104](#)

Convolutional Interleaver block [109](#)

CORDIC algorithms

- Symphony [701](#)

CORDIC Div block [106](#)

CORDIC Exp block [111](#)

CORDIC Log block [113](#)

CORDIC Magnitude block [115](#)

CORDIC Rotator block [117](#)

CORDIC SinCos block [124](#)

CORDIC Sqrt block [126](#)

CORDIC2 block [127](#)

co-simulation

- configuration [842](#)

cosine

- CORDIC SinCos block [124](#)
- SinCos block [513](#)

Counter block [131](#)

CSimOut_.dat file [905](#)

custom block

- creating [805](#)

custom blocks

- advantages [800](#)
- CIC [66](#)
- DDS [143](#)
- description [800](#)
- differentiator [176](#)
- FIR Rate Converter [241](#)

- Flow Control Buffer [275](#)
- Integrator [346](#)
- masks [806](#)
- MinMax [367](#)
- parameterized blocks [816](#)
- Ramp [419](#)
- Random [422](#)
- Sequence [470](#)
- Sign [507](#)
- syn_unlink function [617](#)
- unlinking [617](#)
- custom blocksets [804](#)
- custom libraries
 - converting [825](#)

D

- DAT files, Simulink wrapper [930](#)
- data types [696](#)
 - casting output [697](#)
 - casting output data type [697](#)
 - converting from Simulink to Symphony [605](#)
 - converting input data type [98](#)
 - displaying for model ports [696](#)
 - fixed-point in Symphony [696](#)
 - output data type [697](#)
 - Synopsys implementation [696](#)
 - validating fixed-point algorithm [834](#)
 - validating floating-point algorithm [834](#)
 - word length for RTL generation [23](#)
- DDS block [143](#)
- DDS2 block [149](#)
- decimation
 - defined [717](#)
- decimators
 - CIC [67](#)
 - FIR polyphase [243](#)
- Decommutator block [163](#)
- define_attribute constraint syntax [620](#)
- Delay
 - retiming constraint [169](#), [628](#)
- delay
 - defined [717](#)
- Delay block [169](#)
- delays
 - Register block [441](#)
- demos
 - shlsdemo function [594](#)
 - Symphony [594](#)

- demultiplexer
 - Demux block [171](#)
 - Vector Split block [572](#)
- Demux block [171](#)
 - using in Symphony [562](#)
- design flows
 - Symphony FPGA [20](#), [23](#)
- design options [495](#)
- Differentiator block [176](#)
- Digital down converter demo [594](#)
- direct digital synthesizer. *See* DDS block
- direct digital synthesizer. *See* DDS2 block
- discrete time differentiation [176](#)
- discrete time integration [346](#)
- Divider block [179](#)
- DivMod block [183](#)
- doc shls command [598](#)
- Downsample
 - waveforms [195](#)
- downsample
 - offset [719](#)
- Downsample block [191](#)
 - multicycle constraint [633](#)
- DSP Basics library [31](#)

E

- Edge [198](#)
- entities
 - block consolidation [730](#)
 - subsystem consolidation [729](#)
- error messages
 - C output API [921](#)
- error-correction
 - Reed-Solomon codes [436](#)
 - Reed-Solomon Decoder block [428](#)
 - Reed-Solomon Encoder block [435](#)
- examples
 - Convert block binary point [99](#)
- exponents
 - calculating [111](#)
- Extract block [200](#)

F

- Fast Fourier Transform. *See* FFT
- FDATool block [203](#)
- FFT
 - FFT block [204](#)

- FFT2 block [211](#)
- parallel processing [211](#)
- FFT2 block [211](#)
- FIFO
 - waveforms [223](#)
- FIFO block [220](#)
- filter coefficients [760](#)
- Filtering library [31](#)
- filters
 - CIC [66](#)
 - CIC2 [70](#)
 - differentiator [176](#)
 - FIR [226](#)
 - FIR2 block [246](#)
 - IIR [340](#)
 - Integrator [346](#)
 - polyphase [241](#)
- FIR architecture selection [226](#)
- FIR architectures, FIR2 block [246](#)
- FIR block [226](#)
- FIR Engine block [235](#)
 - adaptive FIR application [448](#)
 - reloadable FIR application [448](#)
- FIR filters
 - implementing [764](#)
- FIR Rate Converter block [241](#)
- FIR2 block [246](#)
- fir_architecture constraint [623](#)
- FIRs
 - adaptive [448](#)
 - reloadable [448](#)
- fixed point data type
 - Synphony [695](#)
- Fixed Point Settings toolbox
 - fixed-point algorithm [834](#)
 - floating-point data type [834](#)
 - full-accuracy algorithm [834](#)
 - plotting [835](#)
- Fixed to FP block [295](#)
- fixed-point algorithm
 - comparing to floating-point [835](#)
- fixed-point data type
 - setting [832](#)
 - setting options [554](#)
 - Synphony [696](#)
- fixed-point numbers
 - importance of binary point in scaling [99](#)
- floating-point algorithm
 - comparing to fixed-point [835](#)
- Flow Control Buffer
 - waveforms [282](#)
- Flow Control Buffer block [275](#)
- folding
 - black boxes [785](#)
 - effect on FIR architecture [227](#)
 - effect on forward-annotation of retiming constraints [629](#)
 - optimizing with [662](#)
 - option [501](#)
 - pattern annotation constraint [626](#)
 - pattern usage report [673](#), [851](#)
 - register balancing [622](#)
 - smart black boxes [785](#)
- forward-annotation
 - multicycle constraints [626](#), [636](#)
- FP Add block [286](#)
- FP Compare block [290](#)
- FP Constant block [292](#)
- FP Fused Mult Add block [298](#)
- FP Mult block [301](#)
- FP Port In block [303](#)
- FP Port Out block [306](#)
- FP to Fixed block [309](#)
- fraction length
 - converting from Simulink to Synphony [605](#)
- frame, defined [718](#)
- functions
 - Synphony [589](#)

G

- Gain block [311](#)
- generic interface protocol [748](#)
- global reset options [495](#)
- global resets
 - Synphony RTL [685](#)
 - Synphony testbench [686](#)

H

- handshake
 - APB [744](#)
 - AVLON_MM [746](#)
 - AX14 [739](#)
 - generic protocol [749](#)
- help [26](#)
- help shls command [598](#)

- hex format
 - reading ROM data 610
- hex records 610
- hierarchy
 - creating (tutorial) 792
 - preserving subsystems 728
 - viewing 794
- histogram
 - M Control demo 594
- HLS constraint file
 - creating 653
- hls constraint options 505
- HLS Subsystem block
 - parameters 319
 - shls.log 791
 - simulation process 320
 - synthesis process 321
 - using 786
- Host Interface block 326
 - synthesizing 678

I

- I/O blocks
 - pattern folding 665
- icons
 - custom block 808
- IIR block 340
- implementation clock 682
- implementation options 490
- implementations
 - clock reset options 497, 505
 - creating 644
 - deleting 489
 - design options 495
 - naming 491
 - output options 492
- In block 345
- info shls command 598
- inherit port variables 588
- input ports 399
- input sign values 507
- input signals
 - sample rate propagation 717
- inputs
 - calculating logarithm, CORDIC Log block 113

- calculating logarithm, Log block 354
- decreasing sample rate 191
- delay 169
- increasing sample rate 557
- interleaving 64, 109
- permutations 64
- shifting, Shifter block 484
- shuffling 64, 109
- square root 539
- square root, CORDIC 126
- installation directory
 - shlsroot function 599
- int CModelDeleteEvent API 915
- int CModelDeleteInstance API 917
- int CModelEvalNext API 920
- int CModelSetInput API 918
- int CSimGetLicense API 923
- int CSimReleasLicense API 924
- Integrator block 346
- interpolation
 - defined 718
- interpolators
 - CIC 68
 - FIR polyphase 242, 311
- Inverter block 350
- IP
 - embedding 532

L

- latency
 - sample rate definition 718
- latency, fixed 504
- Leading Zero Counter block 352
- libraries
 - adding custom 806
 - creating custom 805
 - loading custom 824
 - Symphony blocks 28
- Log block 354
- logarithm
 - CORDIC Log block 113
 - Log block 354
- logic synthesis
 - FPGA 856

M

M built-in functions

 Synphony support [895](#)

M Control

 reversing bit order [590](#)

M Control block [356](#)

M Control block, Synphony

 data types [864](#)

 demo [594](#)

 ports [863](#)

 specifying M functions [858](#)

 timing [863](#)

 using in a design [858](#)

M functions

 creating in Synphony [859](#)

 editing in Synphony [859](#)

 shls_convert [592](#)

 specifying for Synphony M Control [858](#)

 Synphony control logic with M Control
 block [356](#)

 Synphony support [894](#)

M keywords

 Synphony support [894](#)

M language

 Synphony caveats [899](#)

 Synphony support [893](#)

 Synphony unsupported features [898](#)

M operators

 Synphony support [894](#)

M special characters

 Synphony support [894](#)

M structures

 Synphony support [894](#)

M variables

 Synphony support [894](#)

masks

 custom blocks [806](#)

masks, subsystem [793](#)

Math Functions library [33](#)

MATLAB

 starting Synphony [641](#)

 starting Synphony FPGA [23](#)

 version for running Synphony [19](#)

Matrix Mult block [360](#)

matrix multiplier [360](#)

Mealy State Machine block [364](#)

Memories library [34](#)

memory queues [220](#)

MinMax block [367](#)

ModelSim [855](#)

modules

 block consolidation [730](#)

 subsystem consolidation [729](#)

Moore State Machine block [369](#)

Moving Average Filter block [372](#)

Mult block [360](#), [378](#)

multichannelization

 black boxes [785](#)

 optimizing with [674](#)

 register balancing [622](#)

 smart black boxes [785](#)

multi-channels

 option [504](#)

multi_cycle_path constraint [624](#)

multiplexers [381](#)

 vector [561](#)

multipliers [378](#)

 resource usage [850](#)

multirate blocks

 pattern folding [665](#)

multirate design

 defined [719](#)

multi-rate designs

 clock resets [723](#)

Mux block [381](#)

 using in Synphony [562](#)

N

NAND operation

 Binary Logic block [54](#)

NCO. *See* DDS block.

Negate block [386](#)

noise cancellation

 demo [594](#)

NOR operation

 Binary Logic block [54](#)

Note [930](#)

O

offset

 defined for downsample [719](#)

 upsample [719](#)

opens [554](#)

optimizations

 folding [662](#)

- multichannelization [674](#)
- using retiming [655](#)
- OR operation
 - Binary Logic block [54](#)
- order of operations [697](#)
- Out block [388](#)
- output format options [583](#)
- output options [492](#)
- output ports [403](#)
- overflow. *See* saturation options.

P

- Parallel FIR block [389](#)
- parameterized blocks [816](#)
- pattern
 - for pattern folding [665](#)
- pattern folding
 - annotating patterns in constraint file [626](#)
 - annotating patterns in Simulink GUI [668](#)
 - description [665](#)
 - example [666](#)
 - excluding patterns [672](#)
 - option [502](#)
- pattern_annotation constraint [626](#)
- performance
 - improving with gate-level retiming [660](#)
- Permutation block [394](#)
- phase
 - defined [720](#)
- plots [835](#)
- polyphase filtering [724](#)
- polyphase filters [767](#), [769](#)
 - FIR2 block [257](#)
- Port In block [399](#)
- ports
 - data capture [615](#)
 - displaying fixed point data type [696](#)
 - HLS Subsystem block [320](#)
 - M Control blocks [863](#)
 - register latency [616](#)
- Ports & Subsystems library [35](#)
- primitive blocks [800](#)
- punctures
 - depuncturing [173](#)

Q

- QAM 16
 - demos [594](#)
- quantization
 - Convert block [98](#)

R

- Ramp block [419](#)
- RAMs
 - resource usage [850](#)
- Recast block [424](#)
- Reed Solomon decoder demo [594](#)
- Reed Solomon encoder demo [594](#)
- Reed-Solomon coding [436](#)
- Reed-Solomon Decoder block [428](#)
 - pins [429](#)
 - timing [430](#)
- Reed-Solomon Encoder block [435](#)
 - timing [437](#)
- register balancing [622](#)
- Register block [441](#)
- REGISTER_DESIGN API [915](#)
- registers
 - latency parameters [616](#)
 - resource usage [850](#)
- rehash command [806](#)
- reloadable filter
 - FIR2 block [247](#)
- reloadable FIR [448](#)
- resamplers
 - FIR polyphase [243](#)
- resampling
 - defined [720](#)
- reserved characters for RTL generation [22](#)
- resets
 - global, Symphony [683](#)
 - local, Symphony [683](#)
- Reshape block [443](#)
- resource sharing
 - pattern folding [665](#)
- resource usage [850](#)
- retime_across_blackbox constraint [628](#)
- retiming
 - black boxes [785](#)
 - constraint example [629](#)
 - Delay blocks [169](#), [628](#)
 - gate-level (tutorial) [660](#)

- latencies for register balancing [622](#)
- optimizing with [655](#)
- option [503](#)
- smart black boxes [785](#)
- retiming constraints [653](#)
- retiming_scale_factor constraint [628](#)
- RFIR block [448](#)
- ROM block [453](#)
- ROM data
 - syn_read_hex function [610](#)
- ROMs [453](#)
 - specifying values [454](#)
- rounding options [585](#)
- RTL
 - black boxes [780](#)
 - comparing to C output [905](#)
 - consolidating blocks [730](#)
 - global resets, Symphony [685](#)
 - procedure for generating (FPGA) [24](#)
 - requirements for generating [22](#)
 - retaining block names [649](#)
 - retaining signal names [650](#)
 - smart black boxes [839](#)
 - subsystem consolidation [729](#)
- RTL encapsulation
 - port configuration [461](#)
- RTL Encapsulation block [456](#)
 - retiming constraint [627](#)
- RTL generation
 - word length for propagated data types [23](#)

S

- sample period
 - defined [720](#)
- sample rate [717](#)
 - decreasing input rate [191](#)
 - defined [720](#)
 - determining [682](#)
 - increasing input rate [557](#)
- saturation options [585](#)
- scripts
 - Aldec simulator [855](#)
 - Cadence NC simulator [855](#)
 - Modelsim simulator [855](#)
- Sequence block [470](#)
- serial to parallel, demo [594](#)
- set_param command, pattern folding [672](#)

- Shift Register block [476](#)
- shift registers
 - resource usage [850](#)
- Shifter block [484](#)
- shls.log file
 - resource usage [850](#)
- shls_bitrev function [590](#)
- shlsclib custom library [805](#)
- shls_convert function [592](#)
- shlsdemo function [594](#)
- shlsdoc function [596](#)
- shslslib function [597](#)
- shls_retiming_lock constraint [169](#), [628](#)
 - forward-annotation [636](#)
- shlsroot function [599](#)
- SHLSTool
 - shlstool function [600](#)
- SHLSTool block [486](#)
 - implementations [488](#)
 - interface [487](#)
- shlstool function [600](#)
- SHLSTool toolbox
 - using [677](#)
- shlsver function [602](#)
- Sign block [507](#)
- signal clocks [682](#)
- Signal Operations library [36](#)
- signal tracing [650](#)
- simlog.txt file [905](#)
- simulation license API [923](#)
- simulations
 - comparing with plots [835](#)
- Simulink
 - simulating Symphony FPGA design [24](#)
 - version [19](#)
 - wrapper scripts for C output [931](#)
- Simulink configuration
 - checking for Symphony [606](#)
 - setting for Symphony [614](#)
- Simulink simulation speedup [927](#)
- Simulink wrapper file
 - DAT files [930](#)
 - description [930](#)
 - generating [928](#)
 - verification scripts [931](#)
- SincCos2 block [516](#)
- SinCos block [513](#)

- sine
 - CORDIC SinCos block [124](#)
 - SinCos block [513](#)
- Single Clock Downsample [526](#)
- Single Clock Upsample block [529](#)
- single-rate designs
 - clock resets [722](#)
- slot
 - defined [720](#)
- smart black box
 - demo [594](#)
- Smart Black Box block [532](#)
- smart black boxes
 - folding [785](#)
 - retiming [785](#)
 - RTL [839](#)
- smc_synplify_synthesis_pragma_module [797](#)
- Sobel filtering
 - demo [594](#)
- software description [18](#)
- Sources library [37](#)
- Sqrt block [539](#)
- square root
 - CORDIC Sqrt block [126](#)
 - Sqrt block [539](#)
- state machines
 - Mealy [364](#)
 - Moore [369](#)
- Subsystem block [543](#)
 - multicycle constraint [624](#)
- subsystems
 - consolidating in RTL [729](#)
 - custom blocks [806](#)
 - in port [345](#)
 - out port [388](#)
 - preserving hierarchy [728](#)
 - Synphony [543](#)
 - Synphony Subsystem block [319](#)
 - variable values (tutorial) [794](#)
 - viewing hierarchy [794](#)
- subtractor
 - Synphony block [48](#)
- symbol ordering
 - block de-interleaving [62](#)
 - block interleaving [64](#)
- symmetric coefficients [254](#)
- synchronous resets
 - RTL code, Synphony [685](#)
 - testbenches, Synphony [686](#)
- synchronous resets, Synphony [684](#)
- syn_coef_dt variable [588](#)
- syn_coef_fl variable [588](#)
- syn_coef_wl variable [588](#)
- SynCoSimTool block [550](#)
- SynFixPtTool block [554](#)
- SynFixPtTool
 - setting options [832](#)
 - validating algorithms [834](#)
- syn_get_coefs
 - FIR2 block [251](#)
 - using for FIRs [769](#)
 - using for IIRs [772](#)
- syn_get_coefs function [604](#)
- syn_get_datatype function [605](#)
- syn_get_dspstartup function [606](#)
- syn_get_wordlength function [608](#)
- syn_guard_bit variable [588](#)
- syn_inh_dt variable [588](#)
- syn_inh_fl variable [588](#)
- syn_inh_width variable [568](#), [588](#)
- syn_inh_wl variable [588](#)
- syn_inp_dt variable [588](#)
- syn_inp_fl variable [588](#)
- syn_inp_wl variable [588](#)
- syn_mat_columns variable [588](#)
 - Reshape [445](#)
- syn_mat_rows variable
 - Reshape [445](#)
- Synopsys
 - data type implementation [696](#)
- Synphony
 - accessing blockset [642](#)
 - accessing information from the
 - command line [598](#)
 - checking Simulink configuration [606](#)
 - FPGA design flow [20](#), [23](#)
 - FPGA output [856](#)
 - functions [589](#)
 - M language support [893](#)
 - setting Simulink configuration [614](#)
 - starting from MATLAB [641](#)
 - starting from MATLAB (FPGA) [23](#)
 - starting from Simulink [23](#), [641](#)
- Synphony demos [594](#)
- Synphony FDATool block
 - FIR coefficients [768](#)
 - IIR coefficients [771](#)

Synphony FIR block
 decimators [767](#)
Synphony SynFixPtTool block
 using [832](#)
Synplify Pro version [19](#)
syn_read_hex function [610](#)
 using [776](#)
syn_set_atm function [612](#)
syn_set_dspstartup function [614](#)
syn_set_portcapture function [615](#)
syn_set_portregister function [616](#)
synStrEval function [60](#)
synthesis pragmas [797](#)
syn_unlink function [617](#)
syn_write_wave
 writes VCD file [618](#)
System C, using with C output [932](#)

T

target technology (FPGAs) [491](#)
test benches
 generating [853](#)
 option for generating [493](#)
 simulating [855](#)
testbenches
 global reset, Synphony [686](#)
third-party IP, embedding [532](#)
time 0
 defined [721](#)
time0
 determining [721](#)
 single-rate designs [723](#)
timing diagrams. *See* waveforms
timing engine configuration [612](#)
timing modes
 FPGAs [638](#)
timing waveforms. *See* waveforms
toolboxes
 FDATool block [203](#)
 SHLSTool [486](#)
 SynCoSimTool [550](#)
 SynFixPtTool [554](#)
transactions
 APB [744](#)
 AVLON-MM [747](#)
 AX14-Lite [739](#)
 generic protocol [749](#)

Transforms library [38](#)

U

underflow. *See* rounding options.
upsample
 offset [719](#)
Upsample block
 multicycle constraint [635](#)
Upsampler block [557](#)

V

VCD file
 using syn_write_wave [618](#)
Vector Concat block [561](#)
Vector Expand block [567](#)
Vector Extract block [570](#)
Vector Split block [572](#)
vectors
 working with [773](#)
verification
 generating testbenches from Synphony
 [853](#)
 verification speedup [927](#)
Verilog
 asynchronous resets, Synphony [685](#)
 generating [646](#)
 keywords [647](#)
 naming rules [647](#)
 synchronous resets, Synphony [685](#)
 test benches [853](#)
 Verilog 2001 [647](#)
version
 shlsver function [602](#)
VHDL
 asynchronous resets, Synphony [685](#)
 generating [646](#)
 keywords [647](#)
 naming rules [647](#)
 synchronous resets, Synphony [685](#)
 VHDL 93 [647](#)
VHDL test benches [853](#)
Viterbi Decoder block [574](#)
Viterbi Decoder, Synphony
 demo [594](#)
 demo with puncturing [594](#)
void * CModelCreateInstance API [916](#)

W

waveforms

- Downsample [195](#)

- FIFO [223](#)

- Flow Control Buffer [282](#)

word length

- calculating with syn_get_wordlength
[608](#)

- converting from Simulink to Synphony
[605](#)

- propagated data types for RTL
generation [23](#)

- syn_get_wordlength function [608](#)

word length options

- output [584](#)

word size

- converting input word size [98](#)

X

XNOR operation

- Binary Logic block [54](#)

XOR operation

- Binary Logic block [54](#)

