



*Synplicity*<sup>®</sup>

**Simply Better Results**

# Synplify<sup>®</sup> DSP

**User Guide**

**April 2008**

Synplicity, Inc.  
600 West California Avenue  
Sunnyvale, CA 94086  
(U.S.) +1 408 215-6000 direct  
(U.S.) +1 408 222-0263 fax  
[www.synplicity.com](http://www.synplicity.com)

# Copyright and License Agreement

## Disclaimer of Warranty

Synplicity, Inc. makes no representations or warranties, either expressed or implied, by or with respect to anything in this manual, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any indirect, special or consequential damages.

## Copyright Notice

Copyright © 2008 Synplicity, Inc. All Rights Reserved.

Synplicity software products contain certain confidential information of Synplicity, Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the prior written permission of Synplicity, Inc. While every precaution has been taken in the preparation of this book, Synplicity, Inc. assumes no responsibility for errors or omissions. This publication and the features described herein are subject to change without notice.

## Trademarks

Synplicity, the Synplicity “S” logo, Amplify, Amplify ASIC, Amplify FPGA, Behavior Extracting Synthesis Technology, Certify, Embedded Synthesis, Fortify, HDL Analyst, PowerTime, RealPower, SCOPE, Simply Better Results, Simply Better Synthesis, Syndicated, Synplify, Synplify ASIC, Synplify Lite, Synplify Pro, and Synthesis Constraint Optimization Environment are registered trademarks of Synplicity Inc. BEST, DST, Direct Synthesis Technology, Identify, IICE, MultiPoint, Partition-Driven Synthesis, Physical Analyst, Physical Optimizer, PowerPlanner, PowerRoute, Synplify IP, TOPS, and Total Optimization Physical Synthesis are trademarks of Synplicity, Inc.

Verilog is a registered trademark of Cadence Design Systems, Inc. IBM and PC are registered trademarks of International Business Machines Corporation. Microsoft is a registered trademark of Microsoft Corporation. Sun, SPARC, Solaris, and SunOS are trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of X/Open Corporation.

All other product names mentioned herein are the trademarks or registered trademarks of their respective owners.

## **Restricted Rights Legend**

Government Users: Use, reproduction, release, modification, or disclosure of this commercial computer software, or of any related documentation of any kind, is restricted in accordance with FAR 12.212 and DFARS 227.7202, and further restricted by the Synplicity Software License Agreement. Synplicity, Inc., 600 West California Avenue, Sunnyvale, CA 94086, U. S. A.

Printed in the U.S.A  
April 2008

## Synplicity Software License Agreement

---

Important! READ CAREFULLY BEFORE PROCEEDING

---

BY INDICATING YOUR ACCEPTANCE OF THE TERMS OF THIS AGREEMENT, YOU ("LICENSEE") ARE REPRESENTING THAT YOU HAVE THE RIGHT AND AUTHORITY TO LEGALLY BIND YOURSELF OR YOUR COMPANY, AS APPLICABLE, AND CONSENTING TO BE LEGALLY BOUND BY ALL OF THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL THESE TERMS DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE TO THE LOCATION OF PURCHASE FOR A REFUND. This is a legal agreement governing use of the software program provided by Synplicity, Inc. ("Synplicity") to you (the "SOFTWARE"). The term "SOFTWARE" also includes related documentation (whether in print or electronic form), any authorization keys, authorization codes, and license files, and any updates or upgrades of the SOFTWARE provided by Synplicity, but does not include certain "open source" software licensed by third party licensors and made available to you by Synplicity under the terms of such third party licensor's license (such as software licensed under the General Public License (GPL)) ("Third Party Software"). If Licensee is a participant in the University Program or has been granted an Evaluation License or Subscription License, then some of the following terms and conditions may not apply (refer to the sections entitled, respectively, **Evaluation License** and **Subscription License**, below).

**License.** Synplicity grants to Licensee a non-exclusive right to install the SOFTWARE and to use or authorize use of the SOFTWARE by up to the number of nodes for which Licensee has a license and for which Licensee has the security key(s) or authorization code(s) provided by Synplicity or its agents for the purpose of creating and modifying Designs (as defined below). If Licensee has obtained the SOFTWARE under a node-locked license, then a "node" refers to a specific machine, and the SOFTWARE may be installed only on the number of "nodes" or machines authorized, must be used only on the machine(s) on which it is installed, and may be accessed only by users who are physically present at that node or machine. A node-locked license may only be used by one user at a time running one instance of the software at a time. If Licensee has obtained the SOFTWARE under a "floating" license, then a "node" refers to a concurrent user or session, and the SOFTWARE may be used concurrently by up to the number of users or sessions indicated. All SOFTWARE must be used within the country for which the systems were licensed and at Licensee's Site (contained within a one kilometer radius); however, if Licensee has a floating license then remote use is permitted by employees who work at the site but are temporarily telecommuting to that same site from less than 50 miles away (for example, an employee who works at a home office on occasion), but the maximum number of concurrent sessions or nodes still applies. In addition, Synplicity grants to Licensee a non-exclusive license to copy and distribute internally the documentation portion of the SOFTWARE in support of its license to use the program portion of the SOFTWARE. For purposes of this Agreement the "Licensee's Site" means the location of the server on which the SOFTWARE resides, or when a server is not required, the location of the client computer for which the license was issued.

**Evaluation License.** If Licensee has obtained the SOFTWARE pursuant to an evaluation license, then, in addition to all other terms and conditions herein, the following restrictions apply: (a) the license to the SOFTWARE terminates after 20 days (unless otherwise agreed to in writing by Synplicity); and (b) Licensee may use the SOFTWARE only for the sole purpose of internal testing and evaluation to determine whether Licensee wishes to license the SOFTWARE on a commercial basis. Licensee shall not use the SOFTWARE to design any integrated circuits for production or pre-production purposes or any other commercial use including, but not limited to, for the benefit of Licensee's customers. If Licensee breaches any of the foregoing restrictions, then Licensee shall pay to Synplicity a license fee equal to Synplicity's perpetual list price plus maintenance for the commercial version of the SOFTWARE.

**Subscription (Time-Based) License.** If Licensee has obtained a Subscription License to the SOFTWARE, in addition to all other terms and conditions herein, the following restrictions apply: (a) Licensee is authorized to use the SOFTWARE only for a limited time (which time is indicated on the quotation or in the purchase confirmation documents); (b) Licensee's right to use the SOFTWARE terminates on the date the subscription term expires as set forth in the quotation or the purchase confirmation documents, unless Licensee has renewed the license by paying the applicable fees.

**Project Based License.** If Licensee has obtained a Project-Based License to the SOFTWARE, in addition to all other terms and conditions herein, the terms of Exhibit A will apply.

**Copy Restrictions.** This SOFTWARE is protected by United States copyright laws and international treaty provisions and Licensee may copy the SOFTWARE only as follows: (i) to directly support authorized use under the license, and (ii) in order to make a copy of the SOFTWARE for backup purposes. Copies must include all copyright and trademark notices.

**Use Restrictions.** This SOFTWARE is licensed to Licensee for internal use only. Licensee shall not (and shall not allow any third party to): (i) decompile, disassemble, reverse engineer or attempt to reconstruct, identify or discover any source code, underlying ideas, underlying user interface techniques or algorithms of the SOFTWARE by any means whatever, or disclose any of the foregoing; (ii) provide, lease, lend, or use the SOFTWARE for timesharing or service bureau purposes, on an application service provider basis, or otherwise circumvent the internal use restrictions; (iii) modify, incorporate into or with other software, or create a derivative work of any part of the SOFTWARE; (iv) disclose the results of any benchmarking of the SOFTWARE, or use such results for its own competing software development activities, without the prior written permission of Synplicity; or (v) attempt to circumvent any user limits, maximum gate count limits or other license, timing or use restrictions that are built into the SOFTWARE.

**Transfer Restrictions/No Assignment.** The SOFTWARE may only be used under this license at the designated locations and designated equipment as set forth in the license grant above, and may not be moved to other locations or equipment or otherwise transferred without the prior written consent of Synplicity. Any permitted transfer of the SOFTWARE will require that Licensee executes a "Software Authorization Transfer Agreement" provided by Synplicity.

Further, Licensee shall not sublicense, or assign this Agreement or any of the rights or licenses granted under this Agreement, without the prior written consent of Synplicity.

**Security.** Licensee agrees to take all appropriate measures to safeguard the SOFTWARE and prevent unauthorized access or use thereof. Suggested ways to accomplish this include: (i) implementation of firewalls and other security applications, (ii) use of FLEXIm options file that restricts access to the SOFTWARE to identified users; (iii) maintaining and storing license information in paper format only; (iv) changing TCP port numbers every three (3) months; and (v) communicating to all authorized users that use of the SOFTWARE is subject to the restrictions set forth in this Agreement.

**Ownership of the SOFTWARE.** Synplicity retains all right, title, and interest in the SOFTWARE (including all copies), and all worldwide intellectual property rights therein. Synplicity reserves all rights not expressly granted to Licensee. This license is not a sale of the original SOFTWARE or of any copy.

**Ownership of Design Techniques.** "Design" means the representation of an electronic circuit or device(s), derived or created by Licensee through the use of the SOFTWARE in its various formats, including, but not limited to, equations, truth tables, schematic diagrams, textual descriptions, hardware description languages, and netlists. "Design Techniques" means the data, circuit and logic elements, libraries, algorithms, search strategies, rule bases, techniques and technical information incorporated in the SOFTWARE and employed in the process of creating Designs. Synplicity retains all right, title and interest in and to Design Techniques incorporated in the SOFTWARE, including all intellectual property rights embodied therein, provided that to the extent any Design Techniques are included as part of or embedded within Licensee's Designs, Synplicity grants Licensee a personal, nonexclusive, nontransferable license to reproduce the Design Techniques and distribute such Design Techniques solely as incorporated into Licensee's Designs and not on a standalone basis. Additionally, Licensee acknowledges that Synplicity has an unrestricted, royalty-free right to incorporate any Design Techniques disclosed by Licensee into its software, documentation and other products, and to sublicense third parties to use those incorporated design techniques.

**Protection of Confidential Information.** "Confidential Information" means (i) the SOFTWARE, in object and source code form, and any related technology, idea, algorithm or information contained therein, including without limitation Design Techniques, and any trade secrets related to any of the foregoing; (ii) either party's product plans, Designs, costs, prices and names; non-published financial information; marketing plans; business opportunities; personnel; research; development or know-how; (iii) any information designated by the disclosing party as confidential in writing or, if disclosed orally, designated as confidential at the time of disclosure and reduced to writing and designated as confidential in writing within thirty (30) days; and (iv) the terms and conditions of this Agreement; provided, however that "Confidential Information" will not include information that: (a) is or becomes generally known or available by publication, commercial use or otherwise through no fault of the receiving party; (b) is known and has been reduced to tangible form by the receiving party at the time of disclosure and is not subject to restriction; (c) is independently developed by the receiving party without use of the disclosing party's Confidential Information; (d) is lawfully obtained from a third party who has the right to make such disclosure; and (e) is released for publication by the disclosing party in writing.

**Open Source Software.** The SOFTWARE may be delivered with software that is subject to open source licensing terms ("Open Source Software") which are available at [http://www.synplicity.com/products/license\\_agreement.html](http://www.synplicity.com/products/license_agreement.html). If the Open Source Software license also requires source code to be made available, Licensee may reference <http://www.synplicity.com/products/opensource.html> for information on how to obtain such source code. Licensee agrees that all Open Source Software shall be and shall remain subject to the terms and conditions under which it is provided. The Open Source Software is provided "AS IS," WITHOUT ANY WARRANTY OF ANY KIND, AND SYNPLICITY FURTHER DISCLAIMS ALL OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO OPEN SOURCE SOFT-

WARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER SYNPLICITY NOR THE LICENSORS OF OPEN SOURCE SOFTWARE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AN ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE ECLIPSE SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Copyrights to the Open Source Software are held by the copyright holders indicated in the copyright notices in the corresponding source files.

**Third Party Software.** Licensee understands and agrees that, although provided to Licensee by Synplicity, Licensee's use of each component library or module comprising the Third Party Software shall be governed by the relevant terms and conditions of the third party's license agreements.

**Termination.** Synplicity may terminate this Agreement immediately if Licensee breaches any provision, including without limitation, failure by Licensee to implement adequate security measures as set forth above. Upon notice of termination by Synplicity, all rights granted to Licensee under this Agreement will immediately terminate, and Licensee shall cease using the SOFTWARE and return or destroy all copies (and partial copies) of the SOFTWARE and documentation.

**Limited Warranty and Disclaimer.** Synplicity warrants that the program portion of the SOFTWARE will perform substantially in accordance with the accompanying documentation for a period of 90 days from the date of receipt. Synplicity's entire liability and Licensee's exclusive remedy for a breach of the preceding limited warranty shall be, at Synplicity's option, either (a) return of the license fee, or (b) providing a fix, patch, work-around, or replacement of the SOFTWARE. In either case, Licensee must return the SOFTWARE to Synplicity with a copy of the purchase receipt or similar document. Replacements are warranted for the remainder of the original warranty period or 30 days, whichever is longer. Some states/jurisdictions do not allow limitations, so the above limitation may not apply. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES OR CONDITIONS, EITHER EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, ARE MADE BY SYNPLICITY OR ITS LICENSORS WITH RESPECT TO THE SOFTWARE AND THE ACCOMPANYING DOCUMENTATION, AND SYNPLICITY EXPRESSLY DISCLAIMS ALL WARRANTIES AND CONDITIONS NOT EXPRESSLY STATED HEREIN, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE. SYNPLICITY AND ITS LICENSORS DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET LICENSEE'S REQUIREMENTS, BE UNINTERRUPTED OR ERROR FREE, OR THAT ALL DEFECTS IN THE PROGRAM WILL BE CORRECTED. Licensee assumes the entire risk as to the results and performance of the SOFTWARE. Some states/jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply.

**Limitation of Liability.** IN NO EVENT SHALL SYNPLICITY OR ITS LICENSORS OR THEIR AGENTS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL OR INCIDENTAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTIONS, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF SYNPLICITY AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

FURTHER, IN NO EVENT SHALL SYNPLICITY'S LICENSORS BE LIABLE FOR ANY DIRECT DAMAGES ARISING OUT OF LICENSEE'S USE OF THE SOFTWARE. IN NO EVENT WILL SYNPLICITY OR ITS LICENSORS BE LIABLE TO LICENSEE FOR DAMAGES IN AN AMOUNT GREATER THAN THE FEES PAID FOR THE USE OF THE SOFTWARE. Some states/jurisdictions do not allow the limitation or exclusion of incidental or consequential damages, so the above limitations or exclusions may not apply.

**Intellectual Property Right Infringement.** Synplicity will defend or, at its option, settle any claim or action brought against Licensee to the extent it is based on a third party claim that the SOFTWARE as used within the scope of this Agreement infringes or violates any US patent, copyright, trade secret or trademark of any third party, and Synplicity will indemnify and hold Licensee harmless from and against any damages, costs and fees reasonably incurred that are attributable to such claim or action; provided that Licensee provides Synplicity with (i) prompt written notification of the claim or action; (ii) sole control and authority over the defense or settlement thereof (including all negotiations); and (iii) at Synplicity's expense, all available information, assistance and authority to settle and/or defend any such claim or action. Synplicity's obligations under this subsection do not apply to the extent that (i) such claim or action would have been avoided but for modifications of the SOFTWARE, or portions thereof, other than modifications made by Synplicity after delivery to Licensee; (ii) such claim or action would have been avoided but for the combination or use of the SOFTWARE, or portions thereof, with other products, processes or materials not supplied or specified in writing by Synplicity; (iii) Licensee continues allegedly infringing activity after being notified thereof or after being informed of modifications that would have avoided the alleged infringement; or (iv) Licensee's use of the SOFTWARE is not strictly in accordance with the terms of this Agreement. Licensee will be liable for all damages, costs, expenses, settlements and attorneys' fees related to any claim of infringement arising as a result of (i)-(iv) above.

If the SOFTWARE becomes or, in the reasonable opinion of Synplicity is likely to become, the subject of an infringement claim or action, Synplicity may, at Synplicity's option and at no charge to Licensee, (a) obtain a license so Licensee may continue use of the SOFTWARE; (b) modify the SOFTWARE to avoid the infringement; (c) replace the SOFTWARE with a compatible, functionally equivalent, and non-infringing product, or (d) if Synplicity determines that options (a), (b), and (c) are not commercially reasonable, then Synplicity shall have the right to terminate the licenses granted hereunder and refund to Licensee the amount paid for the SOFTWARE, as depreciated on a straight-line 5-year basis, or such other shorter period applicable to Subscription Licenses.

THE FOREGOING PROVISIONS OF THIS SECTION STATE THE ENTIRE AND SOLE LIABILITY AND OBLIGATIONS OF SYNPLICITY, AND THE EXCLUSIVE REMEDY OF LICENSEE, WITH RESPECT TO ANY ACTUAL OR ALLEGED INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS BY THE SOFTWARE (INCLUDING DESIGN TECHNIQUES) AND DOCUMENTATION.

**Export.** Licensee warrants that it is not prohibited from receiving the SOFTWARE under U.S. export laws; that it is not a national of a country subject to U.S. trade sanctions; that it will not use the SOFTWARE in a location that is the subject of U.S. trade sanctions that would cover the SOFTWARE; and that to its knowledge it is not on the U.S. Department of Commerce's table of deny orders or otherwise prohibited from obtaining goods of this sort from the United States.



**Miscellaneous.** This Agreement is the entire agreement between Licensee and Synplicity with respect to the license to the SOFTWARE, and supersedes any previous oral or written communications or documents (including, if you are obtaining an update, any agreement that may have been included with the initial version of the Software). This Agreement is governed by the laws of the State of California, USA excluding its conflicts of laws principals. This Agreement will not be governed by the U. N. Convention on Contracts for the International Sale of Goods and will not be governed by any statute based on or derived from the Uniform Computer Information Transactions Act (UCITA). If any provision, or portion thereof, of this Agreement is found to be invalid or unenforceable, it will be enforced to the extent permissible and the remainder of this Agreement will remain in full force and effect. Failure to prosecute a party's rights with respect to a default hereunder will not constitute a waiver of the right to enforce rights with respect to the same or any other breach.

**Government Users.** If the SOFTWARE is licensed to the United States government or any agency thereof, then the SOFTWARE and any accompanying documentation will be deemed to be "commercial computer software" and "commercial computer software documentation", respectively, pursuant to DFAR Section 227.7202 and FAR Section 12.212, as applicable. Any use, reproduction, release, performance, display or disclosure of the SOFTWARE and accompanying documentation by the U.S. Government will be governed solely by the terms of this Agreement and are prohibited except to the extent expressly permitted by the terms of this Agreement.

March 2008



# Contents

---

## Chapter 1: Getting Started

Introduction to The Synplify DSP Tool .....	1-2
About the Software .....	1-2
Synplify DSP ASIC Edition .....	1-3
Assumptions .....	1-3
Advantages of Synplify DSP .....	1-3
Synplify DSP Design Flows .....	1-5
Synplify DSP FPGA Design Flow .....	1-5
Design Requirements for RTL Generation .....	1-7
FPGA Design Flow Procedure .....	1-8
Synplify DSP ASIC Design Flow .....	1-11
ASIC Design Flow Procedure .....	1-13
Finding Information .....	1-16
Getting Help .....	1-16

## Chapter 2: Synplify DSP FPGA Tutorial

Tutorial Design Flow .....	2-2
Create Algorithm Models .....	2-3
Start the Demo Tutorial .....	2-3
Add Port In and Port Out Blocks .....	2-5
Add the FIR Block .....	2-7
Set up for Verification .....	2-11
Add Stimuli Components .....	2-11
Add Analysis Components .....	2-12
Analyze and Simulate .....	2-15
Explore Quantization Effects .....	2-20
Running Floating-Point Simulation .....	2-20
Analyzing the Impact of Quantization .....	2-22

Synthesize Optimized Architectures .....	2-25
Run DSP Synthesis .....	2-26
Verify RTL .....	2-29
Run Logic Synthesis .....	2-29
Refine Optimizations .....	2-31
Optimization Strategies .....	2-32
Using Retiming for Performance .....	2-33
Using Folding to Decrease Area .....	2-34

## Chapter 3: Synplify DSP Underlying Concepts

Clock Management .....	3-2
Signal Clocks .....	3-2
Implementation Clocks .....	3-2
CORDIC Algorithms .....	3-3
CORDIC Definitions .....	3-4
Unified CORDIC Applications .....	3-13
Data Types .....	3-19
Fixed-Point and Floating-Point Representation .....	3-19
Synplify DSP Data Type Implementation .....	3-20
Fixed-Point Data Type .....	3-20
Data Type Casting: Setting the Output Data Type .....	3-21
Resets in Synplify DSP .....	3-22
Global and Local Resets .....	3-22
Synchronous and Asynchronous Resets .....	3-23
Reset Implementation in RTL Code .....	3-24
Resets and RTL Testbenches .....	3-25
Multi-Rate Design .....	3-25
Sample Rate Terminology .....	3-26
Clock Generation and Clock Reset .....	3-29
Polyphase Filtering .....	3-32

## Chapter 4: Using Synplify DSP for DSP Design

Configuring Synplify DSP for Optimal Use .....	4-2
Configuring Settings for Simulink Simulation .....	4-2
Setting Default Display Modes .....	4-3
Basic Procedures .....	4-4
Starting a Synplify DSP Design .....	4-4
Working with Synplify DSP Blocks .....	4-5

Working with the Output for ASIC Designs . . . . .	4-7
Output Files for ASIC Designs . . . . .	4-7
Running ASIC Logic Synthesis . . . . .	4-9
Working with ASIC Output Tcl Files . . . . .	4-10
Designing Filters . . . . .	4-12
Implementing FIR Filters . . . . .	4-12
Implementing Polyphase FIR Filters . . . . .	4-14
Defining FIR Filter Coefficients with FDATool . . . . .	4-15
Implementing IIR Filters . . . . .	4-17
Defining IIR Filter Coefficients with FDATool . . . . .	4-19
Working with Vectors . . . . .	4-21
Creating Vector Signals . . . . .	4-21
Using Math Operations on Vector Signals . . . . .	4-22
Using Black Boxes and Third-Party IP . . . . .	4-24
Integrating Black Boxes in the Design . . . . .	4-24
Using Optimizations with Black Boxes . . . . .	4-27
Setting Black Box Parameters . . . . .	4-28
Configuring a Black Box - Example . . . . .	4-30
Using Smart RTL Black Boxes . . . . .	4-33
Incorporating Smart Black Boxes in the Design . . . . .	4-33
Configuring the Cosimulation Interface . . . . .	4-35
Creating Smart Black Box Configuration Files . . . . .	4-37
Using Quantization Analysis Tools . . . . .	4-38
Specifying Fixed-Point Options . . . . .	4-38
Validating Algorithms with the Fixed-Point Toolbox . . . . .	4-40
Using Plots . . . . .	4-42
Specifying ROM Data with syn_read_hex . . . . .	4-43
Managing Subsystems and Hierarchy . . . . .	4-44
Running DSP Synthesis with SynDSPTool . . . . .	4-48
Setting up Implementations . . . . .	4-48
Configuring Synplify DSP Timing Modes for FPGAs . . . . .	4-51
Optimizing with Retiming . . . . .	4-53
Using Automatic Gate-level Retiming . . . . .	4-54
Optimizing with Folding . . . . .	4-55
Optimizing with Multichannelization . . . . .	4-60
Running DSP Synthesis for FPGA Targets . . . . .	4-61
Running DSP Synthesis for ASIC Targets . . . . .	4-62

Working with Synplify DSP Output . . . . .	4-62
Verifying the RTL with a Test Bench . . . . .	4-63
Running Logic Synthesis for FPGA Targets . . . . .	4-65
Running Logic Synthesis for ASIC Targets . . . . .	4-66
Working with the Actel Encrypted Flow . . . . .	4-67

## Chapter 5: Working with Custom Blocks

Primitives and Custom Blocks . . . . .	5-2
Design Flow for Building Custom Blocks . . . . .	5-5
Set up a Custom Library . . . . .	5-6
Create a Custom Block . . . . .	5-7
Define Basic Content for Custom Blocks . . . . .	5-13
Define Content for Parameterized Blocks . . . . .	5-16
Define Content for Reconfigurable Blocks . . . . .	5-20
Designing with Custom Blocks . . . . .	5-24
Maintaining Custom Libraries . . . . .	5-25
Maintaining Independent Custom Libraries . . . . .	5-25
Converting Custom Libraries . . . . .	5-26
The MySign M-Generator . . . . .	5-26

## Chapter 6: Using M Control Blocks

Using M Control Blocks . . . . .	6-2
Using M Control Blocks in Synplify DSP Designs . . . . .	6-2
Tips for Designing with M Control Blocks . . . . .	6-4
Coding M Control Blocks . . . . .	6-5
Ports and Timing . . . . .	6-5
M Control Block Data Types . . . . .	6-7
Combinatorial Logic . . . . .	6-11
States with Persistent Variables . . . . .	6-12
Precision Bounds for Persistent Variables . . . . .	6-14
State Machines . . . . .	6-16
Counters . . . . .	6-24
User-Defined Functions . . . . .	6-26
Overridable Parameters . . . . .	6-27

M Language Support	6-27
Keywords, Variables, Functions, and Structures	6-28
Operator Support	6-28
Built-In Function Support	6-29
Synplify DSP Functions	6-31
M Language Limitations	6-31

## **Chapter 7: Cosimulation with ModelSim**

Overview of Cosimulation	7-2
About Cosimulation with ModelSim	7-2
Setup for Cosimulation	7-4
Software Requirements	7-4
Software Configuration	7-5
Cosimulation Flow	7-5
Cosimulation for RTL Regression	7-6
Cosimulation for VHDL	7-6
Cosimulation for Multirate VHDL	7-18
Cosimulation for Verilog	7-24

## **Chapter 8: Synplify DSP Blockset**

Blocks — By Library	8-2
Communications	8-3
Control Logic	8-4
CORDIC	8-4
DSP Basics	8-5
Filtering	8-5
Math Functions	8-6
Memories	8-7
Ports & Subsystems	8-7
Signal Operations	8-8
Sources	8-9
Transforms	8-10
Blocks — Alphabetical List	8-11
Synplify DSP Abs	8-15
Synplify DSP Accumulator	8-17
Synplify DSP Add	8-19
Synplify DSP Binary Logic	8-22

Synplify DSP Black Box .....	8-25
Synplify DSP Block Deinterleaver .....	8-31
Synplify DSP Block Interleaver .....	8-33
Synplify DSP CIC .....	8-35
Synplify DSP Commutator .....	8-39
Synplify DSP Comparator .....	8-41
Synplify DSP Concat .....	8-43
Synplify DSP Constant .....	8-45
Synplify DSP Convert .....	8-48
Synplify DSP Convolutional Deinterleaver .....	8-53
Synplify DSP Convolutional Encoder .....	8-56
Synplify DSP Convolutional Interleaver .....	8-59
Synplify DSP CORDIC Exp .....	8-61
Synplify DSP CORDIC Log .....	8-63
Synplify DSP CORDIC Polar .....	8-65
Synplify DSP CORDIC Rotator .....	8-67
Synplify DSP CORDIC SinCos .....	8-74
Synplify DSP CORDIC Sqrt .....	8-76
Synplify DSP Counter .....	8-77
Synplify DSP DDS .....	8-83
Synplify DSP Decommutator .....	8-89
Synplify DSP Delay .....	8-91
Synplify DSP Demux .....	8-92
Synplify DSP Depuncture .....	8-94
Synplify DSP Differentiator .....	8-96
Synplify DSP DivMod .....	8-99
Synplify DSP Downsample .....	8-106
Synplify DSP Extract .....	8-109
Synplify DSP FDATool .....	8-111



Synplify DSP FFT .....	8-112
Synplify DSP FIFO .....	8-118
Synplify DSP FIR .....	8-120
Synplify DSP FIR Engine .....	8-127
Synplify DSP FIR Rate Converter .....	8-133
Synplify DSP Gain .....	8-138
Synplify DSP IIR .....	8-141
Synplify DSP In .....	8-146
Synplify DSP Integrator .....	8-147
Synplify DSP Inverter .....	8-151
Synplify DSP Log .....	8-152
Synplify DSP M Control .....	8-154
Synplify DSP Mealy State Machine .....	8-157
Synplify DSP MinMax .....	8-160
Synplify DSP Moore State Machine .....	8-162
Synplify DSP Mult .....	8-164
Synplify DSP Mux .....	8-166
Synplify DSP Negate .....	8-168
Synplify DSP Out .....	8-169
Synplify DSP Parallel to Serial .....	8-170
Synplify DSP Permutation .....	8-172
Synplify DSP Port In .....	8-174
Synplify DSP Port Out .....	8-177
Synplify DSP Pow .....	8-179
Synplify DSP Puncture .....	8-181
Synplify DSP RAM .....	8-183
RAM Background Description .....	8-186
Synplify DSP Ramp .....	8-191
Synplify DSP Random .....	8-194

Synplify DSP Recast .....	8-196
Synplify DSP Reed-Solomon Decoder .....	8-200
Synplify DSP Reed-Solomon Encoder .....	8-207
Synplify DSP Register .....	8-211
Synplify DSP RFIR .....	8-213
Synplify DSP ROM .....	8-219
Synplify DSP Sequence .....	8-222
Synplify DSP Serial to Parallel .....	8-224
Synplify DSP Shift Register .....	8-227
Synplify DSP Shifter .....	8-232
Synplify DSP Sign .....	8-234
Synplify DSP SinCos .....	8-236
Synplify DSP Smart Black Box .....	8-238
Synplify DSP Sqrt .....	8-245
Synplify DSP Subsystem .....	8-248
Synplify DSP SynCoSimTool .....	8-249
Synplify DSP SynDSPTool .....	8-253
SynDSPTool Toolbox Interface .....	8-254
Implementation Options Dialog Box .....	8-258
Synplify DSP SynFixPtTool .....	8-261
Synplify DSP Upsample .....	8-263
Synplify DSP Vector Concat .....	8-267
Synplify DSP Vector Expand .....	8-273
Synplify DSP Vector Extract .....	8-275
Synplify DSP Vector Split .....	8-277
Synplify DSP Viterbi Decoder .....	8-279
Common Parameters .....	8-287
Output Format Options .....	8-287

Overflow Saturation Options .....	8-289
Underflow Rounding Options .....	8-289
Special Variables .....	8-292

## **Chapter 9: Synplify DSP Functions**

syn_bitrev .....	9-2
syn_get_coefs .....	9-4
syn_get_datatype .....	9-5
syn_get_dspstartup .....	9-6
syn_get_wl .....	9-7
syn_get_wordlength .....	9-9
syn_read_hex .....	9-11
syn_set_ate .....	9-13
Timing Engine Configuration Dialog Box .....	9-13
syn_set_atm .....	9-15
Timing Engine Configuration Dialog Box .....	9-15
syn_set_dspstartup .....	9-17
syn_set_portcapture .....	9-18
syn_set_portregister .....	9-19
syn_unlink .....	9-20
syndspdemo .....	9-21
syndspdoc .....	9-22
syndsplib .....	9-23
syndsproot .....	9-25
syndsptool .....	9-26
syndspver .....	9-27

## **Appendix A: Blockset Summary**

Summary of Synplify DSP Block Features .....	A-2
--	-----



## CHAPTER 1

# Getting Started

---

The following topics provide a general introduction to the Synplify® DSP software:

- [Introduction to The Synplify DSP Tool, on page 1-2](#)
- [Synplify DSP Design Flows, on page 1-5](#)
- [Finding Information, on page 1-16](#)
- [Getting Help, on page 1-16](#)

# Introduction to The Synplify DSP Tool

This section briefly discusses the following topics:

- [About the Software, on page 1-2](#)
- [Assumptions, on page 1-3](#)
- [Advantages of Synplify DSP, on page 1-3](#)

## About the Software

The Synplify® DSP 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 and ASIC devices. The software achieves significantly higher performance than alternative solutions and provides the designer with a mechanism to evaluate high-level area/performance tradeoffs. The output is synthesizable HDL code ready for use with the Synplicity® Synplify Pro® or ASIC 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

## Synplify DSP ASIC Edition

To target ASIC devices, you must have the Synplify DSP ASIC Edition tool, which requires a different license. In addition to the FPGA functionality, this product includes many additional features that support ASIC designs:

- It supports ASIC targets as well as FPGAs.
- For ASIC targets, it uses ASIC timing characterizations to drive the optimizations.
- It extracts memories for flexible memory support of third-party memory generators.
- The optimized RTL is formatted to support ASIC logic synthesis tools.
- The tool generates a constraint file in .sdc format for the ASIC downstream tools.
- The tool generates an example script for use with the logic synthesis tools.

## Assumptions

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 Synplify DSP output will be synthesized with the Synplify Pro product from Synplicity, so the designer must have this product and be familiar with its use.

## Advantages of Synplify DSP

The traditional DSP to FPGA/ASIC work flow required the combined efforts of a DSP engineer and a hardware engineer familiar with HDL or schematic-based design. The Synplify DSP software offers significant productivity and performance advantages to both these target audiences, and improves and streamlines the work flow:

## DSP Algorithm Designers

Using FPGAs or ASICs for DSP design is a complex task, and the Synplify DSP software makes it easy to maximize the optimizations possible with this design flow. For DSP algorithm designers implementing in FPGAs or ASICs, the Synplify DSP software

- Provides a familiar working environment. The Synplify DSP 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 Synplicity 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.

## Hardware Engineers

For the hardware engineer, the Synplify DSP software

- 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 Synplify DSP 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.



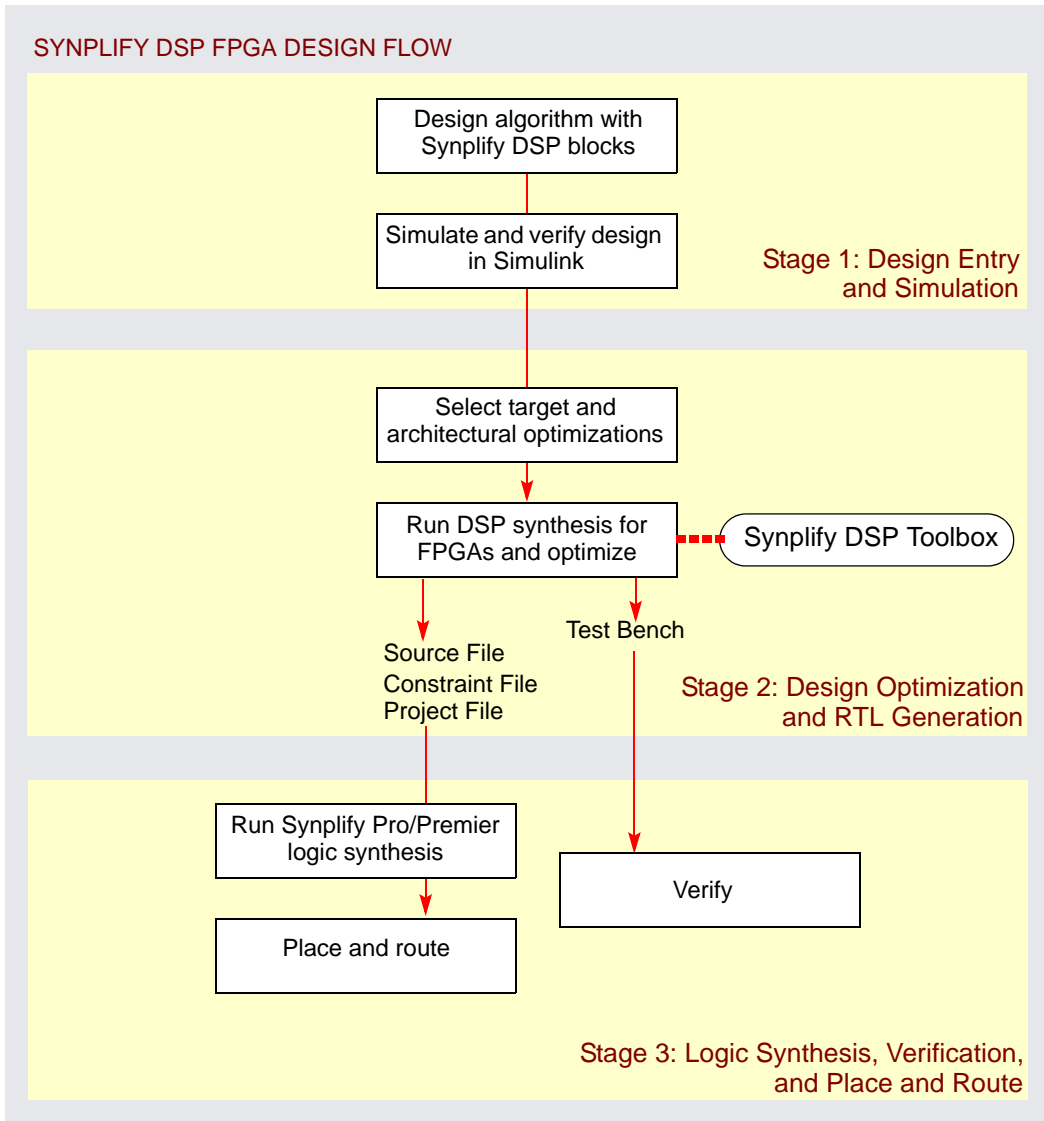
# Synplify DSP Design Flows

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

- [Synplify DSP FPGA Design Flow, on page 1-5](#)
- [Design Requirements for RTL Generation, on page 1-7](#)
- [FPGA Design Flow Procedure, on page 1-8](#)
- [Synplify DSP ASIC Design Flow, on page 1-11](#)
- [ASIC Design Flow Procedure, on page 1-13](#)

## Synplify DSP 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 1-8](#). To step through an example using the tool for an FPGA design, see the tutorial ([Synplify DSP FPGA Tutorial, on page 2-1](#)).



## Stage 1: Design Entry and Simulation

For this first stage of the flow, use the Simulink software and the Synplify DSP 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 Synplicity blockset. Simulate and verify the design at least once with Simulink to ensure correct functionality. For additional details about the flow, see [ASIC Design Flow Procedure, on page 1-13](#).

## Stage 2: Design Optimization and RTL Generation

The strengths of the Synplify DSP software are optimization and RTL generation. To do this, add the SynDSPTool block to the design.

Set system-level optimization settings and the target technology with the SynDSPTool block. Use the same block to generate RTL code. The optimizations are targeted towards the FPGA or ASIC design. For details of the flow, see [ASIC Design Flow Procedure, on page 1-13](#). 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 Synplify DSP 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 [ASIC Design Flow Procedure, on page 1-13](#).

# Design Requirements for RTL Generation

There are only two requirements to generate RTL:

- The design must be bound by the Synplify DSP Port In and Port Out blocks. You must define your design boundaries with Synplify DSP Port In and Port Out blocks. If you do not do this, the Synplify DSP 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 Synplify DSP blockset.

## FPGA Design Flow Procedure

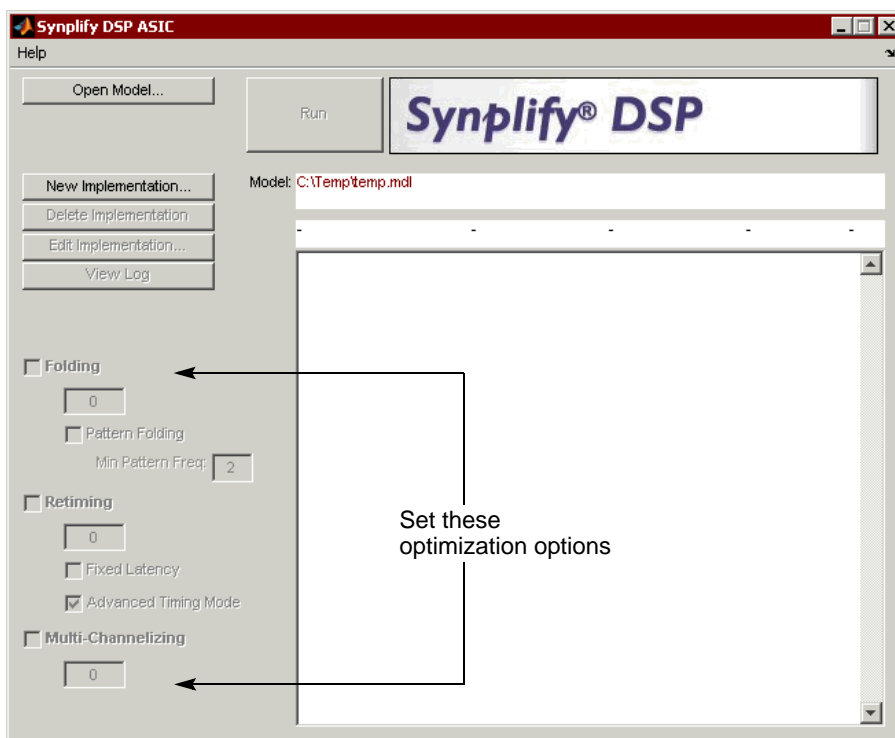
The following procedure describes the steps required to follow the design flow ([Synplify DSP FPGA Design Flow, on page 1-5](#)):

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 Synplify DSP Design, on page 4-4](#).
  - Build your circuit with Synplify DSP blocks. For details, see [Working with Synplify DSP Blocks, on page 4-5](#).
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 1-7](#).
  - In the Simulink window, click Synplicity Blockset, and add the SynDSPTool block to the design. One instance of this block controls the whole design.
  - Double-click the SynDSPTool block in the model window to open the Synplify DSP toolbox.
  - Set up the implementation and the options for it, as described in [Setting up Implementations, on page 4-48](#). In particular, choose the ASIC or FPGA target.

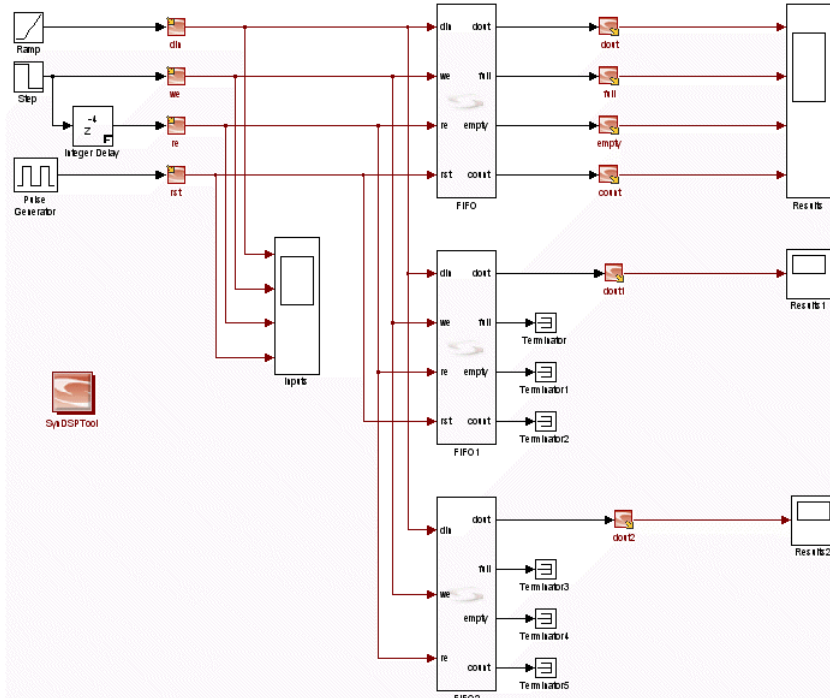
- Click OK in the Implementation Options dialog box.
- In the Synplify DSP window, set the optimization options you want: Folding, Retiming, or Multi-Channelizing. See [Running DSP Synthesis with SynDSPTool](#), on page 4-48 for details.



5. Click Run in the Synplify DSP window to generate RTL code and output files for the optimized design.

Depending on what you selected, the software runs different optimizations and generates the appropriate files for the ASIC or FPGA target you selected. The synthesis and optimization processes are different for ASIC and FPGA targets, as they are based on different timing characterizations that are tuned for the target.

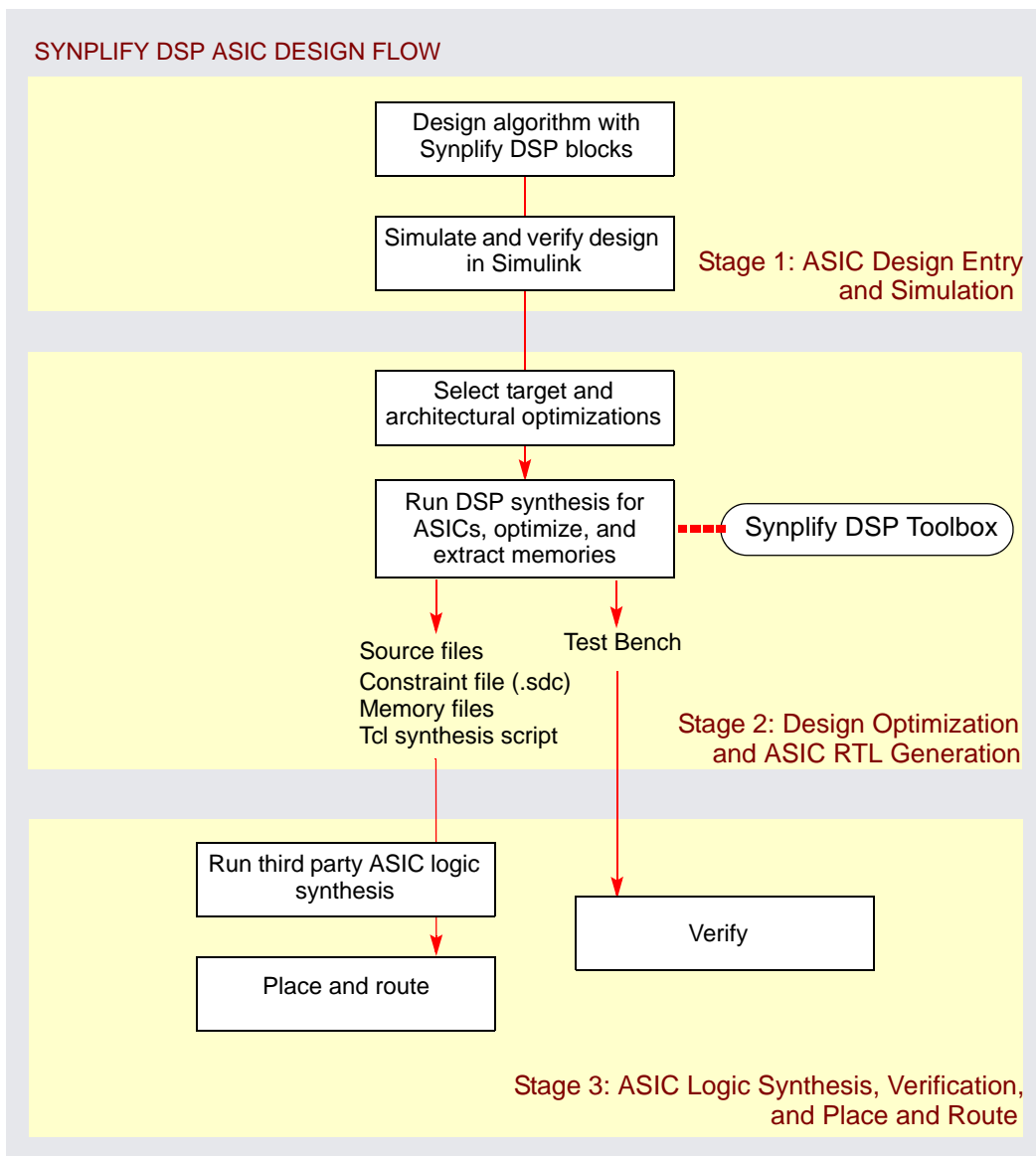
The software generates RTL code for the Synplify DSP block components in the design. It does not generate RTL code for other blocks. The output files are different for ASIC and FPGA targets.



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. For Actel designs, use the procedure described in [Working with the Actel Encrypted Flow, on page 4-67](#).
  - Place and route the design, using the tool appropriate to your design and vendor.

## Synplify DSP ASIC Design Flow

The following figure summarizes the flow for creating an ASIC design, generating RTL code, synthesizing logic, and verification. For more details, see the step-by-step procedure in [ASIC Design Flow Procedure, on page 1-13](#).



## Stage 1: ASIC Design Entry and Simulation

For this first stage of the flow, use the Simulink software and the Synplify DSP 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 Synplicity blockset. Simulate and verify the design at least once with Simulink to ensure correct functionality. For additional details about the flow, see [ASIC Design Flow Procedure, on page 1-13](#).

## Stage 2: Design Optimization and ASIC RTL Generation

Set system-level optimization settings and the target technology with the SynDSPTool block. Use the same block to generate RTL.

- Optimizations

Synplify DSP carries out various optimizations based on the target technology you selected. It also extracts memories.

- RTL Generation

The software generates RTL code and an optional test bench. It generates separate, parameterized RTL for extracted memories, which you can also use for simulation. You can replace these extracted memories with third-party memory IP for best results.

For details of the flow, see [ASIC Design Flow Procedure, on page 1-13](#).

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

For this stage, you use third-party tools for logic synthesis, verification, and place-and-route.

- 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 a third-party tool. The Synplify DSP software generates Tcl scripts that can be used with these tools.
- After synthesis, verify the post-synthesis VHDL code generated by the synthesis software against the Synplify DSP 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 [ASIC Design Flow Procedure, on page 1-13](#).

## ASIC Design Flow Procedure

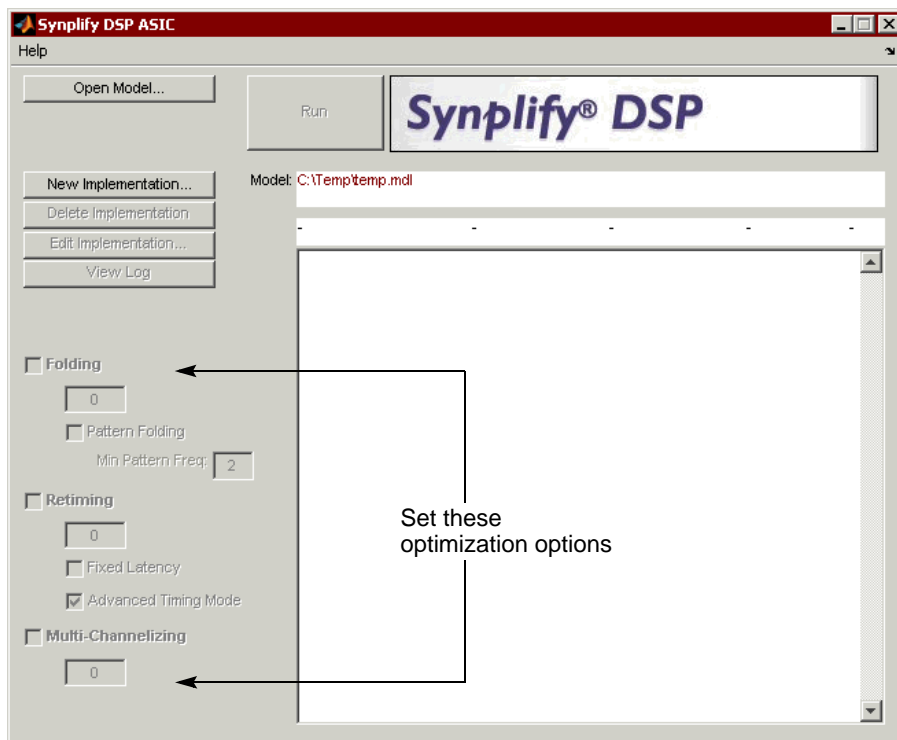
The following procedure describes the steps required to follow the ASIC design flow ([Synplify DSP ASIC Design Flow, on page 1-11](#)):

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 Synplify DSP Design, on page 4-4](#).
  - Build your circuit with Synplify DSP blocks. For details, see [Working with Synplify DSP Blocks, on page 4-5](#).
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 1-7](#).
  - In the Simulink window, click Synplicity Blockset, and add the SynDSPTool block to the design. One instance of this block controls the whole design.

- Double-click the SynDSPTool block in the model window to open the Synplify DSP toolbox.
- Set up the implementation as described in [Setting up Implementations, on page 4-48](#). Set Target to ASIC, and select a technology. Click OK in the Implementation Options dialog box.
- In the Synplify DSP window, set the optimization options you want: Folding, Retiming, or Multi-Channelizing. See [Running DSP Synthesis with SynDSPTool, on page 4-48](#) for details.



5. Click Run in the Synplify DSP window to generate RTL code and output files for the optimized design.

Synplify DSP resolves RTL parameterizations. It optimizes the logic, using techniques like constant propagation and resource sharing. It identifies and extracts memories, automatically creating optimized reset circuitry and mapping local enables and clock interfaces. The tool uses different micro-architectures and timing characterizations, based on the

technology selected. It then generates output files, which are described in [Output Files for ASIC Designs, on page 4-7](#).

6. Run logic synthesis, verify ,and place-and-route your design. For details about using these tasks, consult the documentation for these tools.
  - Use the Synplify DSP output as input to the ASIC logic synthesis tool. See [Running ASIC Logic Synthesis, on page 4-9](#) for some tips on using Synplify DSP output effectively.
  - Compare the test bench to post-synthesis gate-level simulation to verify results.
  - Place and route the design.

## Finding Information

The following table shows you where to find information:

For...	See...
Procedures and tips on using the tool	<a href="#">Using Synplify DSP for DSP Design</a>
Descriptions of individual Synplify DSP blocks	<a href="#">Blocks — By Library</a> <a href="#">Blocks — Alphabetical List</a>
Descriptions of command line functions	<a href="#">Synplify DSP Functions</a>
Help	<a href="#">Getting Help</a>

## Getting Help

The Synplify DSP software includes documentation, which you can access in the following ways:

- For a printed copy, select Synplicity Synplify DSP->Printed Documentation from the Contents tab of the Help system, and open the PDF document you need. You can print out the PDF document.
- For online help, open the Contents tab of the Help Navigator, scroll to Synplicity Synplify DSP, and select the topic you want.

- For 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 online help about blocks in the Simulink model window, right-click on the block and select **Help**. This displays information about the block.
- For online help on a dialog box, click the **Help** button.



## CHAPTER 2

# Synplify DSP FPGA Tutorial

---

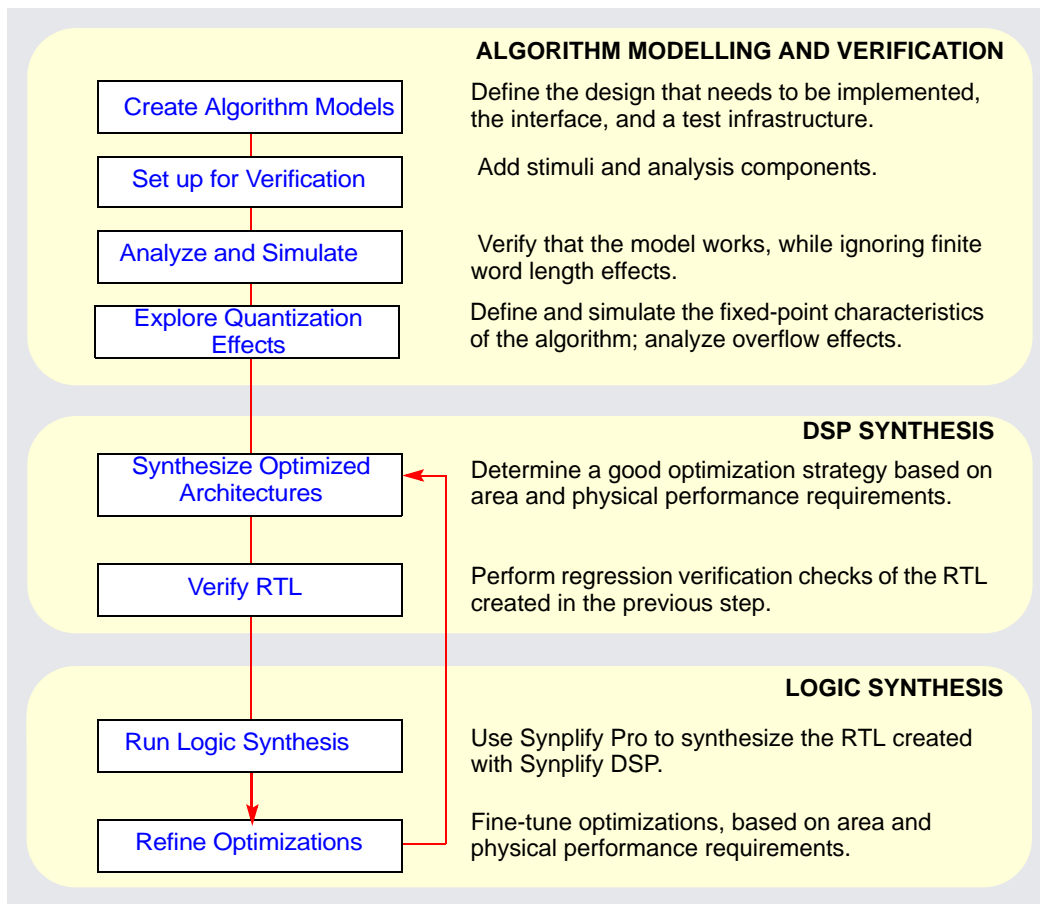
This tutorial gives you a quick introduction to working with the Synplify<sup>®</sup> DSP software for FPGA technologies. It shows you how the Synplify DSP product bridges the technology gap between MathWorks Simulink and the synthesis product line from Synplicity.

The following topics first describe the flow and then describe the stages in the Synplify DSP FPGA tutorial:

- [Tutorial Design Flow, on page 2-2](#)
- [Create Algorithm Models, on page 2-3](#)
- [Set up for Verification, on page 2-11](#)
- [Analyze and Simulate, on page 2-15](#)
- [Synthesize Optimized Architectures, on page 2-25](#)
- [Verify RTL, on page 2-29](#)
- [Run Logic Synthesis, on page 2-29](#)
- [Refine Optimizations, on page 2-31](#)

# Tutorial Design Flow

This tutorial follows the flow for an FPGA DSP design, from algorithm concept to FPGA implementation, using the Synplify DSP software and the Synplify Pro software from Synplicity for synthesis. The tutorial follows an example that has already been set up, describing the steps along the way. It is created for the supported FPGA Actel technologies. If you are targeting another FPGA vendor, you can follow the sequence of the flow to familiarize yourself with it.





# Create Algorithm Models

The design used in this tutorial is a simple, low-pass FIR filter. In this design capture stage, you use Synplify DSP blocks to capture the functionality that must be implemented in FPGA hardware. To capture a Synplify DSP design, there are two simple rules:

- The design must be bounded by Synplify DSP blocks. It must have a Synplify DSP Port In block for each input and a Synplify DSP Port Out block for each output.
- Use the Synplify DSP blockset to implement your algorithm behavior. Any functionality that is to be implemented in hardware must be instantiated from the Synplify DSP blockset.

This section describes these stages:

- [Start the Demo Tutorial, on page 2-3](#)
- [Add Port In and Port Out Blocks, on page 2-5](#)
- [Add the FIR Block, on page 2-7](#)

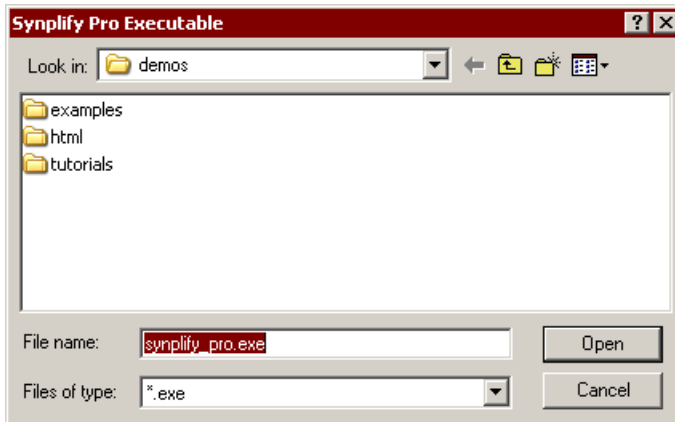
## Start the Demo Tutorial

This tutorial uses the FIR example from the demo directory. The example has already been set up, so the tutorial describes the steps that are automatically implemented in the example.

To follow along and open this example, do the following:

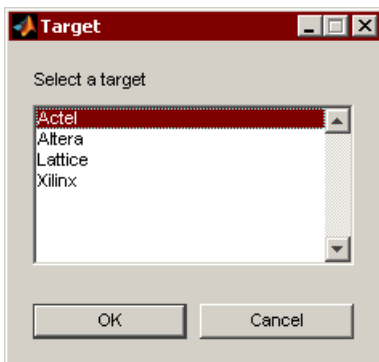
1. Start the demo tutorial:
  - From the MATLAB window, select Help->Demo.
  - Go to Synplify DSP->Tutorials
  - Double-click FIR Tutorial.

2. Specify the path to the Synplify Pro executable in the dialog box that opens.



3. Select the FPGA target architecture when you are prompted.

This tutorial follows an Actel target, and the dialog box settings and examples reflect this. If you choose one of the other three vendors, some settings might be different. You can still run through the sequence in the tutorial and get comfortable with the design flow, even if you are not using any of these four vendors.



The demo tutorial opens with two windows.

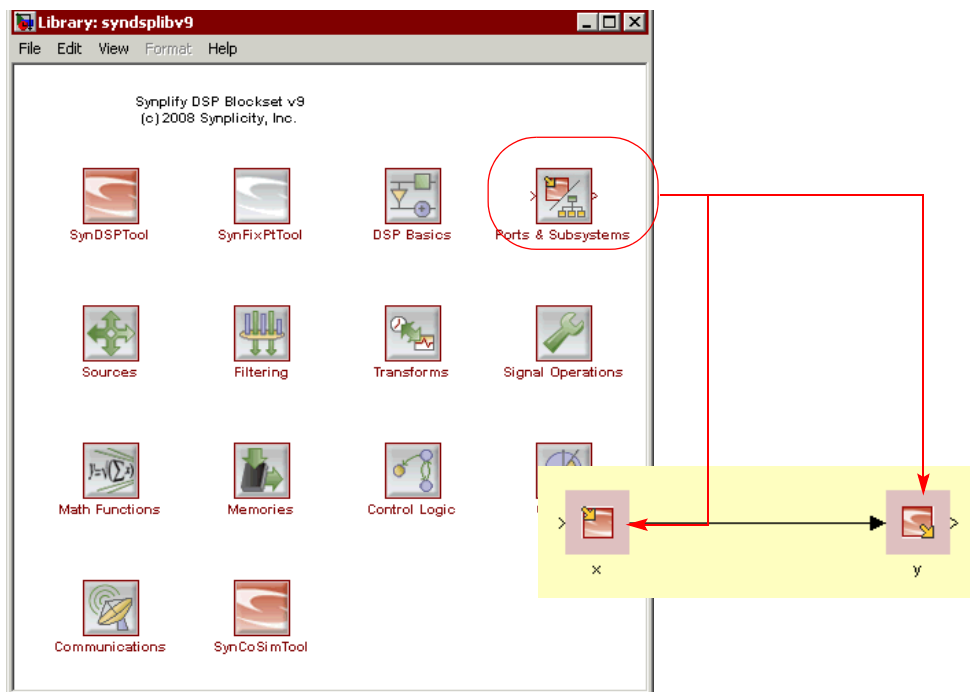
## Add Port In and Port Out Blocks

When you start the demo, two windows open:

- The model window with the first screen of the demo tutorial
- A dialog box for port parameters

The following describes the sequence of steps that was run automatically. If you are working on your own design, you would do these steps manually.

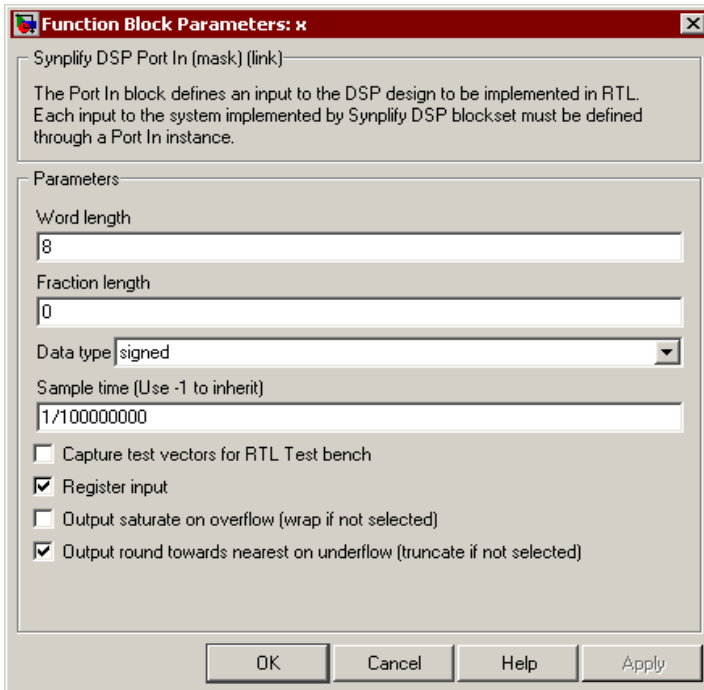
1. The demo first instantiates the Synplify DSP Port In and Port Out blocks from the Synplify DSP Ports & Subsystems library.



The model window shows the Synplify DSP Port In and Port Out blocks instantiated as *x* and *y*, respectively. Putting in these blocks satisfies the first rule for Synplify DSP design (see [Create Algorithm Models, on page 2-3](#)), which is to bound the design with these two blocks.

2. Set parameters for the Port In block.

The parameters that were set automatically are displayed in the open dialog box. Notice the settings for Word length and Sample time as displayed in the dialog box. The value of the settings might vary slightly, depending on the FPGA technology you selected. The following figure shows the Actel parameters.



3. Go to the next screen.

- Close the dialog box.
- Double-click Next in the model window to go to the next stage of the design, which is to add the FIR block.



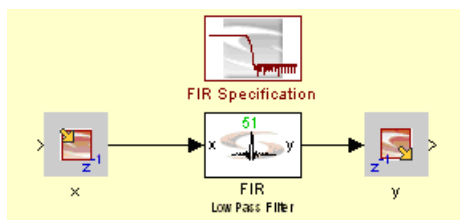
## Add the FIR Block

When you double-click Next, three things happen:

- The model window is updated to include new blocks, including the FIR.
- A dialog box opens with parameters for the FIR.
- A FIR specification toolbox window opens.

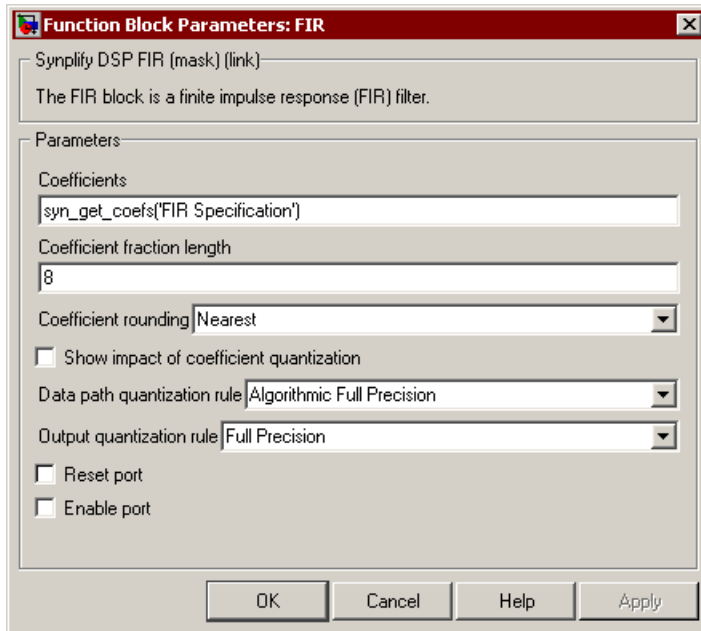
The following describes these steps that the demo runs automatically.

1. The demo automatically instantiates the following blocks:
  - The FIR block from the Synplify DSP DSP Filtering library, which it names FIR Low Pass Filter.
  - The Synplify DSP FDATool block, which it renames FIR Specification.



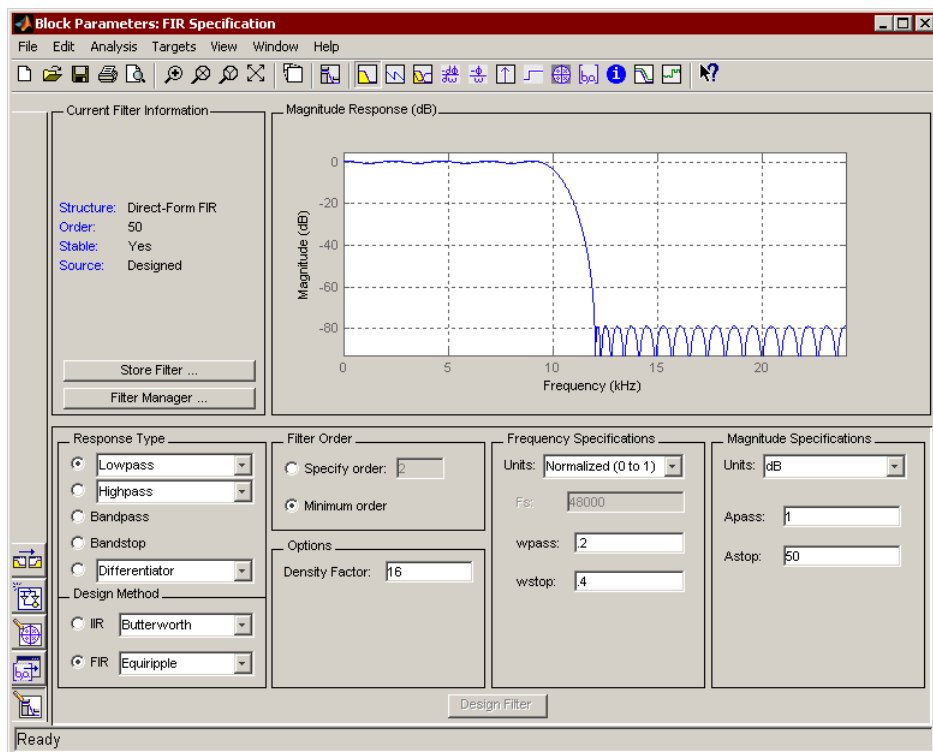
2. The demo sets parameters for the FIR Low Pass Filter block. In particular, note the following settings:

Coefficients	The <code>syn_get_coefficients</code> function in this field specifies that the tool use the coefficients set in the FIR Specification (FDATool) block. Alternatively you can use a MATLAB vector variable.
Coefficient word length	This sets the precision of the coefficient quantization.
Data path format and Output format	This selects the precision of the internal format.



3. Next, the demo defines the FIR coefficients with the FIR Specification (FDATool) block and the MathWorks Filter Design and Analysis Tool. Note the following settings in the MathWorks tool window:
  - Order: The default is 50.
  - Frequency specifications wpass and wstop
  - Magnitude specification: astop

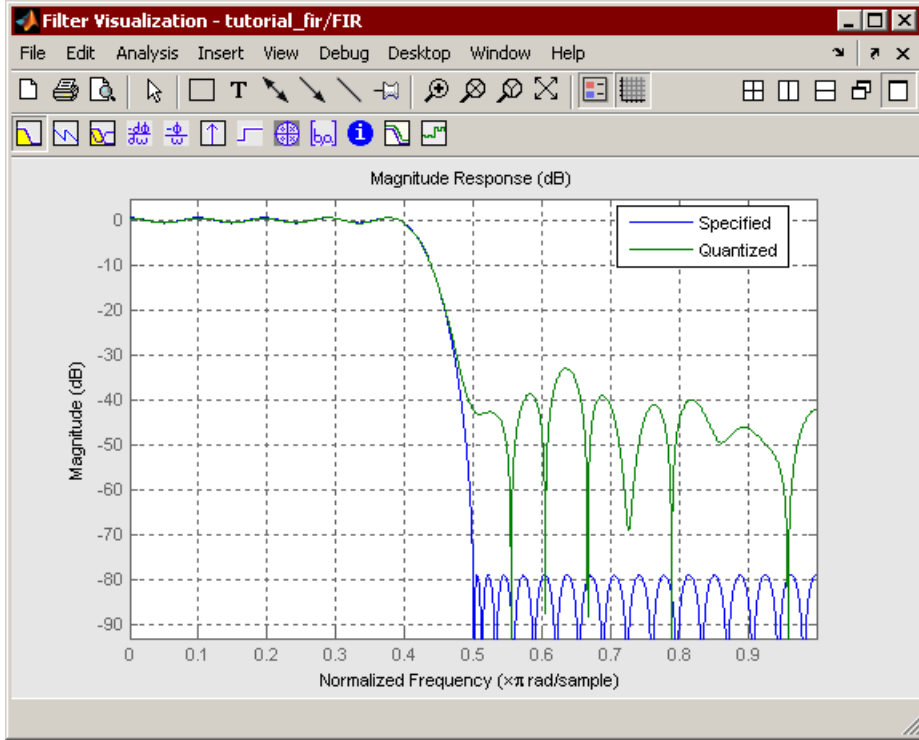
This sets full-precision FIR coefficients. The FIR block quantizes these coefficients. The FIR block icon reflects the settings, showing a 50th order FIR filter with 51 taps, because the number of coefficients (taps) specified was 50.



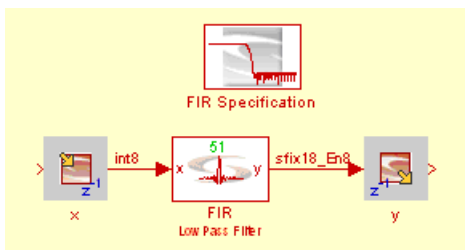
4. To view the results, do the following:

- In the FIR parameters dialog box, click Show Impact of Quantization. Another window opens and shows how the quantized coefficient compares to the full coefficient.

The quantization of a signal is determined by the quantization propagated from input signals. Each block in the Synplify DSP 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.



- To view the propagation of parameters, select Format->Port/Signal Displays in the model window, and enable the following to configure the display: Sample Time Colors, Port Data Types, and Signal Dimensions.



5. Go to the next screen by closing the dialog boxes and tool windows and double-clicking Next in the model window.



# Set up for Verification

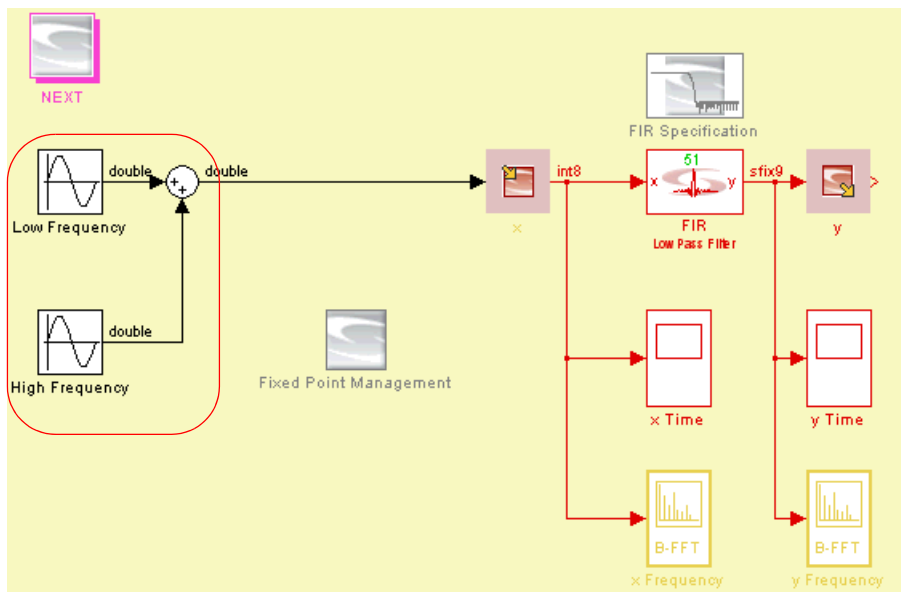
In this stage, the demo adds Simulink stimuli and analysis components to the schematic to verify the model.

- Add Stimuli Components, on page 2-11
- Add Analysis Components, on page 2-12

## Add Stimuli Components

The demo automatically runs the following steps.

1. It creates low and high frequency signals to the input (x) of the algorithm. You see the following:



- The demo automatically adds two instances of the Simulink->Sources->Sine Wave block to the design schematic to generate sine waves. The demo names them Low Frequency and High Frequency.
- It adds a Simulink->Math Operations->Sum block to the design. Note how the blocks are connected to the x input.

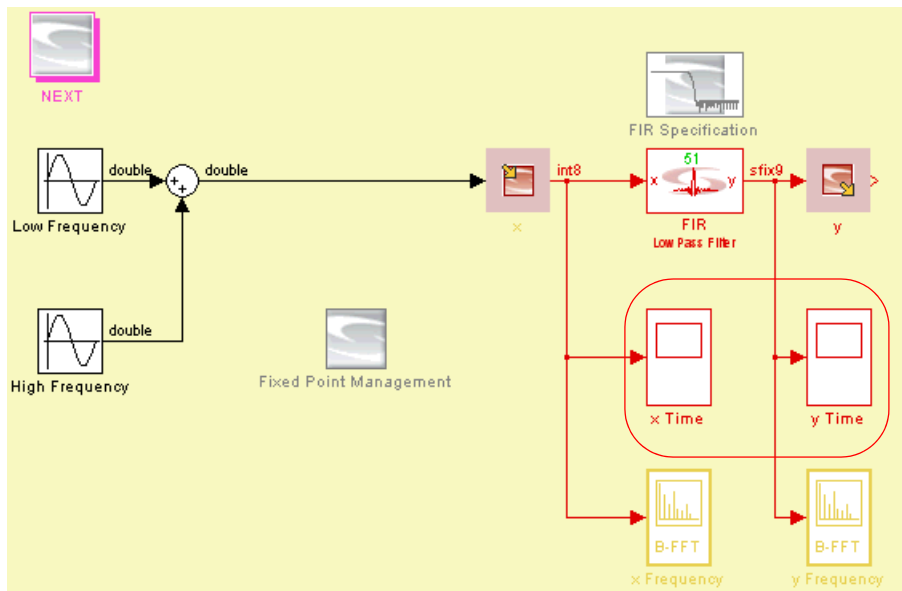
2. The demo then sets sine wave block parameters. Double-click the Low Frequency and High Frequency blocks, and note the settings for the following:

- Amplitude
- Frequency

## Add Analysis Components

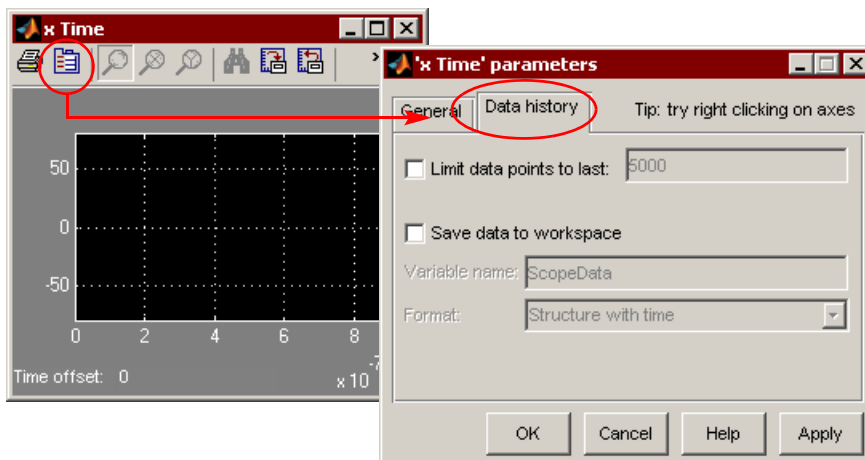
By default, Simulink does not store the data and you must explicitly set up scopes to store the data for subsequent plotting as described below. Set up the design to store data and analyze the simulation results in the time domain. For this tutorial, these steps have been run automatically, and the model window shows the finished results.

1. The demo automatically adds two instances of the Simulink->Sinks->Scope block for time domain analysis. The scopes are named x Time and y Time. The model window shows the following:

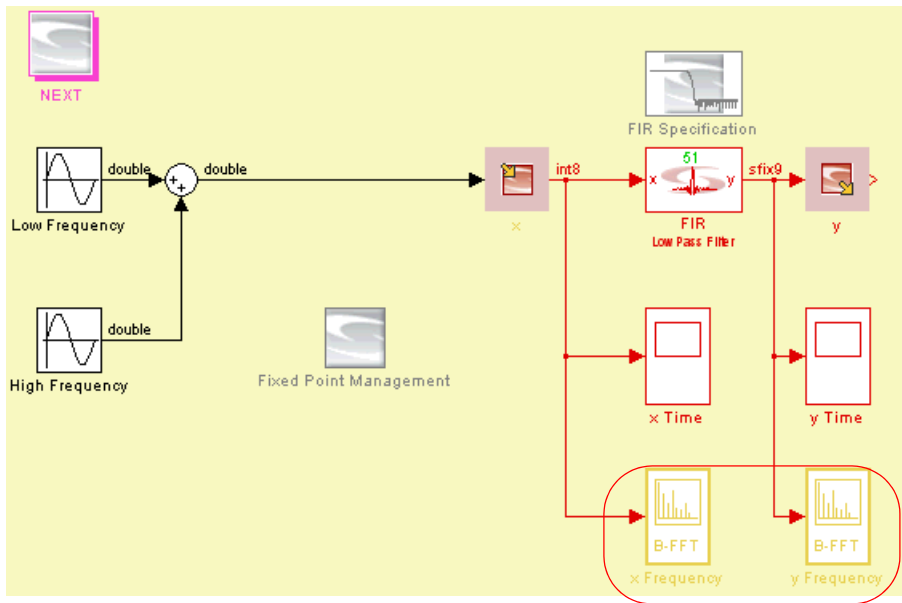


- Note how they are connected to the input and output of the FIR instance.

2. The demo sets scope parameters. You can view the settings by doing the following:
  - Double-click the x Time scope to open the scope window. Click the Parameters icon to open the x Time parameters dialog box. Note that Data History->Limit data points to last has been disabled.

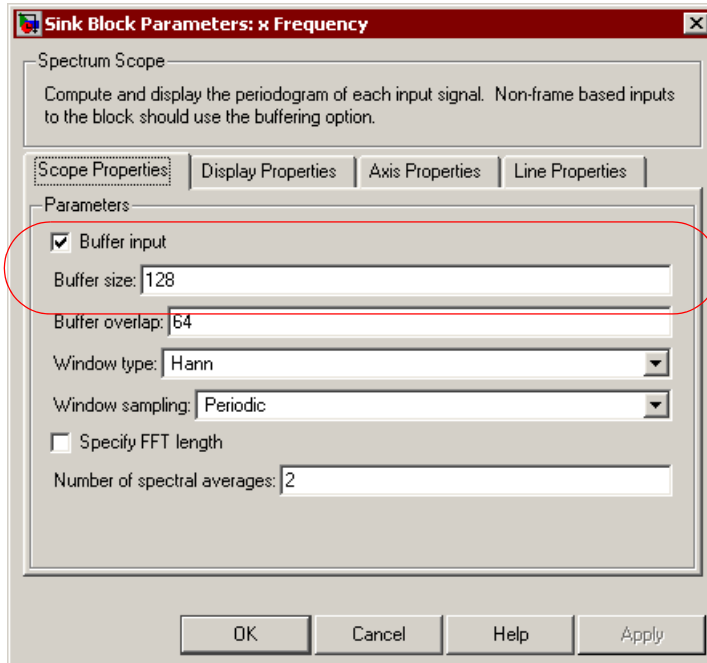


- Repeat the previous step for the y Time scope. Note that Data History->Limit data points to last has been disabled, and that Save data to workspace is enabled. Enabling this parameter stores the data points for visualization and allows the Fixed-Point Settings toolbox to display a comparison between fixed-point and floating-point results later in the flow.
  - Close the scope windows.
3. The demo automatically adds two instances of the Signal Processing Blockset->Signal Processing Sinks->Spectrum Scope block for frequency domain analysis.
    - In the demo, the scopes are named x Frequency and y Frequency.
    - Note how they are connected to the input and output of the FIR.



4. View the settings for the spectrum scopes.

- Double-click x Frequency and y Frequency.
- Note that the buffer size is set to accommodate a 128-word signal for the FFT frame (Buffer Input is enabled, and Buffer size is set to 128). The following shows the settings in the x Frequency dialog box.

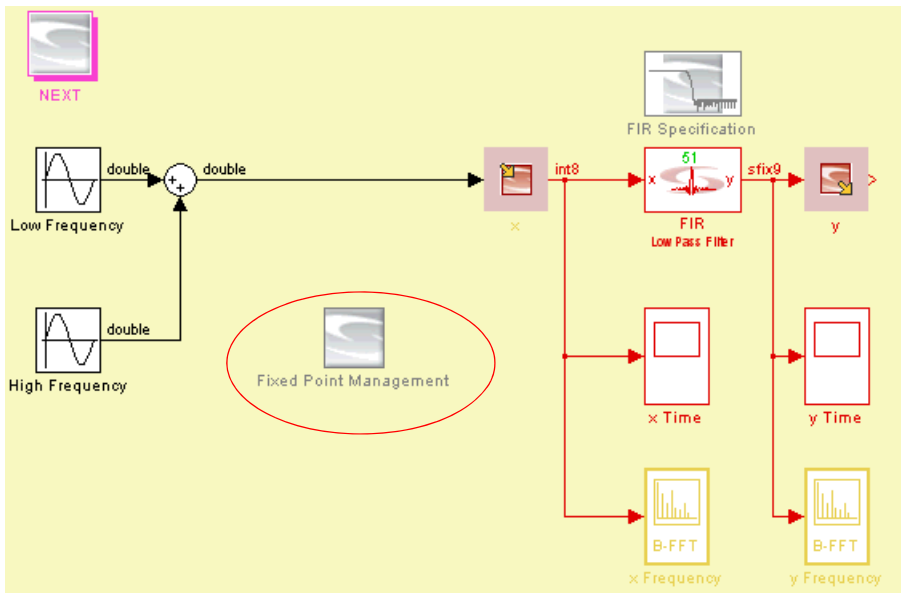


5. Close the dialog boxes for the scopes.

## Analyze and Simulate

1. The demo instantiates SynFixPtTool from the Synplify DSP Blockset to manage fixed-point settings.

Note that the demo renames the block Fixed Point Management. This block instantiates the library path. It lets you explore quantization by overriding the different blocks in the design with floating-point settings and events.



2. View the settings by double-clicking Fixed Point Management to open the Simulink Fixed-Point Settings toolbox. This toolbox provides control over the accuracy for individual levels or blocks in a hierarchy. Note the following settings:
  - Select Current System is set to tutorial\_fir, because this is a small design and so that the settings apply to all blocks.
  - Logging mode is set to Use local settings.
  - Data type override is set to Use local settings.
  - Close the window.

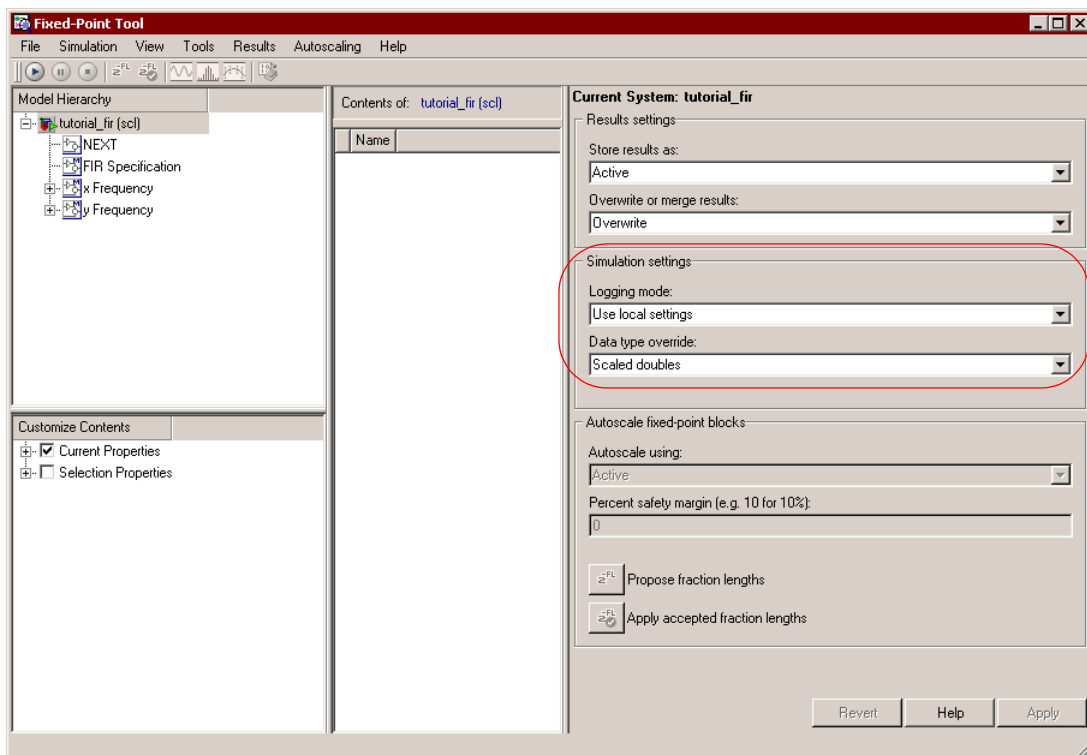
---

**Note:** Overflow logging is only supported with MATLAB 2006B, and is not supported with MATLAB 2007A or 2007B.

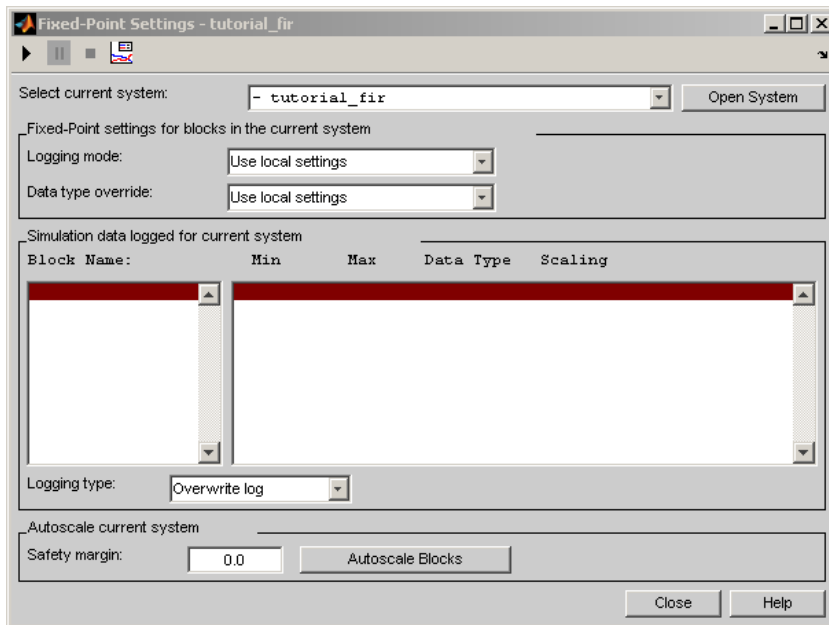
---

The MATLAB GUI in 2006B differs from the GUI in 2007A and later versions. This tutorial shows the 2007B interface, except where overflow logging is used. In this case, the tutorial shows the 2006B interface, which is the most recent MATLAB version with support for overflow logging.

The following shows the relevant portion of the toolbox as it appears in MATLAB 2007B:

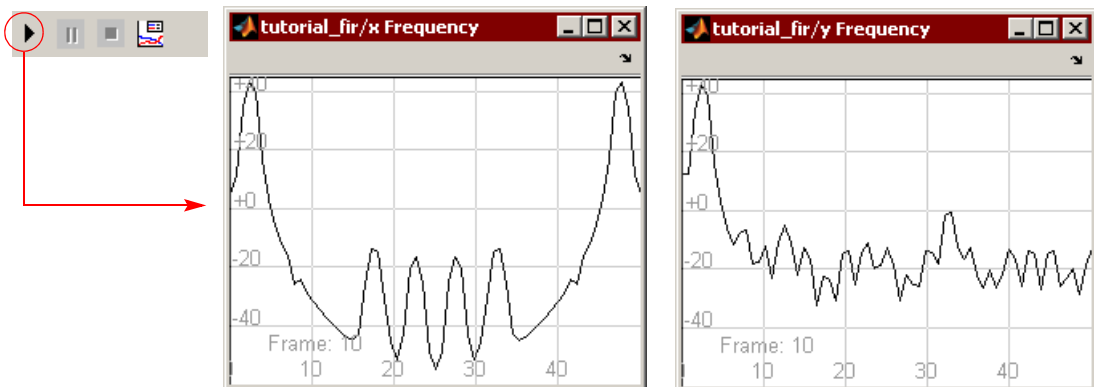


The following is the GUI as it appears in MATLAB 2006B:



3. Click the right arrow in the toolbar of the model window to simulate the design with the fixed-point settings.

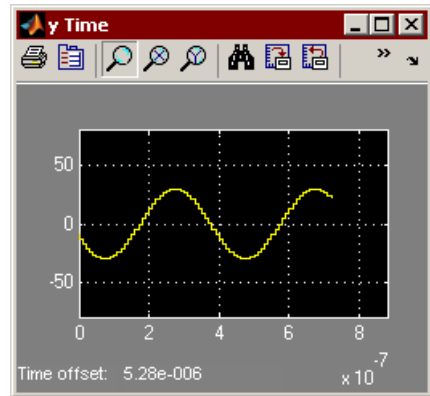
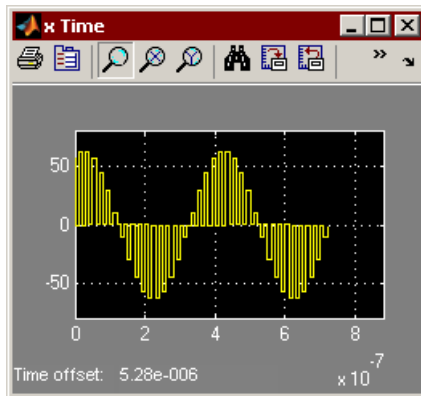
You get the results shown below. The input scope (x Frequency) shows low and high frequency spikes. For y Frequency, the high frequency has been filtered.





4. Double-click x Time and y Time and view the waveforms.

The input (x) scope shows a low-frequency signal superimposed on a high-frequency carrier, and the output scope shows the filtered low-frequency signal.



5. Close the toolbox window and scopes, and double-click Next in the model window.

# Explore Quantization Effects

The following describes how the demo analyzes quantization effects, using floating-point simulation.

- [Running Floating-Point Simulation, on page 2-20](#)
- [Analyzing the Impact of Quantization, on page 2-22](#)

## Running Floating-Point Simulation

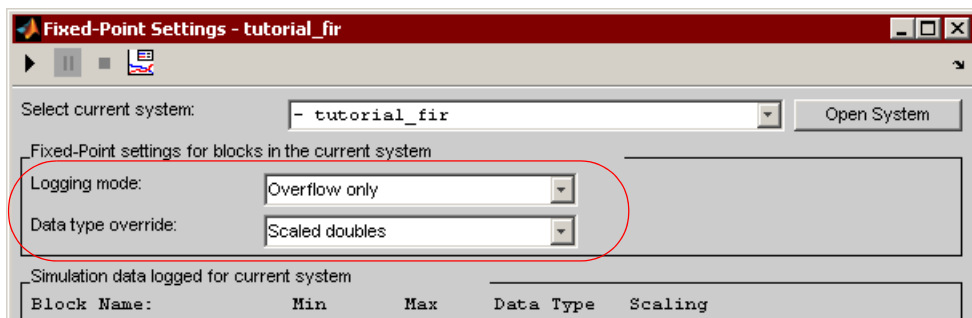
When you click Next, the following are displayed:

- The Fixed-Point Settings toolbox reopens with new settings.
- The scopes reopen with new data.

The demo automatically overrides the fixed-point settings with the floating-point format. Using floating-point settings ensures that simulation validates the algorithm with full-accuracy calculations. The Synplify DSP tool makes it easy to do this without changing your design by allowing the Simulink floating-point override to propagate through the design subsystems automatically. The following description describes how the demo overrides the original settings with the floating-point format.

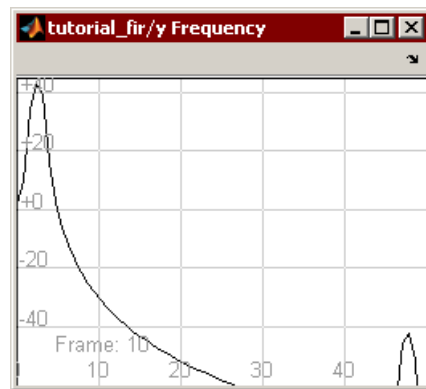
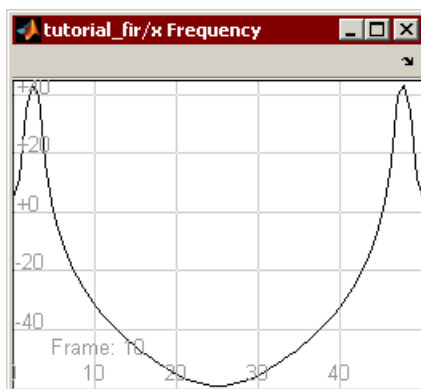
1. In the Fixed-Point Settings toolbox, note the following:
  - Data type override is set to Scaled Doubles.
  - Logging Mode is set to Overflow Only (MATLAB 2006B only). This mode is not supported in MATLAB 2007A or 2007B.

This removes quantization and lets you verify the algorithm. You can use this technique to identify quantization effects. Logging the overflow events also lets you explore the effects of quantization. The following figure shows the MATLAB 2006B interface.

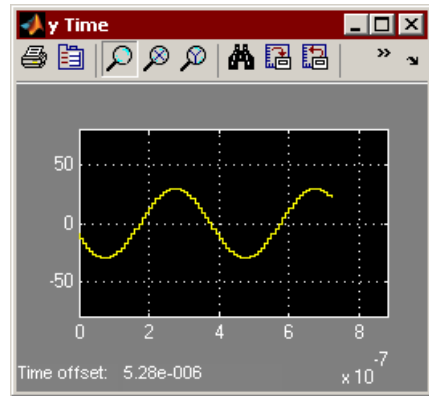
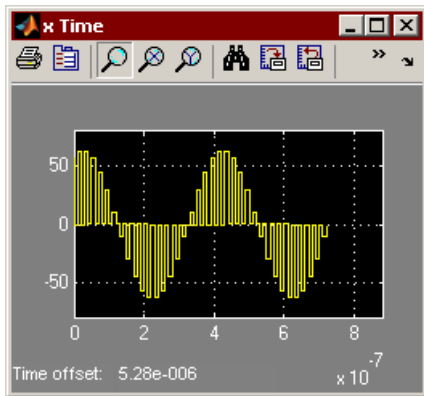


## 2. View the results.

- The demo shows the scope results after a full-accuracy simulation. It runs full floating-point simulation, overriding the previous fixed-point settings. The spectrum scope waveforms have one waveform superimposed over the other. If the spectrum waveforms exceed the graphs, use Axes->Autoscale to fit them.



- The demo shows the following result for the time domain scopes:



3. Double-click Next.

## Analyzing the Impact of Quantization

When you double-click Next after the demo runs floating-point simulation, the following changes occur:

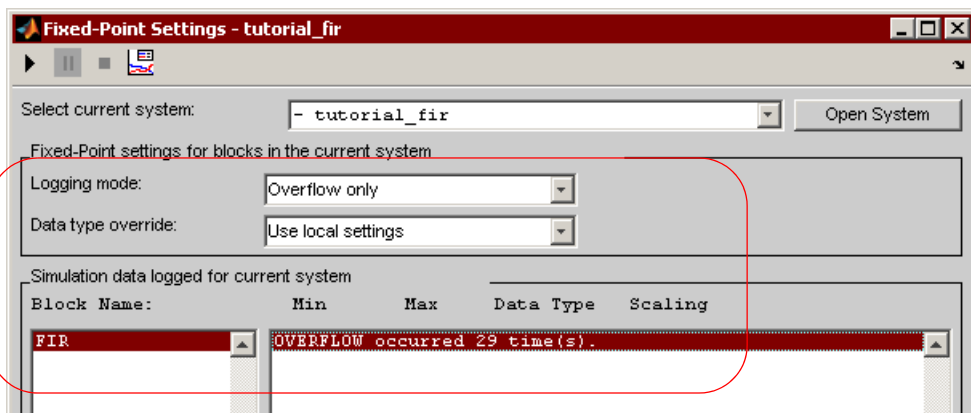
- The FIR dialog box opens with new settings.
- The Fixed-Point Settings toolbox displays new settings.
- The input and output frequency scopes are updated with new data.
- The input and output time scopes show new data.

At this point, the demo deliberately “breaks” the algorithm. This lets you see the process used to analyze the effect of quantization and isolate any problems that might occur. The following steps describe the process.

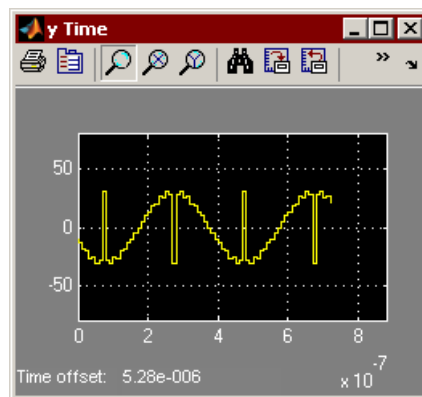
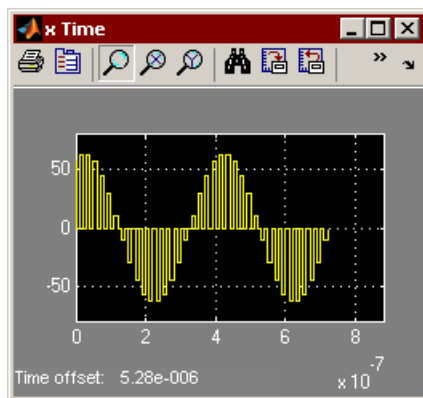
1. In the FIR dialog box, note the following changes:
  - Coefficient fraction length is reduced.
  - Output word length has changed.
  - Output fraction length has changed.

These settings help isolate and analyze quantization problems by deliberately breaking the algorithm.

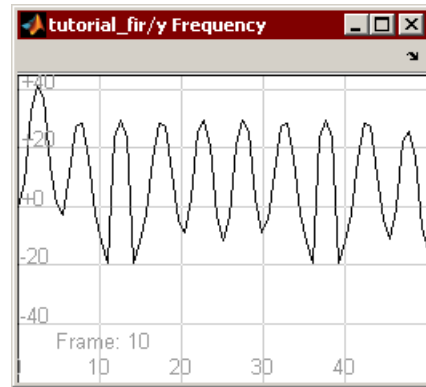
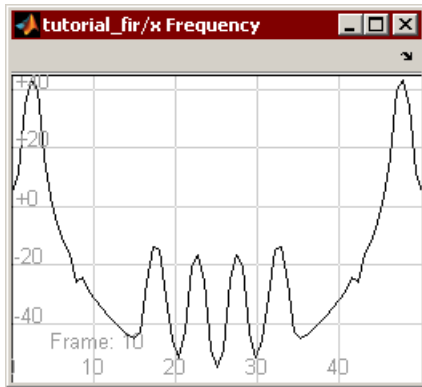
2. In the Fixed-Point Settings toolbox, note the following changes:
  - Logging Mode is set to Overflow Only and the amount of overflow is logged (MATLAB 2006B only). This mode is not supported in MATLAB 2007A or 2007B.
  - Data type override is set to Use Local Settings.



3. Check the results.
  - The time scopes show how the quantization affects the output.

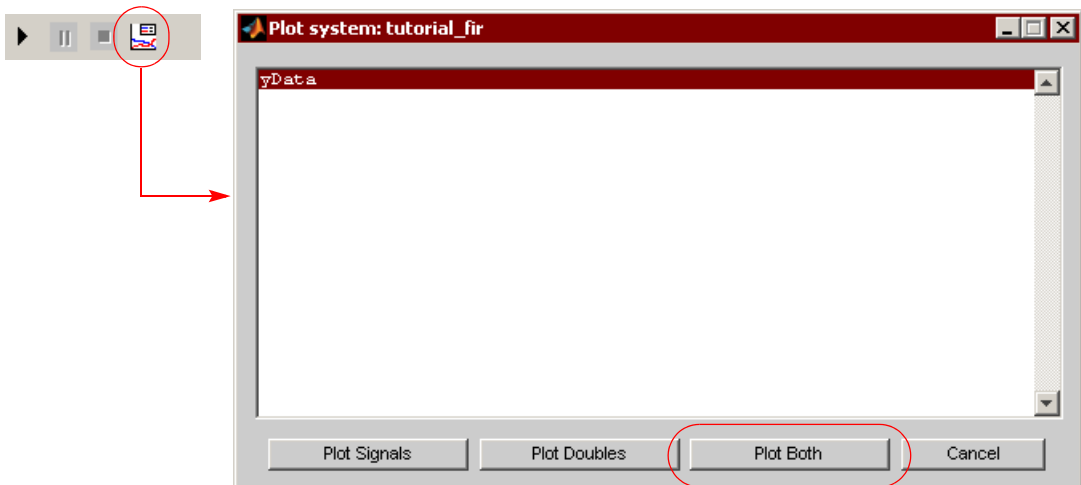


- The frequency scopes reflect similar results.

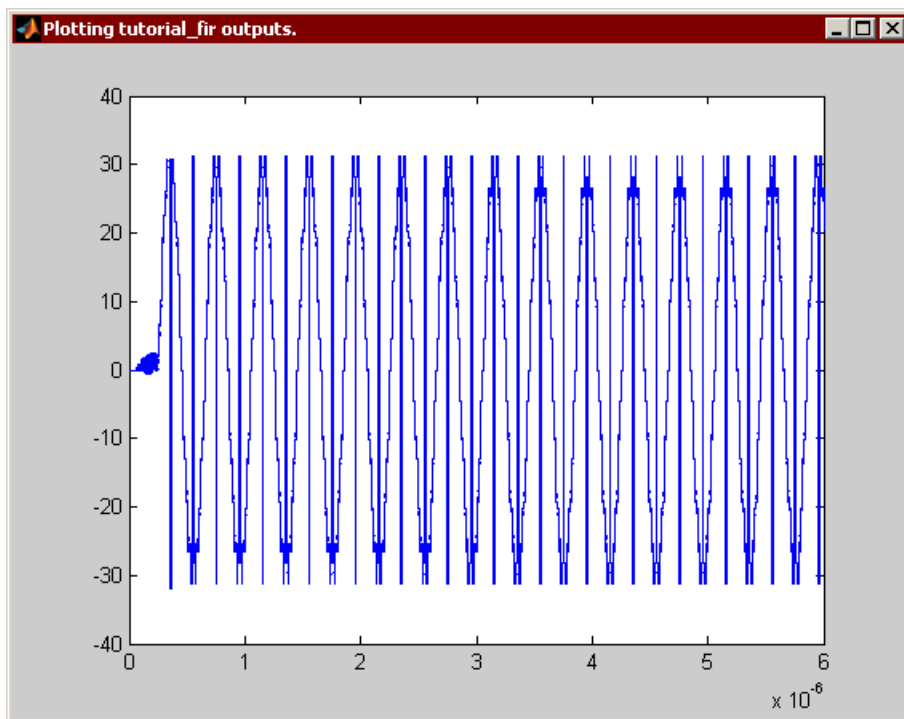


4. Plot and compare the waveforms:

- Simulate the design by clicking the right arrow in the Fixed-Point Settings toolbox.
- Double-click the Plot icon in the Fixed-Point Settings toolbox.
- In the plot window that opens, click Plot Both.



- Check the plotted waveforms.



5. Double-click Next.

## Synthesize Optimized Architectures

For more information about the next stages in the flow, see the following:

- [Run DSP Synthesis, on page 2-26](#)

- [Verify RTL, on page 2-29](#)

The tutorial skips this optional step, but you can refer to [Verifying the RTL with a Test Bench, on page 4-63](#) for a detailed procedure.

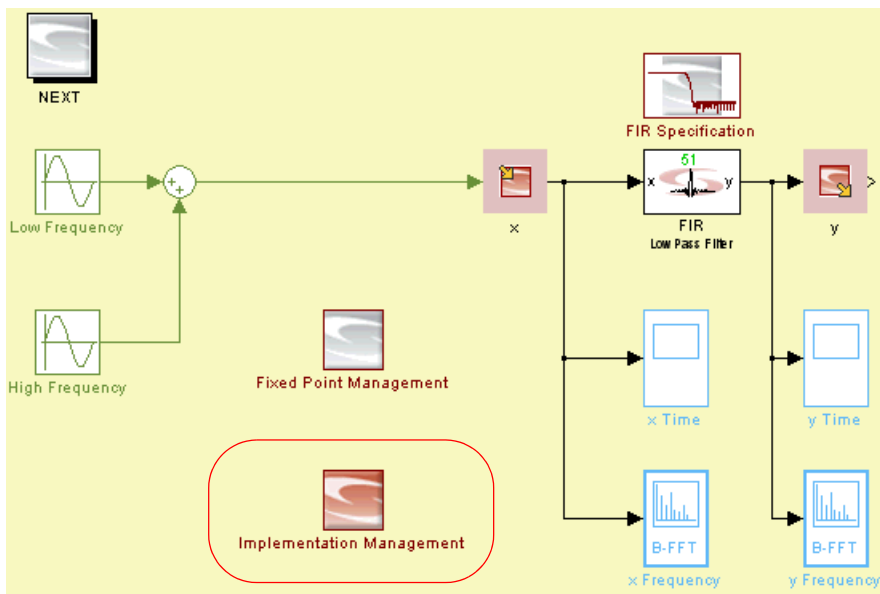
## Run DSP Synthesis

When you double-click Next after exploring quantization effects, the demo resets the fixed-point settings and runs synthesis. You see the following changes:

- The model window now includes the SynDSPTool block.
- The Synplify DSP FPGA window (for DSP synthesis) is open.
- The Synplify Pro UI (for logic synthesis) is open.

You manage optimization strategies with the SynDSPTool block. The following steps describes how the demo instantiates and uses this block and then runs DSP synthesis.

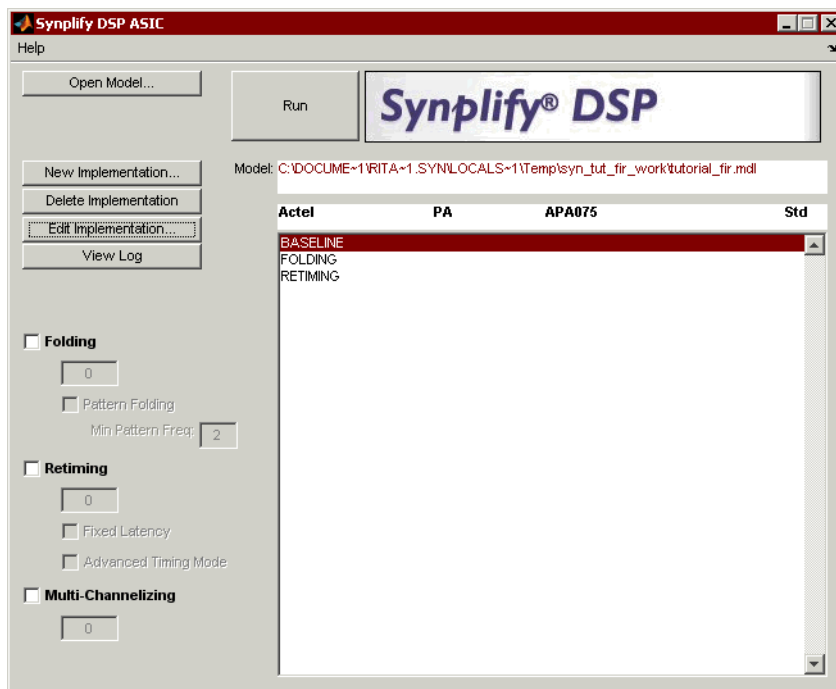
1. The model window shows an instance of the SynDSPTool block from the top-level Synplify DSP library. In the demo, it is renamed Implementation Management.



2. Double-clicking Implementation Management opens the Synplify DSP FPGA window.



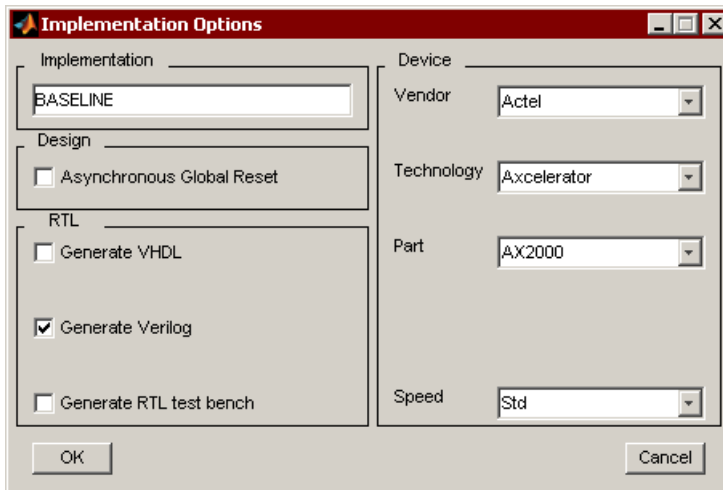
The demo does this automatically and displays a window, customized for the target technology you chose at the beginning of the tutorial. The following figure shows the window with an Actel target.



It also shows three implementations (Baseline, Folding, and Retiming) it has created. Each implementation explores different optimization strategies for the same design and stores it in a separate implementation. The implementation is a subdirectory, parallel with the .mdl file associated with the design, and contains any files generated for that particular implementation. The following steps describe the process that the demo ran through automatically.

3. The demo first set up the implementation and implementation options. You can review the steps by doing the following.
  - Select BASELINE and then click Edit Implementation. If you were trying to create a new implementation, you would click New Implementation. Either of these actions opens the Implementation Options dialog box, where you can set options specific to that implementation.
  - Check the settings. The following shows the settings for the Actel demo, so an Actel part and technology is selected. The selected target

is also reflected in the Synplify DSP window, just above the implementations.



- Click Cancel to close the dialog box.
- Return to the Synplify DSP window and note that no optimizations, like retiming and folding, have been enabled for this implementation.

The other implementations have different settings, which we will explore later. The details are described in [Refine Optimizations, on page 2-31](#).

4. Next, the demo automatically runs DSP synthesis and generates output files. You do not need to do this because this has already been done, but to replicate this step manually, you would select BASELINE in the Synplify DSP window and click Run.
5. Click View Log in the Synplify DSP FPGA window to see a summary of the DSP synthesis run. Close the log window.

The next step, to verify the RTL, is optional, and this tutorial does not do this, but goes on to logic synthesis ([Run Logic Synthesis, on page 2-29](#)).

## Verify RTL

This is an optional step, and the demo does not include it. For a detailed procedure for verifying the RTL, see [Verifying the RTL with a Test Bench, on page 4-63](#).

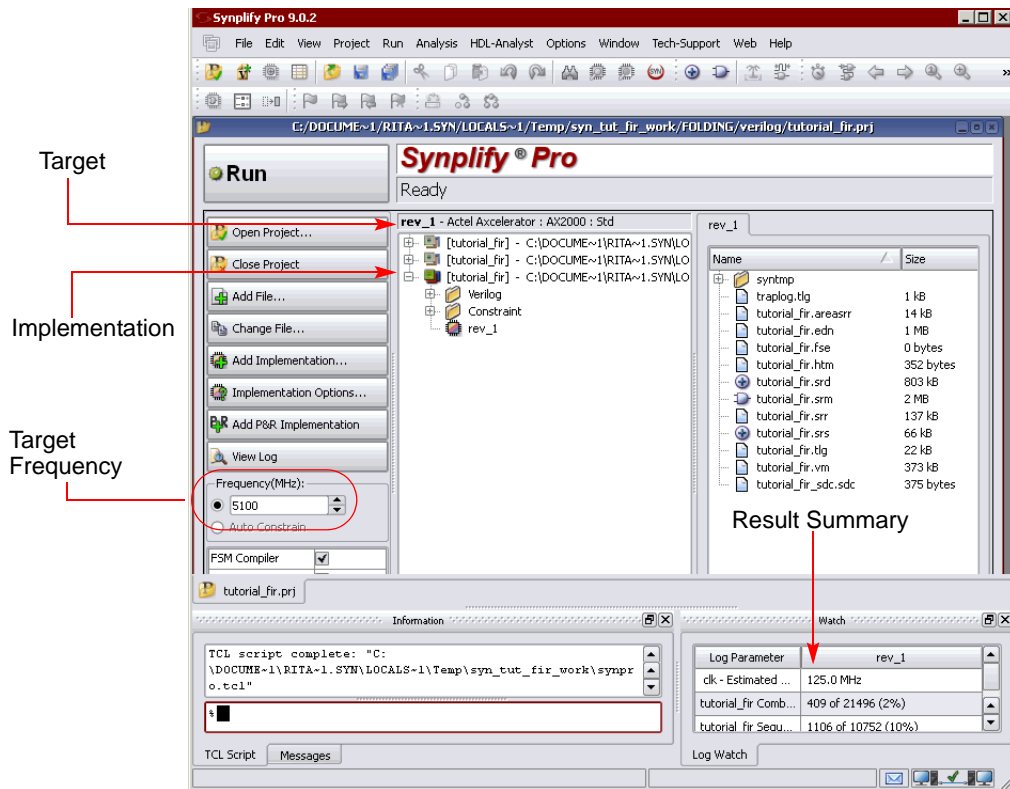
## Run Logic Synthesis

After DSP synthesis, the demo automatically starts Synplify Pro and runs logic synthesis on the design. It displays the Synplify Pro window with the three implementations and their results. This section walks through the procedure step-by-step, using the BASELINE implementation.

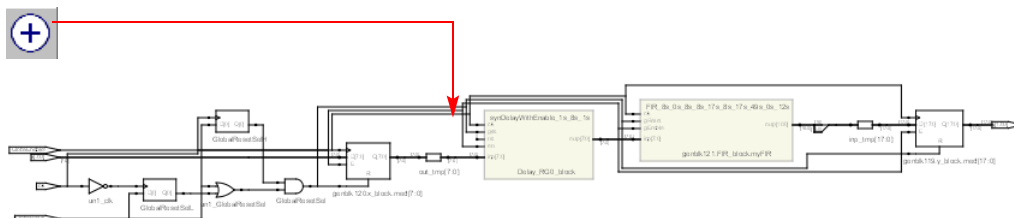
As a result of DSP synthesis, the following files are generated for logic synthesis in the <design\_implementation>/vhdl or verilog subdirectory:


File	Description
<design>.sdc	Synplicity Design Constraints generated for the design.
<design>.prj	Synplicity Project File generated for the design.
<design>.vhd or .v	The RTL associated with the design.

1. The demo automatically ran Synplify Pro. Examine the results of logic synthesis for the BASELINE implementation by doing the following:
  - In the Synplify Pro project window, select the BASELINE implementation.
  - Note that logic synthesis was run with the same FPGA target you selected. This figure shows an Actel implementation.



2. View the implementation.
  - Open the RTL view by clicking on the icon.



- Push down into the FIR module by clicking the  icon and selecting the FIR. View the implemented architecture.

The structure reflects a transposed implementation of the FIR filter: the input goes to different multipliers with each multiplier feeding two different adders (this is a linear phase filter with symmetric coefficients), and the identical coefficients share a multiplier). The adders are registered and accumulated for the final result.

- Close the RTL view.

3. Return to the main Synplify Pro window and check the results summary in the Log Watch window at the lower right. Compare the results to the target frequency.

Note that the results documented here may vary from your results if you used another target or another version of Synplify Pro. The other implementations in the demo illustrate how you can use Synplify DSP optimizations to produce better logic synthesis results. See [Refine Optimizations, on page 2-31](#) for details.

## Refine Optimizations

The demo uses the other implementations to illustrate optimization strategies. In your design cycle, you can iterate with different implementations to fine-tune your design or try out different options and strategies.

This section describes the optimization strategies available and then walks you through using some techniques to improve performance and area optimization in the tutorial design:

- [Optimization Strategies, on page 2-32](#)
- [Using Retiming for Performance, on page 2-33](#)
- [Using Folding to Decrease Area, on page 2-34](#)

## Optimization Strategies

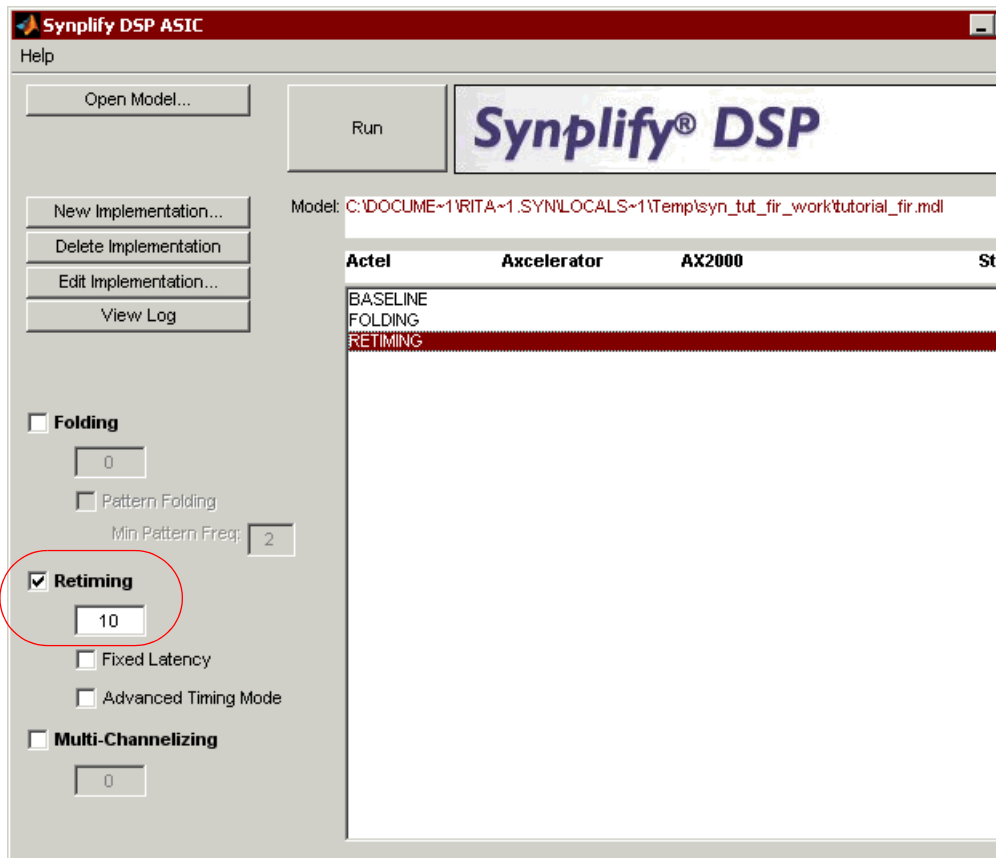
The Synplify DSP software offers the following optimization strategies:

- **Retiming**  
Moves existing registers from non-critical to critical performance situations. Optional extra latency for the complete block adds extra register resources for pipeline insertion. The tutorial illustrates this technique in [Using Retiming for Performance, on page 2-33](#).
- **Multi-Channelization**  
Multiple data streams share hardware for area optimization. This strategy requires the physical clock for the implementation to accommodate a clock rate equivalent to the sample rate of the individual data streams multiplied by the number of streams sharing the hardware. The tutorial does not illustrate this, but you can refer to [Optimizing with Multichannelization, on page 4-60](#).
- **Folding**  
A single data stream shares hardware for area optimization. This strategy requires the physical clock for the implementation to accommodate a clock rate equivalent to the sample rate of the data stream multiplied by the requested folding factor. Folding requires retiming (to bring registers to the folding boundaries). The tutorial illustrates this technique in [Using Folding to Decrease Area, on page 2-34](#).

## Using Retiming for Performance

The following procedure shows you how the demo used retiming to improve performance. It automatically created an implementation called RETIMNG

1. Return to the Synplify DSP window and select the RETIMNG implementation.
2. Note the following:
  - The RETIMING option is set. The following figure shows the Actel implementation.
  - Click View Log and check the file. You see that DSP synthesis was run with this option on and the specified number of latency cycles.



3. Go to the Synplify Pro view and select the Retiming implementation in that window.

The window is updated with the relevant data after the logic synthesis run for this implementation.

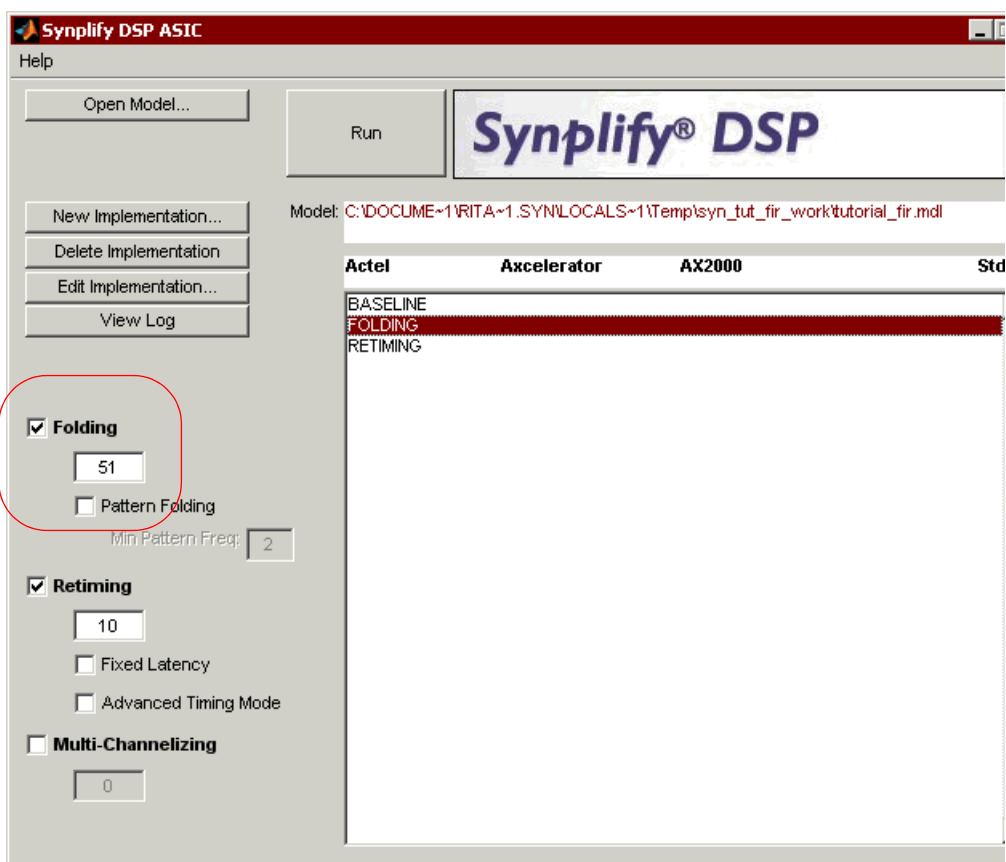
4. Check the following:
  - Check the Log Watch window in the lower right. You see that timing frequency has improved from the BASELINE implementation.
  - When you examine the architecture in the RTL view (see [Run Logic Synthesis, on page 2-29](#) for details), you see that the structure still reflects a direct-form, transposed implementation of the FIR filter. The input of the filter and the outputs of the multipliers are now all registered, and this results in improved timing performance.

## Using Folding to Decrease Area

To deal with area challenges, use folding. Folding executes the hardware with the physical clock running at a multiple of the sample clock.

1. Return to the Synplify DSP window and select the FOLDING implementation. The following figure shows the Actel implementation.





2. Note the following:

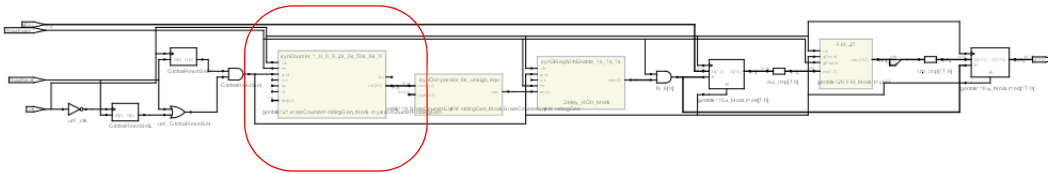
- The Retiming and Folding options are both enabled. Selecting Folding automatically enables Retiming.
- Click View Log and check the file. You see that DSP synthesis was run with a folding factor of 51. This specifies that the physical clock can run 51 times faster than the sample clock to enable resource sharing.

3. Go to the Synplify Pro view and select the Folding implementation in that window.

The window is updated with the relevant data after the logic synthesis run for this implementation.

4. Check the following:

- When you check the Log Watch window in the lower right, you see that the resources (number of cells) has been significantly reduced, compared to the BASELINE and FOLDING implementations.
- When you examine the architecture in the RTL view (see [Run Logic Synthesis, on page 2-29](#) for details), you see that the structure still reflects a direct-form, transposed implementation of the FIR filter, but it now includes a counter, to manage the multiplexers over the shared resources.



This illustrates how resources are shared and implemented efficiently by the folding optimization. You can see the addressing logic for the coefficient ROM and the input RAM data storage. With this optimized architecture, not only does the design meet the target performance, but the area is substantially reduced too.

Now that you have completed the tutorial, you are familiar with the design flow, and can use Synplify DSP for your own designs.

## CHAPTER 3

# Synplify DSP Underlying Concepts

---

This document describes some DSP basics, and how they are handled in the Synplify DSP tool. It describes the following:

- [Clock Management, on page 3-2](#)
- [CORDIC Algorithms, on page 3-3](#)
- [Data Types, on page 3-19](#)
- [Resets in Synplify DSP, on page 3-22](#)
- [Multi-Rate Design, on page 3-25](#)

# Clock Management

The Synplify DSP software uses MATLAB sample domains to derive physical clock domains. This section describes how the Synplify DSP tool handles signal and implementation clocks.

## 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.

## Implementation Clocks

The Synplify DSP 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.

# 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 Volder<sup>1</sup>. Andraka also has a good overview<sup>2</sup>. The extension towards hyperbolic algorithms was first introduced by Walther<sup>3</sup>, with a comprehensive overview by Dawid/Meyer<sup>4</sup>. For an overview of the Synplify DSP implementation, see [Circular, Linear, and Hyperbolic Coordinate Systems, on page 8-67](#). Synplify DSP 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 3-4](#)
- [Unified CORDIC Applications, on page 3-13](#)

---

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/Classes/EE290sp99/lectures/ee290aSp996\\_1/cordic\\_chap24.pdf](http://www.eecs.berkeley.edu/~newton/Classes/EE290sp99/lectures/ee290aSp996_1/cordic_chap24.pdf)

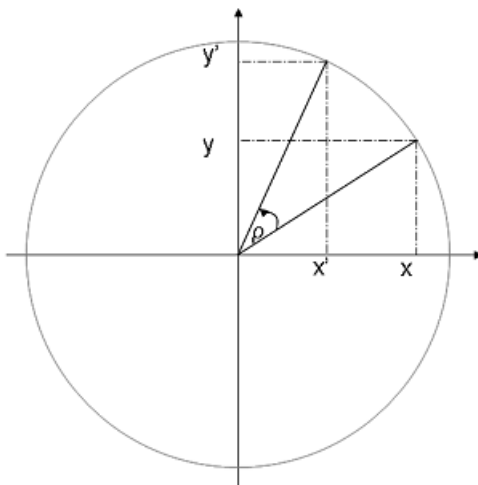
## CORDIC Definitions

This section describes the following, listed here in alphabetical order:

- [CORDIC Algorithm, on page 3-7](#)
- [CORDIC Angle, on page 3-5](#)
- [CORDIC Gain, on page 3-10](#)
- [CORDIC Mode, on page 3-11](#)
- [CORDIC Range, on page 3-9](#)
- [CORDIC Rotation, on page 3-6](#)
- [CORDIC Rotator, on page 3-8](#)
- [Rotation Transform, on page 3-5](#)
- [Unified CORDIC, on page 3-12](#)
- [Vector Rotation, on page 3-4](#)

### Vector Rotation

Vector rotation takes a vector  $(x,y)$  and rotates it over an angle  $\rho$  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

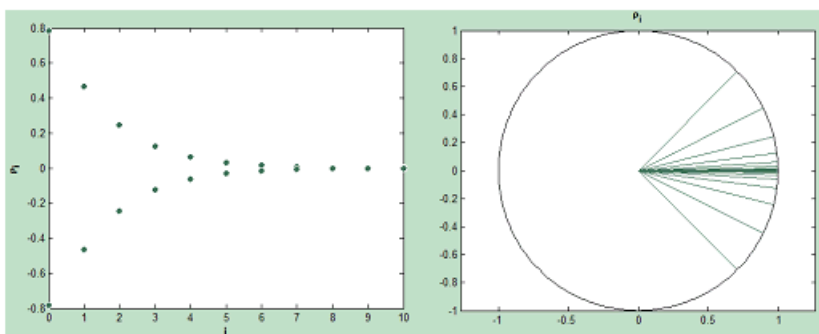
Restrict the rotation angle  $\rho$  to satisfy (integer  $i \geq 0$ ):

$$\tan \rho_i = \pm \frac{1}{2^i} = \delta_i 2^{-i}$$

$$\cos \rho_i = \cos(-\rho_i) = \frac{1}{\sqrt{1 + 2^{-2i}}} = K_i$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = K_i \begin{bmatrix} 1 & -\delta_i 2^{-i} \\ \delta_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

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  $\delta_i = -1$ .

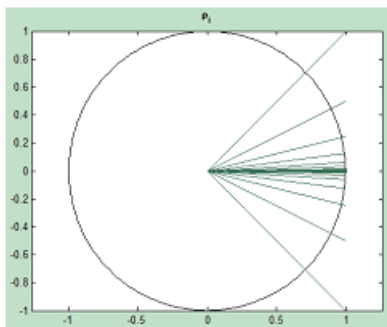


Index	$\tan \rho_i$	$\rho_i$ (rad)	$\rho_i$ (deg)
0	1	.785 ( $\pi/4$ )	45.0
1	1/2	.464	26.6
2	1/4	.245	14.0
3	1/8	.124	7.13
4	1/16	.062	3.58
5	1/32	.031	1.79
6	1/64	.016	.90
7	1/128	.008	.45

## 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

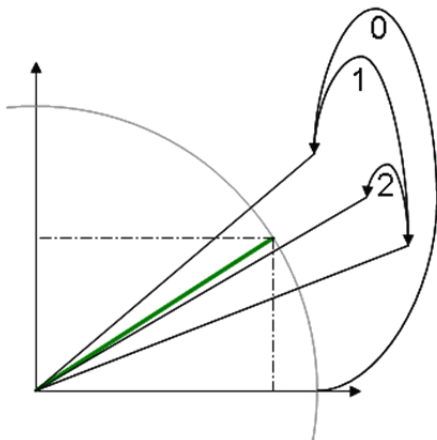
You can obtain an arbitrary rotation angle through a sequence of CORDIC rotations of successively declining CORDIC angles, with variable rotation direction. The decision to rotate clockwise  $\delta_i = 1$  or counter clockwise  $\delta_i = -1$  decomposes the desired rotation angle  $\rho$  into microrotations  $\rho_i$ :

$$\rho = \sum_{i=0}^{n-1} \delta_i \cdot \rho_i \quad \left| \begin{array}{l} z_0 = 0 \\ z_{i+1} = z_i - \delta_i \cdot \rho_i \\ \rho = z_n \end{array} \right.$$

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

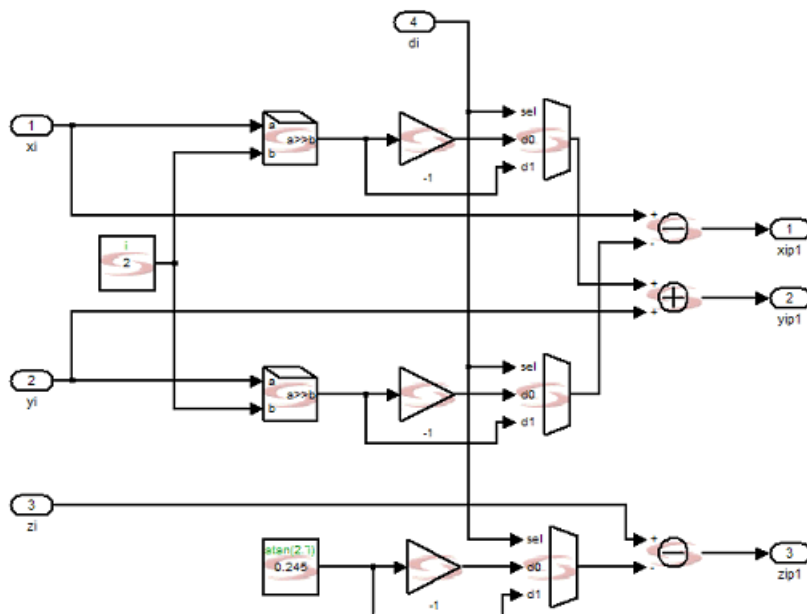
The CORDIC algorithm is an iterative manipulation of 3 simple equations, calculating a vector  $(x_i, y_i)$  and an angle accumulator  $z_i$ . A device capable of doing so is a CORDIC rotator or CORDIC processor.

$$x_{i+1} = x_i - y_i \cdot \delta_i \cdot 2^{-i}$$

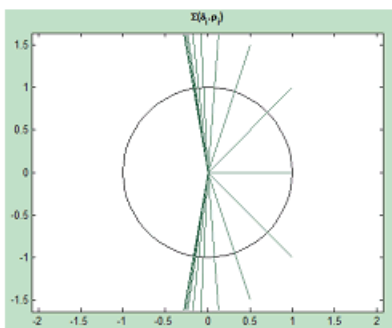
$$y_{i+1} = y_i - x_i \cdot \delta_i \cdot 2^{-i}$$

$$z_{i+1} = z_i - \delta_i \cdot \tan^{-1} \cdot 2^{-i}$$

Note that the CORDIC algorithm or sequence of CORDIC rotations is uniquely defined by the sequence  $\delta_i$ , 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:

Xilinx ISE 9.2i: Xilinx IP Core Catalog, Xilinx ISE 9.2i: Xilinx IP Core Catalog  
Xilinx ISE 9.2i: Xilinx IP Core Catalog, April 2008

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.

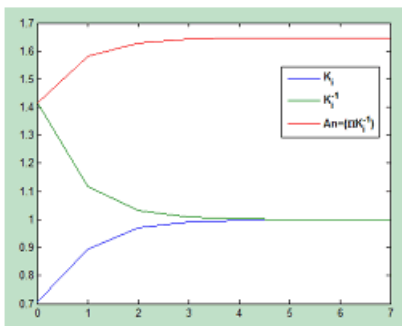
## CORDIC Gain

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  $\rho$ , 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  $\rho$ . 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 <sup>3</sup> 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 8-67.

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 Synplify DSP CORDIC Rotator block does this automatically. The gain of .8282 takes this into account.

	Rotation	Vectoring
<b>Circular</b>	$x_n = A_n (x_0 \cdot \cos \rho - y_0 \cdot \sin \rho)$ $y_n = A_n (y_0 \cdot \cos \rho + x_0 \cdot \sin \rho)$ $z_n = 0$ $A_n = \prod_n \sqrt{1 + 2^{-2i}} \approx 1.647$	$x_n = A_n \sqrt{x_0^2 + y_0^2}$ $y_n = 0$ $z_n = z_0 + \tan^{-1} \frac{y_0}{x_0}$ $A_n = \prod_n \sqrt{1 + 2^{-2i}} \approx 1.647$
<b>Linear</b>	$x_n = x_0$ $y_n = y_0 + x_0 \cdot z_0$ $z_n = 0$	$x_n = x_0$ $y_n = 0$ $z_n = z_0 + \frac{y_0}{x_0}$
<b>Hyperbolic</b>	$x_n = A_n (x_0 \cdot \cosh \rho - y_0 \cdot \sinh \rho)$ $y_n = A_n (y_0 \cdot \cosh \rho + x_0 \cdot \sinh \rho)$ $z_n = 0$ $A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx .8282$	$x_n = A_n \sqrt{x_0^2 - y_0^2}$ $y_n = 0$ $z_n = z_0 + \tanh^{-1} \frac{y_0}{x_0}$ $A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx .8282$

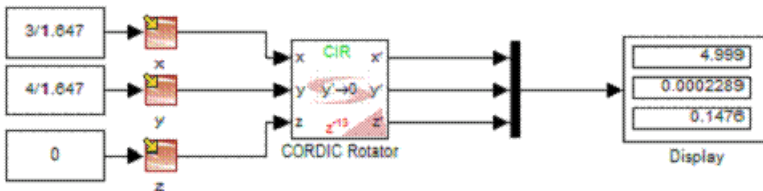
## Unified CORDIC Applications

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.

$$\left. \begin{aligned} x &= \frac{x_0}{A_n} \\ y &= \frac{y_0}{A_n} \\ z &= 0 \end{aligned} \right| \begin{aligned} x' &= M \\ y' &= 0 \\ z' &= \rho \end{aligned}$$



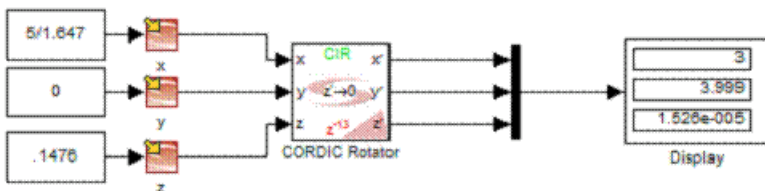
$$\sqrt{3^2 + 4^2} = 5$$

$$\tan^{-1} \frac{4}{3} / 2\pi = .1476$$

## 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=ρ:

$$\left. \begin{aligned} x &= \frac{M}{A_n} \\ y &= 0 \\ z &= \frac{\rho}{2\pi} \end{aligned} \right| \begin{aligned} x' &= M \cos \rho \\ y' &= M \sin \rho \\ z' &= 0 \end{aligned}$$

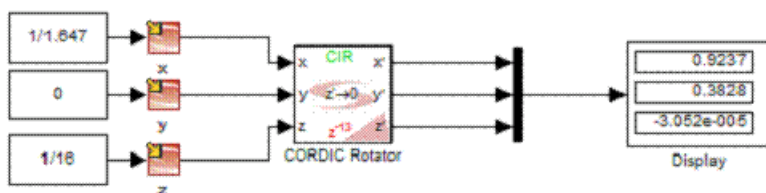




## Cosine and Sine

Use the CORDIC engine in rotation mode, in a circular coordinate system. Apply the unit vector to the X-axis and make  $z=\rho$ :

$$\left. \begin{aligned} x &= \frac{1}{A_n} \\ y &= 0 \\ z &= \frac{\rho}{2\pi} \end{aligned} \right| \begin{aligned} x' &= \cos \rho \\ y' &= \sin \rho \\ z' &= 0 \end{aligned}$$



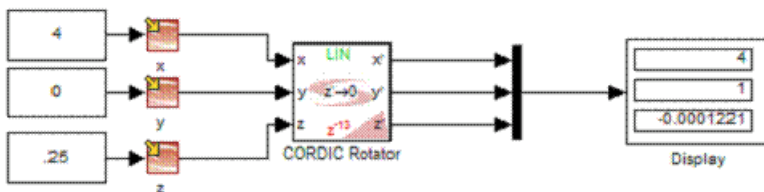
$$\cos \frac{\pi}{8} = .9239$$

$$\sin \frac{\pi}{8} = .3827$$

## 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  $z=0$ .

$$\left. \begin{aligned} x &= a \\ y &= 0 \\ z &= b \end{aligned} \right| \begin{aligned} x' &= a \\ y' &= a \cdot b \\ z' &= 0 \end{aligned}$$

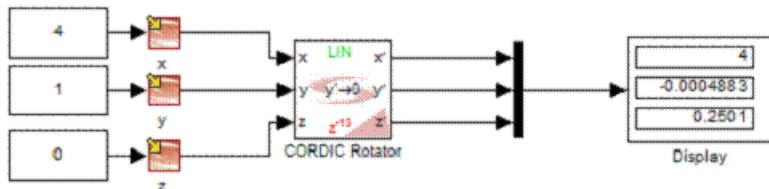


This multiplication is similar to a regular shift-add implementation, but there are more efficient architectures available.

## 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  $z=0$ .

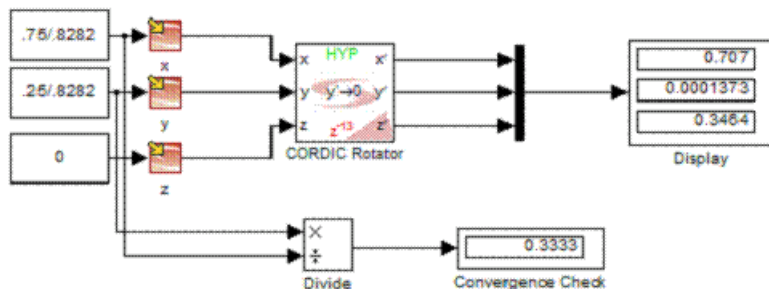
$$\begin{array}{l|l} x = \frac{(a + .25)}{A_n} & x' = \sqrt{a} \\ y = \frac{(a - .25)}{A_n} & y' = 0 \\ z = 0 & z' = \rho \end{array}$$

Note that the hyperbolic convergence puts a limitation on the input:

$$\left| \frac{y}{x} \right| < .81$$

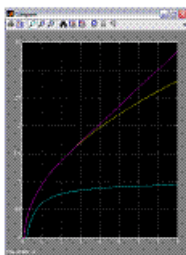
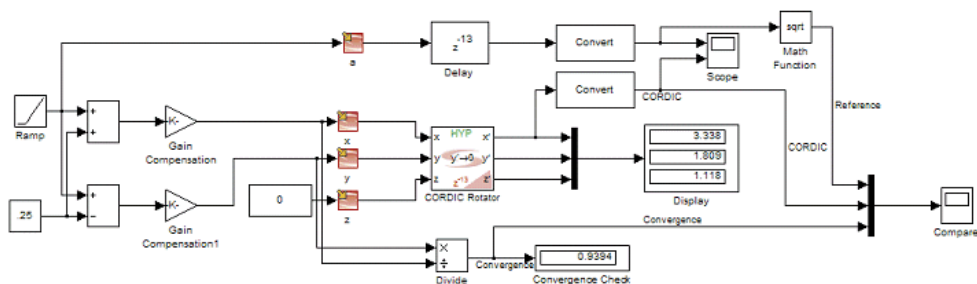
$$\frac{(a-1)/4}{(a+1)/4} < .81$$

$$a \leq 2.38$$



$$\sqrt{5} = .7071$$

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  $\rho$ , 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

$$\sqrt{a} = 2_n \sqrt{\frac{a}{2^{2n}}}$$

# 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 3-19](#)
- [Synplify DSP Data Type Implementation, on page 3-20](#)
- [Fixed-Point Data Type, on page 3-20](#)
- [Data Type Casting: Setting the Output Data Type, on page 3-21](#)

## 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 3-20](#) for additional information.

- Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field.



## Synplify DSP Data Type Implementation

The Synplify DSP 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. Synplify DSP 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 Synplify DSP tool, see [Using Quantization Analysis Tools, on page 4-38](#).

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 <b>Output data type</b> 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 <b>Output word length</b> parameter, or the word length that is inherited from the driving block or through back propagation.
Scaling	Reflects the value of the block's <b>Output fraction length</b> 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 Synplify DSP 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

# Resets in Synplify DSP

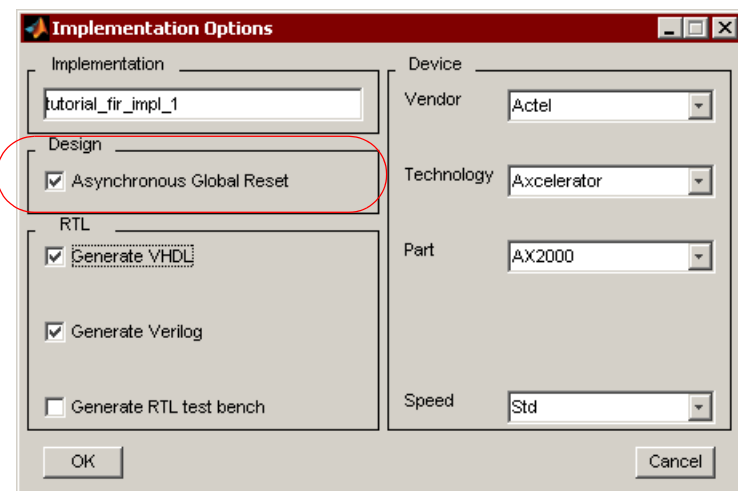
Sequential logic circuitry requires a reset signal to start and/or return to a known state. The following describe how the Synplify DSP tool handles resets in the design.

- [Global and Local Resets, on page 3-22](#)
- [Synchronous and Asynchronous Resets, on page 3-23](#)
- [Reset Implementation in RTL Code, on page 3-24](#)
- [Resets and RTL Testbenches, on page 3-25](#)

## Global and Local Resets

The circuitry produced by the Synplify DSP 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 Implementation Options dialog box.

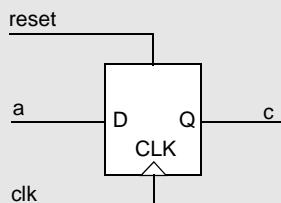




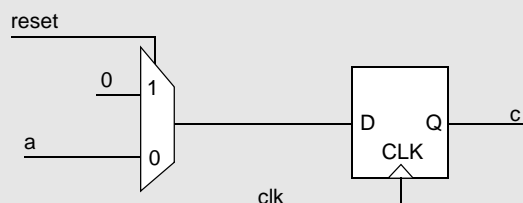
## Synchronous and Asynchronous Resets

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.

**Asynchronous Reset**



**Synchronous Reset**



The synchronous reset approach adds additional logic (a multiplexer) to the datapath compared to an asynchronous reset. However, it results in a simplified flip-flop which could be advantageous for an ASIC implementation with a vendor library that supports synchronous resets. 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 datapath. This leads to higher clock speeds and reduced consumption of the logic resources on the FPGA. For example, the Synplify DSP 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 Synplify DSP 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
```

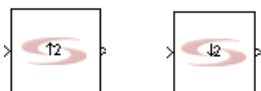
## Resets and RTL Testbenches

The testbenches generated by the Synplify DSP 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 Synplify DSP 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.

## Multi-Rate Design

The sample rate of a signal is determined by the sample rate propagated from input signals. Every block in the Synplify DSP 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 (Synplify DSP Signal Operations library). You can also use the Upsample and Downsample blocks in a multirate design that has inputs with different sample rates.



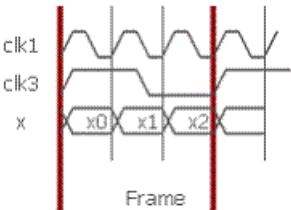
This section discusses the following:

- [Sample Rate Terminology, on page 3-26](#)
- [Clock Generation and Clock Reset, on page 3-29](#)
- [Polyphase Filtering, on page 3-32](#)

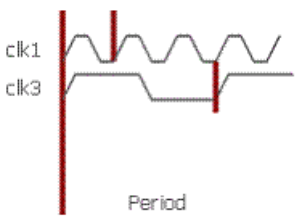
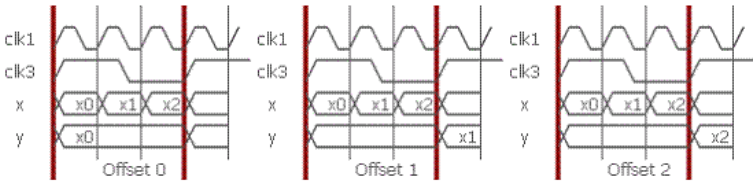
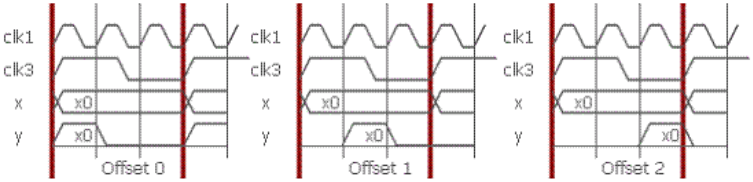
# Sample Rate Terminology

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
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:

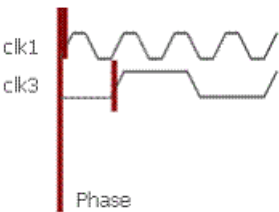


Interpolation	Increase of sample rate by inserting extra samples: <pre>function y = up(x, L, offset)     for o= 1:offset         y(o)=0;     end      for k = 1:length(x)         y((k-1)*L+1+offset) = x(k);         for ll=2:L-1             y((k-1)*L+ll+offset) = 0;         end     end end;</pre>
---------------	--

Term	Description
Latency	<p>Difference between the output frame number and input frame number of a function:</p> <ul style="list-style-type: none"> <li>For any offset in an upsample rate change, the latency is 0.</li> <li>For an offset of 0 in a downsample rate change, the latency is 0.</li> <li>For any other offset in a downsample rate change, the latency is 1.</li> </ul>
Multirate Design	<p>Design that uses multiple sample rates</p>  <p>Simulink can introduce clocks with different periods in these ways:</p> <ul style="list-style-type: none"> <li>Source: sample period definition</li> <li>Data type conversion: samples a continuous signal</li> <li>Upsample/Downsample/Resample blocks</li> </ul>
Offset (downsample rate change)	<p>Slot required to populate the frame at the lower rate.</p> 
Offset (upsample rate change)	<p>Slot to be populated in the frame at the higher rate.</p> 

Term	Description
------	-------------

Phase	Delay of the rising edge of a clock, relative to time zero
-------	--



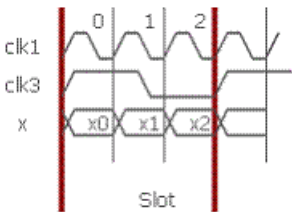
Simulink supports sample phase for single rate systems, but does not support this when applying rate changes. The Synplify DSP tool does not support sample phase for any signal; it assumes all clocks have phase zero.

Resampling	Combination of decimation and interpolation to change the sample rate with a fractional value.
------------	--

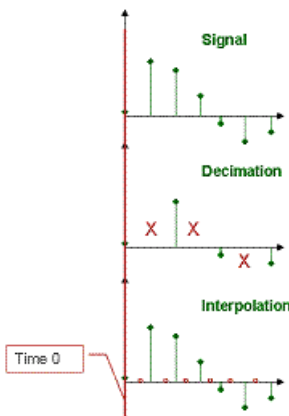
Sample Period	Period of the sample clock
---------------	----------------------------

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.

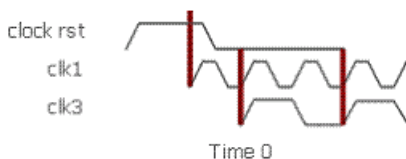


## Clock Generation and Clock Reset

The reset (main reset) provides, by definition, a synchronization point for the complete design. The Synplify DSP tool needs a Time 0 point. The relative location of this point to the clock reset is important because it

- Determines the functionality of simple single-rate functions
- Determines the shared samples of multi-rate functions.

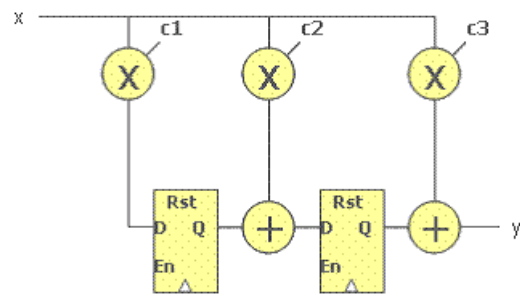
Most clock reset circuitry aligns the rising edges of derived clocks with the rising edge of the highest clock right after clock reset (or generates a corresponding enable at that location).



### Single-Rate Analysis

For a single-rate system, there is no clock reset circuitry, and no dedicated clock reset. The main reset pulse just indicates the position of the first active edge of the clock.

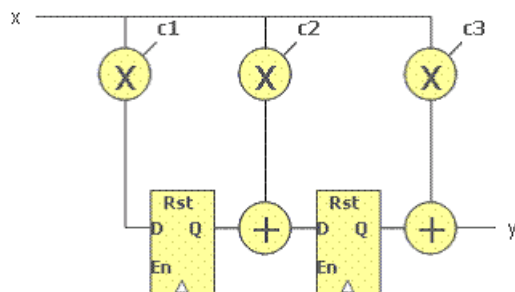
To look at the impact of a single-rate function, consider a 3-tap FIR; The input  $x[0]$  is provided at the first rising edge of a clock after a main reset-pulse. The desired (DSP mathematics) functionality is captured in the following table:



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 DFF's would also update, and destroy the initialization value. This is illustrated in the next table:



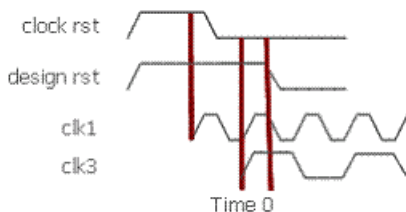
**Time**

-1	U	$C1*U$	0	$C2*U$	0	$C3*U$
0	$X[0]$	$C1*X[0]$	$C1*U$	$C2*X[0] + C1*U$	$C2*U$	$C3*X[0] + C2*U$

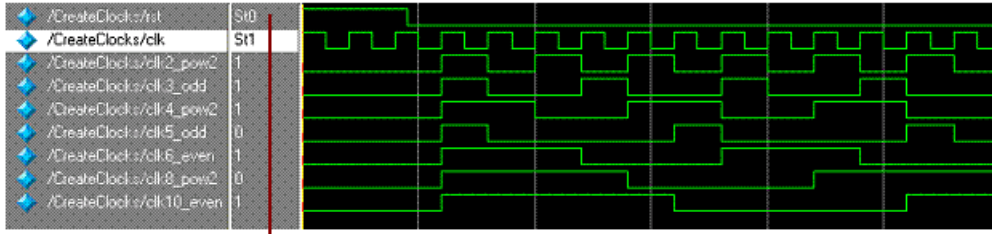
For single rate designs, it is essential that the design reset makes sure that the registers do no update at time zero. This means that the design reset needs to be different from the main reset. The design reset needs to be delayed by one clock cycle.

**Multi-Rate Analysis**

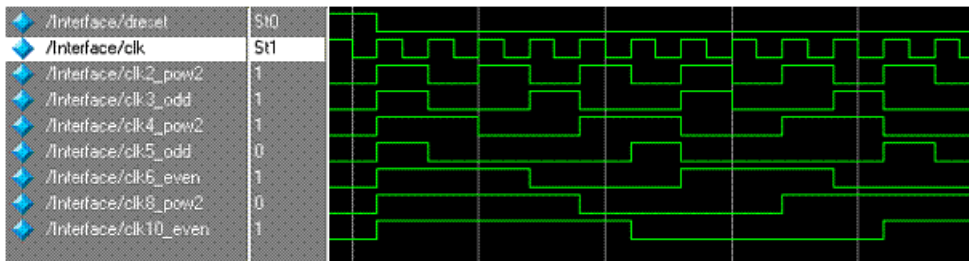
For a multi-rate system, assume that the clock reset circuitry generates the rising edges of all clocks aligned with the first rising edge of the fastest clock right after clock reset (Time 0). For any clock domain, the first rising edge of the respective domain can not trigger an update of the design registers (see [Single-Rate Analysis, on page 3-30](#)). 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:



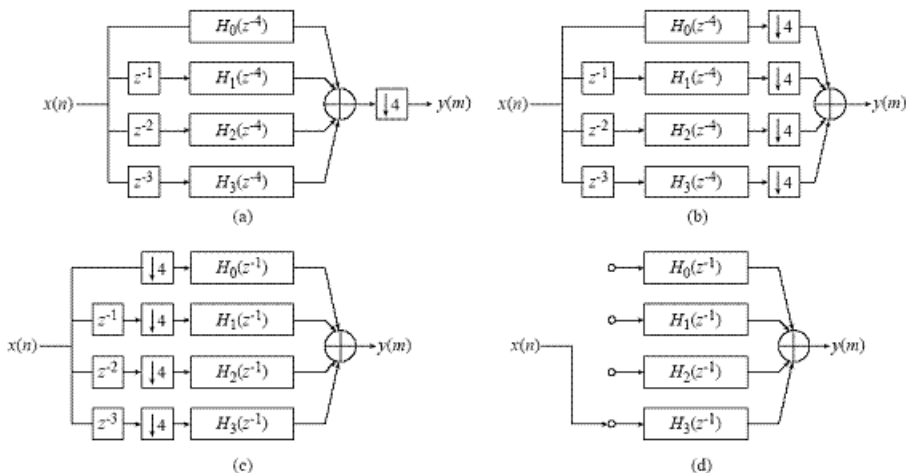
The combination of a design reset and the clocks shows a typical input for a Synplify DSP system:



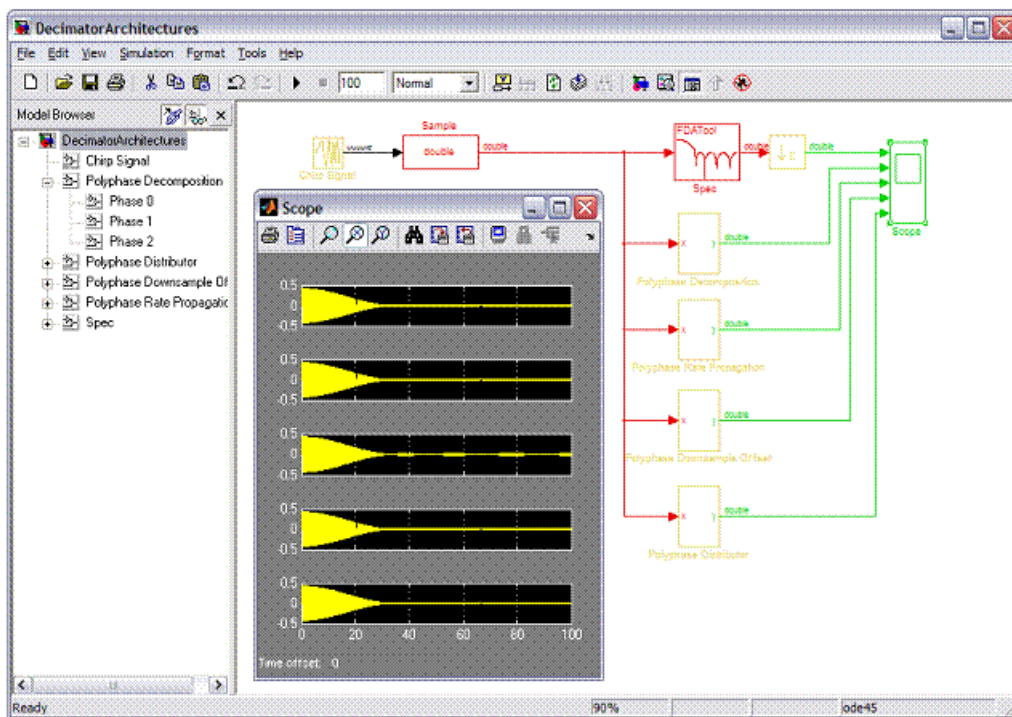
## Polyphase Filtering

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 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:



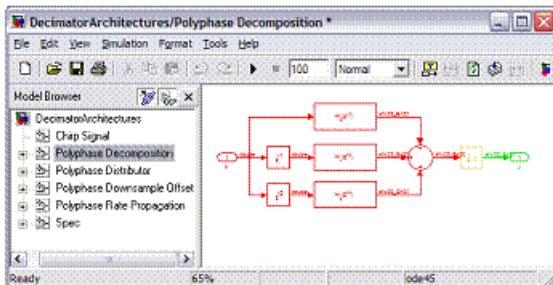
You can validate this in Simulink, using Simulink primitives.



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:

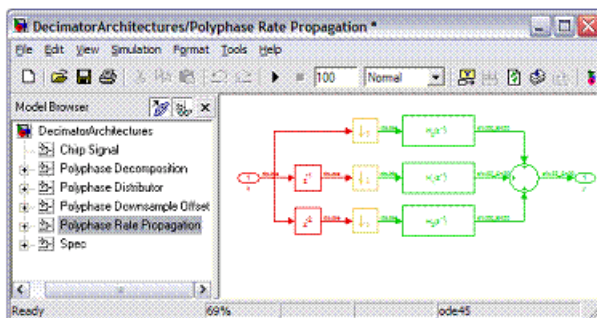
- Polyphase Decomposition

The first transformation re-organizes the FIR transfer function. It creates FIR structures with  $z^{-3}$  delays.



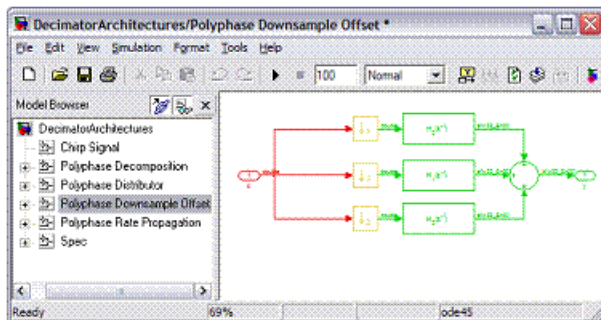
- Polyphase Rate Propagation

The second transformation moves the Downsample block across functions and delays. This turns the  $z^{-3}$  delays into  $z^{-1}$  delays.

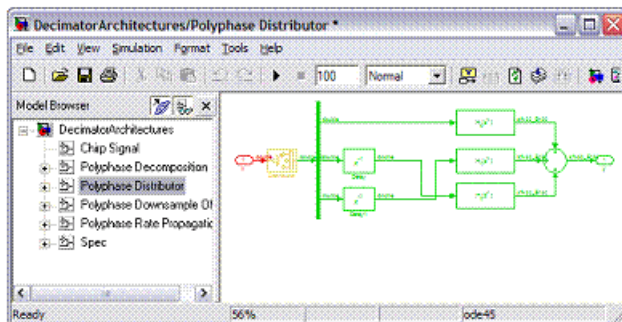


- Polyphase Downsample Offset

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.





## CHAPTER 4

# Using Synplify DSP for DSP Design

---

The following topics describe how to use some Synplify DSP features, so that you can orient yourself and get started. For an overview of the design flow, see [Synplify DSP Design Flows](#), on page 1-5.

The following describe how to work with different features of the tool:

- [Configuring Synplify DSP for Optimal Use](#), on page 4-2
- [Basic Procedures](#), on page 4-4
- [Working with the Output for ASIC Designs](#), on page 4-7
- [Designing Filters](#), on page 4-12
- [Working with Vectors](#), on page 4-21
- [Using Black Boxes and Third-Party IP](#), on page 4-24
- [Using Smart RTL Black Boxes](#), on page 4-33
- [Using Quantization Analysis Tools](#), on page 4-38
- [Specifying ROM Data with syn\\_read\\_hex](#), on page 4-43
- [Managing Subsystems and Hierarchy](#), on page 4-44
- [Running DSP Synthesis with SynDSPTool](#), on page 4-48
- [Working with Synplify DSP Output](#), on page 4-62

# Configuring Synplify DSP for Optimal Use

The following describe the best configuration settings that let you use the Synplify DSP tool most effectively:

- [Configuring Settings for Simulink Simulation, on page 4-2](#)
- [Setting Default Display Modes, on page 4-3](#)

The configuration for timing modes is described in [Configuring Synplify DSP Timing Modes for FPGAs, on page 4-51](#).

## 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 Synplify DSP blockset.

5. When you start a new design, make sure that you have the best Simulink settings for discrete-time DSP design. For the optimal configuration, do the following:
  - Type `syn_set_dspstartup` at the MATLAB command line. This function automatically tunes the settings for the model.
6. To view the current settings, do one of the following:
  - Type `syn_get_dspstartup` at the MATLAB command line. You see a list of the current settings. The following table lists the default settings.

Setting	Value
Solver	FixedStepDiscrete
SolverMode	SingleTasking
FixedStep	Auto
SaveTime	Off
SaveOutput	Off



Setting	Value
AlgebraicLoopMsgError	
InvariantConstants	On
SignalLogging	Off

- Select Simulation->Configuration Parameters from the model window. This opens a dialog box where you can view the current settings.

## Setting Default Display Modes

When you work with Synplify DSP 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

Synplify DSP runs under the MATLAB and Simulink interface, and most of what is described here should be familiar to all MATLAB users.

- [Starting a Synplify DSP Design, on page 4-4](#)
- [Working with Synplify DSP Blocks, on page 4-5](#)

For an overview of the design flow, see [Synplify DSP Design Flows, on page 1-5](#).

## Starting a Synplify DSP Design

The following describes how to start up Synplify DSP 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.

<b>For a new design....</b>	Select File->New->Model or click the icon. An empty model window opens.
<b>For an existing design...</b>	Select File->Open and specify the model you want to open. The model window opens with your design.

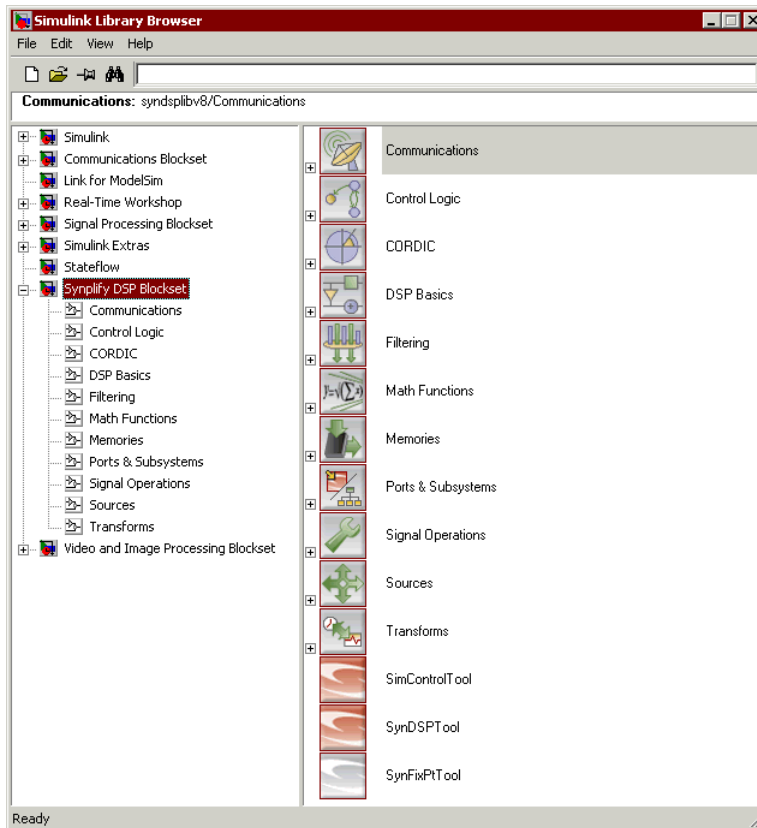
You can now add blocks to your design as described in [Working with Synplify DSP Blocks, on page 4-5](#).

## Working with Synplify DSP Blocks

This basic procedure shows you how to work with Synplify DSP blocks in your design.

1. From the Simulink library browser, double-click Synplify DSP 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 Synplicity blockset. A Synplify DSP design must be bounded by these blocks.
- Build your circuit using the Synplicity blockset. The software only generates RTL for Synplicity 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 SynDSPTool toolbox so that you can run DSP synthesis and generate RTL for the design. For information about using this toolbox, see [Running DSP Synthesis with SynDSPTool, on page 4-48](#).

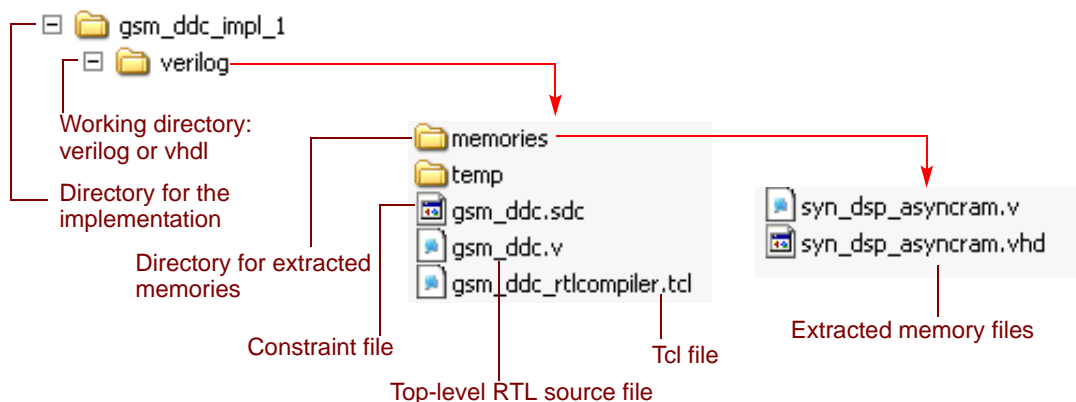
# Working with the Output for ASIC Designs

If you target an ASIC implementation, the Synplify DSP tool generates output files that are targeted and optimized for the choice you made. This following describe the ASIC-specific output:

- [Output Files for ASIC Designs, on page 4-7](#)
- [Running ASIC Logic Synthesis, on page 4-9](#)
- [Working with ASIC Output Tcl Files, on page 4-10](#)

## Output Files for ASIC Designs

After optimization and DSP synthesis, the Synplify DSP tool generates a number of files. The following figure shows the files that the software generates, and the associated directory structure.



The tool creates a directory for the implementation `<design_name>_impl<num>`. Under it is the working directory, called `verilog` or `vhdl`, depending on which format you selected for generating RTL code. The working directory contains the output files, and includes another directory called `memories`, which contains the RTL for any extracted memories.

The following table describes the output files:

<code>&lt;design&gt;.sdc</code> Constraint file	Located in the working directory. This file contains design constraints that can be forward-annotated to the ASIC P&R tool. The file is in the ASIC industry-standard .sdc format.
<code>&lt;design&gt;_rtlcompiler.tcl</code> Do file	A do file that guides ASIC logic synthesis. See <a href="#">Working with ASIC Output Tcl Files, on page 4-10</a> for information about using it.
<code>&lt;design&gt;.v</code> or <code>&lt;design&gt;.vhd</code> Top-level RTL File	A top-level RTL file for the design. It includes RTL code (.v or .vhd) for the Synplify DSP block components in the design.
<code>memories/&lt;memory&gt;.v</code> or <code>&lt;memory&gt;.vhd</code> Memory files	RTL files for extracted memories. See <a href="#">Running ASIC Logic Synthesis, on page 4-9</a> for information about working with these files.

## Running ASIC Logic Synthesis

The following procedure shows you how to use the Synplify DSP output files effectively and run logic synthesis in the ASIC tool. For descriptions of the Synplify DSP output files, see [Output Files for ASIC Designs, on page 4-7](#).

1. Prepare the memories.

You can handle the memories the Synplify DSP tool extracted in any of the following ways:

- Use a vendor compiler to create IP and replace the Synplify DSP memory with that memory IP.
- Include memory models in your ASIC logic synthesis design, so the tool can synthesize the registers.
- Leave the Synplify DSP memory as a black box to be inserted at the gate level.

2. Open the ASIC synthesis tool and read in the Synplify DSP output files. You can either use a script to run synthesis, or read in the RTL and constraint files:

- To run Cadence® Encounter® RTL Compiler logic synthesis from a script, use the Synplify DSP <design>\_rtlcompiler.tcl file. See [Working with ASIC Output Tcl Files, on page 4-10](#).
- To run Synopsys® Design Compiler® logic synthesis from a script, generate a do file, as described in [Working with ASIC Output Tcl Files, on page 4-10](#).
- To run logic synthesis without a script, use the following Synplify DSP files as input for synthesis: constraint file (.sdc), top-level RTL (.v or .vhd), and extracted memory files.

3. Synthesize the design using retiming.

The Synplify DSP tool approximates pipelined register locations, so using the logic synthesis tool can optimize placement. In multi-rate designs, apply retiming to all clocks.

## Working with ASIC Output Tcl Files

You can use the Tcl output from the Synplify DSP tool as input to run ASIC logic synthesis, as follows:

1. If you are using the Cadence® Encounter® RTL Compiler software, read in Synplify DSP <design>\_rtlcompiler.tcl file into the software.

The Synplify DSP tool automatically generates a Tcl script (do) file for ASIC logic synthesis with RTL Compiler.

2. If you are using Synopsys® Design Compiler® software, do the following:
  - Start with the Synplify DSP output Tcl file and create a file for the Design Compiler tool. The Synplify DSP tool does not automatically generate a file for use with Design Compiler, so you must create a file.
  - Use the following table, which shows how to map the generated commands to Design Compiler commands. All commands in the table map to commands in the Design Compiler Tcl file, except for certain indicated commands that are specified in the synopsys\_dc.setup file.

### Cadence RTL Compiler

### Synopsys Design Compiler Equivalents

Update the search path; Set libraries; Identify a temporary place for intermediate files:

	Commands that go in the synopsys_dc.setup file:
	set CURRENT_PATH "."
	set LIBRARY_PATH "path to library"
	set search_path [concat \$search_path \$CURRENT_PATH \$LIBRARY_PATH]
	Tcl file commands:
set_attribute library <lib_path>/ <slow   typical   fast>.lib	set target_library {fast   typical   slow}.db
	Optional command lines:
	set link_library { fast   typical   slow }.db
	set symbol_library <symbol_lib>.sdb
	define_design_lib WORK -path work

Read in Synplify DSP verilog or vhdl source files; Set top level design:



**Cadence RTL Compiler****Synopsys Design Compiler Equivalents**

read\_hdl - vhd1 fir.vhd  
 read\_hdl - verilog fir.v  
 elaborate fir

read\_vhdl fir.vhd  
 read\_verilog fir.v  
 current\_design fir

Optional command lines:

link  
 uniquify  
 check\_design > fir-check\_design.log

Read in Synplify DSP constraint file:

source fir.sdc

read\_sdc fir.sdc

Synthesize design:

synthesize -to\_mapped

compile

Optional compile command options:

-map\_effort medium -area\_effort medium

Report timing and area; Other reports:

report\_timing > timing.rpt  
 report\_area > area.rpt

report\_timing > ./timing.log  
 report\_area > ./area.log

Optional command lines:

report\_hierarchy > ./hierarchy.log  
 report\_resources > ./resources.log  
 report\_constraint > ./constraint.log

Write out gate-level netlist, Write out SDC file:

write -mapped fir > fir\_gate.v

write -hierarchy -format verilog -output fir\_gate.v  
 write -hierarchy -format vhd1 -output fir\_gate.vhd

Optional command lines:

write\_sdc fir-out.sdc

# Designing Filters

This section shows you how to design FIR and IIR filters with Synplify DSP. The Synplify DSP 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. Synplify DSP provides a function to automatically incorporate the coefficients generated by the FDATool.

The following provide more detail:

- [Implementing FIR Filters, on page 4-12](#)
- [Implementing Polyphase FIR Filters, on page 4-14](#)
- [Defining FIR Filter Coefficients with FDATool, on page 4-15](#)
- [Implementing IIR Filters, on page 4-17](#)
- [Defining IIR Filter Coefficients with FDATool, on page 4-19](#)

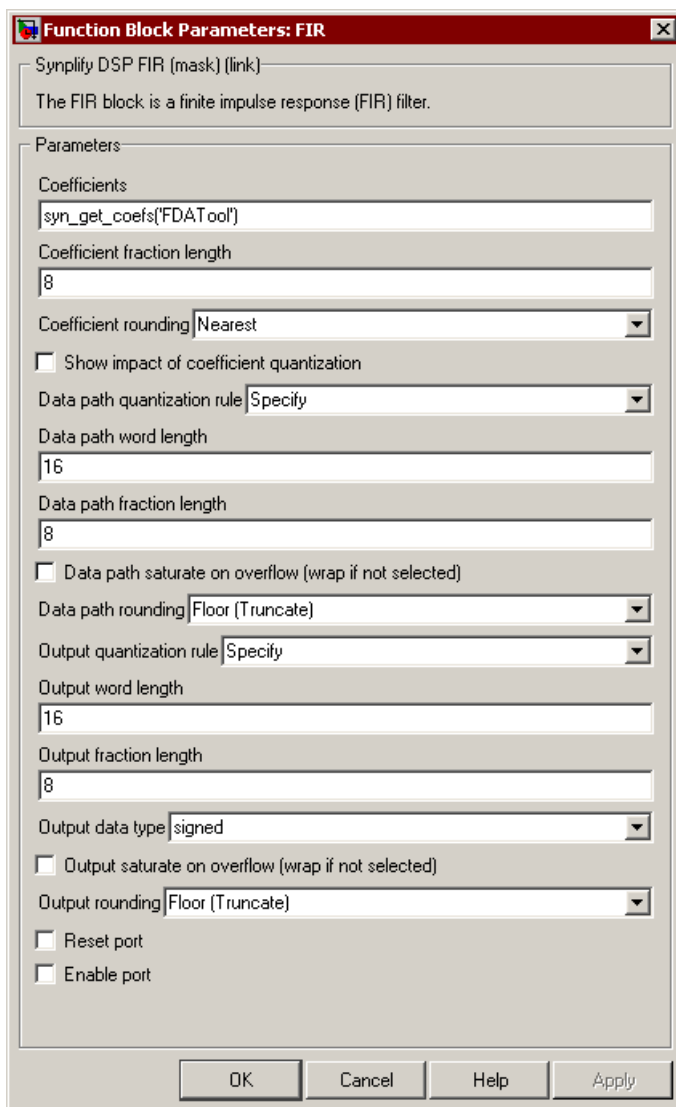
For some information about using adaptive filters, see the example in [Using Math Operations on Vector Signals, on page 4-22](#) and the LMS demo example.

## Implementing FIR Filters

This procedure shows you how to implement FIR filters. If you want a demo, refer to the tutorial.

1. Add the Synplify DSP FIR block to your design.
2. Define the filter coefficients in one of these ways:
  - Double-click the FIR block to open the block parameters dialog box, and use MATLAB vector variables to define the coefficients in the Coefficients field.
  - Add the Synplify DSP FDATool block and define the filter coefficients. See [Defining FIR Filter Coefficients with FDATool, on page 4-15](#), which describes this procedure in detail.
3. Set FIR block parameters.

- If you have not done so, double-click the FIR block to open the block parameters dialog box.



The image shows the 'Function Block Parameters: FIR' dialog box. It has a title bar with a red background and a close button. The main area is divided into sections. The first section is 'Synplify DSP FIR (mask) (link)' with a text box containing 'The FIR block is a finite impulse response (FIR) filter.' Below this is the 'Parameters' section. It contains several fields and checkboxes. The 'Coefficients' field is a text box with 'syn\_get\_coefs('FDATool')'. The 'Coefficient fraction length' is a text box with '8'. The 'Coefficient rounding' is a dropdown menu with 'Nearest'. There is a checkbox for 'Show impact of coefficient quantization'. The 'Data path quantization rule' is a dropdown menu with 'Specify'. The 'Data path word length' is a text box with '16'. The 'Data path fraction length' is a text box with '8'. There is a checkbox for 'Data path saturate on overflow (wrap if not selected)'. The 'Data path rounding' is a dropdown menu with 'Floor (Truncate)'. The 'Output quantization rule' is a dropdown menu with 'Specify'. The 'Output word length' is a text box with '16'. The 'Output fraction length' is a text box with '8'. The 'Output data type' is a dropdown menu with 'signed'. There is a checkbox for 'Output saturate on overflow (wrap if not selected)'. The 'Output rounding' is a dropdown menu with 'Floor (Truncate)'. There are checkboxes for 'Reset port' and 'Enable port'. At the bottom are buttons for 'OK', 'Cancel', 'Help', and 'Apply'.

Function Block Parameters: FIR

Synplify DSP 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)

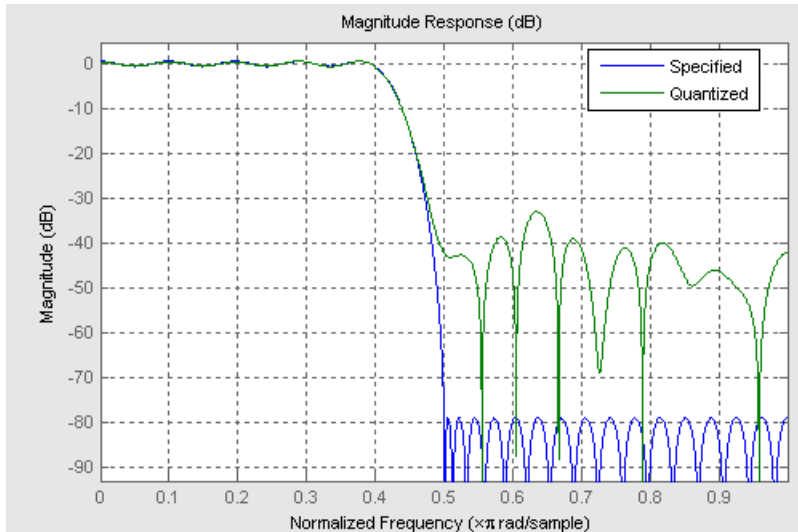
☐ Reset port

☐ Enable port

OK Cancel Help Apply

- 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.

1. Add the Synplify DSP 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.

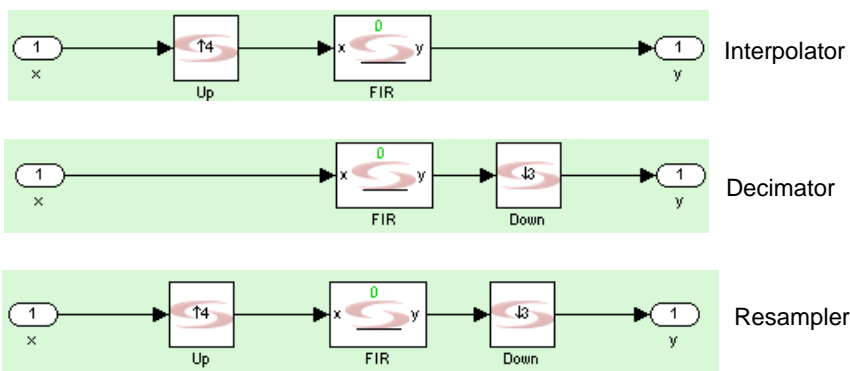
For details of the block parameters, see [FIR Rate Converter Parameters, on page 8-134](#).

3. Set the other options:

- Set upsample and/or downsample rates.
- Set the coefficients. See [Defining FIR Filter Coefficients with FDATool, on page 4-15](#).
- Set the data path and output formats.

4. Click OK.

The software implements a polyphase FIR filter according to the criteria you specified.

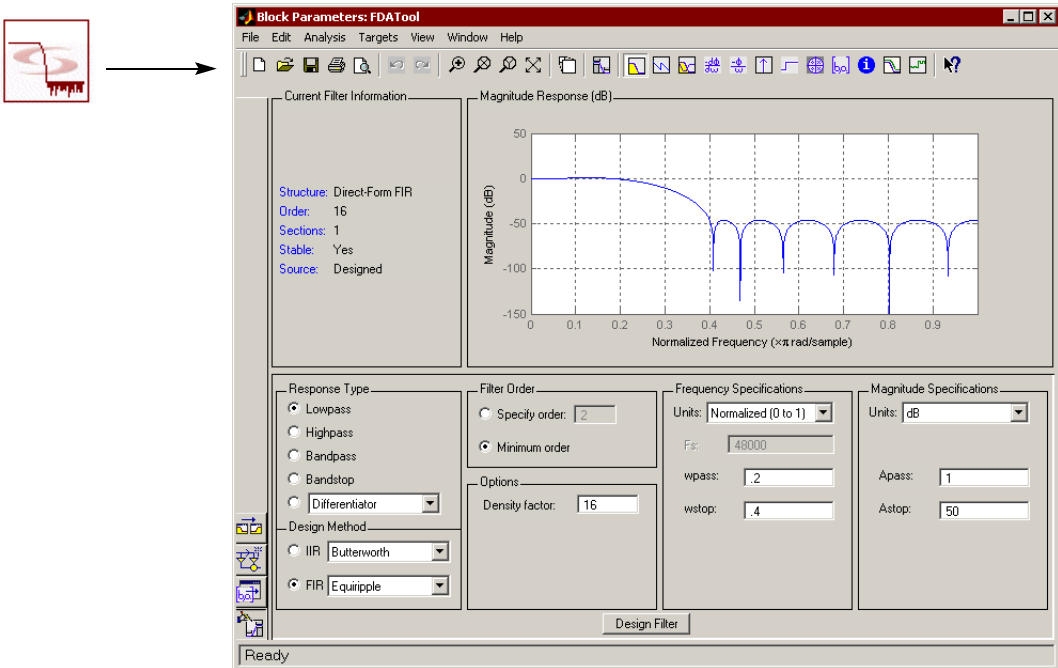


## Defining FIR Filter Coefficients with FDATool

The following describes how to define coefficients for a Synplify DSP FIR filter.

1. Add an FIR filter block from the Synplify DSP blockset to your design.
2. Add the Synplify DSP 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 9-4](#). 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.

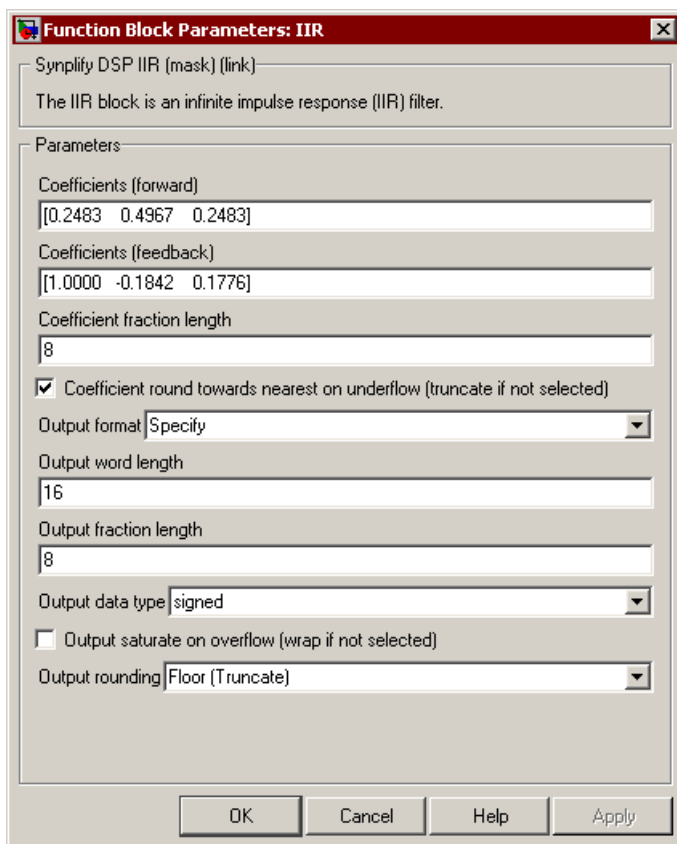
This updates the filter block with the coefficients you defined in the FDATool window. The tool updates the block icon in the Simulink schematic window to reflect the new coefficients.

## Implementing IIR Filters

This procedure shows you how to implement IIR filters.

1. Add the Synplify DSP 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 Synplify DSP FDATool block and define the forward and feedback coefficients. See [Defining IIR Filter Coefficients with FDATool, on page 4-19](#), 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 image shows a dialog box titled "Function Block Parameters: IIR". It contains a description of the IIR block and a section for parameters. The parameters section includes fields for forward and feedback coefficients, coefficient fraction length, output format, output word length, output fraction length, output data type, and output rounding. There are also checkboxes for coefficient rounding and output saturation.

**Function Block Parameters: IIR**

Synplify DSP IIR (mask) (link)

The IIR block is an infinite impulse response (IIR) filter.

**Parameters**

Coefficients (forward)  
[0.2483 0.4967 0.2483]

Coefficients (feedback)  
[1.0000 -0.1842 0.1776]

Coefficient fraction length  
8

☒ Coefficient round towards nearest on underflow (truncate if not selected)

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)

OK Cancel Help Apply

The software implements an IIR filter according to the criteria you specified.



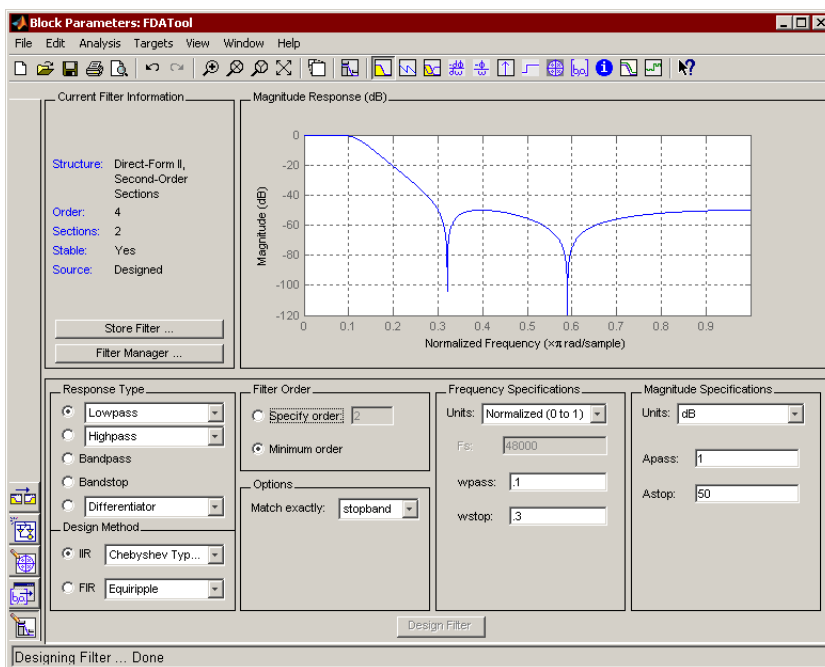
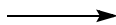
## Defining IIR Filter Coefficients with FDATool

The following describes how to define coefficients for a Synplify DSP IIR filter.

1. Add an IIR filter block from the Synplify DSP blockset to your design.
2. Add the Synplify DSP FDATool block. Double-click this block in the Simulink window.

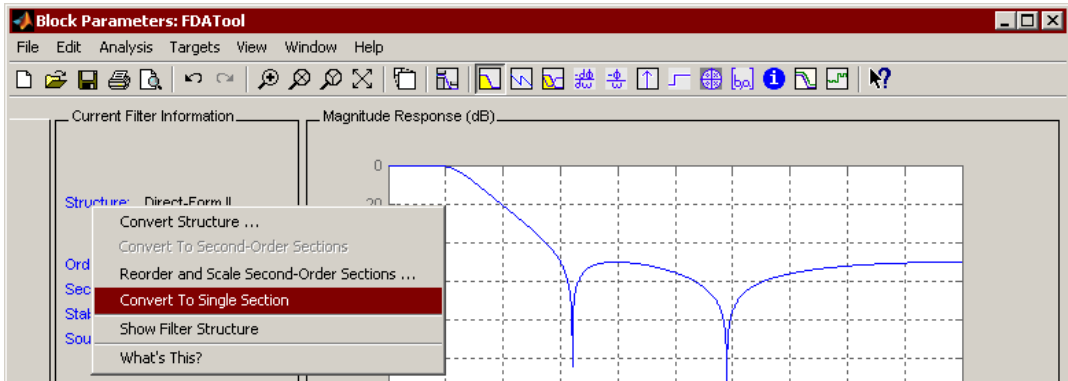
A window opens with the MathWorks Filter Design and Analysis tool.

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('<instances>', 'forward')` and `syn_get_coefs('<instances>', 'feedback')`. If you do not specify an instance name, the function searches for an instance called FDATool. See [syn\\_get\\_coefs](#), on page 9-4 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 Synplify DSP blocks accept vector signal inputs and adjust the operation to process the vector elements. For a quick summary of vector support on a per-block basis, see [Blockset Summary, on page A-1](#).

This section describes the following:

- [Creating Vector Signals, on page 4-21](#)
- [Using Math Operations on Vector Signals, on page 4-22](#)

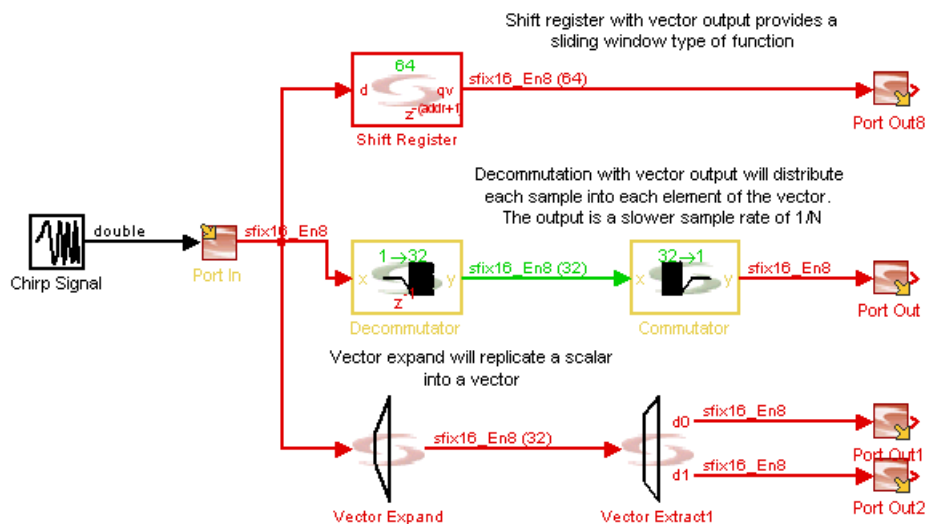
## Creating Vector Signals

1. To bring in a vector signal from Simulink, use the Synplify DSP 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 Synplify DSP Decommulator and Shift Register blocks.
3. To merge or manipulate vectors, use the following blocks.

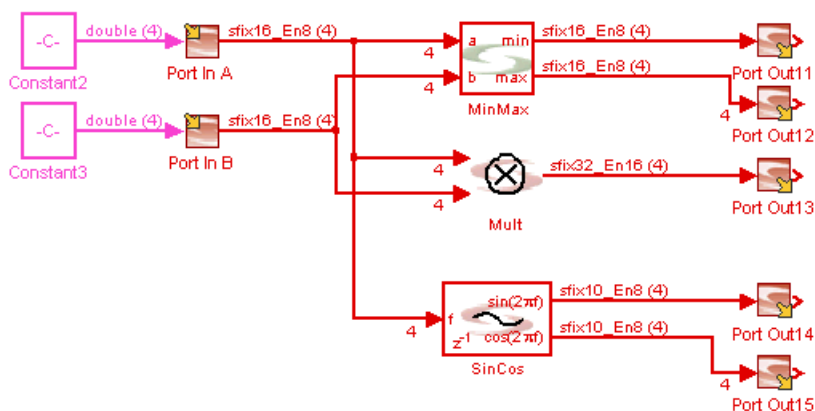
To...	Use...
Replicate a scalar input and create vector output	Vector Expand
Concatenate vector and scalar inputs to a single vector	Vector Concat
Generate scalar data from vector input	Vector Extract
Split vector input	Vector Split



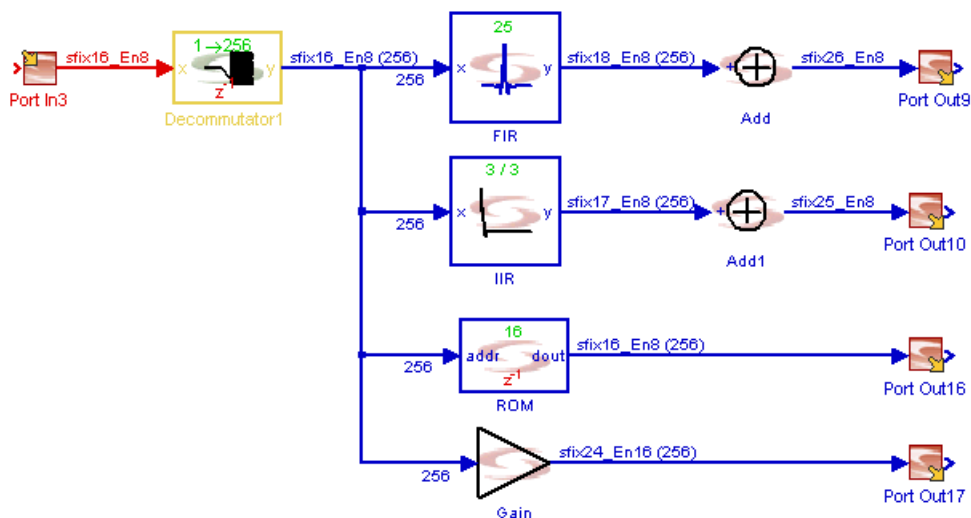
## Using Math Operations on Vector Signals

Many blocks from the Synplify DSP Math Functions library support vectors. For details, see [Blockset Summary, on page A-1](#).

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.



# Using Black Boxes and Third-Party IP

A black box is a specialized Synplify DSP block that lets you incorporate foreign IP into your Synplify DSP design. Synplify DSP 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 RTL Black Boxes, on page 4-33](#).

For details about the Black Box block, see [Synplify DSP Black Box, on page 8-25](#). For an example, see `<install_dir>\mathworks\toolbox\Synplicity\demos\examples`. This section discusses the following:

- [Integrating Black Boxes in the Design, on page 4-24](#)
- [Using Optimizations with Black Boxes, on page 4-27](#)
- [Setting Black Box Parameters, on page 4-28](#)
- [Configuring a Black Box - Example, on page 4-30](#)

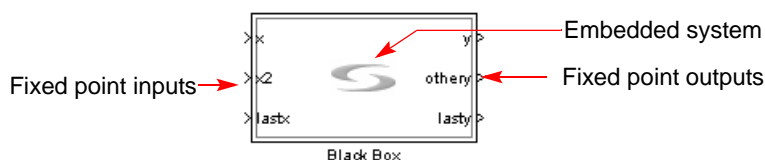
## Integrating Black Boxes in the Design

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 Synplify DSP 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 Synplify DSP tool.

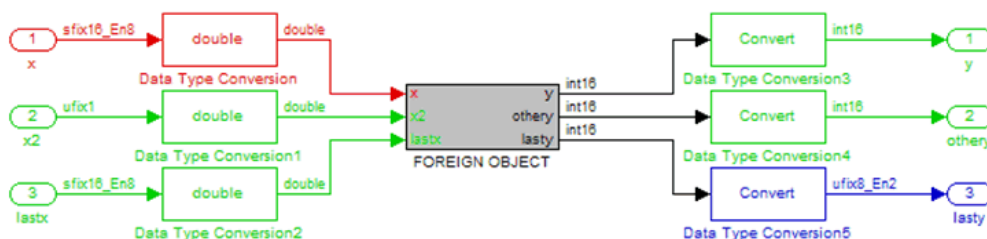
To incorporate a black box in your design, use the following procedure:

1. Select the Black Box block from the Synplify DSP Ports & Subsystems library, and add it to your design.

This block provides the interface to the embedded block.



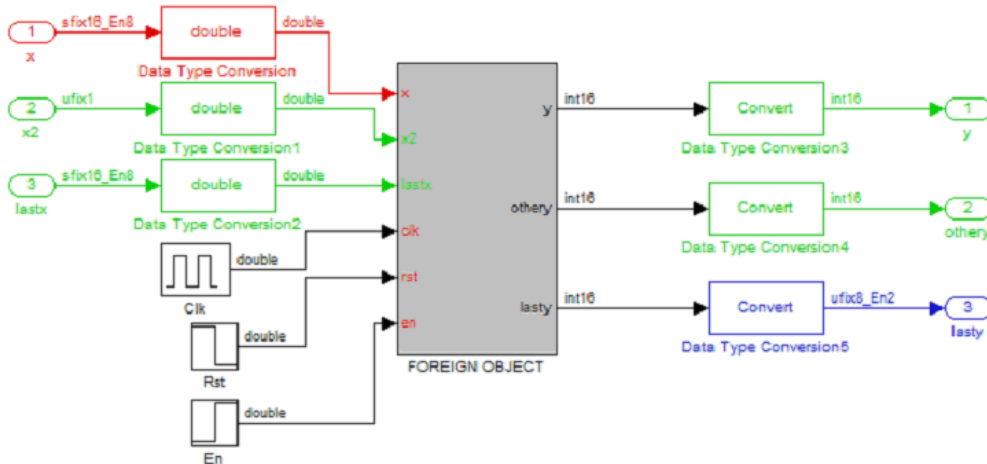
2. Double-click the black box, select Mask Parameters, and configure the black box to match your black box parameters. For an example, see [Configuring a Black Box - Example, on page 4-30](#).
3. Complete the internal black box design.
  - Right-click the black box and select Look under mask.
  - In 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 4-30](#) 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 Synplify DSP blocks.



- The block must have a discrete sample time.
- The embedded object must connect to each black box output port through a Synplify DSP 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 Synplify DSP 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 Link to ModelSim or any other RTL simulator interface. For information about cosimulation, see [Using Smart RTL Black Boxes, on page 4-33](#).
- 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 Synplify DSP synthesis.



- Set the Synplify DSP optimizations you want for the design. For details, see [Using Optimizations with Black Boxes, on page 4-27](#).
- 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.

## 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 registers in the design after and/or before the black box.

## Requirements for Multi-channelizing Black Boxes

Synplify DSP can multi-channelize a design containing a black box. The tool instantiates a black box for each channel.

## 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"> <li>• Set Black Box Definition to Single HDL File.</li> <li>• In HDL File, specify the absolute path to the Verilog/VHDL definition file.</li> <li>• Specify the name of the top level entity in Entity/Model Name.</li> </ul>
Single EDIF file, as with soft cores purchased from a third party	<ul style="list-style-type: none"> <li>• Set Black Box Definition to Single EDIF File.</li> <li>• Specify the absolute path to the EDIF definition file in EDIF File.</li> <li>• In Simulation File, specify the absolute path to the simulation behavioral model file (Verilog or VHDL).</li> <li>• Specify the name of the top level entity in Entity/Model Name.</li> </ul>

Black box defined in...	Specify these options...
Multiple Verilog, VHDL, or EDIF files	<ul style="list-style-type: none"> <li>• Create a text file that lists the absolute paths to each Verilog, VHDL, or EDIF definition file and behavioral simulation file.</li> <li>• Set Black Box Definition to Import File List.</li> <li>• In Black Box File List, specify the absolute path to the text file you created.</li> <li>• Specify the name of the top level entity in Entity/Model Name.</li> </ul>
Another black box block, or one for which you have no definition	<ul style="list-style-type: none"> <li>• Set Black Box Definition to Undefined.</li> <li>• Specify the name of the top level entity in Entity/Model Name.</li> </ul>

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 Synplify DSP design. It is only specified in the RTL generated after Synplify DSP 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 Synplify DSP 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 Synplify DSP design. They are only specified in the RTL generated after Synplify DSP 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.

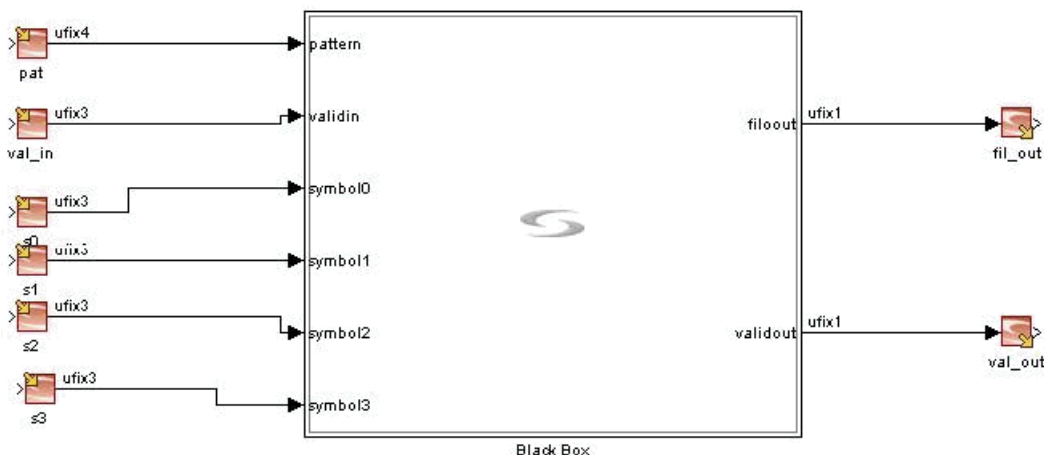
If you do not enable Format Clock, the black box uses the default naming convention for the clocks, with the fastest clock being `clk`, and N-reduced frequency clocks called `clkDivN`.

## Configuring a Black Box - Example

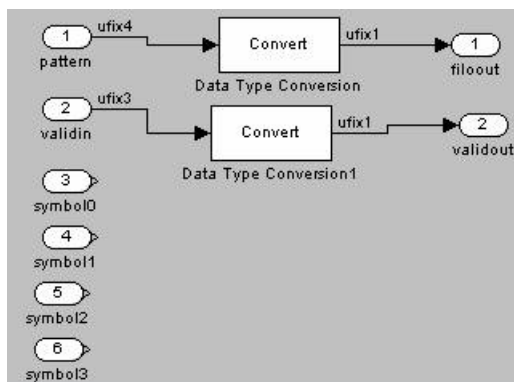
The following procedure illustrates how to configure a black box, using a Viterbi decoder as an example. For this example, the decoder is defined in a Verilog file, `C:\myblackboxes\viterbidecoder.v`.

1. Add the Synplify DSP 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;
```



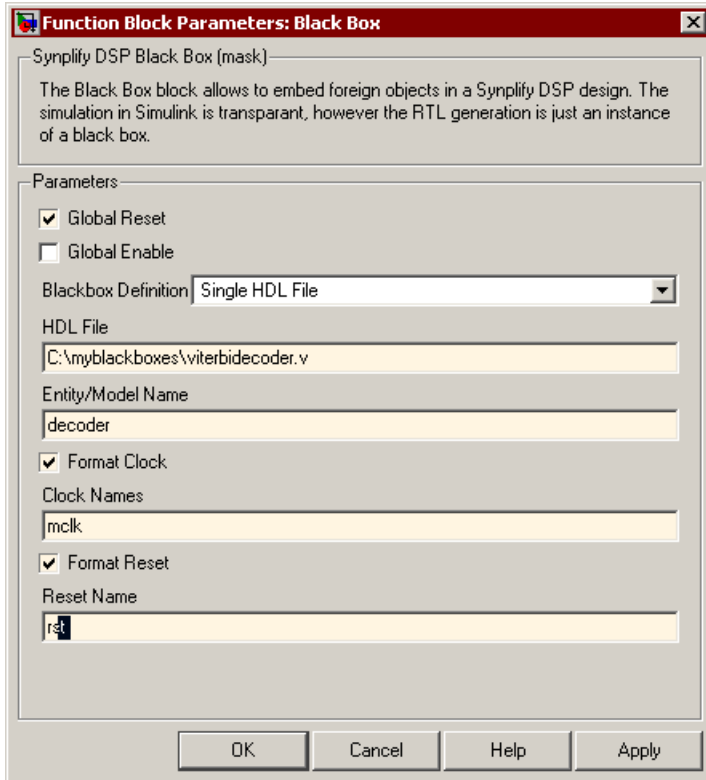
2. Specify the internal design by right-clicking the black box block and selecting Look under mask. For this example, add Convert blocks to adjust the output data type and convert ufix4 and ufix3 to ufix1.



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 4-28](#). 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 RTL Black Boxes](#), on [page 4-33](#).

# Using Smart RTL Black Boxes

A smart black box differs from a simple black box, because you have access to the RTL code for the IP. This section describes the implementation of a smart RTL black box using the Smart Black Box block. For information about implementing a simple black box, see [Using Black Boxes and Third-Party IP, on page 4-24](#).

This section describes the following:

- [Incorporating Smart Black Boxes in the Design, on page 4-33](#)
- [Configuring the Cosimulation Interface, on page 4-35](#)
- [Creating Smart Black Box Configuration Files, on page 4-37](#)

## Incorporating Smart Black Boxes in the Design

1. Make sure you have Link for ModelSim® installed and accessible.

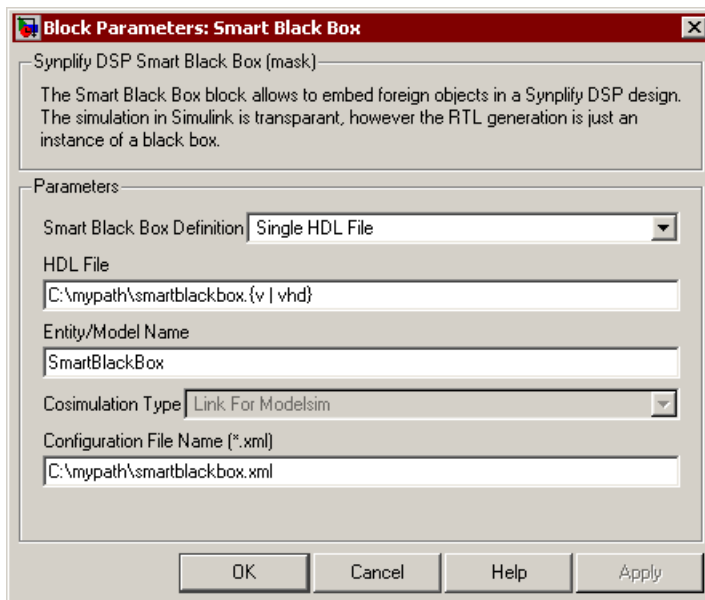
Link for ModelSim is a cosimulation interface between Simulink and the ModelSim® HDL simulator. Currently, Synplify DSP uses only this tool to verify and simulate the embedded RTL-level models.

2. Instantiate the SynCoSimTool block (top-level library) in your design and configure the cosimulation interface. See [Configuring the Cosimulation Interface, on page 4-35](#) 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 4-37](#) 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 `sbblib.txt` that lists four black box definition files:

```
-L sbblib C:\mypath\sbb1.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 [Synplify DSP Smart Black Box, on page 8-238](#) 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 Synplify DSP synthesis.
  - Set the Synplify DSP 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 4-27](#).
  - Synthesize the design and generate RTL.

The tool uses Link for ModelSim to cosimulate and verify the embedded RTL-level models. The Simulink simulation is transparent.



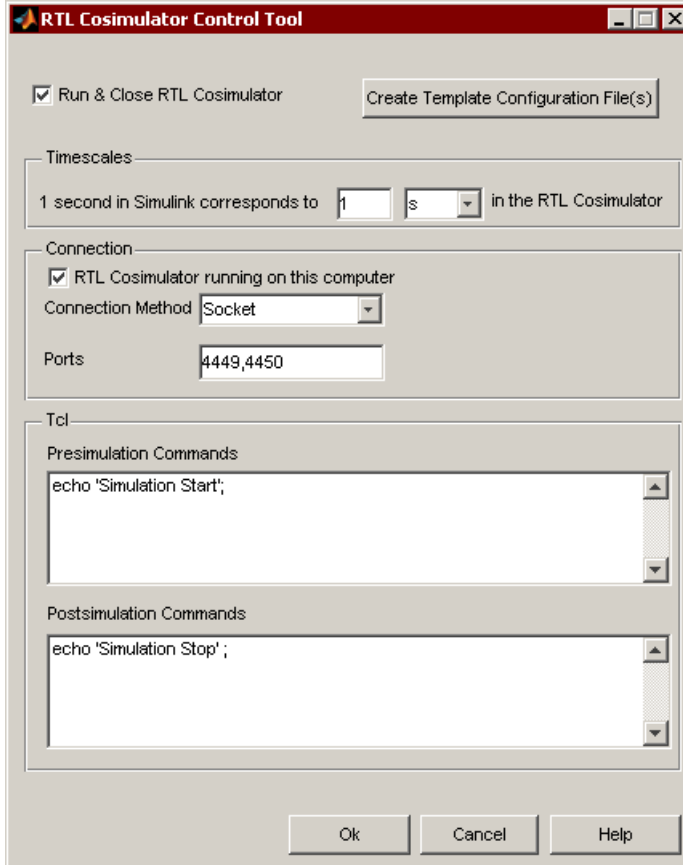
The generated RTL for the design includes an instance for the smart black box. The rest of the design is hooked up to the ports of the black box, with the appropriate connections for global enables, reset, and black box clocks you specified. Timing arcs stop at the input ports of the black box and resume 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 Link for ModelSim 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 4-33](#).

1. Instantiate the SynCoSimTool block from the top-level Synplify DSP library, and double-click it to open the parameters dialog box.

This dialog box lets you configure the cosimulation interface to Link for ModelSim. Do the next few steps in the dialog box. For detailed description of the dialog box options, refer to [Synplify DSP SynCoSimTool, on page 8-249](#).



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 Link for ModelSim 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 4-33](#).

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 8-241](#).
- Create and then edit a template file.
  - Instantiate and configure the SynCoSimTool block as described in [Configuring the Cosimulation Interface, on page 4-35](#).
  - 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 8-241](#) for syntax.

# Using Quantization Analysis Tools

The Synplify DSP tool uses the Simulink fixed-point data type to represent the discrete amplitude of the signals in the Synplify DSP blockset. For background information about this data type in the Synplify DSP tool, see [Fixed-Point Data Type, on page 3-20](#). This section describes how to use the Synplify DSP SynFixPtTool block and the Simulink Fixed-Point Tool interface:

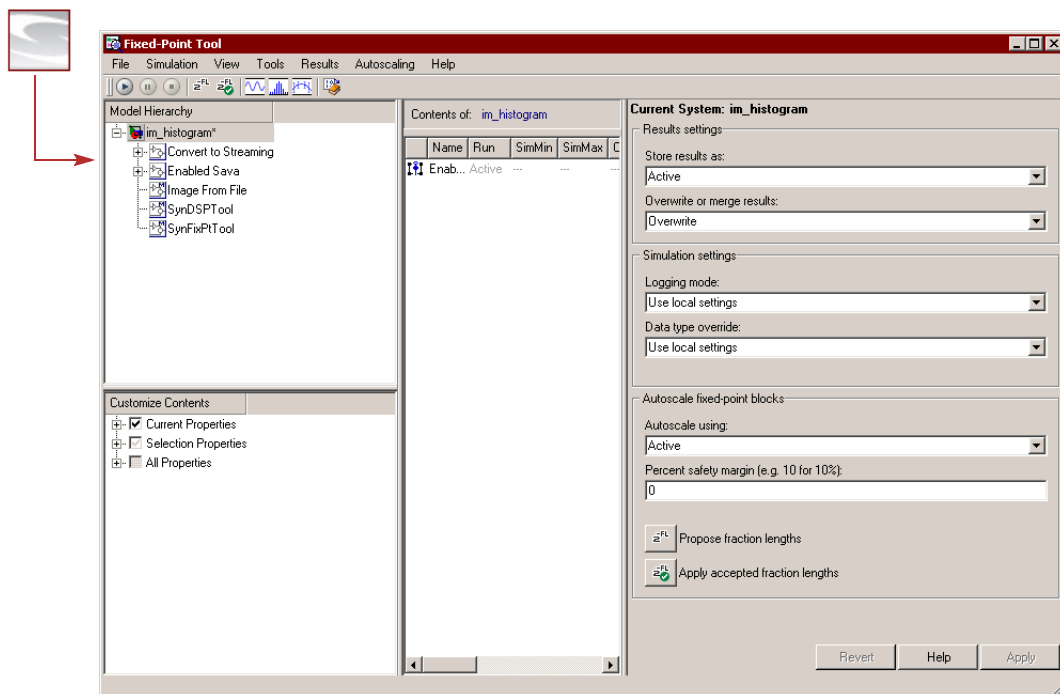
- [Specifying Fixed-Point Options, on page 4-38](#)
- [Validating Algorithms with the Fixed-Point Toolbox, on page 4-40](#)
- [Using Plots, on page 4-42](#)

## Specifying Fixed-Point Options

To use the fixed-point functionality for conversion before simulation, do the following:

1. Add the Synplify DSP SynFixPtTool block to your design.
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. To walk through a design that uses this interface with an FIR filter, see the tutorial.

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 4-40](#) and [Using Plots, on page 4-42](#).

You can use all the toolbox features with the following exceptions, which the Synplify DSP tool does not currently support:

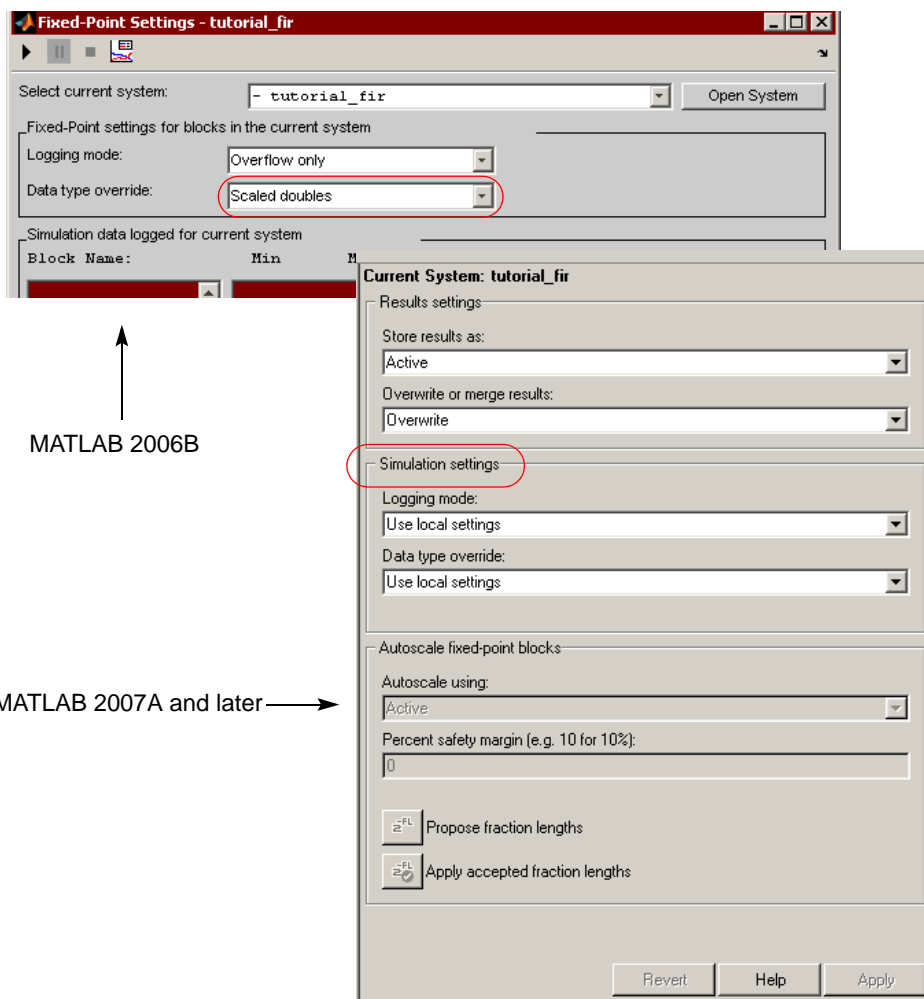
- For the Logging mode option, you can only use Use local settings and Overflow Only (MATLAB 2006B only). This option controls logging for the selected subsystems.
- For the Data type override option, the only valid choices are Use local settings and Scaled Doubles. This option controls data type overrides 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 following outlines the general procedure:

1. Set up your design.
  - Add the Synplify DSP SynFxFPtTool 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 4-42](#).
2. Double-click SynFxFPtTool 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 (MATLAB 2006B only). Currently, the Synplify DSP software does not support the Minimums, maximums, and overflows setting.

- Set Data type override to Scaled Doubles in the Simulation settings area of the settings pane. This mode is not supported in MATLAB 2007A or 2007B. Also, the interface is different, starting with MATLAB 2007A. The following figure shows the relevant portions of the window in different versions of the MATLAB software:



- 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 (MATLAB 2006B only). Currently, the Synplify DSP software does not support the Minimums, maximums, and overflows setting.
  - 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 4-42](#).

## 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 4-40](#)).



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.

## 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 Synplify DSP 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 9-11 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.

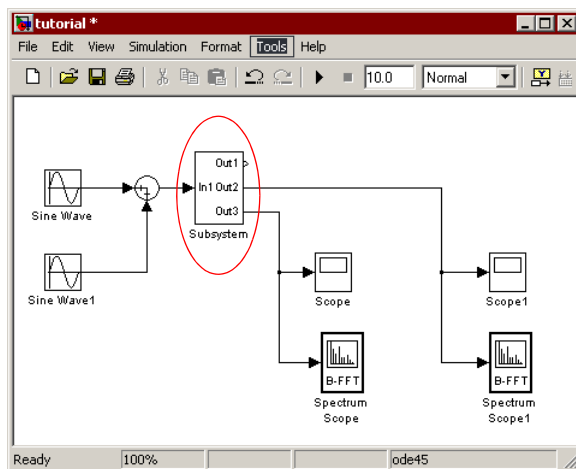
## Managing Subsystems and Hierarchy

You can use subsystems to manage data type settings for a bunch of blocks collectively. The following procedure uses subsystems to manage fixed-point settings collectively for a group of blocks. For an example that uses subsystems and hierarchy, see [Working with Custom Blocks](#), on page 5-1.

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.

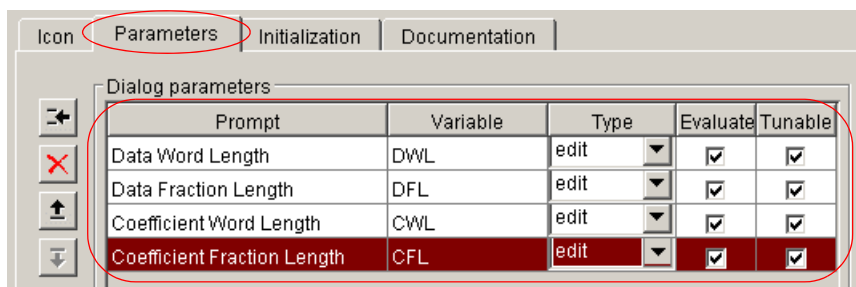
The blocks are grouped together and only the Subsystem block appears at the top level.



If you double-click the subsystem, another window shows you the internal hierarchy of the subsystem with the individual blocks you grouped.

## 2. Set up a mask to manage the fixed-point settings:

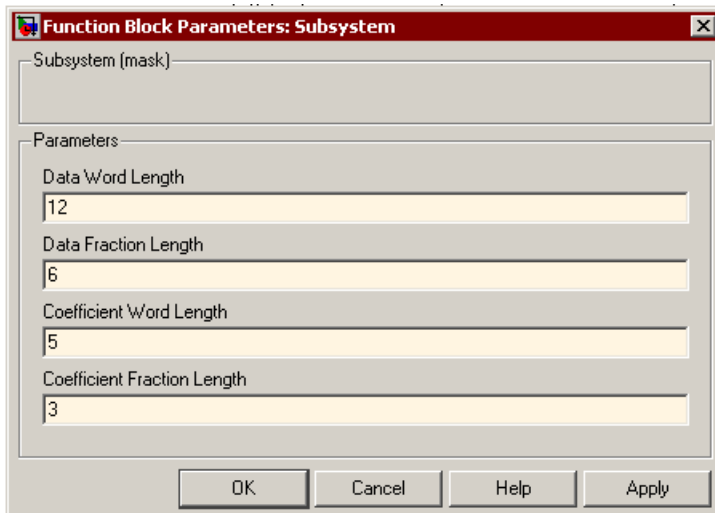
- In the schematic window, right-click the subsystem and select Mask System or Edit Mask to open the Mask editor window.
- Click the Parameters tab, which is relevant for fixed-point conversion.
- Specify variables to manage the fixed-point architecture:



## 3. 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.

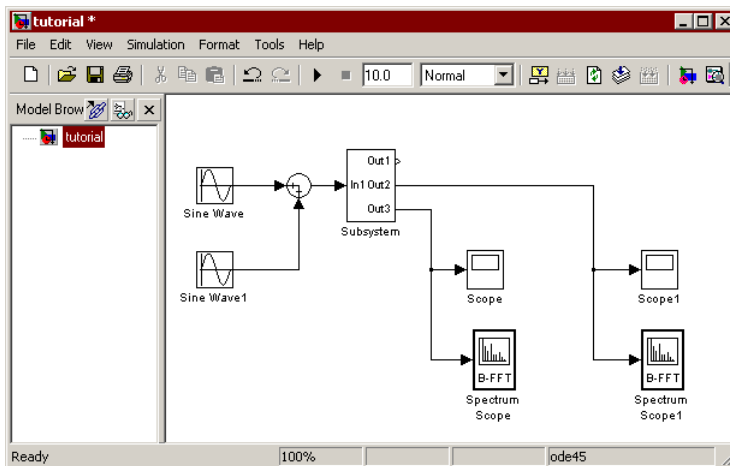
- Set the parameter values and click OK.



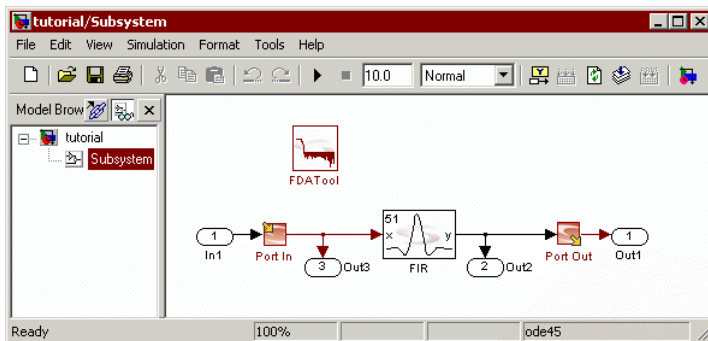
The tool applies these values to the subsystem. You can use any expression of the variables if you use the Specify option for the output format of individual blocks.

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.

# Running DSP Synthesis with SynDSPTool

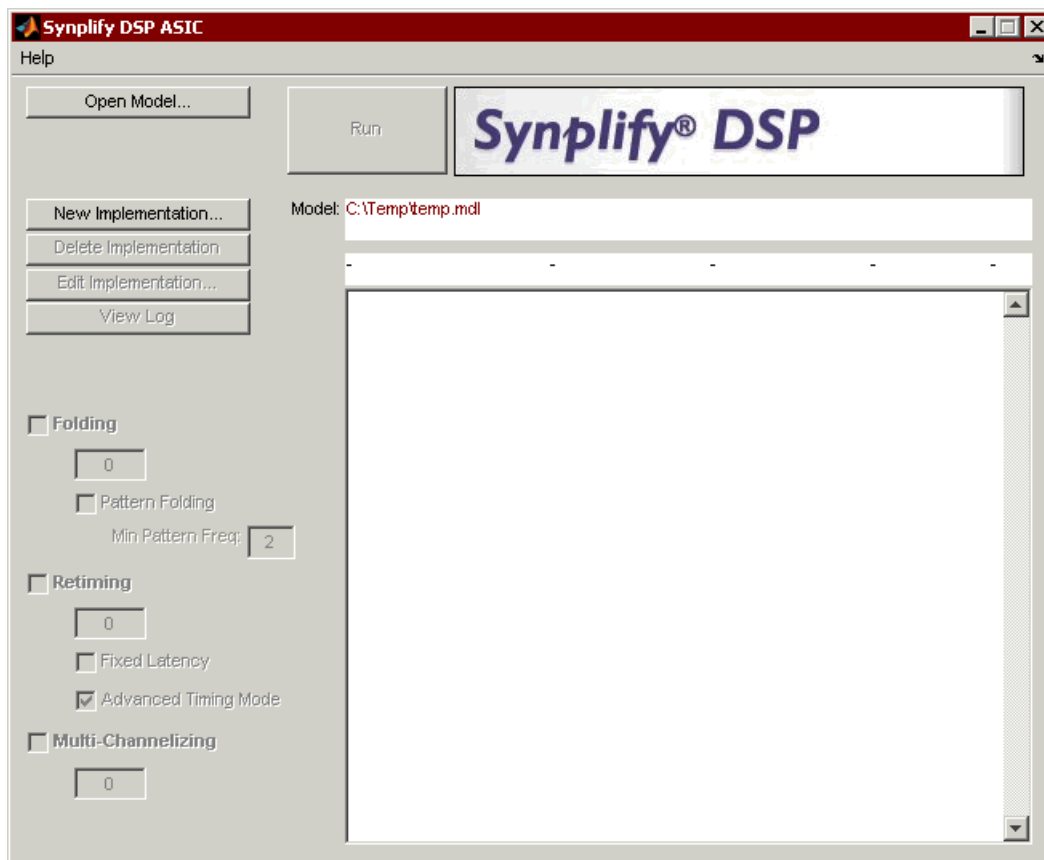
The SynDSPTool toolbox lets you specify different optimizations and generate RTL code. This section describes how you can use the toolbox for the following:

- [Setting up Implementations, on page 4-48](#)
- [Configuring Synplify DSP Timing Modes for FPGAs, on page 4-51](#)
- [Optimizing with Retiming, on page 4-53](#)
- [Optimizing with Folding, on page 4-55](#)
- [Optimizing with Multichannelization, on page 4-60](#)
- [Running DSP Synthesis for FPGA Targets, on page 4-61](#)
- [Running DSP Synthesis for ASIC Targets, on page 4-62](#)

## Setting up Implementations

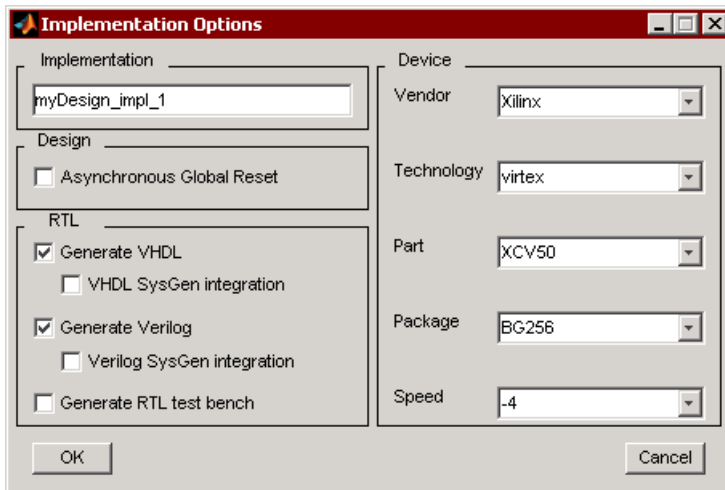
Implementations let you run the same design with different optimizations or target technologies so that you can evaluate the results.

1. Include the SynDSPTool block in your design.
  - Add the SynDSPTool block to your Simulink schematic design. When you add a SynDSPTool instance anywhere in the hierarchy, it controls the complete system you have captured, Synplify DSP optimizations and the generation of RTL code in particular.
  - When you have finished your design and verified it, double-click the SynDSPTool block in the schematic. The SynDSPTool window opens.
2. In this toolbox window, set Model to the appropriate model file, if necessary. By default, it shows the current file.



### 3. Open the implementation.

- You must create a new implementation if this is the first time you have opened the Synplify DSP 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 Synplify DSP window, and click Edit Implementation. This opens the Implementation Options dialog box.



4. Set options for the implementation.
  - In the Design section, enable Asynchronous Global Reset if you want your design to use registers with only asynchronous global resets.
  - For FPGAs, select the vendor, technology, part, package, and speed you want to target.
  - For ASIC designs, select the target technology. In order to set targets, you must have a Synplify DSP ASIC Edition license. See [Getting Started, on page 1-1](#) for information about Synplify DSP ASIC Edition and the design flow.
5. Set options for generating output files:
  - To generate an RTL file in VHDL format, enable Generate VHDL. See [Considerations for Generating RTL Output Files, on page 4-51](#).
  - To generate an RTL file in Verilog format, enable Generate Verilog. See [Considerations for Generating RTL Output Files, on page 4-51](#).
  - To generate a testbench, follow the procedure described in [Verifying the RTL with a Test Bench, on page 4-63](#).
6. Click OK.

The Synplify DSP window reflects the technology choices you made.



You can now run the implementation by clicking the Run button in the Synplify DSP window. Alternatively, you can set other optimization and output options before clicking Run. For information about the optimizations, see [Optimizing with Folding, on page 4-55](#), [Optimizing with Retiming, on page 4-53](#), and [Optimizing with Multichannelization, on page 4-60](#). For additional information about using the output files, see [Working with Synplify DSP Output, on page 4-62](#).

## Considerations for Generating RTL Output Files

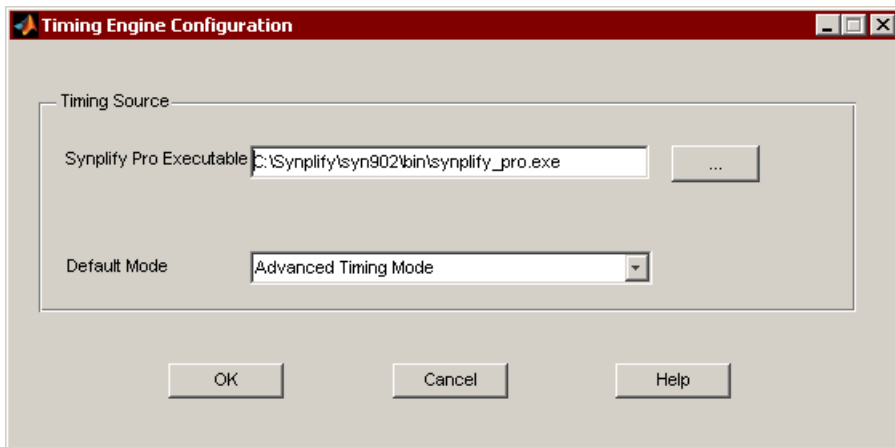
The following are some considerations for generating RTL output files in Verilog or VHDL formats. The procedure for generating the files is described in [Setting up Implementations, on page 4-48](#).

- **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.
- **Keywords**  
Make sure that your Simulink design does not use Verilog or VHDL keywords to name ports and instances.

## Configuring Synplify DSP Timing Modes for FPGAs

The Synplify DSP 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\\_ate, on page 9-13](#)) at the MATLAB command prompt.



- Set Default Mode to the mode you want to use, either advanced timing mode or estimation mode. See [Timing Engine Configuration Dialog Box, on page 9-13](#) for descriptions of this dialog box.
  - If you set the default to Advanced Timing Mode, you must also set Synplify Pro executable to point to the Synplify Pro executable. This is because advanced timing mode uses Synplify Pro 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 SynDSPTool 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 SynDSPTool 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.

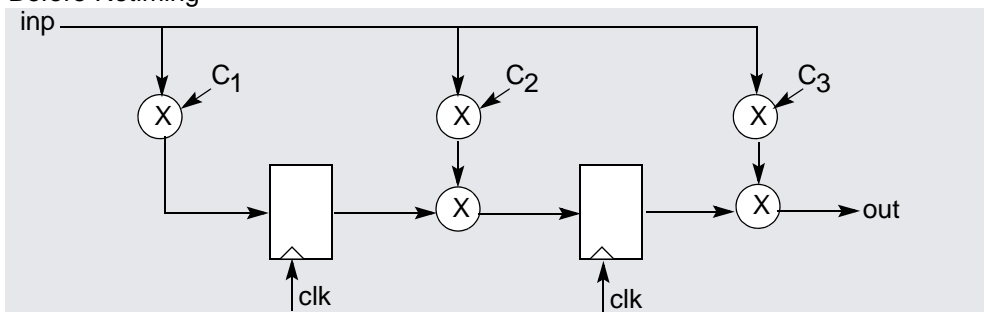
#### 4. Run DSP synthesis.

The tool uses the timing mode you specified to run synthesis. The log file reports blocks that met timing, blocks that did not meet timing, and blocks in timing loops.

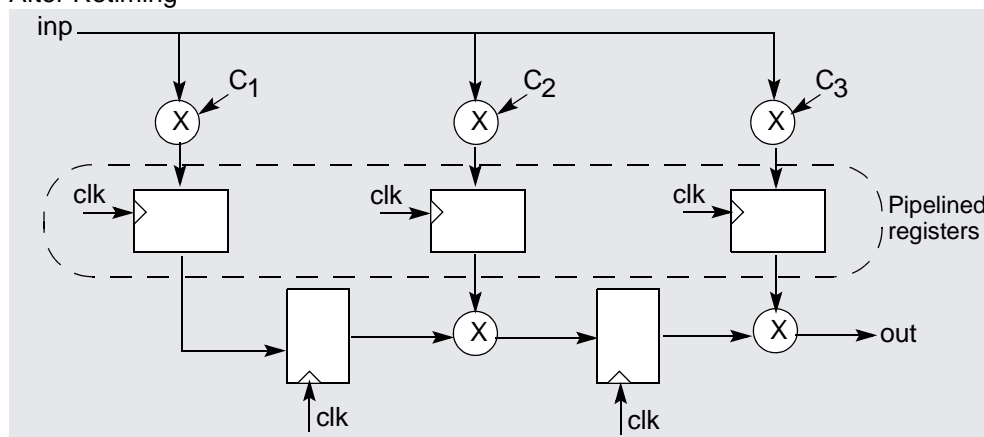
## Optimizing with Retiming

Retiming is a performance optimization that improves speed by rearranging registers or adding extra registers if you specified them. A delay element is implemented as one or more registers.

Before Retiming



After Retiming



1. If you want to use advanced timing mode for retiming calculation, make sure you have selected this timing mode. See [Configuring Synplify DSP Timing Modes for FPGAs, on page 4-51](#) for details.

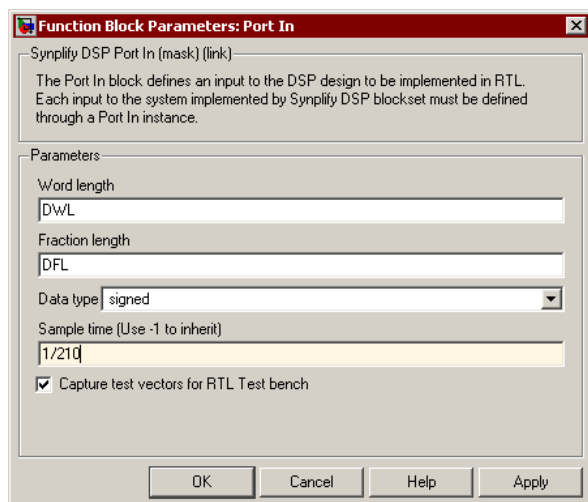
2. After setting up the implementation and target technology (see [Setting up Implementations, on page 4-48](#)), click Retiming in the SynDSPTool window.
3. In the text box that opens, set a value.
  - To retime your design without adding any extra latency to the design, set the value to 0. The Synplify DSP tool rearranges the existing registers 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 Synplify DSP 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. A delay element is implemented as one or more registers. The latency of the design increases, depending on the number of pipeline stages actually used for retiming.
4. Set any other optimizations you want, and click Run.
5. 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 registers.

## 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.

## Optimizing with Folding

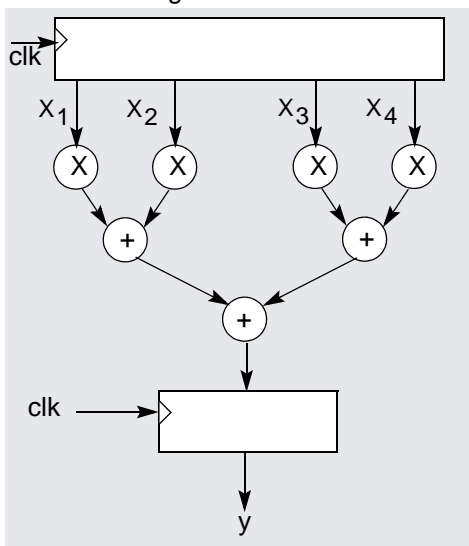
Folding is an area optimization that shares resources. The Synplify DSP tool folds the algorithm by reusing the same resources over multiple physical clock cycles. This helps the designer reduce the number of expensive functions like multipliers that take up a lot of silicon. The Synplify DSP tool automatically puts in control logic near the multipliers so that they can be multiplexed. When you use this option, you cannot use Multi-channelize, which

is an alternative mechanism to trade speed for resources. In multirate designs, when the folding factor is set to 1, the slow clock domain is folded automatically to the clock decimation amount.

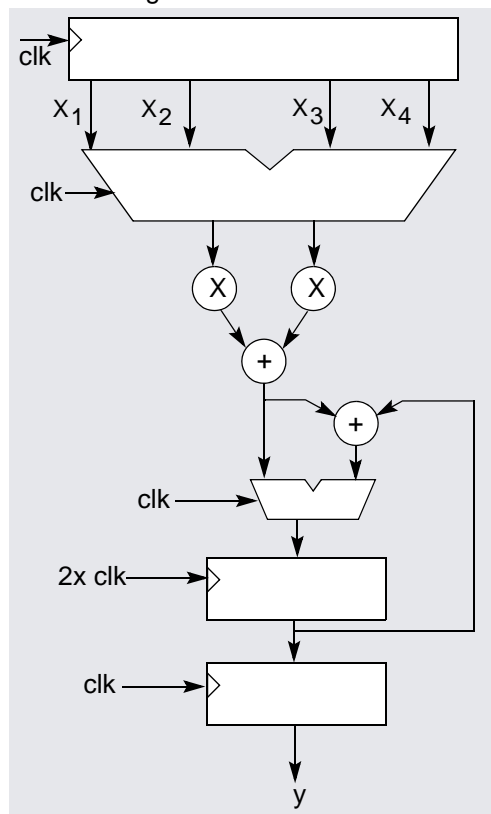
1. In the Simulink window, double-click the SynDSPTool block, and set up the implementation and target technology (see [Setting up Implementations, on page 4-48](#)).
2. Click Folding in the SynDSPTool window. This automatically enables the Retiming and Pattern Folding options.
3. Set the Folding, Pattern 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 [SynDSPTool Toolbox Interface, on page 8-254](#).
  - 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 registers 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.

Before Folding



After Folding



## Pattern Folding

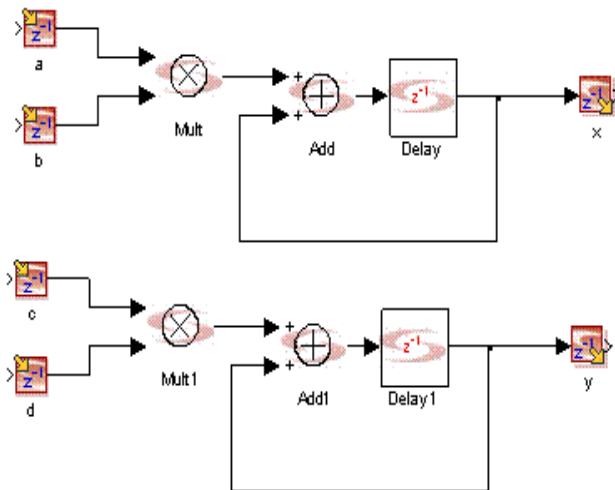
Pattern folding is an optional preprocessing step for folding, where the tool identifies and marks repeating patterns in the design, for time-multiplexed resource sharing when the folding algorithm is run. A pattern, in this context, is a group of Synplify DSP block instances and the interconnect between these instances. Rate-changing blocks and I/O blocks are not identified as part of a pattern.

## 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.

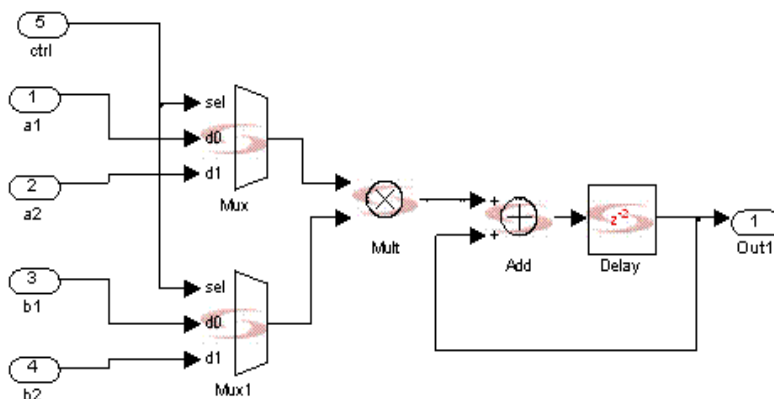
In the following design, folding by 2 results in a multiplexing overhead of 4 multiplexers. Using independent resource-sharing for the multipliers and adders would require 2 multiplexers for the multiplier inputs and another 2 for the adder inputs, resulting in an overhead total of 4.



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.

Pattern folding is optional, so if you want to use pattern folding, you must enable Folding and Pattern Folding on the SynDSPTool toolbox, and supply a minimum threshold value for pattern repetition in Min Pattern Freq (see the preceding procedure, in [Optimizing with Folding, on page 4-55](#)).





## 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.)
- There is no support for user guidance in pattern identification.

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 4-55](#)) are mutually exclusive.

The diagram shows a 2-stage pipeline. An input signal 'inp' is connected to three combinational logic blocks, each containing a multiplier 'X'. The first multiplier is also controlled by 'C1'. The output of the first multiplier is connected to the first D flip-flop. The output of the first flip-flop is connected to a second combinational logic block containing an adder '+', which is also controlled by 'C2'. The output of this adder is connected to the second D flip-flop. The output of the second flip-flop is connected to a third combinational logic block containing another adder '+', which is also controlled by 'C3'. The output of this final adder is labeled 'out'. Both D flip-flops are clocked by a common 'clk' signal.

The diagram illustrates a hardware architecture for processing multiple inputs. On the left, a multiplexer combines inputs  $inp1, inp2, \dots, inpN$ . Each input is then multiplied by a constant  $C_1, C_2, \dots, C_3$  respectively. The results are fed into a series of  $N$  registers, each clocked by  $clkN$ . The outputs of these registers are summed together. The final sum is compared with a value  $O$  to produce  $out1$ , and compared with a value  $N-1$  to produce  $outN$ . A Modulo- $N$  Counter, which takes  $clkN = N \times clk$  as input, provides a counter value  $cnt$  that is used in the comparison stage.

1. In the Simulink window, double-click the SynDSPTool block, and set up the implementation and target technology (see [Setting up Implementations, on page 4-48](#)).
2. Click Multi-channelize in the SynDSPTool 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 registers specified and the number of registers actually used for retiming.

## Running DSP Synthesis for FPGA Targets

After setting the implementation and optimization options described in [Running DSP Synthesis with SynDSPTool, on page 4-48](#), do the following to run DSP synthesis and generate the output files.

1. After setting the optimizations you want, click Run in the Synplify DSP window.

The tool runs with the targets you set and generates the output files you specified.

You can only use the Synplify Pro or Synplify Premier tools for FPGA logic synthesis. Synplify DSP 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 or .v	VHDL or Verilog netlist for synthesis. Each format has a separate subdirectory under the implementation directory.
.sdc	Constraint file for synthesis
..vhd	Optional testbench for simulation

The software also generates a testbench for verification. The files are in the model directory.

## Running DSP Synthesis for ASIC Targets

After setting the implementation and optimization options described in [Running DSP Synthesis with SynDSPTool, on page 4-48](#), do the following to run DSP synthesis and generate the output files.

1. Click Run in the Synplify DSP window.

The tool runs with the targets you set and generates the output files you specified. All DSP synthesis optimizations use ASIC timing characterizations for the blocks. In addition, you can extract memories to support third-party memory generators.

The following are some of the files the tool generates for ASIC logic synthesis:

- A constraint file in .sdc format
- A Tcl script example of a logic synthesis project.

Use these files to run logic synthesis with one of the recommended third-party tools.

## Working with Synplify DSP Output

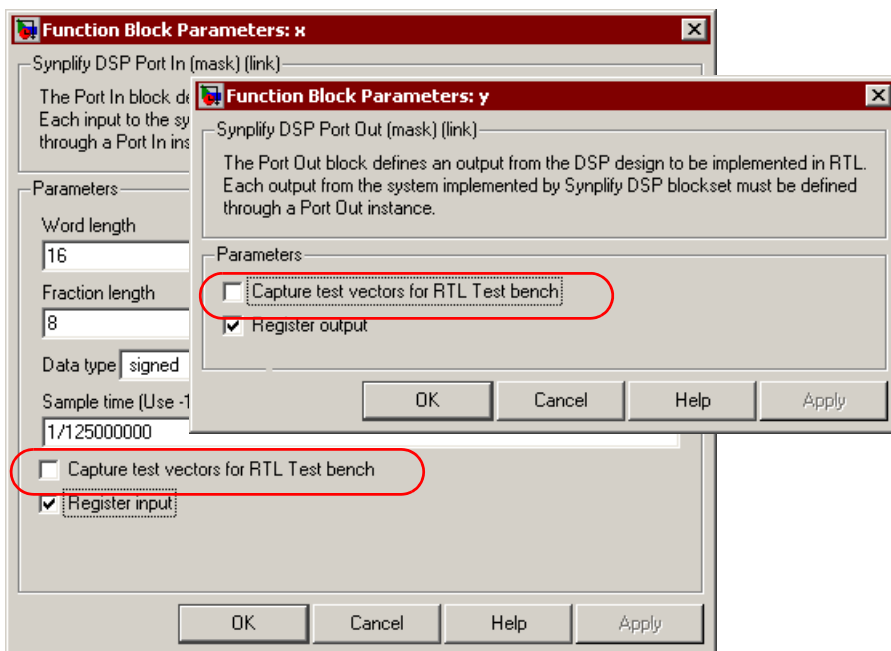
This section describes how to generate and use the Synplify DSP output files:

- [Verifying the RTL with a Test Bench, on page 4-63](#)
- [Running Logic Synthesis for FPGA Targets, on page 4-65](#)
- [Running Logic Synthesis for ASIC Targets, on page 4-66](#)
- [Working with the Actel Encrypted Flow, on page 4-67](#)

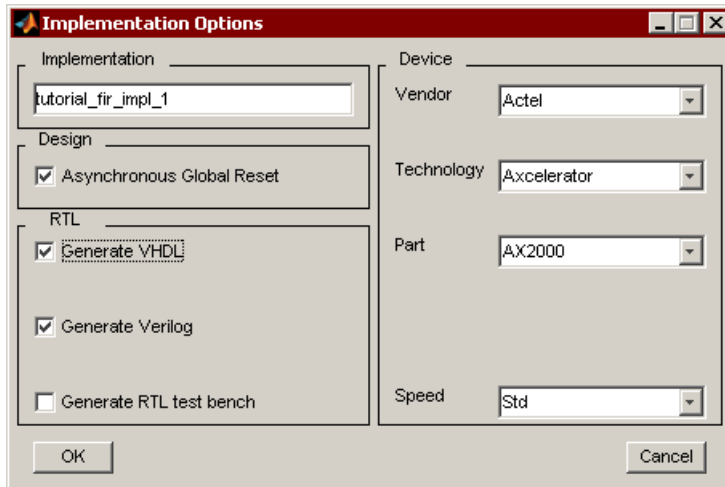
## Verifying the RTL with a Test Bench

The following procedure shows you how to generate a test bench from the Synplify DSP 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 SynDSPTool block in the model window and then click New Implementation or Edit Implementation in the SynDSPTool 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 4-48](#)).
  - Click OK.
4. Click Run in the Synplify DSP window.

The Synplify DSP 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 directory 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.

## Running Logic Synthesis for FPGA Targets

This procedure shows you how to use the Synplify DSP output for logic synthesis with the Synplicity FPGA synthesis tools: Synplify Pro or Synplify Premier. These are the only supported tools.

1. Make sure you have access to a Synplicity FPGA synthesis tool, Synplify Pro or Synplify Premier.

2. Create your Synplify DSP implementation and specify the following implementation options. See [Setting up Implementations, on page 4-48](#) for details.
  - Specify an FPGA target architecture.
  - Specify the format for the output netlist, and any optimizations (see [Optimizing with Retiming, on page 4-53](#), [Optimizing with Folding, on page 4-55](#), and [Optimizing with Multichannelization, on page 4-60](#)).
  - Click Run.

The Synplify DSP software generates an optimized RTL netlist, a project file and other files (see [Running DSP Synthesis for FPGA Targets, on page 4-61](#) for a list of 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 4-63](#)).
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 Synplify DSP tool.
5. Set synthesis options and constraints and synthesize your design.

In the logic synthesis tool, you can further explore device tradeoffs by setting device-specific constraints. You can also use various vendor-specific synthesis optimizations to further refine your design before placing and routing it.

## Running Logic Synthesis for ASIC Targets

This procedure shows you how to use the Synplify DSP output for logic synthesis with an ASIC synthesis tool. See the release notes for recommended third-party tools for ASIC synthesis.

1. Set up your Synplify DSP implementation and specify the following implementation options. See [Setting up Implementations, on page 4-48](#) for details.
  - Specify an ASIC target architecture.



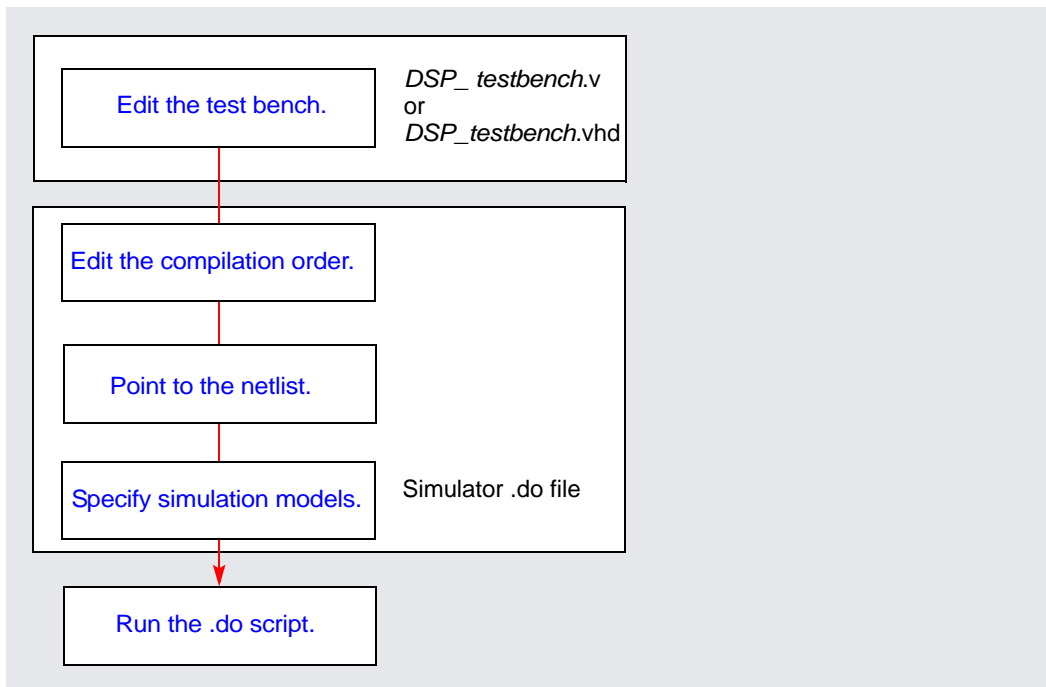
- Specify the format for the output netlist, and any optimizations (see [Optimizing with Retiming](#), on page 4-53, [Optimizing with Folding](#), on page 4-55, and [Optimizing with Multichannelization](#), on page 4-60).
- Click Run.

The Synplify DSP software generates an optimized RTL netlist, a constraint file in the .sdc format, and a Tcl script for running synthesis. If you specified it, it also generates a test bench for RTL verification.

2. Verify your design (see [Verifying the RTL with a Test Bench](#), on page 4-63).
3. Start the ASIC synthesis tool, using the RTL netlist generated by the Synplify DSP tool as inputs.
4. Run the tool and synthesize your design.

## Working with the Actel Encrypted Flow

The RTL netlist generated after Synplify DSP synthesis is encrypted. To work with the encrypted file and simulate Actel designs, you must follow the flow shown below after Synplify DSP synthesis is complete. The procedure after the flow diagram documents the details of the steps shown in the flow.



#### 1. Edit the test bench.

This is the test bench generated after Synplify DSP synthesis, and can be either Verilog or VHDL.

- For Verilog, edit the test bench to make sure that the first line of the testbench is

```
`timescale 1ns/100ps
```

Currently the synthesis software does not write out ``timescale` in the Verilog test-bench, so you have to manually edit this.

- For both VHDL and Verilog test benches, ensure that the correct clock periods and reset period values are passed in when the clock module is instantiated. To do this, edit the `cperiod` and `rperiod` values so that they are the same as the values found at the beginning of the test bench.

```
//Instantiate design under test
my_FIR i_my_FIR (
    .clk(clk_int),
    .GlobalReset(GlobalReset_int),
    .GlobalEnable1(GlobalEnable1_int),
    .Port_Out(Port_Out_int),
    .Port_In(Port_In_int)
);

clocks #(.cperiod(4), .rperiod(50)) CL(
//Change 4 and 50 to the values at the beginning of the test bench.
    .clk(clk_int),
    .rst(GlobalReset_int)
);
```

## 2. Edit the compilation order.

- Open the <simulator>.do file.
- Change the order of compilation for simulation, so that the test bench is the first file to be compiled.

## 3. Point to the netlist.

- Open the <simulator>.do file.
- In the list of files to be compiled, point to the netlist file (.vm or .vhm) instead of pointing to the RTL module. For example, in the ModelSim .do file, replace vlog "<mydesign.v>" with vlog  
"<synthesis\_implementation\_name>/<mydesign.vm/vhm"

## 4. Specify simulation models.

You only need to do this step if you are working outside Libero. If you are using Libero, the simulation models are available already.

- Either copy the relevant simulation models for that technology, or point to the appropriate file in the simulator's .do file list of files to be compiled. If X is the Libero installation directory, the relevant technology file is located here:

```
X:\Actel\designer7101\Designer\lib\vlog\<technology.v>
X:\Actel\designer7101\Designer\lib\vtl\95\<technology.vhd>
```

The ModelSim simulation model file to be compiled varies with the technology, and has the appropriate Verilog or VHDL extension. For example for Pro ASIC, the model file is either of the following:

```
X:\Actel\designer7101\Designer\lib\vlog\apa.v
X:\Actel\designer7101\Designer\lib\vtl\95\apa.vhd
```

- Add the model files as described below:

**Verilog** Use the `vlog` command to add the model file to the list of files to be compiled. For example:

```
vlog "apa.v"  
vlog "<path to apa.v>/apa.v"
```

---

**VHDL** • Create the <technology\_name> library with the `vlib` and `vmap` commands in the `.do` file, as shown in the following example:

```
vlib apa  
vmap work apa
```

- Compile the "<synplify\_installation>/lib/synplify.vhd" file into the synplify library by adding the following lines to the `.do` file:

```
vlib synplify  
vcom -work synplify  
"N:/<install_dir>/lib/vhdl_sim/synplify.vhd"
```

---

## 5. Run the `.do` script.

Start the simulator and run the `.do` script to finish the simulation.

## CHAPTER 5

# Working with Custom Blocks

---

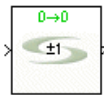
This tutorial describes how to generate a custom block:

- [Primitives and Custom Blocks, on page 5-2](#)
- [Design Flow for Building Custom Blocks, on page 5-5](#)
- [Set up a Custom Library, on page 5-6](#)
- [Create a Custom Block, on page 5-7](#)
- [Define Basic Content for Custom Blocks, on page 5-13](#)
- [Define Content for Parameterized Blocks, on page 5-16](#)
- [Define Content for Reconfigurable Blocks, on page 5-20](#)
- [Designing with Custom Blocks, on page 5-24](#)
- [Maintaining Custom Libraries, on page 5-25](#)
- [The MySign M-Generator, on page 5-26](#)

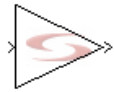
# Primitives and Custom Blocks

The Synplify DSP 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 Synplify DSP 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 5-5](#)). Custom blocks are distinguished from primitive blocks by their icon; the Synplify DSP custom blocks have a green Synplicity S logo instead of a red one.



Custom block with green Synplicity S logo



Primitive block with red Synplicity 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 Synplify DSP primitive blocks and other custom blocks. The Synplify DSP tool will generate RTL for these blocks as it will for the primitives.
- The Synplify DSP custom blocks can be optimized like the primitive blocks.

## List of Custom Blocks

The following table lists the Synplify DSP custom blocks:

<a href="#">Synplify DSP Block Deinterleaver</a>	Shuffles a fixed number of interleaved input symbols to obtain the original sequence.
<a href="#">Synplify DSP Block Interleaver</a>	Shuffles a fixed number of input symbols to a new permutation.
<a href="#">Synplify DSP CIC</a>	Implements a CIC filter.

Synplify DSP Commutator	Sequentially switches the data from the specified number of input ports to a single output port.
Synplify DSP Convolutional Deinterleaver	Reshuffles streaming input symbols according a to a predefined mapping scheme.
Synplify DSP Convolutional Encoder	Corrects feed-forward errors using k/n convolutional codes.
Synplify DSP Convolutional Interleaver	Shuffles streaming input symbols to a new permutation, using a predefined mapping scheme.
Synplify DSP DDS	Creates a direct digital synthesizer, with sin and cos waves based on frequency, phase settings, and modulations.
Synplify DSP Decommutator	Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly.
Synplify DSP Depuncture	Removes specified bits from the input data stream and replaces them with zeroes.
Synplify DSP Differentiator	Performs a discrete time differentiation of the input signal.
Synplify DSP Extract	Provides an n-bit integer based on the value of the bits extracted from specified positions at the input.
Synplify DSP FIR Rate Converter	Implements a polyphase FIR filter.
Synplify DSP Integrator	Performs a discrete time integration of the input signal.
Synplify DSP MinMax	Calculates the minimum, maximum, or minimum and maximum of two inputs.
Synplify DSP 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.
Synplify DSP Puncture	Removes user-specified bits from the input data stream
Synplify DSP Ramp	Creates a ramp based on increments derived from a port or parameter
Synplify DSP Random	Creates a random integer of the requested word length

---

Synplify DSP Recast	Provides a value, based on the requested data type cast at the output and maintaining the same bits as provided at the input
Synplify DSP Register	Inserts a delay.
Synplify DSP RFIR	Custom block that implements a reloadable finite impulse response FIR filter.
Synplify DSP Sequence	Repeats a sequence of specified data
Synplify DSP 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.
Synplify DSP Sign	Provides the 2-bit sign value (+1 or -1) for the input.

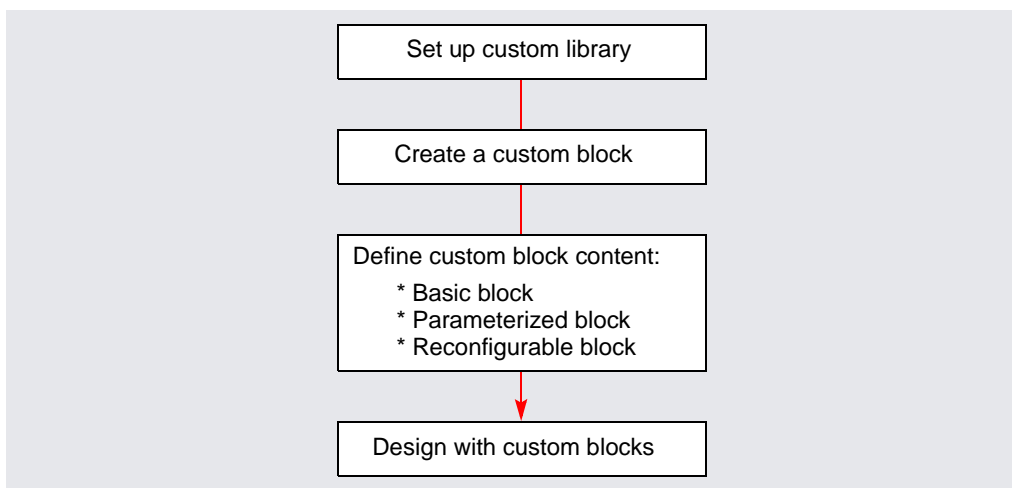
---



# 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 Synplify DSP blocks. For a definition of custom blocks, see [Primitives and Custom Blocks, on page 5-2](#).

The following figure illustrates the steps in the tutorial and shows the different kinds of custom blocks you can create.



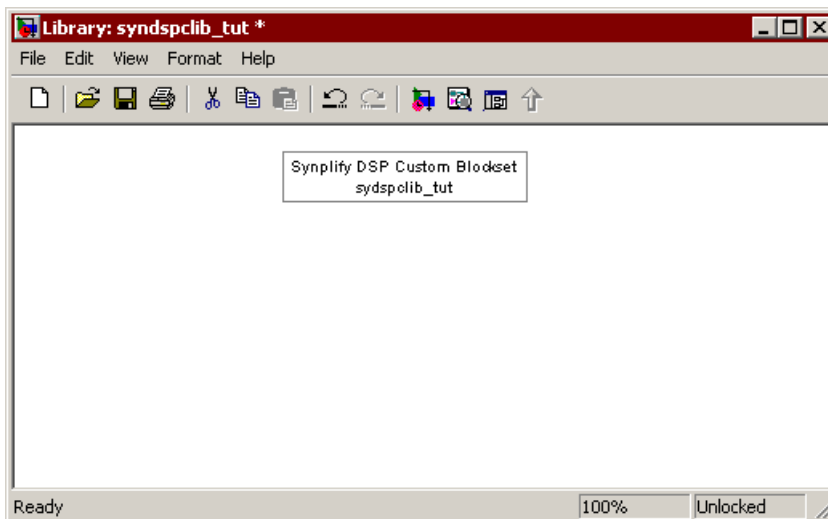
The design flow above is described in the following sections:

- [Set up a Custom Library, on page 5-6](#)
- [Create a Custom Block, on page 5-7](#)
- [Define Basic Content for Custom Blocks, on page 5-13](#)
- [Define Content for Parameterized Blocks, on page 5-16](#)
- [Define Content for Reconfigurable Blocks, on page 5-20](#)
- [Designing with Custom Blocks, on page 5-24](#)
- [The MySign M-Generator, on page 5-26](#)

# Set up a Custom Library

The first step in this tutorial 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 syndspclib\_tut.mdl. All custom libraries must be named syndspclib<string>.mdl. Do not name it syndspclib, as this name is reserved for the main Synplify DSP library.
- Save the library file to a location in the MATLAB search path. For example, a good location is the SynDSP directory in the MATLAB installation hierarchy: <matlab\_install\_dir>/toolbox/Synplicity/SynDSP/syndspclib\_tut.mdl.
- 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`

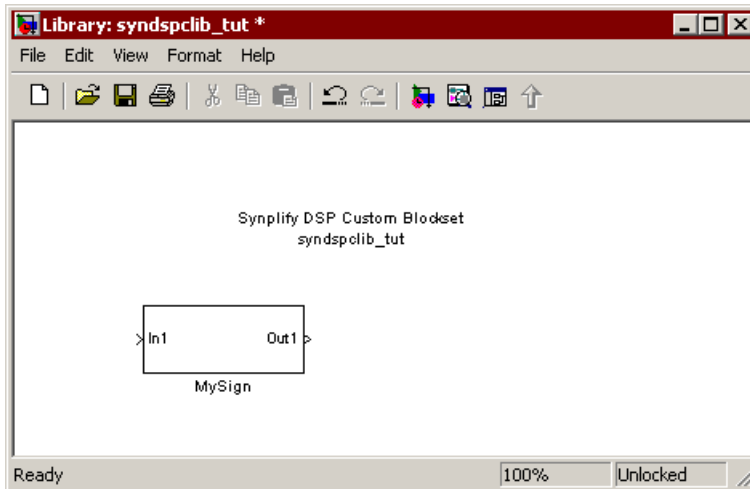
This command rescans all the toolbox directories for new files and updates the MATLAB cache file. You only need to run the `rehash` command once. At subsequent sessions you do not need the command, but can open the library by just typing its name at the MATLAB prompt. You can now create a custom block as described in [Create a Custom Block, on page 5-7](#).

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 5-25](#).

## Create a Custom Block

This tutorial uses a step-by-step procedure to show 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.

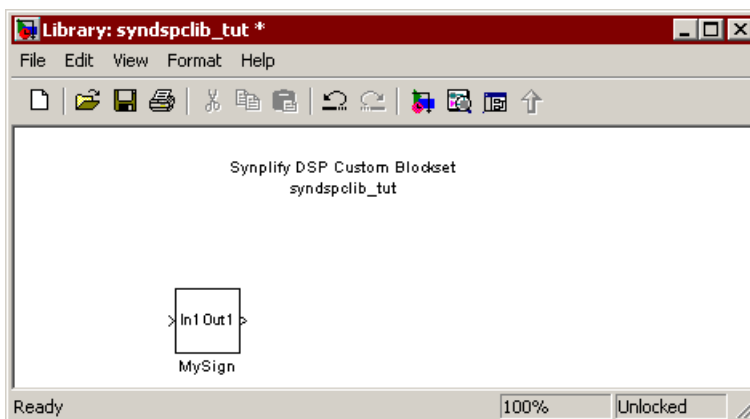
1. After you have created the custom library, create a subsystem for the custom block.
  - Drag a Subsystem block from the Simulink Simulink->Ports & Subsystems library into the library schematic window. The Subsystem block provides the starting point for the block.
  - Close the Simulink library window.
  - In the custom library window, rename the instance by double-clicking the name and typing a new name; in this case, type MySign.



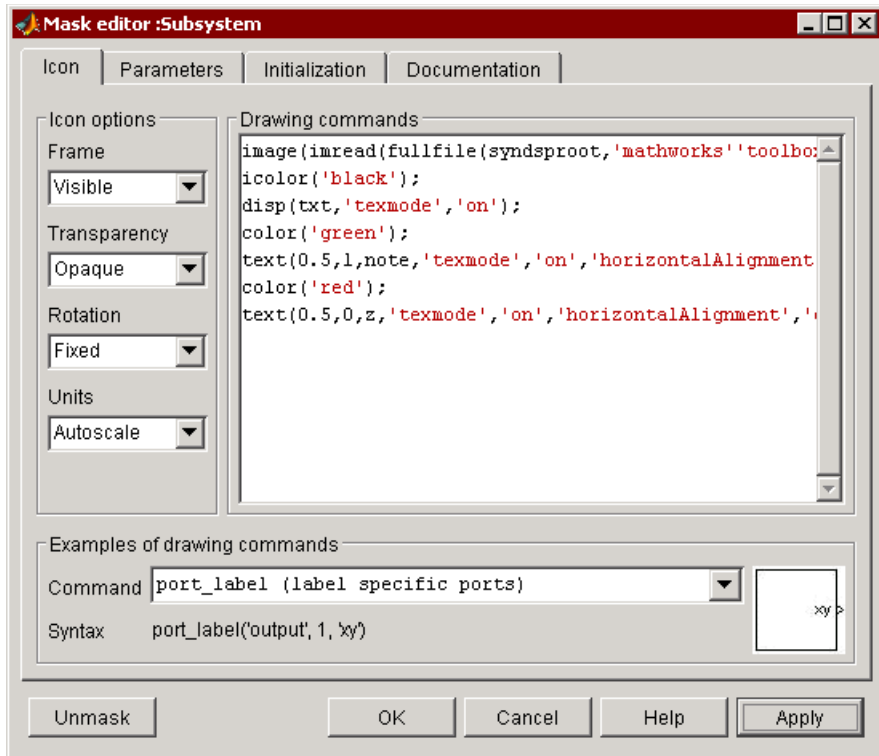
- Position and size the block by typing the following at the MATLAB command prompt:

```
set_param('syndspclib_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  $40 \times 40$  pixels. For positioning, it is a good practice to put the origins of different blocks on a grid of  $100 \times 100$  pixels. The Synplify blocks use a standard size of  $40 \times 40$  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 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.

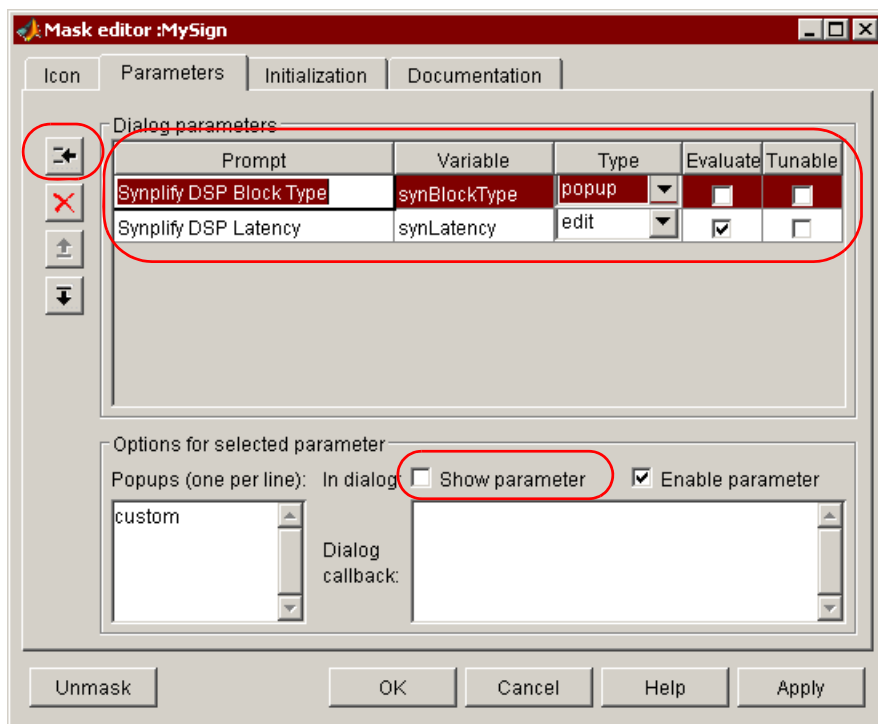


The following table shows the commands specified for the MySign block. Note that no port labels are defined because they are obvious.

Command	Effect
<pre>image(imread(fullfile(syndsproot,'mathworks','toolbox','Synplicity','icons','synplicity40_fg.jpg')),center); icolor('black'); disp(txt,'texmode','on'); color('green'); text(0.5,1,note,'texmode','on','horizontalAlignment','left'); color('red'); text(0.5,0,z,'texmode','on','horizontalAlignment','left');</pre>	Puts the specified logo image on the icon.

Command	Effect
<code>color('black');</code> <code>disp(txt,'texmode','on');</code>	Sets the color of text in the <code>txt</code> variable. The variable itself is defined on the Initialization tab.
<code>color('green');</code> <code>text(0.5,1,note,'texmode','on','horizontalAlignment','center','verticalAlignment','top');</code>	Defines a placeholder for a <code>note</code> variable. The <code>note</code> is defined on the Initialization tab.
<code>color('red');</code> <code>text(0.5,0,z,'texmode','on','horizontalAlignment','center','verticalAlignment','bottom');</code>	Defines a placeholder for the <code>z</code> (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.

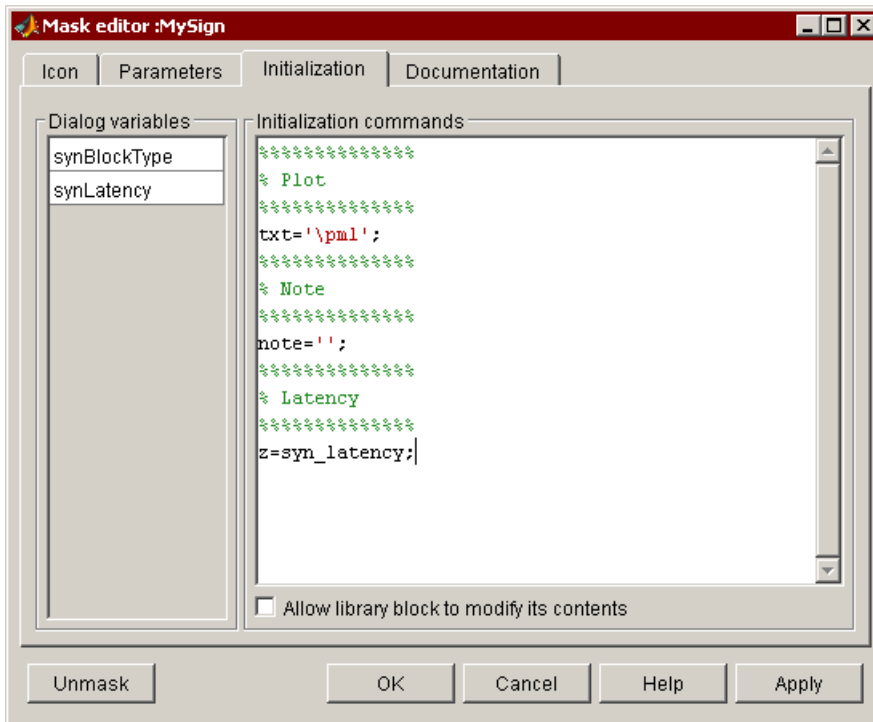


- Add another parameter line to define synLatency, to hold the potential latency of the block. Disable the Tunable and Show Parameters checkboxes. See the previous figure for details.

Setting parameters causes the underlying block schematics to be hidden from the designer in the Library Browser.

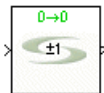
4. Select the Initialization tab and specify initialization code for the variables defined on the Icon tab. For the MySign block, do the following:
  - In the Initialization commands area, define the txt, note, and z variables with the following commands:
 

```
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 description in Mask Type. For example: Synplify DSP 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 5-13).

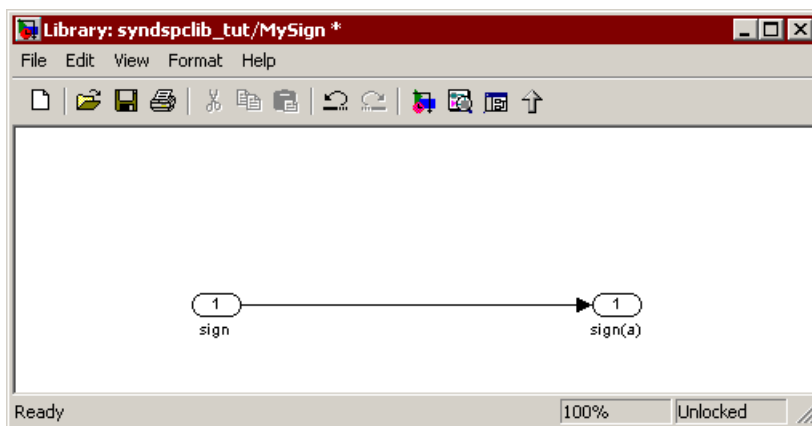
## Define Basic Content for Custom Blocks

After creating a custom library and block (see [Set up a Custom Library](#), on page 5-6 and [Create a Custom Block](#), on page 5-7), you still have to define the contents of the block. The following procedure shows you how to group Synplify DSP 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.

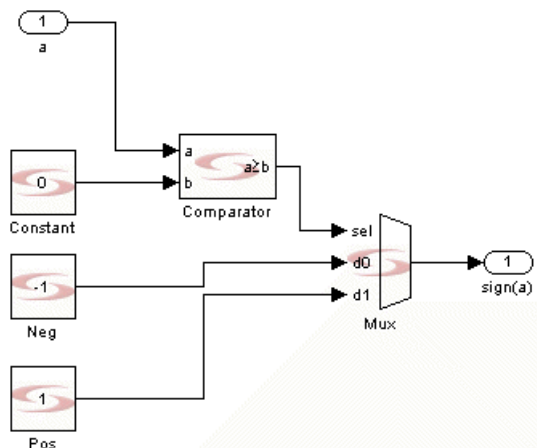
A window opens with the initial content of the masked block, which is one input port and one output port. If you do not use the Look under mask command, you will not be able to see or edit any content when you double-click the block because it has been masked.

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 Synplify DSP library or with other Synplify DSP custom blocks.

The following figure shows the MySign block defined with `sfix2_En0(+1)` on the output for inputs larger than or equal to 0, and `sfix2_En0(-1)` for inputs smaller than 0.

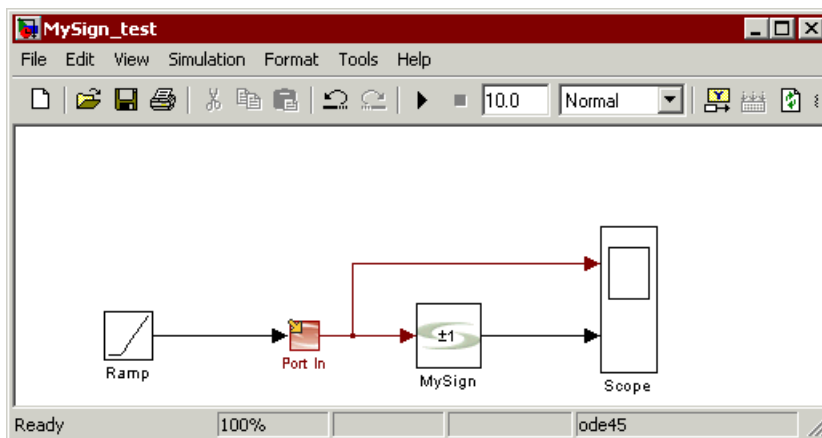


4. In the library window, select File->Save to save the contents of the block.

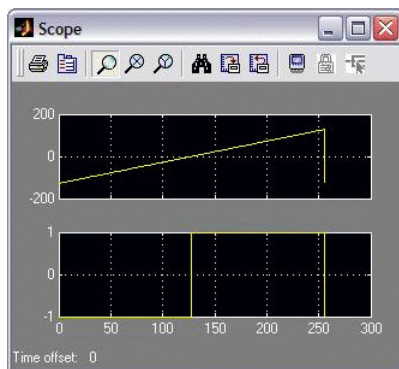
This procedure creates a static block where primitives are grouped together for a higher-level function. For information about more flexible blocks, see [Define Content for Parameterized Blocks, on page 5-16](#) and [Define Content for Reconfigurable Blocks, on page 5-20](#).

## 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:

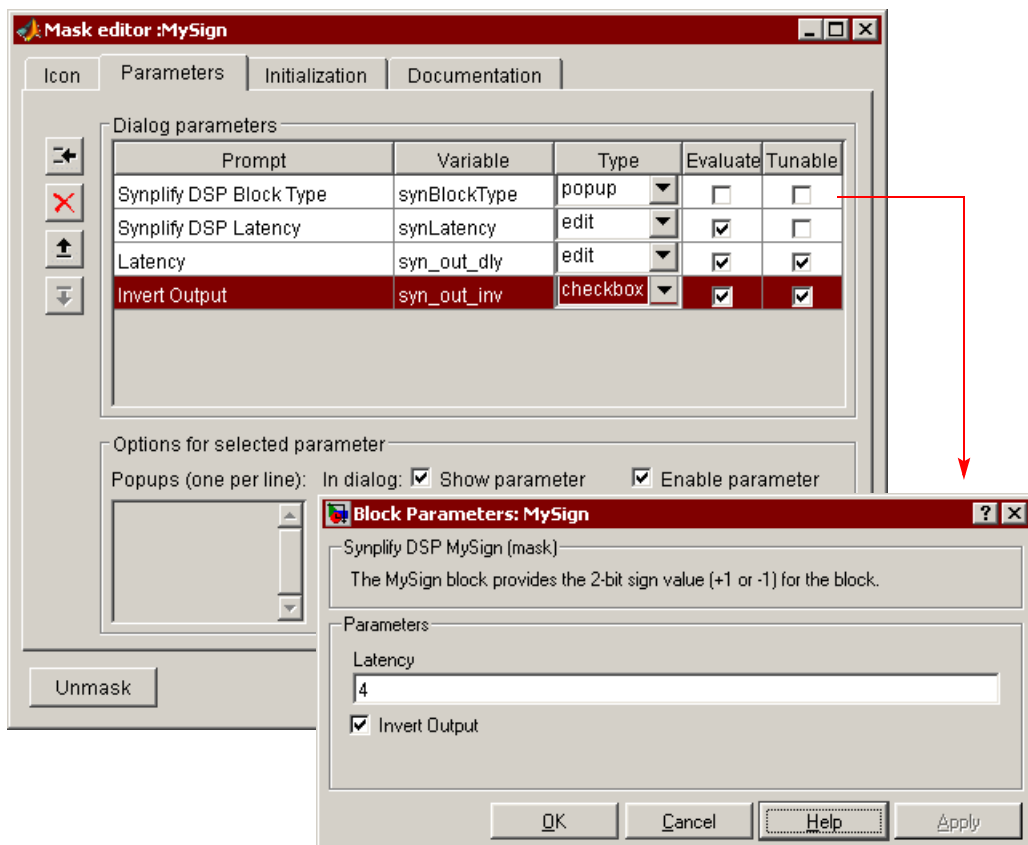


# Define Content for Parameterized Blocks

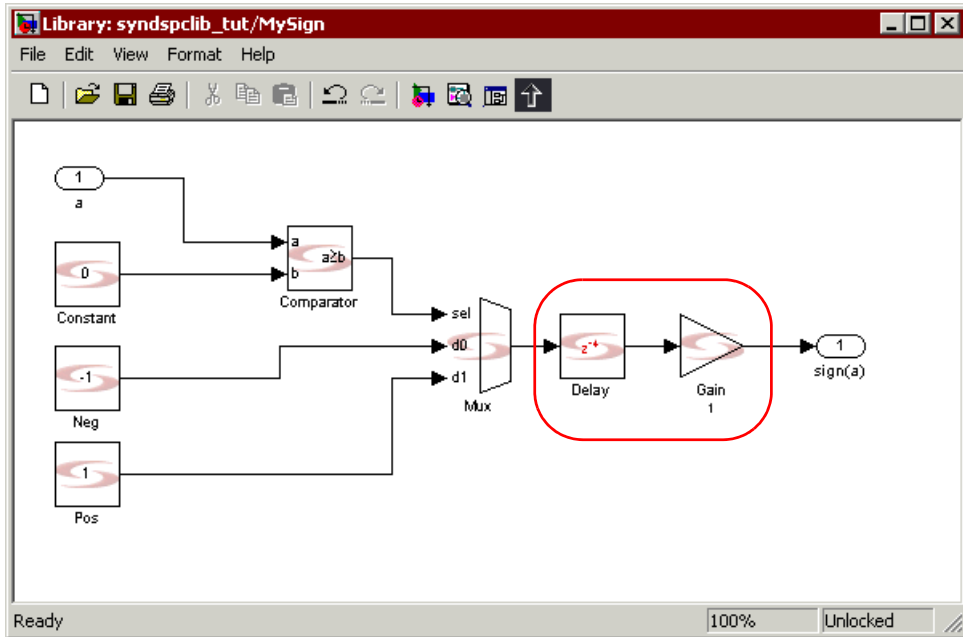
Parameterized blocks include parameters in the block mask, which allow you to fine-tune some options. The following example starts with the MySign block you created earlier (see [Create a Custom Block, on page 5-7](#)), and adds parameters for specifying latency and reversing the output.

1. Open the block.
  - At the MATLAB prompt, type `syndspclib_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 `syndspclib_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

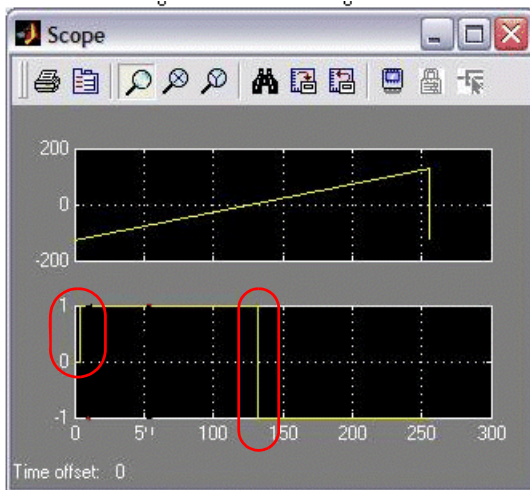
```

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 5-13](#).
- 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 5-16](#)), 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 5-13](#)) 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 `syndspclib_tut` to open the window with the custom blockset.
  - From the library window, select `Edit->Unlock Library`. This allows you to edit the blockset.

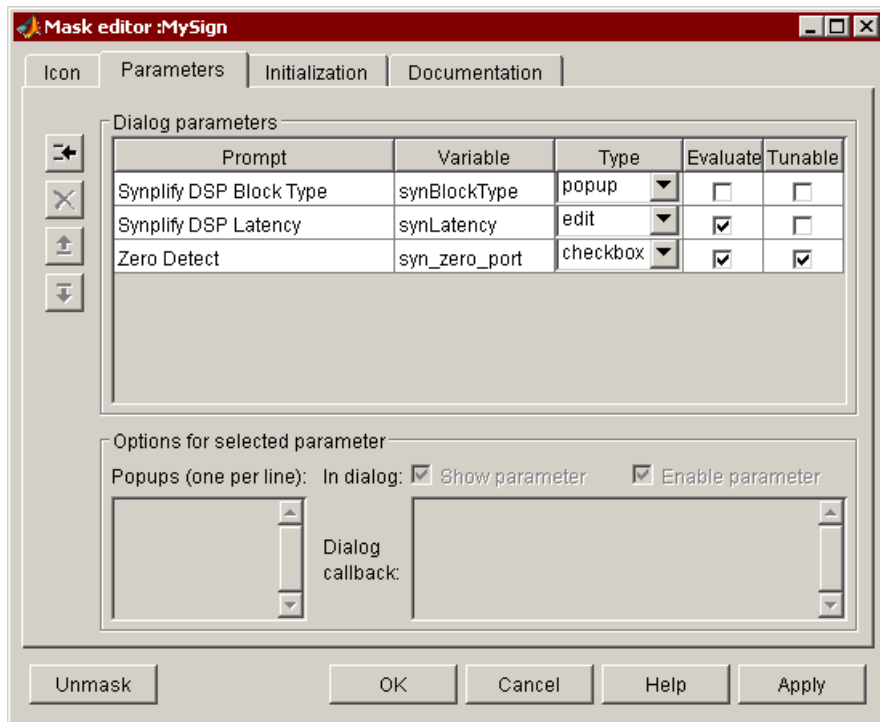


- At the MATLAB prompt, type the following command, which resizes the icon width to 60 pixels:

```
set_param('syndspclib_tut/MySign','Position',[100 100 160 140])
```

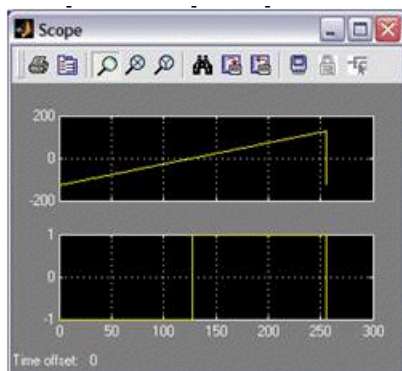
2. Provide an M-generator mask (see [The MySign M-Generator, on page 5-26](#) for an example of an M-generator for the MySign block). When you have an M-generator mask, the tool uses the M-generator to initialize the icon, instead of the settings on the Icon tab.

- 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.



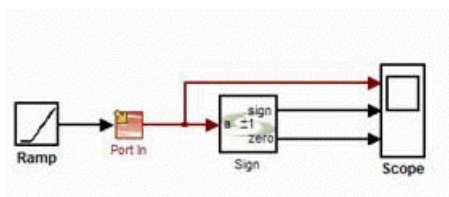
- On the Initialization tab, delete all the code and type `syn_mysign_init;` This is a call to the M-generator, which is called `syn_mysign_init`. See [The MySign M-Generator, on page 5-26](#) for the code.



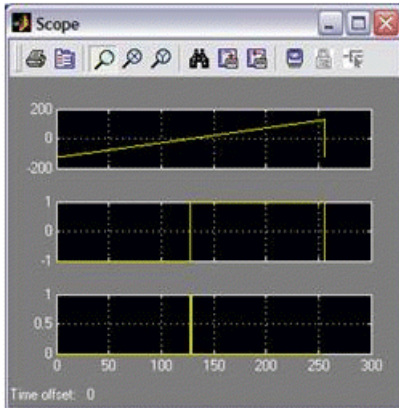


5. Run another test with zero detection.

- 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: Synplify DSP Custom Blockset `syndspclib`.
- 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 Synplify DSP procedures when you instantiate and use the custom block in your design.

4. Generate RTL.

In the Synplify DSP 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 5-26

## 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:

```
C:/Program Files/Synplicity/Synplify_dsp_lib/syndspclib*.mdl
```

2. In the MathWorks release, add this directory to your path:

```
matlabroot/toolbox/local/startup.m:  
addpath(fullfile(syndsproot, '..', 'Synplify_dsp_lib'));
```

3. In the M-generator for a Synplify DSP custom library block, use syndsplib to determine the current release of the library:

```
syn_lib=syndsplib('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 `syndspclib_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 Synplify DSP blocks whose parameters have changed in the latest release. You can access the parameters from the maske 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 `syndspclib`.

## 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(syndsproot','mathworks','toolbox','...',
    ''Synplicity','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('syndsplibv2/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('syndsplibv2/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('syndsplibv2/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('syndsplibv2/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('syndsplibv2/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('syndsplibv2/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 6

# Using M Control Blocks

---

The Synplify DSP M Control block provides a way to implement complex control-intensive functions using MATLAB's M language. The following provide more information about creating and using these blocks:

- [Using M Control Blocks, on page 6-2](#)
- [Tips for Designing with M Control Blocks, on page 6-4](#)
- [Coding M Control Blocks, on page 6-5](#)
- [M Language Support, on page 6-27](#)

# Using M Control Blocks

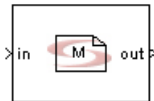
This section shows you how to incorporate an M Control block in your design and some tips on using M Control blocks.

- [Using M Control Blocks in Synplify DSP Designs, on page 6-2](#)
- [Tips for Designing with M Control Blocks, on page 6-4](#)

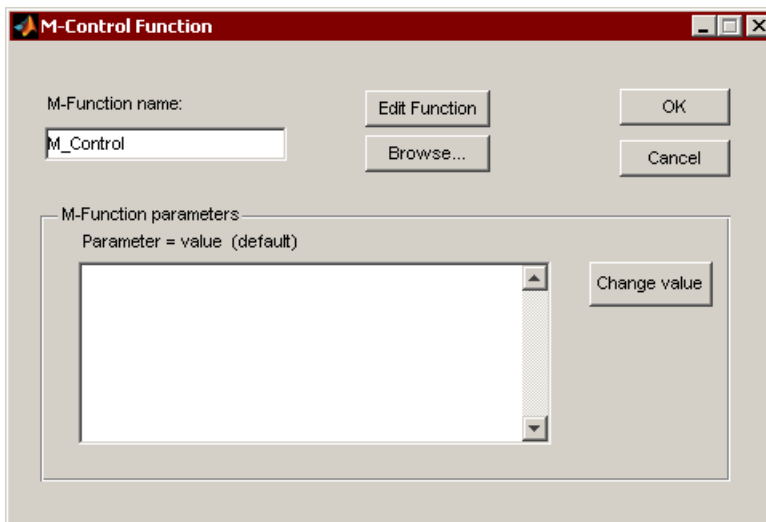
## Using M Control Blocks in Synplify DSP Designs

1. Select the M Control block from the Synplify DSP Control Logic library and instantiate it in your design.

See [Synplify DSP M Control, on page 8-154](#) for information about this block.



2. Specify the function for the block.
  - Double-click the block to open the M Control Function 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 6-27](#).
  - Save the file and close the window.
3. To write a new M function, do the following:
    - Enter a name for the function in the M Control Function dialog box.
    - Click Edit Function to open a MATLAB M file editor window.
    - Write the M function following the guidelines and caveats described in [Tips for Designing with M Control Blocks, on page 6-4](#), [Coding M Control Blocks, on page 6-5](#), and [M Language Support, on page 6-27](#).
    - 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 M Control Function 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.

# Tips for Designing with M Control Blocks

Follow these guidelines when designing with M Control blocks.

- M-Control blocks only support a single sample rate and all inputs must be fixed point.
- You must manually specify the precision of persistent variables used in accumulation-type operations. See [Precision Bounds for Persistent Variables, on page 6-14](#) and [Counters, on page 6-24](#) for more information.
- Use the features built into the M editor like syntax highlighting to step through your M code and identify problems.
- Data type propagation through the M code is full-precision up to a limit of 53 bits. Use quantize commands to control the precision and, if necessary, to limit precision to 53 bits or less. See [Data Type Restrictions, on page 6-9](#) and [Controlling Precision and Signedness with Quantizers, on page 6-10](#).
- When you want to create combinatorial logic, avoid incomplete assignments to outputs and program variables. For more information, refer to [Combinatorial Logic, on page 6-11](#).
- When you want to create state-holding elements, use persistent variables that are initialized to zero with the `isempty` function. See [States with Persistent Variables, on page 6-12](#).

For additional information about the extent of M syntax support, see [M Language Support, on page 6-27](#).

# Coding M Control Blocks

Specifying block behavior with a high-level, sequential programming style makes it easier to develop and debug. The Synplify DSP M Control block provides a way to implement complex control-intensive functions using the MATLAB M language (see [Synplify DSP M Control, on page 8-154](#)). 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 6-5](#)
- [M Control Block Data Types, on page 6-7](#)
- [Combinatorial Logic, on page 6-11](#)
- [States with Persistent Variables, on page 6-12](#)
- [Precision Bounds for Persistent Variables, on page 6-14](#)
- [State Machines, on page 6-16](#)
- [Counters, on page 6-24](#)
- [User-Defined Functions, on page 6-26](#)
- [Overridable Parameters, on page 6-27](#)

## Ports and Timing

You encapsulate the behavior of an M Control 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 M Control block. The named return values of the function define the physical output ports of the M Control 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)
```

The following is a function header for 3-input, 2-output design:

```
function [sum, cout] = adder(a, b, cin)
```

Currently you can only use scalar types for the top-level function arguments that define the input and output ports of the M Control block. 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 6-29](#) for a list of supported MATLAB built-in functions and [User-Defined Functions, on page 6-26](#) for information about user functions.

## Ports

As with all Synplify DSP blocks, the inputs of an M Control block receive signals which have a specific data type (word width and signedness) and a specific sample rate.

- **Sample rate**  
All M Control block inputs must have the same sample rate; you can not have a multi-rate M Control block.
- **Data type**  
The data type of each input can be in either signed 2's complement or unsigned fixed-point format. 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. Note that the software determines the data type of an input port from the system context in which the M Control 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.
- **Word width**  
The total word width of each input port signal must not exceed 53 bits.

## Timing

The simulation timing model for an M Control 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 Control Block Data Types

Each input, output, and internal variable in the M function defining an M Control 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, internal variables, inputs, and outputs must be real-valued scalars. Vector and matrix types and operations are not currently supported. Complex types and operations are not currently 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 the following:

- How input and output data types are assigned
  - [Input Data Type Assignment, on page 6-7](#)
  - [Output Data Type Assignment, on page 6-8](#)
- How data type information is propagated through the M-function to determine the precision and signedness of operations
  - [Internal Data Type Propagation, on page 6-8](#)
  - [Constant Data Type Assignment, on page 6-8](#)
  - [Data Type Restrictions, on page 6-9](#)
- How you can control the precision and signedness of operations
  - [Controlling Precision and Signedness with Quantizers, on page 6-10](#)

### Input Data Type Assignment

The tool determines the data types of primary inputs from the Simulink block diagram in which the M Control block is instantiated. Before compiling an M Control block, the tool determines the fixed-point data type of the signal driving each block input by performing data type propagation on the Simulink model enclosing the block. This means that input port data types are determined outside the M function that defines the block's behavior.

## Internal Data Type Propagation

The fixed-point data types of variables and operations defined in the function for the M Control block are determined by propagating data types from the primary inputs. This is analogous to the procedure used to propagate data types through a Simulink model. The result data type of 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 Control block attempts to follow a “full precision” model. The full precision of the result of each operation is preserved if possible. 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. Likewise, multiplying an N-bit input by an M-bit input will usually generate an N+M-bit result.

Operations such as add and multiply increase precision. Other operations naturally maintain the precision of their inputs. For example, a bitwise AND of two N-bit inputs produces an N-bit result. Finally, 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.

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 6-9](#).

## Output Data Type Assignment

The fixed-point data type of each M Control 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 M Control block outputs.

## 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 are not exactly representable with finite precision in base-2; for example the constant expression  $1/3$ . For these constants, the constant is represented using the same precision (53 bits) that would be used to represent it as a floating point number in IEEE double precision format. This much 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 6-10](#) for information on how to do this

## Data Type Restrictions

Simulation of an M Control block within Simulink is performed using an internally-generated S-Function which calls your original M function. By simulating your M code, you can interactively do a source-level debugging of the M Control 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 Synplify DSP 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 6-10](#).

### 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 Control block input in the Simulink model. To do this, either use a Convert block before the M Control block input or adjust the output data type of the block driving the signal.

## Controlling Precision and Signedness with Quantizers

M Control for Synplify DSP supports the use of quantizers, available in MATLAB's Fixed-Point Toolbox. Quantizers are useful in situations where you need to explicitly define the precision and signedness of a result. Applications include constraining internal operation widths to avoid simulation-synthesis mismatches and situations where the full precision of an operation, input, or constant is not required.

For detailed information about quantizers, refer to the MATLAB documentation for the Fixed-Point Toolbox. In brief, the user defines 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 in M Control blocks, see [Precision Bounds for Persistent Variables, on page 6-14](#).

The Synplify DSP M Control block supports a subset of the full quantizer functionality provided by the Fixed-Point Toolbox, as described in this table:

Quantizer Property Name	Quantizer Property Value	Synplify DSP Support
mode	'double'	No
	'float'	No
	'fixed'	Yes
	'ufixed'	Yes
	'single'	No
roundmode	'ceil'	No
	'convergent'	No
	'fix'	No
	'floor'	Yes
	'nearest'	Yes

Quantizer Property Name	Quantizer Property Value	Synplify DSP Support
overflowmode (fixed-point only)	'saturate'	Yes
	'wrap'	Yes
format	[wordlength fractionleng] 'fixed' or 'ufixed' only)	Yes
	[wordlength exponent length] ('float' mode)	No

## Combinatorial Logic

The simplest M Control block you can create 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. Most useful control functions such as state machines and counters require state-holding elements (see [States with Persistent Variables, on page 6-12](#) and [Counters, on page 6-24](#) for details).

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 Synplify DSP 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.

## States with Persistent Variables

In the M Control programming model, state-holding elements are inferred using persistent variables. A variable that is declared as “persistent” retains its value across function calls when an M function is executed as software. State-holding elements in hardware require this behavior, because they must retain their value across input sample times. Note that persistent variables are a part of the M programming language and are not specific to Synplify DSP. In the M Control programming model, you use them to implement an M Control block with states.

See the following for detailed information about coding with persistent variables:

- [Persistent Variables, on page 6-12](#)
- [Initialization with the isempty Construct, on page 6-13](#)
- [Persistent Variables That Do Not Result in State Registers, on page 6-13](#)
- [Precision Bounds for Persistent Variables, on page 6-14](#)

### Persistent Variables

The following M function describes an M Control block which computes parity (exclusive OR) over a window of three consecutive input samples:

```
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
```

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.

## 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 a Synplify DSP 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 M Control programming methodology.

## Persistent Variables That Do Not Result in 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 Control block. 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 are described:

- [Persistent Variable that Requires More Precision, on page 6-14](#)
- [Defining Precision with the Quantizer Object, on page 6-15](#)
- [Effect of Quantize Call Placement, on page 6-16](#)

### Persistent Variable that Requires More Precision

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.

## Defining Precision with the Quantizer Object

The following shows one way to modify 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).

```

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

```

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
end

```

```
    sum = sum + x;  
    res = sum;  
  
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 previous two examples in [Defining Precision with the Quantizer Object, on page 6-15](#). 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 information about the `quantize()` and `quantizer()`, see [Controlling Precision and Signedness with Quantizers, on page 6-10](#).

## State Machines

The M language makes it easy to define hardware state machines using the coding conventions described in [States with Persistent Variables, on page 6-12](#). 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
        end
    end
end
```

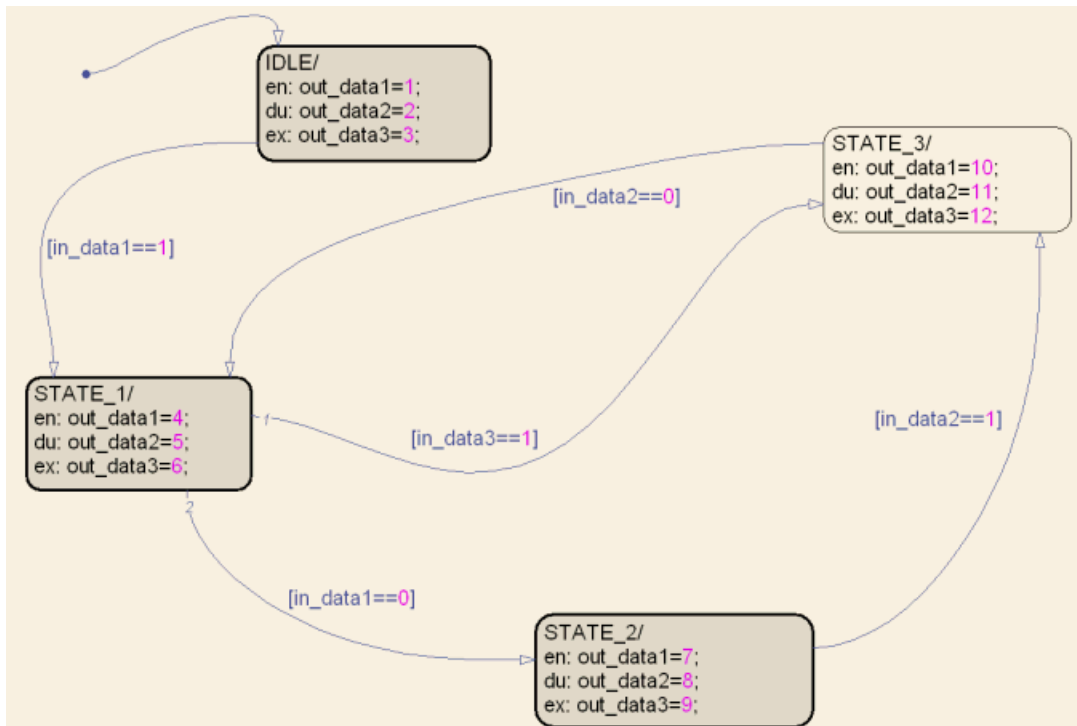
```

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
end

```

## State Flow Example

This example provides guidelines for translating 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
        end
    end
end
```

```

        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 currently 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 Synplify DSP 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. If you specify a one-hot encoding for states where all normal operating states have exactly one bit set in the state register, setting all state bits to zero places the system in an illegal state. Even if a zero state is defined, it still may not correspond to the desired start-up state.

- Where a zero-encoded state (as in one-hot encoding) is not a normal 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.
- 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 solely of the current state, 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 6-16](#)) 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;
    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.

Currently, the M compiler does not detect that the precision of the counter is bounded, 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 modified 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 6-16](#) and [Moore State Machine Example, on page 6-22](#) for coding examples, and [Local Resets, on page 6-21](#) 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

```

## User-Defined Functions

You can define other M functions in addition to the top-level M function which defines the M Control 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 the frequent calls to `quantizer` and `quantize` in user-defined functions, because you would probably 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 6-15](#). 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 any M Control 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-Control block using this function.
3. Select the parameter you want to override from the list, and click Change Value. Specify the override value you want to use.

The original value is shown in parentheses.

While specifying the override value, you can specify an expression which can be evaluated in the base workspace.

## M Language Support

The following describes the extent to which various features that are available in the M language are supported in the Synplify DSP tool.

- [Keywords, Variables, Functions, and Structures, on page 6-28](#)
- [Operator Support, on page 6-28](#)
- [Built-In Function Support, on page 6-29](#)
- [Synplify DSP Functions, on page 6-31](#)
- [M Language Limitations, on page 6-31](#)

## Keywords, Variables, Functions, and Structures

- **Keywords**  
Synplify DSP supports all M language keywords except for while and for. While and for loops are currently 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, scalar types. Currently, the tool does not support array (vector and matrix) variables and operations. It 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
Divide	/	Only when dividing by exact power of 2 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 Synplify DSP software supports the following built-in functions and constants. They have their usual semantics except that they are limited to scalar arguments, and as noted. 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 6-6](#).

Function	Description
bin2dec	<p>Argument must evaluate to a single string constant. For M-Control 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:</p> <pre>result = bin2dec(['101', '011']);</pre> <p>Here, the concatenation expression argument evaluates to the string '101011' and the function returns decimal value 43.</p> <p>The Synplify DSP implementation follows the MATLAB implementation, and allows white spaces in the input string and a length restriction of 52 digits for the input string.</p>
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.
bitxor	
ceil	Argument must be a constant.
fix	Argument must be a constant.
floor	
hex2dec	<p>Argument must evaluate to a single string constant. For M-Control 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.</p> <p>The Synplify DSP implementation follows the MATLAB implementation, and does not allow white spaces in the input string and has no length restrictions for the input string.</p>
isempty	May only be used to initialize persistent variables to zero.
logical	



Function	Description
<code>pi</code>	
<code>quantize</code>	
<code>quantizer</code>	Only supports fixed, unfixed, wrap, saturate, floor, and nearest. See <a href="#">Controlling Precision and Signedness with Quantizers</a> , on page 6-10.
<code>round</code>	

## Synplify DSP Functions

Synplify DSP includes the `syn_bitrev` function that reverses the bits of a specified integer. See [syn\\_bitrev](#), on page 9-2 for details.

## M Language Limitations

The following features are not supported currently:

- Vectors and matrices
- For and while loops
  - While loops, even unrollable while loops, are not supported, and result in error messages.
  - You can have a for loop if the loop body contains only constant-valued assignments to scalar or array variables. This makes it easier to create ROM-like structures by initializing vectors with constant values using for loops.

The loop body can assign to more than one scalar or array variable, but the right hand side of every assignment must evaluate to a constant once the loop is unrolled and constant propagation is performed. This permits rhs expressions which are functions of the loop index such as the following:

```
for i = 1:8
  x(i) = 2 * i; % OK because RHS is constant when
               loop is unrolled.
end
```

- Matrix variables and operations

In this case, matrix is defined as a 2-D array where the size of *both* array dimensions is greater than one. 2-D arrays with one or both dimension sizes equal to one are allowed (with some restrictions on operations that can be performed on vectors as explained in the following bullet.

- Most element-wise operations on vectors (including bitwise, arithmetic, relational) are not supported. Also, the inner product of two vectors is not supported.

You can still define, assign, and access vectors, either as whole vectors or as individual elements. You can concatenate and transpose vectors. However, you can not do element-wise operations on vectors (such as "+", ".\*", "./"). The inner product of two vectors is not supported. Either of these operations results in error messages:

```
"Element-wise '%s' operation on vectors not supported for
M-Control synthesis." (%s contains name of op)
```

```
"Matrix multiply and vector inner product not supported for
M-Control synthesis."
```

- 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`

In addition, M language support currently has the following limitations:

- Function arguments (input ports) must be real scalars.
- 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.
- Function recursion is not supported.

## CHAPTER 7

# Cosimulation with ModelSim

---

---

**Note:** This chapter of the Synplify DSP User Guide is obsolete and will be removed from the documentation in the next release. For cosimulation with ModelSim, use Synplify DSP Smart Black Box.

---

This chapter describes RTL cosimulation using Simulink and the ModelSim simulation technologies. It covers the following topics:

- [Overview of Cosimulation, on page 7-2](#)
- [About Cosimulation with ModelSim, on page 7-2](#)
- [Setup for Cosimulation, on page 7-4](#)
- [Cosimulation Flow, on page 7-5](#)
- [Cosimulation for RTL Regression, on page 7-6](#)

# Overview of Cosimulation

Cosimulation is a useful capability. ModelSim offers HDL and simulation capabilities for hardware modeling, and Simulink lets you simulate system-level designs and complex models. If you are using the Aldec simulator, make sure that the version includes the Simulink cosimulation interface.

Cosimulating the Synplify DSP RTL with a simulation tool offers the following advantages:

- You can regression-test the RTL generated with Synplify DSP
- You can implement a control function in RTL to drive a DSP function in Synplify DSP

The following sections describe cosimulation with the ModelSim tool, including the handling of issues like fixed-point data type and Simulink-defined sample time.

## About Cosimulation with ModelSim

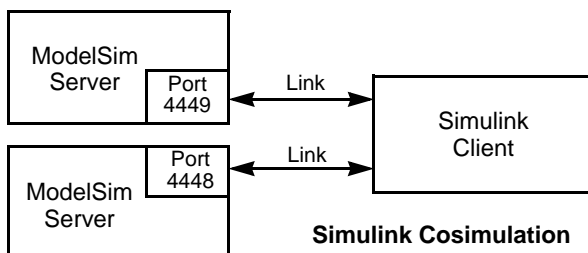
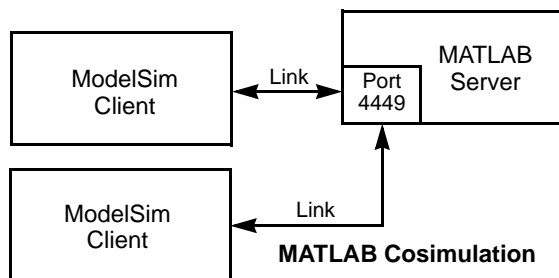
For cosimulation, the following tools need to work together:

- Synplify DSP
- Simulink
- ModelSim

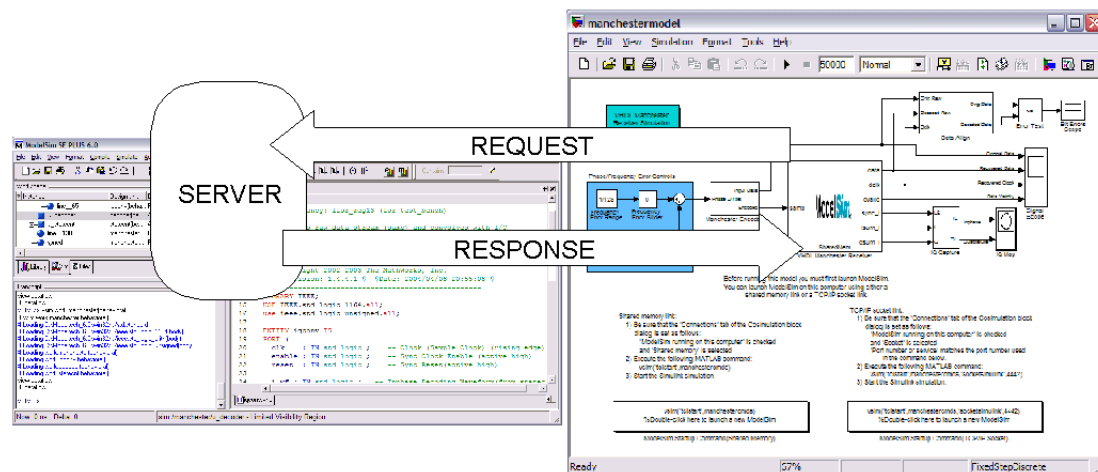
For Simulink and ModelSim to work together, you need the Link for ModelSim interface software, which is available from MATLAB and which sets up the proper relationships between ModelSim and the MathWorks products.

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 are different. For MATLAB cosimulation, MATLAB is the server and ModelSim is the client; for Simulink

cosimulation, ModelSim is the server and Simulink is the client. The second scenario is the one that is relevant to Synplify DSP cosimulation. The following figure illustrates the client-server relationships.



The next figure shows the Link for ModelSim communications interface, where a VHDL Cosimulation block co-simulates a hardware component by applying input signals to and reading output signals from a VHDL model under simulation in ModelSim. The VHDL Cosimulation block is the Simulink client to the ModelSim server.



# Setup for Cosimulation

The following sections describe what you need to set up the environment described in [About Cosimulation with ModelSim, on page 7-2](#).

## Software Requirements

Cosimulation requires that you have the following software:

- MathWorks R14 SP2
  - Simulink 6.2
  - Link for ModelSim 1.3.1. For information on installing and setting up the interface for Simulink, see the Link for ModelSim documentation. Note that Link for ModelSim only supports VHDL. To cosimulate Verilog, you must have a VHDL wrapper. See [Cosimulation for Verilog, on page 7-24](#) for details.
- ModelTech ModelSim SE 6.0e
- Synplify DSP 2.4

## Software Configuration

After you have properly installed the required software, do the following:

1. Configure ModelSim for use with Link for ModelSim by using the `configuremodelsim` MATLAB function.

The `configuremodelsim` function registers new MathWorks-related Tcl commands for the ModelSim simulator by creating the `...\tcl\Model-SimTclFunctionsForMATLAB.tcl` file within the ModelSim installation directory.

2. Configure Simulink for use with Synplify DSP by entering the `dspstartup` command at the MATLAB command line or by adding the command to the Simulink `startup.m` file.

Some of the settings created by `dspstartup` improve simulation performance, so it is a good idea to always specify it for Synplify DSP applications.

## Cosimulation Flow

The following steps give you an overview of the cosimulation flow:

1. Enter the `vsim` function on the command line in the MATLAB command window.

This function starts ModelSim so that it is ready for use with Simulink. If `modelsim` is not in the `PATH`, use the `vsimdir` option with `vsim`.

2. In the ModelSim interface, compile the design.
3. To simulate, type the `vsimulink` command.

This command loads an instance of your VHDL entity for cosimulation with Simulink. The `vsimulink` command is a Link for ModelSim variant of the ModelSim `vsim` command. It is made available as part of the ModelSim configuration, and starts the cosimulation server.

4. Switch back to Simulink and instantiate a VHDL Cosimulation block with the parameters set to manage the link between Simulink and ModelSim.

This block enables cosimulation by acting as a client during cosimulation.

5. Start the cosimulation from the Simulink window.
6. To shut down the simulation, do the following:
  - End the simulation in ModelSim.
  - Quit ModelSim.
  - Close the Simulink model.

## Cosimulation for RTL Regression

The following sections describe the process for taking the RTL generated by Synplify DSP and plugging it back into a Simulink-based test bench for regression. It covers the following:

- [Cosimulation for VHDL, on page 7-6](#)
- [Cosimulation for Multirate VHDL, on page 7-18](#)
- [Cosimulation for Verilog, on page 7-24](#)

## Cosimulation for VHDL

This example describes the process for taking the VHDL output generated by Synplify DSP and plugging it back into a Simulink-based test bench for regression. It describes the following steps of the process:

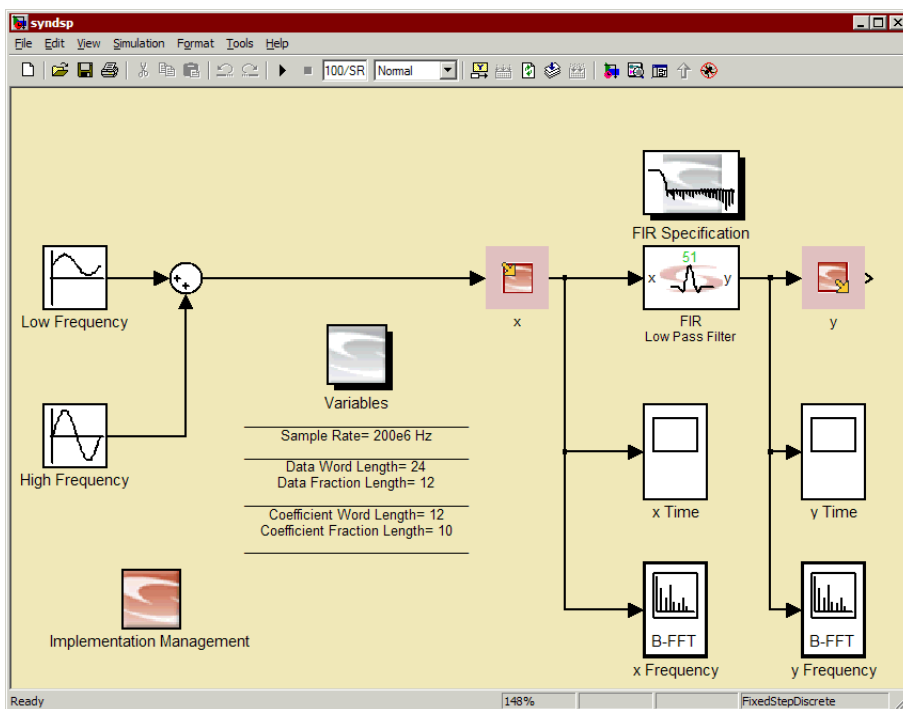
- [Generate Baseline RTL, on page 7-7](#)
- [Configure Software for Cosimulation, on page 7-8](#)
- [Set up the Cosimulation Block, on page 7-10](#)
- [Run Cosimulation, on page 7-14](#)

Some of the main issues for Synplify DSP are

- Handling sample time (invisible in Simulink, explicit clocks in RTL)
- Handling fixed point data type (explicit in Simulink, implicit in RTL)
- Handling global reset/enable (invisible in Simulink, explicit in RTL)



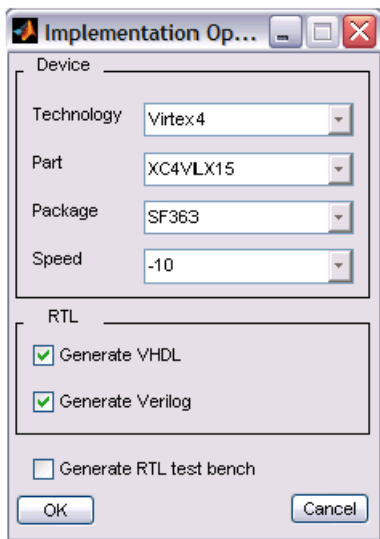
The example used to illustrate the process is the tutorial design. Assume that the design is stored in `SRC/syndsp.mdl`. The example uses an FIR filter to address the criteria to be solved (clock, reset, enable, data type, and sample time).



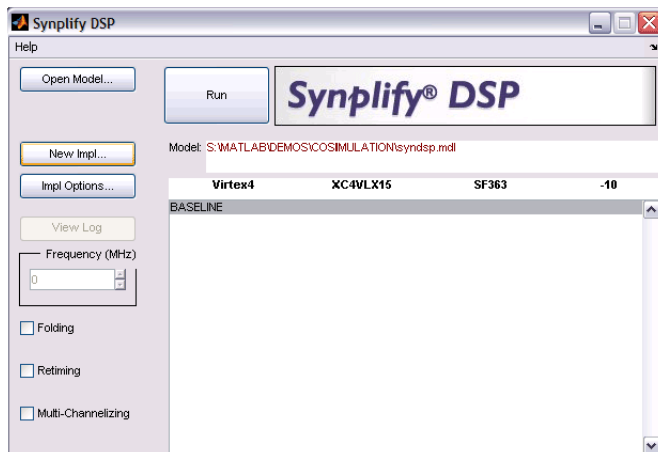
## Generate Baseline RTL

The first step is to generate baseline VHDL code in Synplify DSP. To do this, use the following procedure:

1. Set the targets in the Implementation Options dialog box.



2. Click Run to generate baseline RTL with Synplify DSP.



## Configure Software for Cosimulation

1. Create a MODELSIM directory in parallel with the SRC directory.

2. In this MODELSIM directory, start ModelSim from the Mathworks command line (to ensure that all the scripts for cosimulation are available):

```
vsim('vsimdir','C:\Modeltech_6.0\win32')
```

This starts ModelSim in the same directory.

3. Compile the design.

```
vlib work_vhdl
vmap work work_vhdl

# Copying C:\Modeltech_6.0\win32\../modelsim.ini to modelsim.ini
# Modifying modelsim.ini
# ** Warning: Copied C:\Modeltech_6.0\win32\../modelsim.ini
# to modelsim.ini.
#           Updated modelsim.ini.

vcom C:/Program\
Files/Synplicity/Synplify_dsp_22/lib/dsp/SynLib.vhd

# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08
# Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling package syndatatypes
# -- Loading package syndatatypes
# -- Compiling package synlibfunctions
# -- Compiling package body synlibfunctions
# -- Loading package synlibfunctions
...

vcom ../SRC/BASELINE/vhdl/syndsp.vhd

# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08
# Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Loading package syndatatypes
# -- Loading package synlibfunctions
# -- Compiling entity fir
# -- Compiling architecture behav of fir
# -- Compiling entity syndsp
# -- Compiling architecture structural of syndsp
```

4. Start the ModelSim cosimulation server from the ModelSim command line to provide any information required by Simulink:

```

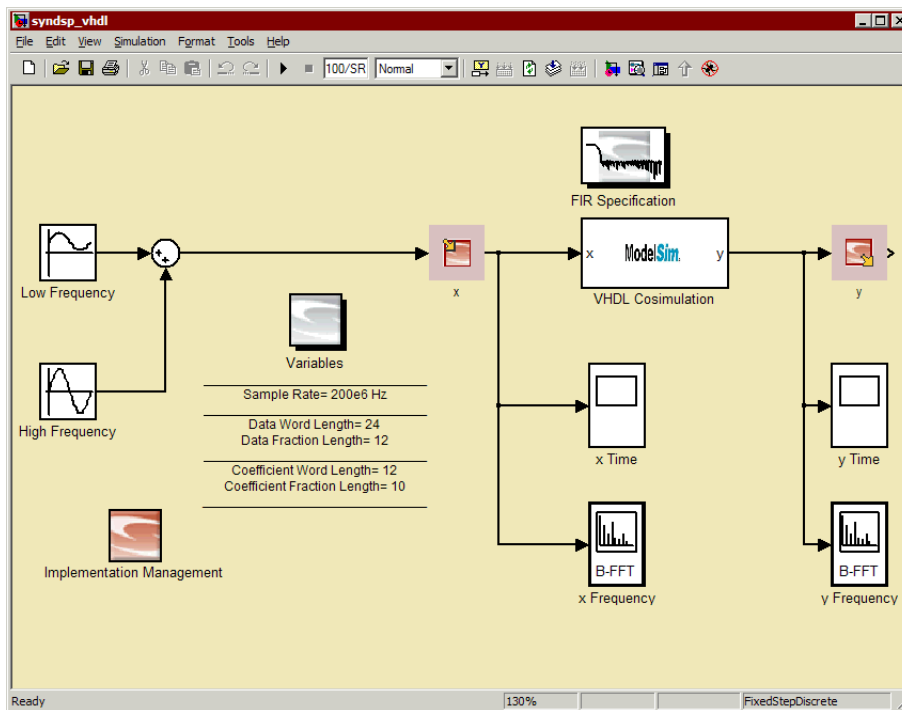
vsimulink syndsp

# vsim -foreign {simlinkserver
C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll} syndsp
# Loading C:\Modeltech_6.0\win32\../std.standard
# Loading C:\Modeltech_6.0\win32\../ieee.std_logic_1164(body)
# Loading C:\Modeltech_6.0\win32\../ieee.numeric_std(body)
# Loading work_vhdl.syndatatypes
# Loading work_vhdl.synlibfunctions(body)
# Loading work_vhdl.syndsp(structural)
# Loading work_vhdl.fir(bhavior)
# Loading C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll

```

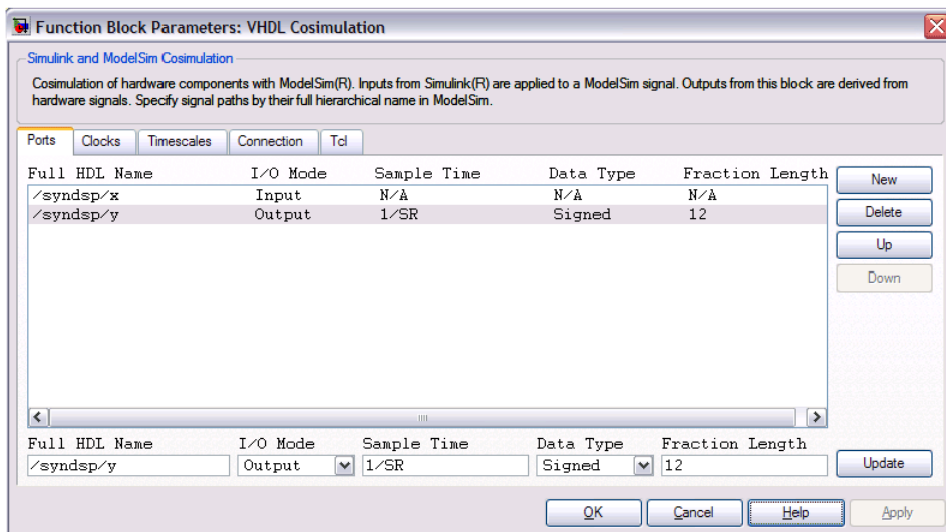
## Set up the Cosimulation Block

1. Copy SRC/syndsp.mdl to MODELSIM/syndsp\_vhdl.mdl and replace the Synplify DSP portion of the design with a VHDL Cosimulation instance from the Link for ModelSim library:

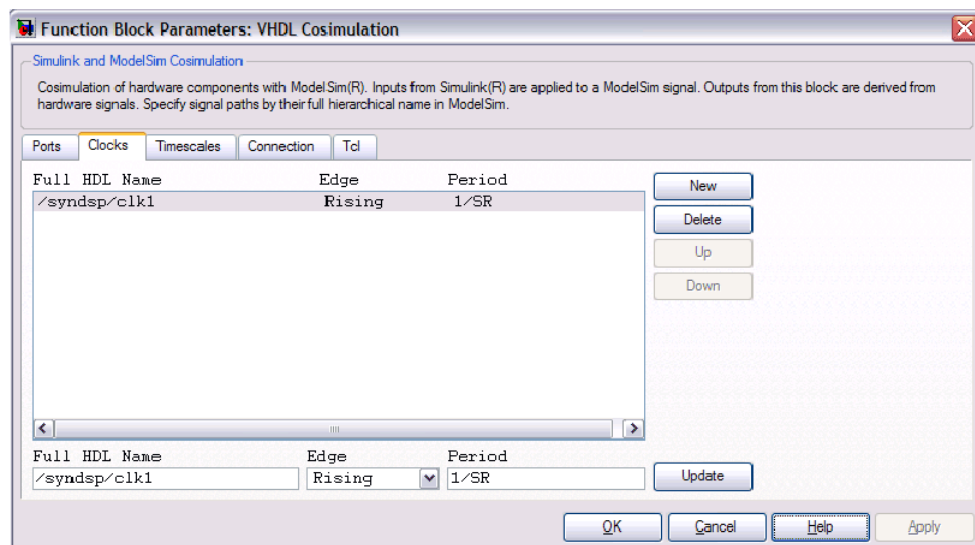


2. Double-click the VHDL Cosimulation block and set parameters for all the ports on the Port tab of the dialog box, following these guidelines:

- The inputs inherit the data type and sample rate
- The outputs must be fully specified for data type and sample rate

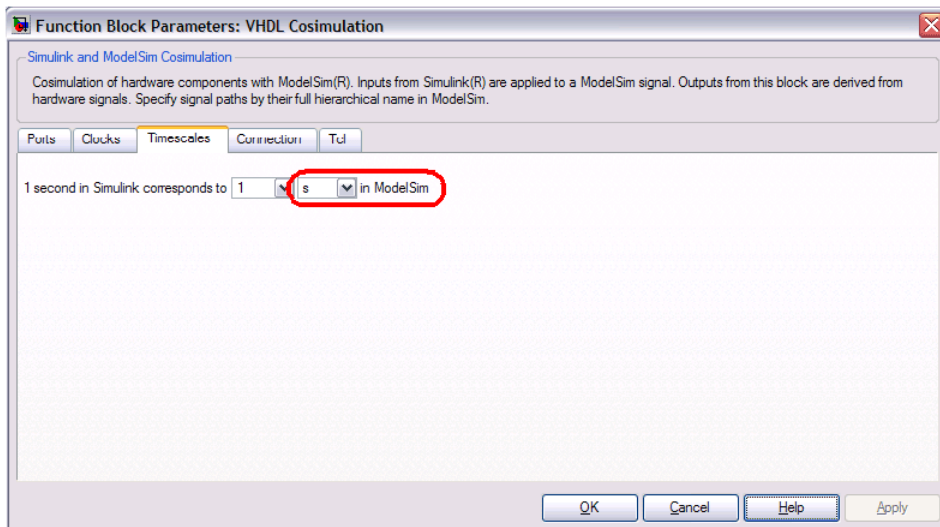


3. On the Clocks tab, specify the clocks with the appropriate sample rate:

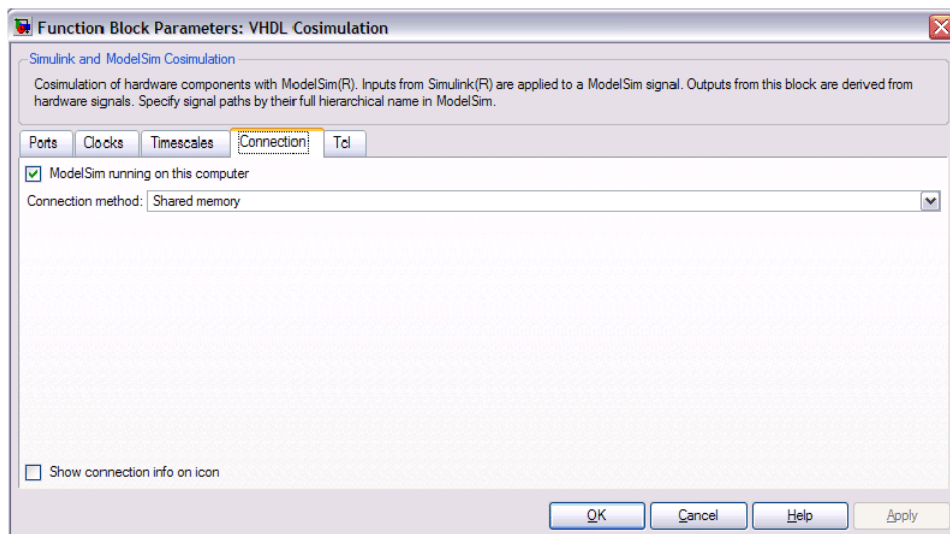


4. Go to the Timescales tab and set matching time units.

Synplify DSP imposes absolute timing information in Simulink. Setting this option synchronizes the ModelSim execution with the Simulink execution by matching timescales.



5. If both simulators are on the same machine, go to the Connection tab and select Shared memory to ensure the fastest performance for this computer setup.



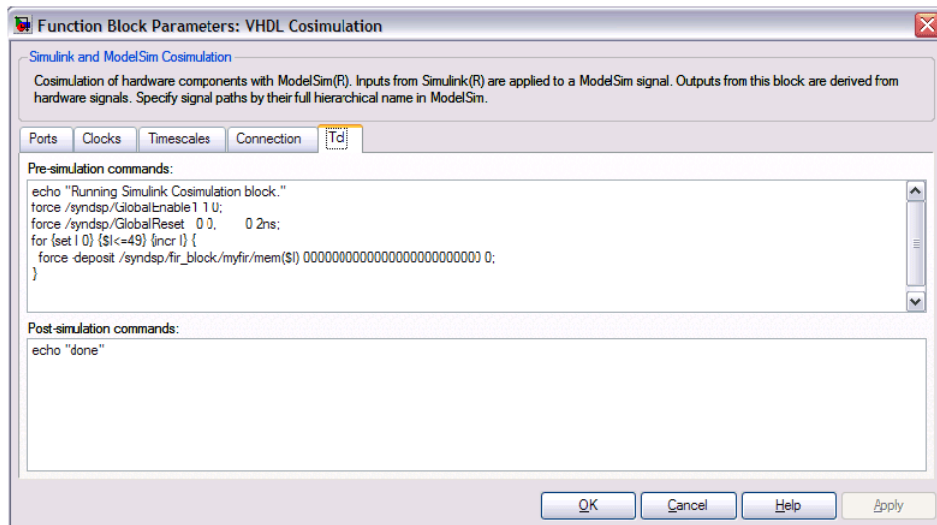
6. On the Tcl tab, use force commands to set the drivers for the global reset and enable lines.

- Specify an explicit reset cycle on the ports. This cycle corresponds to the hardware execution. The reset period must be shorter than the fastest clock:

```
force /syndsp/GlobalEnable1 1 0;
force /syndsp/GlobalReset 1 0, 0 2ns;
force /syndsp/clk1 0 0, 1 1ns, 0 2ns;
```

- Reset all the registers in the design. This reset provides full bit-true compatibility with the Simulink simulation:

```
force /syndsp/GlobalEnable1 1 0;
force /syndsp/GlobalReset 0 0;
for {set I 0} {$I<=49} {incr I} {
    force -deposit /syndsp/fir_block/myfir/mem($I)
    0000000000000000000000000000 0;
}
```



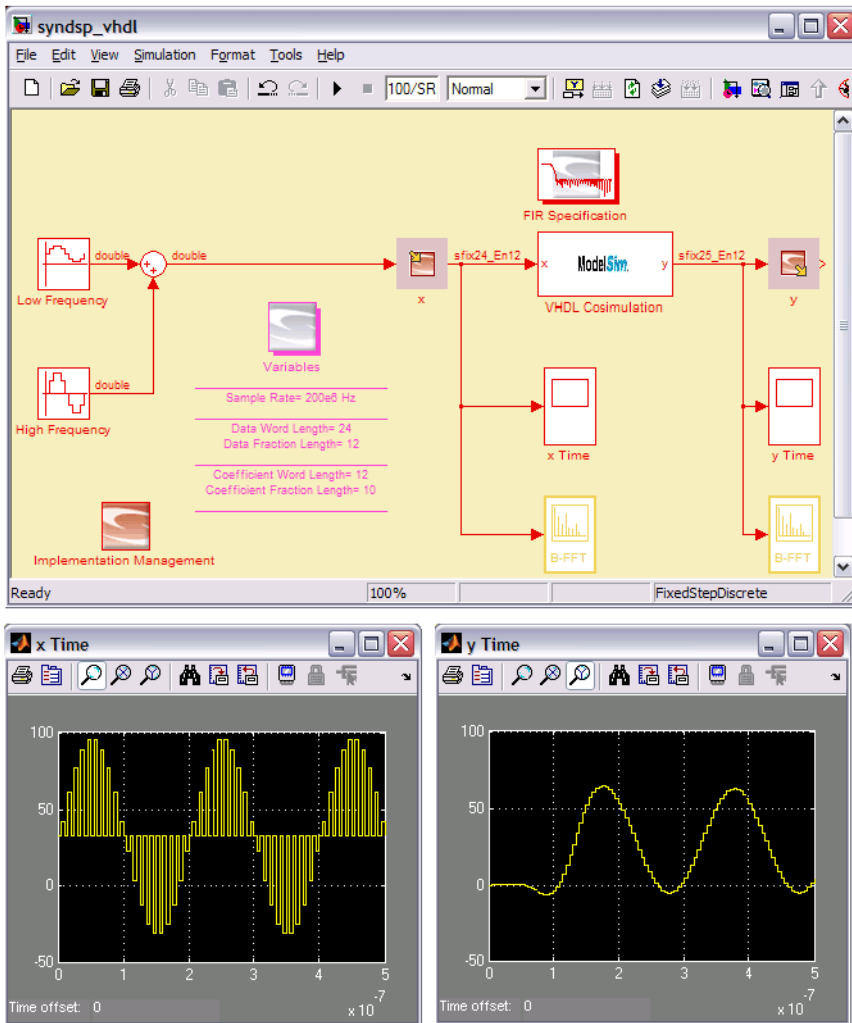
The global reset and enable lines are not visible at the Simulink level and must be driven appropriately for the simulation to work. Note that you cannot use any variables in the `force` commands; the commands run as is in ModelSim.

7. Click OK to apply the parameters you set.

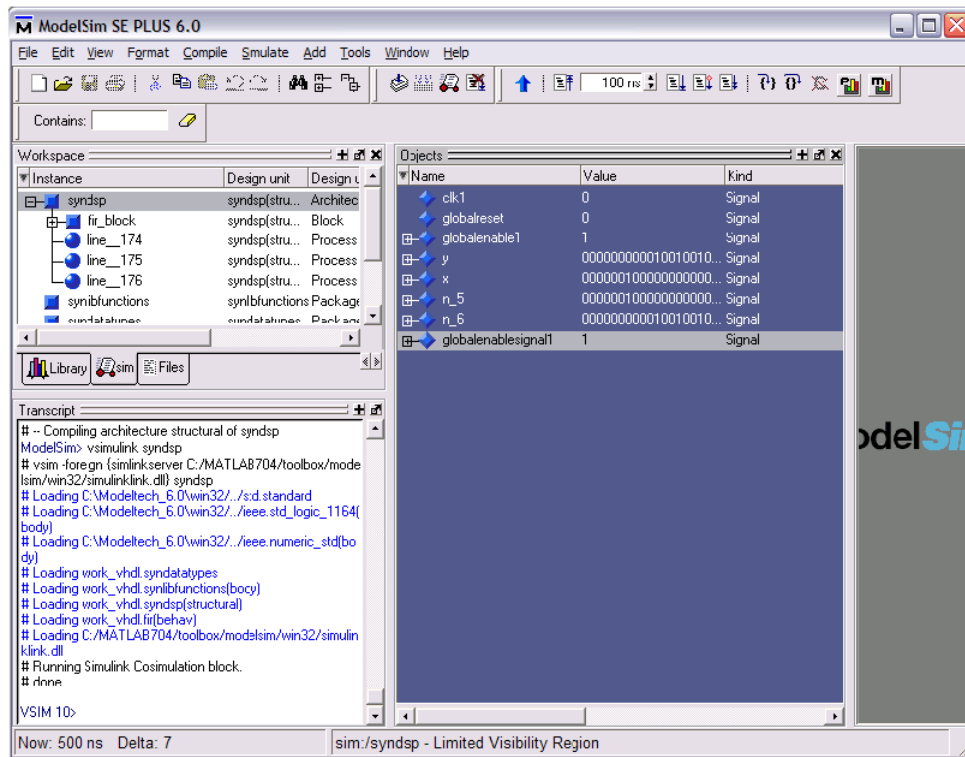
## Run Cosimulation

Connect the cosimulation instance and initiate the simulation from Simulink. The Simulink window shows both the sample times and data types with the simulation results.



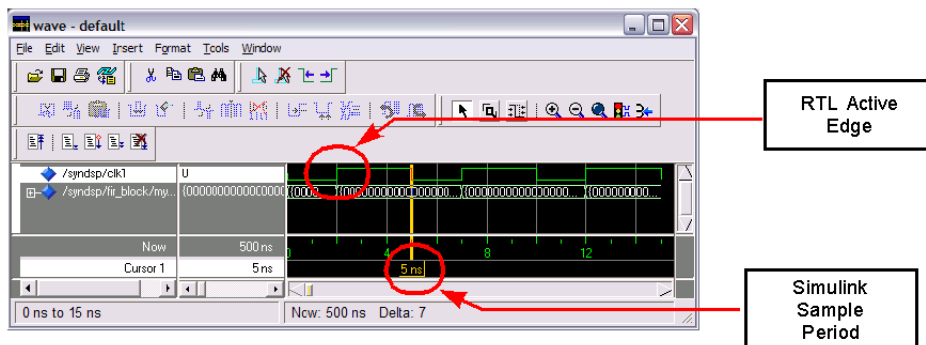


The ModelSim server displays the responses:

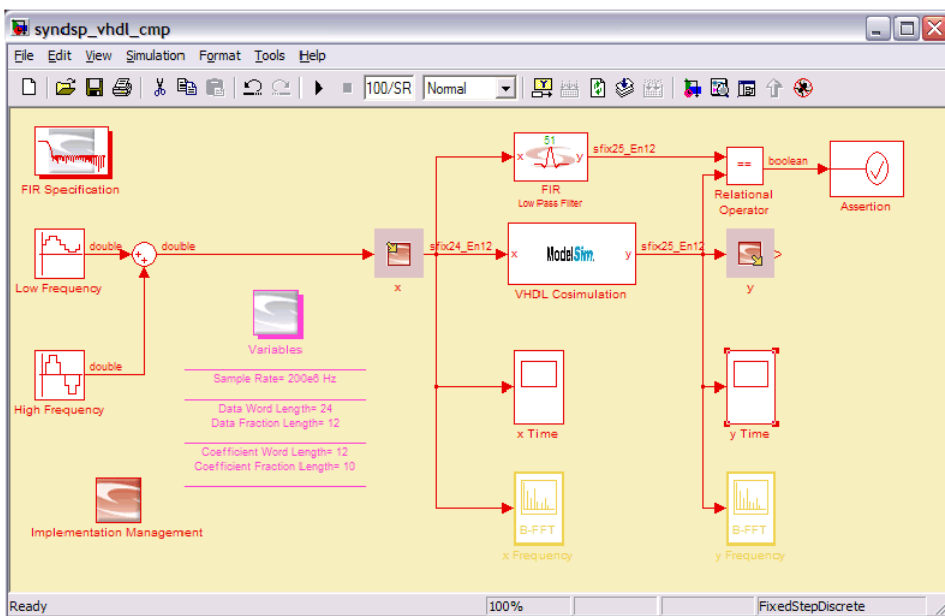


During simulation, analysis of the MTI waveform shows the following clock generated for this definition:

- Simulink uses newly sensed data at the Simulink sample period
- Simulink produces newly calculated data at the Simulink sample period
- RTL updates in between

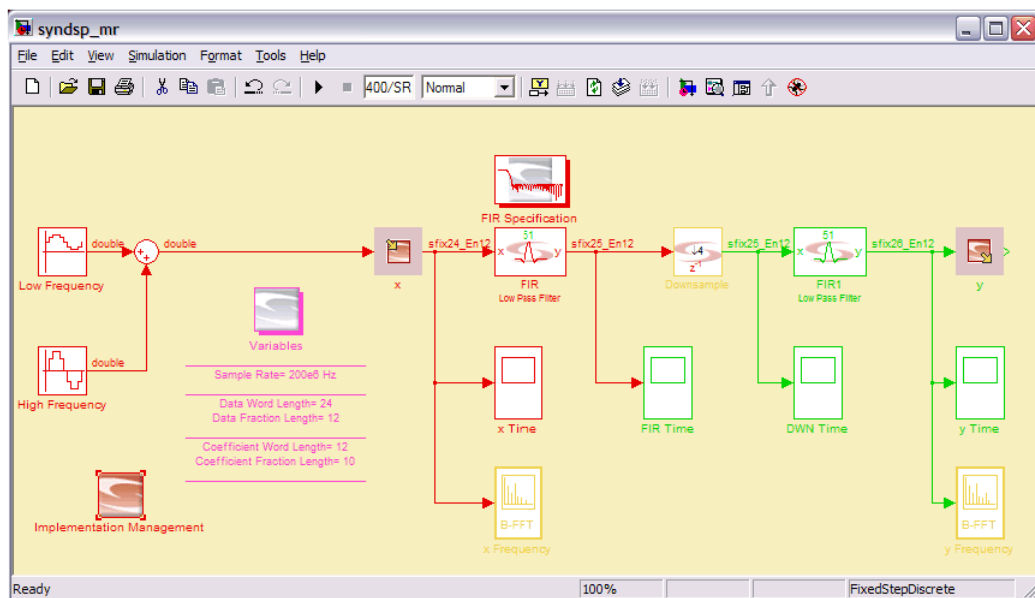


An assertion-based comparison also shows that the cosimulation is exact to the original Simulink simulation. The following figure shows the bit-true comparison.



## Cosimulation for Multirate VHDL

The original example can easily be modified to introduce multirate aspects, as shown in the following figure.



```
vlib work_mr_vhdl
vmap work work_mr_vhdl

# Modifying modelsim.ini
vcom C:/Program\
Files/Synplicity/Synplify_dsp_22/lib/dsp/SynLib.vhd

# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08
# Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling package syndatatypes
# -- Loading package syndatatypes
# -- Compiling package synlibfunctions
# -- Compiling package body synlibfunctions
# -- Loading package synlibfunctions
...
```

```
vcom ../SRC/MULTIRATE/vhdl/syndsp_mr.vhd
# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08
# Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Loading package syndatatypes
# -- Loading package synlibfunctions
# -- Compiling entity fir
# -- Compiling architecture behav of fir
# -- Compiling entity fir1
# -- Compiling architecture behav of fir1
# -- Compiling entity syndsp_mr
# -- Compiling architecture structural of syndsp_mr

vsimulink syndsp_mr
# vsim -foreign {simlinkserver
C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll} syndsp_mr
# Loading C:\Modeltech_6.0\win32\..\std.standard
# Loading C:\Modeltech_6.0\win32\..\ieee.std_logic_1164(body)
# Loading C:\Modeltech_6.0\win32\..\ieee.numeric_std(body)
# Loading work_mr_vhdl.syndatatypes
# Loading work_mr_vhdl.synlibfunctions(body)
# Loading work_mr_vhdl.syndsp_mr(structural)
# Loading work_mr_vhdl.fir1(behav)
# Loading work_mr_vhdl.fir(behav)
# Loading work_mr_vhdl.syndownsample(structural)
# Loading C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll
```

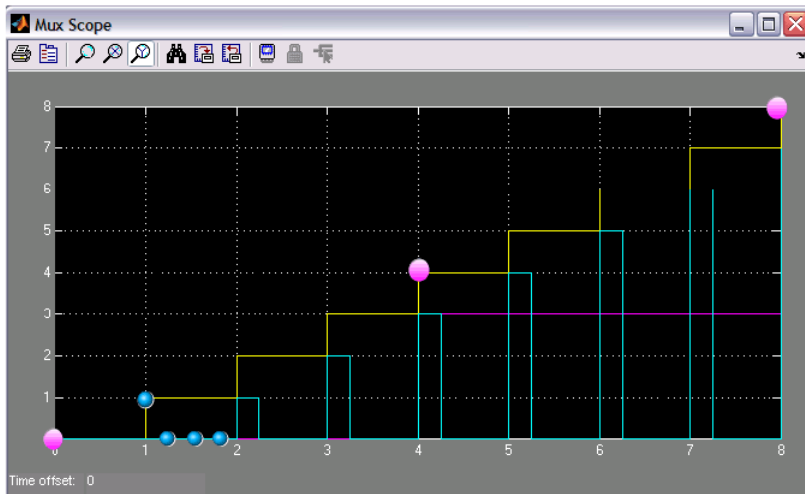
The following sections address cosimulation issues with clocks, resets, and enables.

## Clocks

Clock relationships are essential for maintaining DSP synchronization. Clock edges generated by the different tools are not compatible, and so clock generation must be explicitly defined. The following small test case illustrates the current clock relationships:

Synplify DSP does the following:

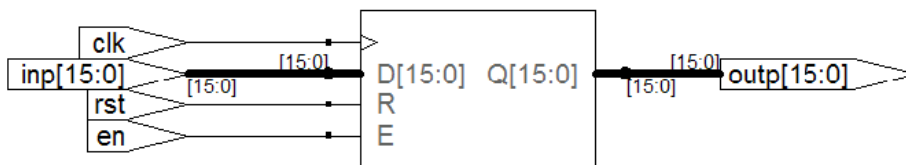
- Downsamples by taking the last sample in the frame defined by the highest clock and then delaying the sample data by this high clock. The following figure illustrates downsampling in Synplify DSP.



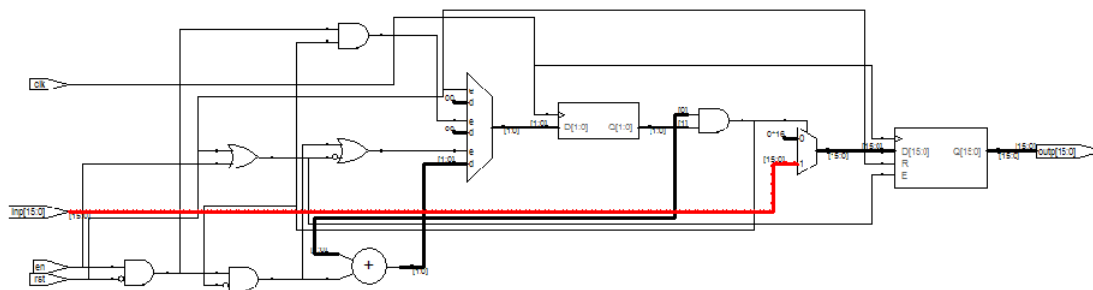
- Upsamples by delaying with the lower clock and then taking the first sample in the frame and creating new samples by inserting zeroes after that.

This choice is driven by the hardware:

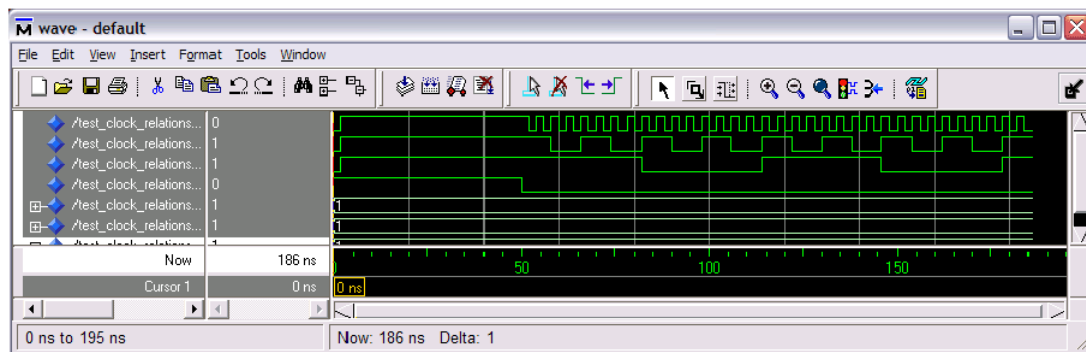
- Downsample: a register at the lowest clock clocks the data from the last slot of the highest clock and is inactive for the other slots.



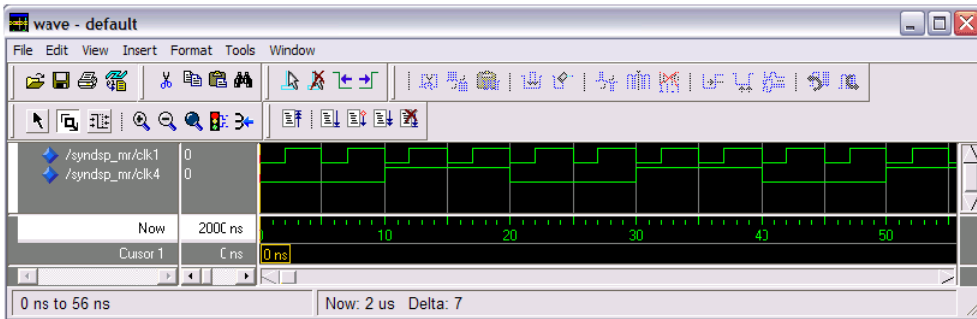
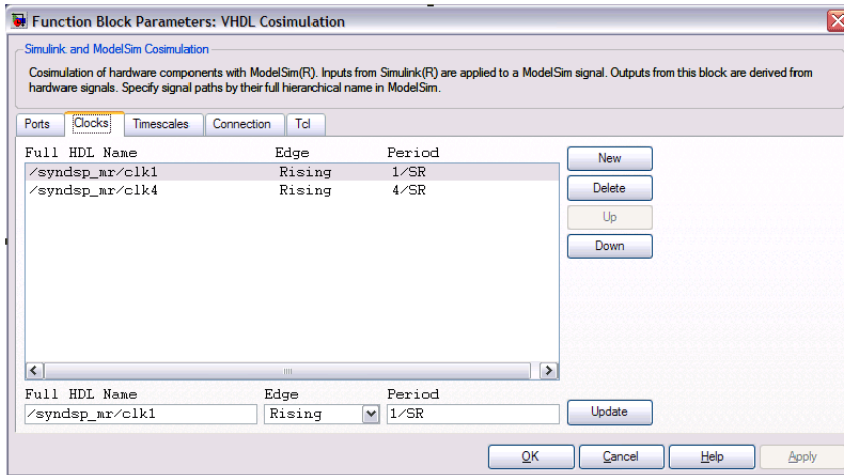
- Upsample: a counter at the highest clock multiplexes out the data in the last slot of the highest clock and inserts zero for the other slots. The result is delayed by one clock cycle at the highest clock.



The hardware in turn, determines the clocks required:

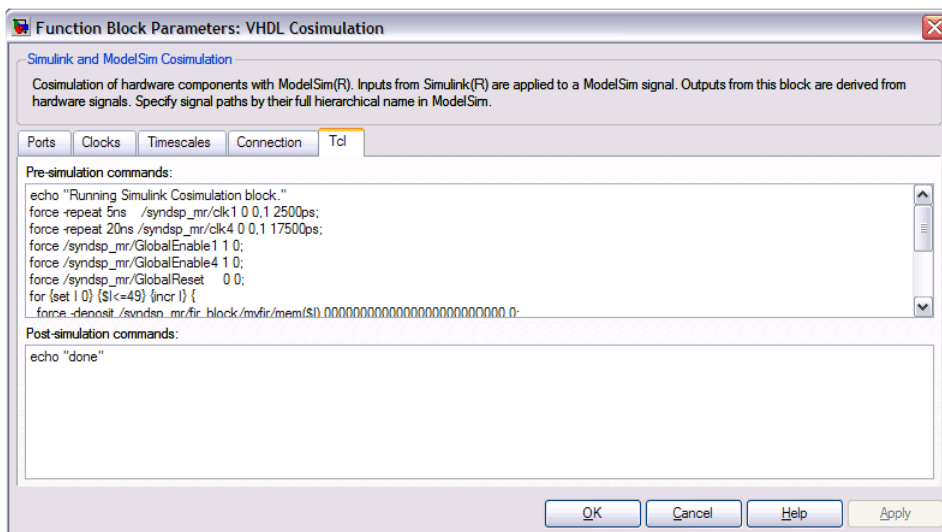


The default clock edges generated by the clock tab are not compatible with the Synplify DSP execution of multirate, nor are they compatible with the standard DSP multirate model that syncs up at time zero:



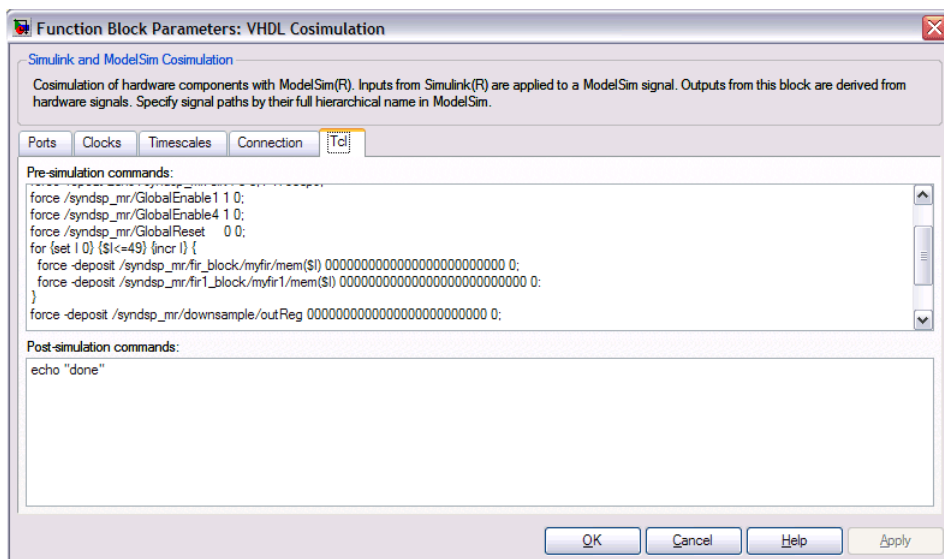
Consequently, the clock generation for Link for ModelSim can not be defined with the built-in mechanism and must be done with separate commands on the Tcl tab, as shown in this figure:





## Reset/Enable

A reset must occur at time 0 even when all enables are tied high (See “*Set up the Cosimulation Block*” on page 10). You explicitly define this with commands on the Tcl tab.



## Cosimulation for Verilog

The flow for a Verilog module is similar to the VHDL flow described in “*Cosimulation for VHDL*” on page 6, but you need a VHDL wrapper. The following lists the steps that are different.

1. Set up the directories and compile the design as described in “*Configure Software for Cosimulation*” on page 8.

```
vlib work_vlog
vmap work work_vlog

# Modifying modelsim.ini
vlog C:/Program\
Files/Synplicity/Synplify_dsp_22/lib/dsp/SynLib.v
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08
# Aug 19 2004
...

vlog ../SRC/BASELINE/verilog/syndsp.v
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08
# Aug 19 2004
# -- Compiling module FIR
# -- Compiling module syndsp
#
# Top level modules:
# syndsp
```

2. Load the Verilog module with the vsim command.

```
vsim syndsp
# vsim syndsp
# Loading work_vlog.syndsp
# Loading work_vlog.FIR
```

3. Generate a VHDL wrapper for the Verilog code with the wrapverilog command.

```
wrapverilog syndsp
# Model Technology ModelSim SE vcom 6.0 Compiler 2004.08
# Aug 19 2004
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Compiling entity syndsp_wrap
# -- Compiling architecture rtl of syndsp_wrap
# -- Loading package vl_types
# -- Loading entity syndsp
```

## 4. Run cosimulation on the Verilog wrapper you created.

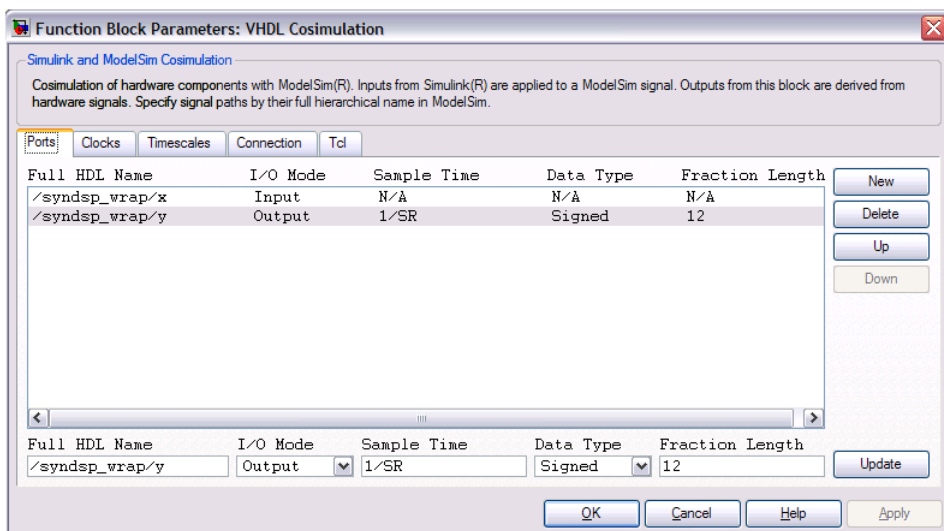
```

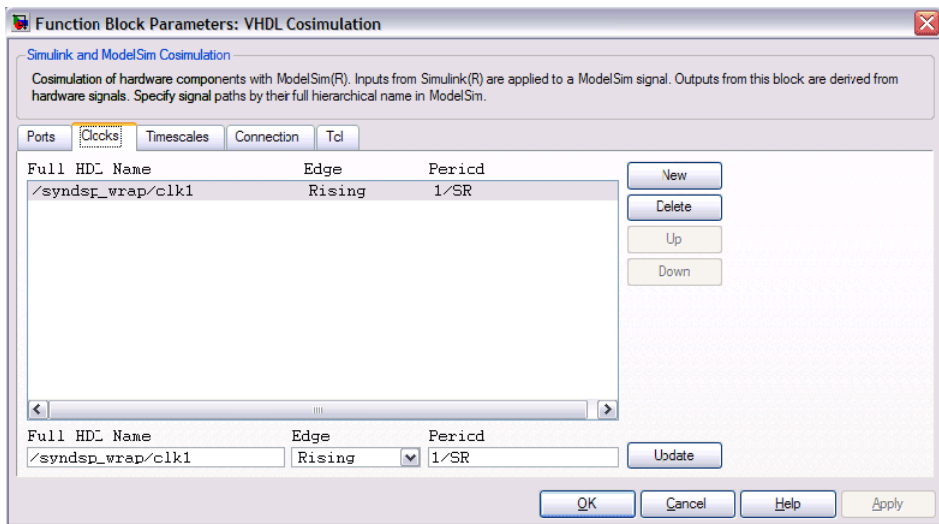
vsimulink syndsp_wrap
# vsim -foreign {simlinkserver
C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll} syndsp_wrap
# Loading C:\Modeltech_6.0\win32\../std.standard
# Loading C:\Modeltech_6.0\win32\../ieee.std_logic_1164(body)
# Loading C:\Modeltech_6.0\win32\../verilog.vl_types(body)
# Loading work_vlog.syndsp_wrap(rtl)
# Loading work_vlog.syndsp
# Loading work_vlog.FIR
# Loading C:/MATLAB704/toolbox/modelsim/win32/simulinklink.dll

```

5. Adjust all the signal names in the Cosimulation block to the new name *designName\_wrap/signalName*.

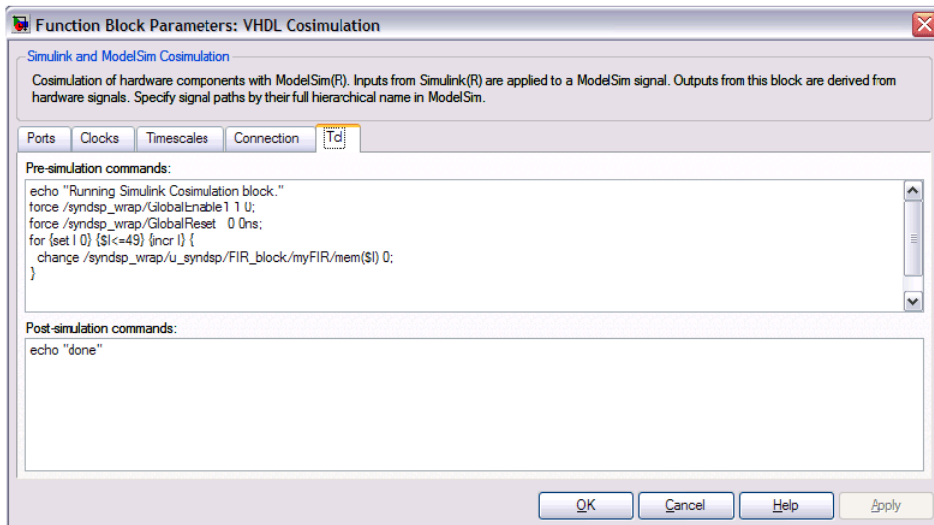
- In the MODELSIM directory, copy `syndsp_vhdl.mdl` to `syndsp_vlog.mdl`
- Make the following wrapper-based name changes to the cosimulation instance:





- On the Tcl tab, use the change command instead of the force command to specify the internal registers:

```
for {set I 0} {$I<=49} {incr I} {
    change syndsp_wrap/u_syndsp/FIR_block/myFIR/mem($I) 0;
}
```



## CHAPTER 8

# Synplify DSP Blockset

---

This chapter describes the Synplify DSP blocks and the Synplify DSP custom blocks, categorizing them by library and alphabetically. See the following:

- [Blocks — By Library, on page 8-2](#)
- [Blocks — Alphabetical List, on page 8-11](#)

Note the following:

- The Synplify DSP library includes some toolboxes at the top level: SynCoSimTool, SynDSPTool and SynFixPtTool. They are documented in this chapter, along with the other blocks.
- Some Synplify DSP blocks are classified as custom blocks. For details, and a list of the custom blocks, see [Primitives and Custom Blocks, on page 5-2](#).
- 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 A-1](#) contains a quick reference list of parameters like saturation and word length for different blocks.

# Blocks — By Library

The Synplify DSP 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 8-11](#)).

<a href="#">Communications</a>	Contains blocks specific to the communications industry.
<a href="#">Control Logic</a>	Contains blocks that implement logic for controlling datapaths.
<a href="#">CORDIC</a>	Contains blocks for specialized CORDIC math operations.
<a href="#">DSP Basics</a>	Contains fundamental blocks used for most DSP functions.
<a href="#">Filtering</a>	Contains blocks for designing and implementing filters.
<a href="#">Math Functions</a>	Contains blocks for specialized math operations.
<a href="#">Memories</a>	Contains blocks for memory structures like RAMs and FIFOs.
<a href="#">Ports &amp; Subsystems</a>	Contains port and black box blocks.
<a href="#">Signal Operations</a>	Contains blocks for the manipulation of signals.
<a href="#">Sources</a>	Contains blocks that generate constants and counters.
<a href="#">Transforms</a>	Contains blocks for transforms that are important to DSP operations.
<a href="#">Synplify DSP SynCoSimTool</a>	Specialized toolbox that manages the cosimulation interface between the smart black boxes in the design and ModelSim.
<a href="#">Synplify DSP SynDSPTool</a>	Specialized toolbox that controls the generation of RTL for synthesis.
<a href="#">Synplify DSP SynFixPtTool</a>	Specialized toolbox that opens the Simulink fixed point interface.

## Communications



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

Synplify DSP Block Deinterleaver	Reshuffles a fixed number of interleaved input symbols to obtain the original sequence.
Synplify DSP Block Interleaver	Shuffles a fixed number of input symbols to a new permutation.
Synplify DSP Convolutional Deinterleaver	Reshuffles streaming input symbols according a to a predefined mapping scheme.
Synplify DSP Convolutional Encoder	Corrects feed-forward errors using k/n convolutional codes.
Synplify DSP Convolutional Interleaver	Shuffles streaming input symbols to a new permutation, using a predefined mapping scheme.
Synplify DSP Depuncture	Removes user-specified symbols from the input data stream and replaces them with zeroes.
Synplify DSP Puncture	Removes user-specified bits from the input data stream
Synplify DSP Reed-Solomon Decoder	Decodes the encoded signal using Reed-Solomon error-correcting codes.
Synplify DSP Reed-Solomon Encoder	Generates an encoded signal, using Reed-Solomon codes.
Synplify DSP Viterbi Decoder	Decodes convolutionally encoded input data.

## Control Logic



This library contains blocks that provide control logic for outputs.

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

## CORDIC



This library contains blocks for specialized CORDIC math operations.

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



## DSP Basics



This library contains blocks for basic DSP operations.

<a href="#">Synplify DSP Add</a>	Implements a full-precision signed adder or subtractor.
<a href="#">Synplify DSP Delay</a>	Delays the input by the specified number of sample clock cycles.
<a href="#">Synplify DSP Gain</a>	Implements a constant gain to the input.

## Filtering



This library contains blocks for designing and implementing filters.

<a href="#">Synplify DSP CIC</a>	Custom block that implements a CIC filter.
<a href="#">Synplify DSP Differentiator</a>	Custom block that performs a discrete time differentiation of the input signal.
<a href="#">Synplify DSP FDATool</a>	Opens the Simulink FDATool interface.
<a href="#">Synplify DSP FIR</a>	Implements a finite impulse response (FIR) filter.
<a href="#">Synplify DSP FIR Engine</a>	Implements a finite impulse response (FIR) filter that uses the coefficients as vector input.
<a href="#">Synplify DSP FIR Rate Converter</a>	Implements a polyphase FIR filter.
<a href="#">Synplify DSP IIR</a>	Implements an infinite impulse response (IIR) filter.
<a href="#">Synplify DSP Integrator</a>	Performs a discrete time integration of the input signal.
<a href="#">Synplify DSP RFIR</a>	Custom block that implements a reloadable finite impulse response FIR filter.

## Math Functions



This library contains blocks for specialized math operations.

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

## Memories



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

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

## Ports & Subsystems



This library contains port and black box blocks.

Synplify DSP Black Box	Provides a way to embed other blocks.
Synplify DSP In	Provides a way to add an in port to a subsystem
Synplify DSP Out	Provides a way to add an out port to a subsystem
Synplify DSP Port In	Defines the input boundaries for the DSP design to be implemented in RTL.

Synplify DSP Port Out	Defines the output boundaries for the DSP design to be implemented in RTL.
Synplify DSP Smart Black Box	Lets you embed third-party IP in a Synplify DSP design.
Synplify DSP Subsystem	Allows you to add a subsystem to a Synplify DSP design.

## Signal Operations



This library contains blocks for the management of signals.

Synplify DSP Commutator	Sequentially switches the data from the specified number of input ports to a single output port.
Synplify DSP Concat	Concatenates the bits of up to 32 input signals.
Synplify DSP 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.
Synplify DSP Decommutator	Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly.
Synplify DSP Demux	Implements a de-multiplexer of up to 32 outputs with a latency of one sample.
Synplify DSP Downsample	Decreases the sample rate of the input by removing samples.
Synplify DSP Extract	Extracts specified bits from the input signal.
Synplify DSP Mux	Implements a multiplexer of up to 32 inputs.
Synplify DSP 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.
Synplify DSP 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

Synplify DSP 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.
Synplify DSP Upsample	Increases the sample rate of the input by inserting zeroes.
Synplify DSP Vector Concat	Constructs vectors by bundling up to 32 inputs together.
Synplify DSP Vector Expand	Converts scalar input to vector output.
Synplify DSP Vector Extract	Extracts selected ports for the output.
Synplify DSP Vector Split	Implements a de-multiplexer of up to 32 outputs.

## Sources



This library contains blocks that generate constants and counters.

Synplify DSP Constant	Implements a source with a constant value.
Synplify DSP Counter	Implements a resettable modulo counter with enable.
Synplify DSP DDS	Custom block that creates a direct digital synthesizer with sin and cos waves based on frequency, phase settings, and modulations.
Synplify DSP Ramp	Custom block that creates a ramp based on increments derived from a port or parameter
Synplify DSP Random	Custom block that creates a random integer of the requested word length.
Synplify DSP Sequence	Custom block that repeats a sequence of specified data

## Transforms



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

[Synplify DSP FFT](#)      Implements a fully pipelined Fast Fourier Transform.

---

# Blocks — Alphabetical List

The following list of blocks includes the Synplify DSP toolboxes, as well as the Synplify DSP blocks and custom blocks:

- [Synplify DSP Abs, on page 8-15](#)
- [Synplify DSP Accumulator, on page 8-17](#)
- [Synplify DSP Add, on page 8-19](#)
- [Synplify DSP Binary Logic, on page 8-22](#)
- [Synplify DSP Black Box, on page 8-25](#)
- [Synplify DSP Block Deinterleaver, on page 8-31](#)
- [Synplify DSP Block Interleaver, on page 8-33](#)
- [Synplify DSP CIC, on page 8-35](#)
- [Synplify DSP Commutator, on page 8-39](#)
- [Synplify DSP Comparator, on page 8-41](#)
- [Synplify DSP Concat, on page 8-43](#)
- [Synplify DSP Constant, on page 8-45](#)
- [Synplify DSP Convert, on page 8-48](#)
- [Synplify DSP Convolutional Deinterleaver, on page 8-53](#)
- [Synplify DSP Convolutional Encoder, on page 8-56](#)
- [Synplify DSP Convolutional Interleaver, on page 8-59](#)
- [Synplify DSP CORDIC Exp, on page 8-61](#)
- [Synplify DSP CORDIC Log, on page 8-63](#)
- [Synplify DSP CORDIC Polar, on page 8-65](#)
- [Synplify DSP CORDIC Rotator, on page 8-67](#)
- [Synplify DSP CORDIC SinCos, on page 8-74](#)
- [Synplify DSP CORDIC Sqrt, on page 8-76](#)
- [Synplify DSP Counter, on page 8-77](#)

- [Synplify DSP DDS, on page 8-83](#)
- [Synplify DSP Decommulator, on page 8-89](#)
- [Synplify DSP Delay, on page 8-91](#)
- [Synplify DSP Demux, on page 8-92](#)
- [Synplify DSP Depuncture, on page 8-94](#)
- [Synplify DSP Differentiator, on page 8-96](#)
- [Synplify DSP DivMod, on page 8-99](#)
- [Synplify DSP Downsample, on page 8-106](#)
- [Synplify DSP Extract, on page 8-109](#)
- [Synplify DSP FDATool, on page 8-111](#)
- [Synplify DSP FFT, on page 8-112](#)
- [Synplify DSP FIFO, on page 8-118](#)
- [Synplify DSP FIR, on page 8-120](#)
- [Synplify DSP FIR Engine, on page 8-127](#)
- [Synplify DSP FIR Rate Converter, on page 8-133](#)
- [Synplify DSP Gain, on page 8-138](#)
- [Synplify DSP IIR, on page 8-141](#)
- [Synplify DSP In, on page 8-146](#)
- [Synplify DSP Integrator, on page 8-147](#)
- [Synplify DSP Inverter, on page 8-151](#)
- [Synplify DSP Log, on page 8-152](#)
- [Synplify DSP M Control, on page 8-154](#)
- [Synplify DSP Mealy State Machine, on page 8-157](#)
- [Synplify DSP MinMax, on page 8-160](#)
- [Synplify DSP Moore State Machine, on page 8-162](#)
- [Synplify DSP Mult, on page 8-164](#)
- [Synplify DSP Mux, on page 8-166](#)



- [Synplify DSP Negate](#), on page 8-168
- [Synplify DSP Out](#), on page 8-169
- [Synplify DSP Parallel to Serial](#), on page 8-170
- [Synplify DSP Permutation](#), on page 8-172
- [Synplify DSP Port In](#), on page 8-174
- [Synplify DSP Port Out](#), on page 8-177
- [Synplify DSP Pow](#), on page 8-179
- [Synplify DSP Puncture](#), on page 8-181
- [Synplify DSP RAM](#), on page 8-183
- [Synplify DSP Ramp](#), on page 8-191
- [Synplify DSP Random](#), on page 8-194
- [Synplify DSP Recast](#), on page 8-196
- [Synplify DSP Reed-Solomon Decoder](#), on page 8-200
- [Synplify DSP Reed-Solomon Encoder](#), on page 8-207
- [Synplify DSP Register](#), on page 8-211
- [Synplify DSP RFIR](#), on page 8-213
- [Synplify DSP ROM](#), on page 8-219
- [Synplify DSP Sequence](#), on page 8-222
- [Synplify DSP Serial to Parallel](#), on page 8-224
- [Synplify DSP Shift Register](#), on page 8-227
- [Synplify DSP Shifter](#), on page 8-232
- [Synplify DSP Sign](#), on page 8-234
- [Synplify DSP SynCoSimTool](#), on page 8-249
- [Synplify DSP SinCos](#), on page 8-236
- [Synplify DSP Smart Black Box](#), on page 8-238
- [Synplify DSP Sqrt](#), on page 8-245
- [Synplify DSP Subsystem](#), on page 8-248

- [Synplify DSP SynDSPTool](#), on page 8-253
- [Synplify DSP SynFixPtTool](#), on page 8-261
- [Synplify DSP Upsample](#), on page 8-263
- [Synplify DSP Vector Concat](#), on page 8-267
- [Synplify DSP Vector Expand](#), on page 8-273
- [Synplify DSP Vector Extract](#), on page 8-275
- [Synplify DSP Vector Split](#), on page 8-277
- [Synplify DSP Viterbi Decoder](#), on page 8-279

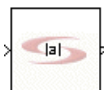
# Synplify DSP Abs

Calculates the absolute value of the scalar or vector input.

## Library

Synplify DSP [Math Functions](#)

## Description

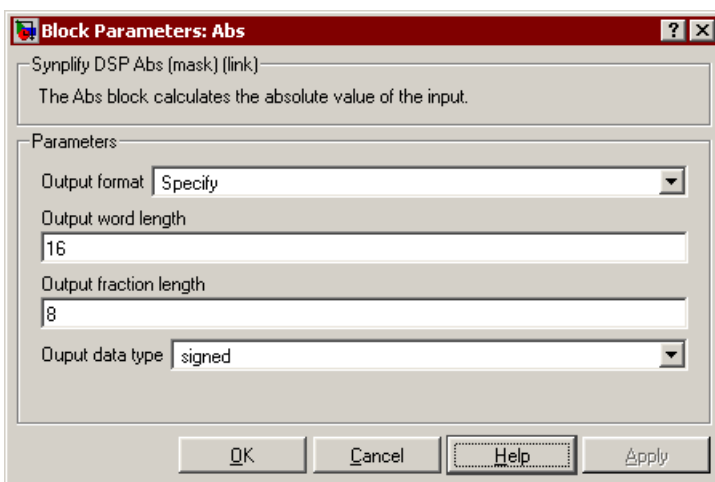


The Synplify DSP Abs block calculates the absolute value of the vector or scalar input. The output has the same signal dimension as the input, with each channel being the absolute value of the corresponding input channel.

## Latency

This block has no latency.

## Abs Parameters



For descriptions of the parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

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.

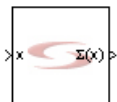
# Synplify DSP Accumulator

Implements an accumulator with optional reset and enable.

## Library

Synplify DSP [Math Functions](#)

## Description



The Synplify DSP Accumulator block implements an adder or subtractor-based accumulator with optional reset and enable ports.

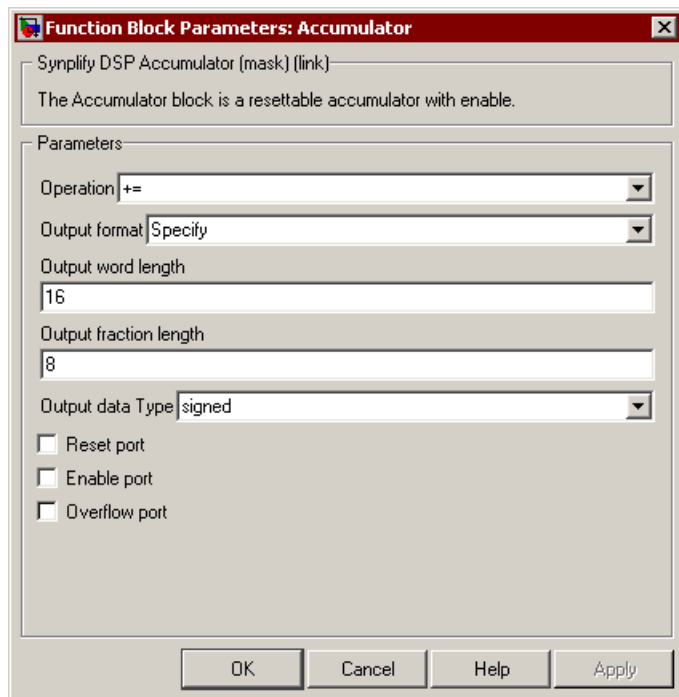
$$y[n] = x[n-1] + y[n-1]$$

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

## Latency

This block provides the result of the accumulating register.

## Accumulator Parameters



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

- A header bar with a close button (X).
- A text area with the text: "Synplify DSP Accumulator (mask) (link)" and "The Accumulator block is a resettable accumulator with enable."
- A "Parameters" section with the following controls:
  - "Operation": A dropdown menu showing "+=".
  - "Output format": A dropdown menu showing "Specify".
  - "Output word length": A text input field containing "16".
  - "Output fraction length": A text input field containing "8".
  - "Output data Type": A dropdown menu showing "signed".
  - Three checkboxes: "Reset port", "Enable port", and "Overflow port", all of which are currently unchecked.
- A footer bar with four buttons: "OK", "Cancel", "Help", and "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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

### 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.

# Synplify DSP Add

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

**Library**

Synplify DSP [DSP Basics](#) and Synplify DSP [Math Functions](#)

**Description**

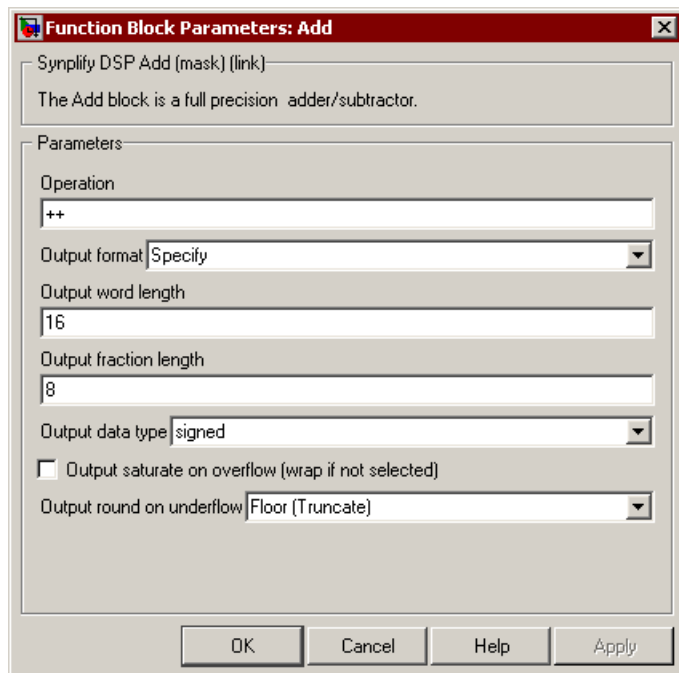
The Synplify DSP Add block implements a signed single-input or multi-input adder, whose inputs can be configured for addition or subtraction. The Add block can have up to 256 input ports. The inputs can be vectorized to a maximum size of 2048 for single-input adder (in sum of elements mode) and multi-input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

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

**Latency**

This block has no latency.

## Add Parameters



### 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  $\text{Input1} + \text{Input2} - \text{Input3}$ . 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 signal input port feeding in the vector signal and if you set Operation to +, the output is the sum of the vector elements. If you



set Operation to -, the output is the negative value of the sum of the vector elements.

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

### 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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output round on underflow	See <a href="#">Underflow Rounding Options, on page 8-289</a> for details about the rounding options available.

# Synplify DSP Binary Logic

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

## Library

Synplify DSP [Math Functions](#)

## Description



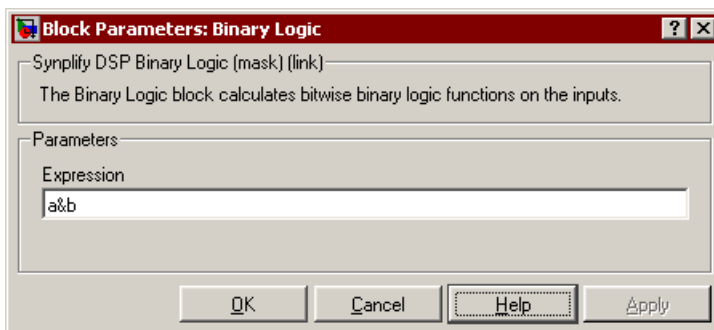
The Synplify DSP Binary Logic block implements bitwise binary logic functions. The input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

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.

## 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 8-24](#). 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.

# Synplify DSP Black Box

Allows you to embed other blocks or IP in a Synplify DSP design.

## Library

Synplify DSP [Ports & Subsystems](#)

## Description



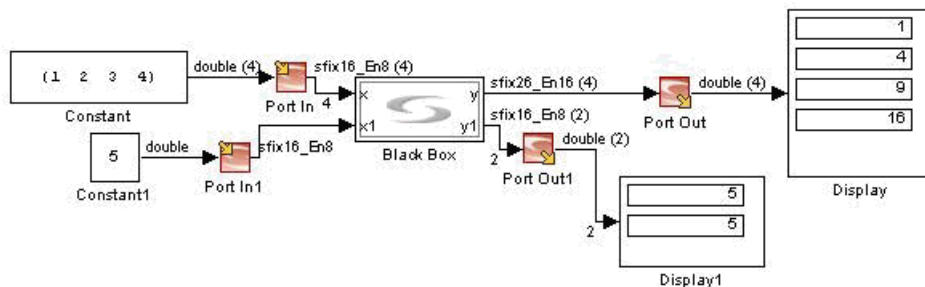
The Synplify DSP Black Box block implements a black box, which allows you to embed other blocks in a Synplify DSP design. For the purposes of simulation with Simulink, the black box is transparent; however, for RTL generation, the contents of the block will just be a black box. See the `<install_dir>\mathworks\toolbox\Synplicity\demos\examples` 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 ([Synplify DSP Smart Black Box, on page 8-238](#)) instead. For details about using a black box in your design, see [Using Black Boxes and Third-Party IP, on page 4-24](#).

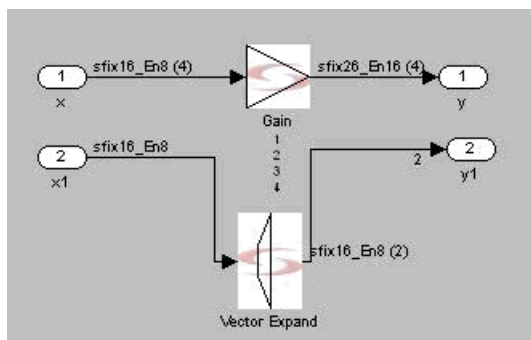
The Black Box consists of just an input and an output, to which you can add other blocks:



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.



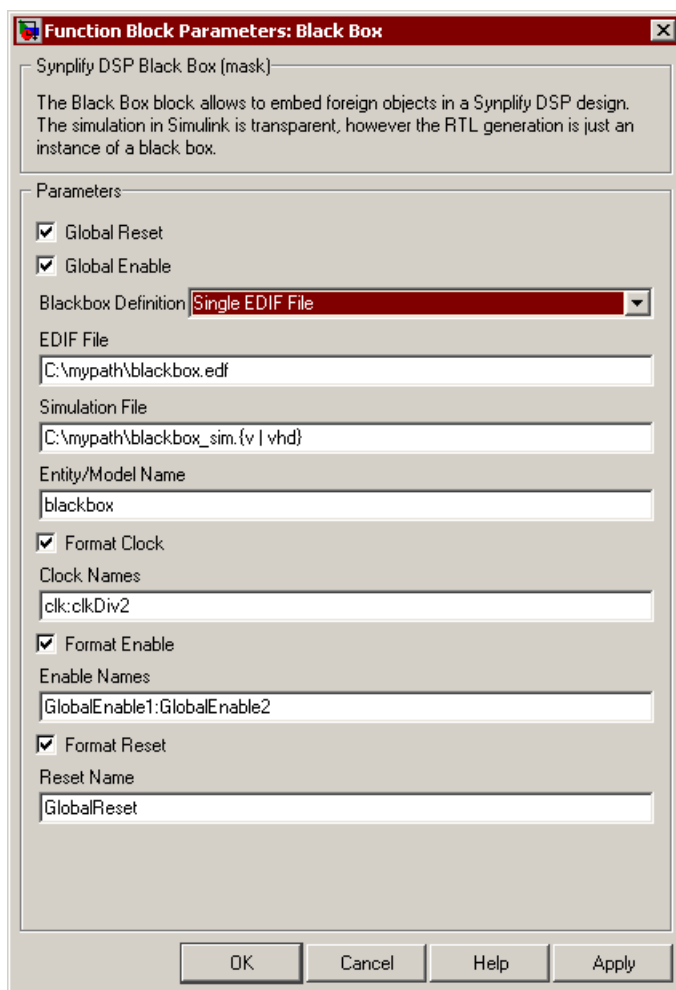
In the design shown above, the Port In block is the input to the x port of the Black Box block. In the HDL, the top-level input ports Port\_In\_e3, Port\_In\_e2, Port\_In\_e1, and Port\_In\_e0 are connected to the duplicated input ports x\_e0, x\_e1, x\_e2, and x\_e3, which are inside the black box.



## Latency

Latency is determined by the contents of the black box.

## Black Box Parameters



For information about setting these parameters, see [Setting Black Box Parameters, on page 4-28](#). 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.

## 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.



## 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.

## 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.

## Entity Model Name

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

## 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 Synplify DSP 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 Synplify DSP 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 Synplify DSP 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 Synplify DSP 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 Synplify DSP reset name, which is `GlobalResetSel`.

**Reset Name**

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

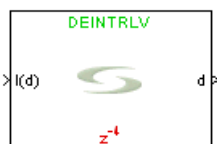
# Synplify DSP Block Deinterleaver

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

## Library

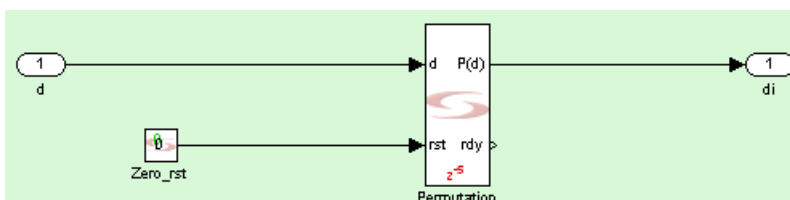
Synplify DSP [Communications](#)

## 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 5-2](#).

The following figure shows the internals of this block:

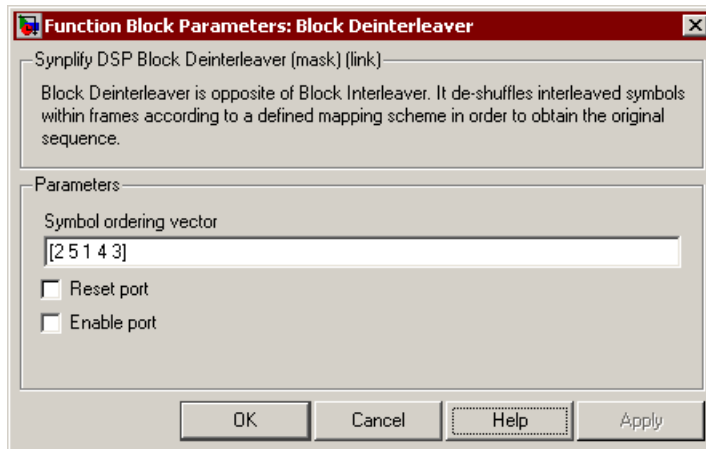


## Icon Annotations

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.

# Synplify DSP Block Interleaver

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

## Library

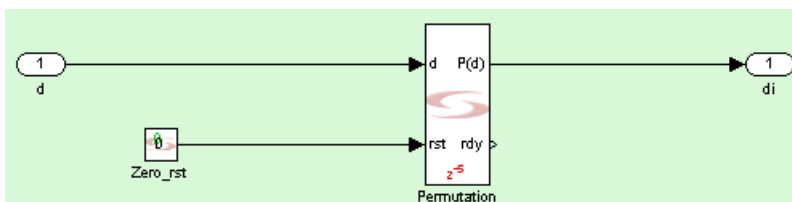
Synplify DSP [Communications](#)

## 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 5-2](#).

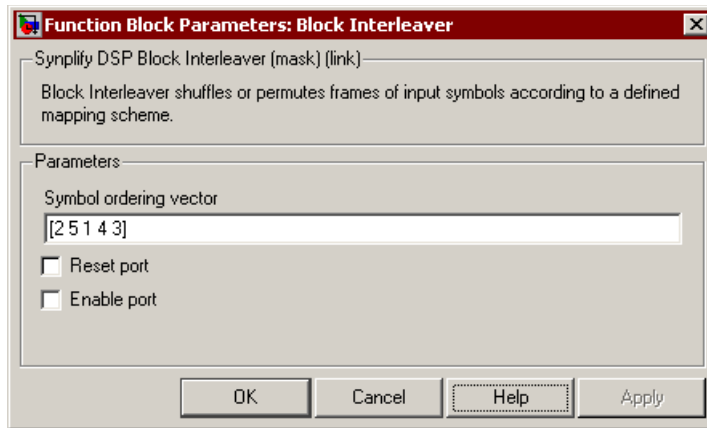
The following figure shows the internals of this block:



## Icon Annotations

Note	Specifies that the block is a deinterleaver.
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.

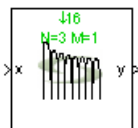
# Synplify DSP CIC

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

## Library

Synplify DSP [Filtering](#)

## Description

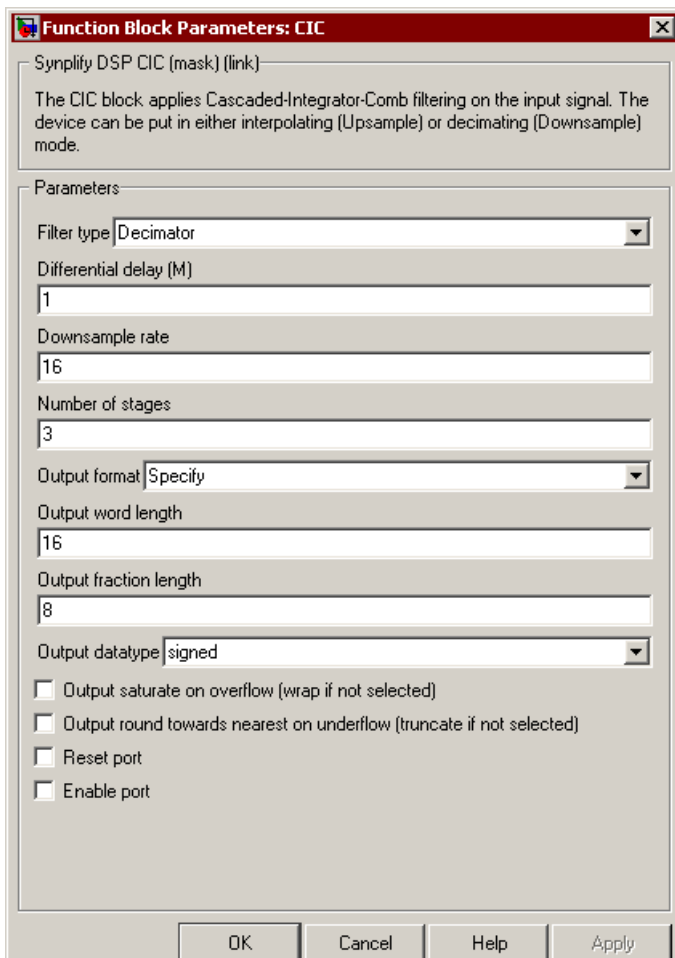


This is a custom block (see [Primitives and Custom Blocks, on page 5-2](#) for a definition) that implements a CIC filter by applying cascaded integrator-comb filtering on the input signal. Cascaded Integrator-Comb filters are a type of linear phase FIR filter, and have a comb section and an integrator section. You can use this filter in either interpolating (upsample) or decimating (downsample) mode.

## 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 sections and controls:

- Synplify DSP CIC (mask) (link):** A text area containing the 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:** A group box containing several controls:
  - Filter type:** A dropdown menu with "Decimator" selected.
  - Differential delay (M):** A text input field with the value "1".
  - Downsample rate:** A text input field with the value "16".
  - Number of stages:** A text input field with the value "3".
  - Output format:** A dropdown menu with "Specify" selected.
  - Output word length:** A text input field with the value "16".
  - Output fraction length:** A text input field with the value "8".
  - Output datatype:** A dropdown menu with "signed" selected.
  - Output saturate on overflow (wrap if not selected):** An unchecked checkbox.
  - Output round towards nearest on underflow (truncate if not selected):** An unchecked checkbox.
  - Reset port:** An unchecked checkbox.
  - Enable port:** An unchecked checkbox.

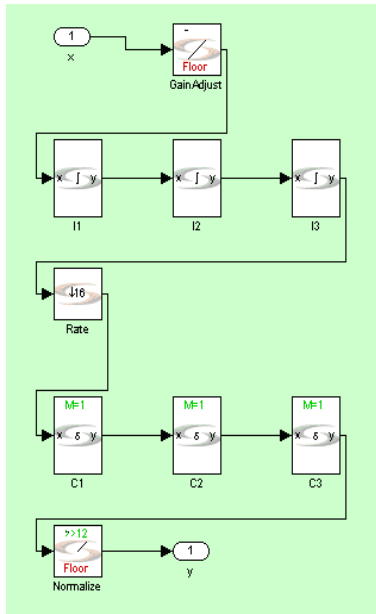
At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

### Filter Type

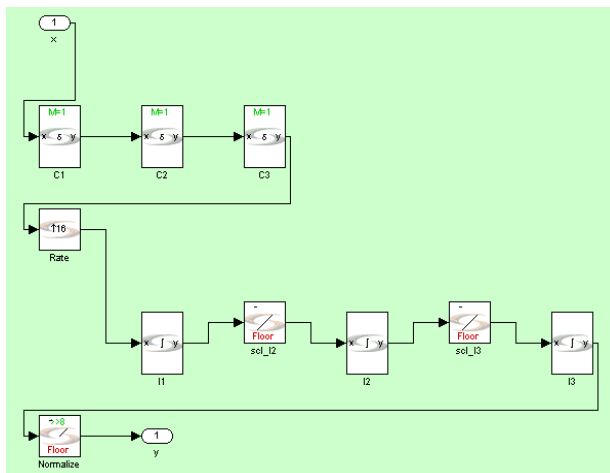
Determines the type of filter. The next figure shows how the filters are implemented, without resets and enables.

- 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## 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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see <a href="#">Underflow Rounding Options, on page 8-289</a> 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.

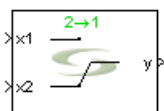
# Synplify DSP Commutator

Sequentially switches the data from the specified number of input ports to a single output port. The output data rate is increased by a factor of the number of input ports.

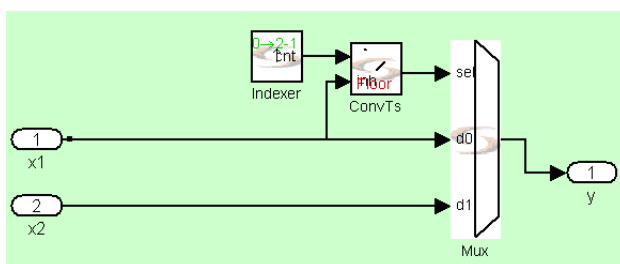
## Library

Synplify DSP [Signal Operations](#)

## Description



The Synplify DSP Commutator block sequentially switches the data from the specified number of 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 5-2 for a definition). The following figure shows the internal modeling:

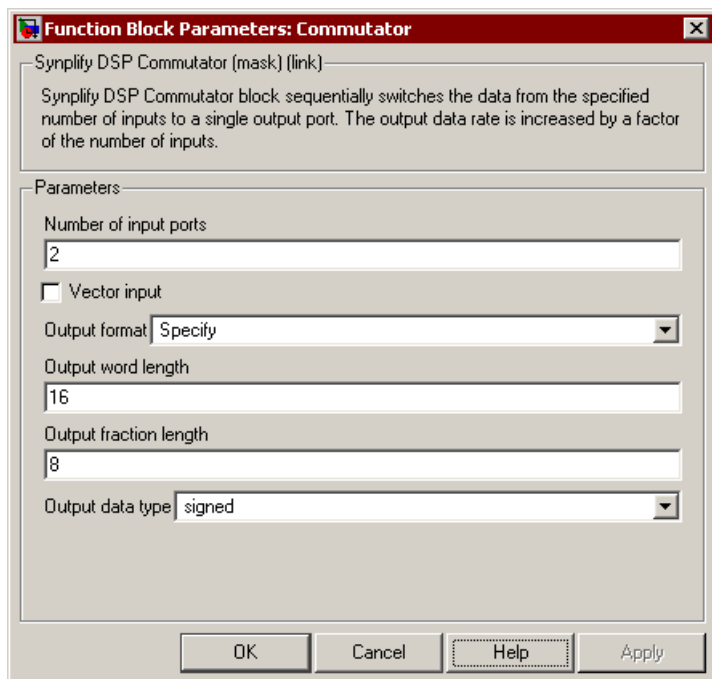


## Icon Annotation

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 Input Ports

Specifies the number of input ports from which data is multiplexed to the output.

### Vector Input

Determines whether the block accepts vector input. When enabled, the block accepts vector input.

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

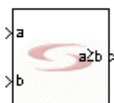
# Synplify DSP Comparator

Implements a programmable comparator.

## Library

Synplify DSP [Math Functions](#)

## Description

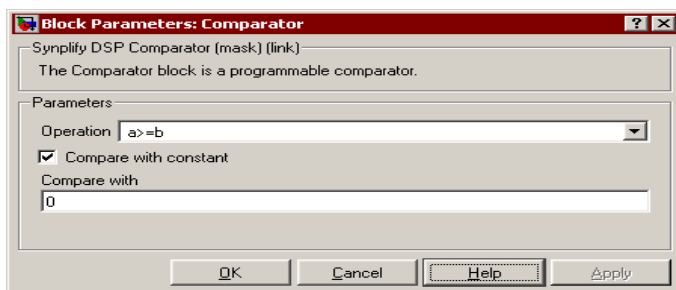


The Synplify DSP Comparator block implements a comparator by comparing two signals and returning a single bit.

## Latency

This block has no latency.

## Comparator Parameters



## Operator

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

- $a == b$
- $a != b$

- $a < b$
- $a \leq b$
- $a > b$
- $a \geq b$  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.

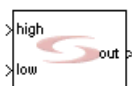
# Synplify DSP Concat

Concatenates the bits of up to 32 input signals.

## Library

Synplify DSP [Signal Operations](#)

## Description

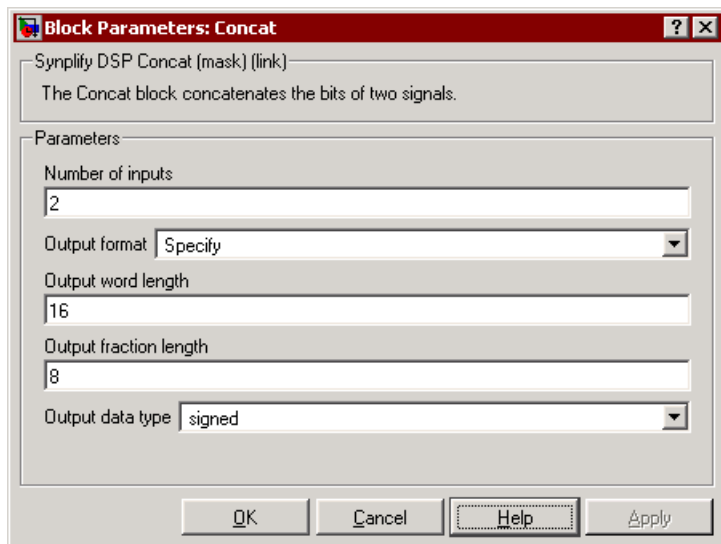


The Synplify DSP Concat block concatenates the bits of up to 32 input signals. This block converts the inputs to unsigned integers, by ignoring the binary point and maintaining the bit representation of the word. The output is an unsigned integer with the word length equal to the sum of the input word 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.

## 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	<a href="#">Output Word Length, on page 8-288</a>
Fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Data type	<a href="#">Output Data Type, on page 8-288</a>



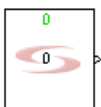
# Synplify DSP Constant

Drives a a constant value of a specified data type.

## Library

Synplify DSP [Sources](#)

## Description



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

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

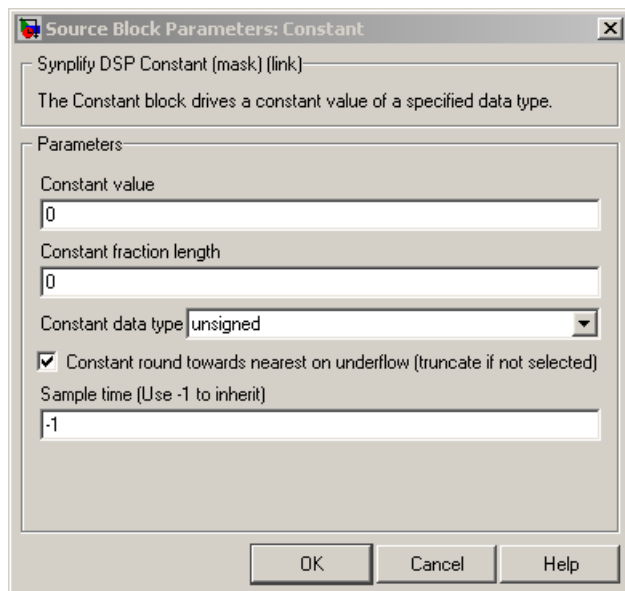
The value is cast to the specifications of the data format, and also displayed in the icon of the instance. The sample period of the constant is usually inherited through back inheritance from the rest of the design, but you can use the parameters to force the sample period.

## 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.
Center Annotation	The annotation in the center displays the rounded or truncated value of the constant.
Latency Annotation	There is no latency, and the block drives the same value independent of the global reset or enable.

## Constant Parameters



### Constant value

Sets the value to be driven (the output value of the block). You can specify a column vector for a vectorized output. Each row 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	<a href="#">Output Fraction Length, on page 8-288</a>
Constant data type	<a href="#">Output Data Type, on page 8-288</a>

### 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 8-289](#) for details.

## Sample Time

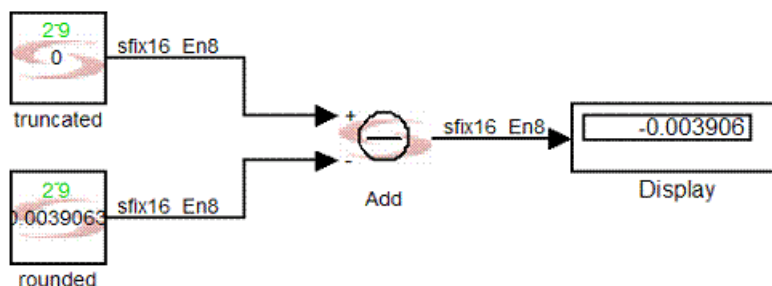
Sets the sample time. Use -1 to inherit.

## Constant Block Examples

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

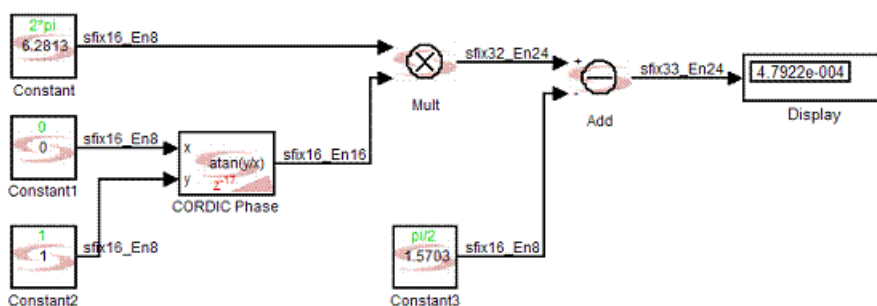
### Example 1

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:



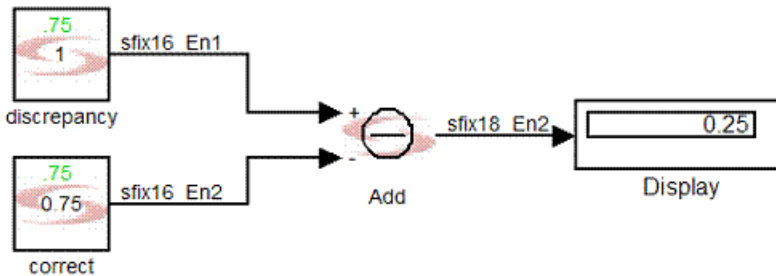
### Example 2

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.

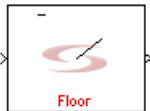
## Synplify DSP Convert

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

Synplify DSP [Signal Operations](#)

### Description



The Synplify DSP Convert block changes the word size and data type of the input. Most Synplify DSP 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 Synplify DSP 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 (Synplify DSP 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:

	<b>Signed (two's complement) arithmetic</b>	<b>Unsigned arithmetic</b>
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:

<b>A Convert block with these input parameters...</b>	<b>Gives you these results...</b>
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

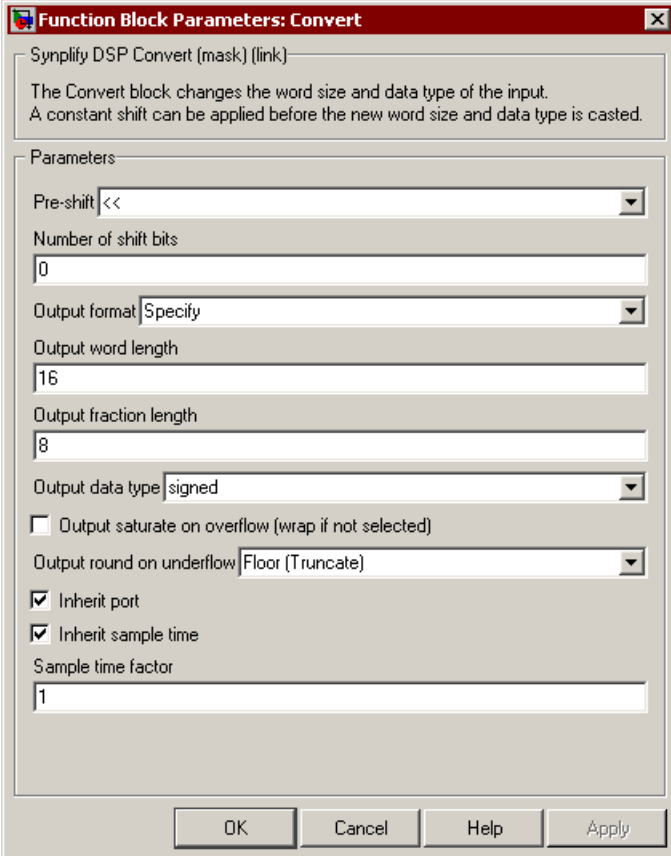
## Icon Annotations

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:

- Title Bar:** "Function Block Parameters: Convert" with a close button (X).
- Description:** "Synplify DSP 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."
- Parameters Section:**
  - Pre-shift:** A dropdown menu with "<<" selected.
  - Number of shift bits:** A text input field with "0".
  - Output format:** A dropdown menu with "Specify" selected.
  - Output word length:** A text input field with "16".
  - Output fraction length:** A text input field with "8".
  - Output data type:** A dropdown menu with "signed" selected.
  - Output saturate on overflow (wrap if not selected):** An unchecked checkbox.
  - Output round on underflow:** A dropdown menu with "Floor (Truncate)" selected.
  - Inherit port:** A checked checkbox.
  - Inherit sample time:** A checked checkbox.
  - Sample time factor:** A text input field with "1".
- Buttons:** "OK", "Cancel", "Help", and "Apply" at the bottom.

### 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 8-292 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 8-287

Output word length    [Output Word Length](#), on page 8-288.  
You can also specify it in terms of the `syn_inp_wl`, `syn_inp_fl`, and `syn_inp_dt` variables. 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 [Special Variables](#), on page 8-292.

Output fraction length    [Output Fraction Length](#), on page 8-288.  
You can also specify it 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 [Special Variables](#), on page 8-292.

## Output Data Type

Determines the data type for the output.

Signed    See [Output Data Type](#), on page 8-288 for details.

Unsigned    See [Output Data Type](#), on page 8-288 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 8-52 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.

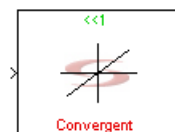
Output saturate on overflow	Saturates the overflow when the option is enabled and wraps the overflow when it is disabled. See <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output round on underflow	See <a href="#">Underflow Rounding Options, on page 8-289</a> for details about 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. 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 8-292](#) for information about these variables.
- Use the inherit option to specify the Output data type. See [Output Data Type, on page 8-51](#) 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**

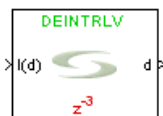
Specifies a time factor for the sample time that the output port inherits from the inherit port. This option is only available when you select Inherit sample time.

# Synplify DSP Convolutional Deinterleaver

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

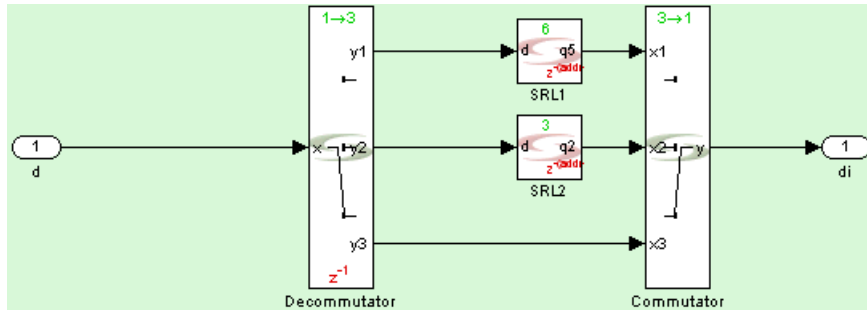
**Library**

Synplify DSP [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 5-2](#).

The following figure shows the internal modeling of this block:

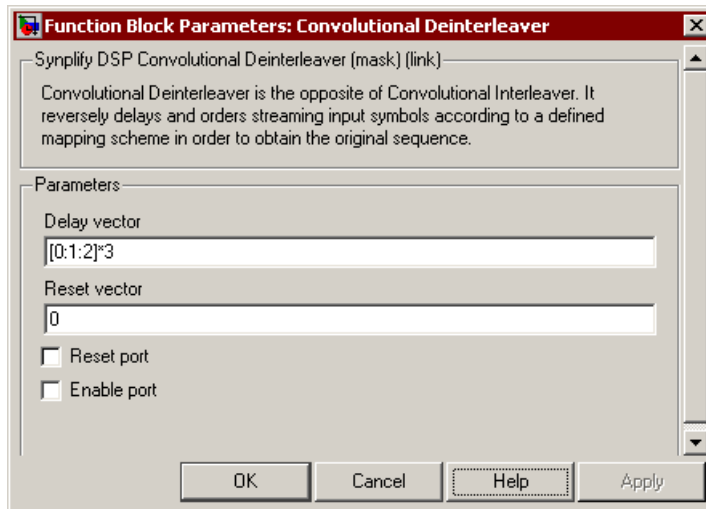


## Icon Annotations

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.

# Synplify DSP Convolutional Encoder

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

## Library

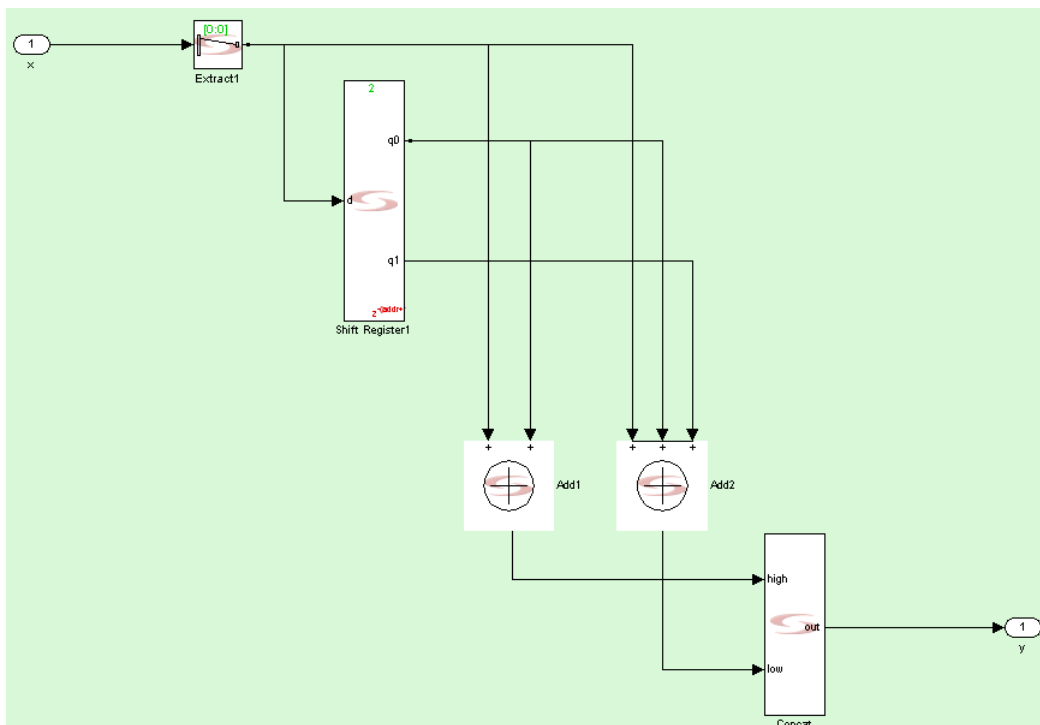
Synplify DSP [Communications](#)

## Description



The Synplify DSP Convolutional Encoder is a custom block (see [Primitives and Custom Blocks](#), on page 5-2 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:

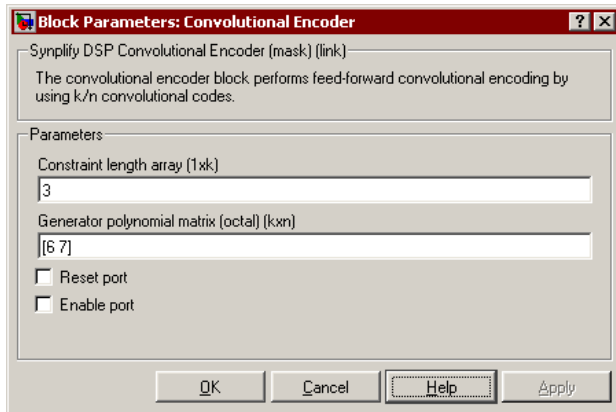


## Icon Annotation

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.

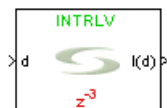
# Synplify DSP Convolutional Interleaver

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

## Library

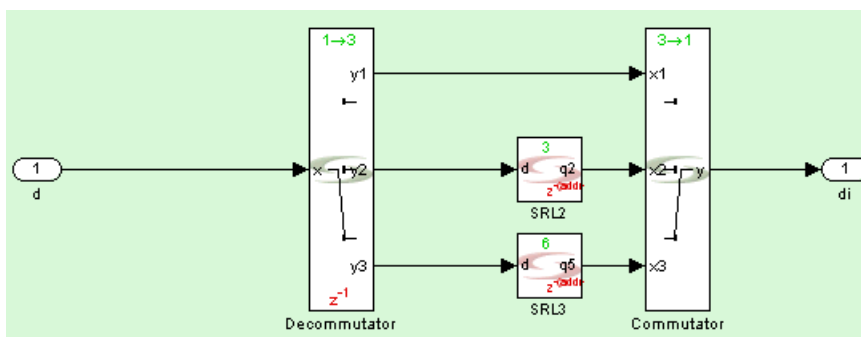
Synplify DSP [Communications](#)

## 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 5-2.

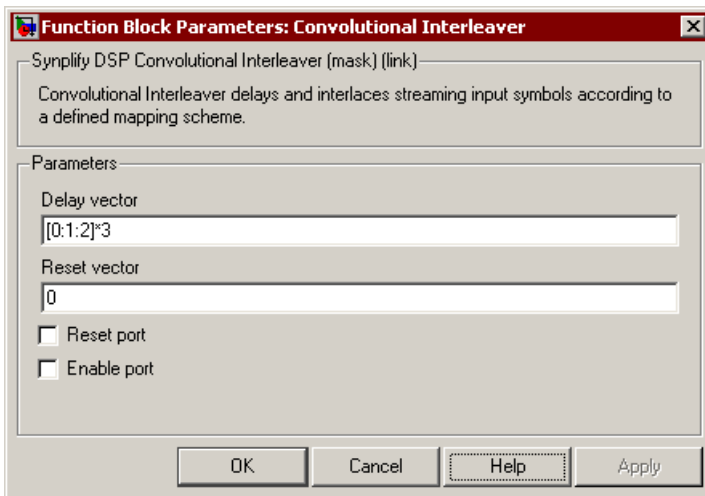
The following figure shows the internals of this block:



## 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.



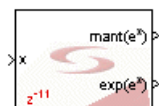
# Synplify DSP CORDIC Exp

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

## Library

Synplify DSP [CORDIC](#)

## Description



The Synplify DSP CORDIC Exp block uses a CORDIC algorithm to calculate the natural exponent of the input. See [CORDIC Algorithms, on page 3-3](#) for a description of the algorithms.

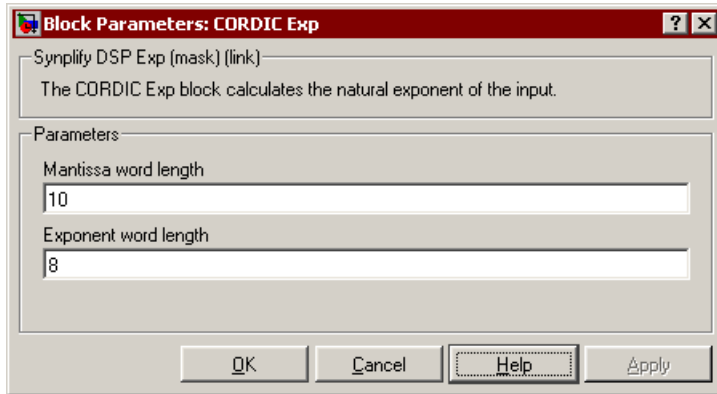
The result from this block is output in the form of a mantissa and an exponent, where  $x = \text{mant} \cdot 2^{\text{exp}}$ . The mantissa is a fraction, with the most significant bit of the mantissa to the left of the binary point. The exponent is an integer. The number of iterations is equal to the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

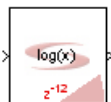
# Synplify DSP CORDIC Log

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

## Library

Synplify DSP [CORDIC](#)

## Description



The Synplify DSP CORDIC Log block calculates the natural logarithm of the input, using a CORDIC algorithm. See [CORDIC Algorithms, on page 3-3](#) for a description of CORDIC algorithms. The number of iterations is equal to output word length.

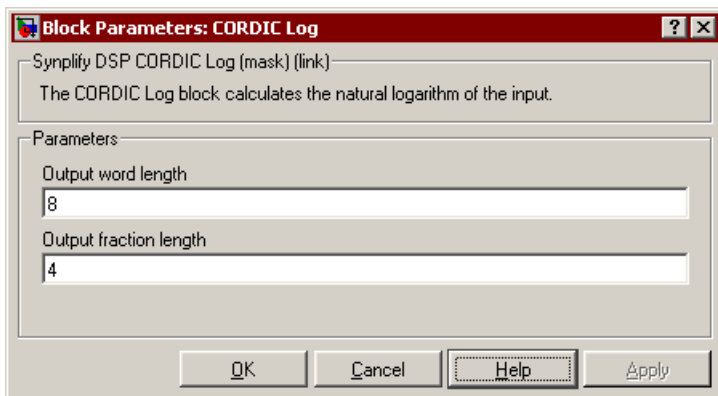
## Icon Annotations

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	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>

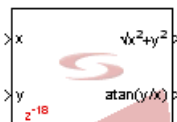
# Synplify DSP CORDIC Polar

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

## Library

Synplify DSP [CORDIC](#)

## Description



The Synplify DSP CORDIC Polar block uses the CORDIC algorithm to perform rectangular-to-polar conversions. See [CORDIC Algorithms, on page 3-3](#) for a description of the algorithms. The CORDIC algorithm is used for computation, and the implementation is fully pipelined.

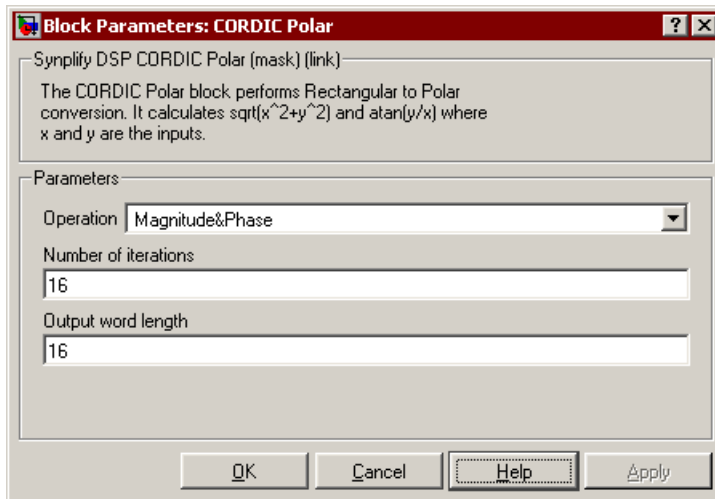
When  $y=0$ , the value of CORDIC phase goes back and forth between  $-0.5$  and  $0.5$  for the values of  $x<0$ . This is because of numerical instability in the CORDIC algorithm, as  $0.5$  and  $-0.5$  correspond to the same angle  $(-\pi, \pi)$ . If this causes a problem in the application, use 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 8-65](#).

### Output word length

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

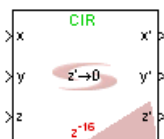
# Synplify DSP CORDIC Rotator

Implements a fully pipelined CORDIC rotator.

## Library

Synplify DSP [CORDIC](#)

## Description



The Synplify DSP 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 3-3](#) 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 Synplify DSP tool supports circular, linear and hyperbolic coordinate systems. For additional background information about the algorithms, see [CORDIC Algorithms, on page 3-3](#).

The Synplify DSP 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 Synplify DSP Rotator block does this automatically.

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

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

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



$$X_0=X[n]$$

$$Y_0=Y[n]$$

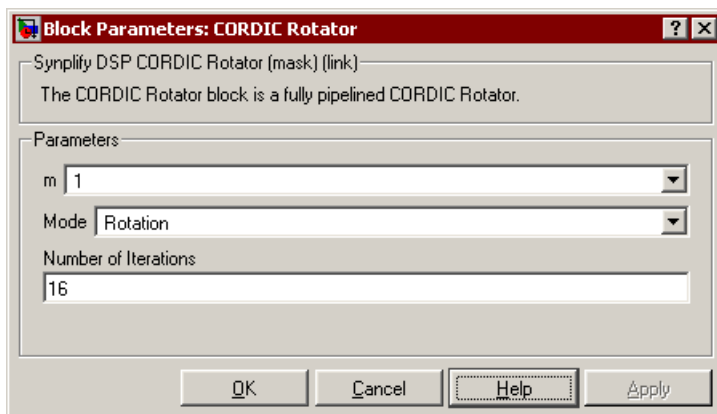
$$Z_0=Z[n]$$

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

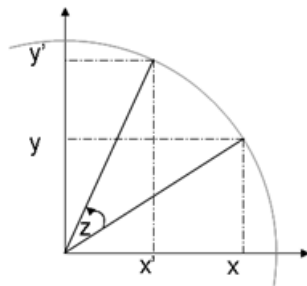
## CORDIC Rotator Parameters



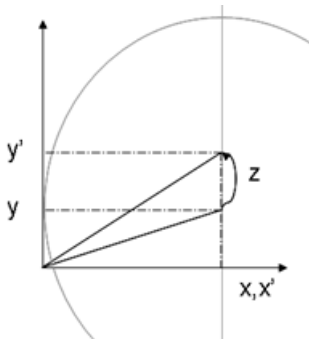
**m**

Determines the coordinate system used by the block. For further details about coordinate systems, see [Circular, Linear, and Hyperbolic Coordinate Systems, on page 8-67](#).

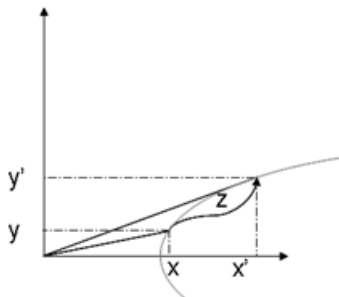
- 1 specifies a circular (trigonometric) coordinate system. The  $z$  input represents a circular coordinate (angle) expressed in normalized radians. The fraction  $[-1,1]$  corresponds to  $[-\pi,\pi]$ ; however the rotator only converges for inputs in the range of  $[-.2775,.2775]$ , which corresponds 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.

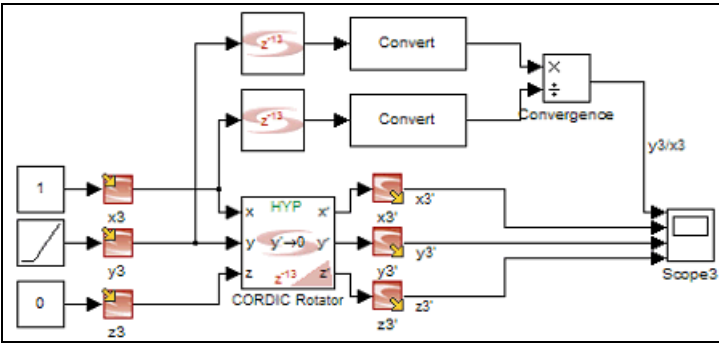
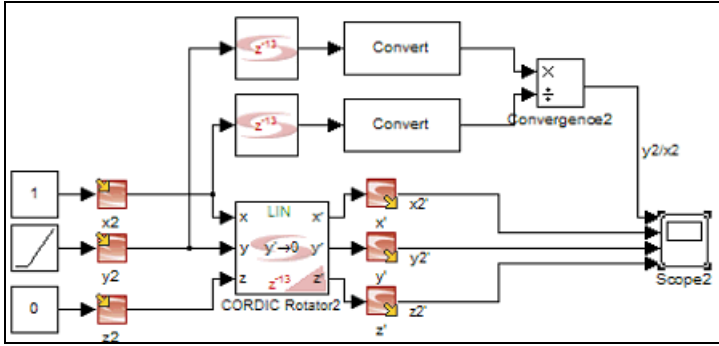
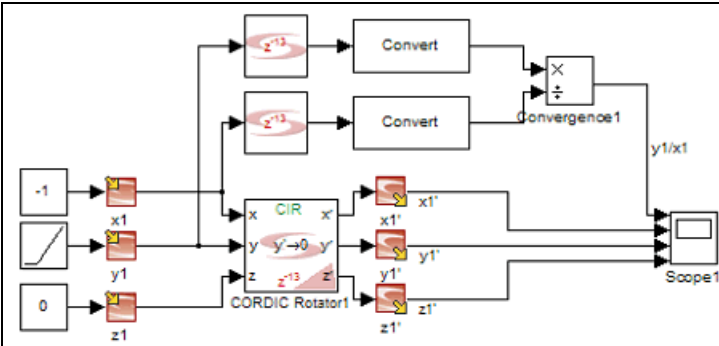
### Data Format

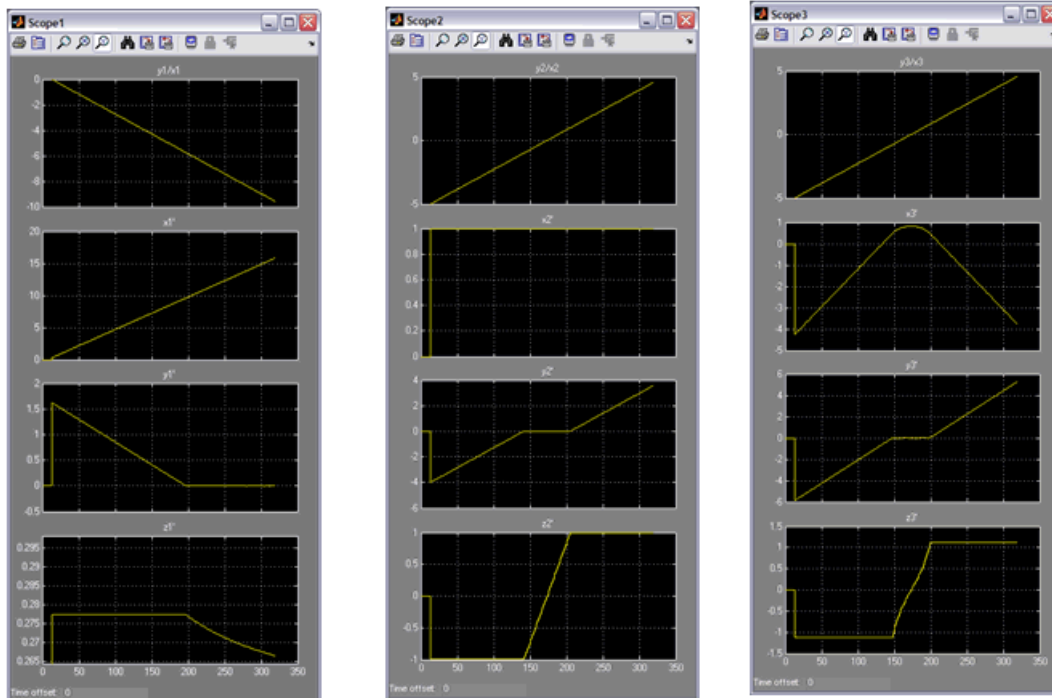
The data formats for rotation and vectoring modes are as follows:

- 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$ .

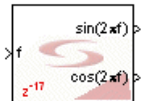
# Synplify DSP CORDIC SinCos

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

## Library

Synplify DSP [CORDIC](#)

## Description



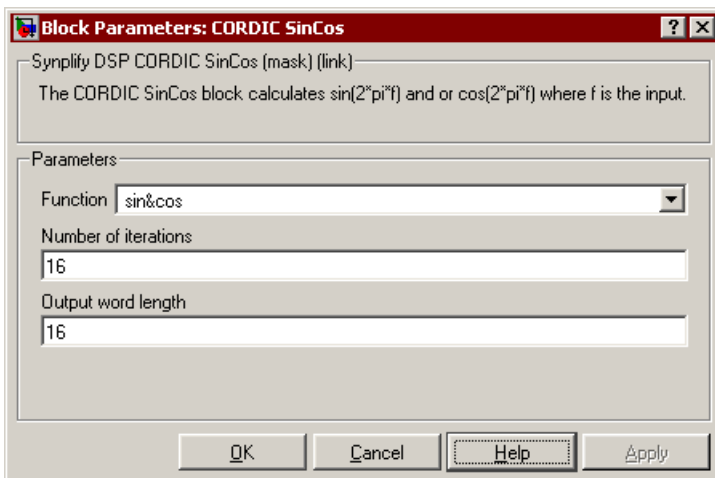
The Synplify DSP CORDIC SinCos block implements a sine and/or cosine generator circuit using a fully parallel CORDIC algorithm in rotation mode. (See [CORDIC Algorithms, on page 3-3](#) for a description of the algorithms.) It calculates  $\sin(2\pi f)$  and/or  $\cos(2\pi f)$  where  $f$  is an input. 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.

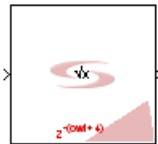
# Synplify DSP CORDIC Sqrt

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

## Library

Synplify DSP [CORDIC](#)

## Description



The Synplify DSP CORDIC Sqrt block calculates the square root of the input using a fully pipelined CORDIC algorithm for the implementation. See [CORDIC Algorithms, on page 3-3](#) 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.
--------------------	--

---



# Synplify DSP Counter

Implements a configurable counter with enable and reset.

## Library

Synplify DSP [Sources](#)

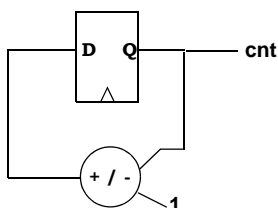
## Description



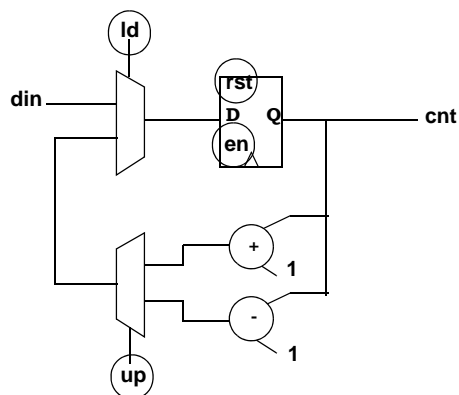
The Synplify DSP Counter block implements a configurable counter with enable and reset, and provides looping control for many algorithms. It offers the following:

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

Basic Counter Operation



Counter with Optional Ports



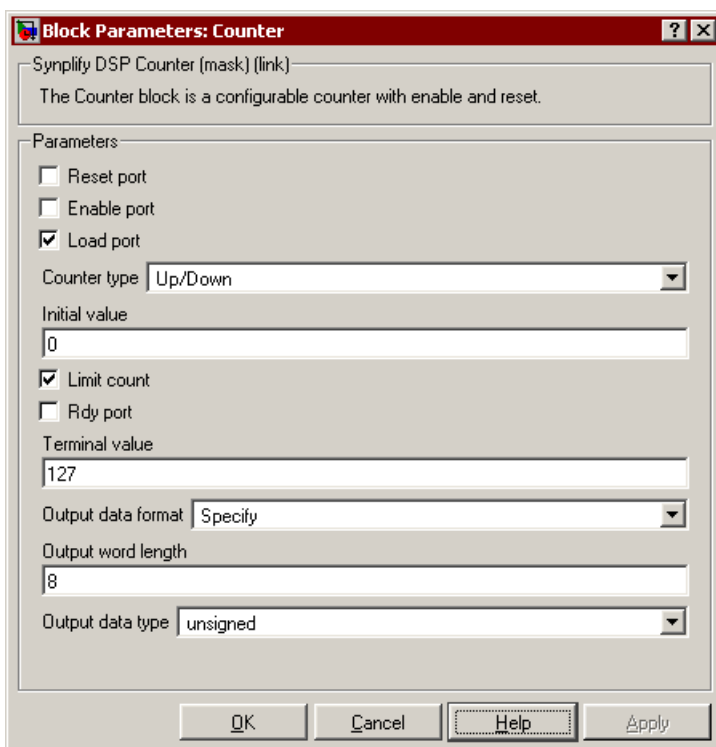
- Operations: counting up, down, and programmable
- Initial and terminal values for the count
- Cast on the output for sizing

## Icon Annotations

The icon for this block displays the following information:

Top Annotation	The annotation at the top of the block 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



### 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.

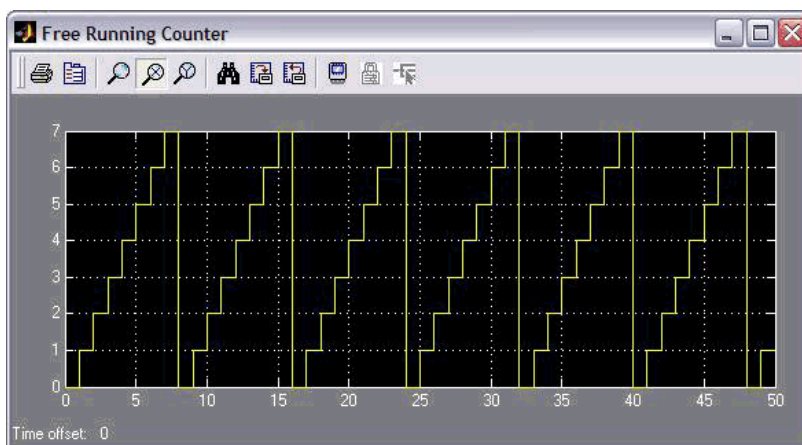
**Initial Value**

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

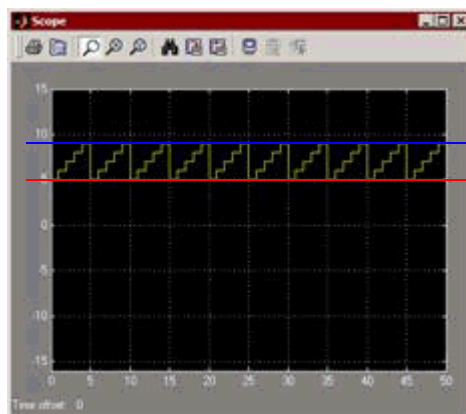
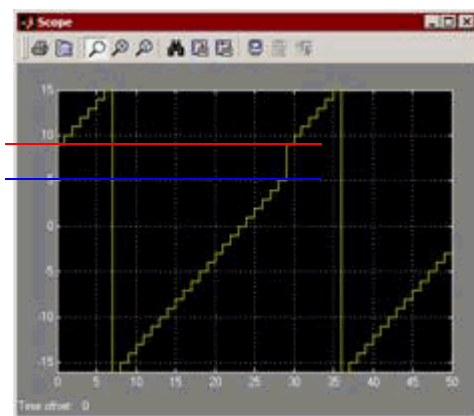
Resets the counter when it reaches the specified value; it resets the count to the Initial Value. The default terminal value is 127. To use this option, you must enable Limit Count.

———— Initial Value    ———— Terminal Value

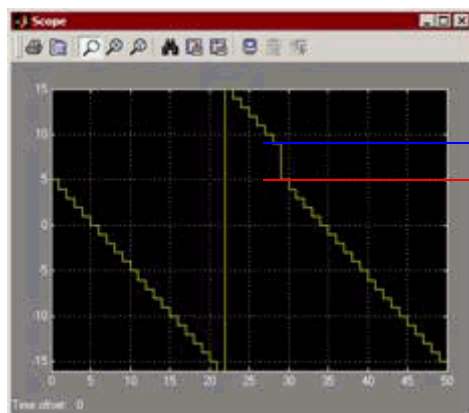
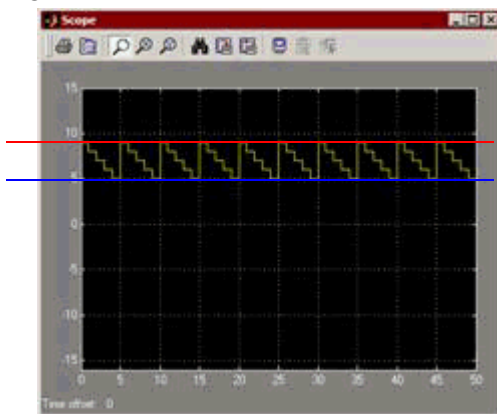
$T < I$

$I < T$

UP



DOWN



## Output Data Format

Determines data type and word length. Unlike other Synplicity 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: <math>(2^N) - 1</math></p> <p>Signed: <math>-2^{(N-1)}</math> to <math>(2^{(N-1)} - 1)</math></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  $1*2^2 + 1*2^1 + 0*2^0 = 6$ .

## Synplify DSP DDS

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

### Library

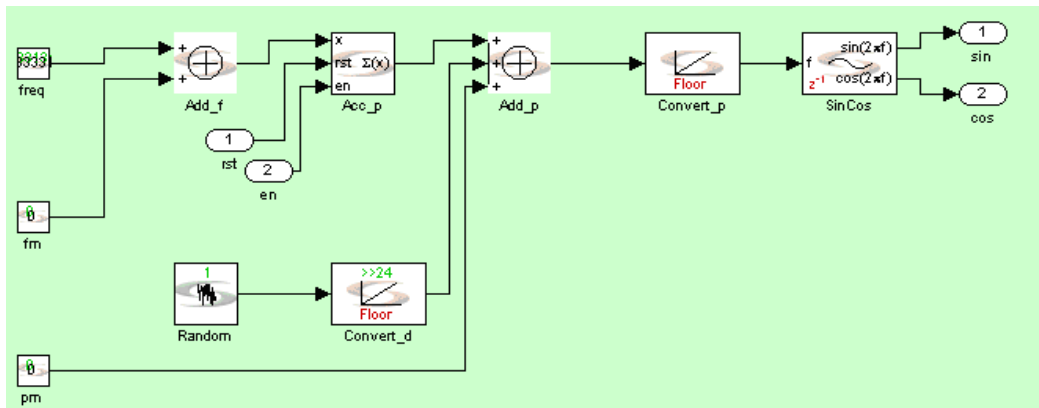
Synplify DSP [Sources](#)

### Description



This custom block (see [Primitives and Custom Blocks, on page 5-2](#) 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.



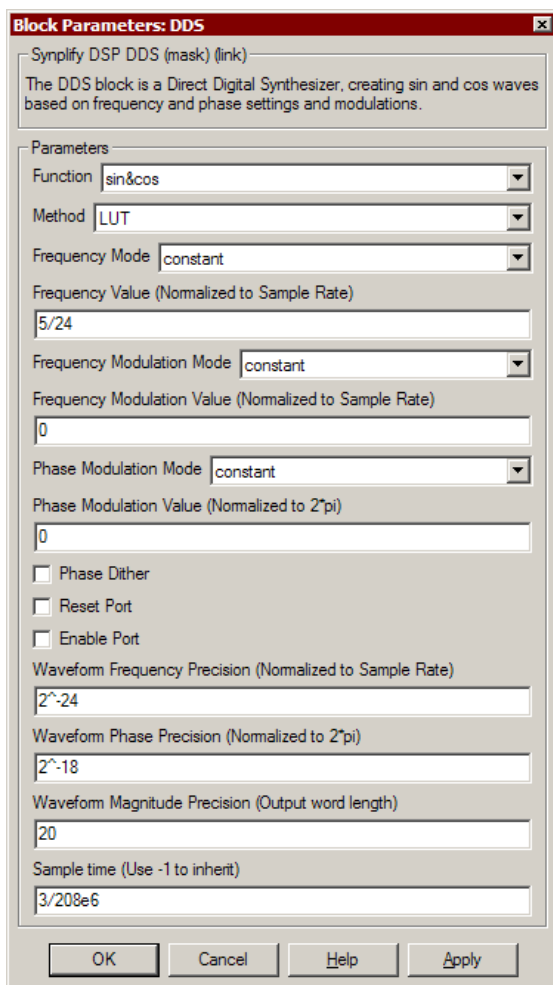
## Latency

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



The dialog box titled "Block Parameters: DDS" contains the following fields and controls:

- 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 2\*pi):
- ☐ Phase Dither
- ☐ Reset Port
- ☐ Enable Port
- Waveform Frequency Precision (Normalized to Sample Rate):
- Waveform Phase Precision (Normalized to 2\*pi):
- Waveform Magnitude Precision (Output word length):
- Sample time (Use -1 to inherit):

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

### 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  $2\pi$ . 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

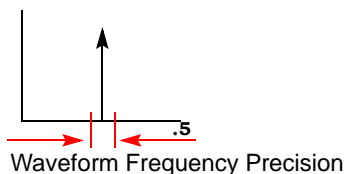
When enabled, the block is implemented with a reset pin. The reset port is connected to the reset signal of the internal shift register.

### 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.

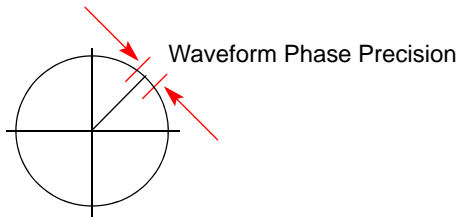
### Waveform Frequency Precision

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.



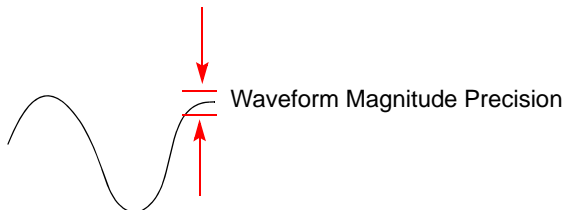
### Waveform Phase Precision

Specifies the input phase precision of the waveform generator, normalized to  $2\pi$ .



### 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.

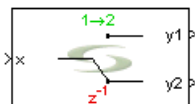
# Synplify DSP Decommutator

Sequentially switches the data at the input port to multiple output ports, reducing the data rate of each output port accordingly.

## Library

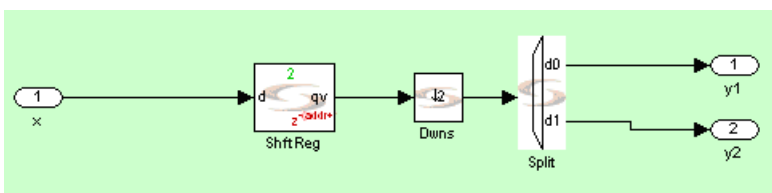
Synplify DSP [Signal Operations](#)

## 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 5-2 for a definition). The following figure shows the internal modeling:

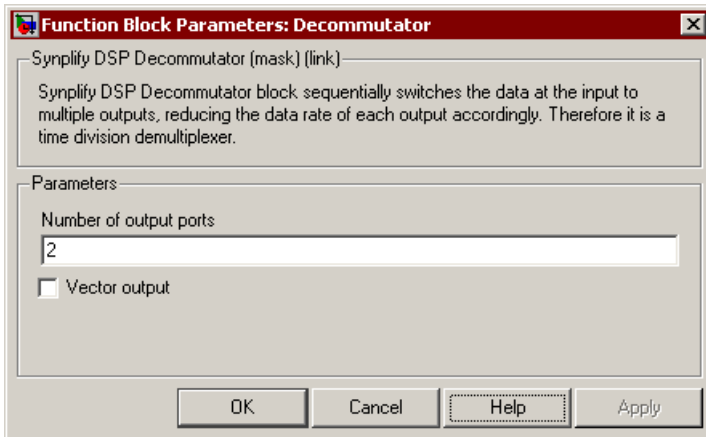


## Icon Annotation

The icon for this block displays the following information:

Top Annotation	The annotation at the top of the block indicates the input port and the number of output ports.
Latency Annotation	One sample latency with respect to the output clock domain.

## Decommutator Parameters



### Number of output ports

Specifies the number of output ports to which the input data must be demultiplexed.

### Vector output

Determines whether the block produces vector output. When enabled, the block sends out vector output.

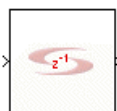
# Synplify DSP Delay

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

## Library

Synplify DSP [DSP Basics](#), Synplify DSP [Memories](#)

## Description

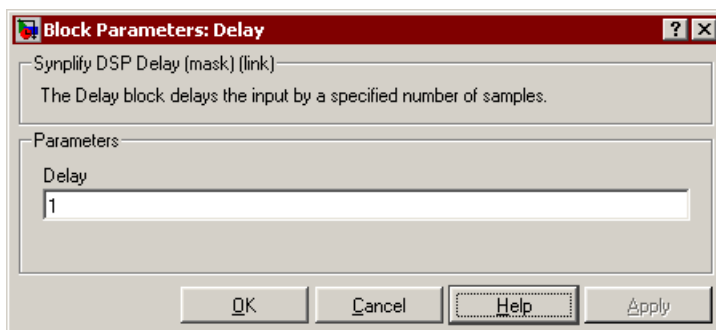


The Synplify DSP Delay block delays the input by a specified number of sample clock cycles. Initial values in the delay line are zero. You can also use the Register block ([Synplify DSP Register, on page 8-211](#)) to specify a delay, but it is recommended that you use the Delay block wherever possible.

## 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.

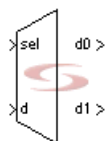
# Synplify DSP Demux

Implements a de-multiplexer of up to 32 outputs.

## Library

Synplify DSP [Signal Operations](#)

## Description

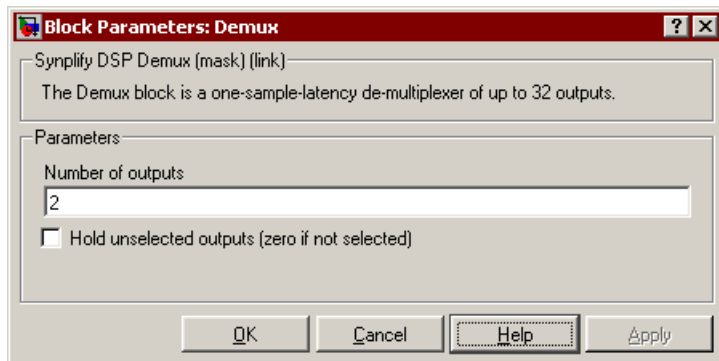


The Synplify DSP Demux block is a de-multiplexer of up to 32 outputs. The sel input determines which of the outputs gets the value of the d data input. This value becomes available on the output, and is retained until overwritten.

## Latency

This block has no latency.

## Demux Parameters





**Number of outputs**

Determines the number of outputs required. You can specify up to 32 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.

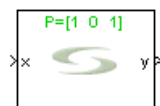
# Synplify DSP Depuncture

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

## Library

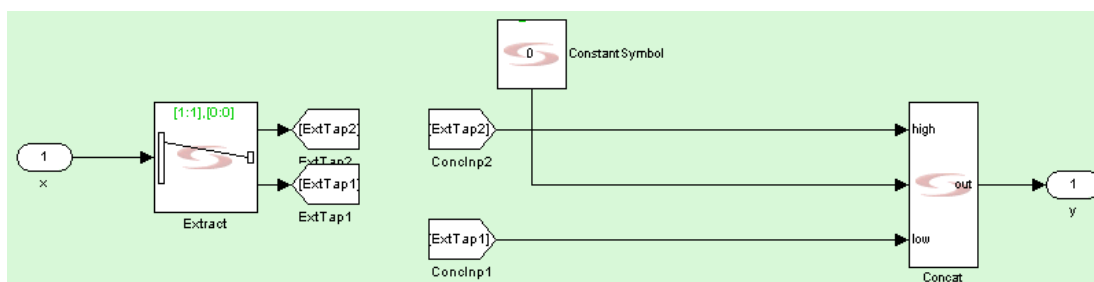
Synplify DSP [Communications](#)

## 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 5-2 for a definition.) The following figure shows how the block is modeled:

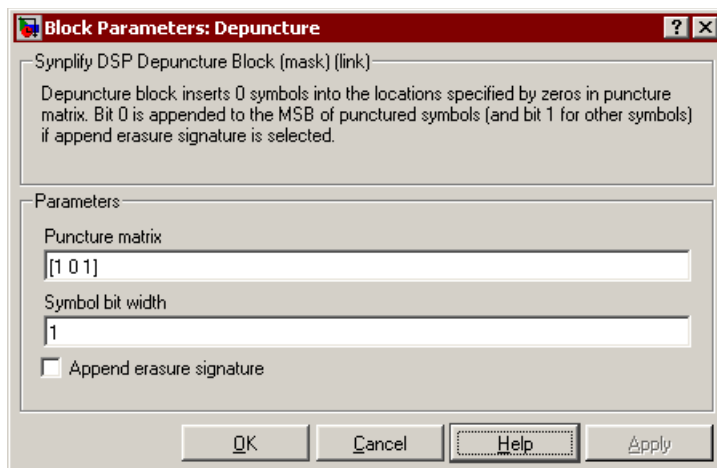


## Icon Annotations

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.

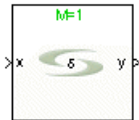
# Synplify DSP Differentiator

Performs a discrete time differentiation of the input signal.

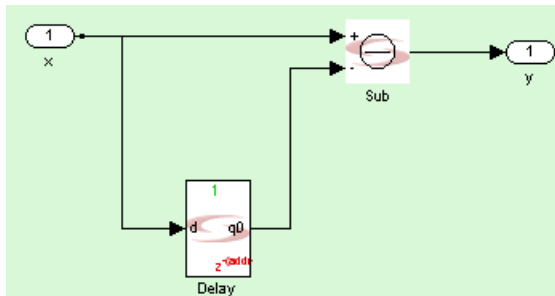
## Library

Synplify DSP [Filtering](#)

## Description



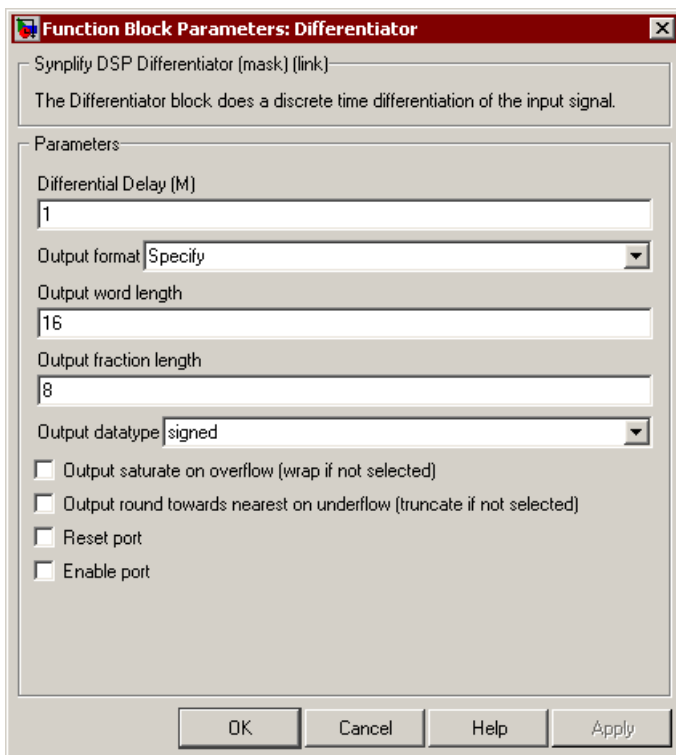
This custom block (see [Primitives and Custom Blocks](#), on page 5-2 for a definition) performs a discrete time differentiation of the input signal.



## Latency

This block has no latency.

## Differentiator Parameters



### 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options</a> , on page 8-289 for details.
Output round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see <a href="#">Underflow Rounding Options</a> , on page 8-289 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.

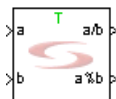
# Synplify DSP DivMod

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

## Library

Synplify DSP [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

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

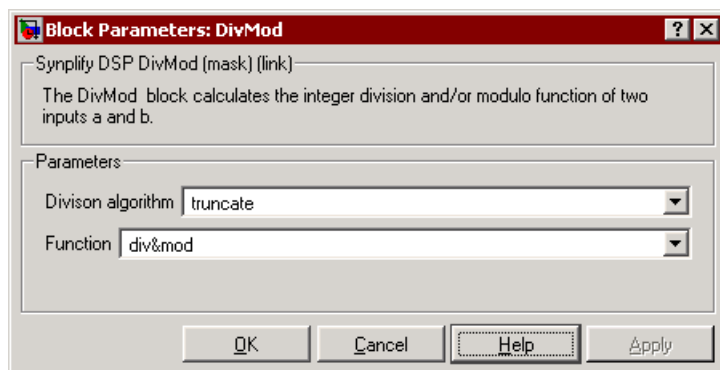
For a detailed discussion of division algorithms, see [Overview of Division Algorithms, on page 8-103](#). Computer languages are notoriously inconsistent in their implementation of mod for negative numbers. Synplify DSP currently uses T-division, which is described more fully in [Division algorithm, on page 8-100](#).

## 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 stands for truncate.
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 of the following:

- Truncate  
Uses T-division, which truncates any fraction of the quotient. This is the same as rounding towards zero. This behavior matches the MATLAB `fix(A/B)` and `rem(A,B)` 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 8-104](#) for details.
- Floor  
Uses F-division, where the quotient is rounded towards minus infinity. This matches the MATLAB `floor(A/B)` and `mod(A,B)` functionality. The algorithm satisfies  $A = Q*B + R$  equality, where



$B \cdot R \geq 0$ . (If nonzero, B and R have the same sign.) See [T-Division](#), [F-Division](#), and [E-Division](#), on page 8-104 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  $A=Q \cdot B+R$ .

## Function

Allows you to set different modes, where only the ports that are needed are made available.

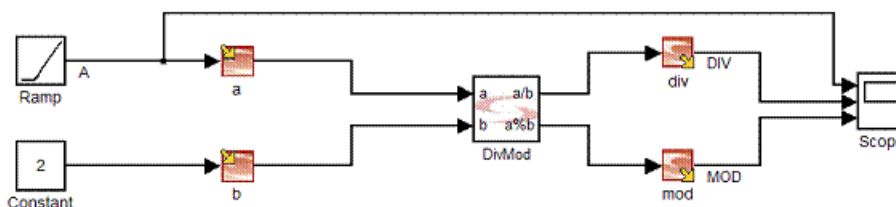
- div sets the mode where only the division output port is required.
- mod sets the mode where only the modulus output port is required.
- div&mod sets the mode where both the division and modulo output ports are required.

## Data Format

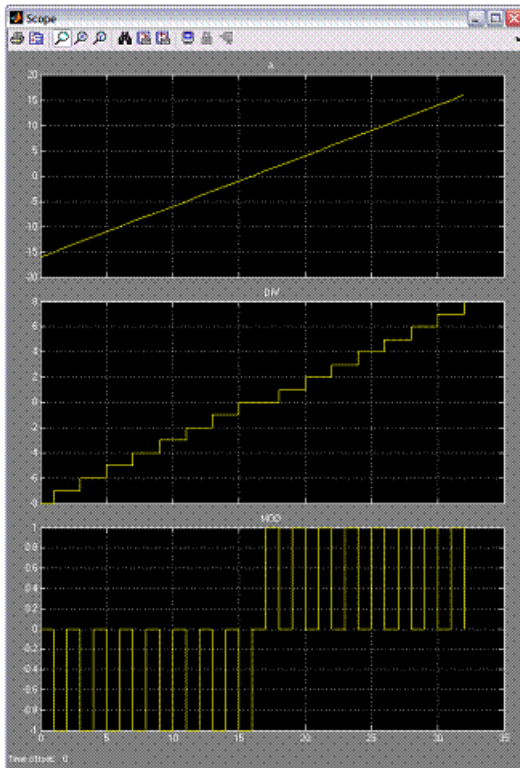
- The WL of Div is the integer WL of A (covers  $B = 1$ ).
- The WL of Mod is the integer WL of B (maximum Mod is  $|B| - 1$ ).

## Examples

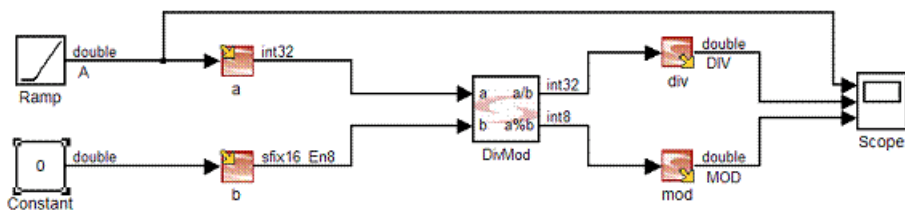
Consider the following sweep:

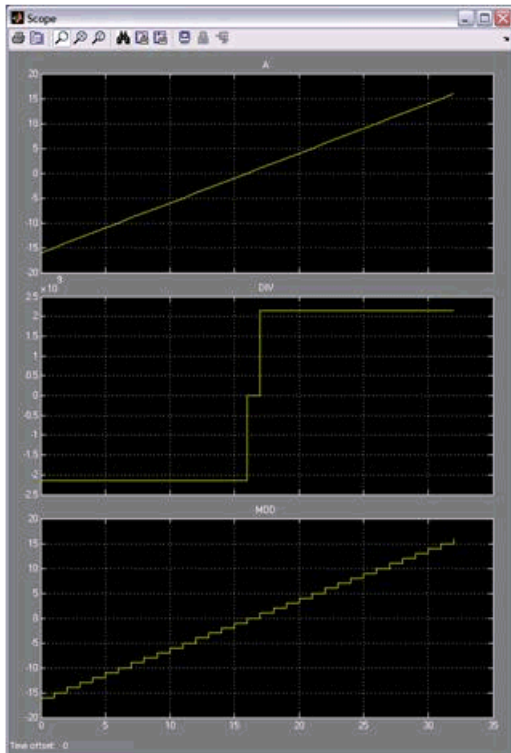


The result of the F-division and corresponding mod implemented by Synplify DSP shows the truncation of the fractions:



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 division<sup>1</sup>:

$Q_T = \text{trunc}(A/B)$	T-division (Truncated, ISO C99)
$Q_F = \lfloor A/B \rfloor$	F-division (Floor)
$Q_R = \text{round}(A/B)$	R-division (Round)
$Q_C = \lceil A/B \rceil$	C-division (Ceil)
$R = A - Q*B$	

There is also a modulo-centric approach to this problem<sup>2</sup>:

$0 \leq R <  B $	
$Q_E = (A - R)/B$	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:

$A \text{ div}_T(-B) = (-A) \text{ div}_T B = -(A \text{ div}_T B)$
$A \text{ mod}_T(-B) = A \text{ mod}_T B$

F-division or floor division is described and promoted by Knuth<sup>3</sup>. The MATLAB mod function is based on F-division.

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

$A \text{ div}_E(-B) = -(A \text{ div}_E B)$
$A \text{ mod}_E(-B) = A \text{ mod}_E B$

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:

<b>A,B)</b>	<b>(QT ,RT)</b>	<b>(QF,RF)</b>	<b>(QR,RR)</b>	<b>(QC,RC)</b>	<b>(QE,RE)</b>
(+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

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

Warning: block 'latency/DivMod': Zero denominator!

# Synplify DSP Downsample

Decreases the sample rate of the input by removing samples.

## Library

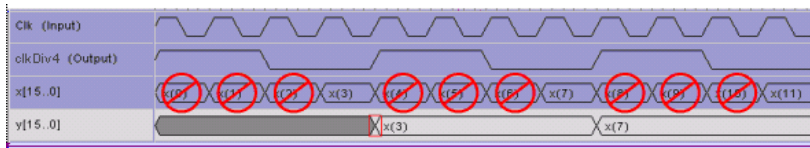
Synplify DSP [Signal Operations](#)

## Description

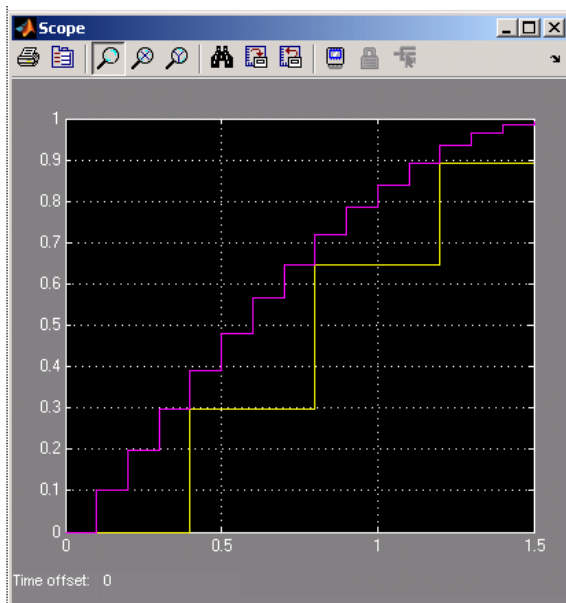


The Synplify DSP Downsample block decreases (downsamples) the sample rate of the input by removing samples. If  $M$  is the specified downsampling rate, for every  $M$  samples at the input, the software keeps 1 sample at the output. This means that the sample rate at the output is the input sample rate 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.

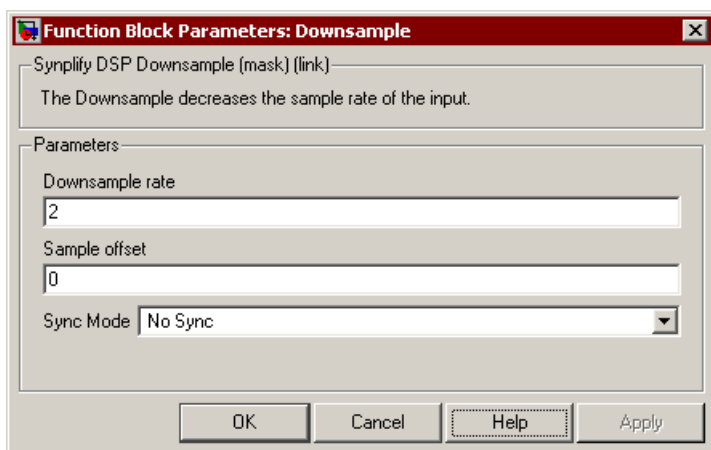
For information about using the Downsample block in multi-rate designs, see [Multi-Rate Design, on page 3-25](#).

## 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



### Downsampling 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 3-25](#).

### 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



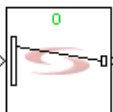
# Synplify DSP Extract

Extracts specified bits from the input signal.

## Library

Synplify DSP [Signal Operations](#)

## Description



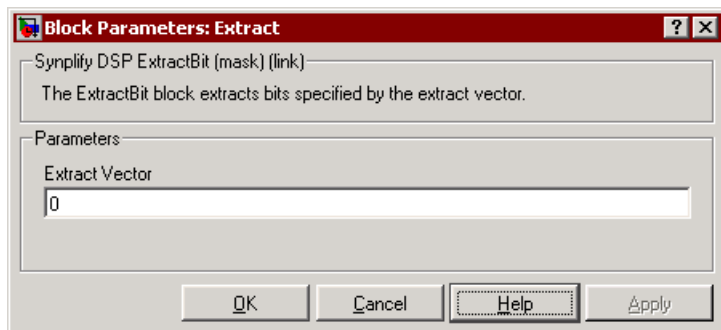
The Extract block extracts specified bits from the input signal. The output is unsigned, but you can recast the output using the Recast block ([Synplify DSP Recast](#), on page 8-196).

## 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 <a href="#">Extract Vector</a> , on page 8-110 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 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)`

Examples of usage:

<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>	

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.

# Synplify DSP FDATool

Opens the Simulink FDATool interface.

## Library

Synplify DSP [Filtering](#)

## Description



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

The Synplify DSP blockset includes the [Synplify DSP FIR](#) and [Synplify DSP IIR](#) blocks, but this block provides an interface to the FDATool software, which is part of the MATLAB® Signal Processing Toolbox. The Synplify DSP 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 Synplify DSP model.

For procedural information about using this block, see [Defining FIR Filter Coefficients with FDATool](#), on page 4-15.

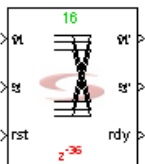
# Synplify DSP FFT

Implements a fully pipelined Fast Fourier Transform.

## Library

Synplify DSP [Transforms](#)

## Description



The Synplify DSP FFT block implements a fully pipelined Fast Fourier Transform. When it is used to perform the blockwise FFT of a streaming signal, you only require a reset for 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

Transform Size	Latency (Bit-reversed Output)	Latency (Natural Order Output)
16384	16402	32787
32768	32789	65558
65536	65557	131094

## FFT Parameters

**Function Block Parameters: FFT**

Synplify DSP FFT (mask) (link)

The FFT block is a fully pipelined Fast Fourier Transform.

Parameters

Transform size: 16

Transform direction: Forward

☐ Real FFT

Scaling: No Scaling

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

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 8-289](#) for details.

N is the FFT size.

- Scale by 1/N divides the DFT summation by N.
- Scale by 1/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 8-289](#) 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 8-289](#) 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 8-112](#) 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a> . 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 <a href="#">Special Variables, on page 8-292</a> .
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> . 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 <a href="#">Special Variables, on page 8-292</a> .
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## 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 <a href="#">Overflow Saturation Options, on page 8-289</a> 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 <a href="#">Underflow Rounding Options, on page 8-289</a> 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.

# Synplify DSP FIFO

Implements a single-rate or multirate FIFO (First in First Out) memory queue.

## Library

Synplify DSP [Memories](#)

## Description



The Synplify DSP FIFO block implements a single-rate or multi-rate FIFO memory queue. The major application of FIFO is 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 which 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 or 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.

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.

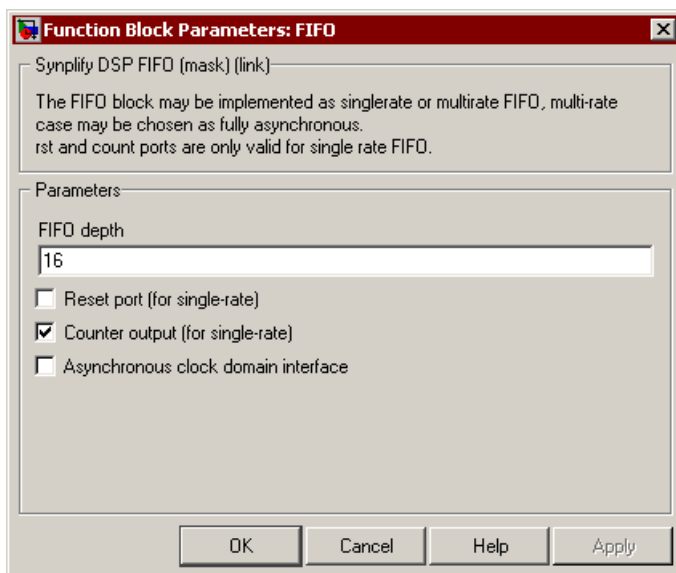
The count output indicates the number of items in the FIFO queue. You can use the count output to implement buffer management algorithms.

## 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.

### Asynchronous clock domain interface

When enabled, it implements an asynchronous FIFO. Enable this option for multi-rate FIFOs only.

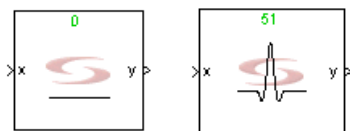
# Synplify DSP FIR

Implements a finite impulse response (FIR) filter.

## Library

Synplify DSP [Filtering](#)

## Description



The Synplify DSP FIR block implements a finite impulse response filter, using either a Transposed Form FIR or a Direct 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 4-14](#). To implement adaptive or reloadable FIRs, use the RFIR custom block, which is described in [Synplify DSP RFIR, on page 8-213](#).

## Micro-Architectural Optimizations

For each implementation, the Synplify DSP tool explores various micro-architectures (MAs) and chooses an optimal configuration based on the target device, sample rates, and the system wide optimization settings (retiming, folding, multi-channelization). It chooses a micro-architecture that meets timing and minimizes area.

For the FIR block, the tool considers the direct form and transposed form MAs, and explores ripple-carry or carry-save adder as options for the adder tree. It also simultaneously considers exploiting positive and negative symmetric coefficients and includes optimizations for values of zero, powers of 2, or shifted versions of other coefficients. Retiming automatically determines if pipelining is required, and the tool also considers these factors during its exploration. The result is a highly optimized implementation specific to the target technology that meets timing, minimizes area, and is bit and cycle accurate with the simulation.

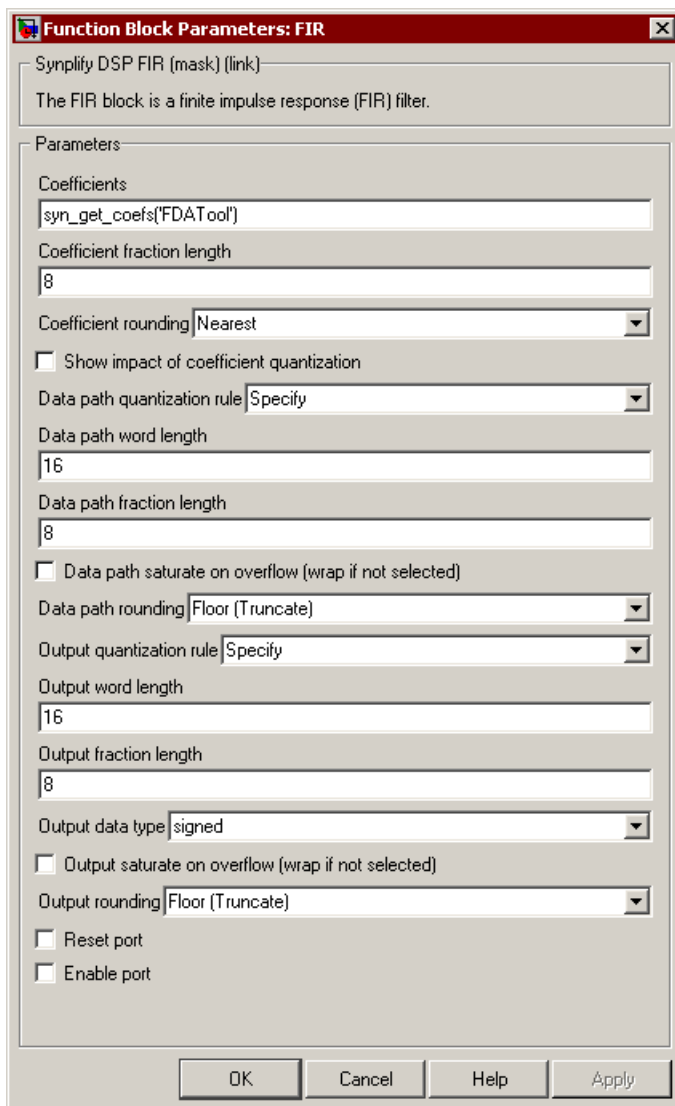
## Icon Annotations

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.

---

## FIR Parameters



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

- A link for "Synplify DSP FIR (mask) (link)".
- A description: "The FIR block is a finite impulse response (FIR) filter."
- A "Parameters" section with the following controls:
  - "Coefficients" text box containing `syn_get_coefs('FDATool')`.
  - "Coefficient fraction length" text box containing `8`.
  - "Coefficient rounding" dropdown menu set to "Nearest".
  - Checkbox for "Show impact of coefficient quantization" (unchecked).
  - "Data path quantization rule" dropdown menu set to "Specify".
  - "Data path word length" text box containing `16`.
  - "Data path fraction length" text box containing `8`.
  - Checkbox for "Data path saturate on overflow (wrap if not selected)" (unchecked).
  - "Data path rounding" dropdown menu set to "Floor (Truncate)".
  - "Output quantization rule" 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 for "Output saturate on overflow (wrap if not selected)" (unchecked).
  - "Output rounding" dropdown menu set to "Floor (Truncate)".
  - Checkbox for "Reset port" (unchecked).
  - Checkbox for "Enable port" (unchecked).
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

### Coefficients

Specifies the filter coefficients. Provide a vector or matrix 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 4-15 for information about extracting coefficients, and [syn\\_get\\_coefs](#), on page 9-4 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 8-292.

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 8-289 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 8-292.

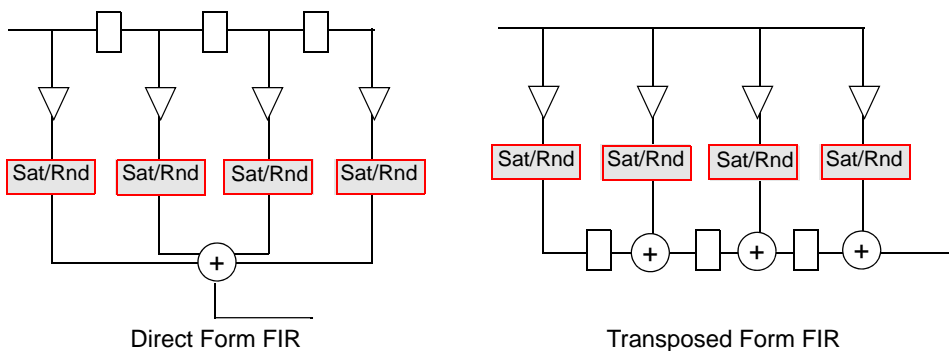
### Data path fraction length

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 8-292.

### 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 8-289 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 8-289](#) 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:

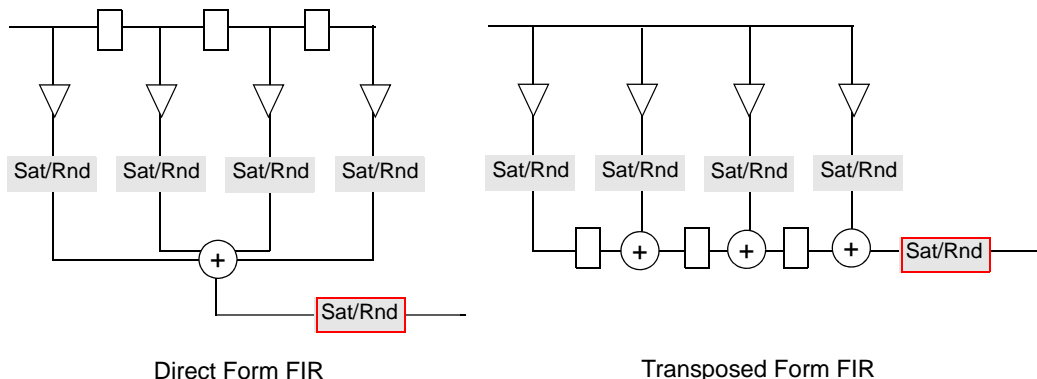
Output word length	<a href="#">Output Word Length, on page 8-288</a> You can also specify word length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , and <code>syn_guard_bit</code> variables, which are described in <a href="#">Special Variables, on page 8-292</a> .
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## 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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output rounding	See <a href="#">Underflow Rounding Options, on page 8-289</a> 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.

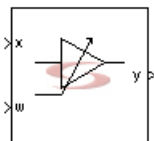
# Synplify DSP FIR Engine

Implements an FIR filter using coefficients that are input as vectors.

## Library

Synplify DSP [Filtering](#)

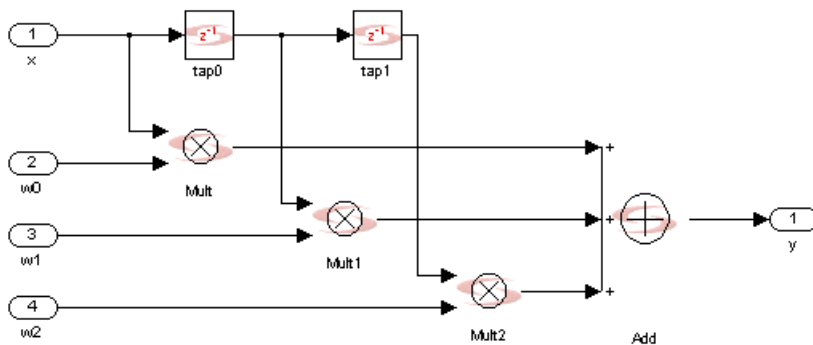
## Description



The Synplify DSP 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.



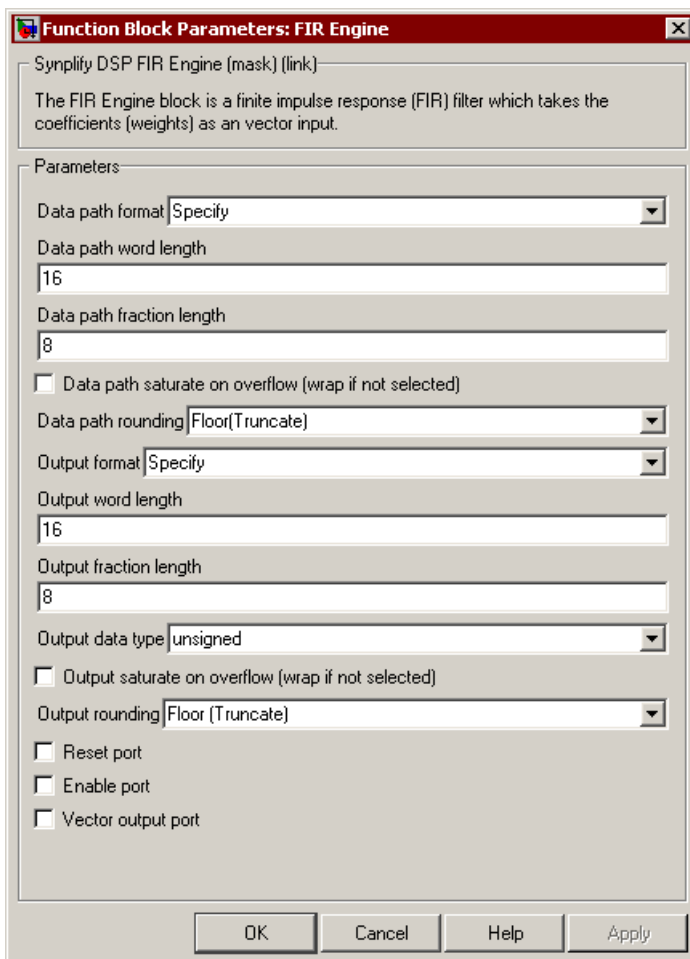
## Folding

In order to use FIR Engine in a folded design, each instance should be able to receive 2 latencies.

## Latency

This block does not introduce any latency.

## FIR Engine Parameters



The dialog box titled "Function Block Parameters: FIR Engine" contains a description and a list of parameters. The description states: "The FIR Engine block is a finite impulse response (FIR) filter which takes the coefficients (weights) as an vector input." The parameters section includes:

- Data path format: Specify (dropdown)
- Data path word length: 16 (text box)
- Data path fraction length: 8 (text box)
- ☐ Data path saturate on overflow (wrap if not selected)
- Data path rounding: Floor(Truncate) (dropdown)
- Output format: Specify (dropdown)
- Output word length: 16 (text box)
- Output fraction length: 8 (text box)
- Output data type: unsigned (dropdown)
- ☐ Output saturate on overflow (wrap if not selected)
- Output rounding: Floor (Truncate) (dropdown)
- ☐ Reset port
- ☐ Enable port
- ☐ Vector output port

At the bottom are buttons for 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 8-292.

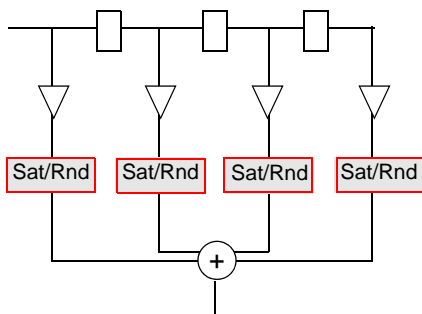
### Data path fraction length

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 8-292.

### 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 8-289 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 8-289 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:

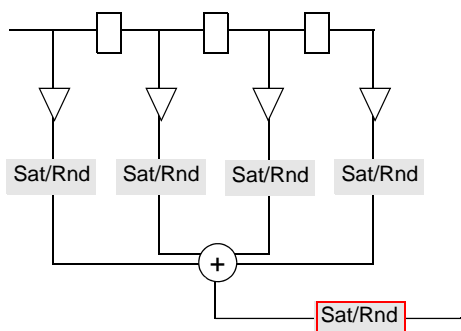
Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a> You can also specify word length in terms of the <code>syn_inp_wl</code> , <code>syn_inp_fl</code> , <code>syn_inp_dt</code> , <code>syn_coef_wl</code> , <code>syn_coef_fl</code> , <code>syn_coef_dt</code> , and <code>syn_guard_bit</code> variables, which are described in <a href="#">Special Variables, on page 8-292</a> .
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## 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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output rounding	See <a href="#">Underflow Rounding Options, on page 8-289</a> 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.



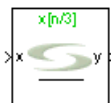
# Synplify DSP FIR Rate Converter

Implements a polyphase FIR filter by inserting upsamplers and downsamplers.

## Library

Synplify DSP [Filtering](#)

## Description



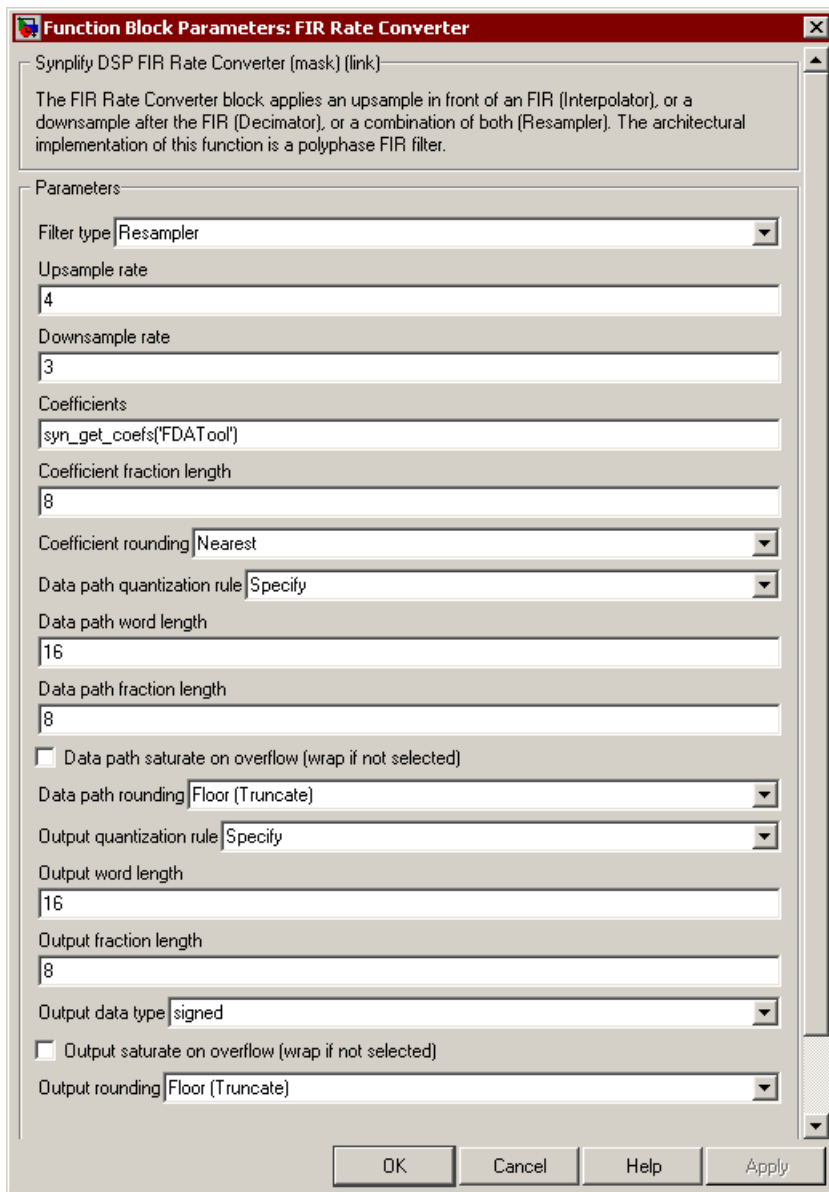
The Synplify DSP FIR Rate Converter is a custom block that implements a polyphase filter, by using upsample and downsample blocks with the FIR block to implement interpolators, decimators, and resamplers. See [Primitives and Custom Blocks, on page 5-2](#) for information about custom blocks.

You can use the FIR block to implement polyphase decimators and interpolators: see [Implementing Polyphase FIR Filters, on page 4-14](#) for details. This block supports vector input.

## Latency

This block has zero latency.

## FIR Rate Converter Parameters



**Function Block Parameters: FIR Rate Converter**

Synplify DSP 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

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)

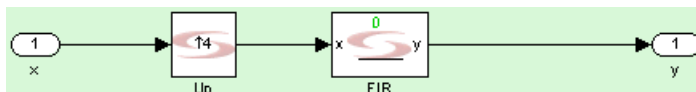
Output rounding: Floor (Truncate)

OK Cancel Help Apply

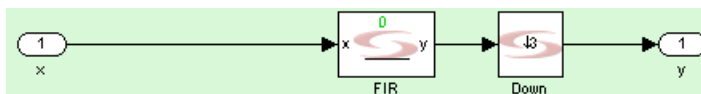
## Filter Type

Specifies the type of polyphase filter to be implemented.

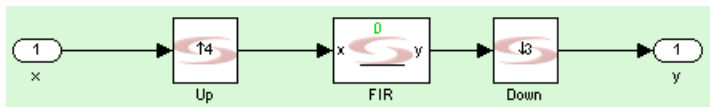
- Interpolator applies an Upsample block before an FIR to implement the polyphase filter.



- 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 4-15](#) for information about extracting coefficients, and [syn\\_get\\_coefs, on page 9-4](#) 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 8-292](#).

### Coefficient rounding

Determines how the coefficient value is rounded. See [Underflow Rounding Options, on page 8-289](#) for details.

### 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 8-292](#).

### 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 8-292](#).

### Data path saturation on overflow

Determines how data path overflow is treated. See [Overflow Saturation Options, on page 8-289](#) 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 8-289](#) 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## Output saturation on overflow

Determines how output overflow is treated. See [Overflow Saturation Options, on page 8-289](#) 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 8-289](#) for details. This option is only available when you set Output quantization rule to Specify.

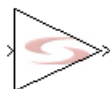
# Synplify DSP Gain

Implements a constant gain to the input.

## Library

Synplify DSP [DSP Basics](#) and Synplify DSP [Math Functions](#)

## Description



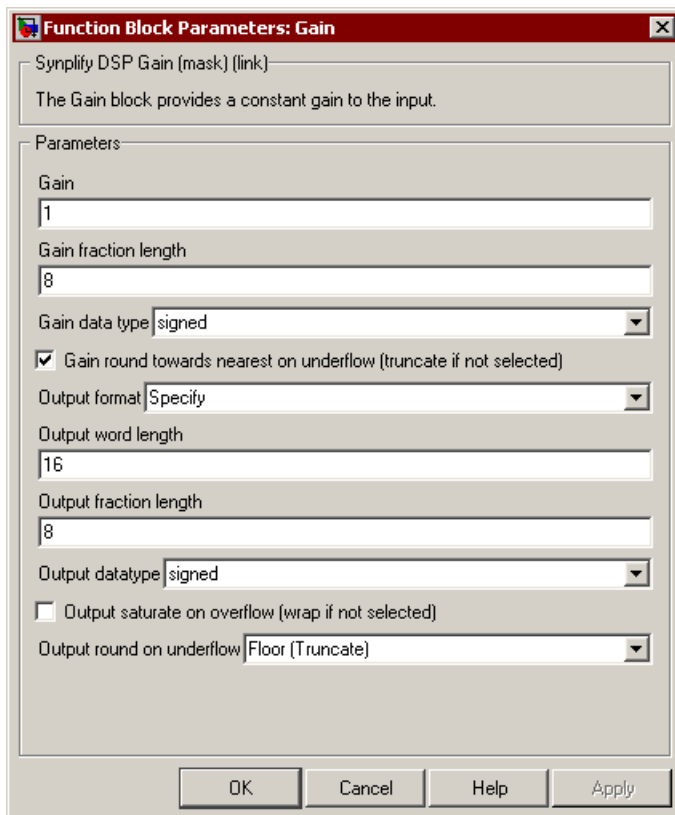
The Synplify DSP Gain block provides a constant gain by multiplying the input by the specified gain factor. For trivial gain values, the software does the following optimizations:

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)

## Latency

This block has no latency.

## Gain Parameters



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

- A link labeled "Synplify DSP Gain (mask) (link)".
- A description: "The Gain block provides a constant gain to the input."
- A "Parameters" section with the following controls:
  - "Gain": A text field containing the value "1".
  - "Gain fraction length": A text field containing the value "8".
  - "Gain data type": A dropdown menu set to "signed".
  - A checked checkbox labeled "Gain round towards nearest on underflow (truncate if not selected)".
  - "Output format": A dropdown menu set to "Specify".
  - "Output word length": A text field containing the value "16".
  - "Output fraction length": A text field containing the value "8".
  - "Output datatype": A dropdown menu set to "signed".
  - An unchecked checkbox labeled "Output saturate on overflow (wrap if not selected)".
  - "Output round on underflow": A dropdown menu set to "Floor (Truncate)".
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

### Gain

Specifies the factor by which the input is multiplied to implement the gain. If the input to the block is vectorized, you can specify the gain value as a column vector, with different gain factors for each channel. The number of rows should be equal to the input vector size.

### 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 8-289](#) for details.

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output round on underflow	Uses the specified algorithm to round the underflow; see <a href="#">Underflow Rounding Options, on page 8-289</a> for descriptions of the algorithms.



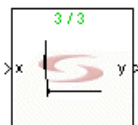
# Synplify DSP IIR

Implements an infinite impulse response (IIR) filter.

## Library

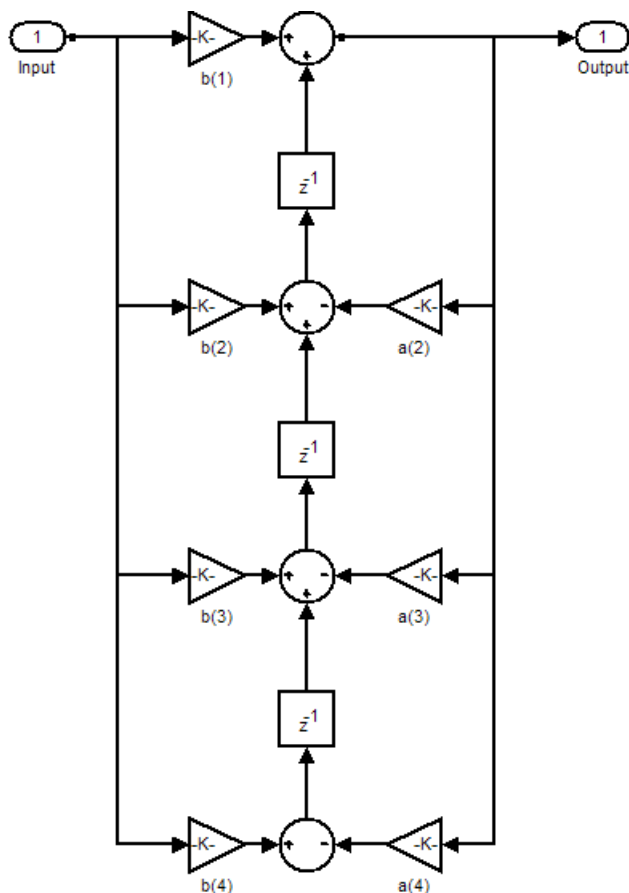
Synplify DSP [Filtering](#)

## Description



The Synplify DSP 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:

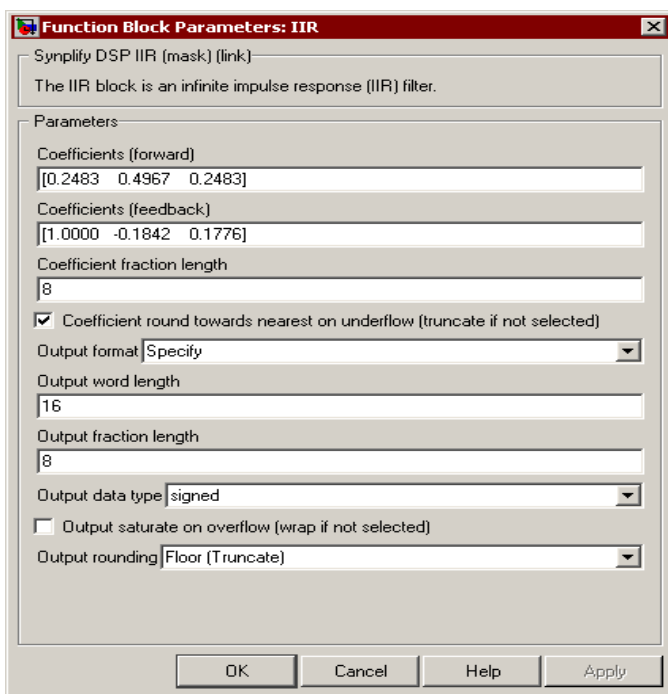
- Both forward and feedback coefficients are row vectors (dim:1xN) and the input is a one-dimensional signal. This is normal operation.

- 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]

## Latency

This block has no latency.

## IIR Parameters



The image shows a dialog box titled "Function Block Parameters: IIR". It contains a description of the block and a list of parameters for configuration.

Synplify DSP IIR (mask) (link)  
The IIR block is an infinite impulse response (IIR) filter.

Parameters

Coefficients (forward)  
[0.2483 0.4967 0.2483]

Coefficients (feedback)  
[1.0000 -0.1842 0.1776]

Coefficient fraction length  
8

☒ Coefficient round towards nearest on underflow (truncate if not selected)

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)

Buttons: OK, Cancel, Help, 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 8-142](#).
- 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 8-142](#) 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 IIR Filter Coefficients with `FDATool`, on page 4-19](#). See `syn_get_coefs`, on [page 9-4](#) 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 8-142](#) 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 8-144](#).

**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 [Underflow Rounding Options, on page 8-289](#) for details.

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a> . Do not use the output format to change the data type. See <a href="#">Description, on page 8-141</a> for details.
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

**Output saturate on overflow, Output rounding**

Determine how overflow and underflow are treated. For descriptions of these parameters, see the following:

Outputsaturate on overflow	Saturates (option enabled) or wraps (option disabled) the overflow. See <a href="#">Overflow Saturation Options, on page 8-289</a>
Output rounding	Uses the specified algorithm to round the underflow; see <a href="#">Underflow Rounding Options, on page 8-289</a> .

# Synplify DSP In

Allows you to add an in port to a subsystem to a Synplify DSP design.

## Library

Synplify DSP [Ports & Subsystems](#)

## Description



The Synplify DSP In block provides an in port for a subsystem added to a Synplify DSP 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.

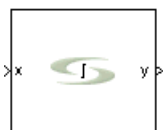
# Synplify DSP Integrator

Performs a discrete time integration of the input signal.

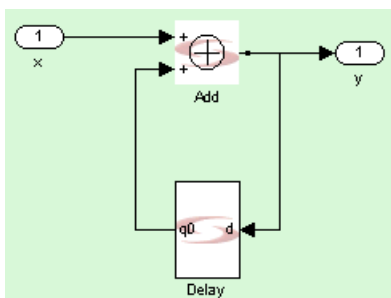
## Library

Synplify DSP [Filtering](#)

## Description



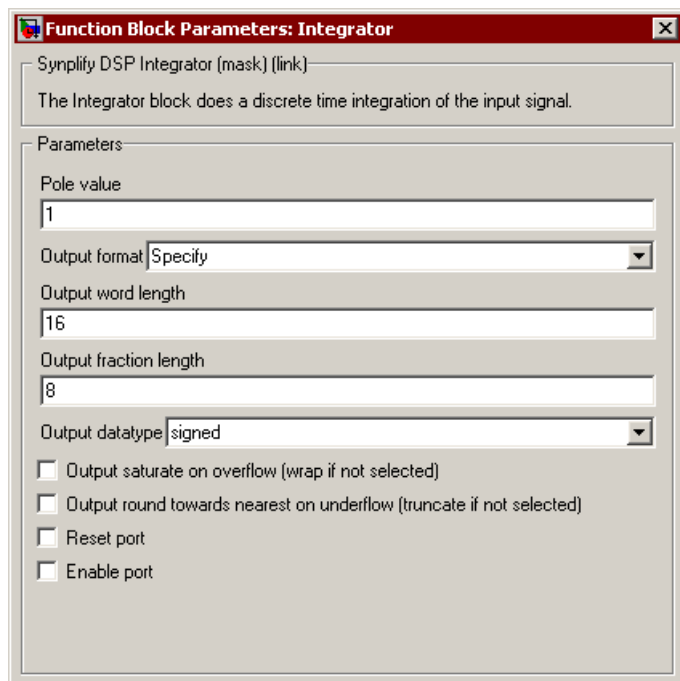
This custom block (see [Primitives and Custom Blocks](#), on page 5-2 for a definition) performs a discrete time integration of the input signal.



## Latency

This block has no latency.

## Integrator Parameters



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

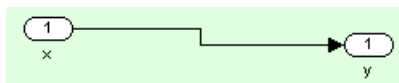
- A header bar with a red background and a close button (X).
- A tab labeled "Synplify DSP Integrator (mask) (link)".
- A description box stating: "The Integrator block does a discrete time integration of the input signal."
- A "Parameters" section with the following controls:
  - "Pole value": A text 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

### Pole Value

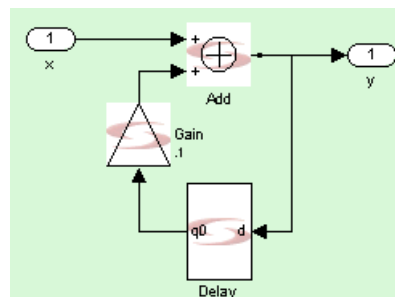
Determines how the block is implemented. The pole value must be between 0 and 1.

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.

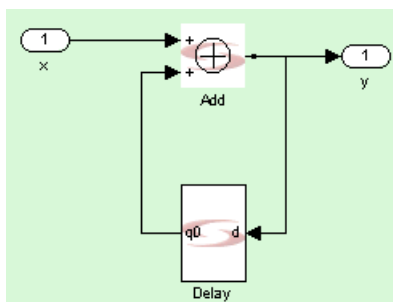




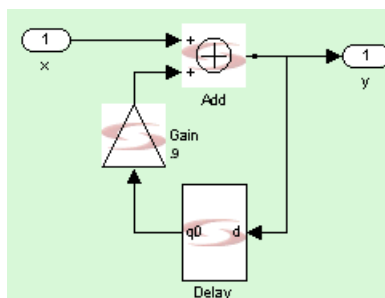
Pole Value = 0



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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> .
-------------------------------	---

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 <a href="#">Underflow Rounding Options, on page 8-289</a> 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.

# Synplify DSP Inverter

Calculates the inverse (one's complement) of the input.

## Library

Synplify DSP [Math Functions](#)

## Description

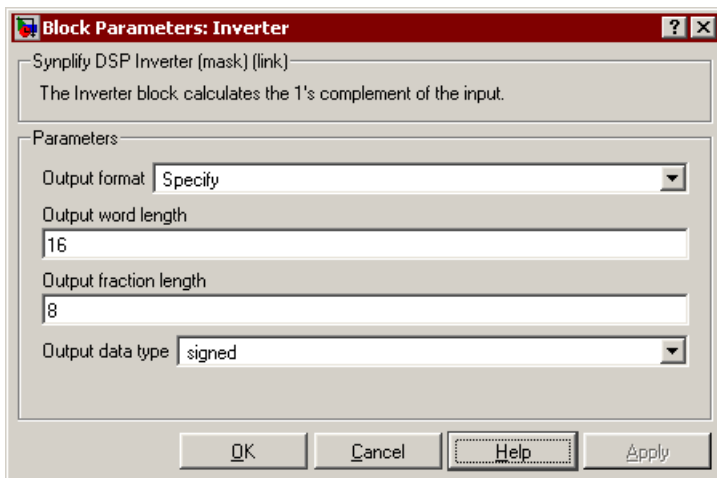


The Synplify DSP Inverter block calculates the one's complement of the input, which is a bitwise inversion of the input.

## Latency

This block has no latency.

## Inverter Parameters



For descriptions of the parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

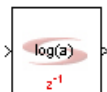
# Synplify DSP Log

Calculates the natural logarithm of the input.

## Library

Synplify DSP [Math Functions](#)

## Description

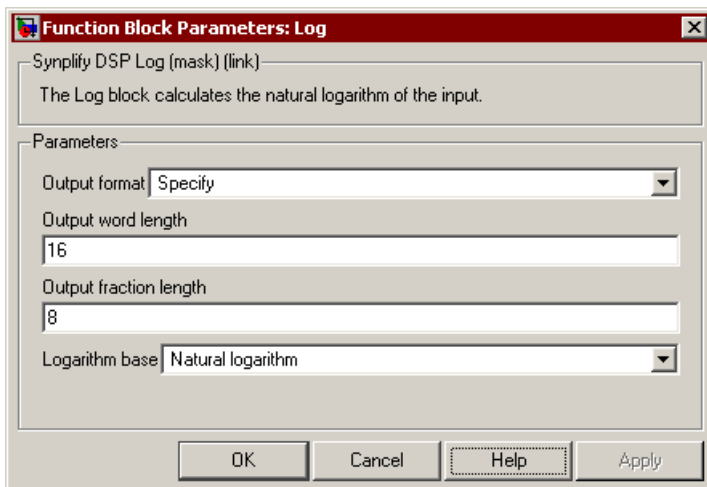


The Synplify DSP Log block calculates the natural logarithm of the input. This implementation is based on a look-up table, the size of which is determined by the input fraction length and the output word length.

## 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	<a href="#">Output Format, on page 8-287</a> 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	<a href="#">Output Word Length, on page 8-288</a> 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	<a href="#">Output Fraction Length, on page 8-288</a> 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

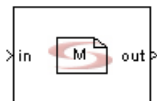
# Synplify DSP M Control

Provides control logic using an M function.

### Library

Synplify DSP [Control Logic](#)

### Description



The Synplify DSP 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 MATLAB's M language. For details on writing the M functions, see [Using M Control Blocks, on page 6-1](#).

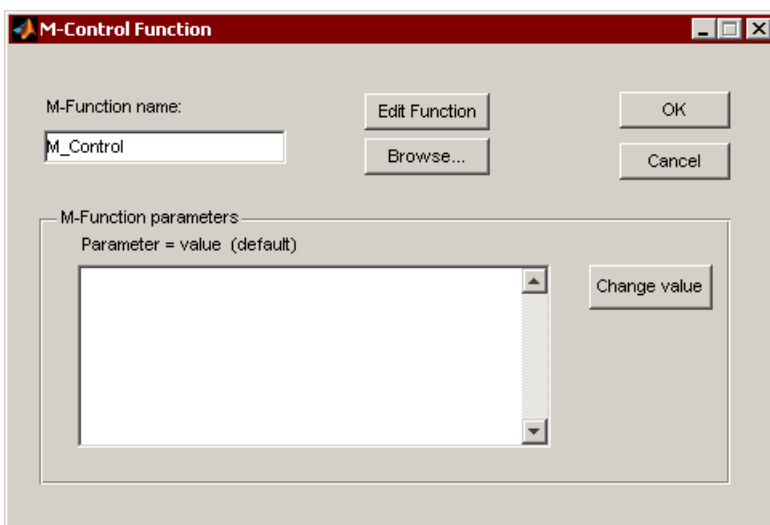
At every simulation tick, the block converts the fixed-point data at the block inputs to double, executes the M-control function on this double data, and then converts the output double data to fixed-point again for the rest of the model. This implementation improves simulation times.

### Latency

Each output of this block can have either no latency or a latency of one.

**Latency Description**

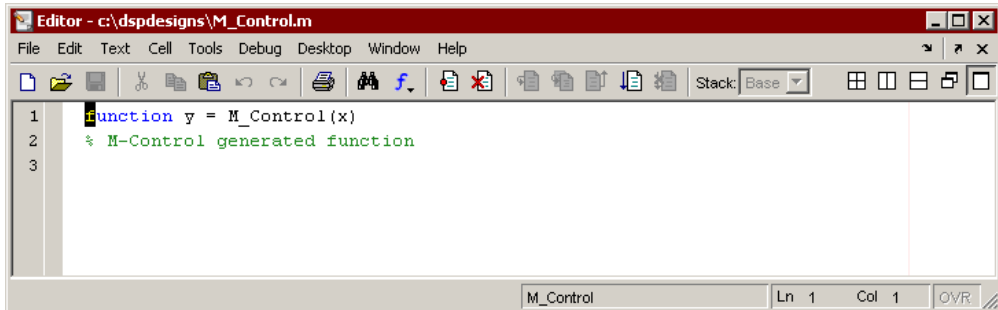
1	An output referencing one or more state-holding elements inferred from persistent variables has a latency of one.
0	In all other cases the output has no latency.

**M Control Parameters****M 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 Control Blocks, on page 6-1](#).

**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.

## 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 6-27](#).

## Change Value

Lets you override the value of the selected parameter in the parameter list. See [Overridable Parameters, on page 6-27](#) for details about defining overridable parameters.



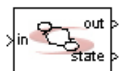
# Synplify DSP Mealy State Machine

Provides control logic where the output depends on the input and an internal state vector.

## Library

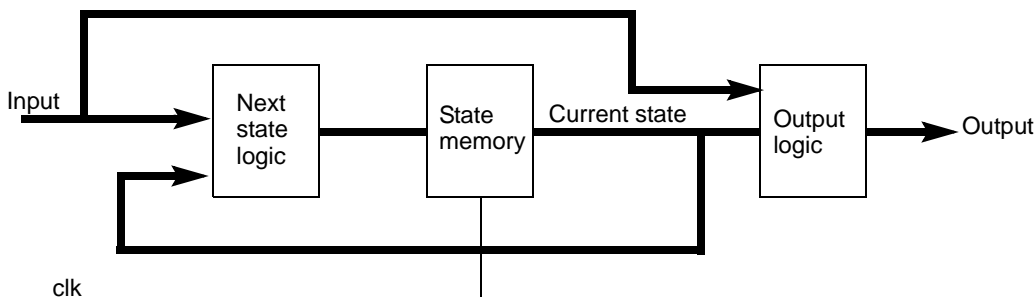
Synplify DSP [Control Logic](#)

## Description



The Synplify DSP 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:

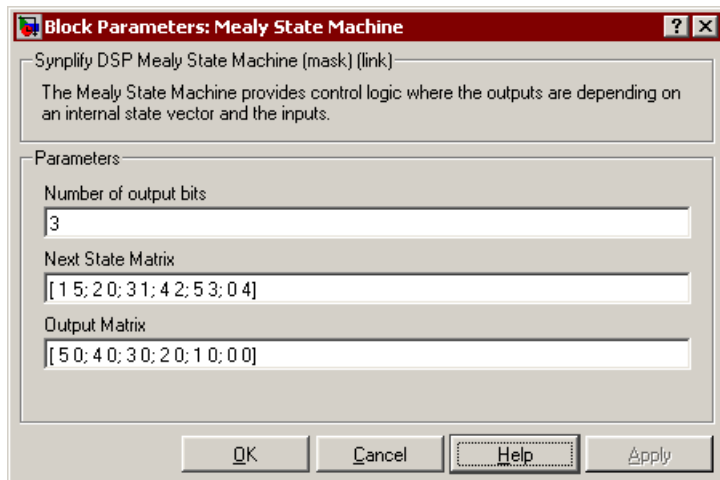
Current State	Input = 0	Input = 1	Next State Matrix	Output Matrix
0	1, 5	5, 0	1 5	5 0
1	2, 4	0, 0	2 0	4 0
2	3, 3	1, 0	3 1	3 0
3	4, 2	2, 0	4 2	2 0
4	5, 1	3, 0	5 3	1 0
5	0, 0	4, 0	0 4	0 0

Diagram illustrating the Mealy State Machine logic. The Current State (0-5) is mapped to Next State and Output values for Input = 0 and Input = 1. The Next State Matrix and Output Matrix are shown. Arrows indicate the flow from the Current State to the Next State and Output.

## 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.

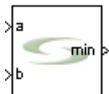
# Synplify DSP MinMax

Determines the minimum, maximum, or minimum and maximum of two inputs.

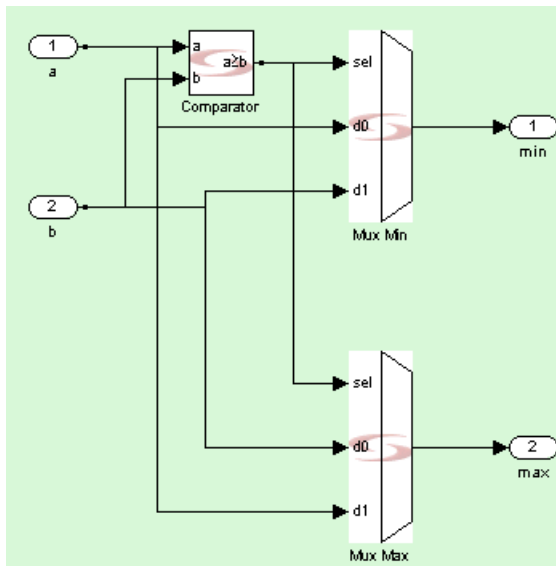
## Library

Synplify DSP [Math Functions](#)

## Description



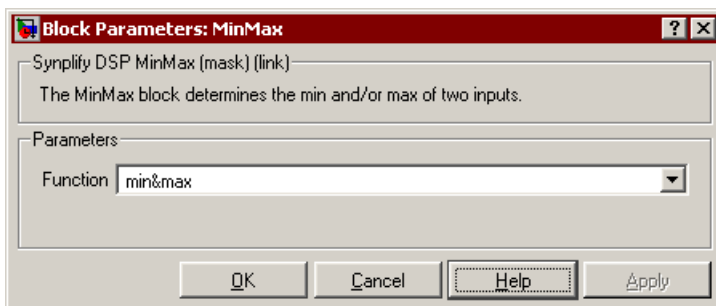
This custom block (see [Primitives and Custom Blocks, on page 5-2](#) for a definition) determines the minimum, maximum, or minimum and maximum of two inputs.



## 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.

# Synplify DSP Moore State Machine

Provides control logic where the outputs depend on the current state.

## Library

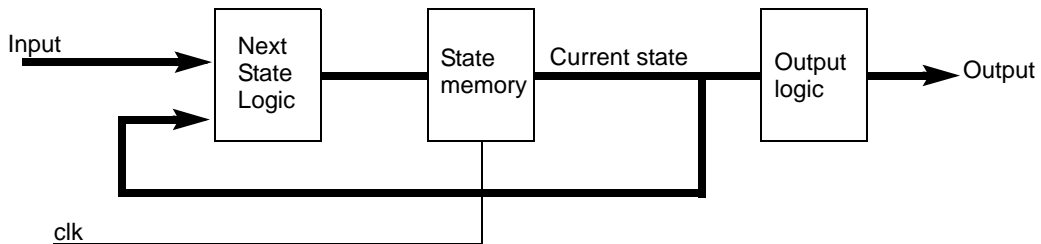
Synplify DSP [Control Logic](#)

## Description



The Synplify DSP 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:

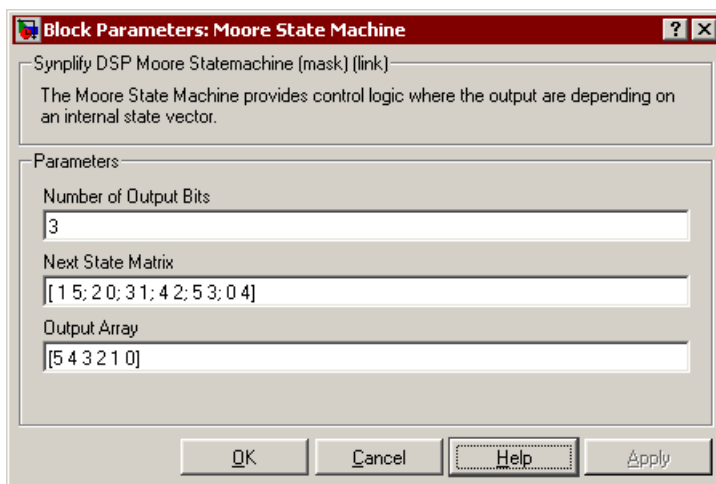
State	Input = 0	Input = 1	Output	Next State Matrix	Output Matrix
0	1	5	5	1 5	5
1	2	0	4	2 0	4
2	3	1	3	3 1	3
3	4	2	2	4 2	2
4	5	3	1	5 3	1
5	0	4	0	0 4	0

Diagram illustrating the state transitions and outputs for the Moore State Machine. The state vector (0 to 5) is shown. The Next State Matrix and Output Matrix are also displayed. Arrows indicate the mapping from the state vector to the matrices: a red arrow points from the state vector to the Next State Matrix, and a blue arrow points from the state vector to the Output Matrix.

## Latency

This block has no latency.

## Moore 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 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 8-162](#)). 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.

## Synplify DSP Mult

Implements a full-precision multiplier.

### Library

Synplify DSP [Math Functions](#)

### Description



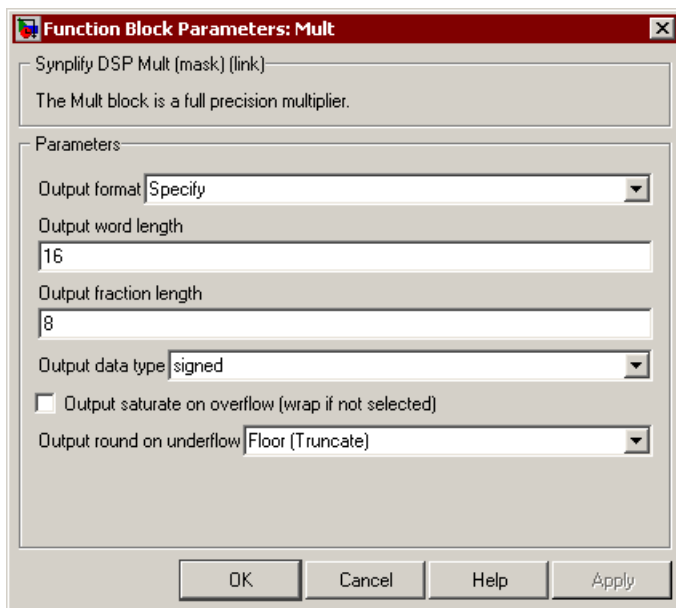
The Synplify DSP Mult block implements a full-precision multiplier. It computes the product of the data on its two input ports, and feeds it to the output port. The inputs can be of different word lengths, as determined by their drivers. The output word length is the sum of input word lengths. Similarly, the output fraction length is the sum of the input fraction lengths.

### 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
-----------------------------	---

Output round on underflow	Uses the specified algorithm to round the underflow; see <a href="#">Underflow Rounding Options, on page 8-289</a> for descriptions of the algorithms.
---------------------------	--

# Synplify DSP Mux

Implements a multiplexer of up to 32 inputs.

**Library**

Synplify DSP [Signal Operations](#)

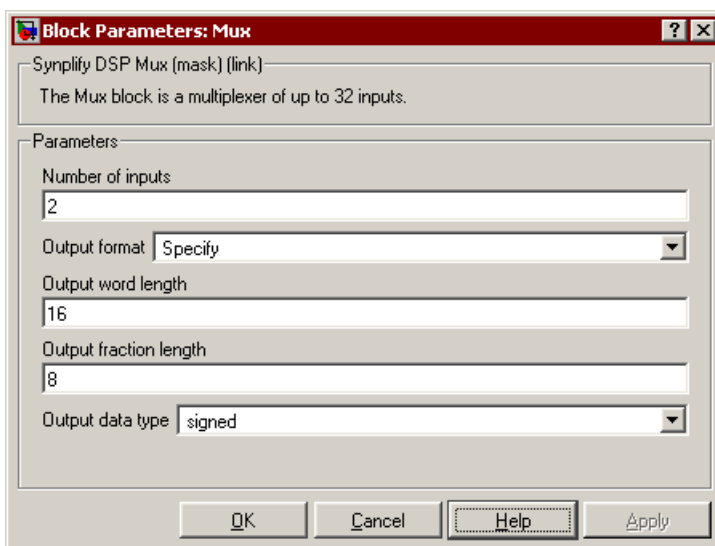
**Description**

The Synplify DSP Mux block implements a multiplexer of up to 32 inputs. The sel input determines which of the data inputs gets multiplexed into the output.

**Latency**

This block has no latency.

## Mux Parameters



### Number of inputs

Sets the number of inputs that are multiplexed. You can specify up to 32 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).

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

# Synplify DSP Negate

Computes the two's complement (arithmetic negation) of an input.

## Library

Synplify DSP [Math Functions](#)

## Description

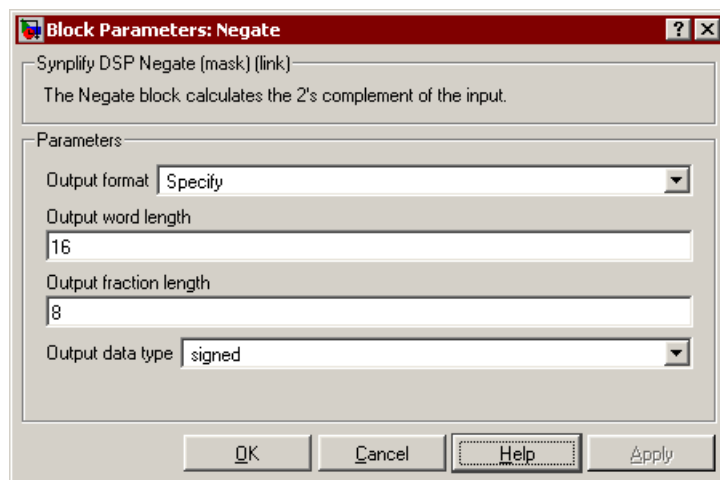


The Synplify DSP Negate block takes the two's complement of a signed input. For a signed value, this means that it multiplies the input by -1.

## Latency

This block has no latency.

## Negate Parameters



For descriptions of the parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

# Synplify DSP Out

Allows you to add an out port to a subsystem to a Synplify DSP design.

## Library

Synplify DSP [Ports & Subsystems](#)

## Description



The Synplify DSP Out block provides an out port for a subsystem in a Synplify DSP 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.

# Synplify DSP 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

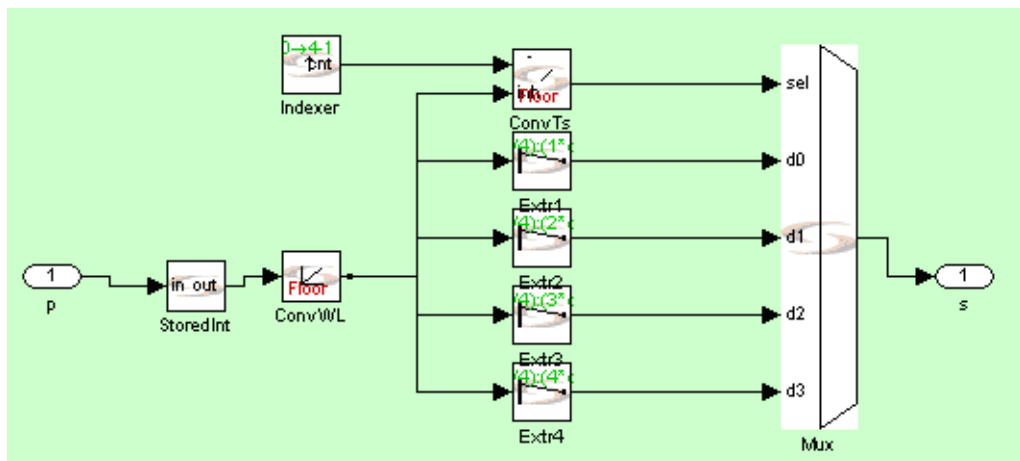
Synplify DSP [Signal Operations](#)

## 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 5-2 for a definition.) The following figure shows how the block is modeled:



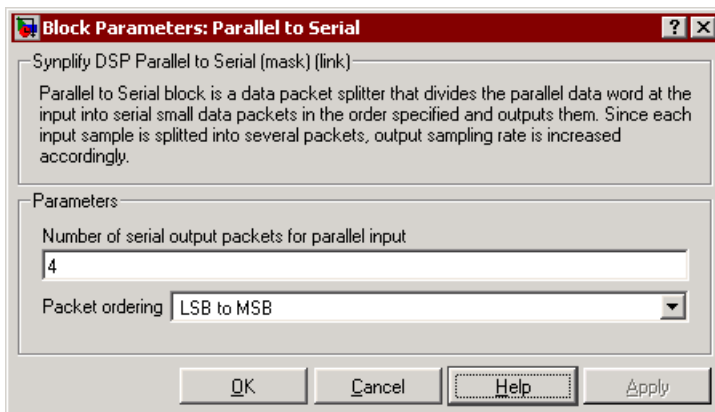
## Icon Annotations

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.

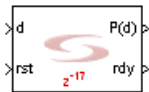
# Synplify DSP Permutation

Shuffles the incoming data according to a specified permutation vector.

## Library

Synplify DSP [Memories](#)

## Description



The Synplify DSP Permutation block shuffles the incoming data according to a specified permutation vector, and produces a streaming output with a delay equal to  $1 + \langle \text{minimum possible delay} \rangle$ . The software sets the order of the outgoing data stream by presenting a permutation of the incoming data stream  $d[i]$ .

When  $\text{rst}$  is 1, the software clears the buffer and resets the frame boundary. When it is used to perform blockwise permutation of a streaming signal,  $\text{rst}$  must be only applied to the first block. The  $\text{rdy}$  signal is the  $\text{rst}$  signal delayed by the input-to-output latency. You can use the  $\text{rdy}$  signal for synchronization and/or latency measurement.

## Icon Annotations

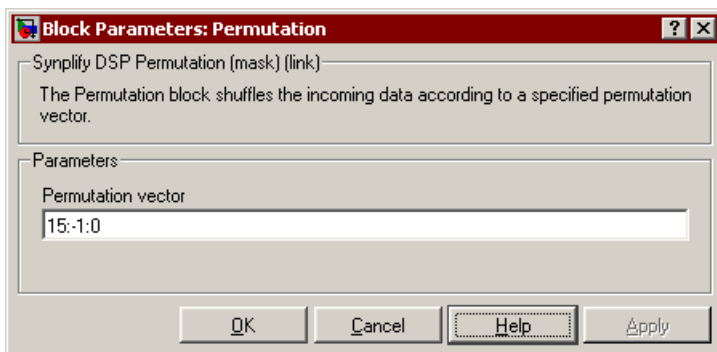
The icon for this block displays the following information:

Latency	For permutation ( $P_0 P_1 \dots P_n$ ), the latency is $\max(P_{j-i}) + 2.s$
---------	---

---



## 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.

# Synplify DSP Port In

Defines inputs for the DSP design to be implemented in RTL.

## Library

Synplify DSP [Ports & Subsystems](#)

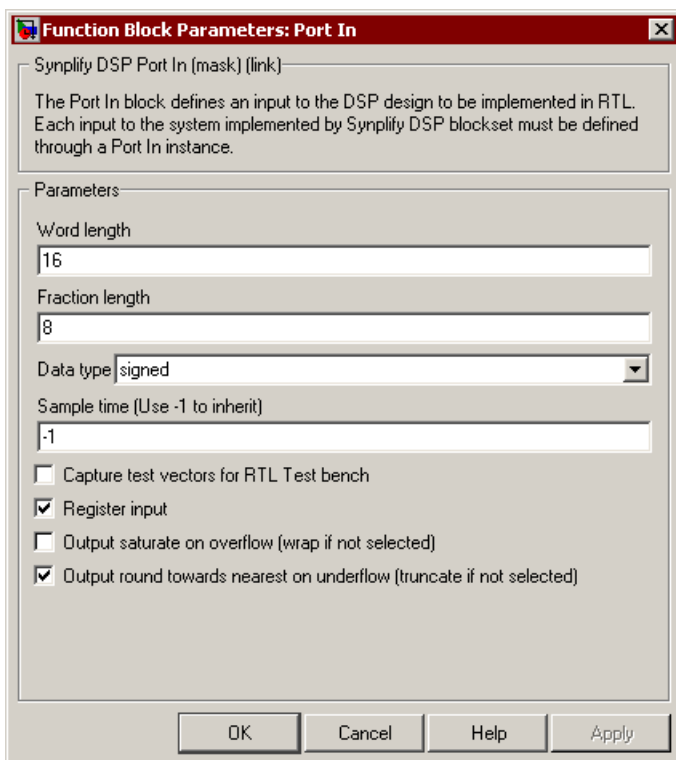
## Description



The Synplify DSP Port In block defines an input to the design to be implemented in RTL. Each input to the subsystem implemented by the Synplify DSP 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.

Do not use this block to define the boundaries of a subsystem. Use the Synplify DSP In block instead (see [Synplify DSP In, on page 8-146](#)).

## Port In Parameters



### 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.

**Register input**

When enabled, registers the input. With registered input, the block has a latency of 1.

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> 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
<b>Rounding Off (truncating)</b>	2.375	010.011	sfix5_En2	010.01	2.25
<b>Saturation Off (wrapping)</b>	128	010000000	sfix8_En0	10000000	-128

	Input Data		Output Data		
<b>Rounding On Saturation Off</b>	3.875	011.111	sfix5_En2	100.00	-4
	-2.875	101.001	sfix5_En2	101.01	-2.75
<b>Rounding Off Saturation On</b>	128	010000000	sfix8_En0	01111111	127
	2.375	010.011	sfix4_En2	01.11	1.75
<b>Rounding On Saturation On</b>	128.375	010000000.011	sfix9_En1	01111111.1	127.5
	3.875	011.111	sfix5_En2	011.11	3.75

# Synplify DSP Port Out

Defines outputs for the DSP design to be implemented in RTL.

## Library

Synplify DSP [Ports & Subsystems](#)

## Description



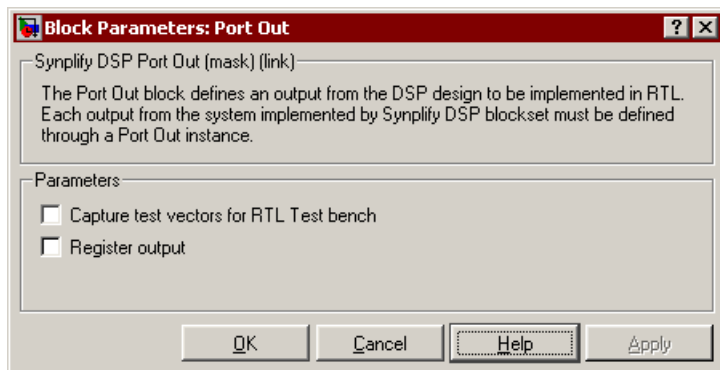
The Synplify DSP Port Out block defines an output of the design to be implemented in RTL. Each output to the subsystem implemented by the Synplify DSP blockset must be defined with a Port Out block. It can also capture data passing through it and store it for the RTL test bench.

Do not use this block to define the boundaries of a subsystem. Use the Synplify DSP Out block instead (see [Synplify DSP Out, on page 8-169](#)).

## 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.

### Register output

When enabled, registers the output. With registered output, the block has a latency of 1.

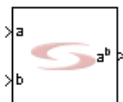
# Synplify DSP Pow

Raises a value to the power of another value.

## Library

Synplify DSP [Math Functions](#)

## Description



The Pow block raises a value to the power of another value.

## Pow Parameters

**Constant Base**

When enabled, it lets you specify a base value and fraction length.

**Base Value**

Specifies the constant base value. This is available when you enable Constant Base.

**Base Fraction Length**

Specifies the fraction length for the constant base. This is available when you enable Constant Base.

**Constant Exponent**

When enabled, it lets you specify an exponent value.

**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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>



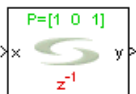
# Synplify DSP Puncture

Removes user-specified bits from the input data stream.

## Library

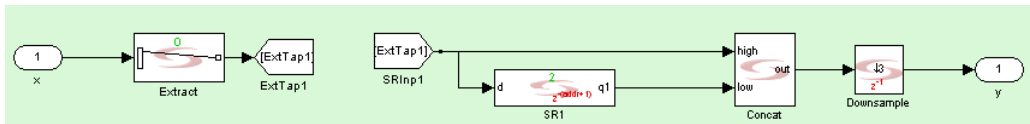
Synplify DSP [Communications](#)

## Description



The Puncture block removes the set of bits you specify from the input data stream. The output rate is the same as the input rate. This block is commonly used in conjunction with a convolutional encoder ([Synplify DSP Convolutional Encoder, on page 8-56](#)) to implement punctured convolutional codes.

This block is a custom block. (See [Primitives and Custom Blocks, on page 5-2](#) for a definition.) The following figure shows how the block is modeled:



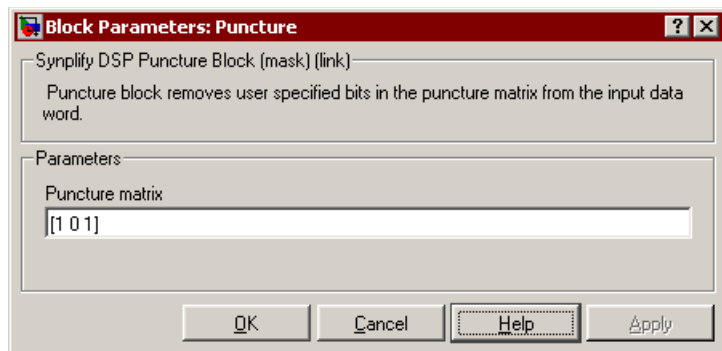
## Icon Annotations

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.

# Synplify DSP RAM

Stores signals in an array with configurable read and write access ports.

## Library

Synplify DSP [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 [RAM Background Description, on page 8-186](#).

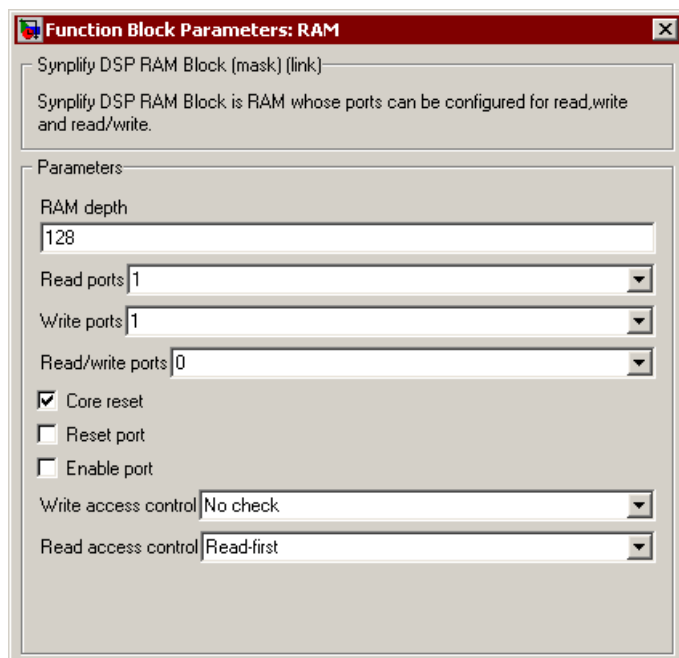
The Synplify DSP tool creates RAM memories that are fully synchronous, with write-first access mode. For information about data format, see [RAM Data Format, on page 8-185](#).

## 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 Synplify DSP RAM Block and a list of parameters. The parameters include RAM depth (128), Read ports (1), Write ports (1), Read/write ports (0), Core reset (checked), Reset port (unchecked), Enable port (unchecked), Write access control (No check), and Read access control (Read-first).

**Function Block Parameters: RAM**

Synplify DSP RAM Block (mask) (link)

Synplify DSP RAM Block is RAM whose ports can be configured for read/write and read/write.

Parameters

RAM depth: 128

Read ports: 1

Write ports: 1

Read/write ports: 0

☒ Core reset

☐ Reset port

☐ Enable port

Write access control: No check

Read access control: Read-first

### 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 8-190](#).

### Write ports

Sets the number of read ports. For additional information about the settings, see [Port Use in Different RAM Configurations, on page 8-190](#).

### Read/write ports

Sets the number of read ports. For additional information about the settings, see [Port Use in Different RAM Configurations, on page 8-190](#).

### 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. When disabled, the RAM output value and contents do not change.

### 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. When disabled, 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

- No check
- Write prioritization

See [Write Access Control, on page 8-189](#) for details.

### Read access control

Determines the kind of logic for the read signal. You can set it to

- 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 8-189](#) for details.

### 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 is the data type of the inputs.

## Diagnostics

Warning: block 'experiment/RAM': Contents of address 93 unknown!

## RAM Background Description

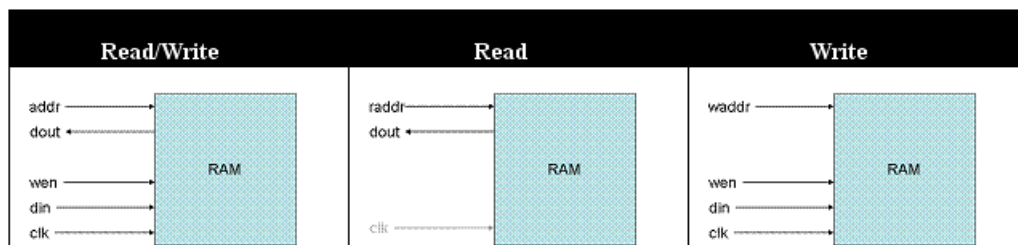
This section contains background information about the terms used and the use of ports in different RAM configurations.

### RAM Term 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:

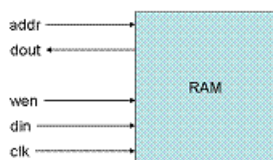
---

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.

---

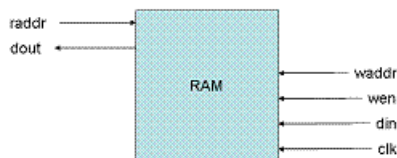
## Single Port RAM

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.



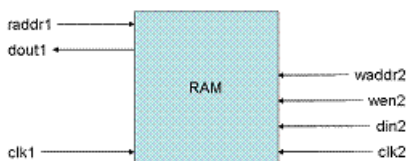
## Two Port RAM

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

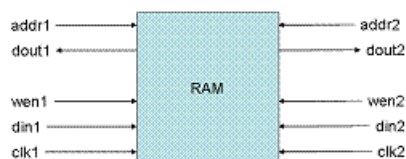
Read and write access to the RAM go through two ports on the RAM, one dedicated to read access and one dedicated to write access; the clocks for read and write are independent.



### True Dual Port RAM

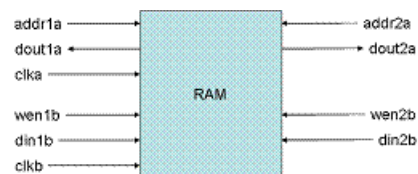
Read and write access to the RAM go through two ports on the RAM, where both ports can be used in either read or write mode; the clocks can be independent. This allows for the following permutations:

- 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.



### Access Control

There are potential clashes:

- Write operations to the same location  
You control this through the write access control mode, as described in [Write Access Control, on page 8-189](#).



- Read and write operation to same location  
You control this through the read access control mode, as described in [Read Access Control, on page 8-189](#).

## 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 clocks, the behavior is undefined.

## Read Access Control

If a read and a write to the RAM core happen to the same location, there are different strategies. The following table is for when read and write to the RAM happen to the same location:

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 read/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 write to the same location. If more than one writes are to be performed write-prioritization is employed for selection of 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 with a read port.

## 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

# Synplify DSP Ramp

Creates a ramp, based on increments derived from a port or a parameter.

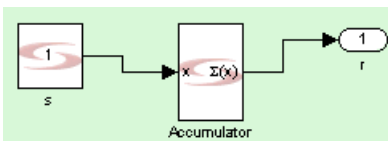
## Library

Synplify DSP [Sources](#)

## Description



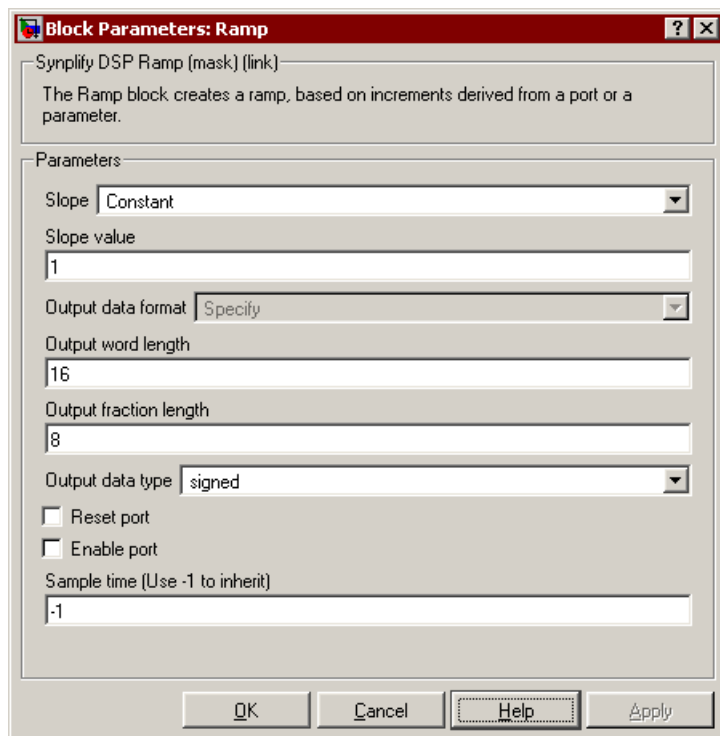
This custom block (see [Primitives and Custom Blocks, on page 5-2](#) for a definition) generates a ramp signal using an accumulator with input increment that is determined by the value of the Slope option. The accumulator value wraps when overflow occurs.



## Latency

This block has no latency.

## Ramp Parameters



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

- A title bar with a red background and a close button.
- A description: "Synplify DSP Ramp (mask) (link)" and "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 box containing "1".
  - "Output data 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".
  - Two unchecked checkboxes: "Reset port" and "Enable port".
  - "Sample time (Use -1 to inherit)" text box containing "-1".
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

### 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	<a href="#">Output Word Length, on page 8-288</a>
Fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Data type	<a href="#">Output Data Type, on page 8-288</a>

### 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.

# Synplify DSP Random

Creates a random integer of the requested word length.

## Library

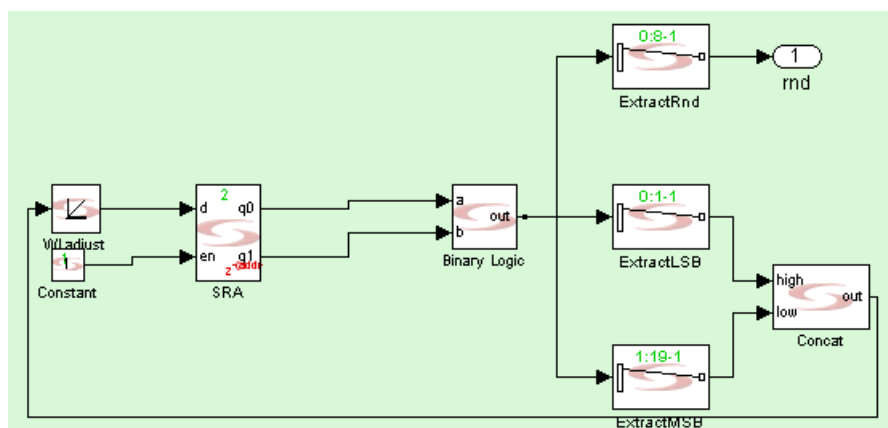
Synplify DSP [Sources](#)

## Description



This custom block (see [Primitives and Custom Blocks, on page 5-2](#) for a definition) creates a random integer of the specified word length.

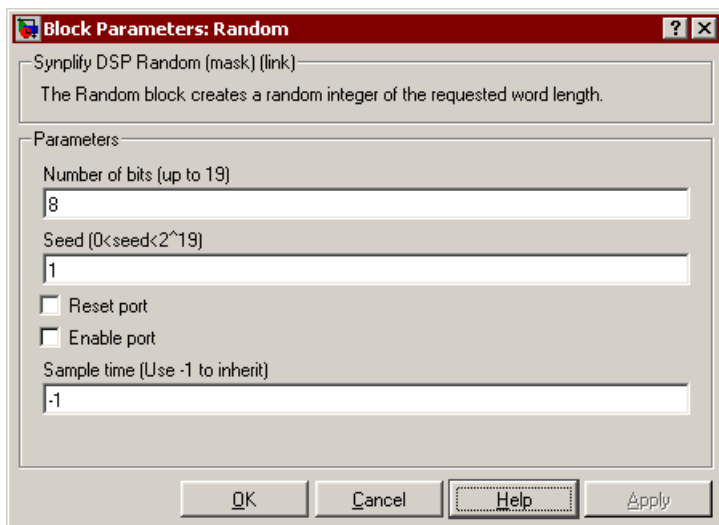
The following figure shows the internal construction of this block, without reset or enable ports:



## 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.

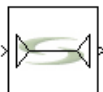
# Synplify DSP Recast

Generates an output value, based on the requested data type you cast at the output.

## Library

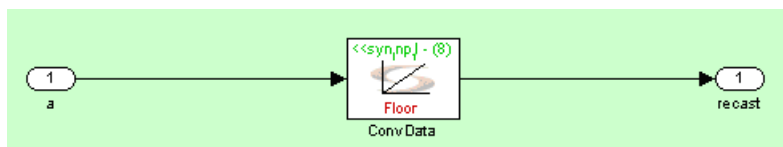
Synplify DSP [Signal Operations](#)

## Description



The Synplify DSP Recast block is a custom block (see [Primitives and Custom Blocks](#), on page 5-2 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:

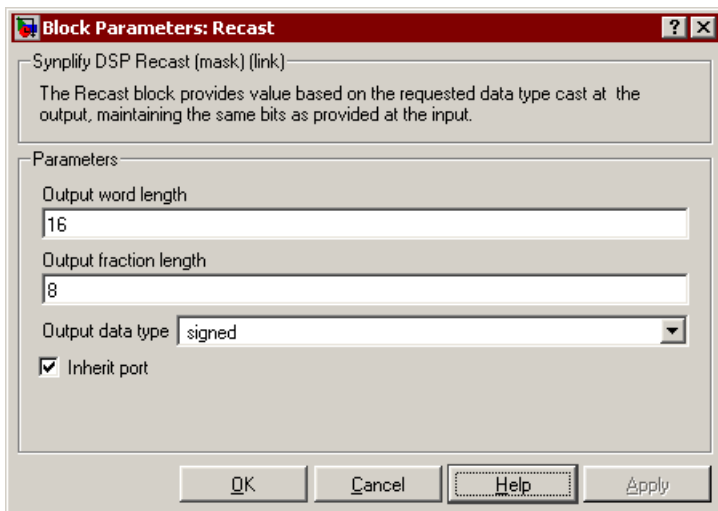


## Latency

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 8-199](#).

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 8-199](#).

## 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 8-198](#) 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 8-199](#) for information about these variables.
- Use the inherit option to specify the Output data type. See [Data Type, on page 8-198](#) 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.

# Synplify DSP Reed-Solomon Decoder

Decodes the encoded signal using Reed-Solomon error-correcting codes.

## Library

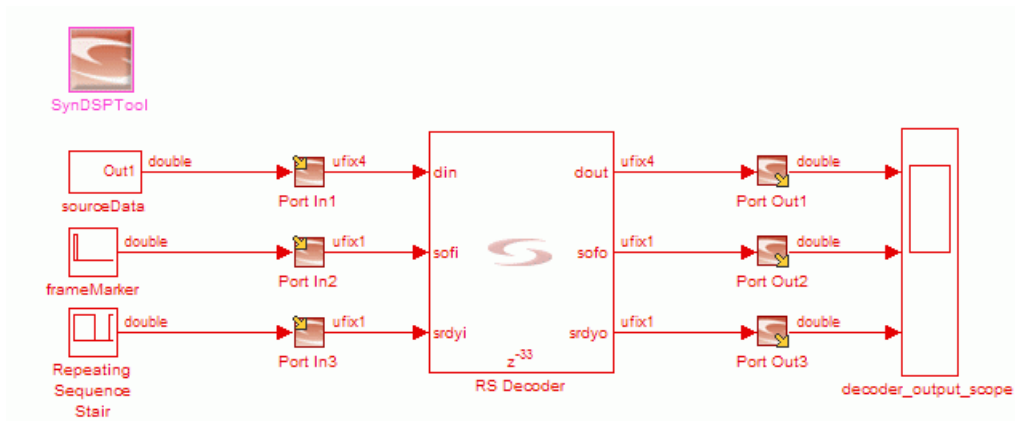
Synplify DSP [Communications](#)

## Description



The Synplify DSP Reed-Solomon Decoder decodes the data block encoded by the Synplify DSP Reed-Solomon Encoder (see [Synplify DSP Reed-Solomon Encoder, on page 8-207](#)). It processes each block and attempts to correct errors and recover the original data. See [Reed-Solomon Coding and Decoding, on page 8-208](#) for more information about Reed-Solomon encoding and decoding.

The following is an example of the Reed-Solomon Decoder block:



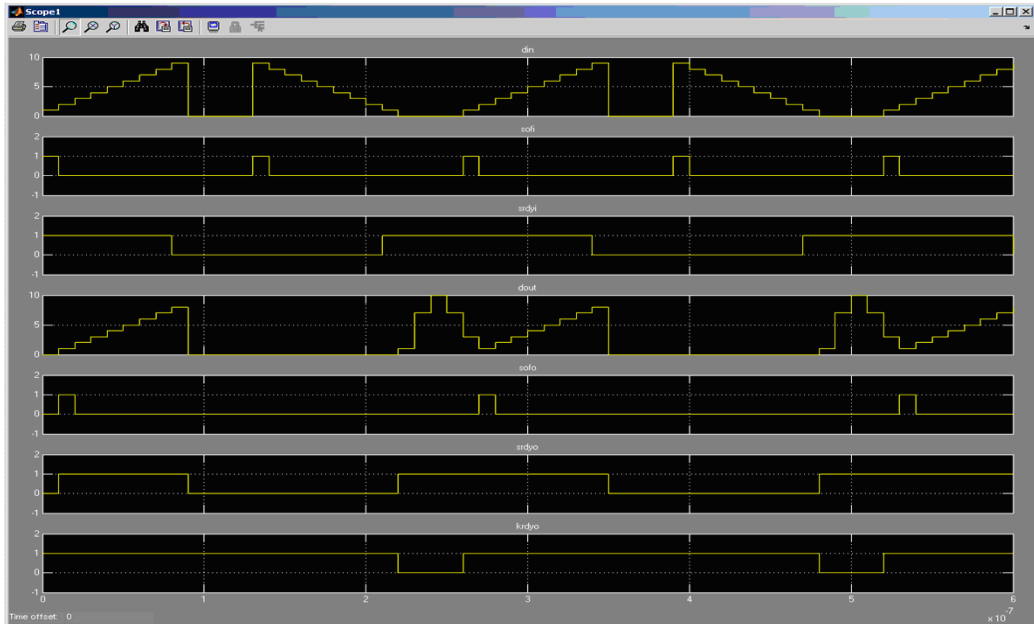
## Input and Output Pins

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 <a href="#">Statistics Ready/End of Frame Output, on page 8-205</a> for a description.
eri Erasure put	This is an optional pin. See <a href="#">Erasure Input, on page 8-205</a> for a description.
symerr Symbol error output	This is an optional pin. See <a href="#">Symbol Error Output, on page 8-205</a> for a description.
ber Bit error count output	This is an optional pin. See <a href="#">Bit Error Count Output, on page 8-205</a> for a description.
ser Symbol error count output	This is an optional pin. See <a href="#">Symbol Error Count Output, on page 8-206</a> for a description.
decfail Decoder failure output	This is an optional pin. See <a href="#">Decoder Failure Output, on page 8-206</a> for a description.

## Timing Diagram

The following figure shows the Reed-Solomon Decoder block timing:



## Latency

The latency for this block is calculated as follows:

Codeword length (N) \* 2 - Message length(K) + Bitwidth (m) + 8.

## Reed-Solomon Decoder Parameters

**Synplify DSP Reed-Solomon Decoder**

Synplify DSP RS Decoder  
This block decodes the RS encoded signal

**Parameters**

Bitwidth m

Codeword length N

Message length K

☐ Specify primitive polynomial

☐ Specify generator polynomial

Specify parameters for the generator polynomial

Generator polynomial coefficients

Starting Power

Root spacing

**Optional Input/Output Selections**

☐ Erasure Input

☐ Symbol Error Output

☐ Statistics Ready / End of Frame Output

☐ Bit Error Count Output

☐ Symbol Error Count Output

☐ Decoder Failure Output

OK Cancel Help Apply

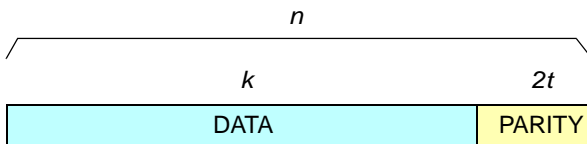
### Bit width M

Specifies the bits per symbol for the message symbols and the parity symbols. The Synplify DSP 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:

$$\text{Codeword length } N - \text{Message Length } K = \text{Parity}$$



See [Reed-Solomon Coding and Decoding](#), on page 8-208 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:

$$\text{Codeword length } N - \text{Message Length } K = \text{Parity}$$

### Specify primitive polynomial

Specifies the primitive polynomial for the Galois field. You can specify it in either decimal or vector form:

$$D^4 + D^3 + 1 \Rightarrow [1 \ 1 \ 0 \ 0 \ 0] \text{ or } 25$$

### 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  
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 (eoffo) 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.

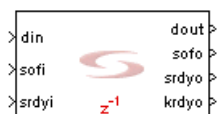
# Synplify DSP Reed-Solomon Encoder

Generates an encoded signal, using Reed-Solomon error-correction codes.

## Library

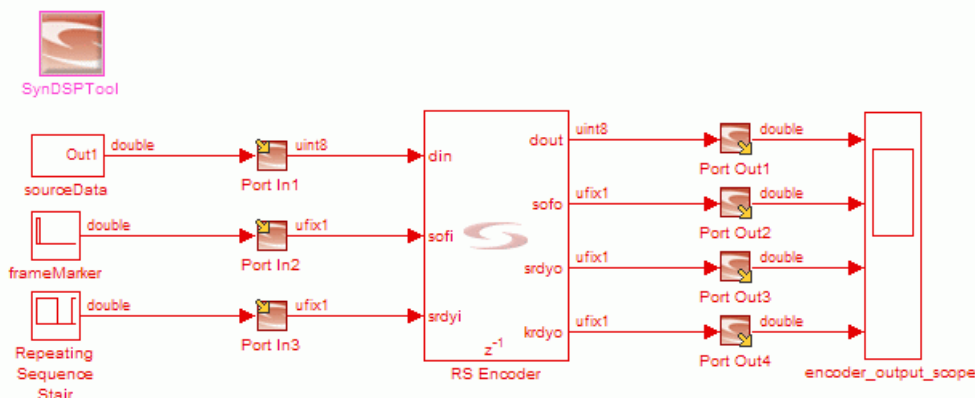
Synplify DSP [Communications](#)

## 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 8-208 for more information about Reed-Solomon encoding.

The following is an example of the Reed-Solomon Encoder block:

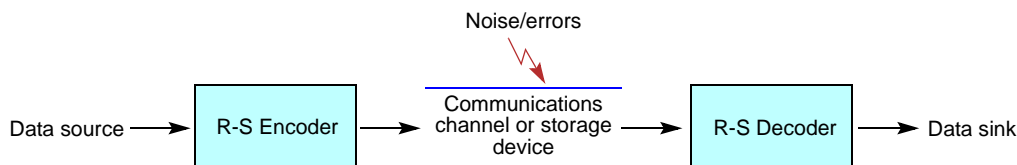


The number of errors that the Reed-Solomon Encoder block can correct is determined by the definition of the encoder. This applies to shortened codes too. For example, the shortened code for RS(255,223) is RS(200,168), but the number of parity symbols is still 32. Thus, the number of correctable errors 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 Synplify DSP Reed-Solomon blocks.

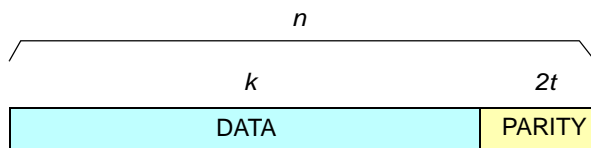
## Reed-Solomon Coding and Decoding

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:



## 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 Synplify DSP 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:

$$D^4 + D^3 + 1 \Rightarrow [1 \ 1 \ 0 \ 0 \ 0] \text{ or } 25$$

**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:

$$(X - \alpha^{fr}) * (X - \alpha^{(fr+sp)}) * \dots * (X - \alpha^{(fr+(N-K-1)*sp)})$$

**Generator polynomial coefficients**

Defines the coefficients for the generator polynomial as vectors.

**Starting power**

Defines the first root when you specify the 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.

# Synplify DSP Register

Inserts a delay, with optional reset and enable ports.

## Library

Synplify DSP [Memories](#)

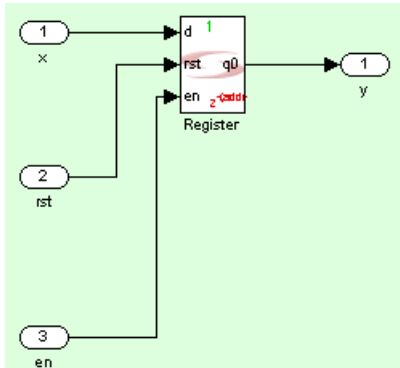
## Description



The Synplify DSP Register block is a custom block (see [Primitives and Custom Blocks](#), on page 5-2 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 ([Synplify DSP Delay](#), on page 8-91) instead of the Register block whenever possible, because some retiming optimizations cannot be implemented with the Register block.

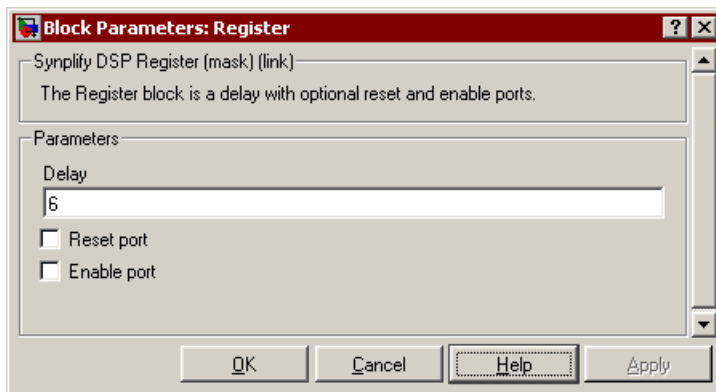
The following figure shows the internal construction of the Register block with optional reset and enable ports:



## 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.

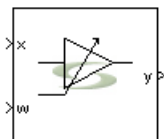
# Synplify DSP RFIR

Implements a reloadable finite impulse response (FIR) filter with a coefficient load register.

## Library

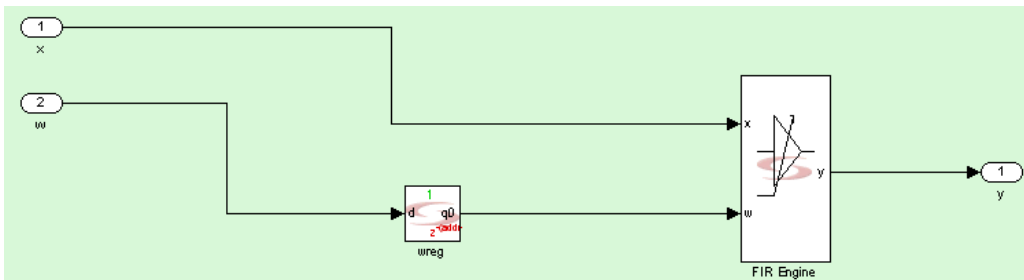
Synplify DSP [Filtering](#)

## Description



The Synplify DSP RFIR block is a custom block (see [Primitives and Custom Blocks, on page 5-2](#) 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.

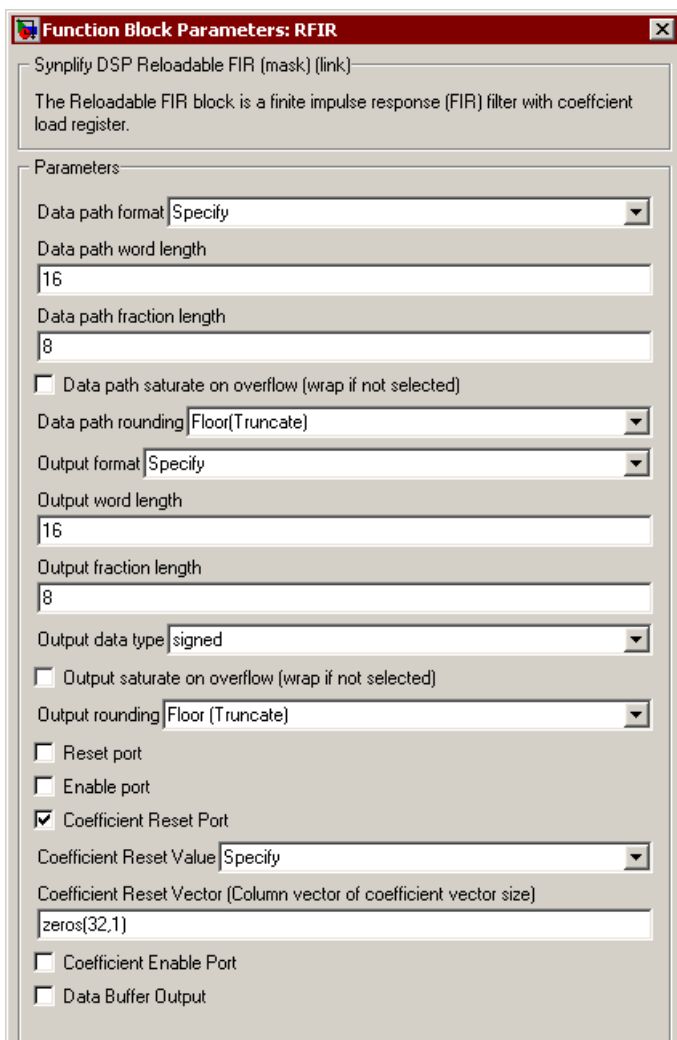


## Latency

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



**Function Block Parameters: RFIR**

Synplify DSP 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 Reset Value: Specify

Coefficient Reset Vector (Column vector of coefficient vector size): zeros(32,1)

☐ Coefficient Enable Port

☐ Data Buffer Output

### 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 8-292](#).

### 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 8-292](#).

### Data path saturate on overflow

Determines how the data path overflow value is handled. See [Overflow Saturation Options, on page 8-289](#) for details.

### Data path rounding

Determines how underflow in the data path is rounded. See [Underflow Rounding Options, on page 8-289](#) 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> 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 <a href="#">Special Variables, on page 8-292</a> .
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## 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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output rounding	Specifies the algorithm to use to round the output underflow. See <a href="#">Underflow Rounding Options, on page 8-289</a> 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 8-122](#) 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.

# Synplify DSP ROM

Models a read-only memory (ROM) with a latency of one sample.

## Library

Synplify DSP [Memories](#)

## Description

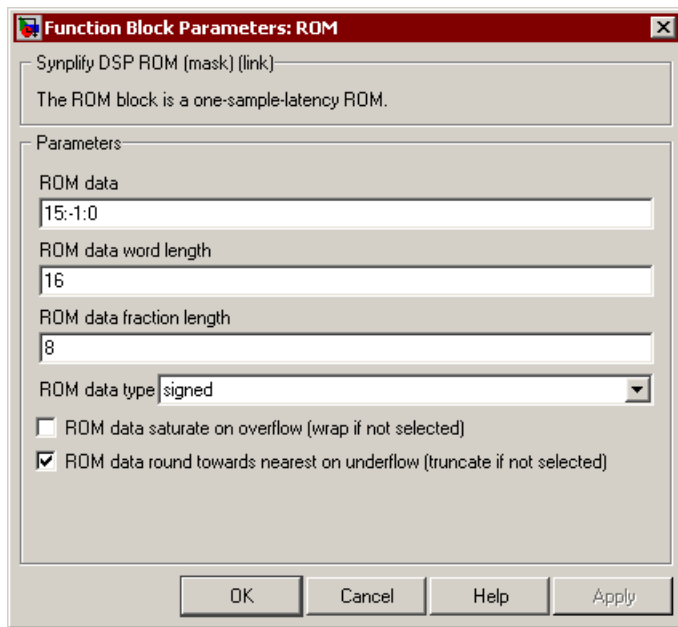


The Synplify DSP ROM block models a ROM with a latency of one sample. If the addr input value exceeds the size of the ROM, the output is 0.

## Latency

This block has a latency of one sample.

## ROM Parameters



### ROM data

Sets the depth of the ROM and the valid input values in one of the following 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 9-11 for the syntax and [Specifying ROM Data with syn\\_read\\_hex](#), on page 4-43 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	<a href="#">Output Word Length, on page 8-288</a>
ROM data fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
ROM data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> .
ROM data round towards nearest on underflow	Uses the Nearest or Floor (Truncate) algorithms to round the underflow; see <a href="#">Underflow Rounding Options, on page 8-289</a> .

# Synplify DSP Sequence

Repeats a sequence of specified data.

## Library

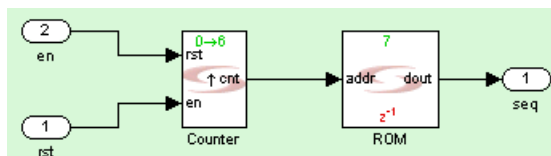
Synplify DSP [Sources](#)

## Description



This custom block (see [Primitives and Custom Blocks, on page 5-2](#) 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

**Block Parameters: Sequence**

Synplify DSP Sequence (mask) (link)

The Sequence block repeats a sequence of specified data.

**Parameters**

Sequence  
[0 1 0 2 0 1 0]

Output word length  
16

Output fraction length  
8

Output data type  
signed

☐ Reset port

☐ Enable port

Sample time (Use -1 to inherit)  
-1

OK Cancel Help Apply

### 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	<a href="#">Output Word Length, on page 8-288</a>
Fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Data type	<a href="#">Output Data Type, on page 8-288</a>

### 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.

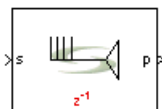
## Synplify DSP Serial to Parallel

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

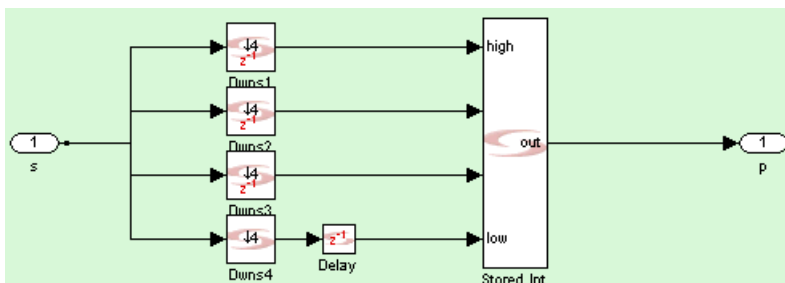
Synplify DSP [Signal Operations](#)

### 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 5-2 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

**Block Parameters: Serial to Parallel**

Synplify DSP 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

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	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

**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 <a href="#">Overflow Saturation Options, on page 8-289</a> for details.
Output round on underflow	Specifies which algorithm is used to round the output underflow. See <a href="#">Underflow Rounding Options, on page 8-289</a> for details of the algorithms.

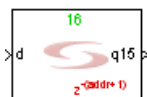
# Synplify DSP Shift Register

Implements a delay line with dynamic or static access to intermediate taps.

## Library

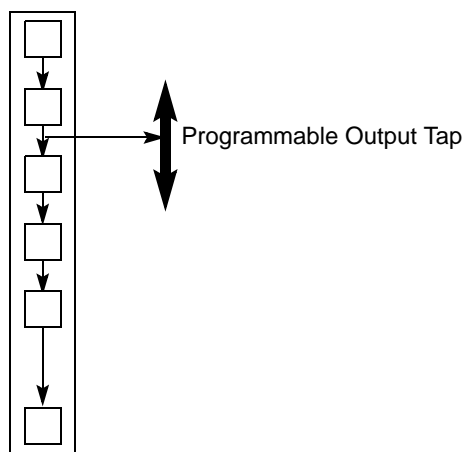
Synplify DSP [Memories](#)

## Description



The Synplify DSP 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.



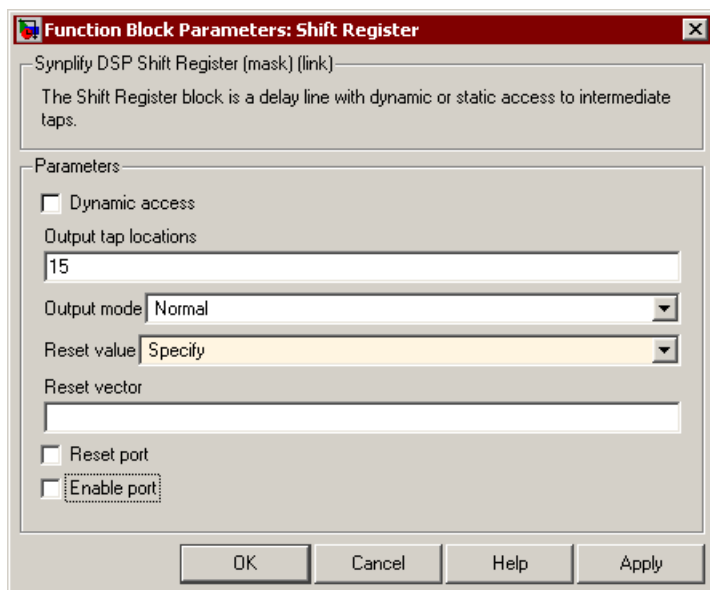
Outputs inherit the input data type. Potential inputs must have the same data type.

## Icon Annotations

The icon for this block displays the following information:

Top Annotation	The annotation at the top of the block indicates the division algorithm being used for the operation. T stands for truncate.
Latency Annotation	The latency of the block is determined by its functionality of the component, because it depends on the intended output and the static or dynamic tap location. The block icon is annotated with the maximum latency.

## 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  
The block output becomes a vector. The first tap of the shift register becomes the first element of the vector and the last tap becomes the last element.
- Merge to vector in reverse order  
The block output becomes a vector. The last tap of the shift register becomes the first element of the vector and the first tap becomes the last element.

**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 of 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.

## Examples

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

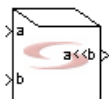
# Synplify DSP Shifter

Performs a variable left or right shift on the input signal.

## Library

Synplify DSP [Math Functions](#)

## Description



The Synplify DSP 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 [Synplify DSP Convert](#) block.

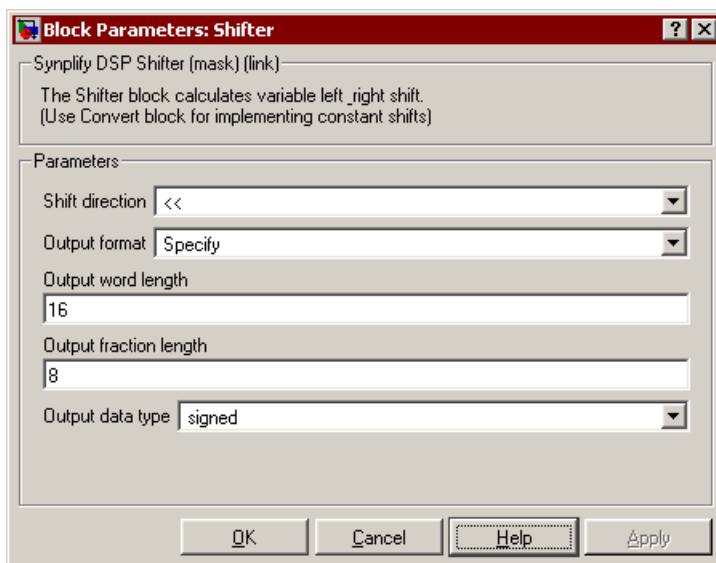
The type of architecture is automatically derived from the input type. For the most optimized solution or if you do not expect negative values, use unsigned data for the input.

- 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.

## 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	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

# Synplify DSP Sign

Provides the 2-bit sign value (=1 or -1) for the input.

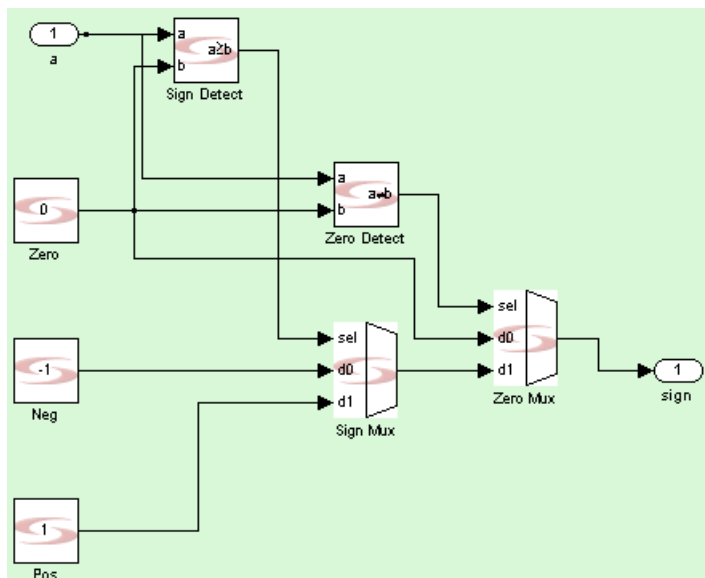
## Library

Synplify DSP [Math Functions](#)

## Description



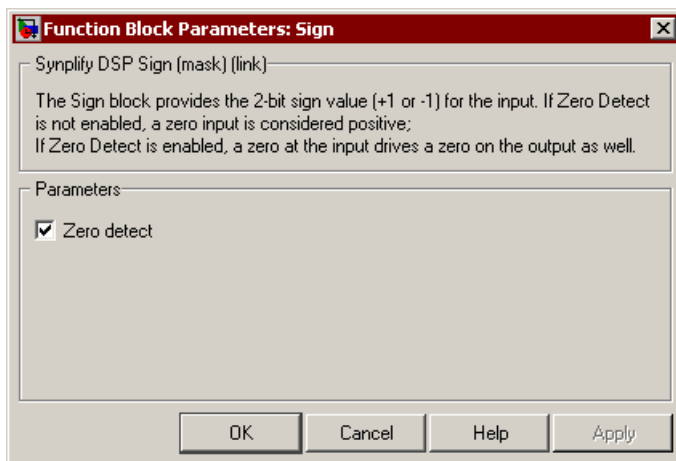
This custom block (see [Primitives and Custom Blocks, on page 5-2](#) for a definition) provides the 2-bit sign value (=1 or -1) for the input.



## 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.

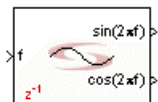
# Synplify DSP SinCos

Calculates  $A \cdot \sin(2\pi f)$  or  $A \cdot \cos(2\pi f)$  for the input  $f$ , where  $A$  is the amplitude parameter.

## Library

Synplify DSP [Math Functions](#)

## Description



The Synplify DSP SinCos block calculates  $A \cdot \sin(2\pi f)$  and/or  $A \cdot \cos(2\pi f)$  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 only considers the fraction portion of the input. Given the scaling with  $2\pi$  on the input, any integer portion corresponds with a full revolution of the trigonometric function, and therefore can be ignored to calculate 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

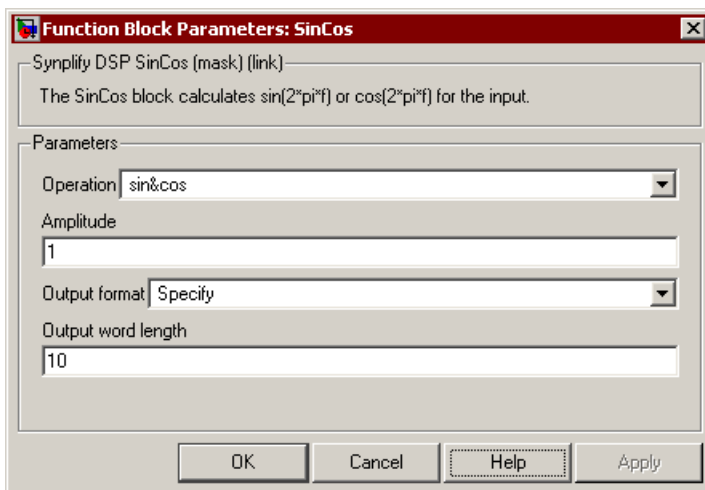
Given the restriction on the input, the output accuracy is limited to 53 bits. Any request for a larger fraction on the output results in the same accuracy of 53 bits, zero-extended at the LSB side.



## Latency

The latency of the SinCos block is 1.

## SinCos Parameters



### Function

Selects the operation to be performed:

- sin calculates  $A \cdot \sin(2\pi \cdot f)$  for the input.
- cos calculates  $A \cdot \cos(2\pi \cdot f)$  for the input.
- sin&cos calculates  $A \cdot \sin(2\pi \cdot f) \& A \cdot \cos(2\pi \cdot f)$  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.

## Synplify DSP Smart Black Box

Lets you embed third-party IP in a Synplify DSP design and automatically cosimulate it.

### Library

Synplify DSP [Ports & Subsystems](#)

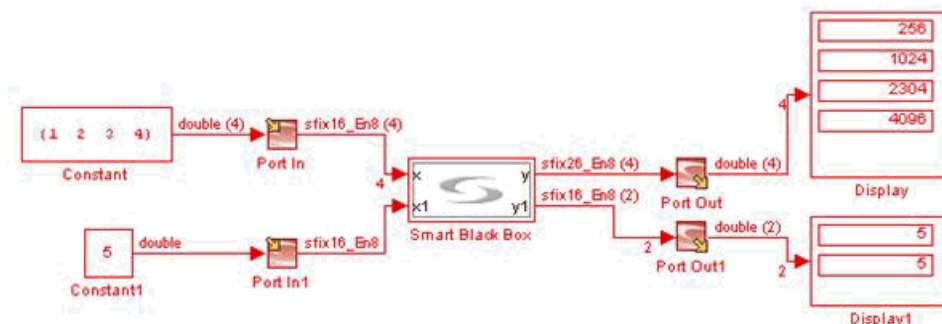
### Description



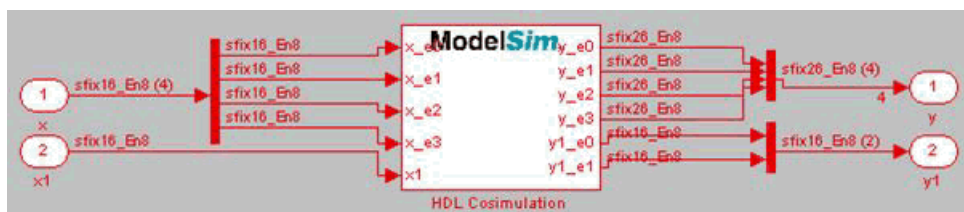
The Synplify DSP Smart Black Box lets you embed third-party blocks for which you have access to the RTL code. If you have IP with no access to the RTL code, use the Black Box block ([Synplify DSP Black Box, on page 8-25](#)) instead.

The Smart Black Box requires that you have a license for Link for ModelSim®, which is a cosimulation interface between Simulink and the ModelSim® HDL simulator. The cosimulation interface must be configured using the SynCo-SimTool block ([Synplify DSP SynCoSimTool, on page 8-249](#)). Synplify DSP uses Link for ModelSim to verify and simulate the embedded RTL-level models. The Simulink simulation is transparent, but the RTL generated by Synplify DSP treats the IP as a black box. See [Using Smart RTL Black Boxes, on page 4-33](#) 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 8-241](#)). The following figure illustrates how vector input and output are handled in a Smart Black Box block.



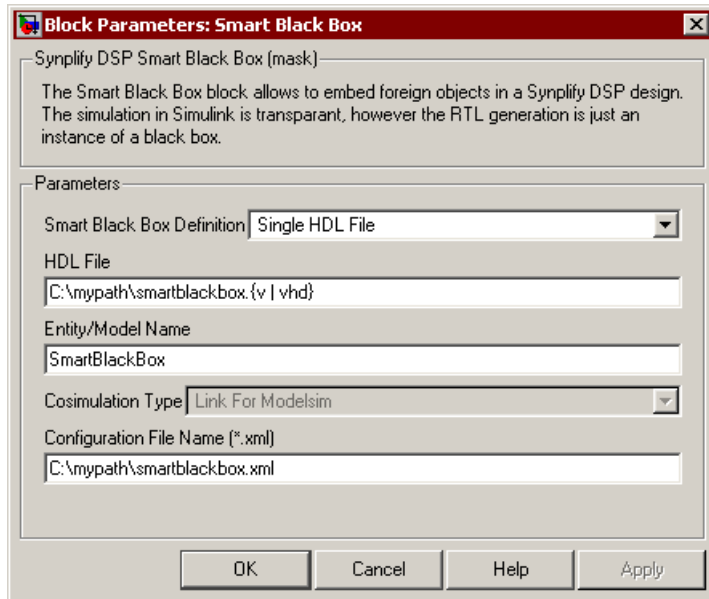
When the input to the Smart Black Box is a vector, the tool demultiplexes the vector input inside, and then transfers it to the ModelSim HDL Cosimulation block. It then multiplexes the output of ModelSim HDL Cosimulation block, and the output of the Smart Black Box again becomes a vector.



## 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 (`sbb1list.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 Link for Modelsim.

## 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 4-37](#) for information about creating this file, and [Configuration File For Smart Black Box, on page 8-241](#) 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 8-242](#)
- [Clocks, on page 8-243](#)
- [Global Enables, on page 8-243](#)

- [Global Resets, on page 8-243](#)
- [Sample Configuration File, on page 8-244](#)

## 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>
```



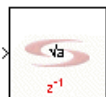
# Synplify DSP Sqrt

Calculates the square root of the input.

## Library

Synplify DSP [Math Functions](#)

## Description



The Synplify DSP 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](#), below, for more information.

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 example, if the input word length is 9 and the number of input fraction bits is 3, then the software sets the output word length to 5 and the number of output fraction bits to 2. For signed input, the sign bit is discarded and the input is treated as unsigned. The following table illustrates:

	Input Parameters	Sqrt Block Output
<b>Unsigned input</b>	Input word length = 9 Input fraction length = 3	Output word length = $(9+1)/2+1 = 6$ Output fraction length = $(3+1)/2 = 2$
<b>Signed input</b>	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 Synplify DSP software uses normalization to improve precision. The Synplify DSP 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 Synplify DSP 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.

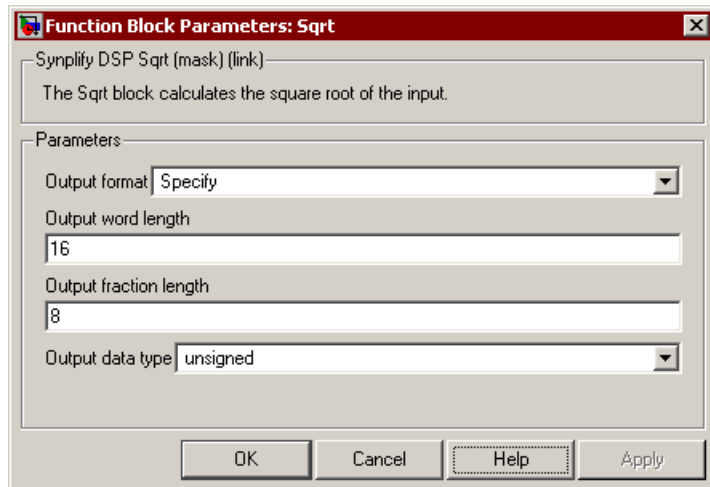
To take a specific example, an input  $x$  is first converted into the form  $x = 2^{(2N)} \cdot x_n$ . The normalized  $x$  is  $256 \leq x_n < 1024$ . Then the square root is calculated as follows:  $\text{sqrt}(x) = 2^N \cdot \text{sqrt\_table}(\text{int}(x_n) - 256)$ .

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 ([Synplify DSP CORDIC Sqrt, on page 8-76](#)).

## Latency

This block has a latency of 1.

## Sqrt Parameters

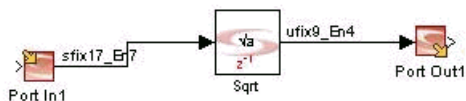
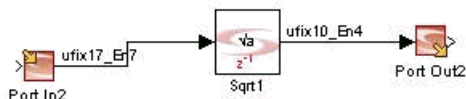


## Output format, Output word length, Output fraction length, and Output data Type

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a> 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	<a href="#">Output Word Length, on page 8-288</a> Output word length = input word length / 2 + 1
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a> If the fraction length of the input is odd, the tool adds another 0-valued fraction bit to the input. Output fraction length = input fraction length / 2
Output data type	<a href="#">Output Data Type, on page 8-288</a>

## Sqrt Block Examples



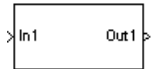
# Synplify DSP Subsystem

Allows you to add a subsystem to a Synplify DSP design.

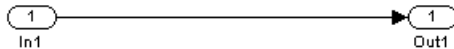
## Library

Synplify DSP [Ports & Subsystems](#)

## Description



The Synplify DSP 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.

## Latency

The latency of this block is determined by its contents.

# Synplify DSP SynCoSimTool

Manages communication between smart black boxes and the RTL cosimulation interface.

## Library

Synplify DSP, top level library

## Description



The SynCoSimTool block controls the interaction between a smart black box (see [Synplify DSP Smart Black Box, on page 8-238](#)) 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 RTL Black Boxes, on page 4-33](#).

## SynCoSimTool Parameters

The screenshot shows the 'RTL Cosimulator Control Tool' dialog box. It has a title bar with a red background and a yellow icon. The dialog is divided into several sections: 'Run & Close RTL Cosimulator' with a checked checkbox and a 'Create Template Configuration File(s)' button; 'Timescales' with a text field '1 second in Simulink corresponds to' followed by a text box containing '1', a dropdown menu showing 's', and the text 'in the RTL Cosimulator'; 'Connection' with a checked checkbox 'RTL Cosimulator running on this computer', a 'Connection Method' dropdown menu showing 'Socket', and a 'Ports' text box containing '4449,4450'; and 'Tcl' with two text areas: 'Presimulation Commands' containing 'echo 'Simulation Start';' and 'Postsimulation Commands' containing 'echo 'Simulation Stop';'. At the bottom are 'Ok', 'Cancel', and 'Help' buttons.

**RTL Cosimulator Control Tool**

☒ Run & Close RTL Cosimulator      Create Template Configuration File(s)

**Timescales**

1 second in Simulink corresponds to   in the RTL Cosimulator

**Connection**

☒ RTL Cosimulator running on this computer

Connection Method:

Ports:

**Tcl**

**Presimulation Commands**

**Postsimulation Commands**

Ok      Cancel      Help

### 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 4-35](#).

### **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 4-37](#) and [Configuration File For Smart Black Box, on page 8-241](#).



# Synplify DSP SynDSPTool

Opens the SynDSPTool interface where you can set system-level optimizations and set parameters for generating RTL code.

## Library

Synplify DSP, top level library

## Description

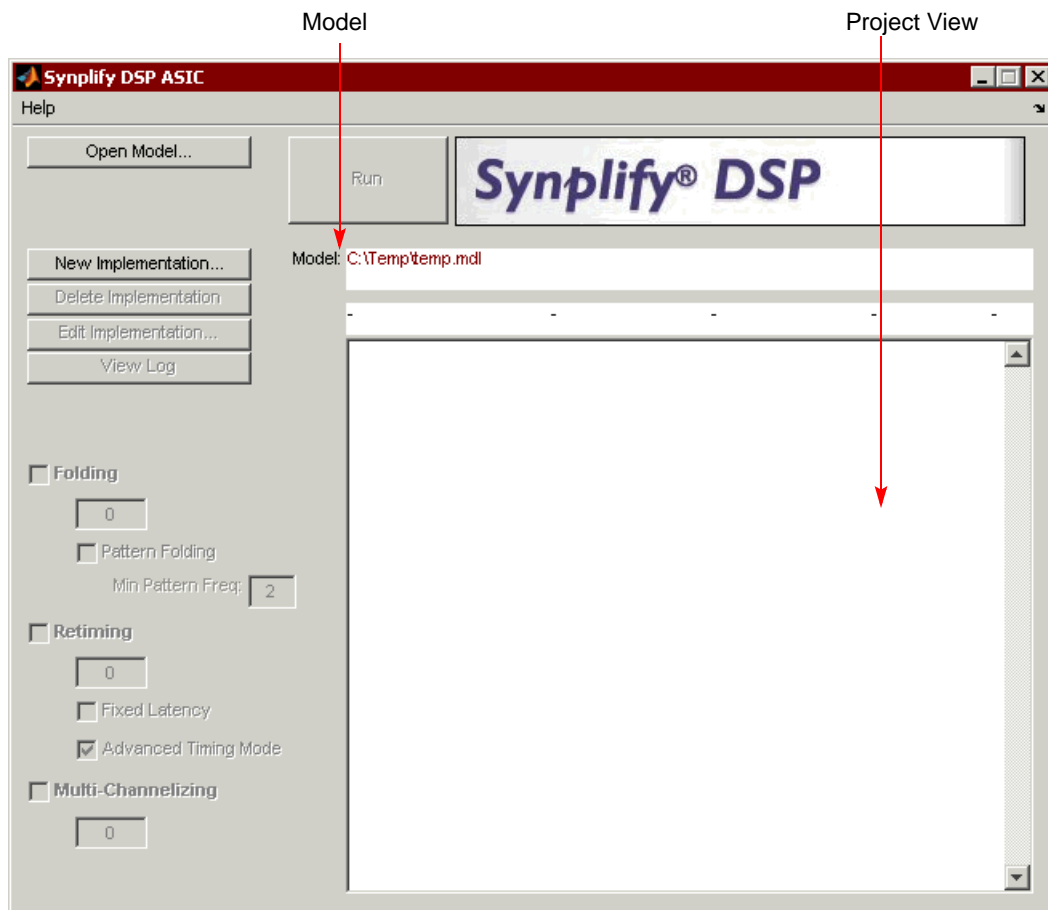


The SynDSPTool toolbox provides an interface where you can specify system-level optimizations and set parameters for generating RTL code. Users of Synplicity synthesis products will find the interface intuitive, as it has the same look and feel as the Synplicity synthesis products. See [Running DSP Synthesis with SynDSPTool, on page 4-48](#) for information about using this toolbox.

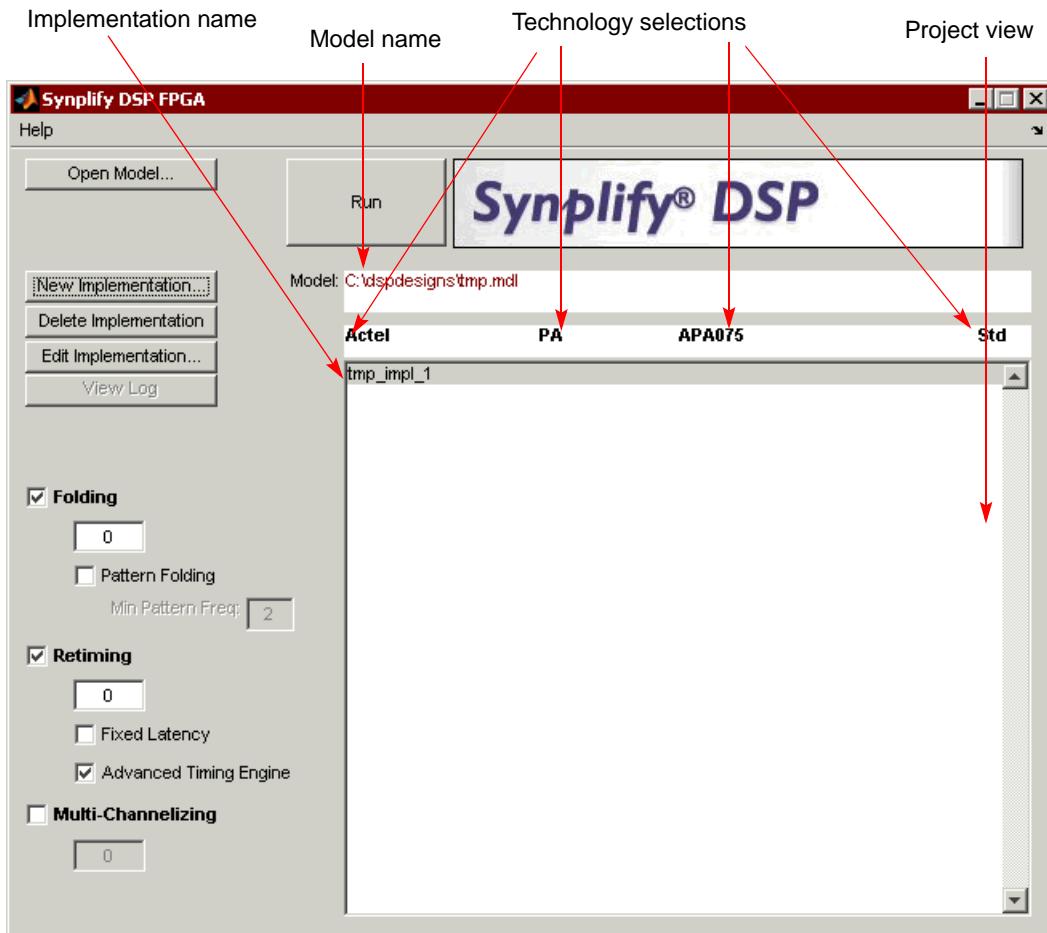
You must have a SynDSPTool block in every Simulink® model that contains any element from the Synplify DSP blockset.

## SynDSPTool Toolbox Interface

The following figure shows the Synplify DSP 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 8-258](#) 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 8-258](#) for details. When you select a technology in this box, the results are reflected in the toolbox:

**View Log**

Opens the log file.

**Folding**

Performs time-multiplexed resource sharing during area/speed tradeoffs 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 4-55](#).

**Pattern Folding**

When enabled, runs the pattern folding optimization on the design to identify recurring patterns and share resources. See [Pattern Folding, on page 4-57](#) 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 compu-

tational complexity of pattern identification by judiciously selecting a value that allows larger patterns to be identified.

## Retiming

Enables retiming. Retiming rearranges registers so as to optimize speed, while preserving functionality. When this option is enabled, a box opens where you can set the number of registers (latency cycles) available for retiming. If you specify very fast sample rates, retiming might insert extra latency to meet the timing requirements.

The default value of 0 retimes the design by moving existing registers; a positive value determines the number of available registers. A positive value sets the number of registers available for retiming. For details about using this option, see [Optimizing with Retiming, on page 4-53](#).

Enabling retiming also enables the Advanced Timing Mode option.

## Advanced Timing Mode

Determines which timing engine is used:

- Enable this option to use the advanced timing mode for timing estimates. 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.
- Disable this option to use estimation mode for timing estimates. In this mode, the tool uses simpler, latency-based device characterizations as a basis for optimizations. Estimation mode is faster, but the results are less accurate.

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.

## 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 registers 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 4-60](#).

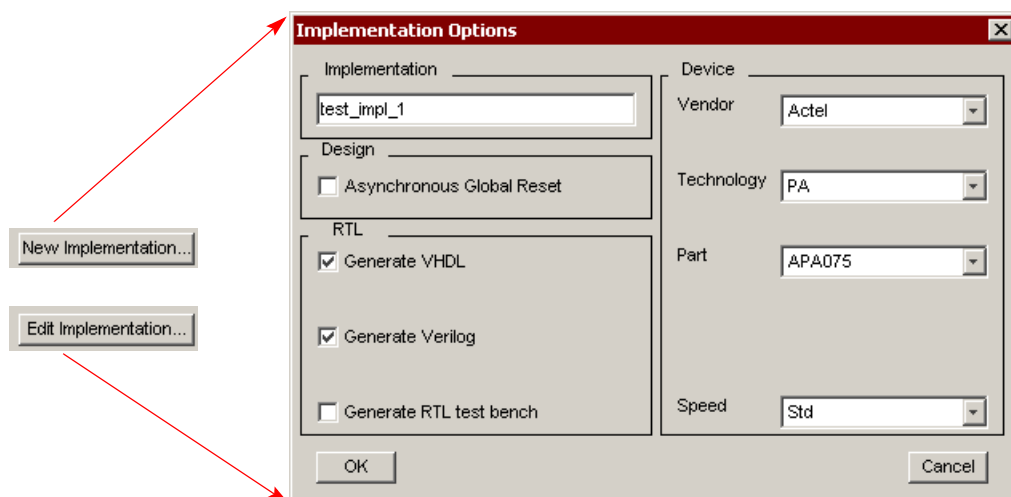
## Run

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

This section describes the options in the Implementation Options dialog box, which opens when you click the New Impl or Impl Options buttons in the Synplify DSP toolbox. For information about using these options, see [Setting up Implementations, on page 4-48](#).



## Implementation

Lets you name or rename the implementation.

## Asynchronous Global Reset

When enabled, sets asynchronous global resets. When disabled, the design does not have asynchronous global resets. For details of how the Synplify DSP tool implements resets in the design, see [Resets in Synplify DSP, on page 3-22](#).

## Generate VHDL

When enabled, this option generates a VHDL design that you can use as input for synthesis with Synplify Pro.

## Generate Verilog

When enabled, this option generates a Verilog design that you can use as input for synthesis with Synplify Pro.

## Generate RTL Test Bench

When enabled, this option creates a VHDL test bench for pre-synthesis functional verification with a VHDL simulator. You use the test bench along with the VHDL file(s) 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,

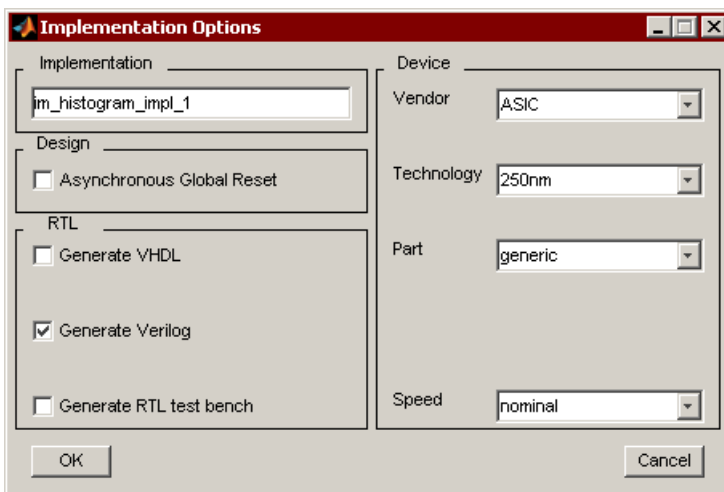
reads out output, and compares it with the output test vectors. The software handles the extra latency introduced by retiming by treating it as one of the inputs when it generates the test bench.

If you are using Verilog design files and require a test bench, you must select the Generate VHDL option in addition to this option. This is because the tool does not generate a Verilog test bench. You can generate a VHDL test bench and use it in a mixed language simulator.

For information about using this option, see [Verifying the RTL with a Test Bench, on page 4-63](#).

## Vendor

Selects a target vendor, and determines the choices available for Technology, Part, Package, and Speed. For ASIC designs, set this to ASIC. This ensures that the information generated after DSP synthesis is in a format that can be read by the ASIC back-end tools. For more information on ASIC designs, see [Synplify DSP ASIC Design Flow, on page 1-11](#) and [Working with the Output for ASIC Designs, on page 4-7](#).



## 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 Synplify Pro to port the design to other devices that are not listed here.



- For ASIC devices, this selects the process you are targeting. For more information on working with ASIC designs, see [Synplify DSP ASIC Design Flow, on page 1-11](#) and [Working with the Output for ASIC Designs, on page 4-7](#).

**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. There is only one setting for ASIC devices.

**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. There is only one setting for ASIC devices.

## Synplify DSP SynFixPtTool

Opens the Simulink Fixed-Point interface.

**Library**

Synplify DSP, top level library

**Description**

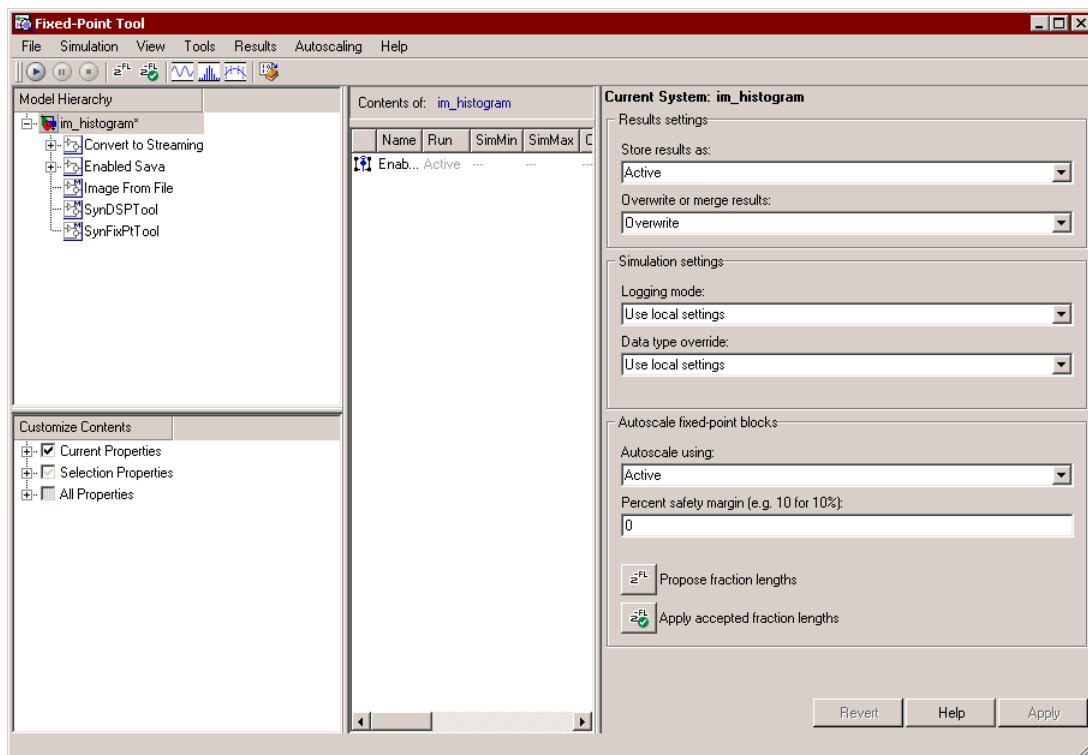
The Synplify DSP SynFixPtTool toolbox opens the Simulink Fixed-Point Settings interface where you can conveniently access global data type overrides and logging settings (MATLAB 2006B only), the logged data, the automatic scaling script, and the Plot System interface. The Fixed-Point Settings GUI is an optional Simulink package, and the Synplify DSP SynFixPtTool toolbox will not function properly unless it is installed.

---

**Note:** The appearance of the Simulink Fixed-Point Settings interface varies with the MATLAB version 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 4-38](#).



# Synplify DSP Upsample

Increases the sample rate of the input by inserting zeroes.

## Library

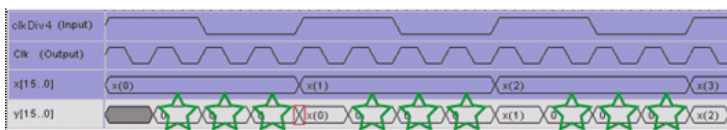
Synplify DSP [Signal Operations](#)

## Description

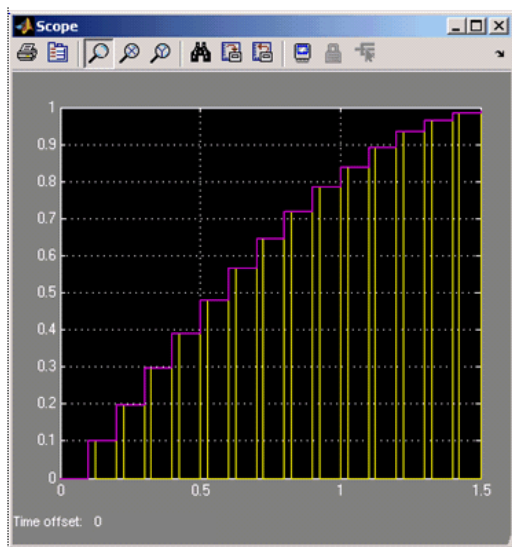


The Synplify DSP Upsample block upsamples the sample rate of the input by adding samples. With an upsampling rate  $L$ , for every sample at the input, the software inserts  $L-1$  samples at the output. This means that the sample rate at the output is the input sample rate multiplied by the upsampling 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:



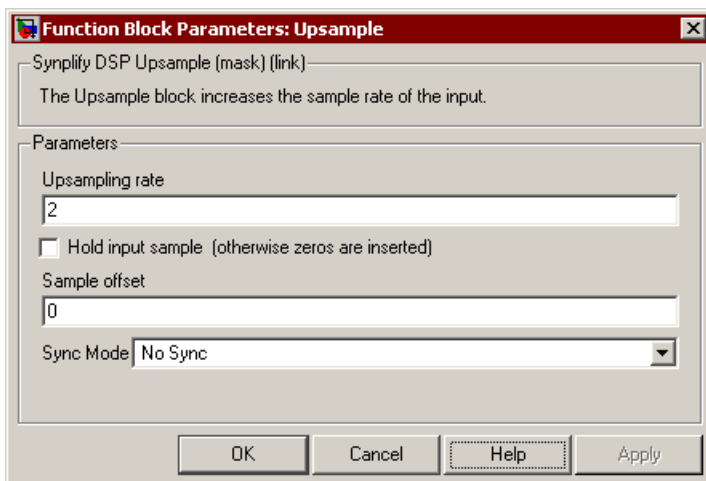
The software uses a delay at the input (based on the input sample rate) followed by a standard upsample operation, where it copies the input as the first sample of every output frame ( $L$  samples), and inserts  $L-1$  zeroes for the other samples in the output frame.

For information about using the Upsample block in multi-rate designs, see [Multi-Rate Design, on page 3-25](#).

## 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

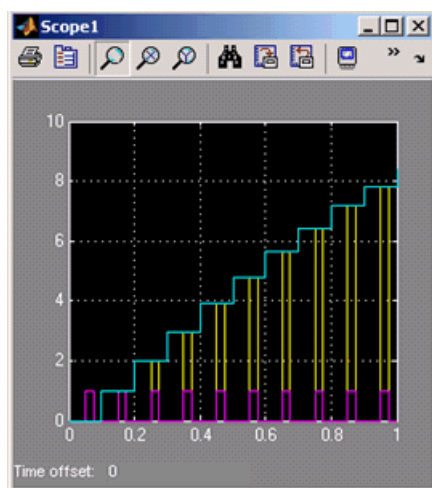
Specifies an offset for the sample rate. The delay you specify here gets added to the output. For a description of sample rates and multi-rate design, see [Multi-Rate Design, on page 3-25](#). This option is available when Hold input sample is disabled.

## 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.



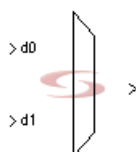
# Synplify DSP Vector Concat

Constructs vectors by bundling up to 32 inputs together.

## Library

Synplify DSP [Signal Operations](#)

## Description



The Synplify DSP 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 32 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 8-267](#) 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 8-267](#).

## 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 Synplify DSP 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.

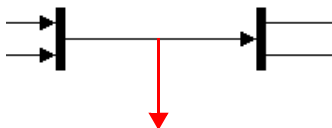
If you connect the input of a block that does not support vector signals, such as IIR, to a vector signal, you get a Simulink error like this one:

```
Error in port widths or dimensions. Output port of  
sinewave_irr.mdl/PortIn is a one-dimensional vector with 4  
elements.
```

Any Synplify DSP 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 Synplify DSP 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 Synplify DSP multi-channelization feature, but at a block level. Synplify DSP's multi-channelization works globally on the entire Simulink model to produce a multi-channel hardware implementation. By using the Synplify DSP 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 Synplify DSP 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 Synplify DSP Vector Concat and Split blocks to create and split vectors.



No Synplify DSP operations allowed

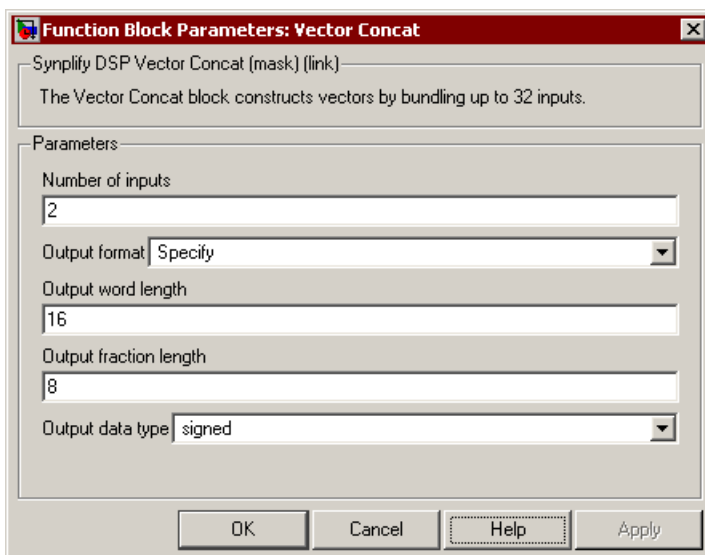
Furthermore, Synplify DSP only supports Simulink Mux-Demux blocks in mux-demux pairs. Having Demux-Mux pairs might create problems in signal routing and decomposition.



## 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 32 inputs.

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

For descriptions of these parameters, see the following:

Output format	<a href="#">Output Format, on page 8-287</a>
Output word length	<a href="#">Output Word Length, on page 8-288</a>
Output fraction length	<a href="#">Output Fraction Length, on page 8-288</a>
Output data type	<a href="#">Output Data Type, on page 8-288</a>

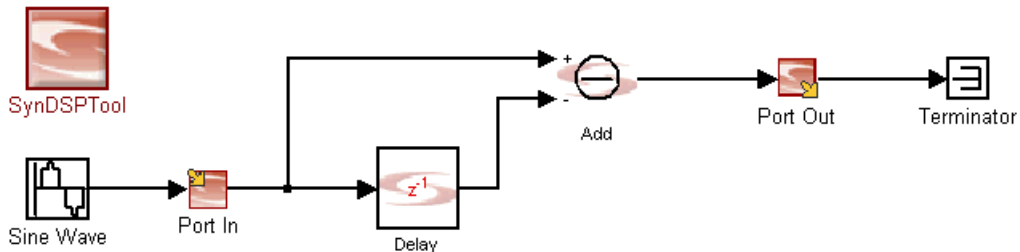
## Vector Signal Examples

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.

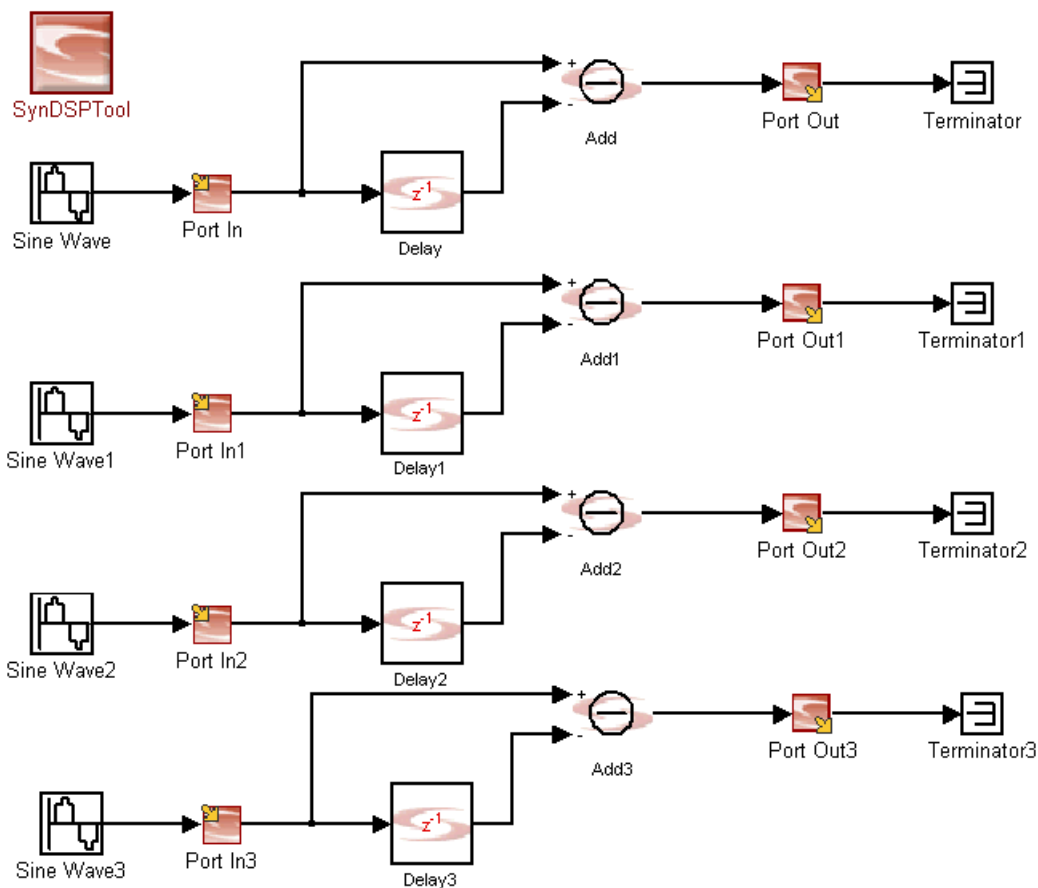
The example shows the vector signals highlighted with bold lines and also displays the vector dimensions.

- 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.

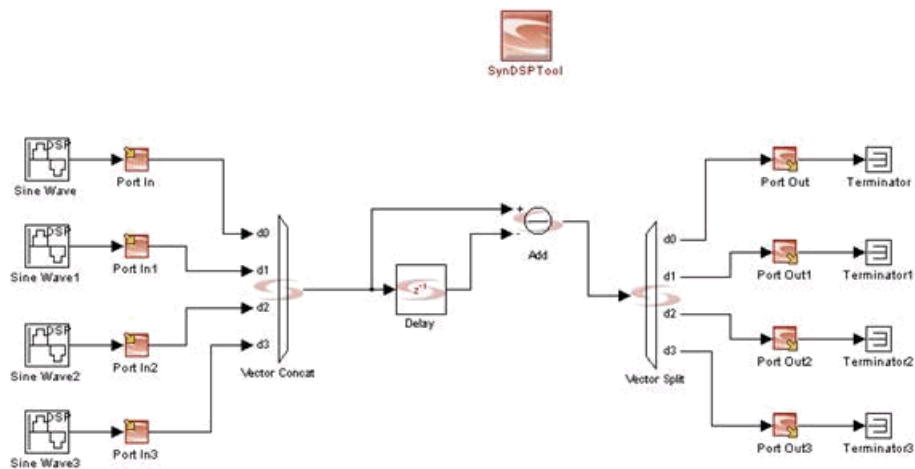
The Sine Wave signal source has been configured to provide a vector signal output. The Port In block inherits the dimensions of this vector signal as do the other Synplify DSP blocks in the model.



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.



# Synplify DSP Vector Expand

Converts scalar input to vector output.

## Library

Synplify DSP [Signal Operations](#)

## Description

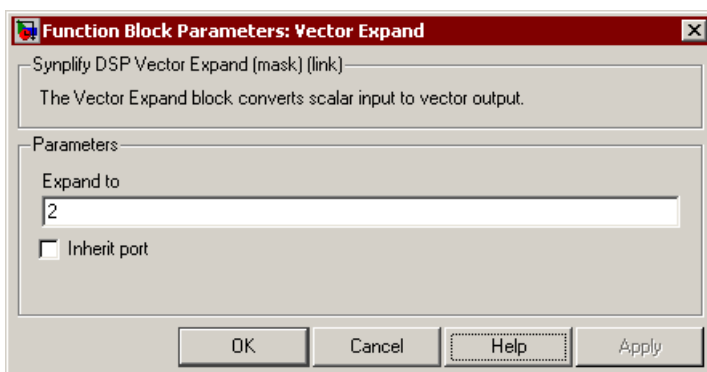


The Synplify DSP Vector Expand block takes the scalar input to the block and converts it to vector output.

## Latency

This block has no latency.

## Vector Expand Parameters



## Expand to

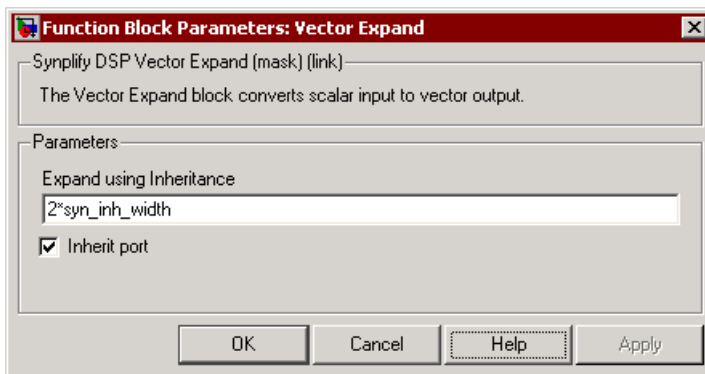
Specifies the output vector size for the block.

## 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 8-292](#) 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:



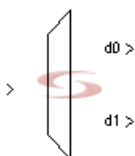
# Synplify DSP Vector Extract

Extracts selected ports from up to a maximum of 2048 input vectors, and outputs up to a maximum of 2048 output ports.

## Library

Synplify DSP [Signal Operations](#)

## Description

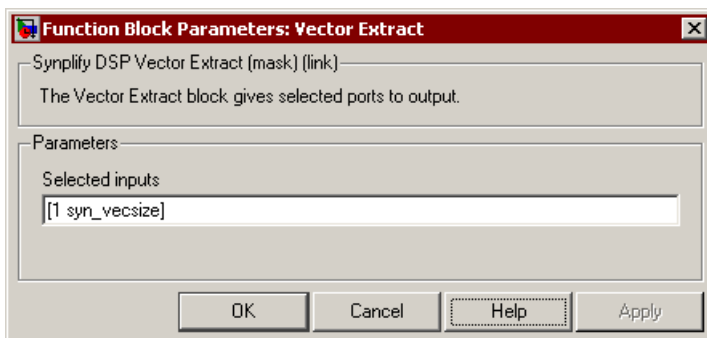


The Synplify DSP Vector Extract extracts the selected inputs and outputs up to 2048 output ports. You can have a maximum of 2048 input vectors. See [Synplify DSP Vector Concat, on page 8-267](#) for a more detailed description. and [Vector Signal Examples, on page 8-270](#) for an example.

## 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.

---



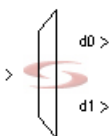
# Synplify DSP Vector Split

Forms signals from vector inputs.

## Library

Synplify DSP [Signal Operations](#)

## Description



The Synplify DSP Vector Split block is a vector de-multiplexer of up to 32 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.

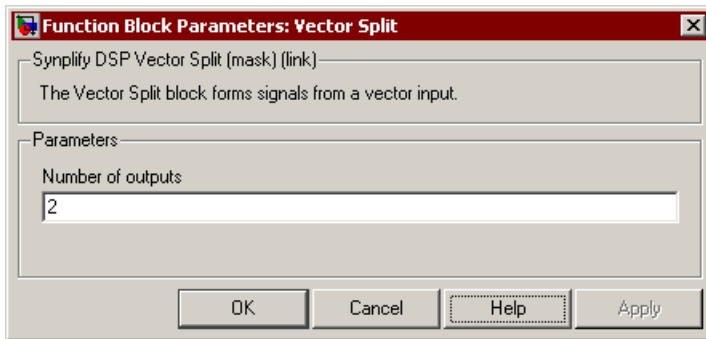
No. of Outputs	Vector Size of Outputs
8	1,1,1,1,1,1,1,1
4	2,2,2,2
3	3,3,2

See [Synplify DSP Vector Concat](#), on page 8-267 for a more detailed description and [Vector Signal Examples](#), on page 8-270 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 32 outputs.

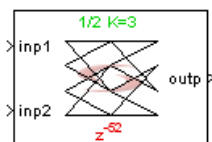
# Synplify DSP Viterbi Decoder

Decodes convolutionally encoded input data.

## Library

Synplify DSP [Communications](#)

## Description



The Synplify DSP Viterbi Decoder decodes convolutionally encoded input data and outputs single bit decoded data.<sup>1</sup> 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 classical ACS unit for speed optimization with an area cost. It uses a modular normalization technique for normalization of state metrics.<sup>2</sup> For further details, see the explanation of these block parameter options ([Viterbi Decoder Parameters, on page 8-281](#)).

This block supports code rates from 1/2 up to 1/7. It allows a maximum constraint length of 8.

- 
1. G. David Forney Jr, "The Viterbi Algorithm", Proceeding IEEE, vol61, pp 268 - 278, March 1973.
  2. 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.

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.

## Icon Annotations

The icon for this block displays the following information:

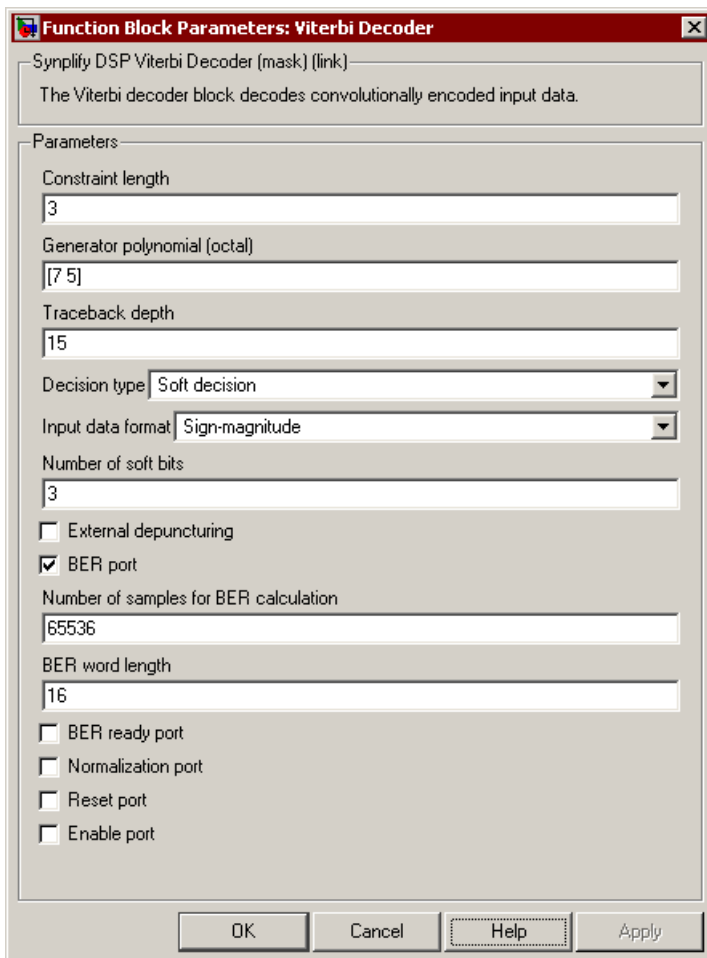
Note	Indicates the code rate and constraint length. For example, $1/2 K = 3$ indicates a $1/2$ code rate, with a constraint length of $K=3$ .
------	--

---

Latency Annotation	The latency of the block is $6 \times \text{ceil}(\text{Traceback depth} / 2) + 4$
--------------------	---

---

## Viterbi Decoder Parameters



The dialog box titled "Function Block Parameters: Viterbi Decoder" contains the following parameters:

- Synplify DSP Viterbi Decoder (mask) (link)  
The Viterbi decoder block decodes convolutionally encoded input data.
- Parameters:
  - Constraint length: 3
  - Generator polynomial (octal): [7 5]
  - Traceback depth: 15
  - Decision type: Soft decision
  - Input data format: Sign-magnitude
  - Number of soft bits: 3
  - ☐ External depuncturing
  - ☒ BER port
  - Number of samples for BER calculation: 65536
  - BER word length: 16
  - ☐ BER ready port
  - ☐ Normalization port
  - ☐ Reset port
  - ☐ Enable port

Buttons at the bottom: 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 constraint length
Punctured data	10 times the constraint length

The traceback method uses a RAM based k-pointer even algorithm (with  $k=3$ ).<sup>1</sup> If you specify an odd value, the traceback depth is adjusted to an even value.

The traceback operation starts from state 0. 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 1 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 8-286](#) for the effects of using soft decision data.

---

1. 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

### 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

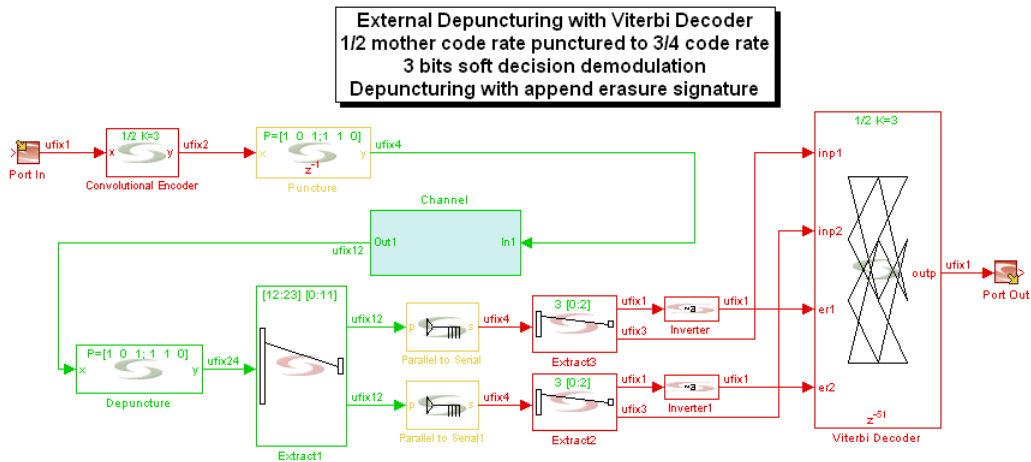
Specifies the bit length of inputs for Soft decision type. For Hard, the bit length of inputs is 1.

### 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 Synplify DSP Depuncture block lets you append erasure signature to output data and work on soft decision inputs.



See [BER Example, on page 8-286](#) 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 option 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 BER Port, this port can be used 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 increase and there may be overflows in state metric update. Normalization port gives an instant rough measure for the error rate on the channel by outputting the number of normalizations (overflows) occurred when state metrics are updated. High 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.

Normalization rates and the BER output can be compared 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**

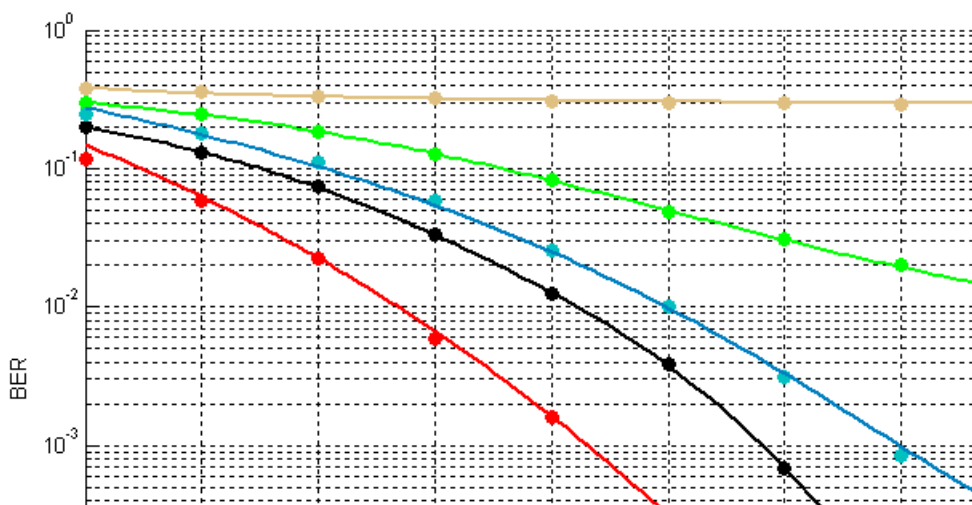
When enabled, it creates an enable port for Viterbi decoder.

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 Synplify DSP blocks. The following parameters are defined here:

- [Output Format Options, on page 8-287](#)
- [Overflow Saturation Options, on page 8-289](#)
- [Underflow Rounding Options, on page 8-289](#)
- [Special Variables, on page 8-292](#)

## Output Format Options

The following options for specifying the output are described here:

- [Output Format, on page 8-287](#)
- [Output Word Length, on page 8-288](#)
- [Output Fraction Length, on page 8-288](#)
- [Output Data Type, on page 8-288](#)

### 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. 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 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
<b>-1.75</b>	-2	-2	-2	-1	-1	-2
<b>-1.50</b>	-2	-1	-2	-1	-1	-2
<b>-1.25</b>	-2	-1	-1	-1	-1	-1
<b>-1.00</b>	-1	-1	-1	-1	-1	-1
<b>-0.75</b>	-1	-1	-1	0	0	-1
<b>-0.50</b>	-1	0	0	0	0	-1
<b>-0.25</b>	-1	0	0	0	0	0
<b>0.00</b>	0	0	0	0	0	0
<b>0.25</b>	0	0	0	0	1	0
<b>0.50</b>	0	1	0	0	1	1
<b>0.75</b>	0	1	1	0	1	1
<b>1.00</b>	1	1	1	1	1	1

Value	Floor	Nearest	Convergent	Fix	Ceil	Round
<b>1.25</b>	1	1	1	1	2	1
<b>1.50</b>	1	2	2	1	2	2
<b>1.75</b>	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
<b>01.10010</b>	01.10	01.10	01.10	01.10	01.11	01.10
<b>10.01110</b>	10.01	10.10	10.10	10.10	10.10	10.10
<b>01011.1100</b>	01011.1	01100.0	01100.0	01011.1	01100.0	01100.0
<b>11010.0100</b>	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
<b>1.5625</b>	1.5	1.5	1.5	1.5	1.75	1.5
<b>-1.5625</b>	-1.75	-1.5	-1.5	-1.5	-1.5	-1.5
<b>11.75</b>	11.5	12	12	11.5	12	12
<b>-11.75</b>	-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, and Number of shift bits, when applicable. You can apply any mathematical operation on these variables. For example, if you specify an Output word length of  $2 * \text{syn\_inp\_wl}$ , 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.



## CHAPTER 9

# Synplify DSP Functions

---

This chapter describes the Synplify DSP functions in alphabetical order.

- [syn\\_bitrev](#), on page 9-2
- [syn\\_get\\_coefs](#), on page 9-4
- [syn\\_get\\_datatype](#), on page 9-5
- [syn\\_get\\_dspstartup](#), on page 9-6
- [syn\\_get\\_wl](#), on page 9-7
- [syn\\_get\\_wordlength](#), on page 9-9
- [syn\\_read\\_hex](#), on page 9-11
- [syn\\_set\\_ate](#), on page 9-13
- [syn\\_set\\_atm](#), on page 9-15
- [syn\\_set\\_dspstartup](#), on page 9-17
- [syn\\_set\\_portcapture](#), on page 9-18
- [syn\\_set\\_portregister](#), on page 9-19
- [syn\\_unlink](#), on page 9-20
- [syndspdemo](#), on page 9-21
- [syndspdoc](#), on page 9-22
- [syndsplib](#), on page 9-23
- [syndsproot](#), on page 9-25
- [syndsptool](#), on page 9-26
- [syndspver](#), on page 9-27

# syn\_bitrev

M Control function that reverses the order of bits in an unsigned integer.

## Syntax

**syn\_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 syn\_bitrev needs to know what bit-width to use when reversing the bits.

## Description

The syn\_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 Synplify DSP 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.

# syn\_get\_coefs

Gets the filter coefficients from a Synplify DSP 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 a Synplify 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 Synplify DSP 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 Synplify DSP information like word length, fraction length, or data type.

## Examples

```
syn_get_datatype('uint13')
[wl fl dt]=syn_get_datatype('sfixed13_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

Note that 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 9-17 for details).

## Examples

```
syn_get_dspstartup  
syn_get_dspstartup('topLevel')
```

# syn\_get\_wl

Calculates the word length required to represent a given set of values. This function will be removed from future releases. Use `syn_get_wordlength` instead, which has more rounding options ([syn\\_get\\_wordlength](#), on page 9-9).

## Syntax

**syn\_get\_wl** (<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 <floatingLength>	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. 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	Affects the value because it affects the underflow behavior to represent the value. If it is not specified, the function uses the round option. This value corresponds to the <code>syn_get_wordlength</code> nearest option i

## 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_wl` function returns the maximum word length required to represent all elements of <value> for the given options.

## Examples

```
syn_get_wl(1)
syn_get_wl(3.14)
syn_get_wl(pi)
[w,v]=syn_get_wl(pi,'ufixed','fl',2,'round')
[w,v]=syn_get_wl({exp(1) pi},'fl',4)
```



# 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 <floatingLength>	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 4-43](#) 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.
TT	2	Type. This can be 00, 01, 02, or 04. <ul style="list-style-type: none"> <li>• 00 - Data record. A record containing data and the 2-byte address for the data to reside.</li> <li>• 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.</li> <li>• 02 - Extended segment address record.</li> <li>• 04 - Extended linear address record.</li> </ul>

Field	Characters	Description
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\_ate

Allows you to configure Synplify DSP timing modes. This function will be removed in the next release. Use `syn_set_atm` instead, for timing configuration ([syn\\_set\\_atm](#), on page 9-15).

## Syntax

```
syn_set_ate
```

## Description

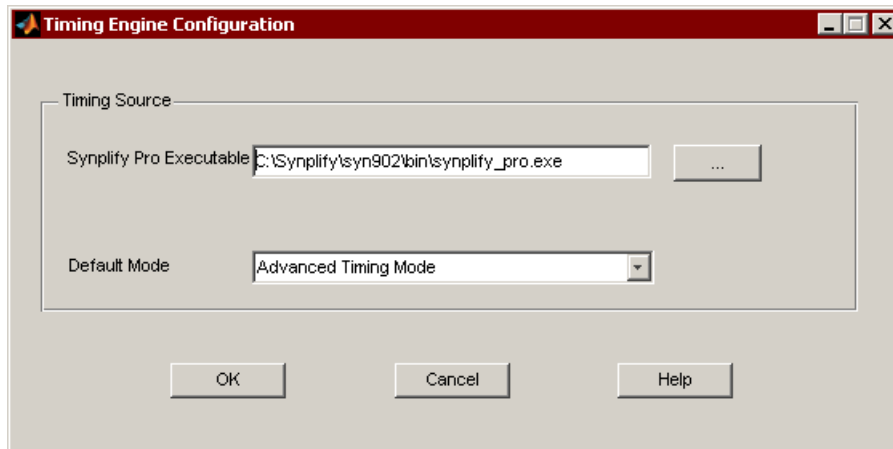
The `syn_set_ate` function opens a dialog box that lets you set the timing mode. See [Timing Engine Configuration Dialog Box](#), on page 9-13 for details.

## Example

```
syn_set_ate
```

## 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_ate` at the MATLAB command prompt. See [Configuring Synplify DSP Timing Modes for FPGAs](#), on page 4-51 for details about setting timing modes.



Synplify Pro executable	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.
Default Mode	<p>Sets the default timing mode used to calculate timing parameters for all DSP synthesis runs.</p> <ul style="list-style-type: none"><li>• <b>Advanced Timing Mode</b> Uses the Synplify Pro timing engine and target-specific timing data to produce more accurate results.</li><li>• <b>Estimation mode</b> Uses simpler, latency-based device characterizations as a basis for optimizations. Estimation mode is faster, but the results are less accurate.</li></ul>

# syn\_set\_atm

Allows you to configure Synplify DSP timing modes.

## Syntax

```
syn_set_atm
```

## Description

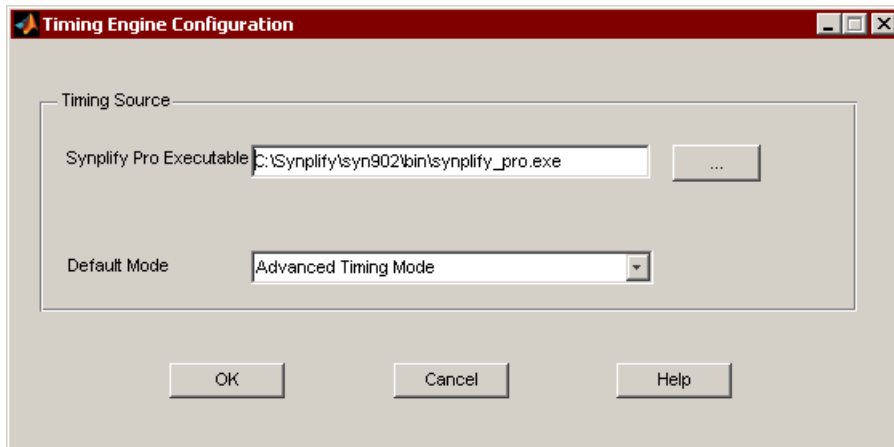
The `syn_set_atm` function opens a dialog box that lets you set the timing mode. See [Timing Engine Configuration Dialog Box, on page 9-13](#) for details.

## 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 Synplify DSP Timing Modes for FPGAs, on page 4-51](#) for details about setting timing modes.



Synplify Pro executable	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.
Default Mode	<p>Sets the default timing mode used to calculate timing parameters for all DSP synthesis runs.</p> <ul style="list-style-type: none"><li>• <b>Advanced Timing Mode</b> Uses the Synplify Pro timing engine and target-specific timing data to produce more accurate results.</li><li>• <b>Estimation mode</b> Uses simpler, latency-based device characterizations as a basis for optimizations. Estimation mode is faster, but the results are less accurate.</li></ul>

---



# 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 4-2](#).

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 9-6](#)).

## 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 parameterf for all ports:

```
syn_set_portregister('set');
```

# syn\_unlink

Unlinks any instance of a Synplify DSP 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 Synplify DSP 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')
```

# syndspdemo

Runs the Synplify DSP demos.

## Syntax

**syndspdemo** [('<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
gsm_ddc	Digital Down Converter
im_histogram	Histogram computation with the Synplify DSP M Control and RAM blocks
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 Synplify DSP Viterbi Decoder block
qam16withviterbiwithpunc	QAM 16 modem with the Synplify DSP Viterbi Decoder using 7/8 puncturing
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 used in the basic tutorial

## Description

The syndspdemo function locates and executes all preparations for the demos that are bundled with the Synplify DSP 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

```
syndspdemo
```

```
syndspdemo ( 'qam16' )
```

# syndspdoc

Shows the documentation for blocks.

## Syntax

```
syndspdoc('<blockName>')
```

The <blockName> argument must be the exact name of the block and must be enclosed in single quotes. For example:

```
syndspdoc( 'Gain' )
```

## Description

The syndspdoc function opens the Help browser with detailed information on the specified block. You must specify the exact name of the block.

## See Also

[syndsplib](#), [syndspver](#)

# syndsplib

Manages the Synplify DSP blockset library.

## Syntax

**syndsplib** [[('<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.

---

**Note:** While older versions are available for converting legacy designs, these libraries are not functional.

---



## Description

The syndsplib function opens the Synplify DSP blockset. Use <version> if you want to open a particular blockset. The version number for the current blockset is 8.

## Examples

```
model=syndsplib('info');  
[model,version]=syndsplib('open');  
[model,version,path]=syndsplib('open',2);
```

## See Also

[syndspdemo](#), [syndspdoc](#), [syndspver](#)



To access other information about the Synplify DSP product, use the following MathWorks commands:

To view the table of contents	help SynDSP
-------------------------------	-------------

To view the Release Notes	info SynDSP
---------------------------	-------------

To view the online documentation	doc SynDSP
----------------------------------	------------

## syndsproot

Returns the location where Synplify DSP was installed.

### Syntax

**syndsproot**

### Description

The `syndsproot` function stores the location of the Synplify DSP installation directory when you are setting up the Synplify DSP MATLAB interface. It is used as a reference point to find files in the Synplify DSP tree.

### See Also

[syndspver](#)

# syndsptool

Manages the Synplify DSP synthesis application.

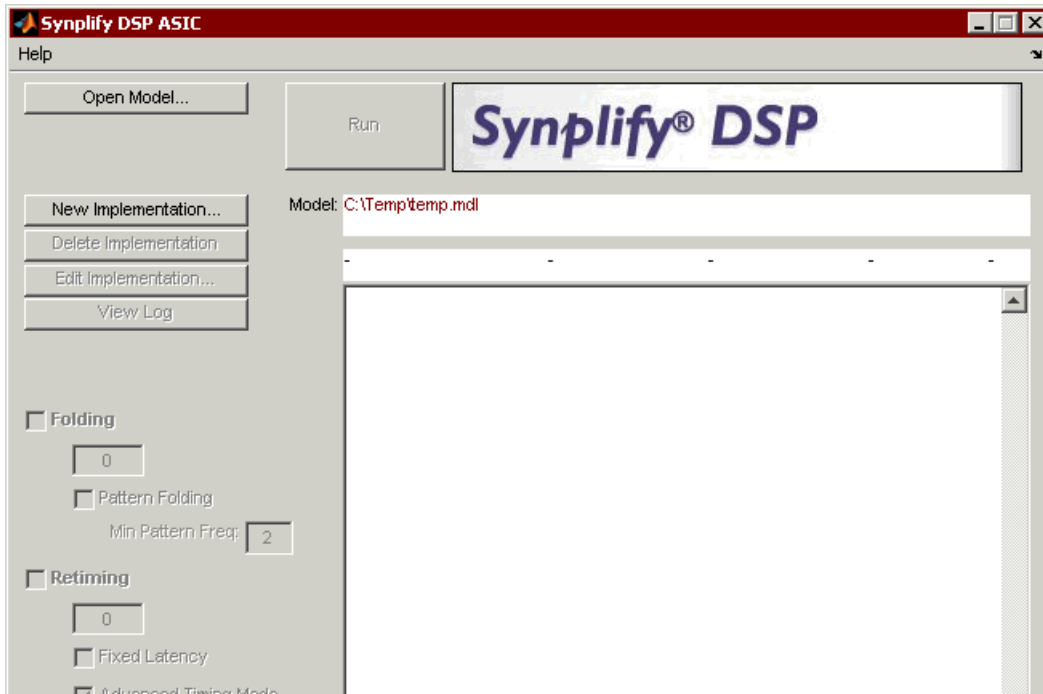
## Syntax

**syndsptool** [('Model', '<modelName>')]

The optional Model argument specifies a design to open in the application and must be enclosed in single quotes.

## Description

The `syndsptool` function opens the Synplify DSP application. It brings up a toolbox interface where you can set optimization options and generate RTL code.



## Examples

```
syndsptool;  
syndsptool('Model','design');  
syndsptool('Model','design.mdl');  
syndsptool('Model','C:\Temp\design.mdl');
```

# syndspver

Displays version information for the current MATLAB, Simulink and Synplify DSP installations.

## Syntax

**syndspver** [*('<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 the version information as well as the build information for the blockset executables.
silent	Suppresses the display of results. Use this argument to assign information to output variables.

---

## Description

The syndspver function displays the standard version information for the Synplify DSP, MATLAB, and Simulink software.

The function returns up to three outputs:

- The software versions
- The version of the Synplify DSP toolbox executable
- The versions of the Synplify DSP blockset executables

## Examples

```
[v,l,b]=syndspver('silent')
```

```
[v,l,b]=syndspver
```

```
[v,l,b]=syndspver('all')
```

The output of this function looks like this:

```
v =  
Name: 'Synplicity Synplify DSP'  
Version: 'ToT'  
Release: '(Development)'  
Date: '07-Sep-2005'  
  
l=  
Expiration: '09-apr-2007'  
Id: 'N2411743111074777'  
Vendor: 'FPGA'  
  
b =  
Syndsptool: 'synplify_dsp.exe Oct 10 2006 18:35:06'  
Syndsplib: {1x55 cell}
```

## See Also

[syndsproot](#), [syndsplib](#)

## APPENDIX A

# Blockset Summary

---

The following table provides a handy summary of various block features.

# Summary of Synplify DSP Block Features

The following table lists the blocks alphabetically and contains information about various features for each block.

## Reading the Table

Note the following about the table:

- **Blocks column**  
Primitive blocks are in bold; custom blocks are in a regular font.
- **Internal Precision Limit column**  
All blocks support the Simulink fixed-point input wordlength of 128 bits or less; the exceptions to this rule are noted in this table.
- **Datapath Quantization Propagation Rules column**  
By default, all blocks with internal arithmetic calculations propagate the fixed-point size in full precision. Blocks indicated in the table include other propagation rule choices for limiting wordlength growth.
- **Output Quantization Rules column**  
All blocks can output their calculated results in full precision. Blocks indicated in the table allow additional quantization or extension to different wordlength or fixed-point formats.

## Block Summary Table

Blocks	Internal Precision Limit	Datapath Quantization Propagation Rules	Output Quantization	Vector Support
<b>Abs</b>			Yes	-
<b>Accumulator</b>			Yes	Yes
<b>Add</b>			Yes	Yes
<b>Binary Logic</b>			-	Yes
<b>Black Box</b>			-	-
Block Deinterleaver			-	-
Block Interleaver			-	-

<b>Blocks</b>	<b>Internal Precision Limit</b>	<b>Datapath Quantization Propagation Rules</b>	<b>Output Quantization</b>	<b>Vector Support</b>
CIC			Yes	Yes
Commutator			Yes	Yes
<b>Comparator</b>			-	Yes
<b>Concat</b>			-	-
<b>Constant</b>			-	Yes
<b>Convert</b>			Yes	Yes
Convolutional Deinterleaver			-	Yes
Convolutional Encoder			-	-
Convolutional Interleaver			-	Yes
<b>CORDIC Exp</b>	120 bits		-	-
<b>CORDIC Log</b>	120 bits		-	-
<b>CORDIC Polar</b>	52 bits	-	-	-
<b>CORDIC Rotator</b>	120 bits	-	-	-
<b>CORDIC SinCos</b>	120 bits	-	-	-
<b>CORDIC Sqrt</b>	64 bits	-	-	-
<b>Counter</b>		-	-	Yes
DDS		-	-	-
Decommutator				Yes
<b>Delay</b>			-	Yes
<b>Demux</b>		-	-	-
<b>Depuncture</b>		-	-	-
Differentiator			Yes	Yes

<b>Blocks</b>	<b>Internal Precision Limit</b>	<b>Datapath Quantization Propagation Rules</b>	<b>Output Quantization</b>	<b>Vector Support</b>
<b>DivMod</b>		-	-	-
<b>Downsample</b>			-	Yes
<b>Extract</b>		-	-	-
<b>FDATool</b>	N/A	N/A	N/A	N/A
<b>FFT</b>		Yes	Yes	Yes
<b>FIFO</b>		-	-	Yes
<b>FIR</b>		Yes	Yes	Yes
<b>FIR Engine</b>			-	Yes
<b>FIR Rate Converter</b>			-	Yes
<b>Gain</b>			Yes	Yes
<b>IIR</b>		Yes	Yes	-
<b>In</b>			-	Yes
<b>Integrator</b>			Yes	Yes
<b>Inverter</b>		-	Yes	-
<b>Log</b>		-	Yes	-
<b>M Control</b>	52 bits	-	-	-
<b>Mealy State Machine</b>		-	-	-
<b>MinMax</b>			-	Yes
<b>Moore State Machine</b>		-	-	-
<b>Mult</b>			Yes	Yes
<b>Mux</b>			Yes	Yes
<b>Negate</b>	126 bits		Yes	-
<b>Out</b>			-	Yes



<b>Blocks</b>	<b>Internal Precision Limit</b>	<b>Datapath Quantization Propagation Rules</b>	<b>Output Quantization</b>	<b>Vector Support</b>
Parallel to Serial		-	-	-
<b>Permutation</b>		-	-	-
<b>Port In</b>			Yes	Yes
<b>Port Out</b>			Yes	Yes
<b>Pow</b>		-	Yes	-
<b>Puncture</b>			-	Yes
<b>RAM</b>			-	Yes
Ramp		-	Yes	-
Random	19 bits	-	-	-
Recast			-	Yes
<b>Reed Solomon Decoder</b>				
<b>Reed Solomon Encoder</b>				
Register			-	Yes
Reloadable FIR		Yes	Yes	
<b>ROM</b>		Yes	-	Yes
Sequence		-	Yes	-
Serial to Parallel		-	Yes	-
<b>Shift Register</b>			-	Yes
<b>Shifter</b>		-	Yes	-
Sign		-	-	-
<b>SinCos</b>	18-bit input fraction length; 54-bit output		Yes	Yes
<b>Sqrt</b>			Yes	Yes
<b>Subsystem</b>		-	-	-

<b>Blocks</b>	<b>Internal Precision Limit</b>	<b>Datapath Quantization Propagation Rules</b>	<b>Output Quantization</b>	<b>Vector Support</b>
<b>SynDSPTool</b>	N/A	N/A	N/A	N/A
<b>SynFixPtTool</b>	N/A	N/A	N/A	N/A
<b>Upsample</b>				Yes
<b>Vector Concat</b>			Yes	Yes
<b>Vector Expand</b>			-	Yes
<b>Vector Extract</b>			-	Yes
<b>Vector Split</b>			-	Yes
<b>Viterbi Decoder</b>				

# Index

---

## A

- Abs block, Synplify DSP [8-15](#)
  - parameters [8-15](#)
- absolute value
  - calculating [8-15](#)
- Accumulator block, Synplify DSP [8-17](#)
  - parameters [8-18](#)
- Active-HDL [4-65](#)
- adaptive filters
  - using [4-12](#)
  - using vectors [4-23](#)
- adaptive FIR [8-213](#)
- Add block, Synplify DSP [8-19](#)
  - parameters [8-20](#)
- adder
  - Synplify DSP block [8-19](#)
- advanced timing engine *See* advanced timing mode
- advanced timing mode [4-51](#)
  - Synplify Pro executable [4-52](#)
- AND operation
  - Binary Logic block [8-23](#)
- area
  - decreasing with folding [2-34](#)
- asynchronous resets
  - RTL code, Synplify DSP [3-24](#)
  - testbenches, Synplify DSP [3-25](#)
- asynchronous resets, Synplify DSP [3-23](#)

## B

- Binary Logic block, Synplify DSP [8-22](#), [8-35](#)
  - parameters [8-22](#)
- binary logic functions [8-22](#)
- binary point
  - interpretation of fixed-point numbers [8-49](#)
- bit reversal
  - M compiler [9-2](#)
- Black Box block, Synplify DSP [8-25](#)
  - parameters [8-27](#)
- black boxes
  - advantages [4-24](#)
  - folding [4-27](#)
  - multichannelization [4-28](#)
  - port interface [4-26](#)
  - retiming [4-27](#)
  - RTL [4-27](#)
  - using [4-24](#)
- block [5-7](#)
- Block Deinterleaver block, Synplify DSP [8-31](#)
- Block Interleaver block, Synplify DSP [8-33](#)
- block parameters
  - Abs, Synplify DSP [8-15](#)
  - Accumulator, Synplify DSP [8-18](#)
  - Add, Synplify DSP [8-20](#)
  - Binary Logic, Synplify DSP [8-22](#)
  - Black Box, Synplify DSP [8-27](#)
  - Block Deinterleaver [8-32](#)
  - Block Interleaver [8-34](#)
  - CIC, Synplify DSP [8-36](#)
  - Commutator, Synplify DSP [8-40](#)
  - Comparator, Synplify DSP [8-41](#)
  - Concat, Synplify DSP [8-44](#)
  - Constant, Synplify DSP [8-46](#)
  - Convert, Synplify DSP [8-50](#)
  - Convolutional Deinterleaver, Synplify DSP [8-54](#)
  - Convolutional Encoder, Synplify DSP [8-58](#)
  - Convolutional Interleaver [8-60](#)
  - CORDIC Div, Synplify DSP [8-62](#)
  - CORDIC Log, Synplify DSP [8-64](#)

---

CORDIC Polar, Synplify DSP [8-66](#)  
 CORDIC Rotator, Synplify DSP [8-69](#)  
 Counter, Synplify DSP [8-78](#)  
 Decommulator, Synplify DSP [8-90](#)  
 Delay, Synplify DSP [8-91](#)  
 Demux, Synplify DSP [8-92](#)  
 Depuncture, Synplify DSP [8-95](#)  
 Differentiator [8-97](#)  
 DivMod, Synplify DSP [8-100](#)  
 Downsampler, Synplify DSP [8-108](#)  
 Extract, Synplify DSP [8-110](#)  
 FFT, Synplify DSP [8-113](#)  
 FIFO, Synplify DSP [8-119](#)  
 FIR Engine, Synplify DSP [8-129](#)  
 FIR Rate Converter, Synplify DSP [8-134](#)  
 FIR, Synplify DSP [8-122](#)  
 Gain, Synplify DSP [8-139](#)  
 IIR, Synplify DSP [8-143](#)  
 Integrator, Synplify DSP [8-148](#)  
 Inverter, Synplify DSP [8-151](#)  
 Log, Synplify DSP [8-153](#)  
 M Control, Synplify DSP [8-155](#)  
 Mealy State Machine, Synplify DSP [8-155](#), [8-158](#)  
 MinMax, Synplify DSP [8-161](#)  
 Moore State Machine, Synplify DSP [8-163](#)  
 Mult, Synplify DSP [8-165](#)  
 Mux, Synplify DSP [8-167](#)  
 Negate, Synplify DSP [8-168](#)  
 Parallel to Serial, Synplify DSP [8-171](#)  
 Permutation, Synplify DSP [8-173](#)  
 Port In, Synplify DSP [8-175](#)  
 Port Out, Synplify DSP [8-178](#)  
 Pow, Synplify DSP [8-179](#)  
 Puncture, Synplify DSP [8-182](#)  
 RAM, Synplify DSP [8-184](#)  
 Recast, Synplify DSP [8-197](#)  
 Reed-Solomon Encoder, Synplify DSP [8-209](#)  
 Register, Synplify DSP [8-212](#)  
 RFIR, Synplify DSP [8-215](#)  
 ROM, Synplify DSP [8-220](#)  
 Serial to Parallel, Synplify DSP [8-225](#)  
 Shift Register, Synplify DSP [8-228](#)  
 Shifter, Synplify DSP [8-233](#)  
 Sign, Synplify DSP [8-235](#)  
 SinCos, Synplify DSP [8-75](#), [8-237](#), [8-250](#)  
 specifying [4-6](#)  
 Sqrt, Synplify DSP [8-240](#), [8-246](#)  
 SynDSPTool, Synplify DSP [8-254](#)  
 Upsampler, Synplify DSP [8-265](#)

Vector Mux, Synplify DSP [8-269](#)  
 Vector Split, Synplify DSP [8-273](#), [8-275](#), [8-278](#)

blocks
 

- Abs, Synplify DSP [8-15](#)
- Accumulator, Synplify DSP [8-17](#)
- Add, Synplify DSP [8-19](#)
- adding (Synplify DSP) [4-5](#)
- alphabetical list, Synplify DSP [8-11](#)
- Binary Logic, Synplify DSP [8-22](#)
- Black Box, Synplify DSP [8-25](#)
- Block Deinterleaver, Synplify DSP [8-31](#)
- Block Interleaver, Synplify DSP [8-33](#)
- casting output data type [3-21](#)
- CIC, Synplify DSP [8-35](#)
- Commutator, Synplify DSP [8-39](#)
- Comparator, Synplify DSP [8-41](#)
- Concat, Synplify DSP [8-43](#)
- Constant, Synplify DSP [8-45](#)
- Convert, Synplify DSP [8-48](#)
- Convolutional Deinterleaver, Synplify DSP [8-53](#)
- Convolutional encoder, Synplify DSP [8-56](#)
- Convolutional Interleaver block, Synplify DSP [8-59](#)
- CORDIC Exp, Synplify DSP [8-61](#)
- CORDIC Log, Synplify DSP [8-63](#)
- CORDIC Magnitude, Synplify DSP [8-65](#)
- CORDIC Rotator, Synplify DSP [8-67](#)
- CORDIC SinCos, Synplify DSP [8-74](#)
- CORDIC Sqrt, Synplify DSP [8-76](#)
- Counter, Synplify DSP [8-77](#)
- Decommulator, Synplify DSP [8-89](#)
- Delay, Synplify DSP [8-91](#)
- Demux, Synplify DSP [8-92](#)
- Depuncture, Synplify DSP [8-94](#)
- Differentiator, Synplify DSP [8-96](#)
- Downsampler, Synplify DSP [8-106](#)
- Extract, Synplify DSP [8-109](#)
- FDATool, Synplify DSP [8-109](#), [8-111](#), [8-196](#)
- feature summary [A-2](#)
- FFT, Synplify DSP [8-112](#)
- FIFO, Synplify DSP [8-118](#)
- FIR Engine, Synplify DSP [8-127](#)
- FIR, Synplify DSP [8-120](#), [8-133](#)
- Gain, Synplify DSP [8-138](#)
- IIR, Synplify DSP [8-141](#)
- Integrator, Synplify DSP [8-147](#)
- Inverter, Synplify DSP [8-151](#)
- libraries, Synplify DSP [8-2](#)

- 
- Log, Synplify DSP 8-152
  - M Control, Synplify DSP 8-154
  - Mealy State Machine, Synplify DSP 8-157
  - MinMax, Synplify DSP 8-160
  - Moore State Machine, Synplify DSP 8-162
  - Mult, Synplify DSP 8-164
  - Mux, Synplify DSP 8-166
  - Negate, Synplify DSP 8-168
  - Parallel to Serial, Synplify DSP 8-170
  - parameterized 5-16
  - Permutation, Synplify DSP 8-169, 8-172
  - Port In, Synplify DSP 8-174
  - Pow, Synplify DSP 8-179
  - Puncture, Synplify DSP 8-181
  - RAM, Synplify DSP 8-183
  - Recast, Synplify DSP 8-196
  - Reed-Solomon Decoder, Synplify DSP 8-200, 8-203
  - Reed-Solomon Encoder, Synplify DSP 8-207
  - Register, Synplify DSP 8-211
  - RFIR, Synplify DSP 8-213
  - ROM, Synplify DSP 8-219
  - Serial to Parallel, Synplify DSP 8-224
  - Shift Register, Synplify DSP 8-227
  - Shifter, Synplify DSP 8-232
  - Sign, Synplify DSP 8-234
  - SimControl Tool, Synplify DSP 8-249
  - SinCos, Synplify DSP 8-236
  - Smart Black Box, Synplify DSP 8-238
  - Sqrt, Synplify DSP 8-245
  - syndspdoc function 9-22
  - SynDSPTool, Synplify DSP 8-253
  - SynFixPtTool 8-261
  - Synplify DSP 8-11
  - types 5-2
  - Upsample, Synplify DSP 8-263
  - Vector Concat, Synplify DSP 8-267
  - Vector Expand, Synplify DSP 8-273
  - Vector Extract, Synplify DSP 8-275
  - Vector Split, Synplify DSP 8-277
  - Viterbi Decoder, Synplify DSP 8-279
  - blocksets
    - custom. *See* custom blocksets 5-5
    - syndsplib function 9-23
    - Synplify DSP libraries 8-2
- ## C
- Cadence IUS54 4-65
  - casting 3-21
  - channels
    - creating multiple 4-60
  - CIC block, Synplify DSP 8-35
  - CIC filters 8-35
  - clock alignment 3-29
  - clock domains 3-2
  - clocks 3-2
    - multi-rate designs 3-32
  - coefficients
    - syn\_get\_coefs 9-4
  - command line commands
    - Synplify DSP information 9-25
  - commands
    - do file 4-10
  - Comparator block, Synplify DSP 8-41
    - parameters 8-41
  - comparison operations 8-41
  - Concat block, Synplify DSP 8-43
  - Constant block, Synplify DSP 8-39, 8-45
    - parameters 8-46
  - constraint file
    - Synplify DSP ASIC 4-8
  - Control Logic library 8-4
  - Convert block, Synplify DSP 8-48
    - examples 8-49
    - parameters 8-50
  - Convolutional Deinterleaver block, Synplify DSP 8-53
  - Convolutional Interleaver block, Synplify DSP 8-59
  - CORDIC algorithms
    - Synplify DSP 3-3
  - CORDIC Div block, Synplify DSP 8-56
    - parameters 8-62
  - CORDIC Exp block, Synplify DSP 8-61
  - CORDIC Log block, Synplify DSP 8-63
    - parameters 8-64
  - CORDIC Magnitude block, Synplify DSP 8-65
  - CORDIC Polar block, Synplify DSP

---

- parameters [8-66](#)
- CORDIC Rotator block, Synplify DSP [8-67](#)
  - parameters [8-69](#)
- CORDIC SinCos block, Synplify DSP [8-74](#)
  - parameters [8-75](#)
- CORDIC Sqrt block, Synplify DSP [8-76](#)
- co-simulation
  - configuration [7-2](#)
- cosine
  - CORDIC SinCos block [8-74](#)
  - SinCos block [8-236](#)
- counter
  - Synplify DSP [8-77](#)
- Counter block, Synplify DSP [8-77](#)
  - parameters [8-78](#)
- counter, free-running [8-80](#)
- custom block
  - creating [5-6](#)
- custom blocks
  - advantages [5-2](#)
  - CIC [8-35](#)
  - DDS [8-83](#)
  - description [5-2](#)
  - differentiator [8-96](#)
  - FIR Rate Converter [8-133](#)
  - Integrator [8-147](#)
  - masks [5-7](#)
  - MinMax [8-160](#)
  - parameterized blocks [5-16](#)
  - Ramp [8-191](#)
  - Random [8-194](#)
  - Sequence [8-222](#)
  - Sign, Synplify DSP [8-234](#)
  - syn\_unlink function [9-20](#)
  - unlinking [9-20](#)
- custom blocksets [5-5](#)
- custom libraries
  - converting [5-26](#)

## D

- data type
  - converting input data type [8-48](#)
  - output data type [3-21](#)
- data types [3-20](#)
  - casting output [3-21](#)

- casting output data type [3-21](#)
- converting from Simulink to Synplify DSP [9-5](#)
- displaying for model ports [3-20](#)
- fixed-point in Synplify DSP [3-20](#)
- Synplicity implementation [3-20](#)
- validating fixed-point algorithm [4-42](#)
- validating floating-point algorithm [4-40](#)
- DDS block, Synplify DSP [8-83](#)
  - parameters [8-85](#)
- decimation
  - defined [3-26](#)
- decimators
  - CIC [8-36](#)
  - FIR polyphase [8-135](#)
- delay
  - defined [3-26](#)
- Delay block, Synplify DSP [8-91](#)
  - parameters [8-91](#)
- delays
  - Register block, Synplify DSP [8-211](#)
- demos
  - syndspdemo function [9-21](#)
  - Synplify DSP [9-21](#)
- de-multiplexer
  - Demux block, Synplify DSP [8-92](#)
- demultiplexer
  - Vector Split block, Synplify DSP [8-277](#)
- Demux block
  - using in Synplify DSP [8-268](#)
- Demux block, Synplify DSP [8-92](#)
  - parameters [8-92](#)
- design capture [2-3](#)
- Design Compiler, do file [4-10](#)
- design flows
  - Synplify DSP ASIC [1-11](#), [1-13](#)
  - Synplify DSP FPGA [1-5](#), [1-8](#)
  - tutorial [2-2](#)
- design synthesis [2-29](#)
- Differentiator block, Synplify DSP [8-96](#)
- Digital down converter demo [9-21](#)
- direct digital synthesizer. *See* DDS block, Synplify DSP.
- discrete time differentiation [8-96](#)
- discrete time integration [8-147](#)
- do files

- commands for synthesis [4-10](#)
  - Design Compiler [4-10](#)
  - RTL Compiler [4-9, 4-10](#)
  - Synplify DSP ASIC [4-8](#)

- doc SynDSP command [9-25](#)

- downsample
  - offset [3-27](#)

- Downsample block, Synplify DSP [8-106](#)
  - parameters [8-108](#)

- DSP Basics library [8-5](#)

## E

- error-correction

- Reed-Solomon codes [8-208](#)
  - Reed-Solomon Decoder [8-200](#)
  - Reed-Solomon Encoder block, Synplify DSP [8-207](#)

- examples

- Convert block binary point, Synplify DSP [8-49](#)

- exponent
  - calculating [8-61](#)

## F

- Fast Fourier Transform [8-112](#)

- FDATool block, Synplify DSP [8-109, 8-111, 8-196](#)

- FFT block, Synplify DSP [8-112](#)
  - parameters [8-113](#)

- FIFO block, Synplify DSP [8-118](#)
  - parameters [8-119](#)

- filter coefficients [4-12](#)

- Filtering library [8-5](#)

- filters

- CIC [8-35](#)
  - differentiator [8-96](#)
  - FIR [8-120](#)
  - IIR [8-141](#)
  - Integrator [8-147](#)
  - polyphase [8-133](#)

- FIR block, Synplify DSP [8-120](#)
  - parameters [8-122](#)

- FIR Engine block, Synplify DSP [8-127](#)
  - adaptive FIR application [8-213](#)
  - parameters [8-129](#)
  - reloadable FIR application [8-213](#)

- FIR filters

- implementing [4-12](#)

- FIR Rate Converter block, Synplify DSP [8-133](#)
  - parameters [8-134](#)

- FIRs

- adaptive [8-213](#)
  - reloadable [8-213](#)

- fixed point data type
  - Synplify DSP [3-19](#)

- Fixed Point Settings toolbox
  - fixed-point algorithm [4-42](#)
  - floating-point data type [4-40](#)
  - full-accuracy algorithm [4-40](#)
  - plotting [4-42](#)

- fixed-point

- comparing to floating-point results (tutorial) [2-13](#)
  - ignoring (tutorial) [2-20](#)

- fixed-point algorithm
  - comparing to floating-point [4-42](#)

- fixed-point data type
  - setting [4-38](#)
  - setting options [8-262](#)
  - Synplify DSP [3-20](#)

- fixed-point numbers
  - importance of binary point in scaling [8-49](#)

- floating-point
  - comparing to fixed-point results (tutorial) [2-13](#)

- floating-point algorithm
  - comparing to fixed-point [4-42](#)

- folding

- black boxes [4-27](#)
  - optimizing with [4-55](#)
  - smart black boxes [4-27](#)
  - tutorial [2-34](#)

- folding option [8-256](#)

- fraction length
  - converting from Simulink to Synplify DSP [9-5](#)

- frame, defined [3-26](#)

- frequency domain analysis (tutorial) [2-13](#)

- functions

- Synplify DSP [9-1](#)

---

## G

Gain block, Synplify DSP 8-138  
  parameters 8-139  
global resets  
  Synplify DSP RTL 3-24  
  Synplify DSP testbench 3-25

## H

help 1-16  
help SynDSP command 9-25  
hex format  
  reading ROM data 9-11  
hex records 9-11  
hierarchy  
  creating (tutorial) 4-44  
  viewing 4-46  
histogram  
  M Control demo 9-21

## I

I/O blocks  
  pattern folding 4-57  
icons  
  custom block 5-9  
IIR block, Synplify DSP 8-141  
  parameters 8-143  
implementation clock 3-2  
implementation options 8-260  
implementations  
  creating 2-26, 4-48  
  deleting 8-256  
  naming 8-259  
  Synplify DSP file hierarchy 4-7  
In block, Synplify DSP 8-146  
info SynDSP command 9-25  
input ports 8-174  
input sign values 8-234  
input signals  
  sample rate propagation 3-25  
inputs  
  calculating logarithm, CORDIC Log  
    block 8-63  
  calculating logarithm, Log block 8-152

  decreasing sample rate 8-106  
  delay 8-91  
  increasing sample rate 8-263  
  interleaving 8-33, 8-59  
  permutations 8-33  
  shifting, Shifter block 8-232  
  shuffling 8-33, 8-59  
  square root 8-245  
  square root, CORDIC 8-76

installation directory  
  syndsroot function 9-25  
Integrator block, Synplify DSP 8-147  
interpolation  
  defined 3-26  
interpolators  
  CIC 8-36  
  FIR polyphase 8-135  
Inverter block, Synplify DSP 8-151  
  parameters 8-151  
IP  
  embedding 8-238

## L

latency  
  Counter block 8-78  
  sample rate definition 3-27  
latency, fixed 8-257  
libraries  
  adding custom 5-7  
  creating custom 5-6  
  loading custom 5-24  
  Synplify DSP blocks 8-2  
Log block, Synplify DSP 8-152  
  parameters 8-153  
logarithm  
  CORDIC Log block 8-63  
  Log block 8-152  
logic synthesis  
  ASIC 4-66  
  FPGA 4-65

## M

M built-in functions  
  Synplify DSP support 6-29  
M Control  
  reversing bit order 9-2



- M Control block, Synplify DSP
    - data types [6-7](#)
    - demo [9-21](#)
    - design tips [6-4](#)
    - parameters [8-155](#)
    - ports [6-6](#)
    - specifying M functions [6-2](#)
    - timing [6-6](#)
    - using in a design [6-2](#)
  - M functions
    - creating in Synplify DSP [6-3](#)
    - editing in Synplify DSP [6-3](#)
    - specifying for Synplify DSP M Control [6-2](#)
    - specifying Synplify DSP control logic [8-154](#)
    - Synplify DSP support [6-28](#)
  - M keywords
    - Synplify DSP support [6-28](#)
  - M language
    - Synplify DSP caveats [6-32](#)
    - Synplify DSP support [6-27](#)
    - Synplify DSP unsupported features [6-31](#)
  - M operators
    - Synplify DSP support [6-28](#)
  - M special characters
    - Synplify DSP support [6-28](#)
  - M structures
    - Synplify DSP support [6-28](#)
  - M variables
    - Synplify DSP support [6-28](#)
  - masks
    - custom blocks [5-7](#)
    - subsystem [4-45](#)
  - Math Functions library [8-6](#)
  - MATLAB
    - starting Synplify DSP [4-4](#)
    - starting Synplify DSP (ASIC flow) [1-13](#)
    - starting Synplify DSP FPGA [1-8](#)
    - version for running Synplify DSP [1-3](#)
  - Mealy State Machine block, Synplify DSP [8-154](#), [8-157](#)
    - parameters [8-158](#)
  - memories
    - handling for ASIC targets [4-9](#)
    - RTL directory [4-7](#)
  - Memories library [8-7](#)
  - memory queues [8-118](#)
  - MinMax block, Synplify DSP [8-160](#)
  - ModelSim [4-65](#)
  - Moore State Machine block, Synplify DSP [8-162](#)
    - parameters [8-163](#)
  - Mult block, Synplify DSP [8-164](#)
    - parameters [8-165](#)
  - multichannelization
    - black boxes [4-28](#)
    - optimizing with [4-60](#)
    - smart black boxes [4-28](#)
  - multi-channels option [8-258](#)
  - multiplexers [8-166](#)
    - vector [8-267](#)
  - multipliers [8-164](#)
  - multirate blocks
    - pattern folding [4-57](#)
  - multirate design
    - defined [3-27](#)
  - multi-rate designs
    - clock resets [3-31](#)
  - Mux block
    - using in Synplify DSP [8-268](#)
  - Mux block, Synplify DSP [8-166](#)
    - parameters [8-167](#)
- ## N
- NAND operation
    - Binary Logic block [8-23](#)
  - NCO. *See* DDS block, Synplify DSP.
  - Negate block, Synplify DSP [8-168](#)
    - parameters [8-168](#)
  - noise cancellation
    - demo [9-21](#)
  - NOR operation
    - Binary Logic block [8-23](#)
- ## O
- offset
    - defined for downsample [3-27](#)
    - upsample [3-27](#)
  - opens [8-262](#)
  - optimization [2-26](#)

---

- strategies [2-32](#)
- optimizations
  - DSP design [8-253](#)
  - folding [4-55](#)
  - multichannelization [4-60](#)
  - using retiming [4-53](#)
- OR operation
  - Binary Logic block [8-23](#)
- order of operations [3-21](#)
- Out block, Synplify DSP [8-169](#)
- output files
  - Synplify DSP ASIC [4-7](#)
- output ports [8-177](#)

## P

- parameterized blocks [5-16](#)
- pattern
  - for pattern folding [4-57](#)
- pattern folding
  - description [4-57](#)
  - example [4-58](#)
- performance
  - improving with gate-level retiming [4-54](#)
  - improving with retiming (tutorial) [2-33](#)
- Permutation block, Synplify DSP [8-172](#)
  - parameters [8-173](#)
- phase
  - defined [3-28](#)
- plots [4-42](#)
- polyphase filtering [3-32](#)
- polyphase filters [4-14, 4-17](#)
- Port In block, Synplify DSP [8-174](#)
  - parameters [8-175](#)
- Port Out block, Synplify DSP
  - parameters [8-178](#)
- ports
  - data capture [9-18](#)
  - displaying fixed point data type [3-20](#)
  - M Control blocks [6-6](#)
  - register latency [9-19](#)
- Ports & Subsystems library [8-7](#)
- primitive blocks [5-2](#)
- punctures
  - depuncturing [8-94](#)

## Q

- QAM [16](#)
  - demos [9-21](#)
- quantization [8-49](#)

## R

- Ramp block, Synplify DSP [8-191](#)
  - parameters [8-192](#)
- Random block, Synplify DSP
  - parameters [8-195](#)
- Reed Solomon decoder demo [9-21](#)
- Reed Solomon encoder demo [9-21](#)
- Reed-Solomon coding [8-208](#)
- Reed-Solomon Decoder block
  - block parameters, Synplify DSP [8-203](#)
  - pins [8-201](#)
- Reed-Solomon Decoder block, Synplify DSP [8-200](#)
- Reed-Solomon Encoder
  - block, Synplify DSP
    - parameters [8-209](#)
- Reed-Solomon Encoder block, Synplify DSP [8-207](#)
- registers
  - latency parameters [9-19](#)
- rehash command [5-7](#)
- reloadable FIR [8-213](#)
- resamplers
  - FIR polyphase [8-135](#)
- resampling
  - defined [3-28](#)
- resets
  - global, Synplify DSP [3-22](#)
  - local, Synplify DSP [3-22](#)
- resource sharing
  - pattern folding [4-57](#)
- retiming
  - black boxes [4-27](#)
  - gate-level (tutorial) [4-54](#)
  - optimizing with [4-53](#)
  - smart black boxes [4-27](#)
  - tutorial [2-33](#)
- retiming option [8-257](#)
- RFIR block, Synplify DSP [8-213](#)

- parameters [8-215](#)
- ROM block, Synplify DSP [8-219](#)
  - parameters [8-220](#)
- ROM data
  - syn\_read\_hex function [9-11](#)
- ROMs [8-219](#)
  - specifying values [8-220](#)
- rotators [8-67](#)
- RTL
  - black boxes [4-27](#)
  - global resets, Synplify DSP [3-24](#)
  - procedure for generating (ASIC) [1-14](#)
  - procedure for generating (FPGA) [1-9](#)
  - requirements for generating [1-7](#)
  - smart black boxes [4-35](#)
  - verifying (tutorial) [2-29](#)
- RTL code
  - generation [8-253](#)
- RTL Compiler
  - do file [4-9](#), [4-10](#)

## S

- sample period
  - defined [3-28](#)
- sample rate [3-25](#)
  - decreasing input rate [8-106](#)
  - defined [3-28](#)
  - determining [3-2](#)
  - increasing input rate [8-263](#)
- saturation
  - block support summary [A-2](#)
- scripts
  - Aldec simulator [4-65](#)
  - Cadence NC simulator [4-65](#)
  - Modelsim simulator [4-65](#)
- Sequence block, Synplify DSP [8-222](#)
  - parameters [8-223](#)
- Shift Register block, Synplify DSP [8-227](#)
- Shifter block, Synplify DSP [8-232](#)
  - parameters [8-233](#)
- Sign block, Synplify DSP [8-234](#)
- signal clocks [3-2](#)
- Signal Operations library [8-8](#)
- simulations
  - comparing with plots [4-42](#)
- Simulink
  - simulating Synplify DSP ASIC design [1-13](#)
  - simulating Synplify DSP FPGA design [1-8](#)
  - version [1-3](#)
- Simulink configuration
  - checking for Synplify DSP [9-6](#)
  - setting for Synplify DSP [9-17](#)
- SinCos block, Synplify DSP [8-236](#)
  - parameters [8-237](#)
- sine
  - CORDIC SinCos block [8-74](#)
  - SinCos block [8-236](#)
- single-rate designs
  - clock resets [3-30](#)
- slot
  - defined [3-28](#)
- smart black box
  - demo [9-21](#)
- Smart Black Box block, Synplify DSP [8-238](#)
  - parameters [8-240](#)
- smart black boxes
  - folding [4-27](#)
  - retiming [4-27](#)
  - RTL [4-35](#)
- Sobel filtering
  - demo [9-21](#)
- software description [1-2](#)
- sources
  - constant value [8-45](#)
- Sources library [8-9](#)
- Sqrt block, Synplify DSP [8-245](#)
  - parameters [8-246](#)
- square root
  - CORDIC Sqrt block [8-76](#)
  - Sqrt block [8-245](#)
- state machines
  - Mealy [8-157](#)
  - Moore [8-162](#)
- Subsystem block, Synplify DSP [8-248](#)
- subsystems
  - custom blocks [5-7](#)
  - in port, Synplify DSP [8-146](#)
  - out port, Synplify DSP [8-169](#)
  - Synplify DSP [8-248](#)

---

- variable values (tutorial) [4-45](#)
- viewing hierarchy [4-46](#)
- subtractor
  - Synplify DSP block [8-19](#)
- symbol ordering
  - block de-interleaving [8-31](#)
  - block interleaving [8-33](#)
- syn\_bitrev function [9-2](#)
- syn\_get\_coefs
  - using for FIRs [4-16](#)
  - using for IIRs [4-20](#)
- syn\_get\_coefs function [9-4](#)
- syn\_get\_datatype function [9-5](#)
- syn\_get\_dspstartup function [9-6](#)
- syn\_get\_wl function [9-7](#)
- syn\_get\_wordlength function [9-9](#)
- syn\_inh\_width variable [8-274](#)
- syn\_read\_hex function [9-11](#)
  - using [4-43](#)
- syn\_set\_ate function [9-13](#)
- syn\_set\_atm function [9-15](#)
- syn\_set\_dspstartup function [9-17](#)
- syn\_set\_portcapture function [9-18](#)
- syn\_set\_portregister function [9-19](#)
- syn\_unlink function [9-20](#)
- synchronous resets
  - RTL code, Synplify DSP [3-24](#)
  - testbenches, Synplify DSP [3-25](#)
- synchronous resets, Synplify DSP [3-23](#)
- SynCoSimTool block, Synplify DSP [8-249](#)
- SynCoSimTool, Synplify DSP
  - parameters [8-250](#)
- syndspclib.mdl custom library [5-6](#)
- syndspdemo function [9-21](#)
- syndspdoc function [9-22](#)
- syndsplib function [9-23](#)
- syndsproot function [9-25](#)
- SynDSPTool block, Synplify DSP [8-253](#)
  - parameters [8-254](#)
- syndsptool function [9-26](#)
- SynDSPTool toolbox
  - using [4-48](#)
- syndspver function [9-27](#)

- SynFixPtTool block [8-261](#)
- SynFxFtTool
  - setting options [4-38](#)
  - validating algorithms [4-40](#)
- synopsys\_dc.setup file [4-10](#)
- Synplicity
  - data type implementation [3-20](#)
- Synplify DSP
  - accessing information from the
    - command line [9-25](#)
  - ASIC design flow [1-13](#)
  - ASIC logic synthesis [4-9](#)
  - ASIC output [4-66](#)
  - ASIC output files [4-7](#)
  - checking Simulink configuration [9-6](#)
  - FPGA design flow [1-5, 1-8](#)
  - FPGA output [4-65](#)
  - functions [9-1](#)
  - M language support [6-27](#)
  - setting Simulink configuration [9-17](#)
  - starting from MATLAB [4-4](#)
  - starting from MATLAB (ASIC flow) [1-13](#)
  - starting from MATLAB (FPGA) [1-8](#)
  - starting from Simulink [1-8, 4-4](#)

- Synplify DSP blockset
  - accessing [4-5](#)
- Synplify DSP demos [9-21](#)
- Synplify DSP Extract block [8-109](#)
- Synplify DSP FDATool block
  - FIR coefficients [4-15](#)
  - IIR coefficients [4-19](#)
- Synplify DSP FIR block
  - decimators [4-14](#)
- Synplify DSP output [4-62](#)
- Synplify DSP Recast block [8-196](#)
- Synplify DSP Register block [8-211](#)
- Synplify DSP SynFxFtTool block
  - using [4-38](#)

- Synplify DSP toolbox
  - syndsptool function [9-26](#)
- Synplify Pro
  - synthesizing (tutorial) [2-29](#)
- Synplify Pro version [1-3](#)

## T

- target technology [8-260](#)

Tcl output files  
     Synplify DSP ASIC [4-10](#)  
 test benches  
     generating [4-63, 8-259](#)  
     simulating [4-65](#)  
 testbenches  
     global reset, Synplify DSP [3-25](#)  
 third-party IP, embedding [8-238](#)  
 time 0  
     defined [3-29](#)  
 time domain analysis, tutorial [2-12](#)  
 time0  
     determining [3-29](#)  
     single-rate designs [3-31](#)  
 timing engine configuration [9-15](#)  
 timing modes [4-51](#)  
 toolboxes  
     FDATool block, Synplify DSP [8-111](#)  
     SynCoSimTool, Synplify DSP [8-249](#)  
     SynDSPTool, Synplify DSP [8-253](#)  
     SynFixPtTool, Synplify DSP [8-261](#)  
 Transforms library [8-10](#)  
 tutorial [2-2](#)

## U

upsample  
     offset [3-27](#)  
 Upsample block, Synplify DSP  
     parameters [8-265](#)  
 Upsampler block, Synplify DSP [8-263](#)

## V

Vector Concat block, Synplify DSP [8-267](#)  
 Vector Expand block, Synplify DSP [8-273](#)  
     parameters [8-273](#)  
 Vector Extract block, Synplify DSP [8-275](#)  
     parameters [8-275](#)  
 Vector Mux block, Synplify DSP  
     parameters [8-269](#)  
 Vector Split block, Synplify DSP [8-277](#)  
     parameters [8-278](#)  
 vectors  
     block support summary [A-2](#)  
     working with [4-21](#)

verification [2-29](#)  
     generating testbenches from Synplify DSP [4-63](#)  
 Verilog  
     asynchronous resets, Synplify DSP [3-24](#)  
     considerations for generating [4-51](#)  
     generating [4-50](#)  
     keywords [4-51](#)  
     naming rules [4-51](#)  
     synchronous resets, Synplify DSP [3-24](#)  
     test benches [4-63](#)  
     Verilog 2001 [4-51](#)  
 version  
     syndspver function [9-27](#)  
 VHDL  
     asynchronous resets, Synplify DSP [3-24](#)  
     considerations for generating [4-51](#)  
     generating [4-50](#)  
     keywords [4-51](#)  
     naming rules [4-51](#)  
     synchronous resets, Synplify DSP [3-24](#)  
 VHDL test benches [4-63](#)  
 Viterbi Decoder block, Synplify DSP [8-279](#)  
 Viterbi Decoder, Synplify DSP  
     demo [9-21](#)  
     demo with puncturing [9-21](#)

## W

word length  
     block support summary [A-2](#)  
     calculating with syn\_get\_wl [9-7](#)  
     calculating with  
         syn\_get\_wordlength [9-9](#)  
     converting from Simulink to Synplify DSP [9-5](#)  
     syn\_get\_wl function [9-7](#)  
     syn\_get\_wordlength function [9-9](#)  
 word size  
     converting input word size [8-48](#)

## X

XNOR operation  
     Binary Logic block [8-23](#)  
 XOR operation  
     Binary Logic block [8-23](#)

