



DSP Builder Handbook

Volume 3: DSP Builder Advanced Blockset



101 Innovation Drive
San Jose, CA 95134
www.altera.com

HB_DSPB_ADV-8.0

Document last updated for Altera Complete Design Suite version:
Document publication date:

14.0
June 2014



© 2014 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



Section I. DSP Builder Advanced Blockset User Guide

Chapter 1. About the DSP Builder Advanced Blockset

Features	1-1
Architecture versus Implementation	1-1
Libraries	1-2
Base Library	1-2
FFT Library	1-2
ModelBus Library	1-3
ModelPrim Library	1-3
ModelIP Libraries	1-3
Cycle Accuracy and Latency	1-4
Sample Rate and Clocks	1-4
Interoperability with the Standard Blockset	1-5
Typical Design Blocks	1-5
Signals Block	1-6
Control Block	1-7
Device Block	1-7
SynthesisInfo Block	1-8
Other Blocks	1-8

Chapter 2. Design Flow

Implementing your Design in DSP Builder Advanced Blockset	2-2
Staging your Design into Subsystems	2-2
ModelPrim Blocks	2-3
Specifying the Output Data Type	2-3
SynthesisInfo Block	2-4
Loops	2-4
Vectors	2-6
ModelPrim Blocks in NonModelPrim Subsystems	2-7
Interfaces as Subsystem Boundaries	2-7
Interfaces as Scheduling Boundaries	2-8
Block Interfaces	2-8
Interface Example	2-10
Connecting Subsystems	2-11
Building Multichannel Systems	2-16
Multichannel Systems	2-16
Channel, Data, and Valid Examples	2-17
Vectorized Inputs	2-19
Verifying your Design in Simulink and MATLAB	2-19
Testbenches	2-19
Using References	2-20
Setting Up Stimulus	2-20
Analyzing your Design	2-20
Exploring Design Tradeoffs	2-20
Bit Growth	2-20
Managing Bit Growth	2-21
Using Rounding and Saturation	2-21

Scaling using ModelPrim Blocks	2-21
Using Convert Blocks and Specifying Output Types	2-22
The Convert Block and Real-world Values	2-23
Output Data Types on ModelPrim Blocks	2-24
Verifying your Design and Debugging	2-26
Automatic Testbench	2-27
ModelSim Simulations	2-27
Integrating into Hardware	2-27
Directory Structure	2-28
Generated Files	2-28
Automatic Testbench Files	2-29
DSP Builder Designs and the Quartus II Project	2-30
Adding a DSP Builder Advanced Blockset Design to an Existing Quartus II Project	2-30
Adding Advanced Blockset Components to Qsys	2-30
Guidelines	2-30
Essential Guidelines for the Top-Level Design	2-31
Essential Guidelines for the ModelPrim Blocks	2-31
Simulink Setup Guidelines	2-31
Recommended Guidelines for the Top-Level Design	2-32
Recommended Guidelines for ModelPrim Subsystems	2-32
Guidelines for Verifying a Design	2-33

Chapter 3. Getting Started

Setting Up Simulink	3-1
Design Examples and Reference Designs	3-1
Using the DSP Builder Menu in Simulink	3-1
Running Automatic Testbenches	3-3
The dspba.runModelsimATB Command Syntax	3-4
Running All Automatic Testbenches	3-4
The command run_all_atbs Command Syntax	3-4
Testbench Error Messages	3-5

Chapter 4. ModelIP Tutorial

Creating a ModelIP Design	4-1
Simulating the Design	4-1
Verifying the Design	4-2
Compiling the Design	4-2
Reparameterizing the FIR Filter	4-3
Doubling the Target Clock Rate	4-3
Retargeting Design to Different Device Family	4-3

Chapter 5. ModelPrim Tutorial

The Fibonacci Sequence	5-1
Creating the Fibonacci Model	5-1
Creating a New Model	5-1
Creating a ModelIP Design	5-1
Adding Blocks from the ModelPrim Library	5-2
Completing the Top-Level Model	5-3
Simulating the Design in Simulink	5-4
Using Vector Types	5-5
Exploring the Generated Files	5-6
Simulating the RTL	5-7
Compiling with the Quartus II Software	5-7

Chapter 6. Design Examples and Reference Designs

Opening a Design Example	6-1
Copying a Design Example	6-2
Running a Design Example	6-2
Base Blocks	6-3
Scale	6-3
Local Threshold	6-3
FFT	6-3
1K Floating-Point FFT	6-4
FFT	6-5
FFT without BitReverseCoreC Block	6-5
IFFT	6-5
IFFT without BitReverseCoreC Block	6-6
Floating-Point FFT	6-6
Floating-Point FFT without BitReverseCoreC Block	6-6
Floating-Point iFFT	6-7
Floating-Point iFFT without BitReverseCoreC Block	6-7
Multichannel FFT	6-7
Multiwire Transpose	6-7
Parallel FFT	6-7
Parallel Floating-Point FFT	6-7
Radix 2 Streaming FFT	6-7
Radix 4 Streaming FFT	6-8
Single-Wire Transpose	6-8
Switchable FFT/iFFT	6-9
Variable-Size Fixed-Point FFT	6-9
Variable-Size Fixed-Point FFT without BitReverseCoreC Block	6-9
Variable-Size Fixed-Point iFFT	6-9
Variable-Size Fixed-Point iFFT without BitReverseCoreC Block	6-9
Variable-Size Floating-Point FFT	6-9
Variable-Size Floating-Point FFT without BitReverseCoreC Block	6-9
Variable-Size Floating-Point iFFT	6-9
Variable-Size Floating-Point iFFT without BitReverseCoreC Block	6-10
Variable-Size Low-Resource FFT	6-10
Variable-Size Low-Resource Real-Time FFT	6-10
Filters	6-11
Complex FIR Filter	6-11
Decimating CIC Filter	6-11
Decimating FIR Filter	6-12
Filter Chain with Forward Flow Control	6-12
FIR Filter with Exposed Bus	6-12
Fractional FIR Filter Chain	6-12
Fractional-Rate FIR Filter	6-13
Half-Band FIR Filter	6-13
Interpolating CIC Filter	6-13
Interpolating FIR Filter	6-14
Interpolating FIR Filter with Multiple Coefficient Banks	6-14
Interpolating FIR Filter with Updating Coefficient Banks	6-15
Root-Raised Cosine FIR Filter	6-15
Single-Rate FIR Filter	6-15
Super-Sample Decimating FIR Filter	6-16
Super-Sample Fractional FIR Filter	6-16
Super-Sample Interpolating FIR Filter	6-16
Variable-Rate CIC Filter	6-17

Folding	6-17
Position, Speed, and Current Control for AC Motors	6-17
About FOC	6-18
Functional Description	6-18
Resource Usage	6-19
Hardware Generation	6-19
Input Position Request	6-20
Output Response for Speed and Torque	6-21
Output Current	6-22
Position, Speed, and Current Control for AC Motors (with ALU Folding)	6-22
Folded FIR Filter	6-23
Floating Point	6-23
Black-Scholes Floating Point	6-24
Double-Precision Real Floating-Point Matrix Multiply	6-24
Fine Doppler Estimator	6-24
Floating-Point Mandelbrot Set	6-24
General Real Matrix Multiply One Cycle Per Output	6-27
Newton Root Finding Tutorial Step 1—Iteration	6-27
Newton Root Finding Tutorial Step 2—Convergence	6-27
Newton Root Finding Tutorial Step 3—Valid	6-27
Newton Root Finding Tutorial Step 4—Control	6-27
Newton Root Finding Tutorial Step 5—Final	6-28
Normalizer	6-28
Single-Precision Complex Floating-Point Matrix Multiply	6-28
Single-Precision Real Floating-Point Matrix Multiply	6-29
Simple Nonadaptive 2D Beamformer	6-29
Flow Control	6-30
Avalon-ST Interface (Input and Output FIFO Buffer) with Backpressure	6-30
Avalon-ST Interface (Output FIFO Buffer) with Backpressure	6-30
Kronecker Tensor Product	6-30
Parallel Loops	6-31
Primitive FIR with Back Pressure	6-31
Primitive FIR with Forward Pressure	6-32
Primitive Systolic FIR with Forward Flow Control	6-33
Rectangular Nested Loop	6-33
Sequential Loops	6-34
Triangular Nested Loop	6-34
Host Interface	6-34
Memory-Mapped Registers	6-34
Platforms	6-35
16-Channel DDC	6-35
16-Channel DUC	6-35
2-Antenna DUC for WiMAX	6-36
2-Channel DUC	6-36
Super-Sample Rate Digital Upconverter	6-37
Primitive Blocks	6-37
8×8 Inverse Discrete Cosine Transform	6-38
Automatic Gain Control	6-38
Bit Combine for Boolean Vectors	6-39
Bit Extract for Boolean Vectors	6-39
Color Space Converter	6-39
CORDIC from Primitive Blocks	6-40
Digital Predistortion Forward Path	6-40
Fibonacci Series	6-40

Folded Vector Sort	6-41
Fractional Square Root Using CORDIC	6-41
Fixed-point Maths Functions	6-41
Hello World	6-42
Hybrid Direct Form and Transpose Form FIR Filter	6-42
Loadable Counter	6-42
Matrix Initialization of LUT	6-42
Matrix Initialization of Vector Memories	6-43
Multichannel IIR Filter	6-44
Quadrature Amplitude Modulation	6-44
Reinterpret Cast for Bit Packing and Unpacking	6-44
Run-time Configurable Decimating and Interpolating Half-Rate FIR Filter	6-45
Square Root Using CORDIC	6-45
Test CORDIC Functions with the CORDIC Block	6-45
Vector Sort—Sequential	6-46
Vector Sort—Iterative	6-46
Vector Initialization of Sample Delay	6-46
Wide Single-Channel Accumulators	6-47
Reference Designs	6-47
1-Antenna WiMAX DDC	6-48
2-Antenna WiMAX DDC	6-49
1-Antenna WiMAX DUC	6-50
2-Antenna WiMAX DUC	6-50
4-Carrier, 2-Antenna W-CDMA DDC	6-51
1-Carrier, 2-Antenna W-CDMA DDC	6-52
4-Carrier, 2-Antenna W-CDMA DUC	6-52
1-Carrier, 2-Antenna W-CDMA DDC	6-53
4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32 ..	6-53
4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48 ..	6-54
4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40 ..	6-55
Cholesky Solver Single Channel	6-56
Cholesky Solver Multiple Channels	6-57
Crest Factor Reduction	6-57
Direct RF with Synthesizable Testbench	6-57
Dynamic Decimating FIR Filter	6-57
Multichannel QR Decomposition	6-58
QR Decomposition	6-58
Reconfigurable Decimation Filter	6-59
Single-Channel 10-MHz LTE Transmitter	6-59
STAP Radar Forward and Backward Substitution	6-60
STAP Radar Steering Generation	6-61
STAP Radar QR Decomposition 192x204	6-61
Time Delay Beamformer	6-61
Variable Integer Rate Decimation Filter	6-61
Waveform Synthesis	6-62
Complex Mixer	6-62
Four Channel, Two Banks NCO	6-62
Four Channel, Four Banks NCO	6-63
Four Channel, Eight Banks, Two Wires NCO	6-64
Four Channel, 16 Banks NCO	6-65
ModelIP	6-66
NCO	6-66
NCO with Exposed Bus	6-66
Real Mixer	6-66

Chapter 7. DDC Design Example

DDC Design Example Description	7-1
Signals Block	7-2
Control Block	7-2
Memory Maps	7-3
EditParams Blocks	7-3
Source Blocks	7-3
Sink Blocks	7-3
DDCChip Subsystem	7-4
Primary Inputs	7-5
Merge Multiplexer	7-5
NCO	7-5
Mixer	7-6
Mixer Scale Block	7-6
Opening the Design Example	7-7
DecimatingCIC and Scale Blocks	7-8
Decimating FIR Blocks	7-9
Building the Design Example	7-12
Simulating the Design Example in Simulink	7-12
Exploring the Generated Files	7-13

Chapter 8. Floating-Point Data Types

Features	8-1
Supported Floating-Point Data Types and Logic Usage	8-2
Round-Off Errors	8-2
Trading Off Logic Utilization and Accuracy	8-2
Upgrading Pre v14.0 Designs	8-3
Floating-Point Sine Wave Generator Tutorial	8-3
Creating a Sine Wave Generator	8-3
Using Data Type Variables to Parameterize Designs	8-5
Using Data-Type Propagation	8-5
Testbench Verification	8-6
Tuning ATB Thresholds	8-7
Writing Application Specific Verification	8-7
Using Bit-Accurate Simulation	8-9
Adder Trees and Scalar Products	8-9
Creating Floating-Point Accumulators for Designs that Use Iteration	8-10
Newton Root Finding Tutorial	8-11
Implementing the Newton Design	8-12
Improving Floating-Point Designs	8-13

Chapter 9. DSP Builder Standard and Advanced Blockset Interoperability

Using Signal Compiler in an Advanced Blockset Design	9-2
Using HIL in a Standard and Advanced Blockset Design	9-6
Archiving Combined Blockset Designs	9-12
Using HIL in Advanced Blockset Designs	9-12

Chapter 10. Techniques for Experienced Users

Creating a New Model From an Existing Design Example	10-1
Managing your Designs	10-2
Managing Basic Parameters	10-3
Creating User Libraries	10-3
Revision Control	10-4

Pruning and Twiddle for FFT Blocks	10-4
Hardware Verification	10-5
Preparing to Use System-In-The-Loop	10-6
System-In-The-Loop Supported Blocks	10-6
Building Custom Board Support Packages	10-7
Setting up Board Support Package for 28 nm Device Families	10-7
Setting up Board Support Packages for Other Device Families	10-7
Publishing the Package in the System-In-The-Loop Wizard	10-8
Third-Party Board Support Packages	10-8
Template Values in the boardinfos.xml File	10-9
Running System-In-the-Loop	10-9
System-In-The-Loop Parameters	10-10
Using Scripts	10-12
Evaluating Setup Scripts	10-12
Visualization Features	10-12
Example Custom Scripts	10-13
Folding	10-15
ALU Folding	10-15
Data Types	10-15
Inputs on Add and Mux Blocks	10-16
Report File and The Resources in a Folded Design	10-16
Simulation Rate	10-17
Using ALU Folding	10-17
Using Automated Verification	10-17
Ready Signal	10-18
Connecting the Ready Signal	10-19
Removing Resource Sharing Folding	10-19
How to Manage Latency	10-20
Reading Latency	10-20
Using Latency Constraints	10-20
Zero Latency Example	10-23
Nonexplicit Delays	10-23
Distributed Delays	10-24
Latency and f_{MAX} Constraint Conflicts	10-26
Control Units Delays	10-27
Flow Control	10-29
Flow Control Using Latches	10-29
Forward Flow Control Using Latches	10-29
Flow Control Using FIFO Buffers	10-29
Flow Control and Backpressure Using FIFO Buffers	10-30
Flow Control using Simple Loop	10-31
Flow Control Using the ForLoop Block	10-31
Interfacing with a Processor Bus	10-31
Base Addresses	10-31
Integrating with Qsys	10-32
System Components with Avalon-ST Interface Blocks	10-32
Extending the Interface Definition	10-33
Adding More Ports to the Avalon-ST Blocks	10-33
Adding Custom Text	10-34
Restrictions	10-34
Updating Registers with the Nios II Processor	10-34
Pruning and Twiddle Specification for FFTs	10-35

Chapter 11. Troubleshooting

Common Problems	11-1
Timed Feedback Loops	11-1
Loops, Clock Cycles, and Data Cycles	11-1
ModelPrim Subsystem Designs to Avoid	11-2

Section II. DSP Builder Advanced Blockset Libraries

Chapter 12. Additional Library

Avalon-ST Output (AStOutput)	12-2
Avalon-ST Input (AStInput)	12-3
Avalon-ST Input FIFO Buffer (AStInputFIFO)	12-3
Nested Loops (NestedLoop1, NestedLoop2, NestedLoop3)	12-3
Operation	12-4
NestedLoop Block Port Interface	12-5
Anchored Delay	12-6
Zero-Latency Latch (latch_0L)	12-6
Single-Cycle Latency Latch (latch_1L)	12-6
Reset-Priority Latch (SRLatch_PS)	12-6
Set-Priority Latch (SRLatch)	12-7
Complex to Real-Imag	12-7
Enabled Delay Line	12-7
Enabled Feedback Delay	12-7
Expand Scalar (ExpandScalar)	12-7
Tapped Delay Line (TappedDelayLine)	12-8
Vector Multiplexer (VectorMux)	12-8

Chapter 13. Base Library

Avalon-MM Slave Settings (AvalonMMSlaveSettings)	13-1
Parameters	13-1
Channel Viewer (ChanView)	13-2
Parameters	13-2
Port Interface	13-2
Updated Help	13-3
Control	13-3
Parameters	13-3
Hardware Generation	13-4
Memory-Mapped Bus Interface	13-4
Memory and Multiplier Trade-Off Options	13-4
Updated Help	13-6
Device	13-6
Edit Params	13-7
Usage	13-11
Enhanced Precision Support	13-11
LocalThreshold	13-11
Parameters	13-11
Run ModelSim	13-12
Usage	13-12
Run Quartus II	13-12
Usage	13-12
Scale	13-13
Parameters	13-14
Port Interface	13-14
Updated Help	13-15

Signals	13-15
Parameters	13-15
Updated Help	13-16

Chapter 14. FFT Library

Bit Reverse Core C (BitReverseCoreC and VariableBitReverse)	14-2
Parameters	14-2
Port Interface	14-3
Bit Vector Combine (BitVectorCombine)	14-3
Parameters	14-3
Port Interface	14-4
Butterfly Unit (BFU)	14-4
Parameters	14-4
Port Interface	14-5
Butterfly I C (BFIC) (Deprecated)	14-5
Parameters	14-5
Port Interface	14-5
Butterfly II C (BFIIC) (Deprecated)	14-6
Parameters	14-6
Port Interface	14-6
Choose Bits (ChooseBits)	14-6
Parameters	14-7
Port Interface	14-7
Crossover Switch (XSwitch)	14-7
Port Interface	14-7
Dual Twiddle Memory (DualTwiddleMemoryC)	14-8
Parameters	14-8
Port Interface	14-8
Edge Detect (EdgeDetect)	14-8
Parameters	14-8
Port Interface	14-9
FFT (FFT, FFT_Light, VFFT, VFFT_Light)	14-9
FFT	14-9
VFFT	14-10
Parameters	14-11
Port Interface	14-12
Floating-Point Twiddle Generator (TwiddleGenF) (Deprecated)	14-13
Parameters	14-13
Port Interface	14-13
Fully-Parallel FFTs (FFT2P, FFT4P, FFT8P, FFT16P, FFT32P, and FFT64P)	14-13
Parameters	14-13
Ports	14-14
Fully-Parallel FFTs with Flexible Ordering (FFT2X, FFT4X, FFT8X, FFT16X, FFT32X, and FFT64X)	14-14
Parameters	14-14
Ports	14-14
General Multitwiddle and General Twiddle (GeneralMultiTwiddle, GeneralTwiddle)	14-15
Parameters	14-15
Ports	14-16
Hybrid FFT (Hybrid_FFT)	14-16
Parameters	14-16
Port Interface	14-17
Multiwire Transpose (MultiwireTranspose)	14-17
Parameters	14-17

Port Interface	14-17
Parallel Pipelined FFT (PFFT_Pipe)	14-18
Parameters	14-18
Port Interface	14-18
Pulse Divider (PulseDivider)	14-18
Parameters	14-19
Port Interface	14-19
Pulse Multiplier (PulseMultiplier)	14-19
Parameters	14-19
Port Interface	14-19
Single-Wire Transpose (Transpose)	14-19
Parameters	14-20
Port Interface	14-20
Split Scalar (SplitScalar)	14-20
Parameters	14-20
Port Interface	14-20
Streaming FFTs (FFT2, FFT4, VFFT2, and VFFT4)	14-21
Parameters	14-21
Ports	14-21
Twiddle Angle (TwiddleAngle)	14-21
Parameters	14-22
Port Interface	14-22
Twiddle Generator (TwiddleGenC) Deprecated	14-22
Parameters	14-22
Port Interface	14-23
Twiddle and Variable Twiddle (Twiddle and VTwiddle)	14-23
Parameters	14-23
Ports	14-24
Twiddle ROM (TwiddleRom, TwiddleMultRom and TwiddleRomF (deprecated))	14-24
Parameters	14-25
Port Interface	14-25

Chapter 15. Filter Library

FIR and CIC Filters	15-1
Common Features	15-1
Updated Help	15-2
Basic FIR Filter Operation	15-3
Half-Band and L-Band Nyquist FIR Filters	15-3
Automatic Pipelining	15-3
High Speed Operation	15-4
Scalability	15-4
Parameterization	15-4
Coefficient Generation	15-4
Channelization	15-4
Decimating CIC	15-5
Features	15-5
Operation	15-5
Parameters	15-6
Port Interface	15-6
Decimating FIR	15-6
Features	15-7
Operation	15-7
Parameters	15-8
Filter Coefficients	15-8

Port Interface	15-9
Fractional Rate FIR	15-9
Features	15-9
Operation	15-10
Parameters	15-11
Filter Coefficients	15-11
Port Interface	15-12
Interpolating CIC	15-12
Features	15-12
Operation	15-13
Parameters	15-13
Port Interface	15-13
Interpolating FIR	15-14
Features	15-14
Operation	15-14
Parameters	15-15
Filter Coefficients	15-16
Port Interface	15-16
Single-Rate FIR	15-17
Features	15-17
Operation	15-17
Parameters	15-17
Filter Coefficients	15-18
Port Interface	15-18

Chapter 16. ModelBus Library

Bus Slave (BusSlave)	16-1
Parameters	16-2
Port Interface	16-2
Bus Stimulus (BusStimulus)	16-2
Parameters	16-3
Port Interface	16-3
Bus Stimulus File Reader (Bus StimulusFileReader)	16-3
Parameters	16-4
Port Interface	16-5
Register Bit (RegBit)	16-5
Parameters	16-5
Port Interface	16-6
Register Field (RegField)	16-6
Parameters	16-6
Port Interface	16-7
Register Out (RegOut)	16-7
Parameters	16-7
Port Interface	16-7
Shared Memory (SharedMem)	16-7
Parameters	16-8
Port Interface	16-8

Chapter 17. ModelPrim Library

Vector and Complex Type Support	17-3
Vector Type Support	17-3
Element by Element Mode	17-3
Mathematical Vector Mode	17-4

Interactions with Simulink	17-4
Complex Support	17-4
Restrictions	17-4
Interactions with Simulink	17-4
Absolute Value (Abs)	17-5
Parameters	17-5
Port Interface	17-6
Accumulator (Acc)	17-6
Parameters	17-6
Port Interface	17-7
Add	17-8
Parameters	17-8
Port Interface	17-9
Add SLoad (AddSLoad)	17-9
Parameters	17-9
Port Interface	17-10
AddSub	17-10
Parameters	17-10
Port Interface	17-10
AddSubFused	17-10
Parameters	17-10
Port Interface	17-11
AND Gate (And)	17-11
Parameters	17-11
Port Interface	17-12
Bit Combine (BitCombine)	17-12
Parameters	17-12
Port Interface	17-13
Bit Extract (BitExtract)	17-13
Parameters	17-13
Port Interface	17-14
Bit Reverse (BitReverse)	17-14
Parameters	17-14
Port Interface	17-14
Channel In (ChannelIn)	17-14
Parameters	17-15
Port Interface	17-15
Channel Out (ChannelOut)	17-15
Parameters	17-16
Port Interface	17-16
Compare (CmpCtrl)	17-16
Parameters	17-16
Port Interface	17-17
Complex Conjugate (ComplexConjugate)	17-17
Parameters	17-17
Port Interface	17-18
Compare Equality (CmpEQ)	17-18
Parameters	17-18
Port Interface	17-18
Compare Greater Than (CmpGE)	17-18
Parameters	17-18
Port Interface	17-19
Compare Less Than (CmpLT)	17-19
Parameters	17-19

Port Interface	17-19
Compare Not Equal (CmpNE)	17-19
Parameters	17-19
Port Interface	17-20
Constant (Const)	17-20
Parameters	17-20
Port Interface	17-21
Constant Multiply (Const Mult)	17-21
Parameters	17-21
Port Interface	17-21
Convert	17-22
Parameters	17-22
Port Interface	17-23
CORDIC	17-23
Parameters	17-25
Port Interface	17-25
Counter	17-25
Parameters	17-26
Port Interface	17-26
Count Leading Zeros, Ones, or Sign Bits (CLZ)	17-27
Parameters	17-27
Port Interface	17-27
Dual Memory (DualMem)	17-27
Parameters	17-28
Port Interface	17-29
Demultiplexer (Demux)	17-29
Parameters	17-30
Divide	17-30
Parameters	17-30
Port Interface	17-31
FIFO	17-31
Parameters	17-31
Port Interface	17-32
ForLoop	17-32
Port Interface	17-32
General Purpose Input (GPIn)	17-34
Parameters	17-34
Port Interface	17-34
General Purpose Output (GPOut)	17-35
Parameters	17-35
Port Interface	17-35
Load Exponent (LdExp)	17-35
Parameters	17-35
Port Interface	17-36
Left Shift (LShift)	17-36
Parameters	17-36
Port Interface	17-37
Loadable Counter (LoadableCounter)	17-37
Look-Up Table (Lut)	17-38
Parameters	17-38
Port Interface	17-39
Loop	17-39
Port Interface	17-40
Math	17-40

Parameters	17-41
Port Interface	17-41
Minimum and Maximum (MinMax)	17-41
Parameters	17-42
Port Interface	17-42
MinMaxCtrl	17-42
Parameters	17-42
Port Interface	17-43
Multiply (Mult)	17-43
Parameters	17-44
Port Interface	17-44
Multiplexer (Mux)	17-44
Parameters	17-45
Port Interface	17-45
NAND Gate (Nand)	17-45
Parameters	17-46
Port Interface	17-46
Negate	17-46
Parameters	17-46
Port Interface	17-47
NOR Gate (Nor)	17-47
Parameters	17-47
Port Interface	17-48
NOT Gate (Not)	17-48
Parameters	17-48
Port Interface	17-48
OR Gate (Or)	17-49
Parameters	17-49
Port Interface	17-49
Polynomial	17-49
Parameters	17-50
Port Interface	17-50
Ready	17-50
Reinterpret Cast (ReinterpretCast)	17-50
Parameters	17-51
Port Interface	17-51
Round	17-51
Parameters	17-52
Port Interface	17-52
Sample Delay (SampleDelay)	17-52
Parameters	17-53
Port Interface	17-53
Scalar Product	17-53
Parameters	17-54
Port Interface	17-54
Performance Tradeoffs	17-54
Select	17-56
Parameters	17-57
Port Interface	17-57
Sequence	17-57
Parameters	17-58
Port Interface	17-58
Shift	17-58
Parameters	17-59

Port Interface	17-59
Sqrt	17-59
Parameters	17-59
Port Interface	17-60
Subtract (Sub)	17-60
Parameters	17-60
Port Interface	17-61
Synthesis Information (SynthesisInfo)	17-61
Parameters	17-62
Port Interface	17-62
Updated Help	17-62
Trig	17-63
Parameters	17-63
Port Interface	17-64
XNOR Gate (Xnor)	17-64
Parameters	17-64
Port Interface	17-65
XOR Gate (Xor)	17-65
Parameters	17-65
Port Interface	17-65

Chapter 18. ModelVectorPrim Library

Sum of Elements (SumOfElements)	18-1
Parameters	18-1
Port Interface	18-2

Chapter 19. Waveform Synthesis Library

Complex Mixer (ComplexMixer)	19-1
Parameters	19-2
Port Interface	19-2
Real Mixer (Mixer)	19-3
Parameters	19-3
Port Interface	19-4
NCO	19-5
Parameters	19-5
Phase Increment and Inversion	19-6
Phase Increment Memory Registers	19-7
Frequency Hopping	19-8
Results Tab	19-9
Port Interface	19-9

Chapter 20. Categorized Block List

Additional	20-1
Base	20-1
FFT	20-2
Filter	20-2
ModelBus	20-3
ModelPrim	20-3
ModelVectorPrim	20-5
Waveform Synthesis	20-5

Additional Information

Document Revision History	Info-2
---------------------------------	--------

How to Contact Altera	Info-10
Typographic Conventions	Info-10



Section I. DSP Builder Advanced Blockset User Guide

DSP Builder advanced blockset consists of several Simulink libraries that allow you to implement DSP designs quickly and easily. DSP Builder is a high-level synthesis technology that optimizes the high-level, untimed netlist into low level, pipelined hardware for your target Altera[®] FPGA device and desired clock rate. DSP Builder implements the hardware as VHDL with scripts that integrate with the Quartus II software and the ModelSim simulator.

You can create designs without needing detailed device knowledge and generate designs that run on a variety of FPGA families with different hardware architectures. DSP Builder allows you to manually describe algorithmic functions and apply rule-based methods to generate hardware optimized code. The advanced blockset is particularly suited for streaming algorithms characterized by continuous data streams and occasional control. For example, RF card designs that comprise long filter chains.

After specifying the desired clock frequency, number of channels, and other top-level design constraints, DSP Builder pipelines the generated RTL to achieve timing closure. By analyzing the system-level constraints, DSP Builder also optimizes folding to balance latency versus resources, with no need for manual RTL editing.

DSP Builder advanced blockset includes its own timing-driven IP blocks that can generate high performance FIR, CIC, and NCO models.

Features

The advanced blockset has the following features:

- Automatic pipelining to enable timing closure
- Automatic folding
- Automatic resource sharing
- Easy to compare and target different device families
- High-performance floating-point designs
- Wizard-based interface (system-in-the-loop) to configure, generate, and run hardware verification system.

Architecture versus Implementation

DSP Builder allows you to focus on architecture rather than implementation. You can concentrate on algorithms rather than FPGA resources.

Generally, FPGA design is highly implementation focused:

- You exploit multipliers based on the available features.
- You map shift registers to memories and balance overall device resources by the chosen implementation to make sure you do not force a design into a larger device based solely on suboptimal memory or multiplier implementations.

- State machines match pipeline latencies of memory or multiplier structures, and you carefully place routing registers in key structures to account for routing delays to and from various hard blocks that complicate FPGA placement.
- Floor planning often allows you to reduce compilation times and help fit a marginal design because board prototypes are already finalized.

DSP Builder separates the algorithm from these tedious rules-based implementations. This design process gives new opportunities for system-level design. Thus, DSP Builder focuses on the design architecture and text-based entry where you describe the design as naturally as possible.

DSP-based designs have many tradeoffs: bit precision, and memory efficiency, for example, but the biggest tradeoff is the algorithm. How much value does using a more sophisticated channel estimation algorithm give you in system versus how much more it costs in FPGA resources? These important tradeoffs are generally made with incomplete information. DSP Builder allows you to measure these tradeoffs, to see real results in hardware. DSP Builder can yield better than hand-coded results in practice.

DSP Builder saves you time and effort to easily implement a DSP design that has a software verification flow.

Libraries

The advanced blockset comprises several Simulink libraries.



For more information about the advanced blockset libraries, refer to the *DSP Builder Advanced Blockset Libraries* section in volume 3 of the *DSP Builder Handbook*.

Base Library

The base library includes all the necessary top-level design blocks. A design may include any number of subsystems that can combine blocks from the advanced blockset with blocks from the MATLAB Simulink libraries.

It also includes other blocks to control and view the signals in your design, and to automatically load your design into the ModelSim simulator or the Quartus II software.

FFT Library

The FFT library contains common blocks that support fast Fourier transform (FFT) design. It also includes several blocks that support the radix-2² algorithm.



For information about the radix-2² algorithm, refer to *A New Approach to Pipeline FFT Processor* – Shousheng He & Mats Torkleson, Department of Applied Electronics, Lund University, Sweden.

ModelBus Library

The ModelBus library provides memories and registers that you can access in your DSP datapath with an external interface. You can use these blocks to configure coefficients or run-time parameters and to read calculated values. This library also includes blocks that you can use to simulate the bus interface in the Simulink environment.

ModelPrim Library

The ModelPrim library allows you to create fast efficient designs captured in the behavioral domain rather than the implementation domain by combining primitive functions. For example, you can use a `SampleDelay` block and let DSP Builder decide how to implement that delay.

The ModelPrim library contains the following functions to provide support for building hardware functions with the MATLAB fixed-point types:

- Basic operators:
 - Add
 - Multiply
 - Delay
- Memories
- Simple control blocks
- Signal type manipulation functions

You do not need to understand the details of the underlying FPGA architecture, as DSP Builder automatically maps the ModelPrim blocks into efficient FPGA constructs.

ModelIP Libraries

The ModelIP (filters and waveform synthesis) libraries include parameterizable multichannel multirate cycle-accurate filters and waveform synthesis blocks that allow you to quickly create designs for digital front-end applications. Altera provides the ModelIP blocks in the following libraries:

- The Filters library contains several decimating and interpolating cascaded integrator-comb (CIC), and finite impulse response (FIR) filters including single-rate, multirate, and fractional-rate FIR filters.
- The Waveform Synthesis library contains a numerically controlled oscillator (NCO), complex mixers, and real mixers.

The ModelIP library has IP macro blocks that implement the most commonly used functions such as FIR filters and NCOs, ready to be plugged in to your design using drag-and-drop.



You cannot use ModelIP blocks in ModelPrim subsystems.

DSP Builder generates the ModelIP blocks at runtime while running simulations in Simulink. The ModelIP blocks do not have a predetermined architecture. Based on the system clock frequency, you can further optimize the blocks by time sharing the internal resources. All the ModelIP blocks accept system parameters. For example for the interpolating FIR, you can specify system variables as parameters such as input rate per channel, rate change factor, and number of channels. It also accepts MATLAB functions to generate coefficients. You can specify the base address for the memory-mapped interface to reload coefficient.

Cycle Accuracy and Latency

The DSP Builder advanced blockset supports:

- ModelIP block designs, such as the FIR and CIC filters. These blocks are 100% cycle accurate and the Simulink behavior represents exactly the RTL behavior. You can turn a display of the latency added by each block.
- Synthesized ModelPrim block designs from the ModelPrim library. These blocks are 100% cycle accurate at their boundaries, therefore interfacing to other blocks is straightforward.

You can also use ModelPrim blocks outside of synthesizable subsystems, to create glue logic around other subsystems. The Boolean logic and delay blocks are cycle accurate, but other ModelPrim blocks are not.



For more information about latency, refer to [“How to Manage Latency” on page 10–20](#).

Sample Rate and Clocks

DSP Builder advanced blockset creates designs with a single datapath clock domain and an optional second memory bus interface clock domain. Generally, Simulink datapath simulation takes one Simulink simulation step for each FPGA clock cycle.

Altera encourages you to target your design to high clock frequencies. Also, you should use the ModelIP blocks, or build systems that time-division multiplex data, to ensure full utilization of resources.

The sample rate is the rate at which real data is clocked through the system at any point. In a multirate environment, the sample rate may vary along the datapath.

The datapath components in DSP Builder use a single clock rate. An external processor uses a separate bus clock to read and write any ModelPrim registers or shared memories and ModelIP block parameters through an Avalon[®] Memory-Mapped (Avalon-MM) interface.



The memory-mapped input and output registers clear when the system is reset but DSP Builder retains the contents of the dual memory that the FIR Filter, NCO, and ModelBus blocks use.



For information about the Avalon-MM Interface, refer to the [Avalon Interface Specifications](#).

Interoperability with the Standard Blockset

You can use the advanced blockset entirely independently of the DSP Builder Standard Blockset, or you can embed its blocks in top-level DSP Builder designs that use the standard blockset.



DSP Builder Standard Blockset is a legacy product and Altera recommends you do not use it for new designs, except as a wrapper for advanced blockset designs.



For information about interoperability with DSP Builder Standard Blockset, refer to *Chapter 9, DSP Builder Standard and Advanced Blockset Interoperability*.

Typical Design Blocks

A top-level DSP Builder advanced blockset design consists of a Simulink testbench, which provides design inputs and allows you to analyze inputs and outputs, and the following DSP Builder top-level parameterization blocks:

- A `Signals` block to specify bus clocks and system clocks
- A `Control` block to specify RTL output directory and top-level threshold parameters

The top-level design may also contain the following blocks:

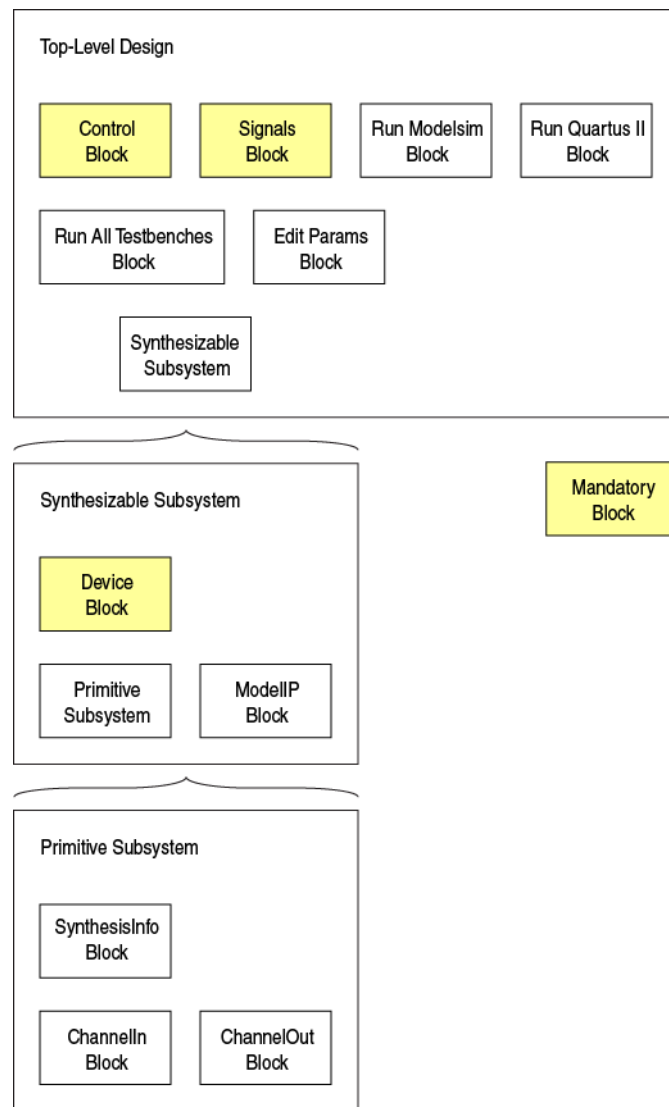
- A `Run ModelSim` block to open post-generation tools:
 - Opens the ModelSim simulator to run a generated RTL testbench
 - Uses Simulink to analyze the design at the synthesizable system level
- A `Run Quartus II` block that opens the generated RTL project in the Quartus II software to allow a full compilation
- A `Run All Testbenches` block, which allows scripts to control system-level, ModelIP and ModelPrim subsystem automatic testbenches
- An `Edit Params` block, which contains shortcuts to parameterization files that run when the model starts up and pre- or post-simulation



Every DSP Builder advanced blockset design must have `Control` block and `Signal` blocks, otherwise you cannot simulate or compile your design. The `Run Quartus II` block and `Run ModelSim` block allow you to compile and simulate your design outside of Simulink. Alternatively, you can specify the run Quartus II and run ModelSim commands from the DSP Builder menu in Simulink.

Figure 1–1 shows the top-level design, synthesizable system, and ModelPrim subsystem relationship.

Figure 1–1. Top-Level Design



Signals Block

Each design example must have a `Signals` block, which you should place at the top level of your model. The `Signals` block specifies the details for the clocks and resets that drive the generated logic.

The DSP Builder advanced blockset uses a single system clock to drive the main datapath logic, and, optionally, a second bus clock to provide an external processor interface. It is most efficient to drive the logic at as high a clock rate as possible. The standard DSP Builder blockset is better for managing multiple clock domains for example, when interfacing to external logic.

The `Signals` block allows you to name the clock, reset, and memory bus signals that the RTL uses, and also provides the following information, which is important for the clock rate:

- Calculates the ratio of sample rate to clock rate to determine the amount of folding (or time-division multiplexing) in the `ModelIP` filters.
- Determines the pipelining required at each stage of logic. For example, the design modifies the amount of pipelining that hard multipliers use, and pipelines long adders into smaller adders based on the absolute clock speed in relation to the FPGA family and speed-grade you specify in the device block.



Cyclone device families do not support a separate bus clock due to the limited multiple clock support in block RAMs on those devices. You must de-activate the separate bus clock option for the `Signals` block when using Cyclone device families.

Control Block

The `Control` block traverses your design, synthesizes the individual `ModelPrim` or `ModelIP` blocks into RTL, and maintains an internal data flow representation of your design. Simulink simulation uses the internal representation of your design example and to write out the RTL and scripts for other tools. A single `Control` block must be present in your top-level model.



DSP Builder applies globally the options in the `Control` block to your design.

Device Block

The `Device` block indicates a particular Simulink subsystem as the top-level of an FPGA device and allows you to specify the target device and speed grade for the device.



All blocks in subsystems below this level of hierarchy, become part of the RTL design. All blocks above this level of hierarchy become part of the testbench.



For more information about these blocks, refer to the *DSP Builder Advanced Blockset Libraries* section in volume 3 of the *DSP Builder Handbook*.

You can hierarchically separate parts of the design into synthesizable systems. You must use a `Device` block, which sets the device family, part number, speed grade, and so on, to indicate the top-level synthesizable system.

You can further hierarchically split the synthesizable system into `ModelPrim` subsystems for `ModelPrim` blocks and `ModelIP` blocks.

You can optionally include `LocalThreshold` blocks to override threshold settings defined higher up the hierarchy.

SynthesisInfo Block

ModelPrim subsystems are scheduled domains for the ModelPrim blocks. ModelPrim subsystems must have a SynthesisInfo block. Generally, you should select **Scheduled** for the synthesis style, which allows DSP Builder to pipeline and redistribute memories optimally to achieve the desired clock frequency.

Other Blocks

ModelPrim subsystems may have a ChannelIn, ChannelOut, GPIn, or GPOut block, to define the ModelPrim subsystem. Within these boundary blocks, DSP Builder optimizes the implementation that the schematic specifies and inserts pipelining registers to achieve the specified system clock rate. When inserting pipelining registers, you must add equivalent latency to parallel signals, which you must keep synchronous so that DSP Builder schedules them together. DSP Builder schedules signals that go through the same input boundary block (ChannelIn or GPIn) to start at the same point in time; it schedules signals that go through the same output boundary block (ChannelOut or GPOut) to finish at the same point in time. DSP Builder adds any pipelining latency, which you added to achieve f_{MAX} , through the signals across the design. DSP Builder applies at the boundary blocks any the corrections to the simulation to account for this latency in HDL generation, such that the whole ModelPrim subsystem remains cycle accurate.

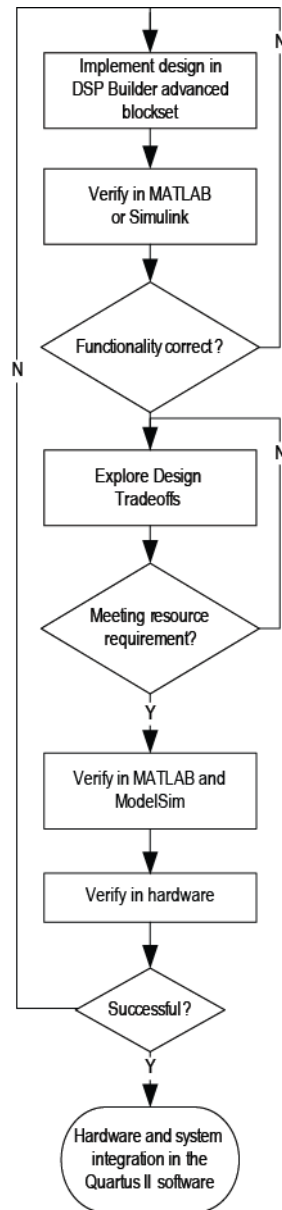
The ChannelIn, ChannelOut, Gpin, or GPOut blocks define the boundary of a ModelPrim subsystem. A ModelPrim subsystem should always include I/O blocks. You must use a **SynthesisInfo** block, with style set to **Scheduled** at the same hierarchical level as these I/O blocks. You can use further subsystems within the ModelPrim subsystem. DSP Builder flattens the ModelPrim subsystem in the optimizations.



You cannot use ModelIP blocks inside ModelPrim subsystems.

Figure 2–1 shows the typical design flow.

Figure 2–1. Design Flow



Typically, the design flow involves the following steps:

1. Implement your design in DSP Builder advanced blockset.

2. Verify the functionality of the design in Simulink. This early verification process focuses on the functionality of your algorithm. Iterate the design implementation if needed.
3. Explore design tradeoffs. You can get early estimates of resource utilization before you go to hardware verification, which allows you to experiment with various implementation optimizations early. You can access memory-logic tradeoff, or logic-multiplier tradeoff by modifying threshold parameters. You may not need to physically modify the design. DSP Builder can automate design space exploration based on your tradeoff options.
4. Verify the design in ModelSim with the automatic testbench flow. This step is the final verification before you port the design to system-level integration. If you sufficiently verify your design early in the design flow, there should be no need to iterate your design.
5. Optionally, verify the design in hardware. To verify the design in hardware use the hardware in the loop (HIL) design from the standard blockset.



For more information, refer to the *Using HIL* chapter in the *DSP Builder Standard Blockset User Guide* section in volume 2 of the *DSP Builder Handbook*.

6. Integrate your DSP Builder advanced blockset design as a black-box design in your top-level design. This integration can involve Qsys integration with a processor, and eventually Quartus II integration.

During the design process you can simulate and debug your design in Simulink. DSP Builder generates synthesizable VHDL for the design at the start of every Simulink simulation. DSP Builder generates an automatic testbench for the whole design and each subsystem. You can use these testbenches to play data that the Simulink simulation captures through the generated VHDL in ModelSim and confirm the results are identical. When you verify these subsystems and modules, you can perform system-level verification and debugging, in both Simulink and the ModelSim simulator. The final step is compilation in the Quartus II software and hardware verification.

Implementing your Design in DSP Builder Advanced Blockset

Staging your Design into Subsystems

Before you start implementing your algorithm:

1. Consider how to stage your design into subsystems. A hierarchical approach makes a design easier to manage, more portable thus easier to update, and easier to debug. If it is a large design, it also makes design partition more manageable.
2. Decide on your timing constraints. DSP Builder advanced blockset achieves timing closure based on your timing constraints, namely sample rate and clock rate. A modular design with well-defined subsystem boundaries, allows you to precisely manage latency and speed of different modules thus achieving time closure effortlessly.

3. Consider the following factors when staging your design into subsystems:
 - Identify the functionality of each submodule of your algorithm, and if you can partition your design into different functional subsystems.
 - In multirate designs consider the sample rate variation at different stages of a datapath. Try not to involve too many different sample rates within a subsystem.
 - If your design has very tight latency requirement, use latency management to define the boundary of a subsystem. DSP Builder advanced blockset applies latency constraints on a subsystem basis.
4. To simplify synchronization, implement modules, which DSP Builder can compute in parallel, in the same subsystem. DSP Builder can apply the same rules more easily to each of the parallel paths. Do not worry about constraining the two paths that may otherwise have different latencies.

ModelPrim Blocks

Most arithmetic and memory blocks from the ModelPrim library allow you to use Simulink 1-D vector (array) and complex types. A design that uses a vector input of width N is the same as connecting N copies of the block with a single scalar connection to each.

Signals that pass through a single ChannelIn, ChannelOut, GPIn, or GPOut block line up correctly in time at their boundaries. However, DSP Builder does not specify the timing relationship between different sets of inputs and outputs, and does not guarantee any fixed relationship if you change clock frequencies or devices.

You can design and debug quickly with zero-latency blocks, without tracking block latencies around your design. DSP Builder displays and calculates the additional latency that your design requires to meet timing constraints as a parameter on the ChannelOut block.

The ChannelIn and ChannelOut blocks define the boundary of a synthesizable ModelPrim subsystem. They pass inputs through to the outputs unchanged, with types preserved. These blocks indicate to the tools that these signals arrive synchronized from their source, and depart from this subsystem synchronized.

Parameters on the ChannelIn and ChannelOut blocks allow the control of automated folding for the ModelPrim subsystems. Folding allows DSP Builder to implement the design with fewer expensive resources, such as multipliers, at the expense of a little more logic by sharing them when the sample rate is below the clock rate.



For more information about folding, refer to [“Folding” on page 10–15](#).

ChannelIn and ChannelOut blocks must be in their own subsystem and directly connected to the subsystem in and out ports. Within the channel in and out pair, you can use nested subsystems, but without the channel in and out pairs. You cannot nest ChannelIn or ChannelOut blocks.

Specifying the Output Data Type

Many of the ModelPrim blocks provide the following options in their **Block Parameters** dialog box to inherit or specify the output data type:

- **Inherit via internal rule:** the number of integer and fractional bits is equal to the maximum of the number of bits in the input data types.
- **Inherit via internal rule with word growth:** the number of fractional bits is equal to the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that DSP Builder can store in the number of bits of the input.
- **Specify via dialog:** additional fields that allow you to set the output type of the block explicitly. For a comparison of Convert blocks and **Specify Via Dialog**, refer to [“Using Convert Blocks and Specifying Output Types” on page 2-22](#).
- **Boolean:** the output type is Boolean.

In general, you can set the output type of a ModelPrim block to **Inherit via internal rule**, or **Inherit via internal rule with word growth**. The type propagates through your design naturally, with blocks potentially growing the word length where necessary. If your design has loops, use the dialog box to specify the output type for at least one block in the loop.



For information about the options that DSP Builder supports by each block, refer to the *ModelPrim Library* chapter in [Volume 3: DSP Builder Advanced Blockset](#) of the *DSP Builder Handbook*.

SynthesisInfo Block

The ModelPrim library includes a SynthesisInfo block that sets the synthesis mode and labels a subsystem described by ModelPrim blocks as the top-level of a synthesizable subsystem tree. DSP Builder flattens and synthesizes the subsystem, and all those below as a unit. If no SynthesisInfo block is present, the style defaults to **WYSIWYG** (rather than **Scheduled**) and DSP Builder issues error messages if insufficient delay is present.

Loops

Consider the following two main cases:

- The simpler case is feed-forward. When no loops exist, feed-forward datapaths are balanced to ensure that all the input data reaches each functional unit in the same cycle. After analysis, DSP Builder inserts delays on all the non-critical paths to balance out the delays on the critical path.
- The case with loops is more complex. Loops cannot be combinational—all loops in the Simulink design must include delay memory. Otherwise Simulink displays an 'algebraic loop' error. In hardware, the signal has to have a specified number of clock cycles latency round the feedback loop. Typically, one or more lumped delays exist with SampleDelay blocks specifying the latency around some or all of the loop. DSP Builder preserves the latency around the loop to maintain correct functional operation. To achieve clock frequency, the total delay of the sum of SampleDelay blocks around the loop must be greater or equal to the required pipelining.

Your design can contain many loops that can interact with or be nested inside each other. Thus, DSP Builder uses standard mathematical linear programming techniques to solve a set of simultaneous constraints.

If the pipelining requirements of the functional units around the loop are greater than the delay specified by the `SampleDelay` blocks on the loop path, DSP Builder generates an error message. The message states that distribution of memory failed as there was insufficient delay to satisfy the f_{MAX} requirement. DSP Builder cannot simultaneously satisfy the pipelining to achieve the given f_{MAX} and the loop criteria to re-circulate the data in the number of clock cycles specified by the `SampleDelay` blocks.

DSP Builder automatically adjusts the pipeline requirements of every `ModelPrim` block according to these factors

- * The type of block: Add or Mult
- * The target f_{MAX}
- * The device family and speedgrade
- * The inputs of inputs (Mux blocks).
- * The bit width in the data inputs (Add and Mult blocks)



Multipliers on Cyclone devices take two cycles at all clock rates. On Stratix V, Arria V, and Cyclone V devices, fixed-point multipliers take two cycles at low clock rates, three cycles at high clock rates. Very wide fixed-point multipliers incur higher latency when DSP Builder splits them into smaller multipliers and adders. You cannot count the multiplier and adder latencies separately because DSP Builder may combine them into a single DSP block. The latency of some blocks depends on what pipelining you apply to surrounding blocks. DSP Builder avoids pipelining every block but inserts pipeline stages after every few blocks in a long sequence of logical components, if f_{MAX} is sufficiently low that timing closure is still achievable.

When you select the **Scheduled** style, you can optionally specify a latency constraint limit that can be a workspace variable or expression, but must evaluate to a positive integer. However, only use this feature to add further latency. Never use the feature to reduce latency to less than the latency required to pipeline the design to achieve the target f_{MAX} .

For more information about the minimum delay on the `SampleDelay` block, refer to [“Sample Delay \(SampleDelay\)” on page 17–54](#).

After you run a simulation in Simulink, the help page for the `SynthesisInfo` block shows the latency, port interface, and estimated resource utilization for the current `ModelPrim` subsystem.

When no loops exist, feed-forward datapaths are balanced to ensure that all the input data reaches each functional unit in the same cycle. After analysis, DSP Builder inserts delays on all the non-critical paths to balance out the delays on the critical path.

In designs with loops, DSP Builder advanced blockset must synthesize at least one cycle of delay in every feedback loop to avoid combinational loops that Simulink cannot simulate. Typically, one or more lumped delays exist. To preserve the delay around the loop for correct operation, the functional units that need more pipelining stages borrow from the lumped delay.

Vectors

Use vectors to build more parameterizable design components. Use vectors with ModelPrim blocks to:

- Make the design more parameterizable
- Speed up simulation
- Simplify the schematic
- Avoid cut-and-paste duplication in many instances
- Enable flexible designs that scale with input vector width

The following three design examples demonstrate using vectors:

- [Matrix Initialization of Vector Memories](#)
- [Matrix Initialization of LUT](#)
- [Vector Initialization of Sample Delay](#)

Vector Utils Library

Often to connect single connections to vectors using parameters—either by replicating or selecting—stops you building a a vector parameterizable library component. For example, replicating a single signal x times to form a vector. If you have to draw and connect this connection when the desired vector changes, you lose the ability to parameterize. You can use Simulink commands in the initialization of masked subsystems to parameterize and reconnect automatically.

The following blocks in the Vector Utils library all use standard Simulink commands for finding blocks, deleting blocks, and lines, also adding blocks, lines, and positioning blocks.

- [Complex to Real-Imag](#)
- [Enabled Delay Line](#)
- [Enabled Feedback Delay](#)
- [Expand Scalar \(ExpandScalar\)](#)
- [Vector Multiplexer \(VectorMux\)](#)
- [Tapped Delay Line \(TappedDelayLine\)](#)

You can use these techniques to build parameterizable utility functions.

Other Vector Blocks—Simulink Selector

The Simulink Selector block enables you to select some signals out of a vector of signals, including operations such as reordering. DSP Builder only supports **Starting index (dialog)** index option, which is equivalent to a static selection. DSP Builder does not support **Starting index (port)**. Set the index and input port size with work space variables.

The Simulink Selector block supports the following features:

- Number of input dimensions—1 only
- Index mode: zero-based or one-based

- Index options
- Select all
- Index vector (dialog)
- Starting index (dialog)

The Simulink Selector block does not support the following features:

- Multidimensional input
- The following index options:
 - Index vector (port)
 - Starting index (port)



For more information about this Simulink block, refer to Simulink help. In hardware, this block synthesizes to wiring.

ModelPrim Blocks in NonModelPrim Subsystems

You can use the following ModelPrim blocks outside of ModelPrim subsystems (but within the synthesizable subsystem):

- And
- BitCombine
- BitExtract
- BitReverse
- Constant
- Nand
- Not
- Nor
- Or
- SampleDelay
- Xnor
- Xor

DSP Builder does not schedule or pipeline these blocks for f_{MAX} . You can use ModelPrim blocks for constants inside the synthesizable part of the design. You should always use ModelPrim constant blocks; never use a Simulink constant block.



Only use all other ModelPrim blocks within scheduled ModelPrim subsystems or outside of the synthesizable part of the design.

Interfaces as Subsystem Boundaries

ModelPrim subsystems allow you to flexibly build your own custom designs, while taking advantage of the optimizations applied by DSP Builder. Optimizations operate hierarchically—DSP Builder optimizes each ModelPrim subsystem or ModelIP block individually.

Interfaces as Scheduling Boundaries

Use a single `ChannelIn` and a single `ChannelOut` block if you want to pipeline all your signals so that inputs are all on the same clock cycle and outputs are all synchronized together on the same output clock cycle. If your subsystem requires inputs to appear on different clock cycles, or to group outputs on to different clock cycles, use multiple `ChannelIn` and `ChannelOut` blocks, or `GPIn` and `GPOut` blocks.

To maintain cycle accuracy at a level outside the `ModelPrim` subsystem, the pipelining that DSP Builder inserts must be accounted for in simulation. The scheduler calculates this added latency depending on factors such as vector widths, data types, and f_{MAX} requirements. As each pipelining stage is added in slices, or cuts, through all parallel signals, just add latency on the inputs or outputs.

Accounting for the latency may give different results to hardware – in particular with multiple inputs and outputs where an input is scheduled after an output.

For more information about latency, refer to [“How to Manage Latency” on page 10–20](#).

Block Interfaces

A DSP Builder design must have the following three basic interface signals:

- `channel`
- `data`
- `valid`

The `channel` (`uint8`) signal is a synchronization counter for multiple channel data on the data signals. It increments from 0 with the changing channels across the data signals within a frame of data.

The data signals can be any number of synchronized signals carrying single or multichannel data.

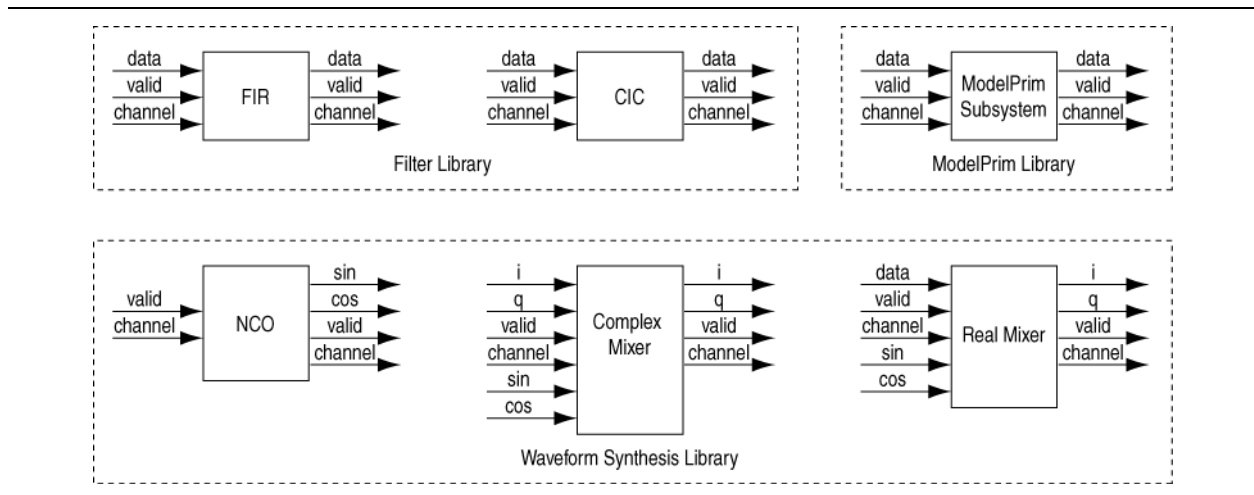
The `valid` (`ufix(1)` or `bool`) signal indicates whether the concurrent data and channel signals have valid information (1), are unknown, or do not care (0).

DSP Builder uses these three synchronized signals, `channel`, `data`, and `valid` to internally connect `ModelIP` or synthesized subsystems and externally connect upstream and downstream blocks. Thus these three signals connect most of the blocks in a DSP Builder advanced blockset design.

Only one set of `valid` and `channel` signals can exist in a `ModelIP` and synthesized subsystem. But multiple data signals can exist in a customized synthesizable subsystem.

Figure 2-2 shows interface protocols of the blocks.

Figure 2-2. Interface Protocols of Blocks.



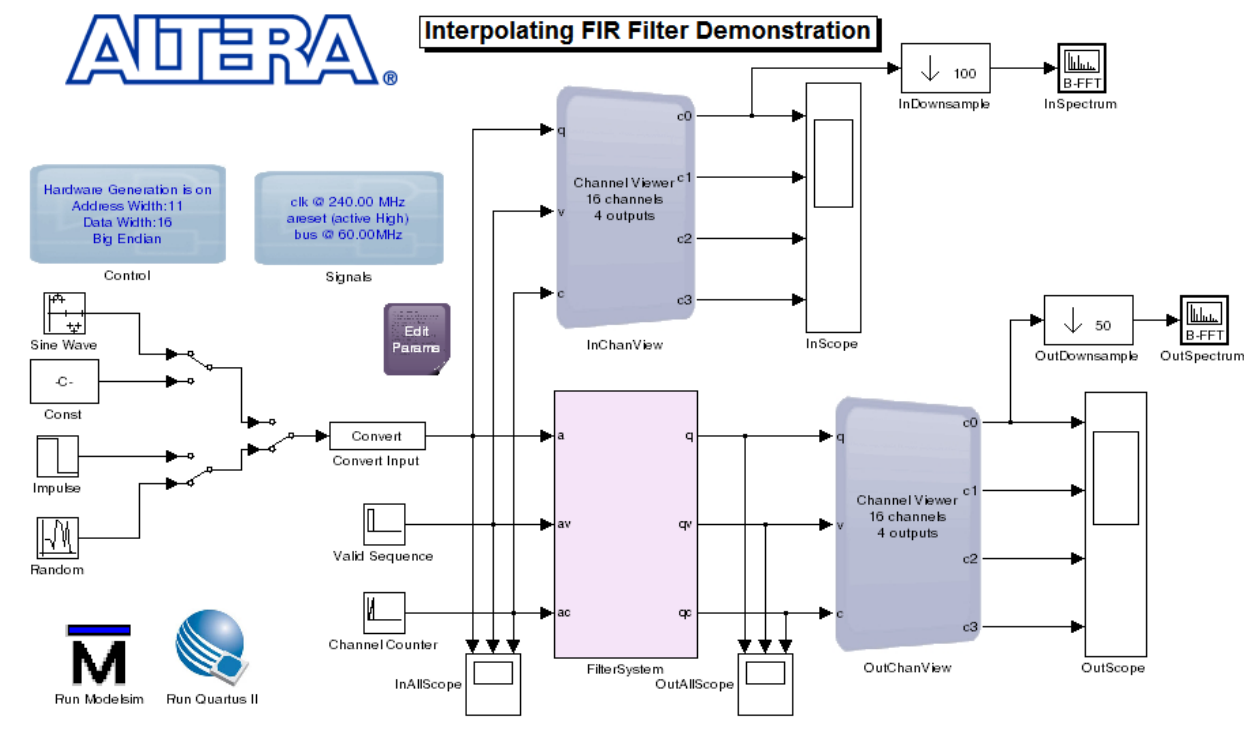
Data on the data wire is only valid when DSP Builder asserts `valid` high. During this clock cycle, `channel` carries an 8-bit integer channel identifier. DSP Builder preserves this channel identifier through the datapath, so that you can easily track and decode data.

This simple protocol is easy to interface with external circuitry. It avoids balancing delays, and counting cycles, because you can simply decode the `valid` and `channel` signals to determine when to capture the data in any downstream blocks. DSP Builder distributes the control structures in each block of your design.

Interface Example

For example, Figure 2-3 shows the Interpolating FIR filter design example.

Figure 2-3. Interpolating FIR Filter Connections



The design example uses just the three data, valid, and channel signals to connect the FilterSystem and ChanView block.

In the FIR filters, the valid signal is an enable. Therefore if the sample rate is less than or equal to the value you specify in the dialog box, data can pass more slowly to the filter. Use this feature when the sample rates of the filters are not all divisible by the clock rate.



Always, take care to generate the first valid signal when some real data exists and to connect this signal throughout the design. The valid signal is a stream synchronization signal.

In ModelPrim subsystems, DSP Builder guarantees all signals that connect to ChannelOut blocks line up in the same clock cycle. That is, the delays balance on all paths from and to these blocks. However, you must ensure all the signals arrive at a ChannelIn block in the same clock cycle.

The ModelIP blocks follow the same rules. Thus it is easy to connect ModelIP blocks and ModelPrim subsystems.

The ModelIP filters all use the same protocol with an additional simplification—DSP Builder produces all the channels for a frame in a multichannel filter in adjacent cycles, which is also a requirement on the filter inputs. If a FIR filter needs to use flow control, pull down the valid signal between frames of data—just before you transmit channel 0 data.

The same `<data, valid, channel>` protocol connects all CIC and FIR filter blocks and all subsystems with ModelPrim blocks. The blocks in the Waveform Synthesis library support separate real and complex (or sine and cosine) signals. The design may require some splitting or combining logic when using the mixer blocks. Use a ModelPrim subsystem to implement this logic.

Connecting Subsystems

To connect ModelIP blocks or a ModelPrim subsystems, use valid and compatible `<data, valid, channel>` pairs.

Periods

For any data signal in a DSP Builder advanced blockset design, the FPGA clock rate to sample rate ratio determines the *period* value of this data signal. In a multirate design, the signal sample rate can change as the data travels through a decimation or interpolation filter. Therefore *period* at different stages of your design may be different.

In a multichannel design, *period* also decides how many channels you can process on a wire, or on one signal. Where you have more channels than you can process on one path, or wire, in a conventional design, you need to duplicate the datapath and hardware to accommodate the channels that do not fit in a single wire. If the processing for each channel or path is not exactly the same, DSP Builder advanced blockset supports vector or array data and performs the hardware and datapath duplication for you. You can use a wire with a one dimensional data type to represent multiple parallel data paths. DSP Builder advanced blockset ModelIP and ModelPrim blocks, such as adder, delay and multiplier blocks, all support vector inputs, or fat wires, so that you can easily connect models using a single bus as if it is a single wire.

Sample Rate

The sample rate may exceed the FPGA clock rate, such as in a super sample rate system, for example in high-speed wireless front-end designs. In a radar or direct RF system with GHz digital-to-analog converters (DAC), the signal driving the DAC can have a sample rate in the GHz range. These high-speed systems require innovative architectural solutions and support for high-speed parallel processing. DSP Builder advanced blockset interpolation filter IP has built in support for super-sample rate signals, and the vector support of its ModelPrim library makes it easy for you to design your super-sample rate module. However, for a super-sample rate design, you must understand how channels are distributed across multiple wires as arrays, and how they are allocated among time slots available on each wire.

Use the following variables to determine the number of wires and the number of channels each wire carries by parameterization:

- *ClockRate* is the system clock frequency.
- *SampleRate* is the data sample rate per channel (MSPS).
- *ChanCount* is the number of channels.



Channels are enumerated from 0 to *ChanCount* – 1.

- The *Period* (or folding factor) is the ratio of the clock rate to the sample rate and determines the number of available time slots:

$$\text{Period} = \max(1, \text{floor}(\text{ClockRate} / \text{SampleRate}))$$

- The *WiresPerChannel* is the number of wires per channel:

$$\text{WiresPerChannel} = \text{ceil}(\text{SampleRate} / \text{ClockRate})$$

- The *WireGroups* is the number of wire groups to carry all the channels regardless of channel rate:


$$\text{WireGroups} = \text{ceil}(\text{ChanCount} / \text{Period});$$

- The number of channel wires the design requires to carry all the channels is the number of channels divided by the folding factor (except for supersampled filters):

$$\text{ChanWireCount} = \text{WiresPerChannel} \times \text{WireGroups}$$

- The number of channels carried per wire is the number of channels divided by the number of channels per wire:

$$\text{ChanCycleCount} = \text{ceil}(\text{ChanCount} / \text{WireGroups})$$

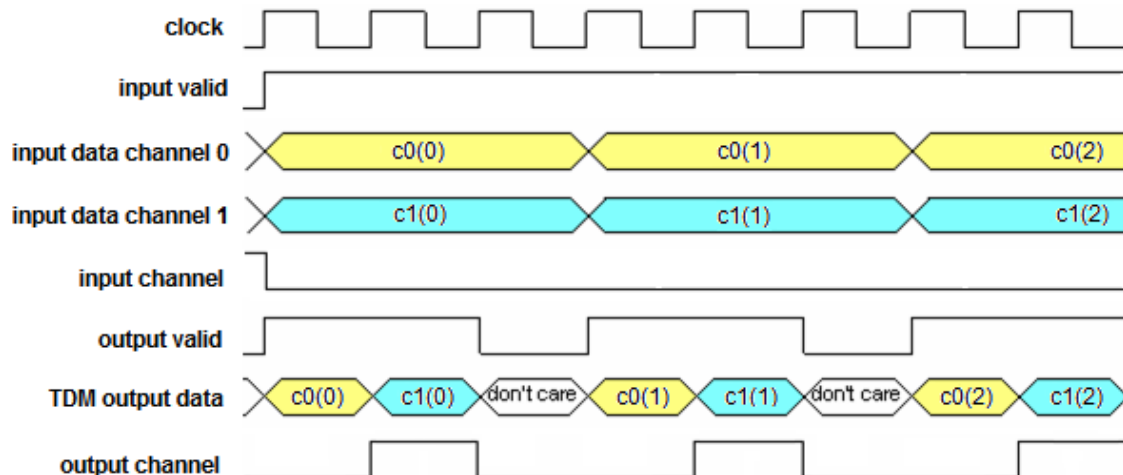
 The channel signal counts through 0 to $\text{ChanCycleCount} - 1$.

Channelization for Two Channels with a Folding Factor of 3

Figure 2-4 on page 2-12 shows how a folding factor of 3 combines two input channels into a single output wire ($\text{ChanCount} = 2$, $\text{ChanWireCount} = 1$, $\text{ChanCycleCount} = 2$).

If the number of channels is greater than the period, multiple wires are required. Each ModelIP block in your design is internally vectorized to build multiple blocks in parallel.

Figure 2-4. Channelization for Two Channels with a Folding Factor of 3



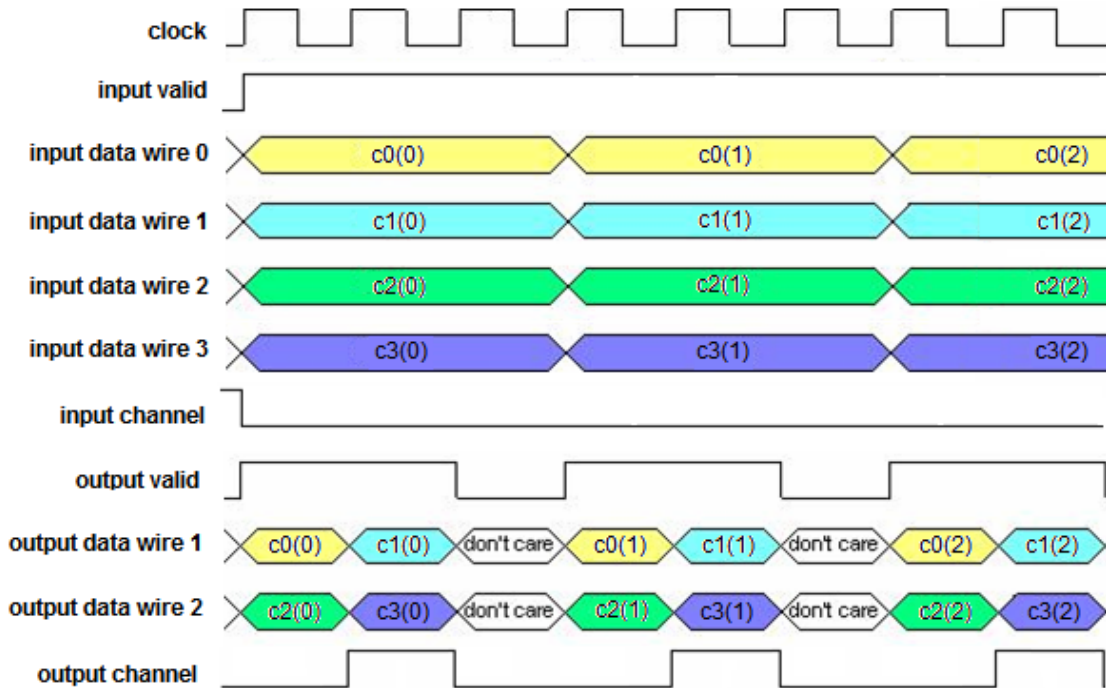
Note to Figure 2-4:

- (1) In this example, three available time slots exist in the output channel and every third time slot has a "don't care" value when the valid signal is low. The value of the channel signal while the valid signal is low does not matter.

Channelization for Four Channels with a Folding Factor of 3

Figure 2-5 on page 2-13 shows how a folding factor of 3 combines four input channels into two wires ($ChanCount = 4$, $ChanWireCount = 2$, $ChanCycleCount = 2$).

Figure 2-5. Channelization for Four Channels with a Folding Factor of 3



Note to Figure 2-5:

- (1) In this example, two wires are required to carry the four channels and the cycle count is two on each wire. DSP Builder distributes the channels evenly on each wire leaving the third time slot as do not care on each wire.



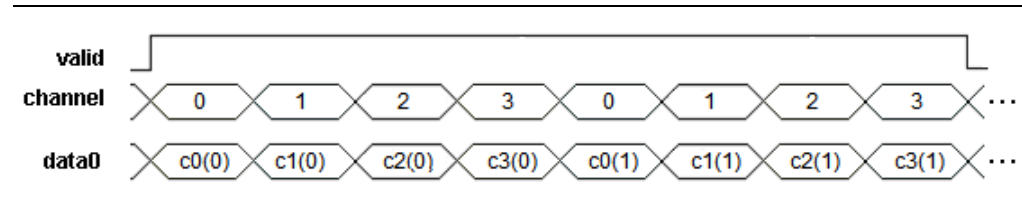
The generated Help page for the block shows the input and output data channel format that the FIR or CIC filter use after you have run a Simulink simulation.

Channel Signal

DSP Builder uses the channel signal for synchronization and scheduling of data. It specifies the channel data separation per wire. The channel signal counts from 0 to $ChanCycleCount - 1$ in synchronization with the data. Thus, for $ChanCycleCount = 1$, the channel signal is the same as the channel count, enumerated 0 to $ChanCount - 1$.

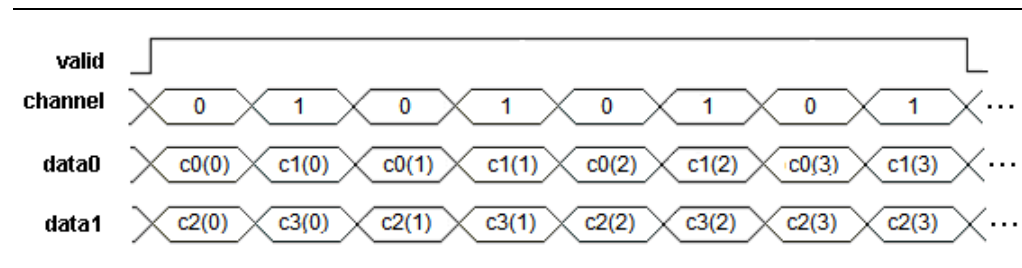
For more than a single data wire, it is not equal to the channel count on data wires, but specifies the synchronous channel data alignment across all the data wires. For example, Figure 2-6 shows the case for four channels of data on one data wire with no invalid cycles.

Figure 2-6. Four Channels on One Wire



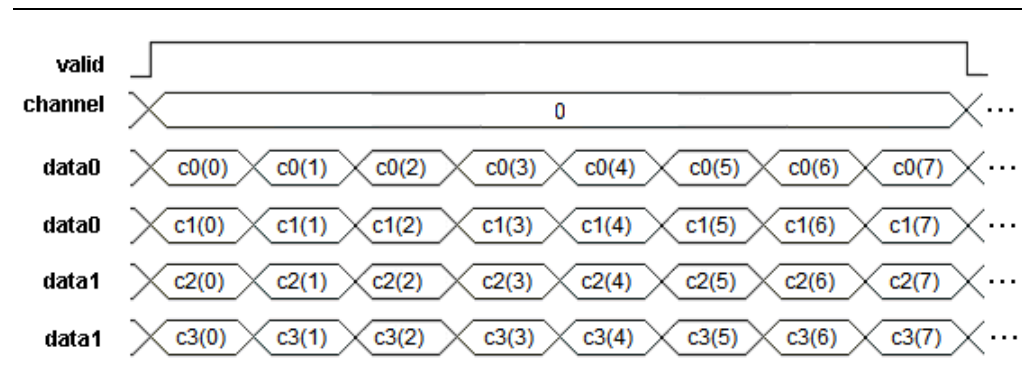
For a single wire, the channel signal is the same as a channel count. However, for $ChanWireCount > 1$, the channel signal specifies the channel data separation per wire, rather than the actual channel number: it counts from 0 to $ChanCycleCount - 1$ rather than 0 to $ChanCount - 1$. Figure 2-7 shows four channels on two wires with no invalid cycles.

Figure 2-7. Four Channels on Two Wires



The channel signal remains a single wire, not a wire for each data wire. It counts over 0 to $ChanCycleCount - 1$, Figure 2-8 shows four channels simultaneously on four wires.

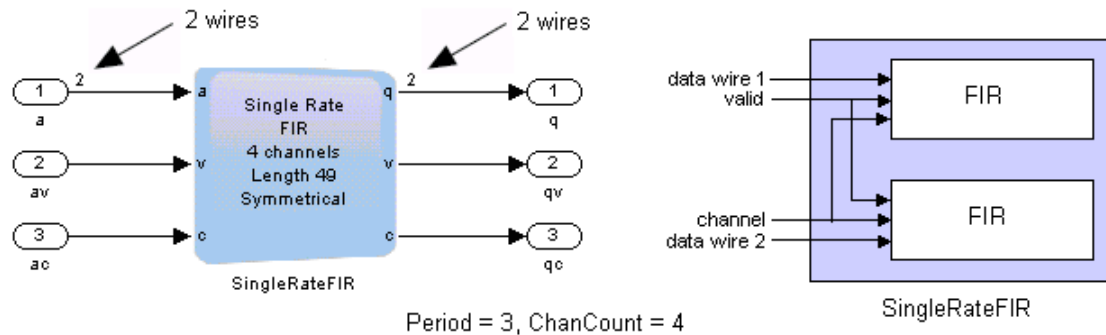
Figure 2-8. Four Channels on Four Wires




Simulink vs Hardware Design Representations

Simulink shows the ModelIP block as a single block, but the data input and output wires from the ModelIP blocks show as a vector with multiple dimension. Multiple wires accommodate all the channels and the Simulink model uses a vector of width 2 (Figure 2-9).

Figure 2-9. Simulink and Hardware Representations of a Single Rate FIR Filter



 To display the *ChanWireCount* in Simulink, point to **Port/Signal Displays** in the Format menu and click **Signal Dimensions**.

 For more information about channelization in a real design, refer to *AN 544: Digital Modem Design with the DSP Builder Advanced Blockset*.

Figure 2-10 and Figure 2-10 show a wideband CDMA digital-up conversion (DUC) example to show how channel's distribution on wires change in a multirate system. In a typical wideband CDMA macro-cell system, the DUC module in the RF card needs to process eight inphase (I) and quadrature (Q) data pairs, resulting in 16 independent channels on the datapath. The input sample rate to a DUC is at sample rate 3.84 MHz as defined in the 3GPP specification. A high-performance FPGA running at 245.76 MHz typically maximizes parallel processing power.

Figure 2-10. 16-channel WCDMA DUC Design



Table 2-1. 16-channel WCDMA DUC Design

Signals	Clock Rate (MHz)	ChanCount	Data Sample Rate (MSPS)	Period	Data Signal Pattern	Interpolation Factor
Input to FIR1	245.76	16	3.48	64	I1, I2, ...I8, Q1, ... Q8, zeros(1, 64-16)	2
Input to FIR2	245.76	16	7.68	32	I1, I2, ...I8, Q1, ... Q8, zeros(1, 32-16)	2

Table 2-1. 16-channel WCDMA DUC Design

Signals	Clock Rate (MHz)	ChanCount	Data Sample Rate (MSPS)	Period	Data Signal Pattern	Interpolation Factor
Input to CIC	245.75	16	15.36	16	I1, I2, ...I8, Q1, ... Q8	8
Output of CIC	245.75	16	122.88	2	I1, I2, I3, I4, I5, I6, I7, I8, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8	8

In this example, the input data at low sample rate 3.84 can accommodate all channels on a single wire. So the *ChanWireCount* is 1. In fact more time slots are available for processing, since *period* is 64 and only 16 channels are present to occupy the 64 time slots. Therefore the *ChanCycleCount* is 16, which is the number of cycles occupied on a wire. As the data travels down the up conversion chain, its sample rate increases and in turn *period* reduces to a smaller number. At the output of CIC filter, the data sample rate increases to 122.88 Msp/s, which means only two time slots are available on a wire. As there are 16 channels, spread them out on 8 wires, where each wire supports two channels. At this point, the *ChanWireCount* becomes 8, and *ChanCycleCount* becomes 2. The *ChanCycleCount* does not always equal *period*, as the input data to FIR1 shows.

For most systems, sample rate is less than clock rate, which gives *WirePerChannel*=1. In this case, *ChanWireCount* is the same as *WireGroups*, and it is the number of wires to accommodate all channels. In a super-sample rate system, a single channel's data needs to be split onto multiple wires. Use parallel signals at a clock rate to give an equivalent sample rate that exceeds the clock rate. In this case, *WiresPerChannel* is greater than one, and *ChanWireCount* = *WireGroups* × *WiresPerChannel* because one channel requires multiple wires.

When connecting two modules in DSP Builder, the output interface of the upstream module must have the same *ChanWireCount* and *ChanCycleCount* parameters as the input interface of the downstream module.

Building Multichannel Systems

To build multichannel systems, use the required channel count, rather than a single channel system and scaling it up.

Multichannel Systems

ModelIP blocks are vectorizable, if data going into a block is a vector requiring multiple instances of, for example a FIR filter, DSP Builder creates multiple FIR blocks in parallel behind a single ModelIP block. If a decimating filter requires a smaller vector on the output, DSP Builder multiplexes data from individual subfilters onto the output vector automatically, to avoid custom glue logic.

Consider the following two cases:

- ModelIP blocks typically take a channel count as a parameter, which is simple to conceptualize. DSP Builder numbers the channels 0 to $(N - 1)$, and you can use the channel indicator at any point to filter out some channels. To merge two streams, DSP Builder creates some logic to multiplex the data. Sequence and counter blocks regenerate valid and channel signals.

- ModelPrim subsystems. The ModelPrim subsystems contain ChannelIn and ChannelOut blocks, but do not have explicit support for multiple channels. To create multichannel logic, draw out the logic required for your design to create a single channel version. To transform to a multichannel system, increase all the delays by the channel count required. Use a mask variable to create a parameterizable component. For an example, refer to the “Multichannel IIR Filter” on page 6-44.

Channel, Data, and Valid Examples

In your design you have a clock rate N (MHz) and a per-channel sample rate M (Mps). If $N = M$, DSP Builder receives one new data sample per channel every clock cycle. For a single-channel design:

```
valid:          < 1 >< 1 >< 1 > ...
channel:        < 0 >< 0 >< 0 > ...
data:           <s00><s01><s02> ...
```

where sPQ = the Q th data sample for channel P .

The frame length, which is the number of clock cycles between data updates for a particular channel, is 1. The out channel count starts (from zero) every clock cycle.

For a multichannel design:

```
valid:          < 1 >< 1 >< 1 > ...
channel:        < 0 >< 0 >< 0 > ...
data:           <s00><s01><s02> ...
                <s10><s11><s12> ...
                <s20><s21><s22> ...
```

If the data is spread across multiple wires, even for multiple channels, the frame length is 1. The channel signal number, which is a channel synchronization counter, rather than an explicit number expressing the actual channels, is again zero on each clock cycle.

If $N > M$ for a single channel design, DSP Builder receives new data samples only every N/M clocks. For example, $N = 300$ MHz and $M = 100$ Mps, gives new data every 3 clock cycles. DSP Builder does not know what the data is on the intervening clocks, and sets the valid to low (0):

```
valid:          < 1 >< 0 >< 0 >< 1 >< 0 >< 0 >< 1 >< 0 >< 0 > ...
channel:        < 0 >< X >< X >< 0 >< X >< X >< 0 >< X >< X > ...
data:           <s00>< X >< X ><s01>< X >< X ><s02>< X >< X > ...
```

where X is unknown or do not care.

The frame length is 3 because of a repeating pattern of channel data every 3 clock cycles. If $N = 300$ MHz and $M = 100$ Mps, and now 2 channel data, the data wire carries the sample for the first channel, the data for the second channel, then a cycle of unknown:

```
valid:          < 1 >< 1 >< 0 >< 1 >< 1 >< 0 >< 1 >< 1 >< 0 > ...
channel:        < 0 >< 1 >< X >< 0 >< 1 >< X >< 0 >< 1 >< X > ...
data:           <s00><s10>< X ><s01><s11>< X ><s02><s12>< X > ...
```

The channel signal now increments as DSP Builder receives the different channel data through the frame. For three channels of data at the same rates, the frame is full along the single data wire:

```
valid:          < 1 >< 1 >< 1 >< 1 >< 1 >< 1 >< 1 >< 1 > ...
channel:        < 0 >< 1 >< 2 >< 0 >< 1 >< 2 >< 0 >< 1 >< 2 > ...
data:           <s00><s10><s20><s01><s11><s21><s02><s12><s22> ...
```

With a fourth channel, the data now spreads across multiple data signals as one wire is not enough to transmit four channels of data in three clock cycles. DSP Builder attempts to distribute the channels evenly on the wires that it has to use:

```
valid:          < 1 >< 1 >< 0 >< 1 >< 1 >< 0 >< 1 >< 1 >< 0 > ...
channel:        < 0 >< 1 >< X >< 0 >< 1 >< X >< 0 >< 1 >< X > ...
dData:          <s00><s10>< X ><s01><s11>< X ><s02><s12>< X > ...
                <s20><s30>< X ><s21><s31>< X ><s22><s32>< X > ...
```

For a fifth channel with the same clock and channel data rates, the data spreads across two data signals that transmit five channels of data in three clock cycles. DSP Builder packs the five channels of data as three on the first wire and two on the second:

```
valid:          < 1 >< 1 >< 1 >< 1 >< 1 >< 1 >< 1 >< 1 >< 1 > ...
channel:        < 0 >< 1 >< 2 >< 0 >< 1 >< 2 >< 0 >< 1 >< 2 > ...
data:           <s00><s10><s20><s01><s11><s21><s02><s12><s22> ...
                <s30><s40>< X ><s31><s41>< X ><s32><s42>< X > ...
```

The channel signal still counts up from zero at the start of each frame and that it specifies a channel synchronization count, rather than expressing all the channels received on a particular clock (which requires as many channel signals as data signals). The `valid` signal also remains one-dimensional, which can under-specify the validity of the concurrent data if, in a particular frame, channel 0 is valid but channel 3 (received on the same clock) is not. In the five-channel example, DSP Builder receives data for channel 2 on the first data signal at the same time as the invalid data on the second data signal. You require some knowledge of the number of channels transmitted.

If $N < M$, DSP Builder receives multiple (M/N) data samples for a particular channel every clock cycle—super-sample data. For example, if $N = 200$ MHz and $M = 800$ Msps, for a single channel:

```
valid:          < 1 >< 1 >< 1 > ...
channel:        < 0 >< 0 >< 0 > ...
data:           <s00><s04><s08> ...
                <s01><s05><s09> ...
                <s02><s06><s0A> ...
                <s03><s07><s0B> ...
```

with four new data samples every clock.

Vectorized Inputs

The data inputs and outputs for the ModelIP blocks can be vectors, which is useful when the clock rate is insufficiently high to carry the total aggregate data. For example, 10 channels at 20 MSPS require $10 \times 20 = 200$ MSPS aggregate data rate. If the system clock rate is set to 100 MHz, two wires must carry this data, and so the Simulink model uses a vector of width 2.

Unlike traditional methods, you do not need to manually instantiate two ModelIP blocks and pass a single wire to each in parallel. Each ModelIP block internally vectorizes. DSP Builder uses the same paradigm on outputs, where it represents high data rates on multiple wires as vectors.

Each ModelIP block determines the input and output wire counts, based on the clock rate, sample rate, and number of channels.

Any rate changes in the ModelIP block affect the output wire count. If a rate change exists, such as interpolating by two, the output aggregate sample rate doubles. DSP Builder packs the output channels into the fewest number of wires (vector width) that supports that rate. For example, an interpolate by two FIR filter may have two wires at the input, but three wires at the output.

The ModelIP block performs any necessary multiplexing and packing. The blocks connected to the inputs and outputs must have the same vector widths, which Simulink enforces. Resolve vector width errors by carefully changing the sample rates.



Most ModelPrim blocks also accept vector inputs.

Verifying your Design in Simulink and MATLAB

Testbenches

A testbench allows you to verify your design. For a DSP Builder advanced blockset design, the testbench is all the subsystems above the subsystem with the Device block. Many of the features of DSP Builder are more accessible if you develop the testbench flexibly.

Before you start implementing your algorithm, consider the modules that connect to and from your design. Understanding the interface to neighboring modules helps you to use the correct stimulus. Consider the sequence of events that you want to test. If multiple channels of data enter your design, align them properly to follow the DSP Builder advanced blockset data format.

Plan your testbench, before you start your design, to allow you to verify and debug your implementation during the design phase. DSP Builder advanced blockset uses a standard interface protocol, and every ModelIP or customized block needs to follow this protocol. The input and output signals of your hierarchical design have a common interface. You can easily bring the output signals of subsystems to the top-level design. If you have the top-level testbench in place, you can conveniently debug your subsystems at all levels, and use the visualization features of Simulink and MATLAB.

Using References

1. Compare your implementation against a reference—a C/C++ bit accurate model, a MATLAB model, or a Simulink design.
2. For a C/C++ model, save your output into a text file and write your C/C++ comparison script.
3. For a MATLAB model, output the DSP Builder advanced blockset testbench data into a workspace or save it into data files.
4. For a Simulink design, put the Simulink model in parallel with your synthesizable design.
5. Use the Simulink scope to compare the two designs.

All the signals in a DSP Builder advanced blockset design use the built-in Simulink fixed-point types. Be careful if you compare your design with a floating-point reference.

Setting Up Stimulus

1. In your top-level testbench, generate stimulus at real time for both data and control signals. Commonly used test data signals include sine waves, random noise, step functions and constants.
2. Generate channel signals and valid signals as repeated sequences.
3. For simulations and tests, format your data, valid, or channel pair according to the DSP Builder advanced blockset interface protocol in MATLAB or Simulink.

Analyzing your Design

To verify the functionality of your design:

1. Use Simulink scope blocks.
2. Use the `SpectrumScope` block to check signal spectrum properties, especially in evaluating filter performance.

Exploring Design Tradeoffs

Bit Growth

DSP Builder uses the built-in Simulink fixed-point types to specify all fixed-point data. You can display the signals as familiar floating-point types.

Using fixed-point types preserves the extra information of binary point position through hardware blocks, so that it is easy to perform rounding and shifting operations without having to manually track the interpretation of an integer value. A fixed-point type change propagates through your design, with all downstream calculations automatically adjusted.

In a typical mathematical algorithm involving multiplication and addition, data width grows as signals travel through the arithmetic blocks. A large data width implies better performance in general, at the price of more hardware resources and potentially slower f_{MAX} (such as in large adders).

Managing Bit Growth

After you update your design or run a simulation:

1. To display the signal type and width turn on Simulink display of signal types.
2. Manage and control bit width at various stages of your design, either because of hardware resource limitation or f_{MAX} speed concerns.
3. Track bit growth by studying the algorithm and determining bit width at various stage of the design from the mathematical model of the design.
4. Use Simulink Fixed-Point Toolbox to visualize the bit width distribution at various places of the design. The fixed-point toolbox displays the histogram of datapath signals you log.
5. To log a data signal in your Simulink design, right-click on the wire and select **Signal Properties**.
6. With the histogram decide how many MSBs are unused in current fixed-point representation, which helps you decide how many MSBs to discard, thus maximizing the dynamic range of your scaled data.

Using Rounding and Saturation

ModelIP blocks such as FIR filters produce output data that use full resolution, that is, DSP Builder performs no rounding or saturation on the output data.

1. Use a Scale block to provide scaling and control your bit growth before data enters the next stage of your ModelIP or ModelPrim subsystems.



For ModelPrim subsystems, use a Convert block to apply rounding and saturation. The Convert block does not perform scaling.

For more information about the Convert and Scale blocks, refer to the *ModelPrim Library* and *Additional and Base Library* chapter in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

1. To reduce bit width of a wide word, use a Convert block instead of just forcing output data type in an arithmetic block.

Whether you choose the Scale block or Convert block to perform rounding and saturation, depends on your algorithm and resource requirement. The Convert block does not support scaling, although you can combine a few ModelPrim blocks to implementing scaling. The Scale block allows you to use a different scaling factor on a cycle basis. It supports both amplification and attenuation of data.

Scaling using ModelPrim Blocks

You can also use ModelPrim blocks to build your own run-time reconfigurable scaling.

1. Use the left shift operation to remove redundant MSBs; use bit extract to remove LSBs and preserve the MSBs.
2. Choose the number of MSBs to discard with the run-time reconfigurable parameter that comes from an input port.

3. Use a control register to connect to this port, and update the shift value by a processor such as a Nios II processor.
4. If the FPGA clock is low, use this implementation to realize different scaling for different channels. If it is a high speed application and your processor bus updates much slower than logic clock rate, you cannot use this circuit to apply different scaling for different channels.

Using Convert Blocks and Specifying Output Types

To change data type with DSP Builder ModelPrim blocks, use one of the following ways:

1. Preserve the real-world value using a Convert block.
2. Preserve bit pattern by setting the output data type mode on any other ModelPrim block or use an ModelPrim Reinterpretcast block.

The Convert Block and Real-world Values

The Convert block converts a data type to preserve the real-world value and optionally rounds and saturates the data type when not possible. Convert blocks can sign extend or discard bits as necessary. Similarly you can convert the same number of bits while preserving the real world value (as far as possible, subject to rounding and saturation). [Figure 2-11](#) shows the Convert block changing data type while preserving real-world value.

Figure 2-11. Convert Block Changing Data Type

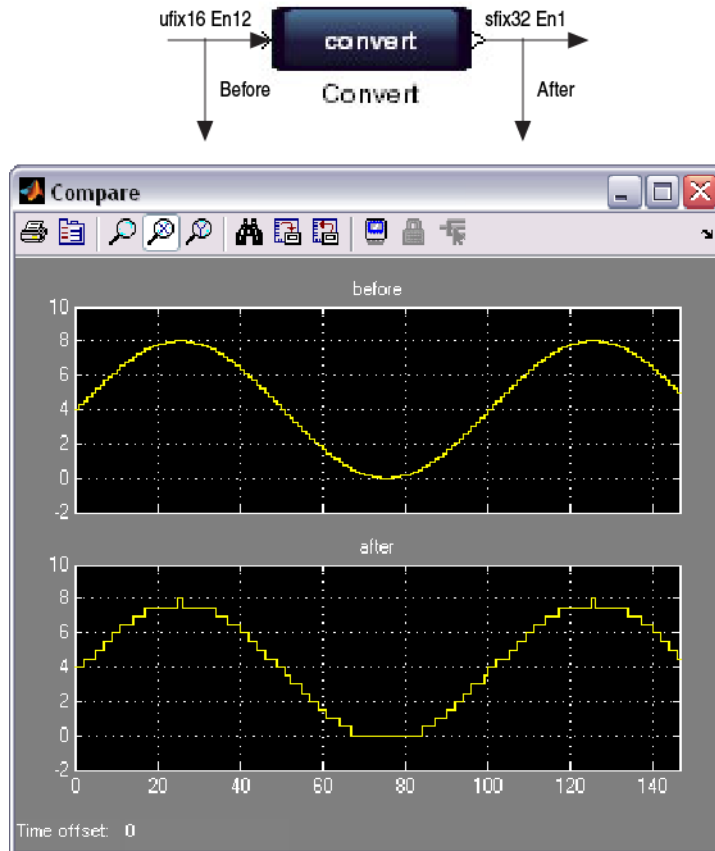
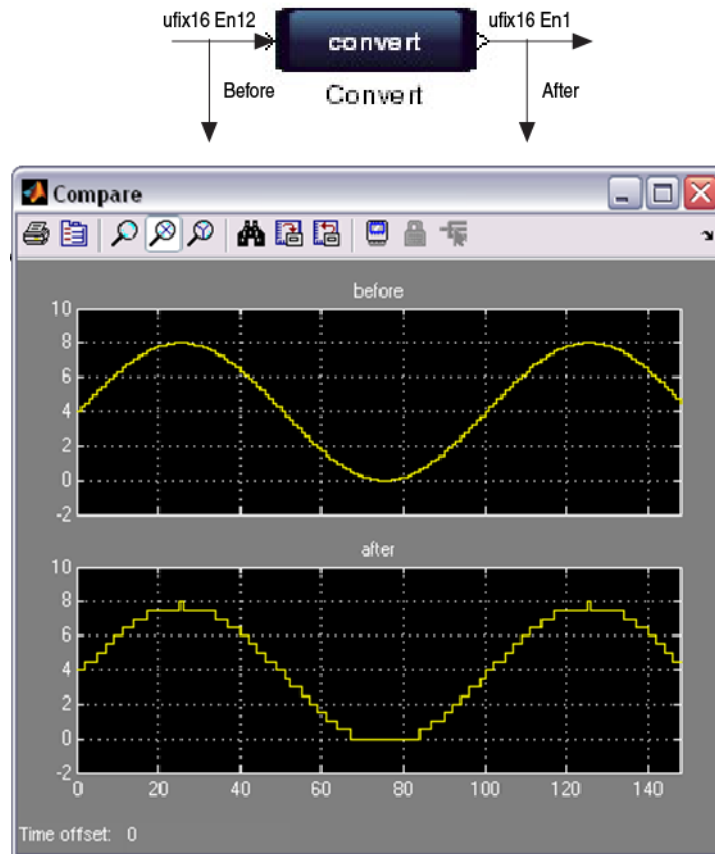


Figure 2-12 shows the Convert block using the same number of bits while preserving real-world value.

Figure 2-12. Convert Block Using Same Number of Bits

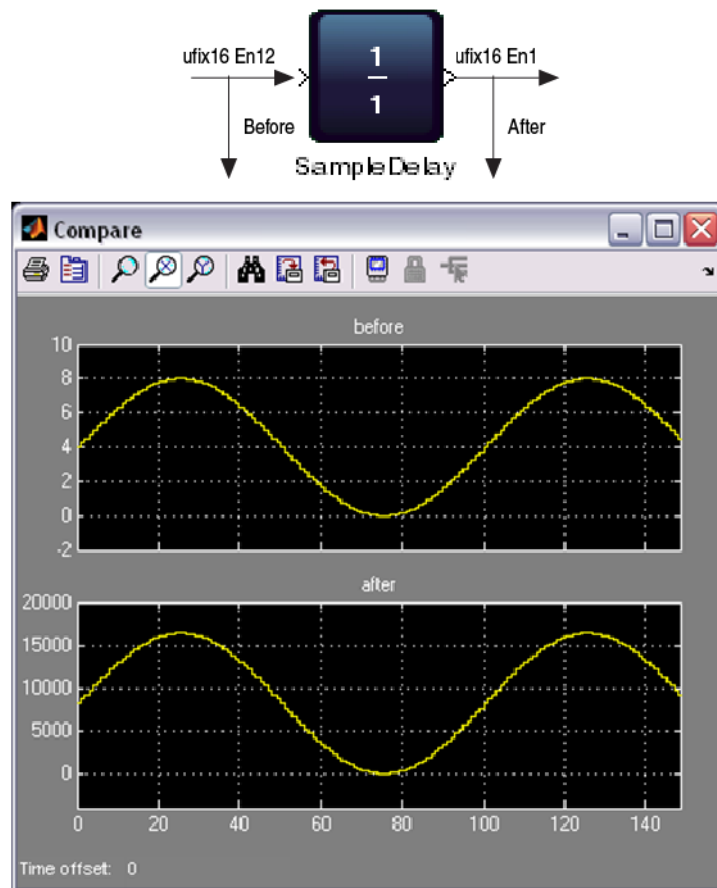


Output Data Types on ModelPrim Blocks

You set the output data type with **Specify via dialog** on any other ModelPrim block (Figure 2-13). For example by using a zero-length sample delay. Specifying the output type with the dialog box is a casting operation. This operation does not preserve the numerical value, just the underlying bits and never adds hardware to a block—just changes the interpretation of the output bits.

Figure 2-13 shows the SampleDelay block with number of delays set to 0.

Figure 2-13. SampleDelay Block



For example, a Mult block with both input data types specified as `sfix16_En15` naturally has an output type of `sfix32_En30`. If you specify the output data type as `sfix32_En28`, the output numerical value is effectively multiplied by four, and a 1*1 input gives an output value of 4.

If you specify output data type of `sfix32_En31`, the output numerical value is effectively divided by two and a 1*1 input gives an output value of 0.5.

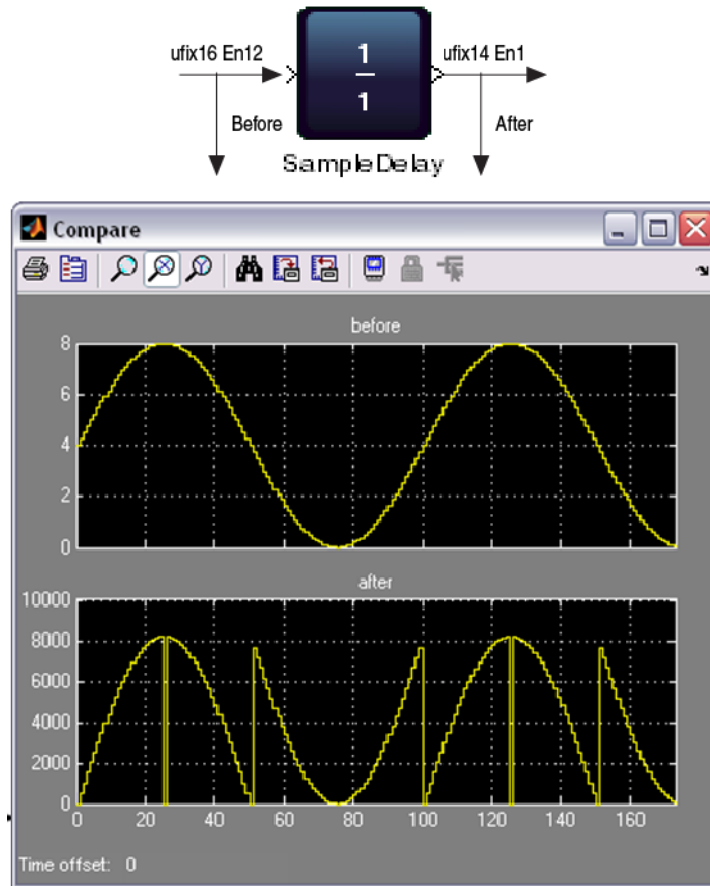
If you want to change the data type format in a way that preserves the numerical value, use a Convert block, which adds the corresponding hardware. Adding a Convert block directly after a ModelPrim block allows you to specify the data type in a way that preserves the numerical value. For example, a Mult block followed by a Convert block, with input values 1*1 always gives output value 1.

To reinterpret the bit pattern and also discard bits, if the type you specify with the **Output data type** is smaller than the natural (inherited) output type, DSP Builder discards the MSBs (most significant bits).

Never set **Specify via dialog** to be bigger than the natural (inherited) bit pattern—DSP Builder performs no zero-padding or sign extension, and the result may generate hardware errors due to signal width mismatches (Figure 2-14). Use the Convert block for any sign extension or zero padding.

Figure 2-14 shows the SampleDelay block when you reduce the bit width.

Figure 2-14. SampleDelay Block and Reducing Bit Width



If you want to use sign extends and zero pads to reinterpret the bit pattern, you can combine these methods.

To set a specific format so that DSP Builder can resolve types, for example, in feedback loops, set **Specify via dialog** on an existing ModelPrim block or insert a zero-cycle sample delay (which generates no hardware and just casts the type interpretation).

To ensure the data type is equal to some other signal data type, force the data type propagation with a Simulink data type propagation block.

Verifying your Design and Debugging

You can verify a design in Simulink or the ModelSim simulator. Also, you can compare Simulink results with the generated RTL, on all synthesizable ModelIP and ModelPrim subsystems.



Altera recommends the automatic testbench flow.

Automatic Testbench

Each ModelIP block, and each synthesized ModelPrim block writes out test vectors to a stimulus file (*.stm) during a Simulink simulation run. DSP Builder creates an RTL testbench for each separate entity in your design (that is, for each ModelIP block and ModelPrim subsystem). These testbenches replay the test vectors through the generated RTL, and compare the output from the RTL to the output from the Simulink model. If a mismatch at any cycle exists, the simulation stops and an error issues. Use these automatic testbenches, to easily verify the correct behavior of the synthesis engine.

The automatic testbench flow uses a stimulus-and-capture method and is therefore not restricted to a limited set of source blocks. The Simulink simulation stores data at the inputs and outputs of each entity during simulation, then the testbench for each entity uses this data as a stimulus and compares the ModelSim output to the Simulink captured output. A result returns that indicates whether the outputs match when the valid signal is high.

ModelSim Simulations

ModelSim simulations compare the complete Simulink model with hardware. This comparison uses the same stimulus capture and comparison method as the automatic testbenches.

DSP Builder captures stimulus files on the device level inputs and records Simulink output data on the device level outputs. It creates a ModelSim testbench that contains the HDL generated for the device that the captured inputs feed. It compares the Simulink outputs to the ModelSim simulation outputs in an HDL testbench process, reports any mismatches, and stops the ModelSim simulation.

Integrating into Hardware

Incorporate the model you implement in DSP Builder advanced blockset into Qsys to create a complete project that integrates a processor, memory, datapath, and control.

Directory Structure

DSP Builder generates the files in a directory structure at the location you specify in the Control block, which defaults to `..\rtl` (relative to the working directory that contains the `.mdl` file).

Generated Files

Table 2-2 shows the generated files within each directory.

Table 2-2. Generated Files

File	Description
rtl directory	
<model name>.xml	An XML file that describes the attributes of your model.

Table 2–2. Generated Files

File	Description
<code><model name>_entity.xml</code>	An XML file that describes the boundaries of the system (for Signal Compiler in designs that combine blocks from the standard and advanced blocksets).
<code>rtl\<model name></code> subdirectory	
<code><block name>.xml</code>	An XML file containing information about each block in the advanced blockset, which translates into HTML on demand for display in the MATLAB Help viewer and for use by the DSP Builder menu options.
<code><model name>.vhd</code>	This is the top-level testbench file. It may contain non-synthesizable blocks, and may also contain empty black boxes for Simulink blocks that are not fully supported.
<code><model name>.add.tcl</code>	This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.
<code><model name>.qip</code>	This file contains information about all the files DSP Builder requires to process your design in the Quartus II software. The file includes a reference to any .qip file in the next level of the subsystem hierarchy.
<code><model name>_<block name>.vhd</code>	DSP Builder generates a VHDL file for each component in your model.
<code><model name>_<subsystem>_entity.xml</code>	An XML file that describes the boundaries of a subsystem as a block-box design (for Signal Compiler in designs that combine blocks from the standard and advanced blocksets).
<code><subsystem>.xml</code>	An XML file that describes the attributes of a subsystem.
<code>*.stm</code>	Stimulus files.
<code>safe_path.vhd</code>	Helper function that the .qip and .add.tcl files reference to ensure that pathnames read correctly in the Quartus II software.
<code>safe_path_msim.vhd</code>	Helper function that ensures a path name reads correctly in ModelSim.
<code><subsystem>_atb.do</code>	Script that loads the subsystem automatic testbench into ModelSim.
<code><subsystem>_atb.wav.do</code>	Script that loads signals for the subsystem automatic testbench into ModelSim.
<code><subsystem>\<block>*.hex</code>	Files that initialize the RAM in your design for either simulation or synthesis.
<code><subsystem>.sdc</code>	Design constraint file for TimeQuest support.
<code><subsystem>.tcl</code>	This Tcl script exists only in the subsystem that contains a Device block. You can use this script to setup the Quartus II project.
<code><subsystem>_hw.tcl</code>	A Tcl script that loads the generated hardware into Qsys.

Automatic Testbench Files

Table 2–3 shows the files for an automatic testbench for an entity under test.

Table 2–3. Files for an Automatic Testbench

File Name	Description
<code><name>.vhd</code>	The HDL that is generated as part of the design (regardless of automatic testbenches).
<code><name>_stm.vhd</code>	An HDL file that reads in data files of captured Simulink simulation inputs and outputs on <code><name></code>

Table 2-3. Files for an Automatic Testbench

File Name	Description
<code><name>_atb.vhd</code>	<p>A wrapper HDL file that performs the following actions:</p> <ul style="list-style-type: none"> ■ Declares <code><name>_stm</code> and <code><name></code> as components ■ Wires the input stimuli read by <code><name>_atb</code> to the inputs of <code><name></code> and the output stimuli and the outputs of <code><name></code> to a validation process that checks the captured Simulink data ■ Channel matches the VHDL simulation of <code><name></code> for all cycles where valid is high ■ Checks that the valid signals match
<code><input>/<output>.stm</code>	The captured Simulink data that the ChannelIn, ChannelOut, GPIn, GPout and ModelIP blocks write.

Each block writes a single stimulus file capturing all the signals through it writing them in columns as doubles with one row for each timestep.

The device-level testbenches use these same stimulus files, following connections from device-level ports to where the signals are captured. Device-level testbenches are therefore restricted to cases where the device-level ports are connected to stimulus capturing blocks.

DSP Builder Designs and the Quartus II Project

DSP Builder creates a Quartus II project in the design directory that contains the `.mdl` file when you add your design to the Quartus II project.

The Quartus II project file (`.qpf`), Quartus II settings file (`.qsf`), and `.qip` files have the same name as the subsystem in your design that contains the Device block. For example, DSP Builder creates the files `DDCChip.qpf`, `DDCChip.qsf`, and `DDCChip.qip` for the `demo_ddc` design.

These files contain all references to the files in the hardware destination directory that the Control block specifies and generate when you run a Simulink simulation. The project automatically loads into the Quartus II software.

When you compile your design the project compiles with the `.tcl` scripts in the hardware destination directory.

The `.qip` file references all the files that the project requires. Use the **Archive Project** command in the Quartus II software to use this file to archive the project.





For information about archiving projects, refer to the Quartus II Help.

Adding a DSP Builder Advanced Blockset Design to an Existing Quartus II Project

To add an advanced blockset design to an existing Quartus II project:

1. Find the `.add.tcl` file in the subsystem that contains the Device block.
2. Alternatively, you can add a reference to the `.qip` file in this subsystem from the `.qip` file for the top-level Quartus II project.

-  For information about the `.add.tcl`, `.qip` and other generated files, refer to [Table 2-2 on page 2-27](#).
-  For information about using Tcl files in the Quartus II software, refer to the Quartus II Help.

Adding Advanced Blockset Components to Qsys

To add a DSP Builder component to Qsys, add a directory that contains generated hardware to the **IP Search Path** in the **Qsys Options** dialog box.

Guidelines

This section summarizes essential guidelines and recommended guidelines that you should follow for all DSP Builder designs.

Essential Guidelines for the Top-Level Design

Use the following essential guidelines for the top-level design:

- Ensure the top-level design has a **Control** block and a **Signals** block.
- Ensure the synthesizable part of your design is a subsystem or contained within a subsystem of the top-level design.
- Ensure there is a **Device** block at the hierarchical level of the synthesizable part of the design.
- Ensure testbench stimulus data types that feed into the synthesizable design are correct, as DSP Builder propagates them.
- For all designs, in the Simulink design, on the **Format** menu, point to **Port/Signal Displays** and turn on **Port Data Types**, **Signal Dimensions** and **Wide Non-Scalar Lines**, which annotates these features to your model to make them visible.

Essential Guidelines for the ModelPrim Blocks

Use the following essential guidelines for ModelPrim blocks:

- Ensure the ModelPrim subsystem contain a **SynthesisInfo** block with style set to **Scheduled**.
- Ensure the ModelPrim subsystems do not contain ModelIP blocks.
- Ensure the `valid` signal is a scalar Boolean signal or `ufix(1)`.
- Ensure the `channel` signal is a scalar `uint(8)`.
- Route all subsystem inputs with associated `valid` and `channel` signals that are to be scheduled together through the same **ChannelIn** blocks immediately following the subsystem inputs. Route any other subsystem inputs through **GPIn** blocks.
- Route all subsystem outputs with associated `valid` and `channel` signals that are to be scheduled together through the same **ChannelOut** blocks immediately before the subsystem outputs. Route any other subsystem outputs through **GPOut** blocks.
- Use **Convert** blocks to change data type preserving real-world value.

- Use **Specify via dialog** options to change data type to preserve bit pattern (with no bits added or removed), or to fix a data type.

Simulink Setup Guidelines

Use the following guidelines for Simulink features when setting up your DSP Builder design:

- Configure Simulink simulation solver and type
- Configure callback functions to load set-up script before running simulation
- Include Scope block or other visualization tools
- Display various port and signal properties to aid debugging and visualization:
 - In Simulink, on the Format menu click **Port/Signal Displays**.
 - Click **Sample Time Colors** to change the color of blocks and wires in particular clock domain—useful when creating multirate designs
 - Click **Port Data Type** option to display the data type of the blocks. You can only connect ports of same data type.
 - Click **Signal Dimensions** to display the dimensions of particular signal wire
- Use a From Workspace block (under Simulink Source block) to import data from workspace and to setup a controlled stimulus in your testbench.
- Use a To Workspace block (under Simulink Sink block) to export data to workspace and to allow further data processing and comparison outside of the testbench.
- Update Diagram (from Model Editor's menu bar **Edit** tab) to display signal properties and check syntax errors before running simulation

Recommended Guidelines for the Top-Level Design

Use the following guidelines for the top-level design:

- Use workspace variables to set parameters you may want to vary. For example, clock rates, sample rates, bit widths, channels.
- Set workspace variables in initialization scripts. Execute them on the model's `PreLoadFcn` and `InitFcn` callbacks, such that the design opens with parameters set, and the next simulation reflects any changes in the next simulation, without explicitly running the script or opening and closing the model.
- Call your main initialization script for the model **setup_<model name>**, and as a shortcut to editing it include the `Edit Params` block in the top level design.
- Build a testbench that is parameterizable with the Channelizer block, which varies correctly with system parameters such as sample rate, clock rate, and number of channels.
- Use the model's `StopFcn` call back to run any analysis scripts automatically.
- Build systems that use the `valid` and `channel` signals for control and synchronization, not latency matching. For example, capture valid output in FIFO buffers to manage dataflow.

- Build up and use your own libraries of reusable components.
- Keep block and subsystem names short, but descriptive. Avoid names with special characters, slashes, or that begin with numbers.
- Use LocalThreshold blocks, with the top-level thresholds, for localized tradeoffs or pipelining.

Recommended Guidelines for ModelPrim Subsystems

Use the following guidelines for ModelPrim subsystems:

- Use vectors to build parameterizable designs—they do not need redrawing when parameters such as number of channels changes.
- Ensure sufficient sample delays exist around loops to allow for pipelining (“[Vector Initialization of Sample Delay](#)” on page 6-46).
- Simulink performs data type, complexity, and vector width propagation. Sometimes Simulink does not successfully resolve propagation around loops, particularly multiple nested loops. If Simulink is unsuccessful, look for where data types are not annotated. You may have to explicitly set data types. Simulink provides a library of blocks to help in such situations, which duplicate data types. For example, the data type prop duplicate block, fixpt_dtprop, (type open fixpt_dtprop from the MATLAB command prompt), which the control library latches use.

Guidelines for Verifying a Design

Use the following guidelines for verifying a design:

- Remember the output only matches the hardware when valid is high.
- Use the **dspba.runModelsimATB** command to display the command it executes. You can cut and paste this command into an open ModelSim UI in the same directory and run manually. With this command you can analyze the behavior of particular subsystems in detail, and can force simulation to continue past errors if necessary.
- Remember if you use FIFO buffers within multiple feedback loops, while the data throughput and frequency of invalid cycles is the same, their distribution over a frame of data may vary (because of the final distribution of delays around the loop). If you find a mismatch, step past errors.

Setting Up Simulink

To set up Simulink for a DSP Builder design, configure the following preferences in Simulink:

1. On the File menu, click **Preferences**.
2. Expand **Configuration defaults** and click **Solver**.
3. For Type, select **Fixed-step solver**, unless you have folding turned on in some part of your design. In that case, you need to select **Variable-step solver**.
4. For Solver select **Discrete (no continuous states)**.
5. Click on Display Defaults and turn on **Show port data types**.

Design Examples and Reference Designs

Altera provides several design examples and reference designs, on which you can base your design.

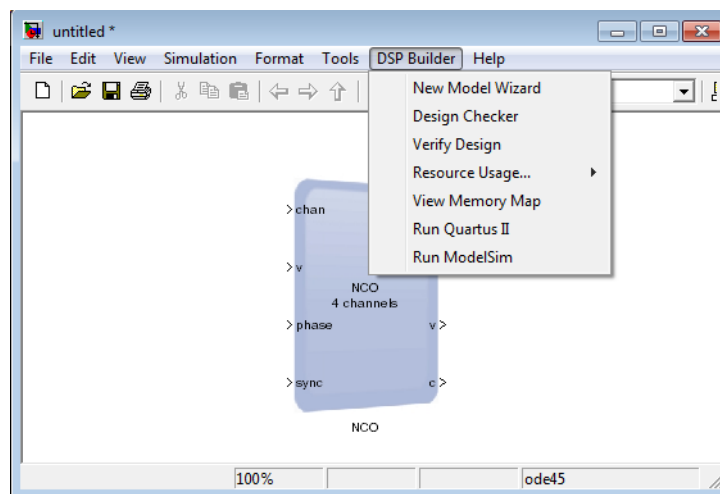


For more information about how to use design examples and the design examples Altera offers, refer to the *ModelIP Tutorial*, the *ModelPrim Tutorial*, or the *Design Examples and Reference Designs* chapters in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

Using the DSP Builder Menu in Simulink

After you open a DSP Builder block or design example, Simulink includes a DSP Builder menu. This menu allows you to easily start all the common tasks you need to perform on your DSP Builder model.

Figure 3–1. DSP Builder Menu




To use the DSP Builder menu, perform the following steps:

1. Start DSP Builder in MATLAB:
 - a. On Windows OS, click on **Start**, point to **All Programs**, click **Altera <version>**, click **DSP Builder**, and click **Start in MATLAB version XX**.
 - b. Click on the Simulink icon, to start Simulink.
 - c. Under libraries, expand **DSP Builder Advanced Blockset**.
 - d. Click on **Base Blocks**.
 - e. Right-click **Control** and click **Add to new model**.
2. To start a new model and setup script that contains everything you need for a DSP Builder model, on the DSP Builder menu, click **New Model Wizard**. You can select a fixed- or floating-point model, you can select the type (simple or with channelizer), enter the model name, and select where to save the model.
3. To check your models for common mistakes click **Design Checker**. You can select individual blocks and click **Check**. Alternatively, select **Errors** or **Guidelines** and click **Run all Checks**.
4. To configure and run automatic testbenches:
 - a. Turn on **Create Automatic Testbenches** and **Coverage in Testbenches** on the Control block. DSP Builder performs stimulus capture for testbenches on the inputs and outputs of ModelIP blocks and by ChannelIn, ChannelOut, GPIn, and GPOut blocks.
 - b. Click **Verify Design**.

When you turn on **Import ModelSim results to MATLAB**, DSP Builder uses the following floating-point tolerances:

 - For single precision:
 - Tolerance = 0.001
 - Zero tolerance = 0.0000001
 - For double precision:
 - Tolerance = 0.000001
 - Zero tolerance = 0.0000000001
5. To set the Avalon-MM slave interface parameters, select **Avalon Interfaces > Avalon-MM slave**.

 Alternatively, you can see the parameters by adding the AvalonMMSlavesettings block to your design.
6. To examine the generated resource summaries:
 - a. On the Simulation menu, click **Start**.
 - b. Point to **Resource Usage** and click **Design** for a top-level design summary.

7. To view the memory map:
 - a. On the Simulation menu, click **Start**.
 - b. Point to **Memory Map** and click **Design**. DSP Builder highlights in red any memory conflicts.



DSP Builder also generates the memory map in the *<design name>_mmap.h* file.



8. To simulate in the ModelSim simulator, click **Run ModelSim**. Alternatively, double-click on the **Run ModelSim** block. You can run individual system or single device-level testbenches.



DSP Builder limits the **Run ModelSim** list to 50 testbenches. If your design has more than 50 testbenches, use the command line.

9. To run an individual subsystem or ModelIP block testbench on a system that is open or a block that you select, type the following command:

```
dspba.runModelsimATB(<model name>,<path to subsystem>)
```

where path to subsystem is *gcs* (get current system) or *gcb* (get current block).

10. To compile your design in the Quartus II software, click **Run Quartus II**. The Quartus II software opens and you can click **Start Compilation** on the Processing menu.

Running Automatic Testbenches

To run automatic testbenches:

- To get a list of the blocks in a design that have automatic testbenches, run the following command in MATLAB:

```
getBlocksWithATBs('model')
```

- To load an automatic testbench from the ModelSim simulator, use the following command:

```
source <subsystem>_atb.do
```

Alternatively, in ModelSim on the Tools menu click **Execute Macro** and select the required **.do** file.

- You can run an automatic testbench targeting a subsystem or a ModelIP block in your design, or you can run an automatic testbench on all of your design.
- To run an automatic testbench from the MATLAB command line on a single entity, use the command `dspba.runModelsimATB`.

- To run testbenches for all subsystems and the device level and set testbench options: in the simulink window, click **DSP Builder > Verify Design** or type:

```
run_all_atbs(<model name>, Run simulation? (0:1), run Quartus (0:1))
```

- To run the device level testbench in the ModelSim simulator, click the Run ModelSim block or click **DSP Builder > Run ModelSim**.

The dspba.runModelsimATB Command Syntax

The `dspba.runModelsimATB` command has the following syntax:

```
dspba.runModelsimATB('model', 'entity', ['rtl_path']);
```

where:

- `model` = design name (without extension, in single quotes)
- `entity` = entity to test (the name of a ModelPrim subsystem or a ModelIP block, in single quotes)
- `rtl_path` = optional path to the generated RTL (in single quotes, if not specified the path is read from the Control block in your model)

For example:

```
dspba.runModelsimATB('demo_fft16_radix2', 'FFTChip');
```

The return values are in the format `[pass, status, result]` where:

- `pass` = 1 for success, or 0 for failure
- `status` = should be 0
- `result` = should be a string such as:

```
"# ** Note: Arrived at end of stimulus data on clk <clock name>"
```

An output file with the full path to the component under test writes in the working directory. DSP Builder creates a new file with an automatically incremented suffix each time the testbench is run. For example:

```
demo_fft_radix2_DUT_FFTChip_atb.6.out
```

This output file includes the ModelSim transcript and is useful for debugging if you encounter any errors.

Running All Automatic Testbenches

To automatically run all the individual automatic testbenches in a design use the **command** `run_all_atbs`. Run this command from the same directory that contains the `.mdl` file.

The command run_all_atbs Command Syntax

This command has the syntax:

```
run_all_atbs('model', [runSimulation], [runFit]);
```

where:

- `model` = design name (without extension, in single quotes)
- `runSimulation` = optional flag that runs a simulation when specified (if not specified, a simulation must run previously to generate the required files)
- `runFit` = optional flag which runs the Quartus II Fitter when specified

For example:

```
run_all_atbs('demo_agc');  
run_all_atbs('demo_agc', true);
```



```
run_all_atbs('demo_agc', false, true);  
run_all_atbs('demo_agc', true, true);
```

The return value is 1 if all tests are successful or 0 if any tests fail. The output is written to the MATLAB command window.

Testbench Error Messages

Typical error messages have the following form:

```
# ** Error (vcom-13) Recompile <path>altera_mf.altera_mf_components  
because <path>iee.std_logic_1164 has changed.  
  
...  
# ** Error: <path>mdl_name_system_subsystem_component.vhd(30):  
(vcom-1195) Cannot find expanded name:  
'altera_mf.altera_mf_components'.  
  
...  
# ** Error: <path>vcom failed.  
  
...  
# At least one module failed to compile, not starting simulation.
```

These errors may occur when a ModelSim precompiled model is out of date, but not automatically recompiled. A similar problem may occur after making design changes when ModelSim has cached a previously compiled model for a component and does not detect when it changes. In either of these cases, delete the `rtl` directory, resimulate your design and run the `dspba.runModelsimATB` or `run_all_atbs` command again.

If you run the Quartus II Fitter, the command also reports whether the design achieves the target f_{MAX} . For example:

```
Met FMax Requirement (FMax(291.04) >= Required(200))
```

A summary also writes to a file `results.txt` in the current working directory. For example:

```
Starting demo_agc Tests at 2009-01-23 14:58:48  
demo_agc: demo_agc/AGC_Chip/AGC hardware matches simulation (atb#1):  
PASSED  
  
demo_agc: Quartus II compilation was successful.  
(Directory=../quartus_demo_agc_AGC_Chip_2): PASSED  
demo_agc: Met FMax Requirement (FMax(291.04) >= Required(200)):  
PASSED  
  
Finished demo_agc Tests at 2009-01-23 15:01:59 (3 Tests, 3 Passes, 0  
Skipped, 0 Failed (fmax), 0 Failed (non-fmax))
```


This tutorial demonstrates how to use blocks from the DSP Builder ModelIP library. It shows how you can double the number of channels through a filter, increase the f_{MAX} , and target a different device family by editing top-level parameters in Simulink.

Creating a ModelIP Design

To create a ModelIP design:

1. Start DSP Builder: on Windows OS, click on **Start**, point to **All Programs**, click **Altera <version>**, click **DSP Builder**, and click **Start in MATLAB version XX**.
2. Type `demo_firi` at the MATLAB command prompt, which opens the FIR design example.
3. From an open Simulink model click on the DSP Builder menu. Select **New Model Wizard**.



You must have a model open, before you use the DSP Builder menu.

4. Specify the following **New Model Settings**:
 - **Fixed**
 - **ModelIP (simple)**
 - **my_firi**
5. Browse to an appropriate output directory.
6. Click **Generate**.
7. Edit the basic parameter values in `setup_my_firi.m` to match the equivalent settings in `setup_demo_firi.m`.

Simulating the Design

To simulate the design, perform the following steps:

1. In the `demo_firi` window, on the Simulation menu click **Start**. MATLAB generates output HDL for the design.
2. On the DSP Builder menu point to **Resource Usage** and click **Design**. You can view the resources of the whole design and the subsystems. Click **Close**.

3. Double click the `FilterSystem` subsystem, right-click on the `InterpolatingFIR` block, and click **Help**. After simulation, DSP Builder updates this help to include the following information:
 - The latency of the filter
 - The port interface
 - The input and output data format
 - The memory interface for the coefficients.
4. To display the latency of the filter on the schematic, right-click on `InterpolatingFIR` block and click **Block Properties**. On the **Block Annotation** tab, in block property tokens double-click on `%<latency>`. In Enter text and tokens for annotation, type **Latency =** before `%<latency>`. Click **OK**. The latency now displays beneath the block.

Verifying the Design

To verify the design, perform the following steps:

1. On the DSP Builder menu click **Verify Design**. Turn on **Verify at subsystem level**, turn off **Run Quartus II Software**, and click **Run Verification**. MATLAB verifies that the Simulink simulation results match a simulation of the generated HDL in the ModelSim simulator.
2. Close both verification windows when MATLAB completes the verification.



If you turn on **Run Quartus II Software**, the verification script also compiles the design in the Quartus II software. MATLAB reports the postcompilation resource usage details in the verification window.

Compiling the Design

To compile the design and check that it meets f_{MAX} requirements, perform the following steps:

1. On the DSP Builder menu click **Run Quartus II**. On the Processing menu, click **Start Compilation**.
2. View timing closure:
 - a. In the **Task** pane, expand **TimeQuest Timing Analyzer**.
 - b. Double-click **View Report**.
 - c. In the **Table of Contents** pane expand **Slow 900mV 85C Model** and click **Fmax Summary**.
3. View the resource utilization:
 - a. On the **Task** pane expand **Fitter (Place & Route)**.
 - b. Double-click **View Report**.
 - c. In the **Table of Contents** pane expand **Resource Section** and click **Resource Usage Summary**, which shows the number of DSP block 18-bit elements.

Reparameterizing the FIR Filter

To reparameterize the interpolating FIR filter to double the number of channels, perform the following steps:

1. Double-click the `EditParams` block to open `setup_my_firi.m` in the MATLAB Editor. Change `my_firi.ChanCount` to **32** and click **Save**.
2. Simulate the design.

On the `InterpolatingFIR` block the latency and the number of multipliers increases.

3. On the DSP Builder menu, click **Verify Design**, and click **Clear Results** to clear the output pane. Turn on **Verify at device level** and **Run Quartus II Software only**, turn off **Verify at subsystem level**, and click **Run Verification**.

The Simulink simulation matches a ModelSim simulation of the generated HDL. The design meets timing but the number of multipliers and logic increases. The number of channels doubles, but the number of multipliers does not double, because the design shares some multipliers.

Doubling the Target Clock Rate

To double the clock rate, perform the following steps:

1. Double-click the `EditParams` block to open `my_firi.m` in the MATLAB Editor. Change `my_firi.ClockRate` to **480.0** and click **Save**.
2. Simulate the design.
3. On the DSP Builder menu, click **Verify Design**, and click **Clear Results** to clear the output pane. Click **Run Verification**.

The design now closes timing above 480 MHz. At the higher clock rate, the design shares multiplier resources, and the multiplier count decreases back to 6.

Retargeting Design to Different Device Family

To target the design to a different device family, perform the following steps:

1. Double-click the `EditParams` block to open `demo_firi.m` in the MATLAB Editor. Change `my_firi.ClockRate` back to **240.0** and click **Save**.
2. On the `Device` block, for **Device Family**, select **Cyclone IV GX**. Leave all other parameters at their defaults and click **OK**.
3. Simulate the design.
4. On the DSP Builder menu, click **Verify Design**, and click **Clear Results** to clear the output pane. Click **Run Verification**.

The design now closes timing above 240 MHz. DSP Builder uses 18 9-bit multipliers on this device.

This tutorial shows how to build a simple design example that uses blocks from the ModelPrim library to generate a Fibonacci sequence.

Each ModelPrim block in the design example is parameterizable. When you double-click a block in the model, a dialog box appears where you can enter the parameters for the block. Click the **Help** button in these dialog boxes to view Help for a specific block.

You can use the **demo_fibonacci.mdl** model in the *<DSP Builder Advanced install path>\Examples\ModelPrim* directory or you can create your own Fibonacci model.

The Fibonacci Sequence

The Fibonacci sequence is the sequence of numbers that you can create when you add 1 to 0 then successively add the last two numbers to get the next number:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Creating the Fibonacci Model



Leave any block parameters that the procedures do not mention with their default values.

Creating a New Model

To create a new model, perform the following steps:

Creating a ModelIP Design

To create a ModelIP design:

1. Start DSP Builder: on Windows OS, click on **Start**, point to **All Programs**, click **Altera <version>**, click **DSP Builder**, and click **Start in MATLAB version XX**.
2. From an open Simulink model click on the DSP Builder menu. Select **New Model Wizard**.



You must have a model open, before you use the DSP Builder menu.

3. Specify the following **New Model Settings**:
 - **Fixed**
 - **Fixed-point ModelPrim (simple)**
 - **my_fibonacci**
4. Browse to an appropriate output directory.
5. Click **Generate**.

Adding Blocks from the ModelPrim Library

To add blocks from the ModelPrim library, perform the following steps:

1. In MATLAB, click **Simulink**.
2. In the Simulink Library Browser, click **DSP Builder Advanced Blockset** and **ModelPrim** to view the blocks in the ModelPrim library.
3. In the generated model, open the **dut > prim** subsystem, which has a ChannelIn and ChannelOut block.
4. Drag and drop two SampleDelay blocks into your model.
5. Select both of the SampleDelay blocks and point to **Rotate and Flip** on the popup menu and click **Flip Block** to reverse the direction of the blocks.
6. Drag and drop Add and Mux blocks from the ModelPrim library into your model.
7. Drag and drop a Constant block:
 - a. Select **Specify via Dialog** for **Output data type mode**.
 - b. For **Output type** enter `ufix(1)`.
 - c. For **Scaling** enter 1
 - d. For **Value** enter 1.
8. Connect the blocks.
9. Double-click on the second SampleDelay block (SampleDelay1) to display the **Function Block Parameters** dialog box and change the **Number of delays** parameter to 2.
10. Double-click on the Add block to display the **Function Block Parameters** dialog box and set the parameters (Table 5-1).

Table 5-1. Parameters for the Add Block

Parameter	Value
Output data type mode	Specify via dialog
Output data type	ufix(120)
Output scaling value	2 ⁻⁰
Number of inputs	2


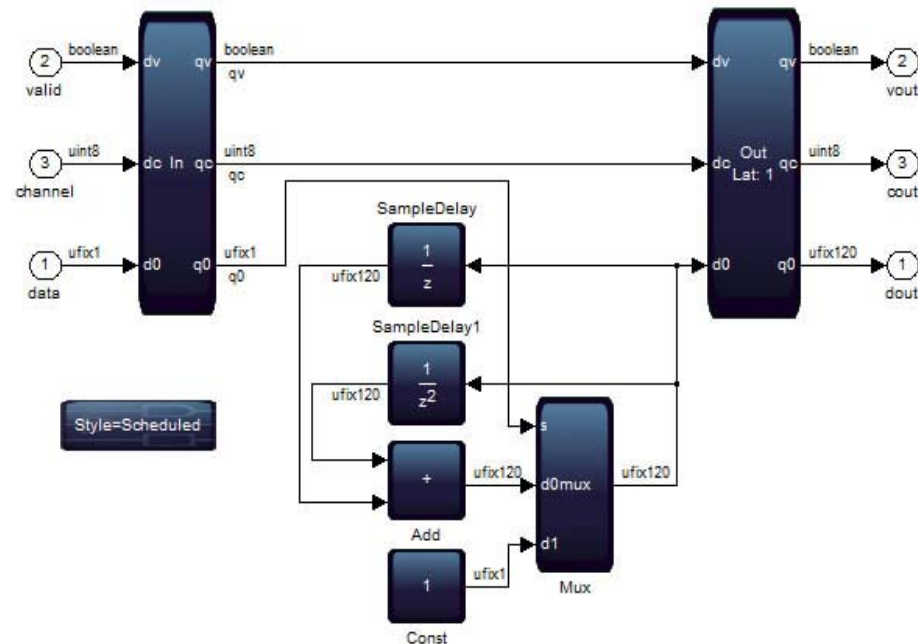
 Specify the data type because this design contains loops and DSP Builder cannot determine the type if one of the inherit data options is set.

Figure 5-1. Updated Fibonacci Subsystem

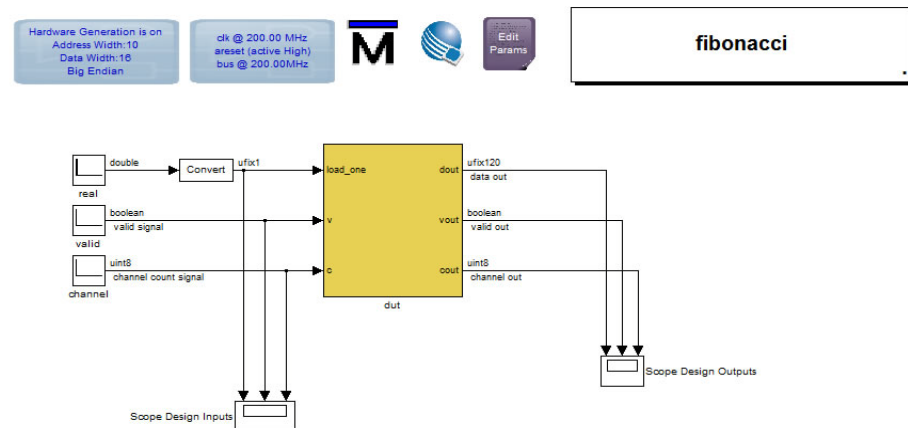


Completing the Top-Level Model

To complete the top-level Fibonacci design example, set the parameters on the test-bench source blocks:

1. Double-click on the Real block to display the **Source Block Parameters** dialog box.
2. Set the **Vector of output values** to `[0 1 1 1 zeros(1,171)].'` in the **Main** tab.
3. Switch to the Editor window for `setup_fibonacci.m`.
4. Change the parameters to:
 - `fibonacci_param.ChanCount = 1;`
 - `fibonacci_param.SampleRate = fibonacci_param.ClockRate;`
 - `fibonacci_param.input_word_length = 1;`
 - `fibonacci_param.input_fraction_length = 0;`
5. In the top-level model, delete the ChannelView, the 'Scope Deserialized Outputs' scope and any dangling connections.
6. Double-click the Convert block and make the input unsigned by changing:

7. `fixdt(1,fibonacci_param.input_word_length,fibonacci_param.input_fraction_length)`
to:
`fixdt(0,fibonacci_param.input_word_length,fibonacci_param.input_fraction_length)`
8. Save the Fibonacci model.

Figure 5-2. Completed Fibonacci Model

For more information about the Control and Signals block, refer to the “[DDC Design Example](#)” on page 7-1, or to their block descriptions in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

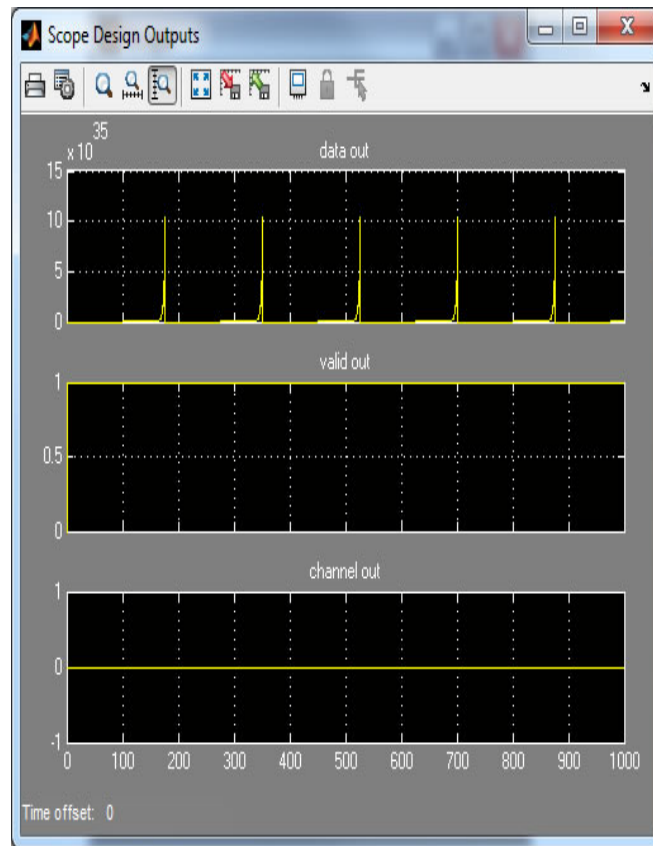
Simulating the Design in Simulink

To check your design, simulate your design example in Simulink, by following these steps:


1. Click **Simulation > Run**.

2. Double-click on the Scope block and click **Autoscale** in the scope to display the simulation results (Figure 5-3 on page 5-5).

Figure 5-3. Fibonacci Sequence in the Simulink Scope



The sequence on the `fib` output starts at 0, and increments to 1 when `q_v` and `q_c` are both high at time 21.0. It then follows the expected Fibonacci sequence incrementing through 0, 1, 1, 2, 3, 5, 8, 13 and 21 to 34 at time 30.0.

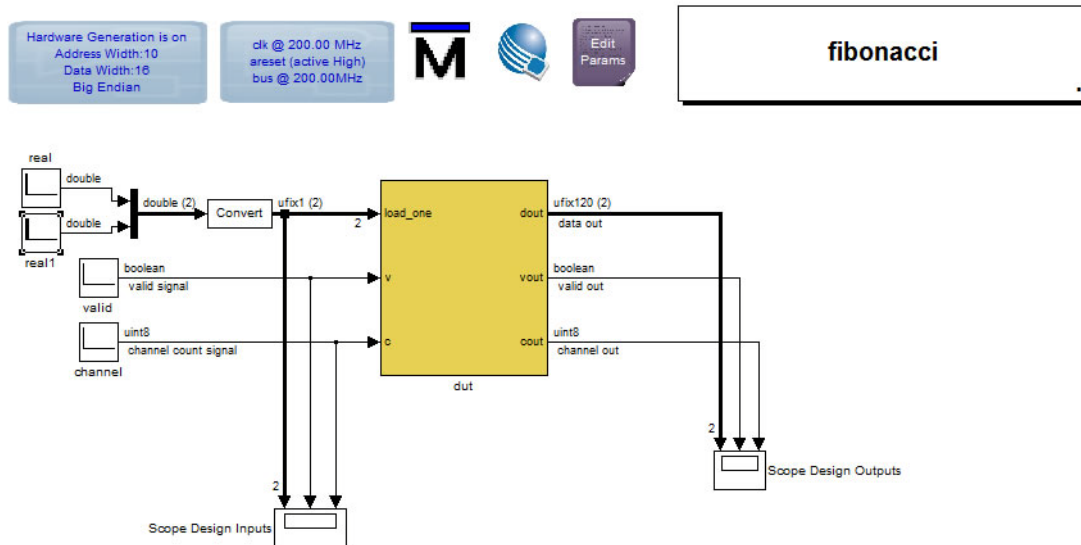
 You can verify that the `fib` output continues to increment according to the Fibonacci sequence by simulating for longer time periods.

Using Vector Types

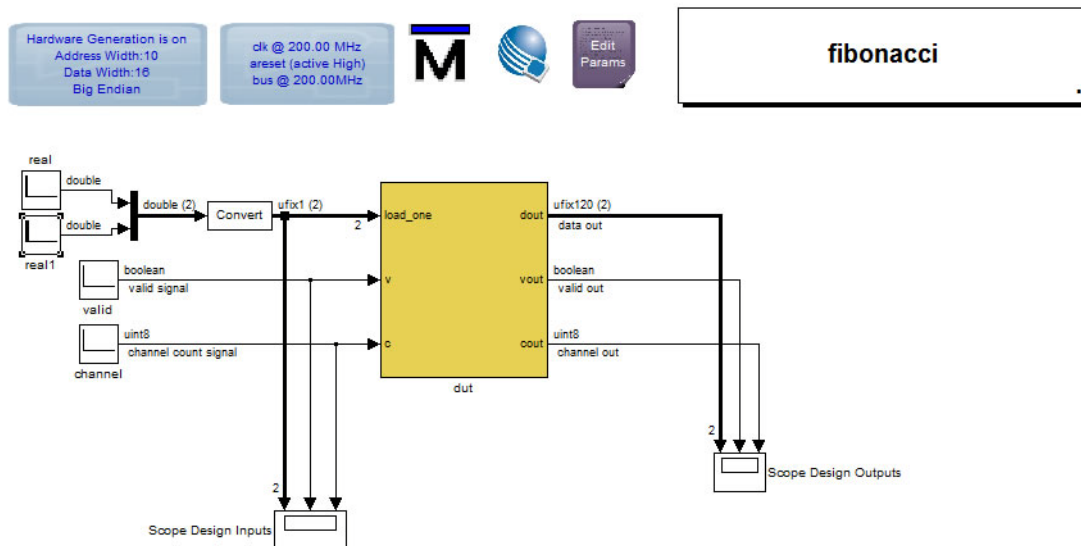
To modify the top-level Fibonacci design example to generate vector signals, perform the following steps:

1. Copy the real input block, add a Simulink mux and connect to the Convert block.

2. Edit the timing of the real1 block, for example [0 1 1 1 zeros(1,50)].

Figure 5-4. Fibonacci Vector Model

3. Run the simulation.
4. In the Scope Design Outputs scope zoom in on the y axis to see the short Fibonacci cycle.

Figure 5-5. Fibonacci Scope Design Outputs

Exploring the Generated Files

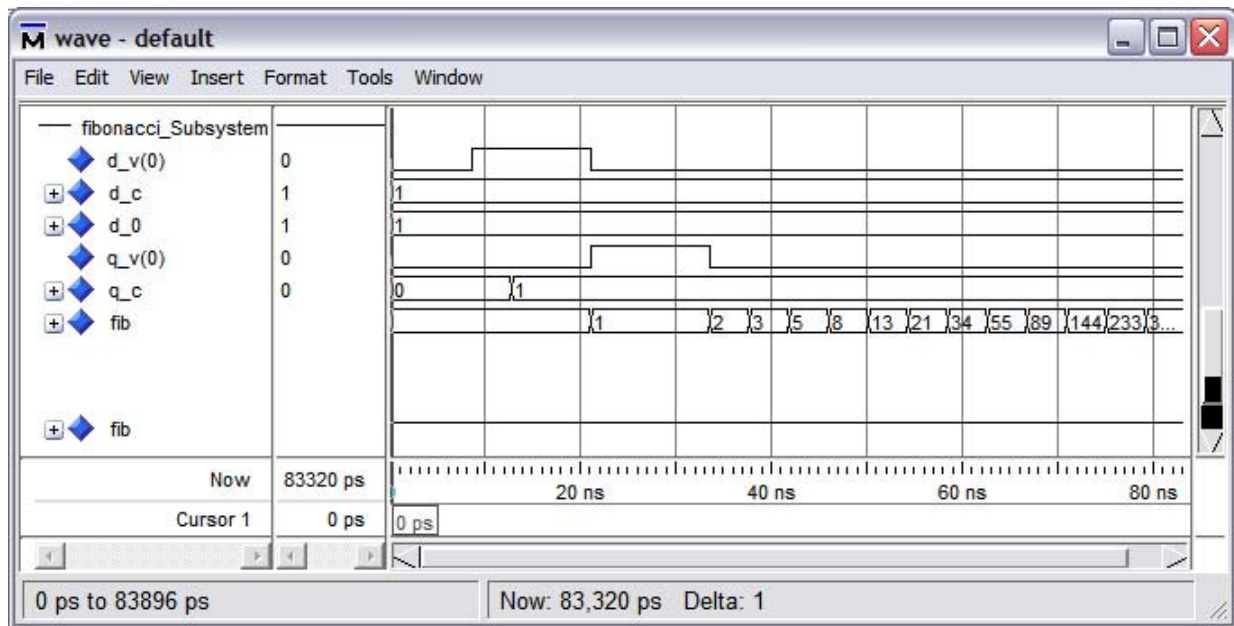
If you turn on the **Generate Hardware** option in the parameters for the **Control** block, every time the simulation runs, the underlying hardware synthesizes, and VHDL writes out into the specified directory.

DSP Builder creates a directory structure that mirrors the structure of your design. The root to this directory can be an absolute path name or a relative path name. For a relative path name (such as `../rtl`), DSP Builder creates the directory structure relative to the MATLAB current directory.

Simulating the RTL

To verify that DSP Builder gives the same results when you simulate the generated RTL, click on the Run ModelSim block (Figure 5-6).

Figure 5-6. Fibonacci Sequence in the ModelSim Wave Window



Compiling with the Quartus II Software

To load your design in the Quartus II software, double-click on the Run Quartus II block and compile your design by clicking **Start Compilation** on the Processing menu.

The Altera DSP Builder advanced blockset provides a variety of design examples, which you can use to learn from or use as a starting point for your own design.

All the design examples have the same basic structure: a top-level testbench containing an instantiated functional subsystem, which represents the hardware design.

The testbench typically includes Simulink source blocks that generate the stimulus signals and sink blocks that display simulation results. You can use other Simulink blocks to define the testbench logic.

The testbench also includes the following blocks from the DSP Builder advanced blockset:

- The `Control` block specifies information about the hardware generation environment, and the top-level memory-mapped bus interface widths.
- The `Signals` block specifies information about the clock, reset, and memory bus signals that the simulation model and the hardware generation use.
- The `Edit Params` block provides direct access to a MATLAB M-script that sets up MATLAB workspace variables that parameterize each block. The `PreLoadFcn` and `InitFcn` callbacks specify the script, which you can access by clicking **Model Properties** on the File menu and is usually named `setup_<demo name>.m`.
- The `Run ModelSim` block loads the testbench into the ModelSim simulator.
- The `Run Quartus II` block loads the system into the Quartus II software.
- The `ChanView` block in a testbench allows you to visualize the contents of the TDM protocol. This block generates synthesizable HDL and can therefore also be useful in a functional subsystem.

The functional subsystem in each design contains a `Device` block that marks the top-level of the FPGA device and controls the target device for the hardware.



For more information about these blocks, refer to the *DSP Builder Advanced Blockset Libraries* section in volume 3 of the *DSP Builder Handbook*.

Opening a Design Example

To view all the design examples and open one, perform the following steps:

1. Start DSP Builder: on Windows OS, click on **Start**, point to **All Programs**, click **Altera <version>**, click **DSP Builder**, and click **Start in MATLAB version XX**.
2. To view the design examples:
 - a. In MATLAB, click the **Help** icon and click **Supplemental software**.
 - b. Alternatively, at the MATLAB command prompt type one of the following commands:

```
demo blockset 'Altera DSP Builder Advanced Blockset'
```

```
demo blockset 'Altera DSP Builder Standard Blockset'
```

3. Expand **Examples**.
4. Expand **DSP Builder Advanced Blockset**. The design examples are in the following folders:
 - **Base Blocks**
 - **FFT**
 - **Filters**
 - **Folding**
 - **Floating Point**
 - **Flow Control**
 - **Host Interface**
 - **Platforms**
 - **Primitive Blocks**
 - **Reference Designs**
 - **Waveform Synthesis**
5. To see the floating-point design examples, expand **Floating Point**.
6. Click on a design example to see a description.
7. In the top right corner of the description, click **Open this model**, to open the design example.



In MATLAB 2014B you must double click **Open this model**.

8. You can also open a design example by typing a command in the MATLAB window, for example:

```
open_system('demo_nco'); ←
```

or:

```
demo_nco ←
```

Copying a Design Example

To make a copy of a design example, ensure that you also copy any corresponding **setup_<name>.m** file that has the same name as the **<name>.mdl** file.

You may also want to change the location of the generated files that the Control block specifies for each design (which is set to the relative path **../rtl** by default).

Running a Design Example

You can perform the following actions on design examples:

- To display help for each block in a design example, right-click and select **Help**.
- To run a design example, on the Simulation menu, click **Start Simulation**.

- To view resource usage for the design example, the subsystem, or a selected block, on the DSP Builder menu, point to **Resource Usage** and click **Design**, **Current Subsystem**, or **Selected block**.

Base Blocks

This folder contains design examples of the Scale and the LocalThreshold block:

- **Scale** (**demo_scale**)
- **Local Threshold** (**demo_nco_threshold**)

Scale

[Open this model](#)

This design example demonstrates the Scale block. The testbench allows you to see a vectorized block in action. Displays in the testbench track the smallest and largest values to be scaled and verify the correct behavior of the saturation modes.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a ChanView block that deserializes the output bus.

The **ScaleSystem** subsystem includes the Device and Scale blocks.

The model file is **demo_scale.mdl**.

Local Threshold

[Open this model](#)

This design example has two identical NCOs—one is in a subsystem with a LocalThreshold block that is set to force soft rather than hard multipliers. After simulation, in the resource table, you can compare the resources for NCO and NCO1. NCO1 uses no multipliers at the expense of extra logic. The resource table also contains resources for the ChannelViewer blocks—synthesizable blocks, that the design example uses outside the device system.

The model file is **demo_nco_threshold.mdl**.

FFT

This folder contains the following FFT design examples:

- **1K Floating-Point FFT** (**demo_fpfft_1024_mr42**)
- **FFT** (**demo_fft**)
- **FFT without BitReverseCoreC Block**
- **IFFT** (**demo_ifft**)
- **IFFT without BitReverseCoreC Block**
- **Floating-Point FFT** (**demo_fpfft**)
- **Floating-Point FFT without BitReverseCoreC Block**
- **Floating-Point iFFT** (**demo_fpifft**)

- Floating-Point iFFT without BitReverseCoreC Block
- Multichannel FFT
- Multiwire Transpose
- Parallel FFT (**demo_parallel_fft**)
- Parallel Floating-Point FFT (**demo_parallel_fft_fp**)
- Radix 2 Streaming FFT (**demo_fft16_radix2**)
- Radix 4 Streaming FFT (**demo_fft256_radix4**)
- Single-Wire Transpose
- Switchable FFT/iFFT
- Variable-Size Fixed-Point FFT
- Variable-Size Fixed-Point FFT without BitReverseCoreC Block
- Variable-Size Fixed-Point iFFT
- Variable-Size Fixed-Point iFFT without BitReverseCoreC Block
- Variable-Size Floating-Point FFT
- Variable-Size Floating-Point FFT without BitReverseCoreC Block
- Variable-Size Floating-Point iFFT
- Variable-Size Floating-Point iFFT without BitReverseCoreC Block
- Variable-Size Low-Resource FFT
- Variable-Size Low-Resource Real-Time FFT

1K Floating-Point FFT

[Open this model](#)

The 1K floating -point FFT design example implements a 1,024 point, variable-streaming, mixed-radix 4/2 floating-point FFT. This design example accepts natural order data at the input and produces radix-reversed (not bit-reversed) data at the output.



The design example has no bit reversal reordering stage. DSP Builder does not output the spectrum components in order of frequency.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the **setup_demo_fpffft_1024_mr42.m** script.

The example uses single-precision complex numbers format for its input and output format. The example has one configuration input, **length**, that can accept value of 16, 32, 64, 128, 256, 512 or 1024, which indicates the number of FFT points to perform.

The **FFT_1K** subsystem includes the Device block and a lower-level **FFTChip** subsystem.

The **FFTChip** subsystem comprises a StageSelect subsystem and five stages of FFT processing, with ChannelIn, ChannelOut, and SynthesisInfo blocks.

The model file is **demo_fpfft_1024_mr42.mdl**.

FFT

[Open this model](#)

This design example implements a 2,048 point, radix 2^2 FFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example includes a `BitReverseCoreC` block, which converts the input data stream from natural order to bit-reversed order, and an `FFT` block, which performs an FFT on bit-reversed data and produces its output in natural order.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 19 for this design example. You can also set the maximum width in bits by setting the *MaxOut* variable. Most applications do not need the maximum width in bits. To save resources, set a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The model file is **demo_fft.mdl**.

FFT without BitReverseCoreC Block

[Open this model](#)

This design example implements a 2,048 point, radix 2^2 FFT. This design example accepts natural order or bit-reversed data at the input and produces bit-reversed or natural order data at the output, respectively. The design example is identical to the FFT design example, but it does not include a `BitReverseCoreC` block, which converts the input data stream from natural order to bit-reversed order.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 19 for this design example. You can also set the maximum width in bits by setting the *MaxOut* variable. Most applications do not need the maximum width in bits. To save resources, set a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The model file is **demo_fft_core.mdl**.

IFFT

[Open this model](#)

This design example implements a 2,048 point, radix 2^2 iFFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example includes a `BitReverseCoreC` block, which converts the input data stream from natural order to bit-reversed order, and an `FFT` block, which performs an FFT on bit-reversed data and produces its output in natural order.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 19 for this design example. To set the maximum width in bits, set the *MaxOut* variable. Most applications do not need the maximum width in bits. To save resources, set a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The model file is **demo_ifft.mdl**.

IFFT without BitReverseCoreC Block

[Open this model](#)

This design example implements a 2,048 point, radix 2^2 iFFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example is identical to the iFFT design example, but it does not include a *BitReverseCoreC* block, which converts the input data stream from natural order to bit-reversed order.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 19 for this design example. To set the maximum width in bits, set the *MaxOut* variable. Most applications do not need the maximum width in bits. To save resources, set a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The model file is **demo_ifft_core.mdl**.

Floating-Point FFT

[Open this model](#)

This design example implements a floating-point, 512 point, radix 2^2 FFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example includes a *BitReverseCoreC* block, which converts the input data stream from natural order to bit-reversed order, and an *FP_FFT* block, which performs an FFT on bit-reversed data and produces its output in natural order.

The model file is **demo_fpfft.mdl**.

Floating-Point FFT without BitReverseCoreC Block

[Open this model](#)

This design example implements a floating-point, 512 point, radix 2^2 FFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example is identical to the floating-point FFT design example, but it does not include a *BitReverseCoreC* block, which converts the input data stream from natural order to bit-reversed order.

The model file is **demo_fpfft_core.mdl**.

Floating-Point iFFT

[Open this model](#)

This design example implements a floating-point 512 point, radix 2^2 iFFT. This design example accepts bit-reversed order data at the input and produces natural order data at the output. The design example includes a BitReverseCoreC block, which converts the input data stream from natural order to bit-reversed order, and an FP_FFT block, which performs an FFT on bit-reversed data and produces its output in natural order.

The model file is **demo_fpifft.mdl**.

Floating-Point iFFT without BitReverseCoreC Block

[Open this model](#)

This design example implements a floating-point 512 point, radix 2^2 iFFT. This design example accepts natural order data at the input and produces natural order data at the output. The design example is identical to the floating-point iFFT design example, but it does not include a BitReverseCoreC block, which converts the input data stream from natural order to bit-reversed order.

The model file is **demo_fpifft_core.mdl**.

Multichannel FFT

[Open this model](#)

The FFT processes the input in blocks of 2K points. Each block contains 4 interleaved sub-channels. An i512-point FFT is performed on each sub-channel.

The model file is **demo_fft_multichannel.mdl**.

Multiwire Transpose

[Open this model](#)

This design example demonstrates how to use the MultiwireTranspose block.

The model file is **demo_multiwiretranspose.mdl**.

Parallel FFT

[Open this model](#)

The model file is **demo_parallel_fft.mdl**.

Parallel Floating-Point FFT

[Open this model](#)

The model file is **demo_parallel_fpfft.mdl**.

Radix 2 Streaming FFT

[Open this model](#)

This design example implements a complex, radix-2, streaming, fast Fourier transform.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks. An `Edit Params` block allows easy access to the setup variables in the `setup_demo_fft16_radix2.m` script.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 15 for this design example.

The DUT subsystem includes the `Device` block and a lower level **FFTChip** subsystem.

The **FFTChip** subsystem uses the `ComplexMult`, `Complex SampleDelay`, and `BFI` blocks from the FFT library. It also includes `ChannelIn`, `ChannelOut`, `Counter`, `BitExtract`, `Lut`, `Convert`, `SampleDelay`, `DualMem`, and `SynthesisInfo` blocks.

Also a **BitReverse** subsystem exists that is very similar to the more general purpose `BitReverseCore` masked subsystem in the FFT library, but without the channel signal.

The model file is `demo_fft16_radix2.mdl`.

Radix 4 Streaming FFT

[Open this model](#)

This design example implements a 256 point, single channel, radix 4, complex fast Fourier transform.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks. An `Edit Params` block allows easy access to the setup variables in the `setup_demo_fft256_radix4.m` script.



The FFT designs do not inherit width in bits and scaling information. The design example specifies these values with the *Wordlength* and *FractionLength* variables in the setup script, which are 16 and 15 for this design example.

The DUT subsystem includes the `Device` block and a lower level **FFTChip** subsystem.

The **FFTChip** subsystem uses the `ComplexMult`, `Complex SampleDelay`, `BFI`, and `BFI` blocks from the FFT library. It also includes `ChannelIn`, `ChannelOut`, `Counter`, `BitExtract`, `Lut`, `SampleDelay`, `Convert`, `DualMem`, and `SynthesisInfo` blocks. Also, a **BitReverse** subsystem exists that is very similar to the more general purpose `BitReverseCore` masked subsystem in the FFT library, but without the channel signal.

The model file is `demo_fft256_radix4.mdl`.

Single-Wire Transpose

[Open this model](#)

This design example demonstrates how to use the `Transpose` block.

The model file is `demo_transpose.mdl`.

Switchable FFT/iFFT

[Open this model](#)

This design example demonstrates a switchable FFT block.

The model file is `demo_dynamic_fft.mdl`.

Variable-Size Fixed-Point FFT

[Open this model](#)

This design example demonstrates the FFT block.

The model file is `demo_vfft.mdl`.

Variable-Size Fixed-Point FFT without BitReverseCoreC Block

[Open this model](#)

This design example demonstrates the FFT block.

The model file is `demo_vfft_core.mdl`.

Variable-Size Fixed-Point iFFT

[Open this model](#)

This design example demonstrates the iFFT block.

The model file is `demo_vifft.mdl`.

Variable-Size Fixed-Point iFFT without BitReverseCoreC Block

[Open this model](#)

This design example demonstrates the iFFT block.

The model file is `demo_vifft_core.mdl`.

Variable-Size Floating-Point FFT

[Open this model](#)

This design example demonstrates the FFT block.

The model file is `demo_fpvfft.mdl`.

Variable-Size Floating-Point FFT without BitReverseCoreC Block

[Open this model](#)

This design example demonstrates the FFT block.

The model file is `demo_fpvfft_core.mdl`.

Variable-Size Floating-Point iFFT

[Open this model](#)

This design example demonstrates the iFFT block.

The model file is **demo_fpvifft.mdl**.

Variable-Size Floating-Point iFFT without BitReverseCoreC Block

[Open this model](#)

This design example demonstrates the iFFT block.

The model file is **demo_fpvifft_core.mdl**.

Variable-Size Low-Resource FFT

[Open this model](#)

This design example implements a low data-rate, low-resource usage FFT suitable for applications such as vibration suppression. The maximum size of the FFT is 4K points. The actual size of each FFT iteration may be any power of 2 that is smaller than 4K points. You dynamically control the size using the size input. Irrespective of the size of the FFT, a new FFT iterates whenever it receives an additional 64 inputs. Successive FFT iterations usually use overlapping input data.

The FFT accepts 16-bit fixed-point inputs. The FFT produces block floating-point output using an 18-bit mantissa and a shared 6-bit exponent.

This FFT implementation is unusual because the FFT gain is 1. Therefore the sum-of-squares of the input values is equal (allowing for rounding errors) to the sum-of-squares of the output values.

This method of scaling gives two advantages:

- The exponent can be smaller.
- The output remains consistently scaled when the FFT dynamic size changes.

To configure the design example, edit any of the parameters in the setup file.

The model file is **demo_servofft.mdl**.

Variable-Size Low-Resource Real-Time FFT

[Open this model](#)

This design example is an extension of the Variable-Size Low-Resource FFT design example. This example runs the Variable-Size Low-Resource FFT in Simulink in real time alongside the DSP System Toolbox FFT block. It produces vector scope outputs that are the same (or as near as) from both.

This design example takes care of the faster sample rate needed by the DSP Builder FFT. The setup file chooses a sample rate that is fast enough for calculation but not so fast that it slows down the simulation unnecessarily. The design also adds buffering to the original MATLAB fft signal path to make the signal processing delays the same in both paths.

The model file is **demo_dspba_ex_fft_tut.mdl**.

Filters

This folder contains the following design examples of cascaded integrator-comb (CIC) and finite impulse response (FIR) filters:

- Complex FIR Filter
- Decimating CIC Filter (**demo_dcic**)
- Decimating FIR Filter (**demo_fird**)
- Filter Chain with Forward Flow Control (**demo_filters_flow_control**)
- FIR Filter with Exposed Bus (**demo_fir_exposed_bus**)
- Fractional FIR Filter Chain (**demo_fir_fractional**)
- Fractional-Rate FIR Filter (**demo_firf**)
- Half-Band FIR Filter (**demo_firih**)
- Interpolating CIC Filter (**demo_icic**)
- Interpolating FIR Filter (**demo_firi**)
- Interpolating FIR Filter with Multiple Coefficient Banks (**demo_firi_multibank**)
- Interpolating FIR Filter with Updating Coefficient Banks
- Root-Raised Cosine FIR Filter (**demo_fir_rrc**)
- Single-Rate FIR Filter (**demo_firs**)
- Super-Sample Decimating FIR Filter (**demo_ssfire**)
- Super-Sample Fractional FIR Filter (**demo_ssfiref**)
- Super-Sample Interpolating FIR Filter (**demo_ssfirei**)
- Variable-Rate CIC Filter (**demo_vcic**)

Complex FIR Filter

[Open this model](#)

This design example implements a complex FIR filter that demonstrates how to implement a complex FIR filter using three real filters. The resource efficient implementation (three real multipliers per complex multiply) maps optimally onto Arria V DSP blocks, using the scan and cascade modes.

The model file is **demo_complex_fir.mdl**.

Decimating CIC Filter

[Open this model](#)

This design example implements a decimating CIC filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_dcic.m** script.

The **CICSystem** subsystem includes the Device and DecimatingCIC blocks.

The model file is **demo_dcic.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Decimating FIR Filter

[Open this model](#)

This design example uses the Decimating FIR block to build a 20-channel decimate by 5, 49-tap FIR filter with a target system clock frequency of 240 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_fird.m** script.

The **FilterSystem** subsystem includes the Device and Decimating FIR blocks.

The model file is **demo_fird.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Filter Chain with Forward Flow Control

[Open this model](#)

This design example builds a filter chain with forward flow control.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_filters_flow_control.m** script.

The **FilterSystem** subsystem includes FractionalRateFIR, InterpolatingFIR, InterpolatingCIC, Const and Scale blocks.

The model file is **demo_filters_flow_control.mdl**.



This design example uses the Simulink Signal Processing Blockset.

FIR Filter with Exposed Bus

[Open this model](#)

This design example is a multichannel single-rate FIR filter with rewritable coefficients. The initial configuration is a high-pass filter, but halfway through the testbench simulation, DSP Builder reconfigures it as a low-pass filter. The testbench feeds in the sum of a fast and a slow sine wave into the filter. The fast one emerges from the originally configured FIR filter; the slow one is all that is left when DSP Builder reconfigures the filter.

The model file is **demo_fir_exposed_bus.mdl**.

Fractional FIR Filter Chain

[Open this model](#)

This design example uses a chain of InterpolatingFIR and DecimatingFIR blocks to build a 16-channel fractional rate filter with a target system clock frequency of 360 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_fir_fractional.m** script.

The **FilterSystem** subsystem includes ChanView, Decimating FIR, InterpolatingFIR, and Scale blocks.

The model file is **demo_fir_fractional.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Fractional-Rate FIR Filter

[Open this model](#)

This design example implements a fractional rate FIR filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_firf.m** script.

The **FilterSystem** subsystem includes the Device and FractionalRateFIR blocks.

The model file is **demo_firf.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Half-Band FIR Filter

[Open this model](#)

This design example implements a half band interpolating FIR filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_firih.m** script.

The **FilterSystem** subsystem includes the Device block and two separate InterpolatingFIR blocks for the regular and interpolating filters.

The model file is **demo_firih.mdl**.

This design example uses the Simulink Signal Processing Blockset.

Interpolating CIC Filter

[Open this model](#)

This design example implements an interpolating CIC filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_icic.m** script.

The **FilterSystem** subsystem includes the Device and InterpolatingCIC blocks.

The model file is **demo_icic.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Interpolating FIR Filter

[Open this model](#)

This design example uses the InterpolatingFIR block to build a 16-channel interpolate by 2, symmetrical, 49-tap FIR filter with a target system clock frequency of 360 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_fir.m** script.

The **FilterSystem** subsystem includes the Device and InterpolatingFIR blocks.

The model file is **demo_fir.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Interpolating FIR Filter with Multiple Coefficient Banks

[Open this model](#)

This design example builds an interpolating FIR filter that regularly switches between coefficient banks.

Multiple sets of coefficients requires storage in memory so that the design can switch easily from one set or bank of coefficients in use to another in a single clock cycle.

The design must perform the following actions:

- Specify the number of coefficient banks
- Initialize the banks
- Update the coefficients in a particular bank
- Select the bank in use in the filter

You specify the coefficient array as a matrix rather than a vector—(bank rows) by (number of coefficient columns).

The addressing scheme has address offsets of base address + bank number * number of coefficients for each bank.

If the number of rows is greater than one, DSP Builder creates a bank select input port on the FIR filter. In a design, you can drive this input from either data or bus interface blocks, allowing either direct or bus control. The data type is unsigned integer of width $\text{ceil}(\log_2(\text{number of banks}))$.

The bank select is a single signal. For example, for a FIR filter with four input channels over two timeslots:

<0><1>

<2><3>

The corresponding input channel signal is:

```
<0><1>
```

Here the design receives more than one channel at a time, but can only choose a single bank of coefficients. Channels 0 and 2 use one set of coefficients and channels 1 and 3 another. Channel 0 cannot use a different set of coefficients to channel 2 in the same filter.

For multiple coefficient banks, you enter an array of coefficients sets, rather than a single coefficient set. For example, for a MATLAB array of 1 row and 8 columns [1 x 8], enter:

```
fi(fir1(7, 0.5 ),1,16,15)
```

For a a MATLAB array of 2 rows and 8 columns [2 x 8] enter:

```
[fi(fir1(7, 0.5 ),1,16,15);fi(fir1(7, 0.5 ),1,16,15)]
```

The number of banks can therefore be determined by the number of rows without needing the number of banks. If the number of banks is greater than 1, add an additional bank select input on the block.

The model file is **demo_fir_multibank.mdl**.

Interpolating FIR Filter with Updating Coefficient Banks

[Open this model](#)

This design example is similar to the Interpolating FIR Filter with Multiple Coefficient Banks design example. While one bank is in use DSP Builder writes a new set of FIR filter coefficients to the other bank. You can see the resulting change in the filter output when the bank select switches to the updated bank.

Write to the bus interface using the BusStimulus block with a sample rate proportionate with the bus clock. Generally, DSP Builder does not guarantee bus interface transactions to be cycle accurate in Simulink simulations. However, in this design example, DSP Builder updates the coefficient bank while it is not in use.

Root-Raised Cosine FIR Filter

[Open this model](#)

This design example uses the Decimating FIR block to build a 4-channel decimate by 5, 199-tap root raised cosine filter with a target system clock frequency of 304 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_fir_rrc.m** script.

The **FilterSystem** subsystem includes the Device and Decimating FIR blocks.

The model file is **demo_fir_rrc.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Single-Rate FIR Filter

[Open this model](#)

This design example uses the `SingleRateFIR` block to build a 16-channel single rate 49-tap FIR filter with a target system clock frequency of 360 MHz.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks, plus `ChanView` block that deserialize the output buses. An `Edit Params` block allows easy access to the setup variables in the `setup_demo_firs.m` script.

The **FilterSystem** subsystem includes the `Device` and `SingleRateFIR` blocks.

The model file is `demo_firs.mdl`.



This design example uses the Simulink Signal Processing Blockset.

Super-Sample Decimating FIR Filter

[Open this model](#)

This design example shows how the filters cope with data rates greater than the clock rate. The design example uses the `DecimatingFIR` block to build a single channel decimate by 2, symmetrical, 33-tap FIR filter. The input sample rate is six times the clock rate. The filter decimates by two the input sample rate to three times the clock rate, which is visible in the vector input and output data connections. The input receives six samples in parallel at the input, and three samples are output each cycle.

After simulation, you can view the resource usage.

The model file is `demo_ssfirst.mdl`.

Super-Sample Fractional FIR Filter

[Open this model](#)

This design example shows how the filters cope with data rates greater than the clock rate. The design example uses the `FractionalFIR` block to build a single channel interpolate by 3, decimate by 2, symmetrical, 33-tap FIR filter. The input sample rate is two times the clock rate. The filter upconverts the input sample rate to three times the clock rate, which is visible in the vector input and output data connections. The input receives two samples in parallel at the input, and three samples are output each cycle.

The model file is `demo_ssfirst.mdl`.

Super-Sample Interpolating FIR Filter

[Open this model](#)

This design example shows how the filters cope with data rates greater than the clock rate. The design example uses the `InterpolatingFIR` block to build a single channel interpolate by 3, symmetrical, 33-tap FIR filter. The input sample rate is twice the clock rate and is interpolated by three by the filter to six times the clock rate, which is visible in the vector input and output data connections. The input receives two samples in parallel at the input, and six samples are output each cycle.

After simulation, you can view the resource usage.

The model file is `demo_ssfirst.mdl`.



This design example uses the Simulink Signal Processing Blockset.

Variable-Rate CIC Filter

[Open this model](#)

CIC filters are extremely hardware efficient, as they require no multipliers. You see CIC filters commonly in applications that require large interpolation and decimation factors. Usually the interpolation and decimation factors are fixed, and you can use the CIC ModelIP block. However, a subset of applications require the interpolation and decimation factors to be change at run time.

This design example shows how to build a variable-rate CIC filter from primitives. It contains a variable-rate decimating CIC filter, which consists of a number of integrators and differentiators with a decimation block between them, where the rate change occurs. You can control the rate change with a register field, which is part of the control interface. The register field controls the generation of a valid signal that feeds into the differentiators.

The design example also contains a gain compensation block that compensates for the rate change dependent gain of the CIC. It shifts the input up so that the MSB at the output is always at the same position, regardless of the rate change that you select.

The associated setup file contains parameters for the minimum and maximum decimation rate, and calculates the required internal data widths and the scaling number. To change the decimation factor for simulation, adjust variable `CicDecRate` to the desired current decimation rate.

The model file is **demo_vcic.mdl**.

Folding

This folder contains the following design examples:

- [Folded FIR Filter](#)
- [Position, Speed, and Current Control for AC Motors](#)
- [Position, Speed, and Current Control for AC Motors \(with ALU Folding\)](#)

Position, Speed, and Current Control for AC Motors

[Open this model](#)

This design example implements a field-oriented control (FOC) algorithm for AC motors such as permanent magnet synchronous machines (PMSM). Industrial servo motors, where the precise control of torque is important, commonly use these algorithms. This design example includes position and speed control, which allow the control of rotor speed and angle.



Altera has not tested this design on hardware and Altera does not provide a model of a motor.

The model file is **psc_ctrl.mdl**. Also, an equivalent fixed-point design, **psc_ctrl_fixed.mdl**, exists. To change the precision this design uses, refer to the **setup_position_speed_current_controller_fixed.m** script.

About FOC

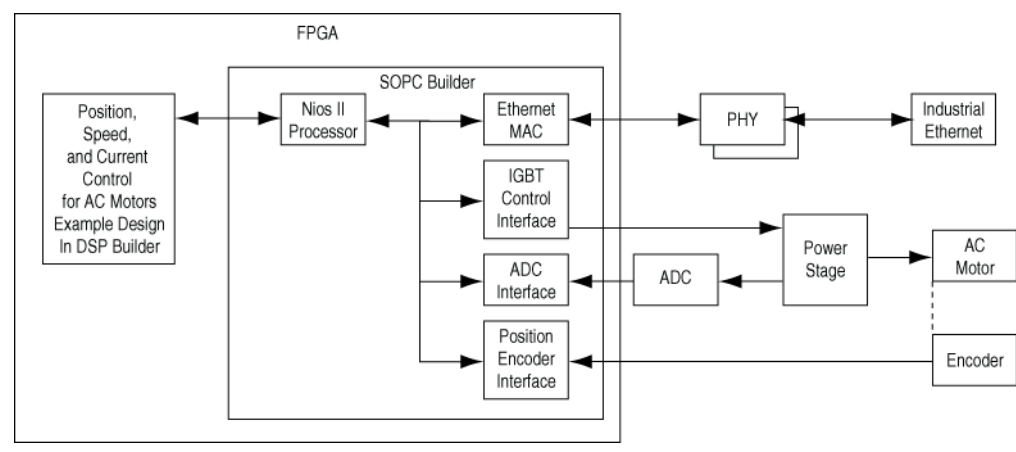
FOC involves controlling the motor's sinusoidal 3-phase currents in real time, to create a smoothly rotating magnetic flux pattern, where the frequency of rotation corresponds to the frequency of the sine waves. FOC controls the amplitude of the current vector that is at 90 degrees with respect to the rotor magnet flux axis (*quadrature* current) to control torque. The *direct* current component (0 degrees) is set to zero. The algorithm involves the following steps:

- Converting the 3-phase feedback current inputs and the rotor position from the encoder into quadrature and direct current components with the Clarke and Park transforms.
- Using these current components as the inputs to two proportional and integral (PI) controllers running in parallel to control the direct current to zero and the quadrature current to the desired torque.
- Converting the direct and quadrature current outputs from the PI controllers back to 3-phase currents with inverse Clarke and Park transforms.

Functional Description

An encoder measures the rotor position in the motor, which the FPGA then reads. An analog-to-digital converter (ADC) measures current feedback, which the FPGA then reads. Figure 6-1 shows a block diagram of a AC motor control system.

Figure 6-1. AC Motor Control System Block Diagram



Each of the FOC, speed, and position feedback loops use a simple PI controller to reduce the steady state error to zero. In a real-world PI controller, you may also need to consider integrator *windup* and tune the PI gains appropriately. The feedback loops for the integral portion of the PI controllers are internal to the DSP Builder advanced system (they were external in DSP Builder v11.0).

The example assumes you sample the inputs at a rate of 100 kHz and the FPGA clock rate is 100 MHz (suitable for Cyclone IV devices). ALU folding reduces the resource usage by sharing operators such as adders, multipliers, cosine. The folding factor is set to 100 to allow each operator to be timeshared up 100 times, which gives an input sample rate of 1 Msps, but as the real input sample rate is 100 ksps, only one out of every ten input timeslots are used. DSP Builder identifies the used timeslots when

`valid_in` is 1. Use `valid_in` to enable the latch in the PI controller, which stores data for use in the next valid timeslot. The `valid_out` signal indicates when the `ChannelOut` block has valid output data. You can calculate nine additional channels on the same DSP Builder advanced system without incurring extra latency (or extra DSP Builder advanced resources).

You should adjust the folding factor to see the effect it has on hardware resources and latency. To adjust, change the **Sample rate (MHz)** parameter in the `ChannelIn` and `ChannelOut` blocks of the DSP Builder advanced subsystem either directly or change the `FoldingFactor` parameter in the setup script. For example, a clock frequency of 100 MHz and sample rate of 10 MHz gives a folding factor of 10. Disabling folding, or setting the factor to 1, results in no resource sharing and minimal latency. Generally, you should not set the folding factor greater than the number of shareable operators, that is, for 24 adders and 50 multipliers, use a maximum folding factor 50.



The testbench does not support simulations if you adjust the folding factor.

The control algorithm, with the FOC, position, speed, control loops, vary the desired position across time. The three control loops are parameterized with minimum and maximum limits, and PI values. These values are not optimized and are for demonstrations only.

Resource Usage

Table 6-1 shows some example resource usage with and without folding.

Table 6-1. Position, Speed, and Current Control for AC Motors Design Example Resource Usage

Folding Factor	Add and Sub Blocks	Mult Blocks	Cos Blocks	Latency
No folding	22	22	4	170
>22	1	1	1	279

The example uses floating-point arithmetic that automatically avoids arithmetic overflow, but you can implement it in a fixed-point design and tune individual accuracies while manually avoiding overflows.

Hardware Generation

When hardware generation is disabled, the Simulink system simulates the DSP Builder advanced system at the external sample rate of 100 kHz, so that it outputs a new value once every 100 kHz. When hardware generation is enabled, the DSP Builder advanced system simulates at the FPGA clock rate (100 MHz), which represents real-life latency clock delays, but it only outputs a new value every 100 kHz. This mode slows the system simulation speed greatly as the model is evaluated 1,000 times for every output.

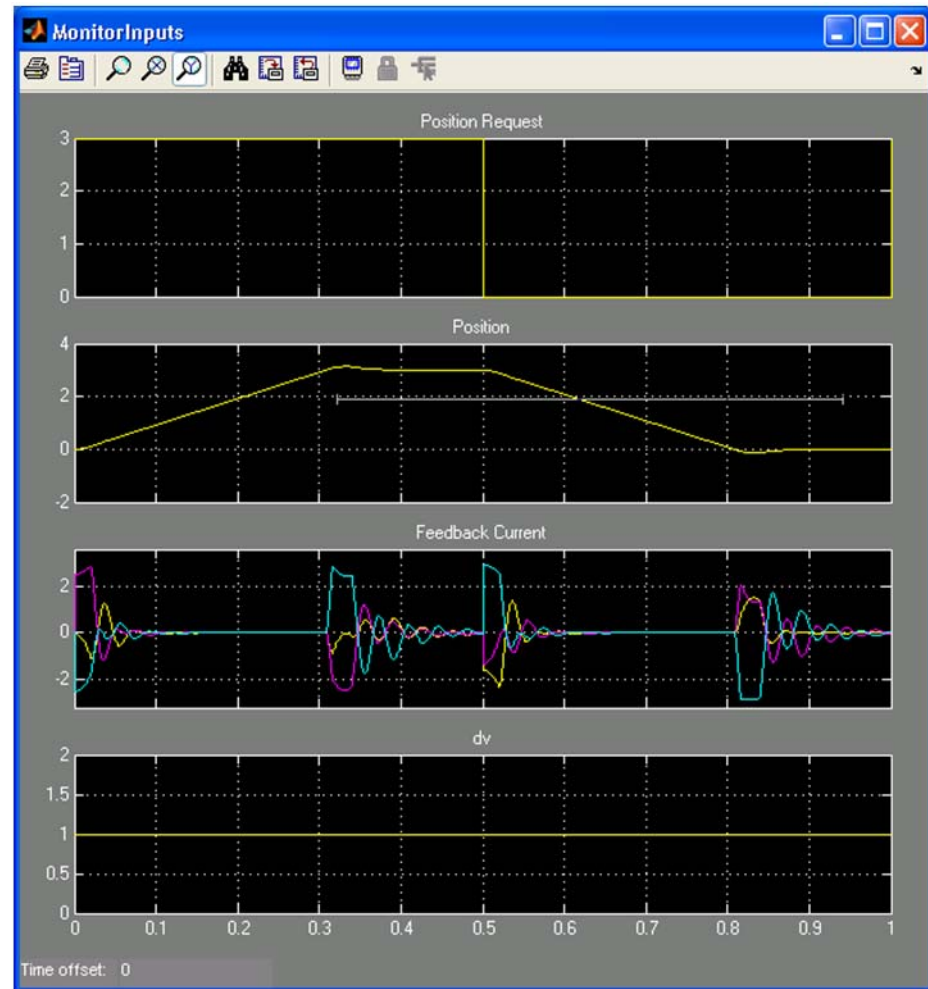
The setup script for the design example automatically detects whether hardware generation is enabled and sets the sample rates accordingly.

The example is configured with hardware generation disabled, which allows fast simulations. When you enable hardware generation, set a very small simulation time (for example 0.0001 s) as simulation may be very slow.

Input Position Request

Figure 6-2 shows the input position request. At 0 s, a position of 3 is requested and then at 0.5 s a position of 0 is requested. Figure 6-2 also shows the actual position and motor feedback currents.

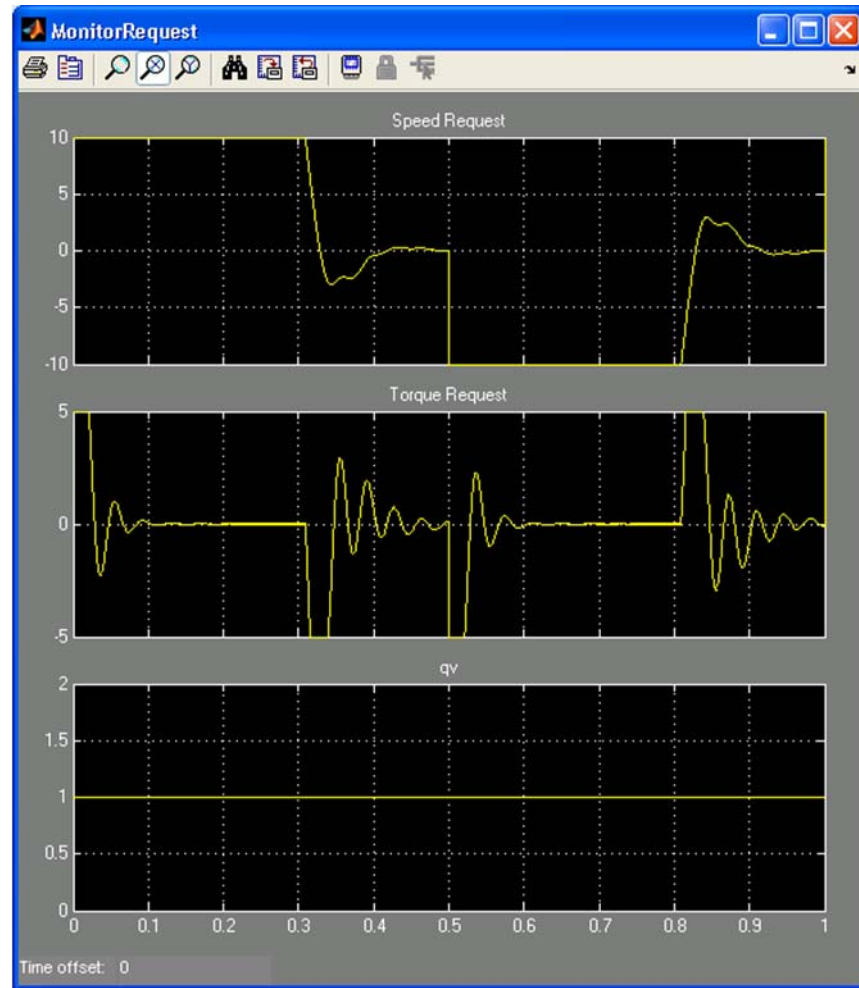
Figure 6-2. Input Position Request



Output Response for Speed and Torque

Figure 6-3 shows the output response for speed and torque. The maximum speed request saturates at 10 and the torque request saturates at 5 as set by parameters of the model. Also, some oscillation exists on the speed and torque requests because of nonoptimal settings for the PI controller causing an under-damped response.

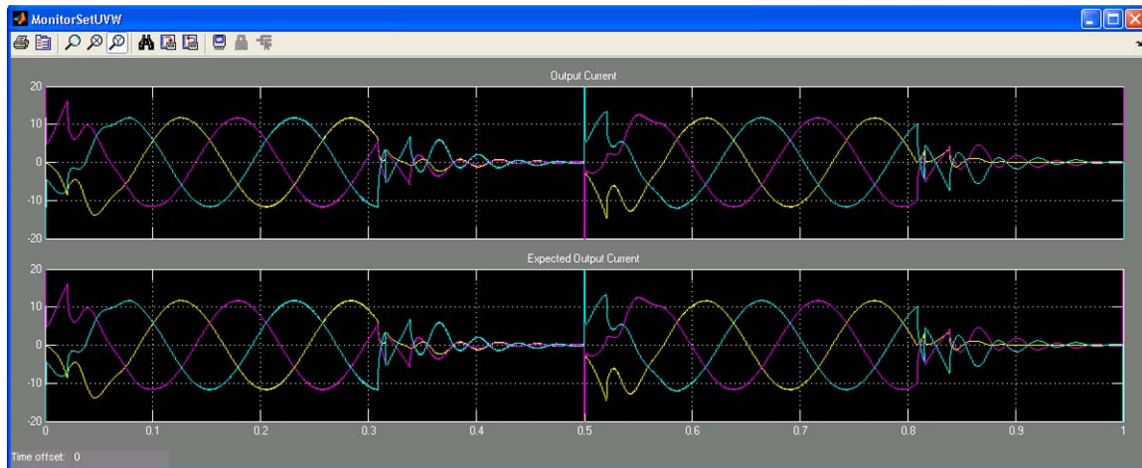
Figure 6-3. Output Response for Speed and Torque



Output Current

Figure 6-4 shows the output current. From 0 to 0.1, the motor is accelerating; 0.1 to 0.3, it is at a constant speed; 0.3 to 0.5, it is decelerating to stop. From 0.5 to 0.6, the motor accelerates in the opposite direction; from 0.6 to 0.8, it is at a constant speed; from 0.8 to 1, it is decelerating to stop.

Figure 6-4. Output Current



Position, Speed, and Current Control for AC Motors (with ALU Folding)

[Open this model](#)

The position, speed, and current control for AC motors (with ALU folding) design example is a FOC algorithm for AC motors, which is identical to the position, speed, and current control for AC motors design example. However this design example uses ALU folding.

For more information about ALU folding, refer to “[ALU Folding](#)” on page 10-15.

The model file is `psc_ctrl_alu.mdl`.

The design example targets a Cyclone V device (speed grade 8). Cyclone V devices have distributed memory (MLABs). ALU folding uses many distributed memory components. ALU folding performs better in devices that have distributed memories, rather than devices with larger block memories.

The design example includes a setup script `setup_position_speed_current_controller_alu.m`. [Table 6-2](#) lists some of the variables in the setup script.

Table 6-2. Setup Script

Variables	Description
<code>dspb_psc_ctrl.SampleRateHz = 10000</code>	Sample rate. Default set to 10000, which is 10 kHz sample rate.
<code>dspb_psc_ctrl.ClockRate = 100</code>	FPGA clock frequency. Default set to 100, which is 100 MHz clock
<code>dspb_psc_ctrl.LatencyConstraint = 1000</code>	Maximum latency. Default 1,000 clock cycles

This design example uses a significantly large maximum latency, so resource consumption is the factor to optimize in ALU folding rather than latency.

Generally, industrial designs require a testbench that operates in the real-world sample rate. This example emulates the behavior of a motor sending current, position, and speed samples at a rate of 10 kHz.

When you run this design example without folding, the DSP Builder system operates at the same 10 kHz sample rate. Therefore, the system calculates a new packet of data for every Simulink sample. Also, the sample times of the testbench are the same as the sample times for the DSP Builder system.

The Rate Transition blocks translate between the Simulink testbench and the DSP Builder system. These blocks allow Simulink to manage the different sample times that the DSP Builder system requires. You need not modify the design example when you run designs with or without folding.

The Rate Transition blocks produce Simulink samples with a sample time of `dspb_psc_ctrl.SampleTime` for the testbench and `dspb_psc_ctrl.DSPBASampleTime` for the DSP Builder system. The samples are in the stimuli system, within the dummy motor. To hold the data consistent at the inputs to the Rate Transition blocks for the entire length of the output sample (`dspb_psc_ctrl.SampleTime`), turn on **Register Outputs**.

The data valid signal consists of a one Simulink sample pulse that signifies the beginning of a data packet followed by zero values until the next data sample, as required by ALU folding. The design example sets the period of this pulsing data valid signal to the number of Simulink samples for the DSP Builder system (at `dspb_psc_ctrl.DSPBASampleTime`) between data packets. This value is `dspb_psc_ctrl.SampleTime/dspb_psc_ctrl.DSPBASampleTime`.

The verification script within ALU folding uses the To Workspace blocks. The verification script searches for To Workspace blocks on the output of systems to fold. The script uses these blocks to record the outputs from both the design example with and without folding. The script compares the results with respect to valid outputs. To run the verification script, enter the following command at the MATLAB prompt:

```
Folder.Testing.RunTest('psc_ctrl_alu');
```

Folded FIR Filter

[Open this model](#)

This design example implements a simple non-symmetric FIR filter using primitive blocks, with a data sample rate much less than the system clock rate. This design example uses ALU folding to minimize hardware resource utilization.

The model file is **demo_alu_fir.mdl**.

Floating Point

This folder contains the following design examples:

- [Black-Scholes Floating Point](#)
- [Fine Doppler Estimator](#)
- [General Real Matrix Multiply One Cycle Per Output](#)
- [Newton Root Finding Tutorial Step 1—Iteration](#)

- [Newton Root Finding Tutorial Step 2—Convergence](#)
- [Newton Root Finding Tutorial Step 3—Valid](#)
- [Newton Root Finding Tutorial Step 4—Control](#)
- [Newton Root Finding Tutorial Step 5—Final](#)
- [Floating-Point Mandelbrot Set](#)
- [Normalizer](#)
- [Single-Precision Complex Floating-Point Matrix Multiply](#)
- [Single-Precision Real Floating-Point Matrix Multiply](#)
- [Simple Nonadaptive 2D Beamformer](#)

Black-Scholes Floating Point

[Open this model \(single\)](#)

[Open this model \(double\)](#)

The Black-Scholes single- and double-precision floating-point design examples implement the calculation of a Black-Scholes equation and demonstrate the load exponent, reciprocal square root, logarithm and divide floating-point ModelPrim blocks for single- or double-precision floating-point designs.

The model files are `blackScholes_S.mdl` and `blackScholes_D.mdl`.

Double-Precision Real Floating-Point Matrix Multiply

[Open this model](#)

A simpler design example of a floating-point matrix multiply implementation than the complex multiply example. Each vector multiply is performed simultaneously, using many more multiply-adds in parallel.

The model file is `matmul_flash_RD.mdl`.

Fine Doppler Estimator

[Open this model](#)

The fine Doppler estimator design example is an interpolator for radar applications. The example has three complex input values. It calculates the magnitude of each value, then performs a parabolic curve fit, identifies the location of the peak, and calculates the peak magnitude. The example performs all processing in single-precision floating-point data.

For more information about fine Doppler estimators, refer to *Fundamentals of Radar Signal Processing* by Mark A. Richards, McGraw-Hill, ISBN 0-07-144474-2, ch. 5.3.4.

The model file is `FineDopplerEstimator.mdl`.

Floating-Point Mandelbrot Set

[Open this model](#)

This design example plots the Mandelbrot set for a defined region of the complex plane, shows many advanced blockset features, and highlights recommended design styles.

A complex number C is in the Mandelbrot set if for the following equation the value remains finite when repeatedly squared and added to the original number:

$$z(n+1) = zn^2 + C$$

where n is the iteration number and C is the complex conjugate

The latency of the system takes longer to perform floating-point calculations than for the corresponding fixed-point calculations. You cannot wait around for partial results to be ready, if you want to achieve maximum efficiency. Instead, you must ensure the floating-point math's calculation engines of your algorithm keep busy and fully used. DSP Builder provides two floating-point math subsystems: one for scaling and offsetting pixel indices to give a point in the complex plane; the other to perform the main square-and-add iteration operation.

For this design example, the total latency is approximately 19 clock cycles, depending on target device and clock speed. The latency is not excessive; but long enough that it is inefficient to wait for partial results.

FIFO buffers control the circulation of data through the iterative process. The FIFO buffers ensure that if a partial result is available for a further iteration in the $z(n+1) = zn^2 + C$ progression, DSP Builder works on that point.

Otherwise, DSP Builder starts a new point (new value of C). Thus, DSP Builder maintains a full flow of data through the floating-point arithmetic. This main iteration loop can exert back pressure on the new point calculation engine. If DSP Builder does not read new points off the command queue FIFO buffers quickly enough, such that they fill up, the loop iteration stalls. DSP Builder does not explicitly signal the calculation of each point when it is required (and thus avoid waiting through the latency cycles before you can use it). DSP Builder does not attempt to exactly calculate this latency in clock cycles and try to issue generate point commands the exact number of clock-cycles before you need them, as you must change them each time you retarget a device, or change target clock rate. Instead, DSP Builder calculates the points quickly from the start, catches them in a FIFO buffer. If the FIFO buffer starts to get full—a sufficient number of cycles ahead of full—DSP Builder stops the calculation upstream without loss of data. This flow is self regulating that mitigates latency while remaining flexible.

Avoid inefficiencies by designing algorithm implementation around the latency and availability of partial results. Data dependencies in processing can stall processing.

The design example has the following features:

- The design example uses the `FinishedThisPoint` signal as the valid signal. Although the system constantly produces data on the output, it marks the data as valid only when the design finishes a point. Downstream components can then just process valid data, just as the enabled subsystem in the testbench captures and plot the valid points.

- In both feedback loops, you must allow sufficient delay for the scheduler to redistribute as pipelining. In feed-forward paths you can add pipelining without changing the algorithm—only the timing of the algorithm. But in feedback loops, inserting a delay can alter the meaning of an algorithm. For example, adding N cycles of delay to an accumulator loop increments N different numbers, each incrementing every N clock cycles. In loops, the design example must give the scheduler in charge of pipelining for timing closure enough slack in the loop to be able to redistribute this delay to meet timing, while not changing the total latency around the loop, and thus ensuring the function of the algorithm is unaltered. Such slack delays are in the top-level design of the synthesizable design in the feedback loop controlling the generation of new points, and in the **FeedBackFIFO** subsystem controlling the main iteration calculation. DSP Builder uses the minimum delay feature on the `SampleDelay` blocks to set these slack delays to the minimum possible delay that satisfies the scheduling solver. The example sets the `SampleDelay` block to the minimum latency that satisfies the schedule, which the DSP Builder solves as part of the integer linear programming problem that finds an optimum pipelining and scheduling solution. You can group delays into numbered equivalence groups to match other delays. In this design example, the single delay around the coordinate generation loop is in one equivalence group, and all the slack delays round the main calculation loop are in another equivalence group. The equivalence group field allows any Matlab expression that evaluates to a string. The `SampleDelay` block displays the delay that DSP Builder uses.
- The FIFO buffers operate in show-ahead mode—they display the next value to be read. The read signal is a read acknowledgement, which reads the output value, discards it, and shows the next signal. DSP Builder uses multiple FIFO buffers with the same control, which are full and give a valid output at the same time. Thus we only need the output control signals from one of the FIFO buffers and can ignore the corresponding signals from the other FIFO buffers. As floating-point simulation is not bit accurate to the hardware, some points in the complex plane take fewer or more iterations to complete in hardware compared to the Simulink simulation. Thus the results, when you decide you are finished with a particular point, may come out in a different order. You must build a testbench mechanism that is robust to this feature. Use the testbench override feature in the `Run All Testbenches` block:
 - Set the condition on mismatches to **Warning**
 - Use the `Run All Testbenches` block to set an import variable, which brings the ModelSim results back into MATLAB and a custom verification function that sets the pass or fail criteria.

The model file is **Mandelbrot_S.mdl**.

```
>Newton Root Finding Tutorial Step 1 -
Iteration</label><file>demo_newton_iteration.html</file></demoitem>

<demoitem><label>Newton Root Finding Tutorial Step 2 -
Convergence</label><file>demo_newton_convergence.html</file></demoitem>

<demoitem><label>Newton Root Finding Tutorial Step 3 -
Valid</label><file>demo_newton_valid.html</file></demoitem>
```



```
<demoitem><label>Newton Root Finding Tutorial Step 4 -  
Control</label><file>demo_newton_control.html</file></demoitem>  
  
<demoitem><label>Newton Root Finding Tutorial Step 5 -  
Final</label><file>demo_newton_final.html</file></demoitem>
```

General Real Matrix Multiply One Cycle Per Output

[Open this model](#)

This design example implements a floating-point matrix multiply. The design performs each vector multiply simultaneously, using many multiply-adds in parallel.

The model file is `gemm_flash.mdl`.

Newton Root Finding Tutorial Step 1—Iteration

[Open this model](#)

This design example is part of the Newton-Raphson tutorial. It demonstrates a naive test for convergence and exposes problems with rounding and testing equality with zero.

The model file is `demo_newton_iteration.mdl`.

Newton Root Finding Tutorial Step 2—Convergence

[Open this model](#)

This design example is part of the Newton-Raphson tutorial. It demonstrates convergence criteria exposing mismatches between Simulink and ModelSim that you can correct by bit-accurate simulation. The discrepancies are worse when you use faithful rounding.

The model file is `demo_newton_convergence.mdl`.

Newton Root Finding Tutorial Step 3—Valid

[Open this model](#)

This design example is part of the Newton-Raphson tutorial. It demonstrates how you avoid having the same answer multiple times on the output. It introduces a valid control signal, parallel to the datapath, to keep track of which pipeline slots DSP Builder empties. It uses equivalence groups in the minimum `SampleDelay` blocks

The model file is `demo_newton_valid.mdl`.

Newton Root Finding Tutorial Step 4—Control

[Open this model](#)

This design example is part of the Newton-Raphson tutorial. It demonstrates flow control which allows DSP Builder to buffer inputs in a FIFO buffer and insert data into pipeline slots as they become available.

The model file is `demo_newton_control.mdl`.

Newton Root Finding Tutorial Step 5—Final

[Open this model](#)

This design example is part of the Newton-Raphson tutorial. It demonstrates a parallel integer datapath for counting iterations. It detects divergence in cases where the Newton method oscillates between two finite values.

The model file is **demo_newton_final.mdl**.

Normalizer

[Open this model](#)

The normalizer design example demonstrates the `ilogb` block and the multifunction `ldexp` block. The parameters allow you to select the `ilogb` or `ldexp`. The design example implements a simple floating-point normalization. The magnitude of the output is always in the range 0.5 to 1.0, irrespective of the (non-zero) input.

The model file is **demo_normalizer.mdl**.

Single-Precision Complex Floating-Point Matrix Multiply

[Open this model](#)

This design example uses a similar flow control style to that in the floating-point Mandelbrot set design example. The design example uses a limited number of multiply-adds, set by the vector size, to perform a complex single precision matrix multiply.

A matrix multiplication must multiply row and column dot product for each output element. For 8×8 matrices A and B:

Figure 6–5. Matrix Multiply Equation

$$AB_{ij} = \sum_{k=1}^8 A_{ik} \cdot B_{kj}$$

You may accumulate the adjacent partial results, or build adder trees, without considering any latency. However, to implement with a smaller dot product, use resource usage folding to use a smaller number of multipliers rather than performing everything in parallel, split up the loop over k into smaller chunks. Then reorder the calculations to avoid adjacent accumulations.

A traditional implementation of a matrix multiply design is structured around a delay line and an adder tree:

$$A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} \text{ and so on.}$$

The traditional implementation has the following features:

- The length and size grow with folding size (typically 8 to 12)
- Uses adder trees of 7 to 10 adders that are only used once every 10 cycles.
- Each matrix size needs different length, so you must provide for the worst case

A better implementation is to use FIFO buffers to provide self-timed control. New data is accumulated when both FIFO buffers have data. This implementation has the following advantages:

- Runs as fast as possible
- Is not sensitive to latency of dot product on devices or f_{MAX}
- Is not sensitive to matrix size (hardware just stalls for small N)
- Can be responsive to back pressure, which stops FIFO buffers emptying and full feedback to control

The model file is **matmul_CS.mdl**.

Single-Precision Real Floating-Point Matrix Multiply

[Open this model](#)

This design example is a simpler design example of a floating-point matrix multiply implementation than the complex multiply example. The design example uses many more multiply-adds in parallel (128 single precision multiply adds in the default parameterization), to perform each vector multiply simultaneously.

The model file is **matmul_flash_RS.mdl**.

Simple Nonadaptive 2D Beamformer

[Open this model](#)

A beamformer is a key algorithm in radar and wireless and is a signal processing technique that sensor arrays use for directional signal transmission or reception. In transmission, a beamformer controls the phase and amplitude of the individual array elements to create constructive or destructive interference in the wavefront. In reception, information from different elements are combined such that the expected pattern of radiation is preferentially observed. A number of different algorithms exist. An efficient scheme combines multiple paths constructively.

This design example demonstrates a simple nonadaptive 2D beamformer using vectors and single precision arithmetic. The parameters are the number of beams, angle, focus and intensity of each beam.

The simulation calculates the phases in MATLAB code (as a reference), simulates the beamformer 2D design to calculate the phases in DSP Builder Advanced Blockset, compares the reference to the simulation results and plots the beam pattern.

The design example uses vectors of single precision floating-point numbers, with state-machine control from two for loops.

The model file is **beamform_2d.mdl**.

Flow Control

This folder contains the following design examples:

- Avalon-ST Interface (Input and Output FIFO Buffer) with Backpressure (`demo_avalon_st_input_fifo`)
- Avalon-ST Interface (Output FIFO Buffer) with Backpressure (`demo_avalon_st`)
- Kronecker Tensor Product (`demo_kronecker`)
- Parallel Loops (`forloop_parloop`)
- Primitive FIR with Back Pressure (`demo_back_pressure`)
- Primitive FIR with Forward Pressure (`demo_forward_pressure`)
- Primitive Systolic FIR with Forward Flow Control (`demo_forward_pressure_systolic`)
- Rectangular Nested Loop
- Sequential Loops
- Triangular Nested Loop

Avalon-ST Interface (Input and Output FIFO Buffer) with Backpressure

[Open this model](#)

This example demonstrates the Avalon-ST input interface with FIFO buffers and the AvalonST output interface blocks. This example has FIFO buffers in the input and output interfaces. Use the manual switches in the testbench to change when downstream is ready for data or to turn off input. The simulation ends by turning off incoming data and ensures that it writes out as many valid data cycles as it receives.

The model file is `demo_avalon_st_input_fifo.mdl`.

Avalon-ST Interface (Output FIFO Buffer) with Backpressure

[Open this model](#)

This example demonstrates the Avalon-ST input interface and the Avalon-ST output interface blocks. This example has FIFO buffers in the output interface only. Use manual switches in the testbench to change when downstream is ready for data or to turn off input. The simulation ends by turning off incoming data and ensures that it writes out as many valid data cycles as it receives

The model file is `demo_avalon_st.mdl`.

Kronecker Tensor Product

[Open this model](#)

This design example generates a Kronecker tensor product.

The design example shows how to use the Loop block to generate datapaths that operate on regular data.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The **Chip** subsystem includes the Device block and a lower-level **KroneckerSubsystem** subsystem.

The **KroneckerSubsystem** subsystem includes ChannelIn, ChannelOut, Loop, Const, DualMem, Mult, and SynthesisInfo blocks.

In this design example, the top level of the FPGA device (marked by the Device block) and the synthesizable **KroneckerSubsystem** subsystem (marked by the SynthesisInfo block) are at different hierarchy levels.

The model file is **demo_kronecker.mdl**.

Parallel Loops

[Open this model](#)

This design example has two inner loops nested within the outer loop. The inner loops execute in parallel rather than sequentially. The two inner loops are started simultaneously by duplicating the control token but finish at different times. The RendezVous block waits until both of them finish and then passes the control token back to the outer loop.

The model file is **forloop_parloop.mdl**.

Primitive FIR with Back Pressure

[Open this model](#)

This design example uses ModelPrim blocks to implement a FIR design with flow control and back pressure.

The design example shows how you use the ModelPrim FIFO block to implement back pressure and flow control.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The **FirChip** subsystem includes the Device block and a lower-level ModelPrim FIR subsystem.

The ModelPrim FIR subsystem includes ChannelIn, ChannelOut, FIFO, Not, And, Mux, SampleDelay, Const, Mult, Add, and SynthesisInfo blocks.

In this design example, the top level of the FPGA device (marked by the Device block) and the synthesizable ModelPrim FIR subsystem (marked by the SynthesisInfo block) are at different hierarchy levels.

The model file is **demo_back_pressure.mdl**.

This design example shows how back pressure from a downstream block can halt upstream processing. DSP Builder provides three FIR filters. A FIFO buffer follows each FIR filter that can buffer any data that is flowing through the FIFO buffer. If the FIFO buffer becomes half full, the design asserts the ready signal back to the upstream block. This signal prevents any new input (as flagged by valid) entering the FIR block. The FIFO buffers always show the next data if it is available and the valid signal is asserted high. You must AND this FIFO valid signal with the ready signal to consume the data at the head of the FIFO buffer. If the AND result is high, you can consume data because it is available and you are ready for it.

You can chain several blocks together in this way, and no ready signal has to feed back further than one block, which allows you to use modular design techniques with local control.

The delay in the feedback loop represents the lumped delay that spreads throughout the FIR filter block. The delay must be at least as big as the delay through the FIR filter. This delay is not critical. Experiment with some values to find the right one. The FIFO buffer must be able to hold at least this much data after it asserts full. The full threshold must be at least this delay amount below the size of the FIFO buffer (64 – 32 in this design example).

The final block uses an external ready signal that comes from a downstream block in the system.

Primitive FIR with Forward Pressure

[Open this model](#)

This design example uses ModelPrim blocks to implement a FIR design with forward flow control.

The design example shows how you can add a simple forward flow control scheme to a FIR design so that it can handle invalid source data correctly.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The **FirChip** subsystem includes the Device block and a lower-level ModelPrim FIR subsystem.

The ModelPrim FIR subsystem includes ChannelIn, ChannelOut, Mux, SampleDelay, Const, Mult, Add, and SynthesisInfo blocks.

In this design example, the top level of the FPGA device (marked by the Device block) and the synthesizable ModelPrim FIR subsystem (marked by the SynthesisInfo block) are at different hierarchy levels.

The model file is **demo_forward_pressure.mdl**.

The design example has a sequence of three FIR filters that stall when the valid signal is low, preventing invalid data polluting the datapath. The design example has a regular filter structure, but with a delay line implemented in single-cycle latches—effectively an enabled delay line.

You need not enable everything in the filter (multipliers, adders, and so on), just the blocks with state (the registers). Then observe the output valid signal, which DSP Builder pipelines with the logic, and observe the valid output data only.

You can also use vectors to implement the constant multipliers and adder tree, which also speeds up simulation.

You can improve the design example further by using the `TappedDelayLine` block from the Vector Utils library.

Primitive Systolic FIR with Forward Flow Control

[Open this model](#)

This design example uses `ModelPrim` blocks to implement a FIR design with forward flow control. The design example shows how you can add a simple forward flow control scheme to a FIR design so that it can handle invalid source data correctly.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks.

The **FirChip** subsystem includes the `Device` block and a lower-level `ModelPrim FIR` subsystem.

The `ModelPrim FIR` subsystem includes `ChannelIn`, `ChannelOut`, `Mux`, `SampleDelay`, `Const`, `Mult`, `Add`, and `SynthesisInfo` blocks.

In this design example, the top level of the FPGA device (marked by the `Device` block) and the synthesizable `ModelPrim FIR` subsystem (marked by the `SynthesisInfo` block) are at different hierarchy levels.

The design example has a sequence of three FIR filters that stall when the valid signal is low, preventing invalid data polluting the datapath. The design example has a regular filter structure, but with a delay line implemented in single-cycle latches—effectively an enabled delay line.

You need not enable everything in the filter (multipliers, adders, and so on), just the blocks with state (the registers). Then observe the output valid signal, which DSP Builder pipelines with the logic, and observe the valid output data only.

You can also use vectors to implement the constant multipliers and adder tree, which also speeds up simulation. You can improve the design example further with the `TappedDelayLine` block from the Vector Utils library.

The model file is **demo_forward_pressure.mdl**.

Rectangular Nested Loop

[Open this model](#)

In this design example all initialization, step, and limit values are constant. At the corners (at the end of loops) there may be cycles where the count value goes out of range, then the output valid signal from the loop is low.

The token-passing structure is typical for a nested-loop structure. The `bs` port of the innermost loop (`ForLoopB`) connects to the `bd` port of the same loop, so that the next loop iteration of this loop starts immediately after the previous iteration.

The `bs` port of the outer loop (`ForLoopA`) connects to the `ls` port of the inner loop; the `ld` port of the inner loop loops back to the `bd` port of the outer loop. Each iteration of the outer loop runs a full activation of the inner loop before continuing on to the next iteration.

The `ls` port of the outer loop connect to external logic and the `ld` port of the outer loop is unconnected, which is typical of applications where the control token is generated afresh for each activation of the outermost loop.

The model file is `forloop_rectangle.mdl`.

Sequential Loops

[Open this model](#)

In this design example, two inner loops (`InnerLoopA` and `InnerLoopB`) both nest within the outer loop. The design example daisy chains the `ld` port of `InnerLoopA` to the `ls` port of `InnerLoopB` rather than connecting it directly to the `bd` port of `OuterLoop`. Thus each activation of `InnerLoopA` is followed by an activation of `InnerLoopB`.

The model file is `forloop_seqloop.mdl`.

Triangular Nested Loop

[Open this model](#)

The initialization, step, and limit values do not have to be constants. By using the count value from an outer loop as the limit of an inner loop, the counter effectively walks through a triangular set of indices.

The token-passing structure for this loop is identical to that for the rectangular loop, except for the parameterization of the loops.

The model file is `forloop_triangle.mdl`.

Host Interface

This folder contains a design example showing how you can control memory-mapped registers:

- [Memory-Mapped Registers \(demo_regs\)](#)

Memory-Mapped Registers

[Open this model](#)

This design example is an extreme example of using the processor registers to implement a simple calculator. Registers and shared memories write arguments and read results.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks.

This design also includes `BusStimulus` and `BusStimulusFileReader` blocks.

The **RegChip** subsystem includes `RegField`, `RegBit`, `RegOut`, `SharedMem`, `Const`, `Add`, `Sub`, `Mult`, `Convert`, `Select`, `BitExtract`, `Shift`, and `SynthesisInfo` blocks.

The model file is `demo_regs.mdl`.

Platforms

This folder contains the following design examples that illustrate how you can implement a DDC or digital up converter (DUC) for use in a radio basestation. These designs provide a starting point to build your own filter chain that meets your exact needs.

- 16-Channel DDC ([demo_ddc](#))
- 16-Channel DUC ([demo_duc](#))
- 2-Antenna DUC for WiMAX ([demo_wimax_duc](#))
- 2-Channel DUC ([demo_AD9856](#))
- Super-Sample Rate Digital Upconverter ([demo_ssduc](#))

16-Channel DDC

[Open this model](#)

This design example shows how to use using ModelIP and ModelBus blocks to build a 16-channel digital-down converter for modern radio systems.

Decimating CIC and FIR filters down convert eight complex carriers (16 real channels) from 61.44 MHz. The total decimation rate is 64. A real mixer and NCO isolate the eight carriers. The testbench isolates two channels of data from the TDM signals using a channel viewer.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a ChanView block that deserializes the output bus. An Edit Params block allows easy access to the setup variables in the `setup_demo_ddc.m` script.

The **DDCChip** subsystem includes Device, Decimating FIR, DecimatingCIC, Mixer, NCO, Scale, RegBit, and RegField blocks.

The model file is `demo_ddc.mdl`.

The system tutorial ("[DDC Design Example](#)" on page 7-1) describes the 16-channel DDC design example in detail and shows how you can combine blocks to build a system-level design.



This design example uses the Simulink Signal Processing Blockset.

16-Channel DUC

[Open this model](#)

This design example shows how to build a 16-channel DUC as found in modern radio systems using ModelIP, ModelBus, and ModelPrim blocks.

An interpolating filter chain is presented. interpolating CIC and FIR filters up convert eight complex channels (16 real channels). The total interpolation rate is 50. DSP Builder integrates several ModelPrim subsystems into the datapath. This design example shows how you can integrate ModelIP blocks with ModelPrim subsystems:

- A programmable **Gain** subsystem at the start of the datapath exists. This subsystem shows how you can use processor-visible register blocks to control a datapath element.

- The **Sync** subsystem is a ModelPrim subsystem that shows how to manage two data streams coming together and synchronizing. The design writes the data from the NCOs to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly. Alternatively, you can simply delay the NCO value by the correct number of cycles to ensure that the NCO and channel data arrive at the Mixer on the same cycle.

Extensive use is made of Simulink multiplexer and demultiplexer blocks to manage vector signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a ChanView block that deserializes the output bus. An Edit Params block allows easy access to the setup variables in the `setup_demo_duc.m` script.

The **DUCChip** subsystem includes a Device block and a lower level **DUC16** subsystem.

The **DUC16** subsystem includes InterpolatingFIR, InterpolatingCIC, ComplexMixer, NCO, and Scale blocks.

It also includes lower level **Gain**, **Sync**, and **CarrierSum** subsystems which make use of other ModelBus and ModelPrim blocks including AddSLoad, And, BitExtract, ChannelIn, ChannelOut, CompareEquality, Const, SampleDelay, DualMem, Mult, Mux, Not, Or, RegBit, RegField blocks, and SynthesisInfo blocks.

The model file is `demo_duc.mdl`.



This design example uses the Simulink Signal Processing Blockset.

2-Antenna DUC for WiMAX

[Open this model](#)

This design example shows how to build a 2-antenna DUC to meet a WiMAX specification.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a ChanView block that deserializes the output bus.

The **DUCChip** subsystem includes a Device block and a lower level **DUC2Antenna** subsystem.

The **DUC2Antenna** subsystem includes InterpolatingFIR, SingleRateFIR, Const, ComplexMixer, NCO, and Scale blocks.

The model file is `demo_wimax_duc.mdl`.



This design example uses the Simulink Signal Processing Blockset.

2-Channel DUC

[Open this model](#)

This design example shows how to build a 2-channel DUC as found in an ASSP chip.

Interpolating CIC and FIR filters up convert a single complex channel (2 real channels). A NCO and Mixer subsystem combine the complex input channels into a single output channel.

This design example shows how quick and easy it is to emulate the contents of an existing datapath. A ModelPrim block implements the mixer in this design example as the data rate is low enough to save resource using a time-shared hardware technique.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a ChanView block that deserializes the output bus. An Edit Params block allows easy access to the setup variables in the `setup_demo_AD9856.m` script.

The **AD9856** subsystem includes a Device block and a lower level DUCIQ subsystem.

The **DUCIQ** subsystem includes Const, InterpolatingFIR, SingleRateFIR, InterpolatingCIC, NCO, Scale blocks, and a lower level **Mixer** subsystem.

The **Mixer** subsystem includes ChannelIn, ChannelOut, Mult, Const, BitExtract, CompareEquality, And, Delay, Sub, and SynthesisInfo blocks.

The model file is `demo_AD9856.mdl`.



This design example uses the Simulink Signal Processing Blockset.

Super-Sample Rate Digital Upconverter

[Open this model](#)

The model file is `demo_ssduc.mdl`.

Primitive Blocks

This folder contains the following design examples built with ModelPrim blocks:

- [8x8 Inverse Discrete Cosine Transform \(demo_idct8x8\)](#)
- [Automatic Gain Control \(demo_agc\)](#)
- [Bit Combine for Boolean Vectors \(demo_bitcombine\)](#)
- [Bit Extract for Boolean Vectors \(demo_bitextract\)](#)
- [Color Space Converter](#)
- [CORDIC from Primitive Blocks](#)
- [Digital Predistortion Forward Path \(demo_dpd_fwdpath\)](#)
- [Fibonacci Series \(demo_fibonacci\)](#)
- [Fractional Square Root Using CORDIC \(demo_fracsqrt\)](#)
- [Fixed-point Maths Functions](#)
- [Hello World \(helloWorld\)](#)
- [Hybrid Direct Form and Transpose Form FIR Filter \(demo_hybrid_fir_mc\)](#)
- [Loadable Counter \(demo_ld_counter\)](#)
- [Matrix Initialization of LUT \(demo_lut_matrix_init.mdl\)](#)

- Matrix Initialization of Vector Memories (**demo_dualmem_matrix_init**)
- Multichannel IIR Filter (**demo_iir**)
- Quadrature Amplitude Modulation (**demo_QAM256**)
- Reinterpret Cast for Bit Packing and Unpacking (**demo_reinterpret_cast**)
- Run-time Configurable Decimating and Interpolating Half-Rate FIR Filter (**demo_fir_tdd**)
- Square Root Using CORDIC (**demo_sqrt**)
- Test CORDIC Functions with the CORDIC Block (**demo_cordic_lib_block**)
- Vector Initialization of Sample Delay (**demo_sample_delay_vector**)
- Vector Sort—Sequential
- Folded Vector Sort
- Wide Single-Channel Accumulators (**demo_wide_accumulators**)

8×8 Inverse Discrete Cosine Transform

[Open this model](#)

This design example uses the Chen-Wang algorithm to implement a fully pipelined 8×8 inverse discrete cosine transform (IDCT).

Separate subsystems perform the row transformation (**Row**), corner turner (**CornerTurn**), and column transformation (**Col**) functions. The design example synthesizes each separate subsystem separately. In the **Row** and **Col** subsystems, additional levels of hierarchy for the different stages exist, but because the **SynthesisInfo** block is at the row or column level, the design example flattens these subsystems before synthesis.



The **CornerTurn** turn block makes extensive use of Simulink **Goto/From** blocks to reduce the wiring complexity.

The top-level testbench includes **Control**, **Signals**, **Run ModelSim**, and **Run Quartus II** blocks.

The **IDCTChip** subsystem includes the **Device** block and a lower level IDCT subsystem.

The **IDCT** subsystem includes lower level subsystems that it describes with the **ChannelIn**, **ChannelOut**, **Const**, **BitCombine**, **Shift**, **Mult**, **Add**, **Sub**, **BitExtract**, **SampleDelay**, **OR Gate**, **Not**, **Sequence**, and **SynthesisInfo ModelPrim** blocks.

The model file is **demo_idct8x8.mdl**.

Automatic Gain Control

[Open this model](#)

This design example implements an automatic gain control.

This design example shows a complex loop with several subloops that it schedules and pipelines without inserting registers. The design example spreads a lumped delay around the circuit to satisfy timing while maintaining correctness. Processor visible registers control the thresholds and gains.

In complex algorithmic circuits, the zero-latency blocks make it easy to follow a data value through the circuit and investigate the algorithm without offsetting all the results by the pipelining delays.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The **AGC_Chip** subsystem includes the Device block, a RegField block and a lower level **AGC** subsystem.

The **AGC** subsystem includes RegField, ChannelIn, ChannelOut, Mult, SampleDelay, Add, Sub, Convert, Abs, CmpGE, Lut, Const, SharedMem, Shift, BitExtract, Select, and SynthesisInfo blocks.

The model file is **demo_agc.mdl**.

Bit Combine for Boolean Vectors

[Open this model](#)

This design example demonstrates different ways to use the BitCombine primitive block to create signals of different widths from a vector of Boolean signals. The one input BitCombine block is a special case that concatenates all the components of the input vector and output one wide scalar signal. You can apply 1-bit reducing operators to vectors of Boolean signals. The BitCombine block supports multiple input concatenation. When vectors of Boolean signals are input on multiple ports, corresponding components from each vector are combined so that the output is a vector of signals.

The model file is **demo_bitcombine.mdl**.

Bit Extract for Boolean Vectors

[Open this model](#)

This design example demonstrates different ways to use the BitExtract block to split a wide signal into a vector of narrow signal components. This block converts a scalar signal into a vector of Boolean signals. You use the initialization parameter to arbitrarily order the components of the vector output by the BitExtract block. If the input to a BitExtract block is a vector, different bits can be extracted from each of the components. The output does not always have to be a vector of Boolean signals. You may split a 16-bit wide signal into four components each 4-bits wide.

The model file is **demo_bitextract.mdl**.

Color Space Converter

[Open this model](#)

This design example demonstrates ModelPrim subsystems with the following simple RGB to Y'CbCr color space conversion:

- $Y = 0.257R + 0.504G + 0.098B + 16$
- $Cb = -0.148R - 0.291G + 0.439B + 128$
- $Cr = 0.439R - 0.368G - 0.071B + 128$

The RGB data arrives as three parallel signals each clock cycle. The model file is **demo_csc.mdl**.

CORDIC from Primitive Blocks

[Open this model](#)

This design example demonstrates building a CORDIC out of basic operators. This design has the same functionality as the CORDIC library block in the **demo_cordic_lib_block** example.

The model file is **demo_cordic_primitives.mdl**.

Digital Predistortion Forward Path

[Open this model](#)

This design example represents forward paths that digital predistortion (DPD) uses. Forward paths compensate for nonlinear power amplifiers by applying the inverse of the distortion that the power amplifier generates, such that the pre-distortion and the distortion of the power amplifier cancel each other out. The power amplifier's non-linearity may change over time, therefore such systems are typically adaptive.

This design example is based on "A robust digital baseband pre-distorter constructed using memory polynomials," L. Ding, G. T. Zhou, D. R. Morgan, et al., *IEEE Transactions on Communications*, vol. 52, no. 1, pp. 159-165, 2004.

This design example only implements the forward path, which is representative of many systems where you implement the forward path in FPGAs, and the feedback path on external processors. The design example sets the predistortion memory, Q , to 8; the highest nonlinearity order K is 5 in this design example. The file **setup_demo_dpd_fwdpath** initializes the complex valued coefficients, which are stored in registers. During operation, the external processor continuously improves and adapts these coefficients with a microcontroller interface.

The model file is **demo_dpd_fwdpath.mdl**.

Fibonacci Series

[Open this model](#)

This design example generates a Fibonacci sequence.

This design example shows that even for circuitry with tight feedback loops and 120-bit adders, designs can achieve high data rates by the pipelining algorithms. The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. The **Chip** subsystem includes the Device block and a lower level **FibSystem** subsystem. The **FibSystem** subsystem includes ChannelIn, ChannelOut, SampleDelay, Add, Mux, and SynthesisInfo blocks.



In this design example, the top-level of the FPGA device (marked by the `Device` block) and the synthesizable `ModelPrim` subsystem (marked by the `SynthesisInfo` block) are at different hierarchy levels.

The model file is `demo_fibonacci.mdl`.

Folded Vector Sort

[Open this model](#)

This design sorts the values on the input vector from largest to smallest. The design is a masked subsystem that allows for sorting with either a comparator and mux block, or a minimum and a maximum block. The first implementation is more efficient. Both use the reconfigurable subsystem to choose between implementations using the **BlockChoice** parameter.

Folded designs repeatedly use a single dual sort stage. The throughput of the design is limited in the number of channels, vector width, and data rate. The data passes through the dual sort stage (vector width)/2 times. The vector sort design example uses full throughput with (vector width)/2 dual sort stages in sequence.

Look under the mask to view the implementation; of reconfigurable subsystem templates and the blocks that reorder and interleave vectors.

The model file is `demo_foldedsort.mdl`.

Fractional Square Root Using CORDIC

[Open this model](#)

This design example demonstrates the `CORDIC` block. It configures the `CORDIC` block for `fixdt(0,32,31)` input and output. This design example is fully iterative.

The design example allows you to generate a valid signal. The design example only generates output and can only accept input every N cycles, where N depends on the number of stages, the data output format, and the target f_{MAX} . The valid signal goes high when the output is ready. You can use this output signal to trigger the next input. For example, a FIFO buffer read for bursty data.

The model file is `demo_cordic_fracsqrt.mdl`.

Fixed-point Maths Functions

[Open this model](#)

This design example demonstrates how the `Math`, `Trig` and `Sqrt` functions support fixed-point types and the fixed-point `Divide` function. You can use fixed-point types of width up to and including 32 bits. DSP Builder generates results using the same techniques as in the floating point functions but at generally reduced resource usage, depending on data bit width. Outputs are faithfully rounded. If the exact result is between two representable numbers within the data format, DSP Builder uses either of them. In some instances you see a difference in output result between simulation and hardware by one LSB. To get bit-accurate results at the subsystem level, this example uses the **Bit Exact** option on the `SynthesisInfo` block.

The model file is `demo_fixed_math.mdl`.

Hello World

[Open this model](#)

This design example is a simple design example that outputs a simple text message that it stores in a look-up table.

An external input enables a counter that addresses a lookup-table (LUT) that contains some text. The design example writes the result to a MATLAB array. You can examine the contents with a `char(message)` command in the MATLAB command window.

This design example does not use any `ChannelIn`, `ChannelOut`, `GPIn`, or `GPOut` blocks. The design example uses Simulink ports for simplicity although they prevent the automatic testbench flow from working.

The top-level testbench includes `Control`, `Signals`, `Run ModelSim`, and `Run Quartus II` blocks.

The **Chip** subsystem includes `Device`, `Counter`, `Lut`, and `SynthesisInfo` blocks.



In this design example, the top-level of the FPGA device (marked by the `Device` block) and the synthesizable `ModelPrim` subsystem (marked by the `SynthesisInfo` block) are at the same level.

The model file is **helloWorld.mdl**.

Hybrid Direct Form and Transpose Form FIR Filter

[Open this model](#)

This FIR filter shows a FIR architecture that is a hybrid between the direct form and transpose form FIR filter. It combines the advantages of both. The design example uses small, four-tap direct form filters to use the structure inside the DSP block efficiently. The design example combines these direct form minifilters into a transpose structure, which minimizes the logic and memory that the sample pipe uses.

The model file is **demo_hybrid_fir_mc.mdl**.

Loadable Counter

[Open this model](#)

This design example demonstrates the `LoadableCounter` block.

The testbench reloads the counter with new parameters every 64 cycles. A manual switch allows you to control whether the counter is permanently enabled, or only enabled on alternate cycles. You can view the signals input and output from the counter with the provided scope.

The model file is **demo_ld_counter.mdl**.

Matrix Initialization of LUT

[Open this model](#)

You can initialize Lut blocks in exactly the same way. The design example feeds a vector of addresses to the ModelPrim block such that DSP Builder gives each vector component a different address. This design example also shows Lut blocks working with complex data types.

Using this design example avoids demultiplexing, connecting, and multiplexing, so that you can build parameterizable systems.

You can use one of the following ways to specify the contents of the Lut block:

- Specify table contents as single row or column vector. The length of the 1D row or column vector determines the number of addressable entries in the table. If DSP Builder reads vector data from the table, all components of a given vector share the same value.
- When a look-up table contains vector data, you can provide a matrix to specify the table contents. The number of rows in the matrix determines the number of addressable entries in the table. Each row specifies the vector contents of the corresponding table entry. The number of columns must match the vector length, otherwise DSP Builder issues an error.



The default initialization of the LUT is a row vector `round([0:255]/17)`. This vector is inconsistent with the default for the DualMem block, which is a column vector `[zeros(16, 1)]`. The latter form is consistent with the new matrix initialization form in which the number of rows determines the addressable size.

The model file is **demo_lut_matrix_init.mdl**.

Matrix Initialization of Vector Memories

[Open this model](#)

The design example file is **demo_dualmem_matrix_init.mdl**.

You can initialize both the dual memory and LUT ModelPrim blocks with matrix data.

Use this feature in designs that handle vector data and require individual components of each vector in the dual memory to be initialized uniquely.

The number of rows in the 2D matrix that you provide for initialization determines the addressable size of the dual memory. The number of columns must match the width of the vector data. So the n^{th} column specifies the contents of the n^{th} dual memory. Within each of these columns the i^{th} row specifies the contents at the $(i-1)^{\text{th}}$ address (the first row is address zero, second row address 1, and so on).

The exception for this row and column interpretation of the initialization matrix is for 1D data, where the initialization matrix consists of either a single column or single row. In this case, the interpretation is flexible and maps the vector (row or column) into the contents of each dual memory defaults (the previous behavior, in which all dual memories have identical initial contents).

The **demo_dualmem_matrix_init** design example uses complex values in both the initialization and the data that it later writes to the dual memory. You set up the contents matrix in the model's set-up script, which runs on model initialization.

Multichannel IIR Filter

[Open this model](#)

This design example implements a masked multi-channel infinite impulse response (IIR) filter with a masked subsystem that it builds from ModelPrim blocks.

This design example has many feedback loops. The design example implements all the pipelined delays in the circuit automatically. The multiple channels provide more latency around the circuit to ensure a high clock frequency result. Lumped delays allow you to easily parameterize the design example when changing the channel counts. For example, masking the subsystem provides the benefits of a black-box IP block but with visibility.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses.

The **IIRChip** subsystem includes the Device block and a masked **IIRSubsystem** subsystem. The coefficients for the filter are set from `[b, a] = ellip(2, 1, 10, 0.3);` in the callbacks for the masked subsystem. You can look under the mask to see the implementation details of the **IIRSubsystem** subsystem which includes ChannelIn, ChannelOut, SampleDelay, Const, Mult, Add, Sub, Convert, and SynthesisInfo blocks.

The model file is **demo_iir.mdl**.

Quadrature Amplitude Modulation

[Open this model](#)

This design example implements a simple quadrature amplitude modulation (QAM256) design example with noise addition. The testbench uses various Simulink blocks.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The **QAM256Chip** subsystem includes Add, GPIn, GPOut, BitExtract, Lut, BitCombine, and SynthesisInfo blocks.

The model file is **demo_QAM256.mdl**.



This design example uses the Simulink Communications Blockset.

Reinterpret Cast for Bit Packing and Unpacking

[Open this model](#)

This design example demonstrates the ReinterpretCast block, which packs signals into a long word and extracts multiple signals from a long word. The first datapath reinterprets a single precision complex signal into raw 32-bit components that separate into real and imaginary parts. A BitCombine block then merges it into a 64-bit signal. The second datapath uses the BitExtract block to split a 64-bit wide signal into a two component vectors of 32-bit signals. The ReinterpretCast block then converts the raw bit pattern into single-precision IEEE format. The HDL that the design synthesizes is simple wire connections, which performs no computation.

The model file is `demo_reinterpret_cast.mdl`.

Run-time Configurable Decimating and Interpolating Half-Rate FIR Filter

[Open this model](#)

This design example contains a half-rate FIR filter, which can perform either decimation or interpolation by a factor of two during run time. In decimation mode, the design example accepts a new sample every clock cycle, and produces a new result every two clock cycles. When interpolating, the design example accepts a new input every other clock cycle, and produces a new result every clock cycle. In both cases, the design example fully uses multipliers, making this structure very efficient compared to parallel instantiations of interpolate and decimate filters, or compared to a single rate filter with external interpolate and decimate stages.

The coefficients are set to [1 0 3 0 5 6 5 0 3 0 1] to illustrate the operation of the filter in `setup_demo_fir_tdd.m`.

The model file is `demo_fir_tdd.mdl`.

Square Root Using CORDIC

[Open this model](#)

This design example demonstrates the CORDIC block. It configures the CORDIC block for uint (32) input and uint (16) output. The example is partially parallelized (four stages).

The design example allows you to generate a valid signal. The design example only generates output and can only accept input every N cycles, where N depends on the number of stages, the data output format, and the target f_{MAX} . The valid signal goes high when the output is ready. You can use this output signal to trigger the next input. For example, a FIFO buffer read for bursty data.

The model file is `demo_cordic_sqrt.mdl`.

Test CORDIC Functions with the CORDIC Block

[Open this model](#)

This design example demonstrates how to use the ModelPrim CORDIC block to implement the coordinate rotation digital algorithm (CORDIC) algorithm. A Mode input can either rotate the input vector by a specified angle, or rotate the input vector to the x-axis while recording the angle required to make that rotation. You can experiment with different size of inputs to control the precision of the CORDIC output.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The SinCos and AGC subsystem includes ChannelIn, ChannelOut, CORDIC, and SynthesisInfo blocks.

The model file is `demo_cordic_lib_block.mdl`.



For more information, refer to the Help page for the **CORDIC** block.

Vector Sort—Sequential

[Open this model](#)

This design example sorts the values on the input vector from largest to smallest. The sorting is a configurable masked subsystem: sortstages.

For sorting, the sortstages subsystem allows either a comparator and mux based block, or one based on a minimum and a maximum block. The first is more efficient. Both use the reconfigurable subsystem to choose between implementations using the **BlockChoice** parameter.

The design repeatedly uses a dual sort stage in series. The data passes through the dual sort stage (vector width)/2 times.

Look under the mask to view the implementation; of reconfigurable subsystem templates and the blocks that reorder and interleave vectors.

The model file is **demo_vectorsort.mdl**.

Vector Sort—Iterative

[Open this model](#)

This design sorts the values on the input vector from largest to smallest. The design is a masked subsystem that allows for sorting with either a comparator and mux block, or a minimum and a maximum block. The first implementation is more efficient. Both use the reconfigurable subsystem to choose between implementations using the **BlockChoice** parameter.

Folded designs repeatedly use a single dual sort stage. The throughput of the design is limited in the number of channels, vector width, and data rate. The data passes through the dual sort stage (vector width)/2 times. The vector sort design example uses full throughput with (vector width)/2 dual sort stages in sequence.

Look under the mask to view the implementation; of reconfigurable subsystem templates and the blocks that reorder and interleave vectors.

The model file is **demo_foldedsort.mdl**.

Vector Initialization of Sample Delay

[Open this model](#)

When the SampleDelay ModelPrim block receives vector input, you can independently specify a different delay for each of the components of the vector.

This design example shows that one sample delay can replace what usually requires a Demultiplex, SampleDelay, and Multiplex combination.

You may give individual components zero delay resulting in a direct feed through of only that component. Avoid algebraic loops if you select some components to be zero delays.

This rule only applies when DSP Builder is reading and outputting vector data. A scalar specification of delay length still sets all the delays on each vector component to the same value. You must not specify a vector that is not the same length as the vector on the input port. A negative delay on any one component is also an error. However, as in the scalar case, you can specify a zero length delay for one or more of the components.

The model file is **demo_sample_delay_vector.mdl**.

Wide Single-Channel Accumulators

[Open this model](#)

This example design shows various ways to connect up an adder, sample delay (depth=1), and optional multiplexer to implement reset or load. The output type of the adder is propagated from one of the inputs. You must select the correct input, otherwise the accumulator fails to schedule. You may add a Convert block to ensure the accumulator also maintains sufficient precision.

The wide single-channel accumulator consists of a two-input adder and sample-delay feedback with one cycle of latency. If you use a fixed-point input to this accumulator, you can make it arbitrarily wide provided the types of the inputs match with a data type prop duplicate block. The output type of the Add block can be with or without word growth. Alternatively, you can propagate the input type to the output of the adder.

The optional use of a two-to-one multiplexer allows the accumulator to load values according to a Boolean control signal. The inputs differ in precision, so the type with wider fractional part must be propagated to the output type of the adder, otherwise the accumulator fails to schedule. Converting both inputs to the same precision ensures that the single-channel accumulator can always be scheduled even at high f_{MAX} targets.

If neither input has a fixed-point type that is suitable for the adder to output, use a Convert block to ensure that the precision of both inputs to the Add block are the same. Scheduling of this accumulator at high f_{MAX} fails.

The model file is **demo_wide_accumulators.mdl**.

Reference Designs

The installation also includes reference designs that demonstrate the design of DDC and DUC systems for digital intermediate frequency (IF) processing.



For more information about these designs, refer to *AN 544: Digital IF Modem Design with the DSP Builder Advanced Blockset*.

This folder accesses groups of reference designs that illustrate the design of DDC and DUC systems for digital intermediate frequency (IF) processing.

The first group implements IF modem designs compatible with the Worldwide Interoperability for Microwave Access (WiMAX) standard. Altera provide separate models for one and two antenna receivers and transmitters:

- **1-Antenna WiMAX DDC (wimax_ddc_1rx)**

- 2-Antenna WiMAX DDC (`wimax_ddc_2rx_iiqq`)
- 1-Antenna WiMAX DUC (`wimax_duc_1tx`)
- 2-Antenna WiMAX DUC (`wimax_ddc_2tx_iiqq`)

The second group implement IF modem designs compatible with the wideband Code Division Multiple Access (W-CDMA) standard:

- 4-Carrier, 2-Antenna W-CDMA DDC (`wcdma_multichannel_ddc_mixer`)
- 1-Carrier, 2-Antenna W-CDMA DDC (`wcdma_picocell_ddc_mixer`)
- 4-Carrier, 2-Antenna W-CDMA DUC (`wcdma_multichannel_duc_mixer`)
- 1-Carrier, 2-Antenna W-CDMA DUC (`wcdma_picocell_duc_mixer`)
- 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32 (`wcdma_multichannel_duc_mixer_96x_32R`)
- 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48 (`wcdma_multichannel_duc_mixer_96x_48R`)
- 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40 (`wcdma_multichannel_duc_mixer_80x_40R`)

This folder also accesses the following reference designs:

- Cholesky Solver Single Channel
- Cholesky Solver Multiple Channels
- Crest Factor Reduction
- Direct RF with Synthesizable Testbench
- Dynamic Decimating FIR Filter
- QR Decomposition
- Single-Channel 10-MHz LTE Transmitter
- Time Delay Beamformer
- Variable Integer Rate Decimation Filter
- Space-time adaptive processing (STAP) radar:
 - STAP Radar Forward and Backward Substitution
 - STAP Radar QR Decomposition 192x204
 - STAP Radar Steering Generation

STAP for radar systems applies temporal and spatial filtering to separate slow moving targets from clutter and null jammers. Applications demand high- processing requirements and low latency for rapid adaptation. High-dynamic ranges demand floating-point datapaths.

1-Antenna WiMAX DDC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 2-channel, 1-antenna, single-frequency modulation DDC for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. The design includes an Edit Params block to allow easy access to the setup variables in the `setup_wimax_ddc_1rx.m` script.

The **DDCChip** subsystem includes Device, Decimating FIR, Mixer, NCO, SingleRateFIR, and Scale blocks. Also, an **Interleaver** subsystem extracts the correct I and Q channel data from the demodulated data stream.

The FIR filters implement a decimating filter chain that down convert the two channels from a frequency of 89.6 MSPS to a frequency of 11.2 MSPS (a total decimation rate of eight). The real mixer, NCO, and **Interleaver** subsystem isolate the two channels. The design configures the NCO with a single-channel to provide one sine and one cosine wave at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA that the Device block defines inside the **DDCChip** subsystem.

The model file is `wimax_ddc_1rx.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

2-Antenna WiMAX DDC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, 2-frequency modulation DDC for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. The design includes an Edit Params block to allow easy access to the setup variables in the `setup_wimax_ddc_2rx_iiqq.m` script.

The **DDCChip** subsystem includes Device, Decimating FIR, Mixer, NCO, SingleRateFIR, and Scale blocks.

The FIR filters implement a decimating filter chain that down convert the two channels from a frequency of 89.6 MSPS to a frequency of 11.2 MSPS (a total decimation rate of 8). The real mixer and NCO isolate the two channels. The design configures the NCO with two channels to provide two sets of sine and cosine waves at the same frequency of 22.4 MHz. The NCO has the same sample rate of (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA, which the Device block defines inside the **DDCChip** subsystem.

The model file is `wimax_ddc_2rx_iiqq.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

1-Antenna WiMAX DUC

[Open this model](#)

This reference design uses ModelIP, ModelBus, and ModelPrim blocks to build a 2-channel, 1-antenna, single-frequency modulation DUC for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. The design includes an Edit Params block to allow easy access to the setup variables in the `setup_wimax_duc_1tx.m` script.

The **DUCChip** subsystem includes a Device block to specify the target FPGA device, and a **DUC2Channel** subsystem which contains SingleRateFIR, Scale, InterpolatingFIR, NCO, and ComplexMixer blocks. Also an **deinterleaver** subsystem exists that contains a series of ModelPrim blocks including delays and multiplexers that de-interleave the two I and Q channels.

The FIR filters implement an interpolating filter chain that up converts the two channels from a frequency of 11.2 MSPS to a frequency of 89.6 MSPS (a total interpolating rate of 8). The complex mixer and NCO modulate the two input channel baseband signals to the IF domain. The design configures the NCO with a single channel to provide one sine and one cosine wave at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA, which the Device block defines inside the **DUCChip** subsystem.

The model file is `wimax_duc_1tx.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

2-Antenna WiMAX DUC

[Open this model](#)

This reference design uses ModelIP, ModelBus, and ModelPrim blocks to build a 4-channel, 2-antenna, single-frequency modulation DUC for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. The design includes an Edit Params block to allow easy access to the setup variables in the `setup_wimax_duc_2tx_iqq.m` script.

The **DUCChip** subsystem includes a Device block to specify the target FPGA device, and a **DUC2Channel** subsystem which contains SingleRateFIR, Scale, InterpolatingFIR, NCO, ComplexMixer, and Const blocks. It also contains a **Sync** subsystem which shows how to manage two data streams coming together and synchronizing. The design writes the data from the NCOs to a memory with the channel index as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly. (Alternatively, you can simply delay the NCO value by the correct number of cycles to ensure that the NCO and channel data arrive at the Mixer on the same cycle). Also, a **deinterleaver** subsystem exists that contains a series of ModelPrim blocks including delays and multiplexers that de-interleave the four I and Q channels.

The FIR filters implement an interpolating filter chain that up converts the two channels from a frequency of 11.2 MSPS to a frequency of 89.6 MSPS (a total interpolating rate of 8).

A complex mixer and NCO modulate the two input channel baseband signals to the IF domain. The design configures the NCO to provide two sets of sine and cosine waves at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

The **Sync** subsystem shows how to manage two data streams coming together and synchronizing. The design writes the data from the NCOs to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

A system clock rate of 179.2 MHz drives the design on the FPGA, which the Device block defines inside the **DUCChip** subsystem.

The model file is **wimax_duc_2tx_iiqq.mdl**.



This reference design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna W-CDMA DDC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 16-channel, 2-antenna, multiple-frequency modulation DDC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks, plus a ChanView block that isolates two channels of data from the TDM signals.

The **DDCChip** subsystem includes Device, DecimatingCIC, Decimating FIR, Mixer, NCO, and Scale blocks. It also contains a **Sync** subsystem which provides the synchronization of the channel data to the NCO carrier waves.

The CIC and FIR filters implement a decimating filter chain that down converts the eight complex carriers (16 real channels from two antennas with four pairs of I and Q inputs from each antenna) from a frequency of 122.88 MSPS to a frequency of 7.68 MSPS (a total decimation rate of 16). The real mixer and NCO isolate the four channels. The design configures the NCO with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (122.88 MSPS) as the input data sample rate.

The **Sync** subsystem shows how to manage two data streams that come together and synchronize. The data from the NCOs writes to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

A system clock rate of 245.76 MHz drives the design on the FPGA, which the Device block defines inside the **DDCChip** subsystem.

The model file is **wcdma_multichannel_ddc_mixer.mdl**.



This reference design uses the Simulink Signal Processing Blockset.

1-Carrier, 2-Antenna W-CDMA DDC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, single-frequency modulation DDC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks, plus a ChanView block that isolates two channels of data from the TDM signals.

The **DDCChip** subsystem includes Device, DecimatingCIC, Decimating FIR, Mixer, NCO, and Scale blocks.

The CIC and FIR filters implement a decimating filter chain that down converts the two complex carriers (4 real channels from two antennas with one pair of I and Q inputs from each antenna) from a frequency of 122.88 MSPS to a frequency of 7.68 MSPS (a total decimation rate of 16). The real mixer and NCO isolate the four channels. The design configures the NCO with a single channel to provide one sine and one cosine wave at a frequency of 17.5 MHz. The NCO has the same sample rate (122.88 MSPS) as the input data sample rate.

A system clock rate of 122.88 MHz drives the design on the FPGA, which the Device block defines inside the **DDCChip** subsystem.

The model file is `wcdma_picocell_ddc_mixer.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna W-CDMA DUC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 16-channel, 2-antenna, multiple-frequency modulation DUC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The **DUCChip** subsystem includes a Device block to specify the target FPGA device, and a **DUC** subsystem that contains InterpolatingFIR, InterpolatingCIC, NCO, ComplexMixer, and Scale blocks.

The FIR and CIC filters implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). The complex mixer and NCO modulate the four channel baseband input signal onto the IF region. The design configures the NCO with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The subsystem **SyncMixSumSel** uses ModelPrim blocks to implement the synchronization, mixing, summation, scaling, and signal selection. This subsystem separates each operation into further subsystems. The **Sync** subsystem shows how to manage two data streams that come together and synchronize. The data from the NCOs writes to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

The **Sum** and **SampSelectr** subsystems sum up the right modulated signals to the designated antenna.

A system clock rate of 245.76 MHz drives the design on the FPGA, which the Device block defines inside the **DUC** subsystem.

The model file is `wcdma_multichannel_duc_mixer.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

1-Carrier, 2-Antenna W-CDMA DDC

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, single-frequency modulation DUC for an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The **DUCChip** subsystem includes a Device block to specify the target FPGA device, and a **DUC** subsystem that contains InterpolatingFIR, InterpolatingCIC, NCO, ComplexMixer, and Scale blocks.

The FIR and CIC filters implement an interpolating filter chain that up convert the four channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). The complex mixer and NCO modulate the four channel baseband input signal onto the IF region.

The design example configures the NCO with a single channel to provide one sine and one cosine wave at a frequency of 17.5 MHz. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

A system clock rate of 122.88 MHz drives the design on the FPGA, which the Device block defines inside the **DDC** subsystem.

The model file is `wcdma_picocell_duc_mixer.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation DUC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The **DUCChip** subsystem includes a Device block to specify the target FPGA device, and a **DUC** subsystem that contains InterpolatingFIR, InterpolatingCIC, NCO, ComplexMixer, and Scale blocks.

The FIR and CIC filters implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). This design example uses dummy signals and carriers to achieve the desired rate up conversion, because of the unusual FPGA clock frequency and total rate change combination. The complex mixer and NCO modulate the four channel baseband input signal onto the IF region. The design example configures the NCO with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz and 27.5 MHz, respectively. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The **Sync** subsystem shows how to manage two data streams that come together and synchronize. The data from the NCOs writes to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

The **GenCarrier** subsystem manipulates the NCO outputs to generate carrier signals that can align with the datapath signals.

The **CarrierSum** and **SignalSelector** subsystems sum up the right modulated signals to the designated antenna.

A system clock rate of 368.64 MHz, which is 96 times the input sample rate, drives the design on the FPGA, which the Device block defines inside the **DUC** subsystem. The higher clock rate can potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The model file is **mcducmix96x32R.mdl**.



This reference design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48

[Open this model](#)

This reference design uses ModelIP and ModelBus blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation DUC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The **DUCChip** subsystem includes a **Device** block to specify the target FPGA device, and a **DUC** subsystem that contains **InterpolatingFIR**, **InterpolatingCIC**, **NCO**, **ComplexMixer**, and **Scale** blocks.

The FIR and CIC filters implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 184.32 MSPS (a total interpolation factor of 48).

The complex mixer and NCO modulate the four channel baseband input signal onto the IF region. The design configures the NCO with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (184.32 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The **Sync** subsystem shows how to manage two data streams that come together and synchronize. The data from the NCOs writes to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

The **CarrierSum** and **SignalSelector** subsystems sum up the right modulated signals to the designated antenna.

A system clock rate of 368.64 MHz, which is 96 times the input sample rate, drives the design on the FPGA, which the **Device** block defines inside the **DUC** subsystem. The higher clock rate can potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The model file is **mcducmix96x48R.mdl**.



This reference design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40

[Open this model](#)

This reference design uses **ModelIP** and **ModelBus** blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation DUC for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes **Control**, **Signals**, and **Run Quartus II** blocks. A **Spectrum Scope** block computes and displays the periodogram of the outputs from the two antennas.

The **DUCChip** subsystem includes a **Device** block to specify the target FPGA device, and a **DUC** subsystem that contains **InterpolatingFIR**, **InterpolatingCIC**, **NCO**, **ComplexMixer**, and **Scale** blocks.

The FIR and CIC filters implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 153.6 MSPS (a total interpolation factor of 40).

The complex mixer and NCO modulate the four channel baseband input signal onto the IF region. The design configures the NCO with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (153.6 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The **Sync** subsystem shows how to manage two data streams that come together and Synchronize. The design writes data from the NCOs to a memory with the channel as an address. The data stream uses its channel signals to read out the NCO signals, which resynchronizes the data correctly.

The **CarrierSum** and **SignalSelector** subsystems sum up the right modulated signals to the designated antenna.

A system clock rate of 307.2 MHz, which is 80 times the input sample rate, drives the design on the FPGA, which the `Device` block defines inside the `DUC` subsystem. The higher clock rate can potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The model file is `mcducmix80x40R.mdl`.



This reference design uses the Simulink Signal Processing Blockset.

Cholesky Solver Single Channel

[Open this model](#)

The Cholesky Solver Single Channel reference design performs Cholesky decomposition to solve column vector x in $Ax = b$, where:

A is a Hermitian, positive definite matrix (for example covariance matrix) and b is a column vector.

Forward and backward substitution solve x .

The design decomposes A into L^*L' , therefore $L^*L'^*x = b$, or $L^*y = b$, where $y = L'^*x$. The design solves y with forward substitution and x with backward substitution.

The design calculates the diagonal element of the matrix first before proceeding to subsequent rows to hide the processing latency of the inverse square root function as much as possible. Performance of multiple banks operations with vector size of about 30 may increase up to 6%. Expect no performance gain when vector size is the same as matrix size.

To input the lower triangular elements of matrix A and b with the input bus, specify the column, row, and channel index of each element. The design transposes and appends the column vector b to the bottom of A and treats it as an extension of A in terms of column and row addressing.

The output is column vector x with the bottom element output first.

A multiple channel design uses different techniques to enhance performance.

The single-channel model file is `cholseky_solver_sc.mdl`; the multiple-channel model file is `cholseky_solver_mc.mdl`.

Cholesky Solver Multiple Channels

[Open this model](#)

The Cholesky Solver Multiple Channels reference design performs Cholesky decomposition to solve column vector x in $Ax = b$, where:

A is a Hermitian, positive definite matrix (for example covariance matrix) and b is a column vector.

Forward and backward substitution solve x .

The design decomposes A into L^*L' , therefore $L^*L'x = b$, or $L^*y = b$, where $y = L'x$. The design solves y with forward substitution and x with backward substitution.

This design uses cycle stealing and command FIFO techniques to enhance performance. Although it targets multiple channels, it also works well with single channels.

To input the lower triangular elements of matrix A and b with the input bus, specify the column, row, and channel index of each element. The design transposes and appends the column vector b to the bottom of A and treats it as an extension of A in terms of column and row addressing.

The output is column vector x with the bottom element output first.

A multiple channel design optimizes performance by prioritizing diagonal element calculation over non-diagonal ones.

The single-channel model file is **cholseky_solver_sc.mdl**; the multiple-channel model file is **cholseky_solver_mc.mdl**.

Crest Factor Reduction

[Open this model](#)

This reference design implements crest factor reduction, based on the peak cancelling algorithm.

For further information refer to the [Altera crest factor reduction for wireless systems](#) web page.

You can change the simulation length by clicking on the Simulink Length block.

The model file is **demo_cfr.mdl**.

Direct RF with Synthesizable Testbench

[Open this model](#)

This very large reference design implements a digital upconversion to RF and digital predistortion, with a testbench that you can synthesize to hardware for easier on-chip testing. The model file is **DirectRFTest_and_DPD_SV.mdl**.

Dynamic Decimating FIR Filter

[Open this model](#)

The dynamic decimating FIR reference design offers multichannel run-time decimation ratios in integer power of 2 and run-time control of channel count (in trading with bandwidth). The design supports dynamic channel count to signal bandwidth trade off: e.g., if you halve the channel count, the input sample rate doubles

The FIR filter length is $2 \times (D_{\max} / D_{\min}) \times N + 1$ where D_{\max} and D_{\min} are the maximum and minimum decimation ratios and N is the number of (1 sided) symmetric coefficients at D_{\min} .

All channels must have the same decimation ratio. The product of the number of Channels and the minimum decimation ratio must be 4 or more. The design limits the wire count to 1 and:

number of channels \times sample rate = clock rate.

The model file is **demo_dyndeci.mdl**

Multichannel QR Decomposition

[Open this model](#)

This reference design is a complete linear equations system solution that uses QR decomposition. To optimize the overall throughput the solver can interleave multiple data instances at the same time. The inputs of the design are system matrices $A [n \times m]$ and input vectors.

The reference design uses the Gram-Schmidt method to decompose system matrix A to Q and R matrices. It calculates the solution of the system by completing backward substitution.

The reference design is fully parametrizable: system dimensions n and m , the processing vector size, which defines the parallelization ratio of the dot product engine, and the number of channels that the design processes in parallel. This design uses single-precision Multiply and Add blocks that perform most of the floating-point calculations to implement a parallel dot product engine. The design uses a processor, which executes a fixed set of micro-instructions and generates operation indexes, to route different phases of the calculation through these blocks. The design uses for-loop macro blocks, which allow very efficient, flexible, and high-level implementation of iterative operations, to implement the processor.

The model file is **demo_mcqrd.mdl**.

QR Decomposition

[Open this model](#)

This reference design is a complete linear equations system solution that uses QR decomposition. The input of the design is a system matrix $A [n \times m]$ and input vector.

The reference design uses the Gram-Schmidt method to decompose system matrix A to Q and R matrices, and calculates the solution of the system by completing backward substitution.

The reference design is fully parametrizable—system dimensions n and m , and the processing vector size, which defines the parallelization ratio of the dot product engine. This design uses single-precision `Multiply` and `Add` blocks that perform most of the floating-point calculations to implement a parallel dot product engine. The design uses a processor, which executes a fixed set of microinstructions and generates operation indexes, to route different phases of the calculation through these blocks. The design uses for-loop macro blocks, which allow very efficient, flexible and high-level implementation of iterative operations, to implement the processor.

This design uses the `Run All Testbenches` block to access enhanced features of the automatically-generated testbench. An application-specific m-function verifies the simulation output, to correctly handle the complex results and the numerical approximation because of the floating-point format.

The model file is `demo_qrd.mdl`.

Reconfigurable Decimation Filter

[Open this model](#)

The reconfigurable decimation filter reference design uses primitive blocks to build a variable integer rate decimation FIR filter. The reference design has the following features:

- Supports arbitrary integer decimation rate (including the cases without rate change), arbitrary number of channels and arbitrary clock rate and input sample rate, if the clock rate is high enough to process all channels in a single data path (i.e. no hardware duplication).
- Supports run-time reconfiguration of decimation rate.
- Uses two memory banks for filter coefficients storage instead of prestoring coefficients for all rates in memory. Updates one memory bank while the design is reading coefficients from the other bank.
- Implements real time control of scaling in the FIR datapath.

You can modify the parameters in the `setup_vardownsampler.m` file, which you access from the Edit Params icon.

The model file is `vardownsampler.mdl`.

Single-Channel 10-MHz LTE Transmitter

[Open this model](#)

This reference design uses `ModelIP`, `ModelPrim`, and blocks from the FFT Blockset library to build a single-channel 10-MHz LTE transmitter.

The top-level testbench includes blocks to access control and signals, and to run the Quartus II software. It also includes an `Edit Params` block to allow easy access to the configuration variables in the `setup_sc_LTEtxr.m` script. A discrete-time scatter plot scope displays the constellation of the modulated signal in inphase versus quadrature components.

The `LTE_txr` subsystem includes a `Device` block to specify the target FPGA device, and `64QAM`, `1K_IFFT`, `ScaleRnd`, `CP_bReverse`, `Chg_Data_Format`, and `DUC` blocks.

The **64QAM** subsystem uses a lookup table to convert the source input data into 64 QAM symbol mapped data. The **1K_IFFT** subsystem converts the frequency domain quadrature amplitude modulation (QAM) modulated symbols to time domain. The **ScaleRnd** subsystem follows the conversion, which scales down the output signals and converts them to the specified fixed-point type.

The bit **CP_bReverse** subsystem adds extended cycle prefix (CP) or guard interval for each orthogonal frequency-domain multiplexing (OFDM) symbol to avoid intersymbol interference (ISI) that causes multipaths. The **CP_bReverse** block reorders the output bits of IFFT subsystems, which are in bit-reversed order, so that they are in the correct order in time domain. The design adds the cyclic prefix bit by copying the last 25% of the data frame, then appends to the beginning of it.

The **Chg_Data_Format** subsystem changes the output data format of **CP_bReverse** subsystem to match the working protocol format of **DUC** subsystem.

The **DUC** subsystem uses an interpolating filter chain to achieve an interpolation factor of 16, such that the design interpolates the 15.36 Msps input channel to 245.76 Msps. In this design, an interpolating finite impulse response (FIR) filter interpolates by 2, followed by a cascaded integrator-comb (CIC) filter with an interpolation rate of 8. A NCO generates orthogonal sinusoids at specified carrier frequency. The design mixes the signals with complex input data with a **ComplexMixer** block. The final **SINC** compensation filter compensates for the digital analog converter (DAC) frequency response roll-off.

A system clock rate of 245.76 MHz drives the design on the FPGA. The **Signals** block of the design defines this clock. The input random data for the 64QAM symbol mapping subsystem has a data rate of 15.36 Msps.

The model file is **sc_LTEtxr.mdl**.

STAP Radar Forward and Backward Substitution

[Open this model](#)

The QR decomposition reference design produces an upper triangular matrix and a lower triangular matrix. The design applies this linear system of equations to the steering vector in the following two steps:

- Forward substitution with the lower triangular matrix
- Backward substitution with the lower triangular matrix

A command pipeline controls the routing of floating-point vectors. Nested **ForLoop** blocks generate these commands. Another **FIFO** unit queues the commands. This decoupled system of **FIFO** buffers maximizes the usage of the shared vector floating-point block while automatically throttling the rate of the **ForLoop** system.

This design uses advanced settings from the **DSP Builder > Verify Design** menu to access enhanced features of the automatically generated testbench. An application specific m-function verifies the simulation output, to correctly compare complex results and properly handle floating-point errors that arise from the ill-conditioning of the **QRD** output.

The model file is **STAP_ForwardAndBackwardSubstitution.mdl**.

STAP Radar Steering Generation

[Open this model](#)

The STAP radar steering generation reference design uses ForLoop blocks and floating-point primitives to generate the steering vector. You input the angle of arrival and Doppler frequency.

The model file is **STAP_steeringGen.mdl**.

STAP Radar QR Decomposition 192x204

[Open this model](#)

The QR decomposition reference design is as a sequence of floating-point vector operations. Single-precision Multiply and Add blocks perform most of the floating-point calculations. The design routes different phases of the calculation through these blocks with a controlling processor that executes a fixed set of microinstructions. FIFO units ensure this architecture maximizes the usage of the Multiply and Add blocks.

This design uses the Run All Testbenches block to access enhanced features of the automatically generated testbench. An application specific m-function verifies the simulation output, to correctly handle the complex results and the numerical approximation due to the floating-point format.

The model file is **STAP_qrd192x204.mdl**. The parallel version model file is **STAP_qrd192x204_p.mdl**.

Time Delay Beamformer

[Open this model](#)

The time delay beamformer reference design implements a time-delay beamformer that has many advantages over traditional phase-shifted beamformer. It uses a (full-band) Nyquist filter and farrow-like structure for optimal performance and resource usages. The design also includes the following features so you can simulate and verify the transmit and receive beamforming operations:

- Waveform (chirp) generation
- Target emulation
- Receiver noise emulation
- Aperture tapering
- Pulse compression

Variable Integer Rate Decimation Filter

[Open this model](#)

The variable integer rate decimation filter reference design implements a 16-channel interpolate-by-2 symmetrical 49-tap FIR filter. The target system clock frequency is 320 MHz.

You can modify the parameters in the **setup_vardecimator_rt.m** file, which you access from the Edit Params icon.

The model file is **vardecimator_rt.mdl**.

Waveform Synthesis

This folder contains the following design examples that synthesize waveforms with a NCO or direct digital synthesis (DDS):

- **Complex Mixer** (**demo_complex_mixer**)
- **Four Channel, Two Banks NCO** (**demo_mc_nco_2banks_mem_interface**)
- **Four Channel, Four Banks NCO** (**demo_mc_nco_4banks_mem_interface**)
- **Four Channel, Eight Banks, Two Wires NCO** (**demo_mc_nco_8banks_2wires**)
- **Four Channel, 16 Banks NCO** (**demo_mc_nco_16banks**)
- **ModelIP**
- **NCO** (**demo_nco**)
- **NCO with Exposed Bus** (**demo_nco_exposed_bus**)
- **Real Mixer** (**demo_mix**)

Complex Mixer

[Open this model](#)

This design example shows how to mix complex signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_complex_mixer.m** script.

The **FilterSystem** subsystem includes the Device and ComplexMixer blocks.

The model file is **demo_complex_mixer.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Four Channel, Two Banks NCO

[Open this model](#)

This design example implements a NCO with four channels and two banks. This design example demonstrates frequency-hopping with the NCO block to generate four channels of sinusoidal waves that you can switch from one set (bank) of frequencies to another.

The phase increment values are set directly into the **NCO Parameter** dialog box as a 2 (rows) × 4 (columns) matrix. The input for the bank index is set up so that it alternates between the two predefined banks with each one lasting 2000 steps.

A `BusStimulus` block sets up an Avalon-MM interface that writes into the phase increment memory registers. It shows how you can use the Avalon-MM interface to dynamically change the frequencies of the NCO-generated sinusoidal signals at run time. This design example uses a 16-bit memory interface (as the `Control` block specifies) and a 24-bit the accumulator in the `NCO` block. The design example requires two registers for each phase increment value. With the base address of the phase increment memory map set to 1000 in this design example, the addresses [1000 1001 1002 1003 1012 1013 1014 1015] write to the phase increment memory registers of channels 1 and 2 in bank 1, and to the registers of channels 3 and 4 in bank 2. The write data is also made up of two parts with each part writing to one of the registers feeding the selected phase increment accumulators.

Two banks of frequencies exist in this design example with each bank processed for 2,000 steps before switching to the other. You should write a new value into the phase increment memory register for each bank to change the NCO output frequencies after 8,000 steps during simulation. To avoid writing new values to the active bank, the design example configures the write enable signals in the following way:

```
[zeros(1,7000) 1 1 1 1 zeros(1,2000) 1 1 1 1 zeros(1,8000)]
```

This configuration ensures that a new phase increment value for bank 0 is written at 7000 steps when the NCO is processing bank 1; and a new phase increment value for bank 1 is written at 9000 steps when the NCO is processing bank 0.

Four writes for each bank exist to write new values for channel 1 and 2 into bank 0, and new values for channel 3 and 4 into bank 1. Each new phase value needs two registers due to the size of the memory interface.

The `Spectrum Scope` block shows three peaks for a selected channel with the first two peaks representing the two banks and the third peak showing the frequency that you specify through the memory interface. The scope of the select channel shows the sinusoidal waves of the channel you select. You can zoom in to see that smooth and continuous sinusoidal signals at the switching point. You can also see the frequency changes after 8000 steps where the phase increment value alters through the memory interface.

The top-level testbench includes `Control`, `Signals`, `BusStimulus`, `Run ModelSim`, and `Run Quartus II` blocks, plus `ChanView` blocks that deserialize the output buses. An `Edit Params` block allows easy access to the setup variables in the `setup_demo_mc_nco_2banks_mem_interface.m` script.

The `NCOSubSystem` subsystem includes the `Device` and `NCO` blocks.

The model file is `demo_mc_nco_2banks_mem_interface.mdl`.



This design example uses the Simulink Signal Processing Blockset.

Four Channel, Four Banks NCO

[Open this model](#)

This design example implements a NCO with four channels and four banks. This design example is similar to the [Four Channel, Two Banks NCO](#) design, but it has four banks of frequencies defined for the phase increment values. Five peaks on each spectrum plot exist: the fifth peak shows the changes the design example writes through the memory interface.

The design example uses a 32-bit memory interface with a 24-bit accumulator. Hence, the design example requires only one phase increment memory register for each phase increment value—refer to the address and data setup on the `BusStimulus` block inside this design example.

This design example has four banks of frequencies with each bank processed for 2,000 steps before switching to the other. You should write a new value into the phase increment memory register for each bank to change the NCO output frequencies after 16,000 steps during simulation. To avoid writing new values to the active bank, the design example configures the write enable signals in the following way:

```
[zeros(1,15000) 1 zeros(1,2000) 1 zeros(1,2000) 1 zeros(1,2000) 1 zeros(1,8000)].'
```

This configuration ensures that a new phase increment value for bank 0 is written at 15000 steps when the NCO is processing bank 3; a new phase increment value for bank 1 is written at 17000 steps when the NCO is processing bank 0; a new phase increment value for bank 2 is written at 19000 steps when the NCO is processing bank 1; and a new phase increment value for bank 3 is written at 21000 steps when the NCO is processing bank 2.

There is one write for each bank to write a new value for channel 1 into bank 0; a new value for channel 2 into bank 1; a new value for channel 3 into bank 2; and a new value for channel 4 into bank 3. Each new phase value needs only one register due to the size of the memory interface.

The top-level testbench includes `Control`, `Signals`, `BusStimulus`, `Run ModelSim`, and `Run Quartus II` blocks, plus `ChanView` blocks that deserialize the output buses. An `Edit Params` block allows easy access to the setup variables in the `setup_demo_mc_nco_4banks_mem_interface.m` script.

The `NCOSubSystem` subsystem includes the `Device` and `NCO` blocks.

The model file is `demo_mc_nco_4banks_mem_interface.mdl`.



This design example uses the Simulink Signal Processing Blockset.

Four Channel, Eight Banks, Two Wires NCO

[Open this model](#)

This design example implements a NCO with four channels and eight banks. This design example is similar to the [Four Channel, 16 Banks NCO](#) design, but has only eight banks of phase increment values (specified in the setup script for the workspace variable) feeding into the NCO. Furthermore, the sample time for the NCO requires two wires to output the four channels of the sinusoidal signals. Two wires exist for the NCO output, each wire only contains two channels. Hence, the channel indicator is from 0 .. 3 to 0 .. 1.

You can inspect the eight peaks on the spectrum graph for each channel and see the smooth continuous sinusoidal waves on the scope display.

This design example uses an additional subsystem (`Select_bank_out`) to extract the NCO-generated sinusoidal signal of a selected bank on a channel.

The design example outputs the data to the workspace and plots through with the separate **demo_mc_nco_extracted_waves.mdl**, which demonstrates that the output of the bank you select does represent a genuine sinusoidal wave. However, from the scope display, you can see that the sinusoidal wave is no longer smooth at the switching point, because the design example uses the different values of phase increment values between the selected banks. You can only run the **demo_mc_nco_extracted_waves.mdl** model after you run **demo_mc_nco_8banks_2wires.mdl**.

The top-level testbench includes Control, Signals, BusStimulus, Run ModelSim, and Run Quartus II blocks, plus ChanView blocks that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_mc_nco_8banks_2wires.m** script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The **Select_bank_out** subsystem contains Const, CompareEquality, and AND Gate blocks.

The model file is **demo_mc_nco_8banks_2wires.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Four Channel, 16 Banks NCO

[Open this model](#)

This design example implements a NCO with four channels and 16 banks. This design example demonstrates frequency-hopping with the NCO block to generate 4 channels of sinusoidal waves, which you can switch from one set (bank) of frequencies to another in the 16 predefined frequency sets.

A workspace variable *phaseIncr* defines the 16 (rows) \times 4 (columns) matrix for the phase increment input with the phase increment values that the setup script calculates.

The input for the bank index is set up so that it cycles from 0 to 15 with each bank lasting 1200 steps.

The spectrum display shows clearly 16 peaks for the selected channel indicating that the design example generates 16 different frequencies for that channel. The scope of the selected channel shows the sinusoidal waves of the selected channel. You can zoom in to see that the design example generates smooth and continuous sinusoidal signals at the switching point.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView blocks that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_mc_nco_16banks.m** script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The model file is **demo_mc_nco_16banks.mdl**.



This design example uses the Simulink Signal Processing Blockset.

ModelIP

The ModelIP design example describes how you can build a NCO design with the NCO block from the Waveform Synthesis library.

“ModelIP Tutorial” on page 4-1 describes this design example.



This design example uses the Simulink Signal Processing Blockset.

NCO

[Open this model](#)

This design example uses the NCO block from the Waveform Synthesis library to implement an NCO. A Simulink double precision sine or cosine wave compares the results.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView blocks that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_nco.m** script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The model file is **demo_nco.mdl**.



This design example uses the Simulink Signal Processing Blockset.

NCO with Exposed Bus

[Open this model](#)

This design example is a multichannel NCO that outputs four waveforms with slightly different frequencies. Halfway through the simulation, DSP Builder reconfigures the NCO for smaller increments, which gives a waveform with a longer period.

The model file is **demo_nco_exposed_bus.mdl**.

Real Mixer

[Open this model](#)

This design example shows how to mix non-complex signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus ChanView block that deserialize the output buses. An Edit Params block allows easy access to the setup variables in the **setup_demo_mix.m** script.

The MixerSystem subsystem includes the Device and Mixer blocks.

The model file is **demo_mix.mdl**.



This design example uses the Simulink Signal Processing Blockset.

Intermediate frequency (IF) modem designs have multichannel, multirate filter lineups. The filter lineup is often programmable from a host processor, and has stringent demands on DSP performance and accuracy.

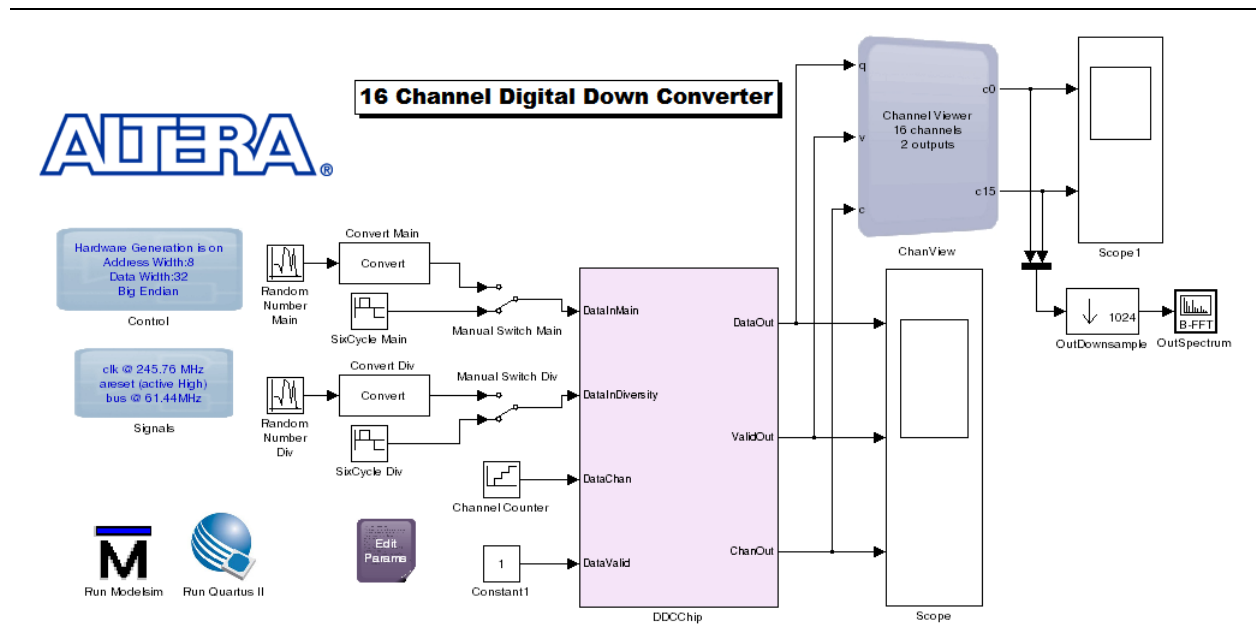
The DDC design example is a high-performance design running at over 300 MHz in a Stratix IV device. The design example is very efficient and at under 4,000 logic registers gives a low cost per channel.

DDC Design Example Description

This design example uses NCO/DDS, mixer, CIC, and FIR filter ModelIP components to build a 16-channel programmable DDC for use in a wide range of radio applications.

Figure 7-1 shows the entire design for the DDC (the testbench). View the **DDCChip** subsystem to see the components you require to build a complex, production ready system.

Figure 7-1. Testbench for the DDC Design



The top-level testbench includes Run ModelSim and Run Quartus II blocks, Control and Signals blocks, and some Simulink blocks to generate source signals and visualize the output. The full power of the Simulink blocksets is available for your design.



For full details of each block parameter, refer to their block descriptions in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

The **DDCChip** subsystem block contains the following blocks that form the lowest level of the design hierarchy:

- The NCO and mixer
- Decimate by 16 CIC filter
- Two decimate by 4 FIR odd-symmetric filters: one with length 21, the other length with 63.

The other blocks in this subsystem perform a range of rounding and saturation functions. They also allow dynamic scaling. The **Device** block specifies the target FPGA.

Signals Block

The **Signals** block allows you to define the relationship between the sample rates and the system clock, to tell the synthesis engines how much folding or time sharing to perform. Increasing the system clock permits more folding, and therefore typically results in more resource sharing, and a smaller design.

You also need a system clock rate so that the synthesis engines know how much to pipeline the logic. For example, by considering the device and speed grade, the synthesis tool can calculate the maximum length that an adder can have. If the design exceeds this length, it pipelines the adder and adjusts the whole pipeline to compensate. This adjustment typically results in a small increase in logic size, which is usually more than compensated for by the decrease in logic size through increased folding.

The **Signals** block specifies the clock and reset names, with the system clock frequency. The bus clock or FPGA internal clock for the memory-mapped interfaces can be run at a lower clock frequency. This lets the design move the low-speed operations such as coefficient update completely off the critical path.



To specify the clock frequency, clock margin, and bus clock frequency values in this design, use the MATLAB workspace variables *ClockRate* and *ClockMargin*, which you can edit by double-clicking on the **Edit Params** block.

Control Block

The **Control** block controls the whole DSP Builder advanced blockset environment. It examines every block in the system, controls the synthesis flow, and writes out all RTL and scripts. A single control block must be present in every top-level model.

In this design, hardware generation creates RTL. DSP Builder places the RTL and associated scripts in the directory *./rtl*, which is a relative path based on the current MATLAB directory. DSP Builder creates automatic self-checking testbenches, which saves the data that a Simulink simulation captures to build testbench stimulus for each block in your design. DSP Builder generates scripts to run these simulations.

The threshold values control the hardware generation. They control the trade-offs between hardware resources, such as hard DSP blocks or soft LE implementations of multipliers. You can perform resource balancing for your particular design needs with a few top-level controls.

Memory Maps

Many memory-mapped registers in the design exist such as filter coefficients and control registers for gains. You can access these registers through a memory port that DSP Builder automatically creates at the top-level of your design. DSP Builder creates all address decode and data multiplexing logic automatically. DSP Builder generates a memory map in XML and HTML that you can use to understand the design. To access this memory map, after simulation, on the DSP Builder menu, point to **Resource Usage** and click **Design, Current Subsystem**, or **Selected block**. The address and data widths are set to 8 and 32 in the design.

EditParams Blocks

The EditParams block allows you to edit the script `setup_demo_ddc.m`, which sets up the MATLAB variables that configure your model. Use the MATLAB design example properties callback mechanism to call this script.



The *PreloadFcn* callback uses this script to setup the parameters when your design example opens and the *InitFcn* callback re-initializes your design example to take account of any changes when the simulation starts.

You can edit the parameters in the `setup_demo_ddc.m` script by double-clicking on the Edit Params block to open the script in the MATLAB text editor.

The script sets up MATLAB workspace variables. The *SampleRate* variable is set to 61.44 MHz, which is typical of a CDMA system, and represents a quarter of the system clock rate that the FPGA runs at. You can use the feature to TDM four signals onto any given wire.

Source Blocks

The Simulink environment enables you to create any required input data for your design. In the DDC design, use manual switches to select sine wave or random noise generators. DSP Builder encodes a simple six-cycle sine wave as a table in a Repeating Sequence Stair block from the Simulink Sources library. This sine wave is set to a frequency that is close to the carrier frequencies that you specify in the NCOs, allowing you to see the filter lineup decoding some signals. DSP Builder creates VHDL for each block as part of the testbench RTL.

Sink Blocks

Simulink Sink library blocks display the results of the DDC simulation. The Scope block displays the raw output from the DDC design. The design has TDM outputs and all the data shows as data, valid and channel signals.

At each clock cycle, the value on the data wire either carries a genuine data output, or data that you can safely discard. The valid signal differentiates between these two cases. If the data is valid, the channel wire identifies the channel where the data belongs. Thus, you can use the valid and channel wires to filter the data. The ChanView block automates this task and decodes 16 channels of data to output channels 0 and 15. The block decimates these channels by the same rate as the whole filter line up and passes to a spectrum scope block (OutSpectrum) that examines the behavior in the frequency domain.

DDCChip Subsystem

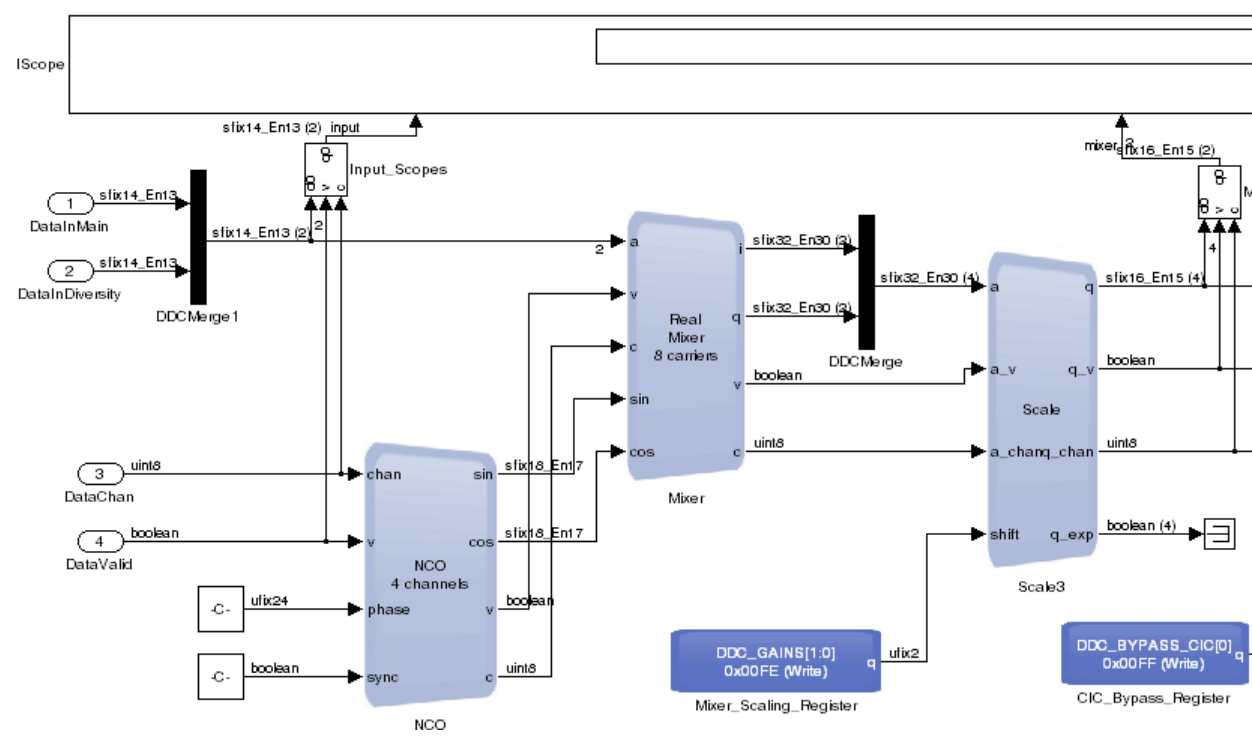
The **DDCChip** subsystem contains a Device block. This block labels this level of design hierarchy that compiles onto the FPGA. DSP Builder generates VHDL for all levels of the hierarchy, but this level has additional script files that build a project in the Quartus II software. The Device block sets the FPGA family, device and speed grade. The family and speed grade optimize the hardware. In combination with the target clock frequency, the device determines the degree of pipelining.

The following sections describe the blocks in the **DDCChip** subsystem. The following three types of block exist:

- The grey blocks are ModelIP blocks. These represent functional IP such as black box filters, NCOs, and mixers.
- The blue blocks are processor visible registers.
- The black and white blocks are Simulink blocks.

Figure 7-2 shows the first part of the datapath.

Figure 7-2. DDCChip datapath (NCO and Mixer)



The inputs, NCO, and mixer stages show with Simulink signal formats turned on.

Primary Inputs

The primary inputs to the hardware are two parallel data signals (`DataInMain` and `DataInDiversity`), a channel signal (`DataChan`), and a valid signal (`DataValid`). The parallel data signals represent inputs from two antennas. They are of type `sfixed14_13` which is a Simulink fixed-point type of total width 14 bits. The type is signed with 13 bits of fraction, which is a typical number format that an analog-to-digital converter generates.

The data channel `DataChan` is always an 8-bit unsigned integer (`uint8`) and DSP Builder synthesizes away the top bits if not used. The valid signal `DataValid` indicates when real data transmits. The first rising edge of the valid signal starts operation of the first blocks in the chain. As the first blocks start producing outputs, their valid outputs start the next blocks in the chain. This mechanism ensures that filter chain start up is coordinated without having a global controller for the latencies of each block. The actual latencies of the blocks may change based on the clock frequency and FPGA selection.

Merge Multiplexer

The ModelIP blockset supports vectors on its input and output data wires, which ensures that a block diagram is scalable when, for example, changing channel counts and operating frequencies. The merge multiplexer (`DDCMerge1`) takes two individual wires and combines them into a vector wire of width 2. Although this is a Simulink Mux block, it does not perform any multiplexing in hardware—it is just as a vectorizing block. If you examine the RTL, it contains just wires.

NCO

The NCO block generates sine and cosine waveforms to a given precision. These waveforms represent a point in the complex plane rotating around the origin at a given frequency. DSP Builder multiplies this waveform by the incoming data stream to obtain the data from the transmitted signal.



Four frequencies exist, because the vector in the **Phase Increment and Inversion** field is of length 4.

DSP Builder configures the NCO block to produce a signed 18-bit value with 17 bits of fraction. The internal accumulator width is set to 24 bits. This internal precision affects the spurious free dynamic noise (SFDN). DSP Builder specifies the initial frequencies for the simulation as phase increments. The phase accumulator width in bits is 2^{24} , thus one complete revolution of the unit circle corresponds to a value of 2^{24} . Dividing this number by 5.95, means that the design requires 5.95 cycles to perform one complete rotation. That is, the wavelength of the sine and cosine that the design produces are 5.95 cycles. The sample rate is 61.44 MHz, therefore the frequency is $61.44/5.95$, which is 10.32 MHz.

The input frequency in the testbench rotates every 6 cycles for a frequency of $61.44/6=10.24$ MHz. Therefore, you can expect to recover the difference of these frequencies (0.08 MHz or 80 kHz), which fall in the low-pass filters pass bands, because DSP Builder mixes these signals.



For more information about how to configure the NCO block refer to the “[ModelIP Tutorial](#)” on page 4-1.

The design exposes phase values through a memory-mapped interface at the address specified by the variable `DDC_NCO_PHASE_INCR`, which is set to address `0x0000` in the setup script. After simulation, to view resource usage for the design example, the subsystem, or a selected block, on the DSP Builder menu, point to **Resource Usage** and click **Design**, **Current Subsystem**, or **Selected block**.

DSP Builder reports for each register, the name, width, reset value, and address. This report collates all the registers from your design into a single location.

You can view the estimated results for this NCO configuration in the **Results** tab of the dialog box).

Based on the selected accumulator width and output width, DSP Builder calculates an estimated SFDR and accumulator precision. To verify this precision in a separate testbench, use `demo_nco.mdl` as a start.

Mixer

The Mixer block performs the superheterodyne operation by multiplying each of the two received signals (`DataInMain` and `DataInDiversity`) by each of the four frequencies. This action produces eight complex signals or 16 scalar signals (the 16 channels in the DDC design).

The mixer requires sufficient multipliers to perform this calculation. The total number of real \times complex multipliers required for each sample is $2 \text{ signals} \times 4 \text{ frequencies} = 8$.

Thus, 8 real \times complex multiplies require $8 \times 2 = 16$ scalar multipliers. This processing is spread over four cycles (the folding factor given by the ratio of clock rate to sample rate), therefore DSP Builder requires four physical multipliers.

After simulation, to view resource usage for the design example, the subsystem, or a selected block, on the DSP Builder menu, point to **Resource Usage** and click **Design**, **Current Subsystem**, or **Selected block**.

You can list the input and output ports that DSP Builder creates for this block, with the data width and brief description, by right-clicking on the block and clicking **Help**. DSP Builder suffixes the vector inputs with 0 and 1 to implement the vector. This list of signals corresponds to the signals in the VHDL entity.

DSP Builder provides the results for the mixer as separate in phase and quadrature outputs—each is a vector of width 2. It performs the remaining operations on both the I and Q signals, so that DSP Builder can combine them with another Simulink multiplexer to provide a vector of width 4. This operation carries the 16 signals, with a folding factor of 4. At this point the channel counts count 0, 1, 2, 3, 0, 1,

Mixer Scale Block

At this point in the datapath, the data width is 32 bits representing the full precision output of multiplying a 14-bit data signal with an 18-bit sine or cosine signal.

DSP Builder needs to reduce the data width to a lower precision to pass on to the remaining filters, which reduces the resource count considerably, and does not cause significant information loss. The `Scale3` block performs a shift-round-saturate operation to achieve this reduction. The shift is usually a 1 or 2 bit shift that you can set to adjust the gain in your design at run time.

To determine the setup, DSP Builder usually uses a microprocessor, which writes to a register to set the shift amount. This design uses a `RegField` block (`Mixer_Scaling_Register`). This block behaves like a constant in the Simulink simulation, but in hardware the block performs as a processor-writable register that initializes to the value in your design example.

This parameterization results in a register mapped to address `DDC_GAINS`, which is a MATLAB variable that you specify in the `setup_demo_ddc.m` script.

The register is writable from the processor, but not readable.

The register produces a 2-bit output of type `ufix(2)`—an unsigned fixed-point number. The scaling is 2^{-0} so is, in effect, a 2-bit unsigned integer. These 2 bits are mapped into bits 0 and 1 of the word (another register may use other bits of this same address). The initial value for the register is set to 0. DSP Builder provides a description of the memory map in the resource usage. Sometimes, Simulink needs an explicit sample time, but you can use the default value of `-1` for this tutorial.

The 2-bit unsigned integer is fed to the `Scale3` block. This block has a vector of width 4 as its data input. The `Scale3` block builds a vector of 4 internal scale units. These parameters are not visible through the user interface, but you can see them in the resource usage.

The block produces four outputs, which DSP Builder presents at the output as a vector of width 4. DSP Builder preserves the order in the vector. You can create quite a large block of hardware by passing many channels through a `ModelIP` block. The exception output of the scale block provides signals to say when saturation occurs, which this design does not require, so this design terminates them.

The design sets the output format to 16-bit signed with 15 bits of fraction and uses the `Unbiased` rounding method. This method (convergent rounding or round-to-even) typically avoids introducing a DC bias.

The saturation method uses `Symmetric` rounding which clips values to within `+0.9999` and `-0.9999` (for example) rather than clipping to `-1`. Again this avoids introducing a DC bias.

The number of bits to shift is a vector of values that the scaling register block (`Mixer_Scaling_Register`) indexes. The vector has 4 values, therefore DSP Builder requires a 2-bit input.

An input of 0 uses the 0th value in the vector (address 1 in Simulink), and so on. Therefore, in this example `inout0` shifts by 0 and the result at the input has the same numerical range as the input. An input of 1 shifts left by 1, and so multiplies the input value by 2, thus increasing the gain.

Opening the Design Example

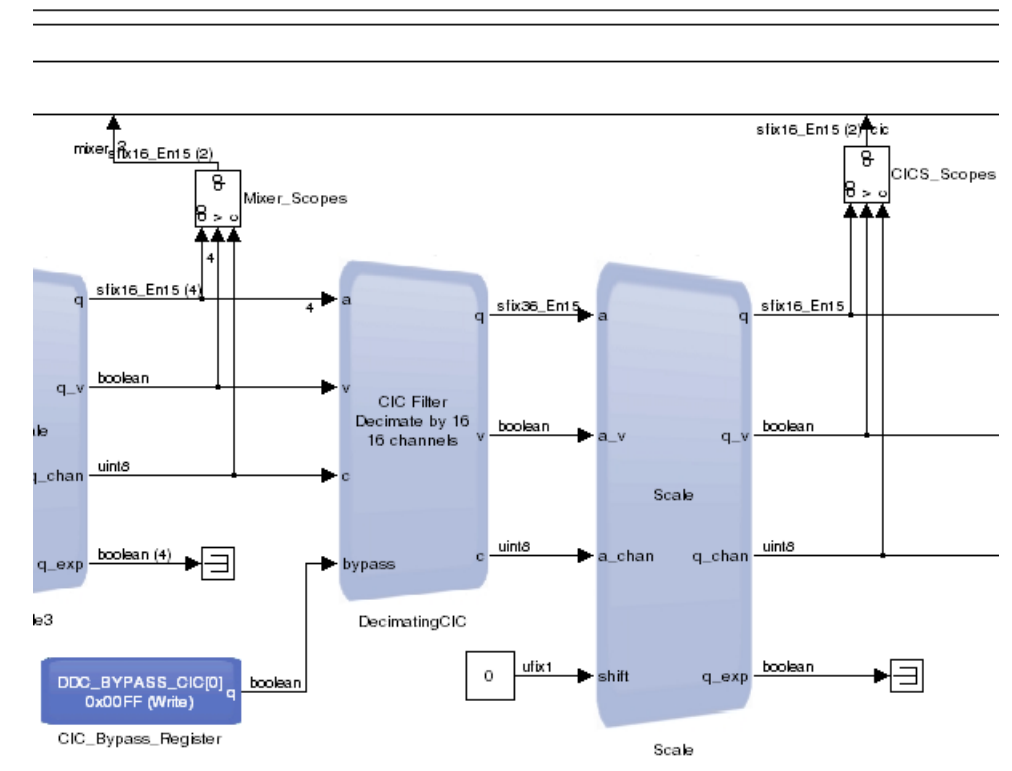
To open the model, type the following command in the MATLAB window:

```
demo_ddc ↵
```


DecimatingCIC and Scale Blocks

This part of the datapath comprises a decimating cascaded integrator comb (CIC) filter and scale block (Figure 7-3).

Figure 7-3. DDCChip Datapath (DecimatingCIC and Scale Blocks)



Two main blocks exist—the DecimatingCIC and the Scale block. To configure the CIC Filter, double click on the DecimatingCIC block.

The input sample rate is still the same as the data from the antenna. The *dspbd_dc.SampleRate* variable specifies the input sample rate. The number of channels, *dspbd_dc.ChanCount*, is a variable set to 16. The CIC filter has 5 stages, and performs decimation by a factor of 16. 1/16 in the dialog box indicates that the output rate is 1/16th of the input sample rate. The CIC parameter differential delay controls how many delays each CIC section uses—nearly always set to 1.

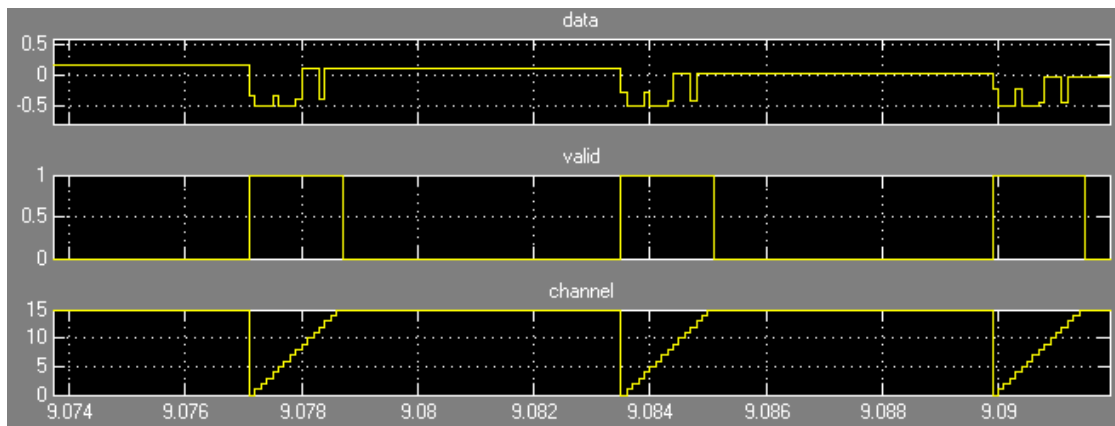
The CIC has no registers to configure, therefore no memory map elements exist.

The input data is a vector of four elements, so DSP Builder builds the decimating CIC from four separate CICs, each operating on four channels. The decimation behavior reduces the data rate at the output, therefore all 16 data samples (now at 61.44/16 MSPS each channel) can fit onto 1 wire.

The DecimatingCIC block multiplexes the results from each of the internal CIC filters onto a single wire. That is, four channels from vector element 1, followed by the four channels from vector element 2. DSP Builder packs the data onto a single TDM wire. Data is active for 25% of the cycles because the aggregate sample rate is now 61.44 MSPS × 16 channels/16 decimation = 61.44 MSPS and the clock rate for the system is 245.76 MHz.

Figure 7-4 shows this behavior, which is the `CIC_All_Scope` in the `CIC_Scopes` subsystem. Bursts of data occur, with 16 contiguous samples followed by a gap. Each burst is tagged with the valid signal. Also the channel indicator shows that the channel order is 0..15.

Figure 7-4. CIC_All_Scope Showing Output From the DecimatingCIC Block



The number of input integrator sections is 4, and the number of output comb sections is 1. The lower data rate reduces the size of the overall group of 4 CICs. The Help page also reports the gain for the DCIC to be 1,048,576 or approximately 2^{20} . The Help page also shows how DSP Builder combines the four channels of input data on a single output data channel. The comb section utilization (from the DSP Builder menu) confirms the 25% calculation for the folding factor.

The Scale block reduces the output width in bits of the CIC results.

In this case, the design requires no variable shifting operation, so it uses a Simulink constant to tie the shift input to 0. However, because the gain through the DecimatingCIC block is approximately 2^{20} division of the output, enter a scalar value -20 for the **Number of bits to shift left** in the dialog box to perform data.



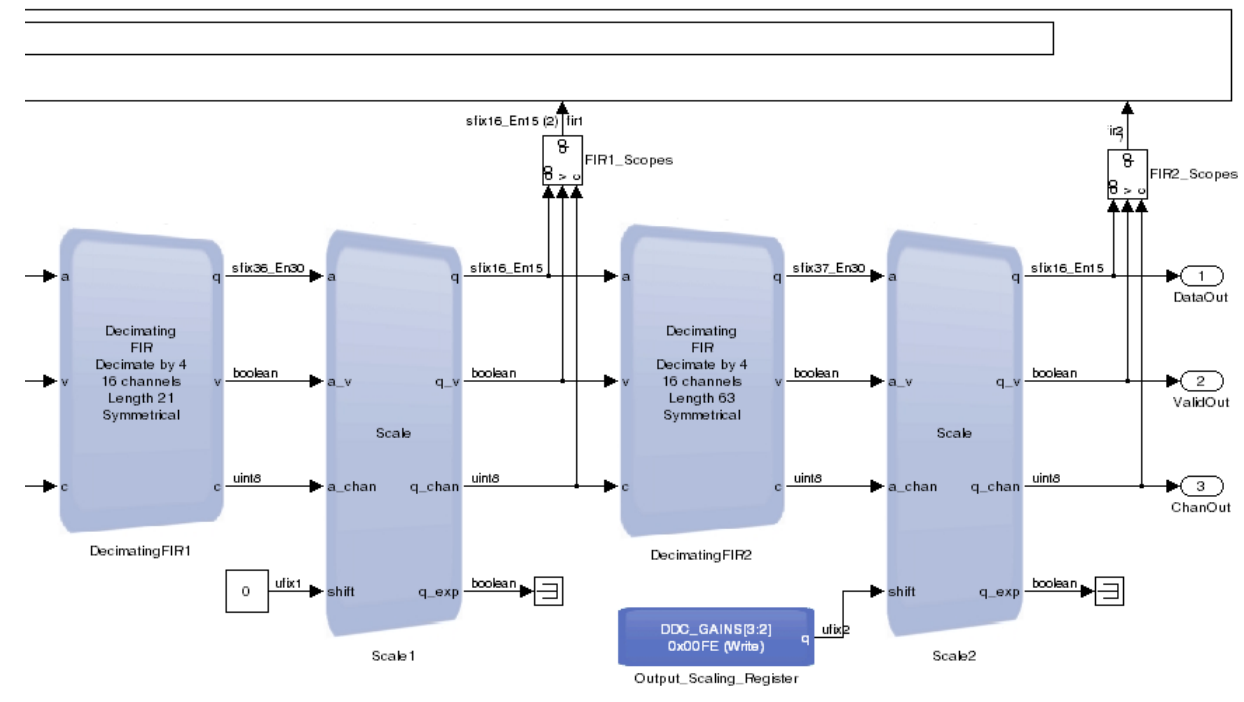
Enter a scalar rather than a vector value to indicate that the scaling is static.

Decimating FIR Blocks

The last part of the datapath (Figure 7-5 on page 7-10) comprises two decimating finite impulse response (FIR) blocks (DecimatingFIR1 and DecimatingFIR2) and their corresponding scale blocks (Scale1 and Scale2).


These two stages are very similar, the first filter typically compensates for the undesirable pass band response of the CIC filter, and the second FIR fine tunes the response that the waveform specification requires.

Figure 7-5. DDCChip Datapath (Decimating FIR and Scale Blocks)




The first decimating FIR decimates by a factor of 4.

The input rate per channel is the output sample rate of the decimating CIC, which is 16 times lower than the raw sample rate from the antenna.

 You can enter any MATLAB expression, so DSP Builder can extract the 16 out as a variable to provide additional parameterization of the whole design.

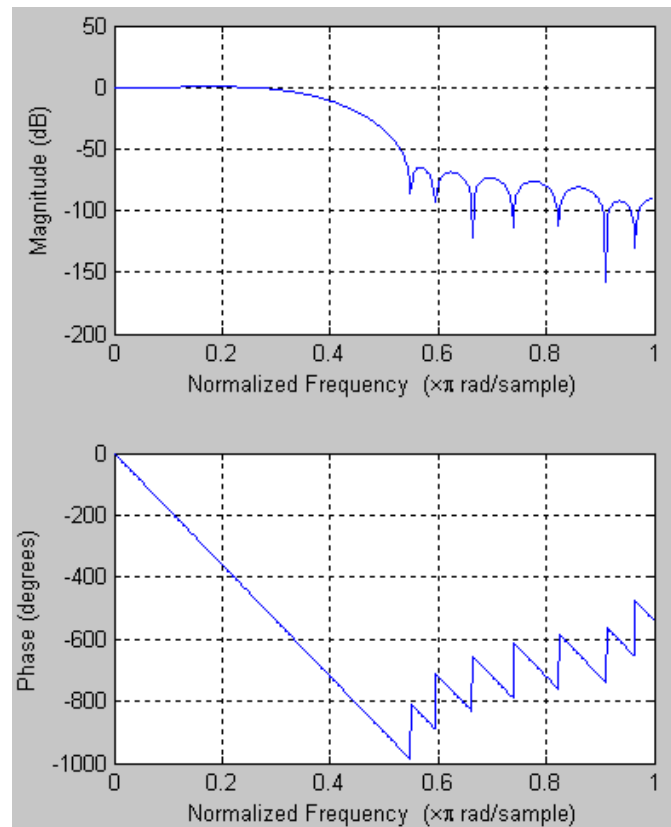
This filter performs decimation by a factor of 4 and the calculations reduce the size of the FIR filter. 16 channels exist to process and the coefficients are symmetrical.

The **Coefficients** field contains information that passes as a MATLAB fixed-point object (`fi`), which contains the data, and also the size and precision of each coefficient. Specifying an array of floating-point objects in the square brackets to the constructor to achieve this operation. The length of this array is the number of taps in the filter. At the end of this expression, the numbers 1, 16, 15 indicate that the fixed-point object is signed, and has 16-bit wide elements of which 15 are fractional bits.

 For more information about `fi` objects, refer to the MATLAB Help.

This simple design uses a low-pass filter. Figure 7-6 shows the response. In a real design, more careful generation of coefficients may be necessary.

Figure 7-6. DecimatingFIR1 Magnitude and Phase Responses



The output of the FIR filter fits onto a single wire, but because the data reduces further, there is a longer gap between frames of data.

Access a report on the generated FIR filter from the Help page.

You can scroll down in the Help page to view the port interface details. These match the hardware block, although the RTL has additional ports for clock, reset, and the bus interface.

The report shows that the input data format uses a single channel repeating every 64 clock cycles and the output data is on a single channel repeating every 256 clock cycles.

Details of the memory map include the addresses DSP Builder requires to set up the filter parameters with an external microprocessor.

You can show the total estimated resources by clicking on the DSP Builder menu, pointing to **Resources**, and clicking **Device**. Altera estimates this filter to use 338 LUT4s, 1 18×18 multiplier and 7844 bits of RAM.

The Scale1 block that follows the DecimatingFIR1 block performs a similar function to the DecimatingCIC block.

The DecimatingFIR2 block performs a second level of decimation, in a very similar way to DecimatingFIR1. The coefficients use a MATLAB function. This function (`fir1`) returns an array of 63 doubles representing a low pass filter with cut off at 0.22. You can wrap this result in a `fi` object:

```
fi(fir1(62, 0.22),1,16,15)
```

Building the Design Example

To build the design example, perform the following steps:

1. Simulate the design example in Simulink.
2. Explore the generated files.
3. Simulate the generated RTL in ModelSim.
4. Use the Quartus II software to synthesize and fit the RTL.

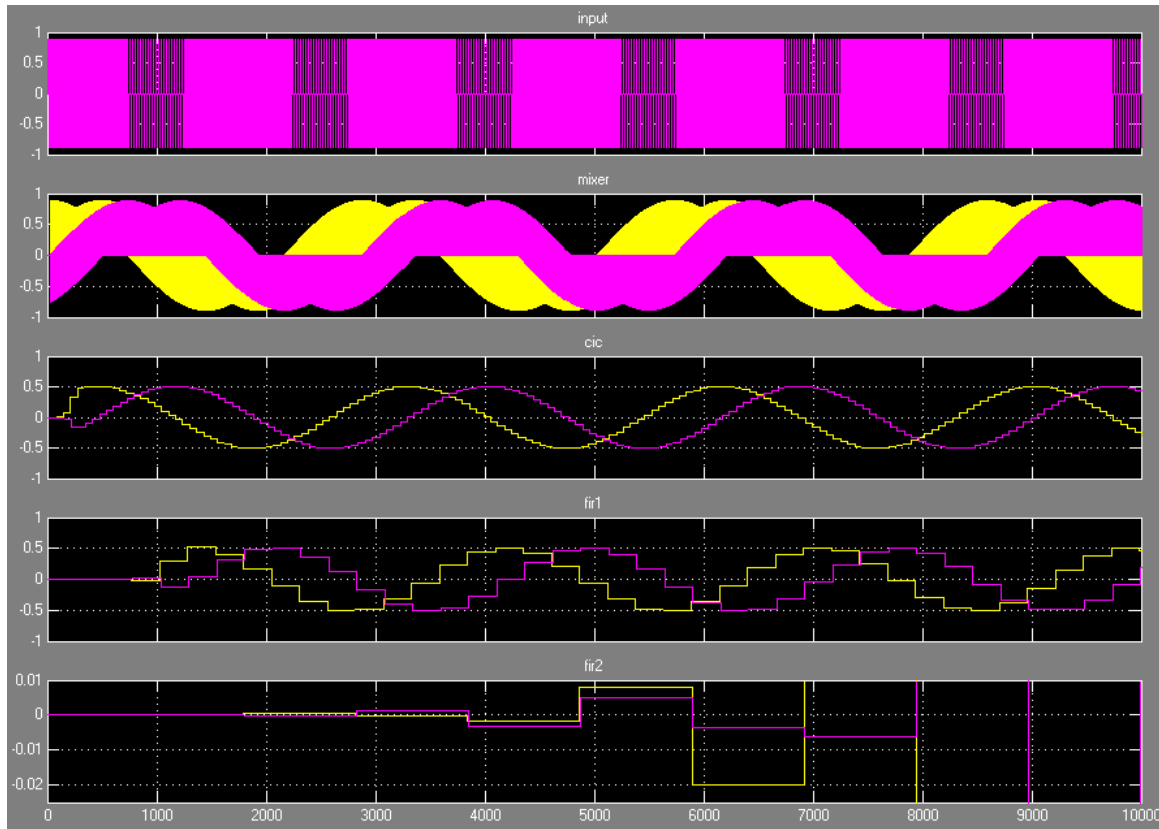
Simulating the Design Example in Simulink

You can run the simulation in Simulink by typing the following command in the MATLAB window:

```
sim('demo_ddc', 550000.0*demo_ddc.SampleTime);
```

The IScope block (Figure 7-7) shows the first two channels (1 real and 1 complex for the first carrier) of data (magenta and yellow) as the input signals. The first trace shows the rapidly changing input signal that the testbench generates. The second signal shows the result of the mixer. This slowly changing signal contains the information to be extracted, plus a lot of high frequency residue. Applying the series of low-pass filters and decimating results in the required data.

Figure 7-7. Simulation Results Shown in the IScope Block

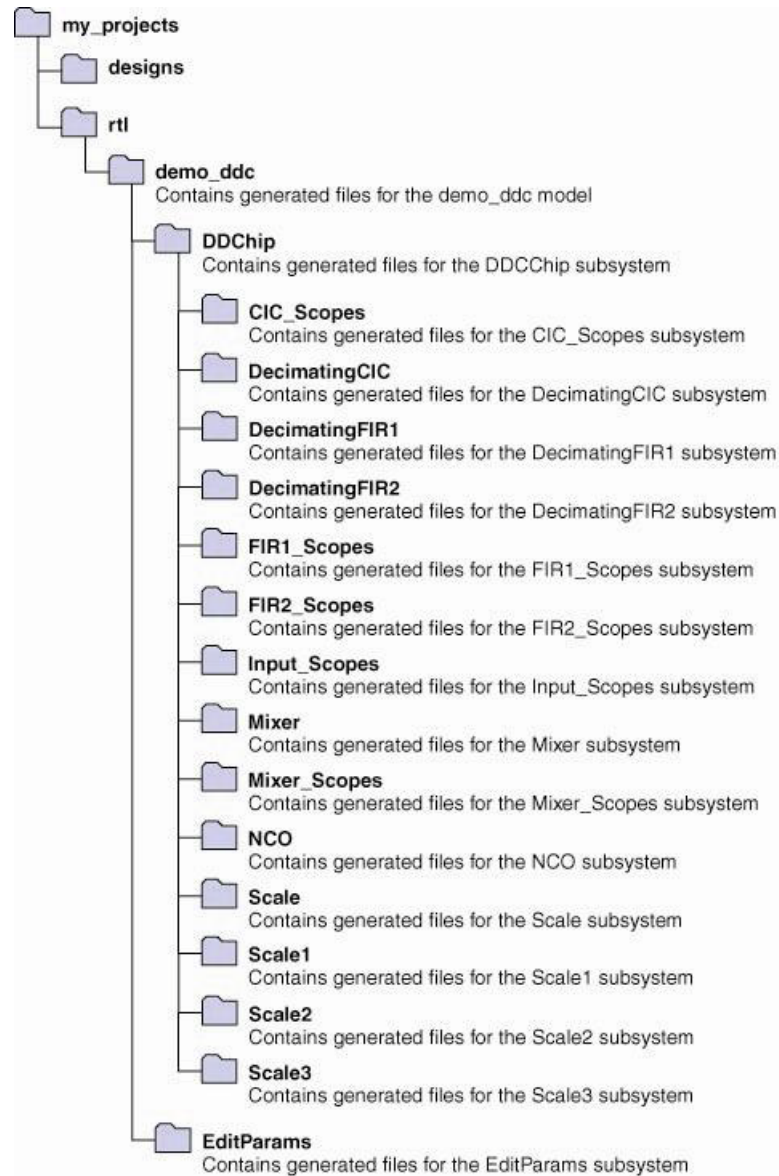


Exploring the Generated Files

If you turn on the **Generate Hardware** option in the parameters for the **Control** block, every time the simulation runs, DSP Builder synthesizes the underlying hardware, and writes out VHDL into the directory you specify.

DSP Builder creates a directory structure that mirrors the structure of your model. The root to this directory can be an absolute path name or as a relative path name. For a relative path name (such as `../rtl`), the directory structure is relative to the MATLAB current directory. For example, if the current working directory is `C:\my_projects\designs`, DSP Builder creates the structure that Figure 7-8 shows.

Figure 7-8. Generated Directory Structure for the DDC Design Example



Separate subdirectories exist corresponding to each hierarchical level in your design.

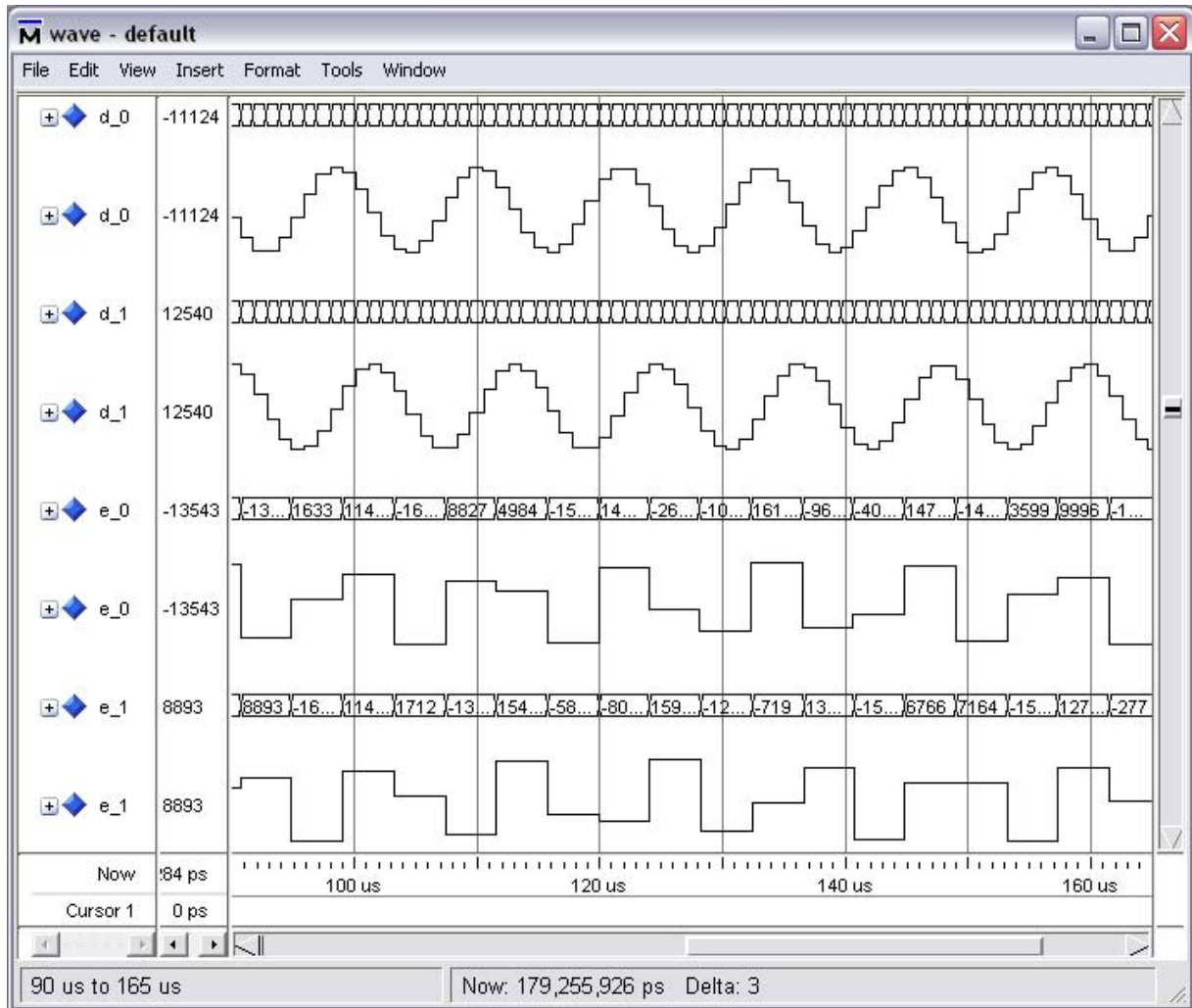
Table 7-1 lists the generated files for the DDC design.

Table 7-1. Generated Files for the DDC Design Example

File	Description
rtl directory	
demo_ddc.xml	An XML file that describes the attributes of your model.
demo_ddc_entity.xml	An XML file that describes the boundaries of the system (for Signal Compiler in designs that combine blocks from the standard and advanced blocksets).
rtl/demo_ddc subdirectory	
<block name>.xml	An XML file containing information about each block in the advanced blockset which translates into HTML on demand for display in the MATLAB Help viewer and for use by the DSP Builder menu options.
demo_ddc.vhd	This is the top-level testbench file. It may contain non-synthesizable blocks, and may also contain empty black boxes for Simulink blocks that are not fully supported.
demo_ddc.add.tcl	This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.
demo_ddc.qip	This file contains all the assignments and other information DSP Builder requires to process the demo_ddc design example in the Quartus II software. The file includes a reference to the .qip file in the DDCChip subsystem hierarchy.
<block name>.vhd	DSP Builder generates a VHDL file for each component in your model.
demo_ddc_DDCChip_entity.xml	An XML file that describes the boundaries of the DDCChip subsystem as a black box (for Signal Compiler in designs that combine blocks from the standard and advanced blocksets).
DDCChip.xml	An XML file that describes the attributes of the DDCChip subsystem.
*.stm	Stimulus files.
safe_path.vhd	Helper function that ensures a pathname is read correctly in the Quartus II software.
safe_path_msim.vhd	Helper function that ensures a pathname is read correctly in ModelSim.
rtl/demo_ddc<subsystem> subdirectories	
Separate subdirectories exist for each hierarchical level in your design. These subdirectories include additional .xml .vhd, .qip and .stm files describing the blocks contained in each level. Also additional .do and .tcl files exist, which it automatically calls from the corresponding files in the top-level of your model.	
<subsystem>_atb.do	Script that loads the subsystem automatic testbench into ModelSim.
<subsystem>_atb.wav.do	Script that loads signals for the subsystem automatic testbench into ModelSim.
<subsystem>/<block>/*.hex	Intel format .hex files that initialize the RAM blocks in your design for either simulation or synthesis.
<subsystem>.sdc	Design constraint file for TimeQuest support.
<subsystem>.tcl	Use this Tcl file to setup a Quartus II project.
<subsystem>_hw.tcl	A Tcl script that loads the generated hardware into Qsys.

The **Signal View Depth** parameter in the Control block determines the signals that add to the Wave window. Where appropriate, the signals automatically display in analog format (Figure 7-9). To display a particular signal, attach a Simulink scope, regenerate, and resimulate.

Figure 7-9. Simulation Results in the ModelSim Wave Window for the DDC Design Example



Most ModelPrim blocks support floating-point data types. DSP Builder generates a parallel datapath optimized for Altera FPGAs from the Simulink model.

Floating-point designs are useful in:

- Scientific applications
- Numerical algorithms
- High-dynamic range data designs
- Statistical modelling

Fixed-point designs often cannot support data with a high dynamic range unless the design explicitly uses a high precision type. Floating-point designs can represent data over a high dynamic range with limited precision. A compact representation makes efficient use of memory and minimizes data widths. The lowest precision type that DSP Builder supports is *float16_m10*, otherwise known as binary16, which occupies 16 bits of storage. It can represent a range between -2^{16} to $+2^{16}$ (exclusive) and non-zero magnitudes as small as 2^{-14} .

Typically, fixed-point designs may include fixed-point types of various bit widths and precisions. When you create fixed-point designs, keep variations in word growth and word precision within acceptable limits. When you create floating-point designs, you must limit rounding error to ensure an accurate result. A floating-point design typically has only one or two floating-point data types.

DSP Builder Advanced provides a comprehensive library of elementary mathematical functions with complete support for all floating-point types. Each core is parameterized by precision, clock frequency, and device family.

Features

- Variable precision. Seven floating-point precisions including half-precision, single-precision, and double-precision data.
- High throughput. Architecture adapts to f_{MAX} .
- Bit-accurate simulation.
- Efficient DSP block usage across range of device families .
- Standards compliant. Fundamental components (add, sub, mult, and div) are IEEE compliant. Elementary mathematical functions are OpenCL compliant.
- Configurable. Trade off logic utilization against accuracy. Significant DSP block reduction using faithful rounding.

Supported Floating-Point Data Types and Logic Usage

Table 8-1 lists the floating-point data types that DSP Builder supports.

Table 8-1. Supported Floating-Point Data Types

Precision	Range	Smallest Increment	Relative Error	Description
float16_m10	-2^{16} to $+2^{16}$	2^{-14}	2^{-11}	Binary16 (half-precision IEEE 754-2008).
float26_m17	-2^{128} to $+2^{128}$	2^{-126}	2^{-18}	—
float32_m23	-2^{128} to $+2^{128}$	2^{-126}	2^{-24}	Single-precision IEEE 754.
float35_m26	-2^{128} to $+2^{128}$	2^{-126}	2^{-27}	—
float46_m35	-2^{512} to $+2^{512}$	2^{-510}	2^{-36}	—
float55_m44	-2^{512} to $+2^{512}$	2^{-510}	2^{-45}	—
float64_m52	-2^{1024} to $+2^{1024}$	2^{-1022}	2^{-53}	Double-precision IEEE 754.

Round-Off Errors

Every mathematical operation on floating-point data incurs a round-off error. For the fundamental operations (add, subtract, multiply, divide) this error is determined by the rounding mode:

- **Correct.** A typical relative error is half the magnitude of the LSB in the mantissa.
- **Faithful.** A typical relative error is equal to the magnitude of the LSB in the mantissa.

The relative error for float16_m10 is approximately 0.1% for faithful rounding, and 0.05% for correct rounding. The rounding mode is a configurable mask parameter.

The elementary mathematical functions conform to the error tolerances specified in the OpenCL standard. In practice, the relative error exhibited by the DSP Builder mathematical library lies comfortably within the specified tolerances.

Bit cancellations can occur when subtracting two floating-point samples that are very close in value, which can introduce very large errors. You need to take the same precautions with floating-point designs as numerical software to prevent bit cancellations.

Trading Off Logic Utilization and Accuracy

If your design exceeds accuracy but is using too much hardware:

- Use the next lowest precision
- Use faithful rounding instead of correct rounding
- Enable the fused datapath option

Each of these changes reduces logic utilization at the expense of accuracy. A design may use more than one floating-point precision for different sections of the circuit, however if there are too many different precisions you will need to have more type conversion blocks. Each convert block will increase logic utilization.

Upgrading Pre v14.0 Designs

DSP Builder designs in v14.0 onwards have floating-point data turned on by default, which provides access to all seven floating-point data types. Floating-point designs from before v14.0 have a dedicated EnhancedPrecision block in the primitive subsystem. These designs work correctly in v14.0, but Altera recommends you find and remove any EnhancedPrecision block from your primitive subsystems.

Floating-Point Sine Wave Generator Tutorial



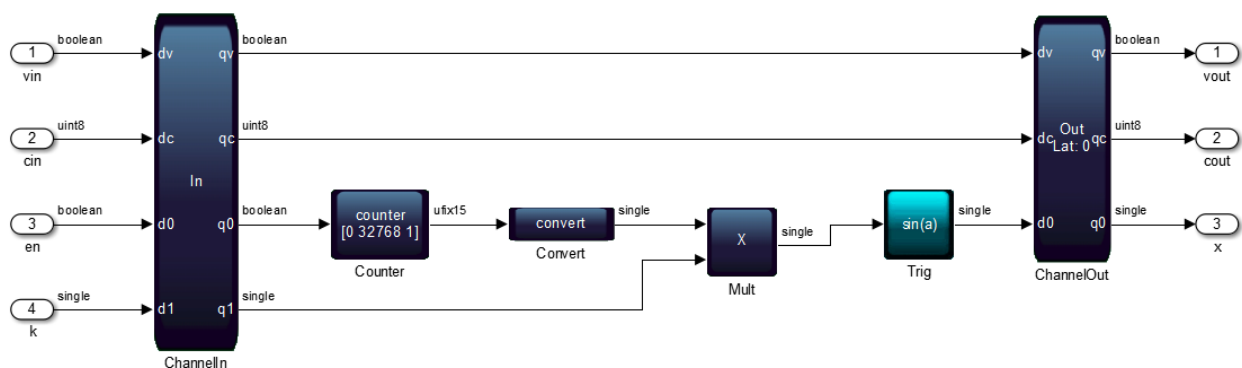
For more information on floating-point design examples that Altera provides, refer to [“Floating Point” on page 6–23](#).

Creating a Sine Wave Generator

To create a sine wave generator:

1. In Simulink, click **DSP Builder > New Model wizard** and create a floating-point ModelPrim (simple) model.
2. Create a primitive subsystem with the following components:
 - ChannelIn and ChannelOut
 - SynthesisInfo (configure to **Scheduled**)
 - Counter with a large period (e.g. 32,768) and incrementing in steps of 1
 - Convert block (set the mask parameter **Output data type mode** to **single**)
 - Mult block
 - Trig block (configure the **Mask parameter function** to **sin(x)**)
3. Connect the blocks. ([Figure 8–1](#))

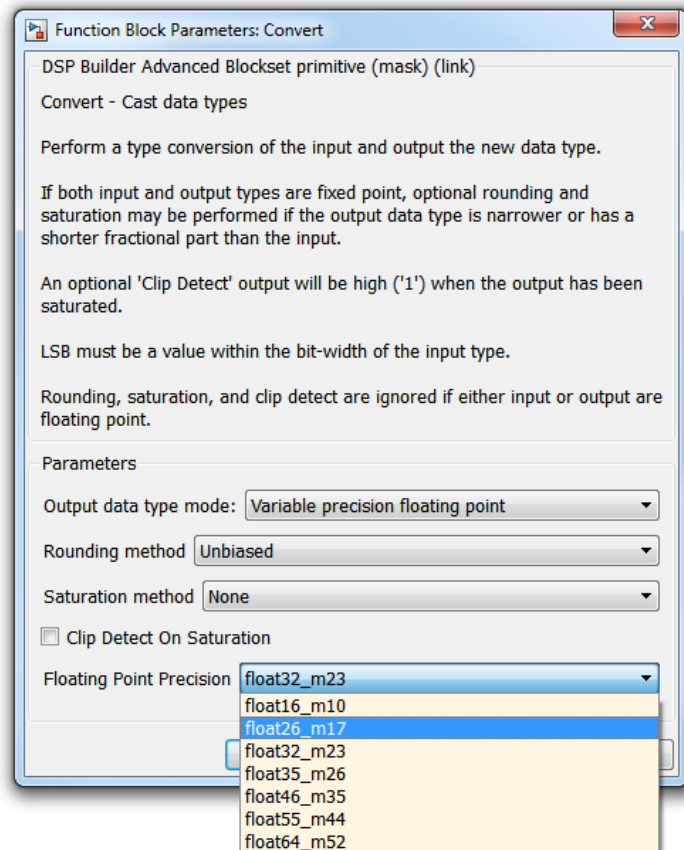
Figure 8–1. Connect Sine Wave Generator Blocks



4. Simulate the design.
5. Connect the single-precision input of the subsystem to a Simulink built-in source. For example, repeating sequence stair.

6. Set the repeating sequence stair block parameter **Output Data Type** to **single**. Hence, both inputs to the Mult block are single. This data type propagates through to the Trig block.
7. Simulate this design with hardware generation turned on. DSP Builder generates HDL files for a single-precision IEEE multiply, a single-precision sine function, and a fixed-to-float conversion component.
8. Click **DSP Builder > Resource Usage > Resource Usage Report** and record the DSP and LUT usage for the design.
9. To change the floating-point precision of the synthesized design, insert a Convert block on the floating-point input wire.
10. Parameterize the Convert block:
 - a. Set **Output data type mode** to **Variable precision floating point**.
 - b. In the **Floating Point Precision** drop-down menu select **float26_m17**.

Figure 8-2. Floating-Point Precision



11. Apply the same parameters to the Convert block in the primitive subsystem.

12. To connect the floating-point output port of the subsystem to a scope, or some other built-in Simulink sink block:
 - a. Insert a Convert block on the floating-point output wire.
 - b. Set the **Output data type mode** to **double**.



If you do not connect a Convert block, you cannot simulate our design. Simulink scopes do not recognize the DSP Builder custom floating-point type.

13. Simulate the design to generate HDL for the reduced precision multiplier, sine function, and fixed-to-float convert.
14. Re-examine the resource usage report. The DSP and LUT utilization is significantly lower than for the single-precision design.

Using Data Type Variables to Parameterize Designs

Commonly, you change the precision of a design by using scripts. However, writing scripts to update the floating-point precision drop-down menu for all blocks in the design is tedious. Use data-type variables to parameterize designs by data type.

1. At the MATLAB console, initialize a variable with the following command:

```
>> inputType = dspba.vpfloat(26,17)
inputType =
    Class: 'FLOAT'
      ExpBits: 8
    FracBits: 17
```

This MATLAB structure specifies the floating-point precision similar to how `fixdt()` specifies fixed-point precisions.

2. For the top-level Convert block on the input wire, open the parameter dialog box:
 - a. Set the **Output data type mode** to **Specify via dialog**.
 - b. Delete the **Output data type** field and type the variable name `inputType`.
3. Repeat for the Convert block in the primitive subsystem, so that the same data type propagates to both inputs of the Mult blocks.
4. Change the floating-point precision of the design, by assigning a different type to the variable `inputType`. You can also initialize the type variable using the data type name:

```
>> inputType = dspba.namedVPFloat('float26_m17')
>> inputType = dspba.namedVPFloat('single')
```

Using Data-Type Propagation

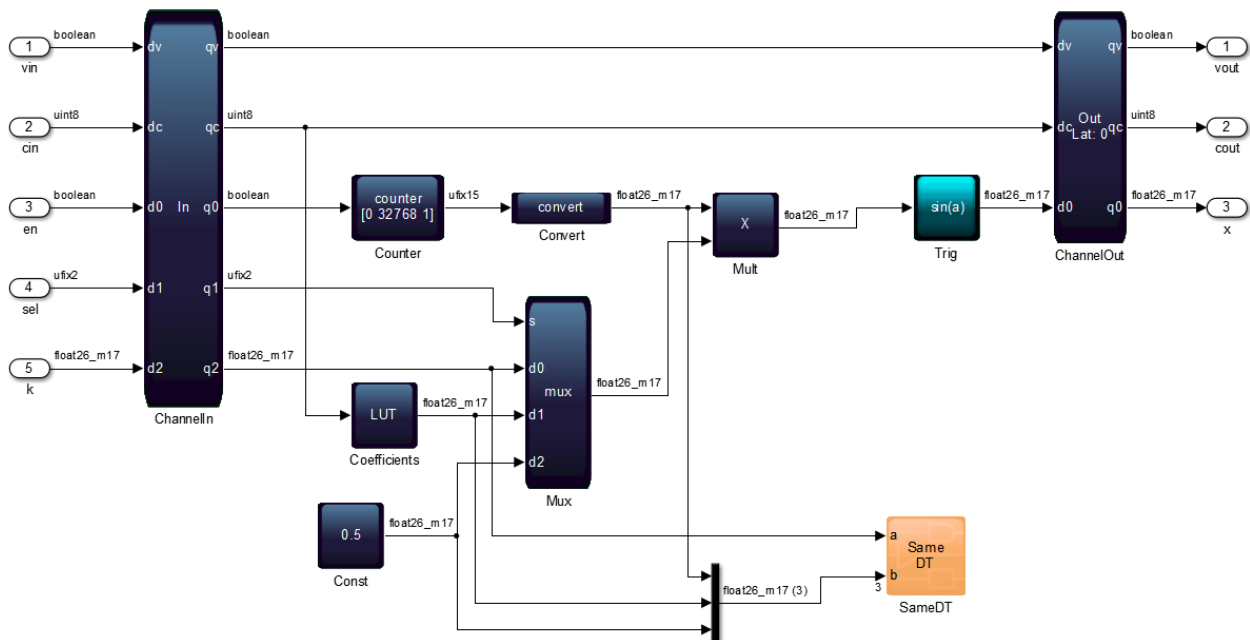
To simplify setup scripts for parameterized designs, use data-type propagation.

1. Add the DSP Builder custom SameDT block to your design. Do not use the built-in Simulink same-DT block, which does not propagate data types.

2. Use any of the following blocks to allow you to back propagate data types via their output ports:
 - Const
 - Lut
 - Convert
 - ReinterpretCast
3. Set the **Output data type mode** parameter for these blocks to **Inherit via back propagation**. Using this option and the custom **SameDT** block minimizes scripting for setting up data types in your design.

Figure 8-3 shows the data type **float26_m17** propagating from the input pin 5 (k) through the **d2-q2** port of the **ChannelIn** block and then to the **SameDT** block. The data type propagates via the built-in multiplex to three different wires, and then back propagates via the respective output ports of the **Convert** block, (coefficients) **LUT** block, and **Const** block.

Figure 8-3. Data Type Propagation



Testbench Verification

The Simulink simulation model generates the stimulus files for the automated testbench. A multiple precision floating-point library processes the variable precision floating-point signal values. All functions round the output results to the nearest representable value, even if you configure the fundamental operators to use faithful rounding. Also, the elementary mathematical functions do not need to round to nearest to comply with the IEEE754 standard. Hence, the hardware does not always output the same bit pattern as the Simulink simulation. DSP Builder provides tools to help analyze floating-point signals when simulating your designs

Tuning ATB Thresholds

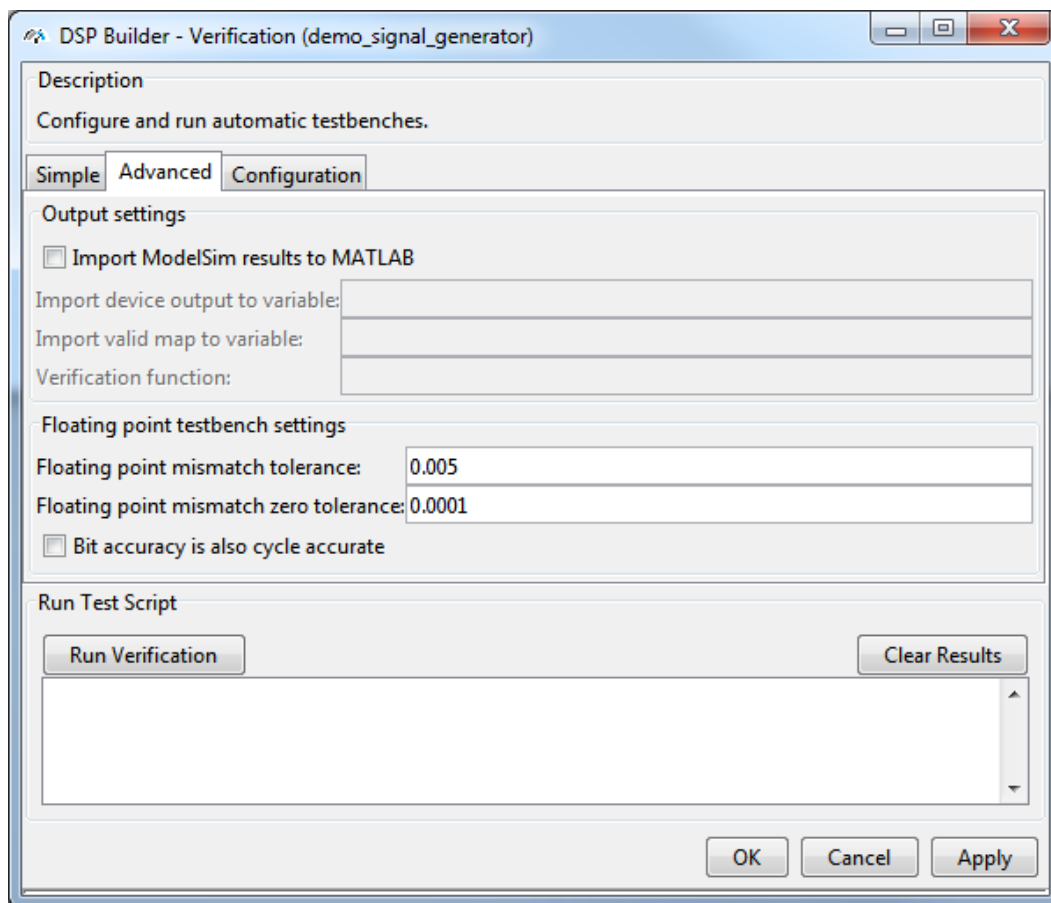
If your design uses floating-point components, the autogenerated testbench uses a special floating-point comparison when detecting mismatches. Two thresholds influence the sensitivity of the mismatch detection:

- Floating-point mismatch tolerance, which is the largest relative error that is not flagged as a mismatch
- Floating-point mismatch zero tolerance, which is the largest magnitude for a signal value to be considered as equivalent to zero.

To override the default values of these thresholds:

1. Click **DSP Builder > Verify Design > Advanced**.
2. Enter new values in **Floating-point testbench settings**.

Figure 8-4. Floating-Point Testbench Settings



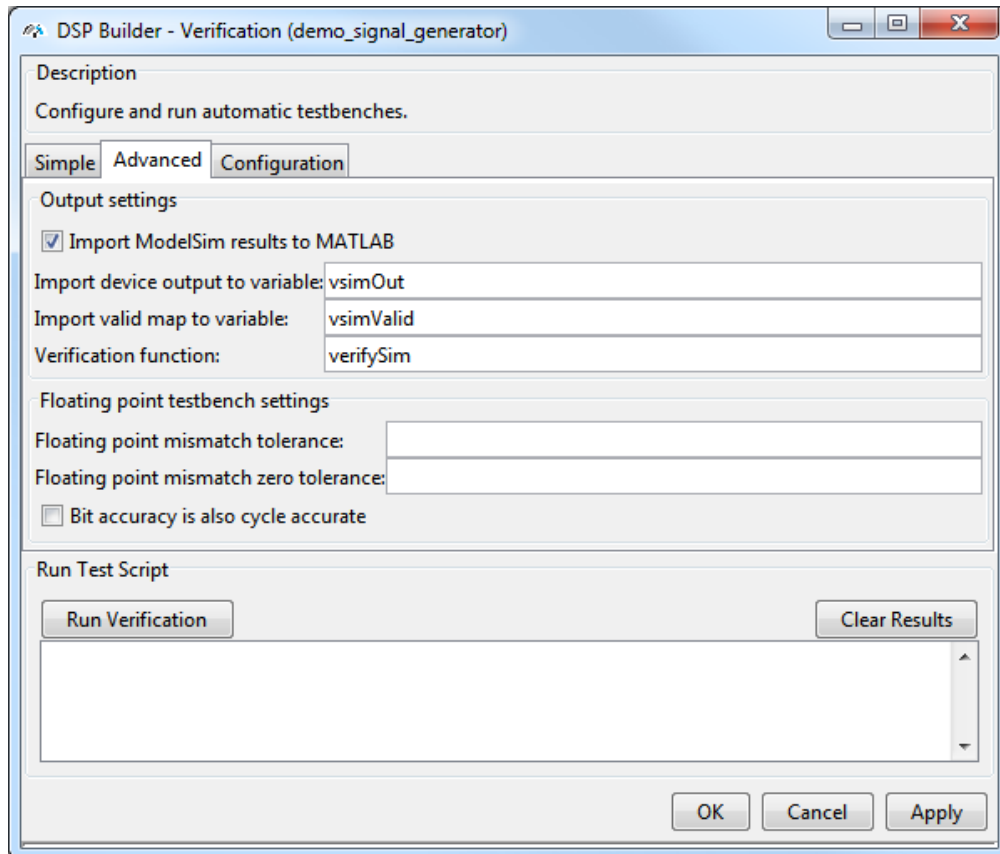
Writing Application Specific Verification

Generally, use the threshold method for detecting mismatches in hardware for most designs. For more sophisticated designs you can write your own application specific verification function.

1. Click **DSP Builder > Verify Design > Advanced**.

2. Turn on **Import ModelSim results to MATLAB**.
3. Enter a MATLAB variable name in **Import device output to variable**.
4. Optionally, enter a different variable name in **Import valid map to variable** field.
5. Enter the name of the verification m-function in **Verification function**.

Figure 8–5. Application Specific Verification



6. Simulate and generate the hardware. DSP Builder modifies the ATB to write to a file the output signal values that ModelSim simulates:
 - a. Click **DSP Builder > Verify Design**
 - b. Turn on **Run simulation** and **Verify at device level**,
 - c. Click **OK**.

MATLAB stores the simulation results using field names derived from the names of the output ports in your design.

```
>> atbPaths = vsimOut.keys; vsimOut(atbPaths{1})
```

```
ans =
```

```
vout: [2000x1 embedded.fi]
```



```
vout_stm: [2000x1 embedded.fi]
cout: [2000x1 embedded.fi]
cout_stm: [2000x1 embedded.fi]
x: [2000x1 double]
x_stm: [2000x1 double]
```

The fields ending in `_stm` are from the stimulus files that Simulink normally writes out during simulation. You can use these as the golden standard against which to compare the simulated hardware output. The verification function you specified is started, passing this struct as the first parameter.

Using Bit-Accurate Simulation

DSP Builder supports bit-accurate simulation of floating-point designs.

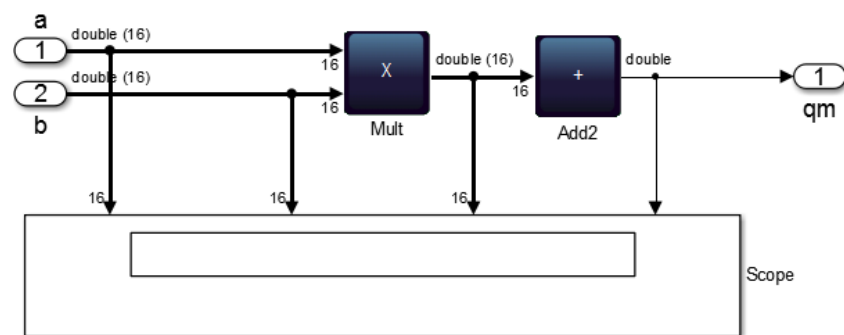
1. Open the `SynthesisInfo` block parameter dialog box.
2. Turn on **Bit Accurate Simulation**.

When you simulate the design, the Simulink simulation is not based on the multiprecision floating-point library. It is based on signal values output by the `ChannelOut` block, which exactly match what you expect from hardware.

Adder Trees and Scalar Products

The `matmul_flash_RS` and `matmul_flash_RD` design examples use vector signals with floating-point components. Figure 8-6 shows how to compute an inner product by combining a component-wise multiply block with an adder tree.

Figure 8-6. Mutliply block with Adder Tree



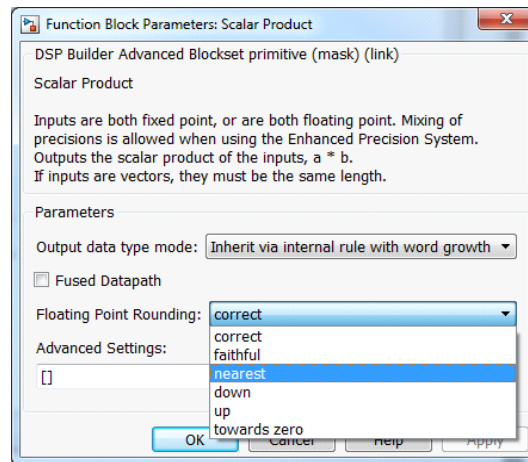
The `gemm_flash` design example is a generalized matrix multiplication design that uses the `Scalar Product` block to calculate an inner product. The adder tree and the `Scalar Product` block have similar parameters:

- Fused datapath. Enable this option to reduce logic utilization at the expense of IEEE compliance

- Rounding modes:
 - a. Nearest
 - b. Down (towards negative infinity)
 - c. Up (towards positive infinity)
 - d. Towards zero

When you turn on **Fused datapath**, you can select only the rounding modes **Nearest** and **Towards zero**. Logic utilization is highest when your design uses rounding mode **Nearest**.

Figure 8-7. Floating-Point Rounding



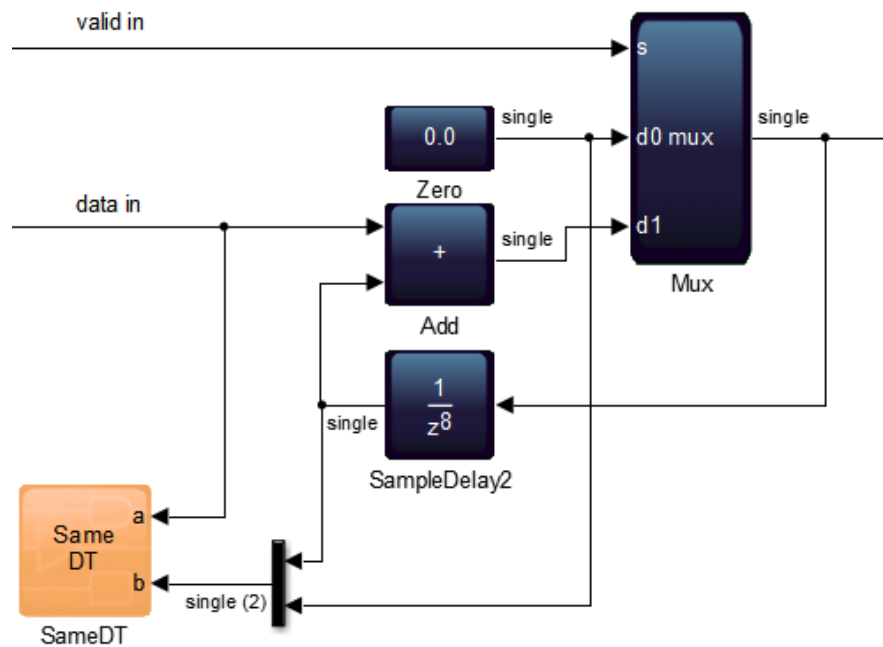
Creating Floating-Point Accumulators for Designs that Use Iteration

In DSP Builder, you can create fixed-point accumulators with one or more channels, but you cannot create floating-point equivalents. To create a floating-point accumulator:

1. If your design requires a single channel accumulator, use the **Acc** block.

2. Use multiple channels where the number of channels is at least as large as the latency of the floating-point adder. The lower limit on the number of channels depends on f_{MAX} , device family, and speed grade:
 - a. DSP Builder redistributes sample delays in your multiple channel accumulator. You cannot rely on DSP Builder preserving the reset state of zero. Use a 2:1 Mux block in your accumulator to make the initial state explicit.
 - b. Use the custom SameDT block to avoid any data type propagation problems in feedback cycles.

Figure 8–8. Multiple Channel Floating-Point Accumulator



Newton Root Finding Tutorial

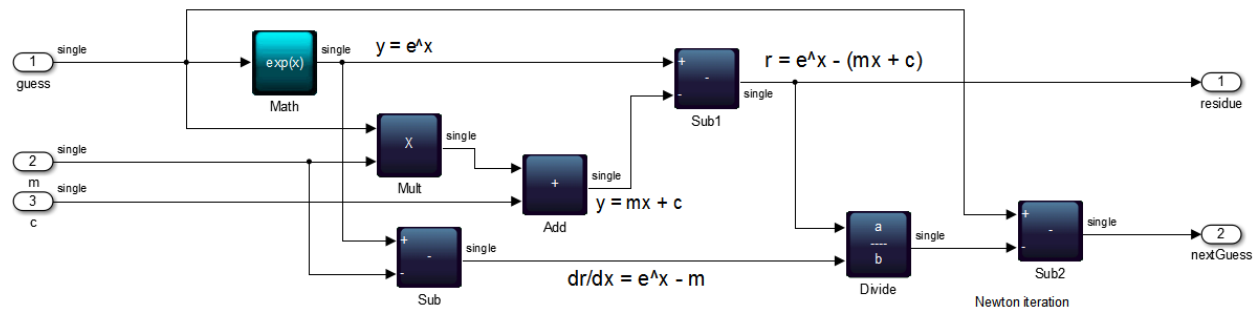
This tutorial implements a floating-point iterative algorithm. The tutorial also demonstrates how to exploit pipeline parallelism. Consider an application that finds the intersections of two equations:

$$y = e^x$$

$$y = mx + c$$

The design finds the roots of the equation, $e^x - mx - c = 0$, using Newton-Raphson iteration. Figure 8-9 shows the Newton part of the design, which derives an improved approximation to the root from the previous guess.

Figure 8-9. Newton Design



The following design examples show the various stages of the Newton root finding tutorial:

- demo_newton_iteration.mdl
- demo_newton_convergence.mdl
- demo_newton_valid.mdl
- demo_newton_control.mdl
- demo_newton_final.mdl

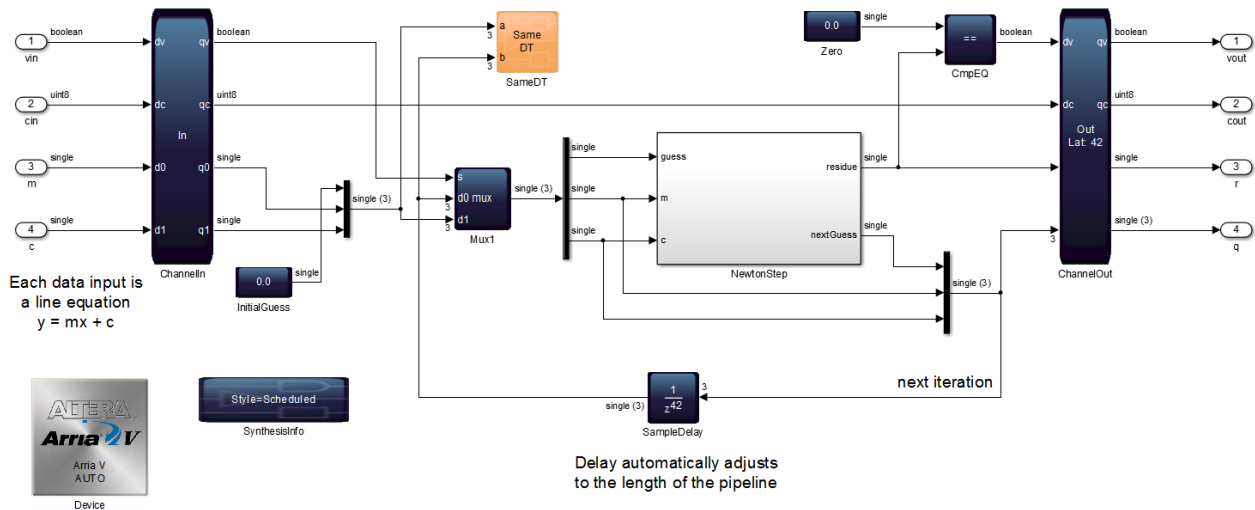
Implementing the Newton Design

To implement the Newton design:

1. Add and connect the blocks in Figure 8-9.
2. Reduce logic usage by configuring the Mult, Add, Sub, and Divide blocks to use faithful rounding.

3. Create the iteration loop by feeding back the output guess to the input guess through a SampleDelay block. The design detects when a sample finishes iterating by comparing the residue with zero.

Figure 8-10. Iteration Loop



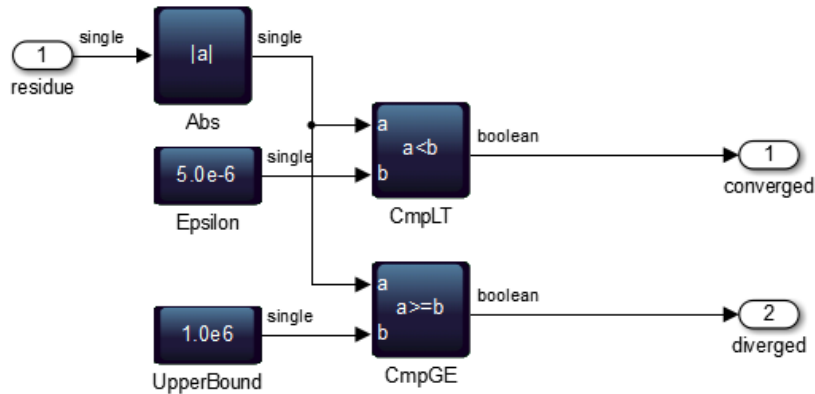
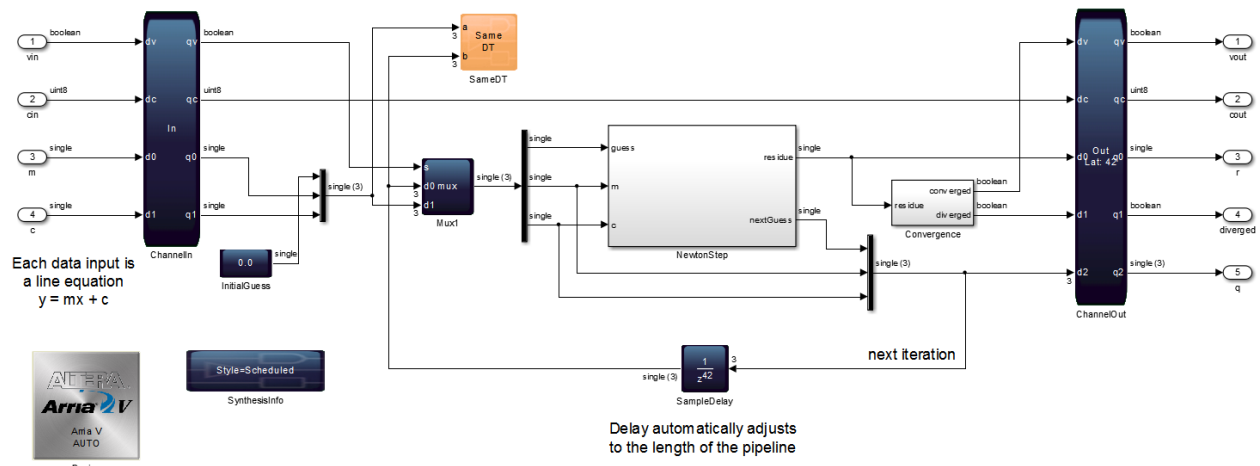
4. Ensure that the length of this delay is sufficiently large so that the scheduling succeeds.
5. Turn on the SampleDelay block **Minimum delay** parameter so that DSP Builder determines this length automatically.

Improving Floating-Point Designs

In floating-point designs when comparing against zero, many of the samples never terminate and circulate through the feedback loop indefinitely. Because of the round-off errors accumulating in the **NewtonStep** subsystem, the residue may never reach exactly zero for many of the data samples.

To improve the design:

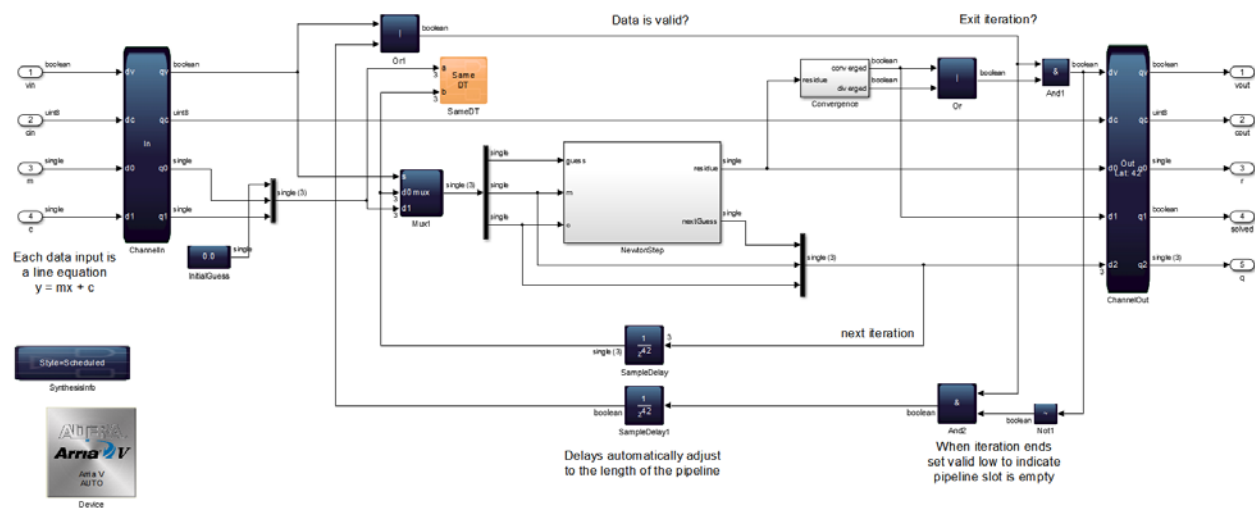
1. Use a subsystem to detect convergence and divergence of the iterative feedback loop

Figure 8-11. Subsystem to Detect Convergence and Divergence**Figure 8-12. Convergence and Divergence Subsystem in Newton Design**

2. Simulate the design. The number of valid samples on the output far exceeds the number of valid samples on the input.

3. To track which pipeline slots contain a valid sample, add a control signal path that lies parallel to the datapath feedback.

Figure 8-13. Newton Design with Control Signal Path

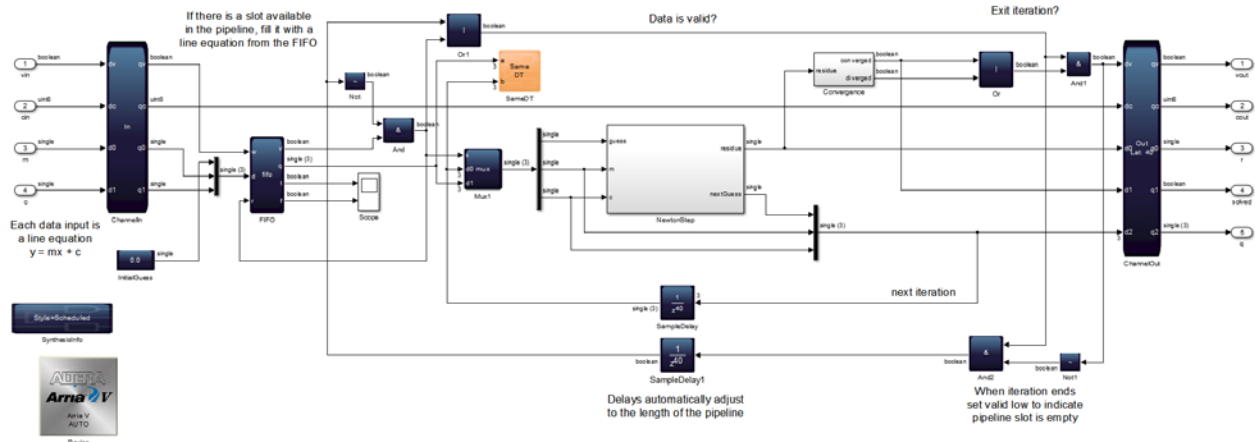


4. This 1-bit wide control path must be the same latency as the datapath.
 - a. Ensure that for both **SampleDelay** blocks you turn on **Minimum delay enabled**.
 - b. Set the **Equivalence Group** to the string, **newton**. When a sample converges to a root, DSP Builder outputs it as a valid solution and marks the pipeline slot as empty. If a sample diverges, DSP Builder marks the pipeline slot as empty but keeps valid low.
5. Simulate this version of the design and verify that the number of valid samples output equals the number of valid samples input.

The design may exceed the pipeline capacity if you provide too many valid samples on the input. The scheduled size of the sample delays indicates the maximum number of pipeline slots that are available for iterative computation. If you input more than this number, you risk overwriting previous valid samples before their iteration converges.

- To overcome this limitation, introduce a FIFO buffer for the input samples. When an empty pipeline slot becomes available at the front of the pipeline, DSP Builder removes a sample from the queue from the FIFO buffer and inserts it into the free slot to begin iterating around the **NewtonStep** feedback loop.

Figure 8-14. NewtonStep Feedback Loop



- Simulate the design and verify that you can safely input more valid samples than the pipeline depth of the iterative loop.
- Set the size of the FIFO buffer to governs the limit and ensure that it is as large as your application requires.

9. The rounding errors of the floating-point blocks can interact in such a way that a sample forever oscillates between two values: never converging and never diverging. To detect this oscillation add another control path in parallel to the datapath feedback to count the number of iterations each sample passes through.

Figure 8-15. Oscillation Detector

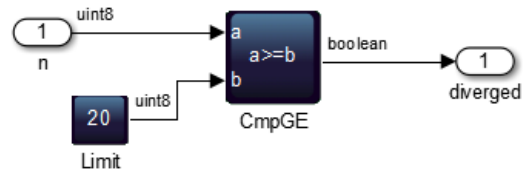
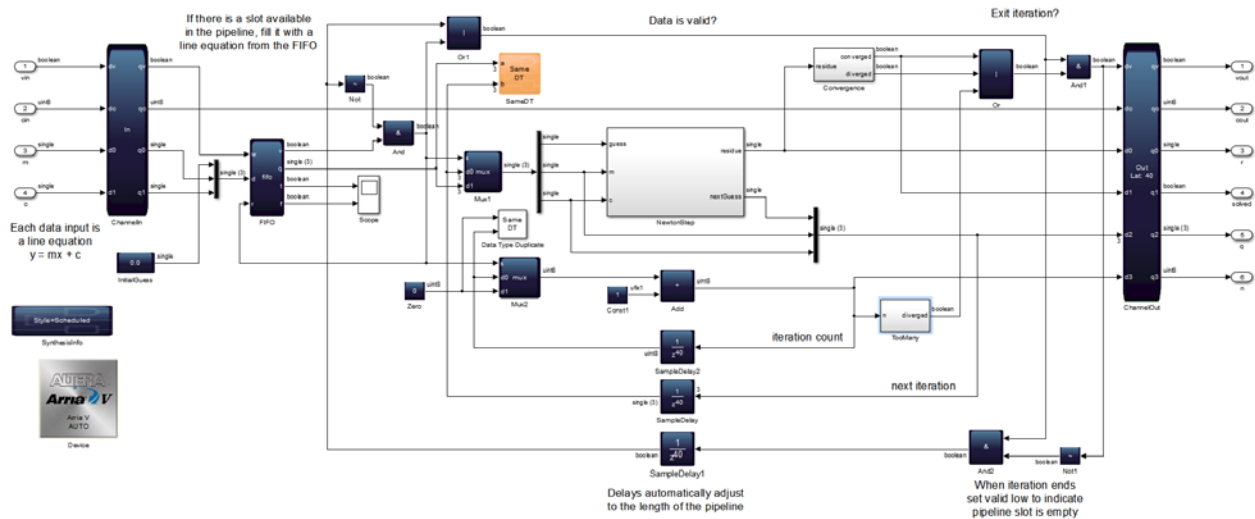





Figure 8-16. Newton Design with Oscillation Detector



-  The **TooMany** subsystem compares the iteration count against a threshold to detect divergence.
-  The **Mandelbrot_S** design example implements another iterative algorithm that shows parallel feedback loops for floating-point data and control paths.
-  The **matmul_CS** design example exploits both vector and pipeline parallelism. This example also shows how to incorporate memories in the feedback path to store intermediate results.

This chapter describes how you can combine blocks from the DSP Builder standard and advanced blocksets in the same design. The top-level design example can contain the `Control` and `Signals` blocks from the advanced blockset. However, you must embed the `Device` block and the functional blocks in the advanced blockset design in a lower level subsystem.



The DSP Builder standard blockset is a legacy product and Altera recommends you do not use it for new designs, except as a wrapper for advanced blockset designs.

The `Run ModelSim` block from the advanced blockset does not work in a combined blockset design. However, you can use the `TestBench` block from the standard blockset to generate a testbench and compare the simulation results with the ModelSim simulator.

You can still use the `Run Quartus II` block from the advanced blockset in a combined blockset design but it only creates a Quartus II project for the advanced blockset subsystem containing the `Device` block. Use a `Signal Compiler` block from the standard blockset to create a Quartus II project for the whole combined blockset design.

The mechanism for embedding an advanced blockset subsystem within a top-level DSP Builder design is similar to that for embedding a HDL subsystem as a black-box design, with the `Device` block from the advanced blockset taking the place of the HDL `Entity` block from the standard blockset.



DSP Builder generates the advanced blockset design when you simulate the design in Simulink. Perform this simulation before DSP Builder generates the top-level design example by running `Signal Compiler`.

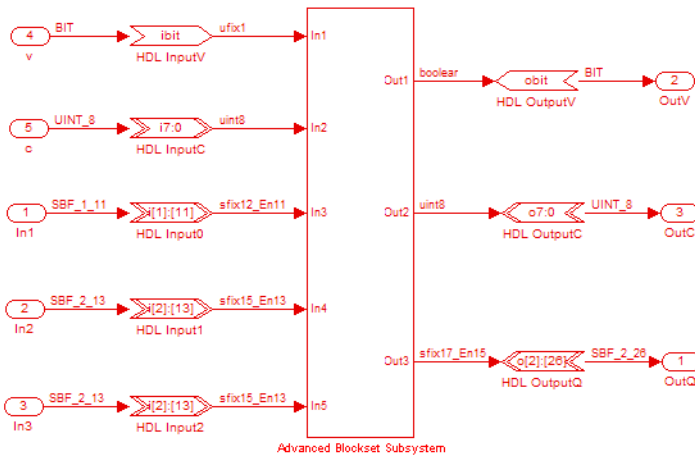
The following settings and parameters must match across the two blocksets in an integrated design:

- Use forward slash (/) separator characters to specify the hardware destination directory that you specify in the `Control` block as an absolute path.
- The device family that you specify in the `Device` block must match the device family you specify in the top level `Signal Compiler` block and the device on your development board. However, you can set the specific device to **Auto**, or have different values. The target device in the generated Quartus II project is the device that you specify in the `Signal Compiler` block. HIL specifies its own Quartus II project, which can have a different device provided that the device family is consistent.
- The reset type that you specify in the advanced blockset `Signals` block must be active **High**.
- When you run the `TestBench` for a combined blockset design, expect mismatches when the valid signal is low.

- The standard blockset does not support vector signals. To convert any vectors in the advanced blockset design, use multiplexer and demultiplexer blocks.

Use HDL Input and HDL Output blocks (Figure 9-1 on page 9-2) to connect the subsystem that contains the advanced blockset design. The signal dimensions on the boundary between the advanced blockset subsystem and the HDL Input/HDL Output blocks must match.

Figure 9-1. Advanced Blockset Subsystem Enclosed by HDL Input and Output Blocks



The signal types are on either side of the HDL Input/HDL Output blocks after you simulate the subsystem. If the signal types do not display, check that **Port Data Types** is turned on in the Simulink Format menu.

If the signal types do not match, there may be error messages of the form:

"Error (10344): VHDL expression error at <subsystem>_<HDL I/O name>.vhd (line no.): expression has N elements, but must have M elements"

For example in Figure 9-1, DSP Builder issues an error because the signal type for the HDL OutputQ block is incorrect. Change it from **Signed Fractional [2].[26]** to **Signed Fractional [2].[15]**.

After this change, the signal type shows as **SBF_2_15** (representing a signed binary fractional number with 2 integer bit and 15 fractional bits) in the standard blockset part of the design (before the HDL Input block). The same signal shows as **sfix17_En15** (representing a Simulink fixed-point type with word length 17 and 15 fractional bits) in the advanced blockset design (after the HDL Input block).



For more information about the fixed-point notation that the standard blockset uses, refer to the *Fixed-Point Notation* section in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*. For more information about Simulink fixed-point types, refer to the MATLAB help.

Using Signal Compiler in an Advanced Blockset Design

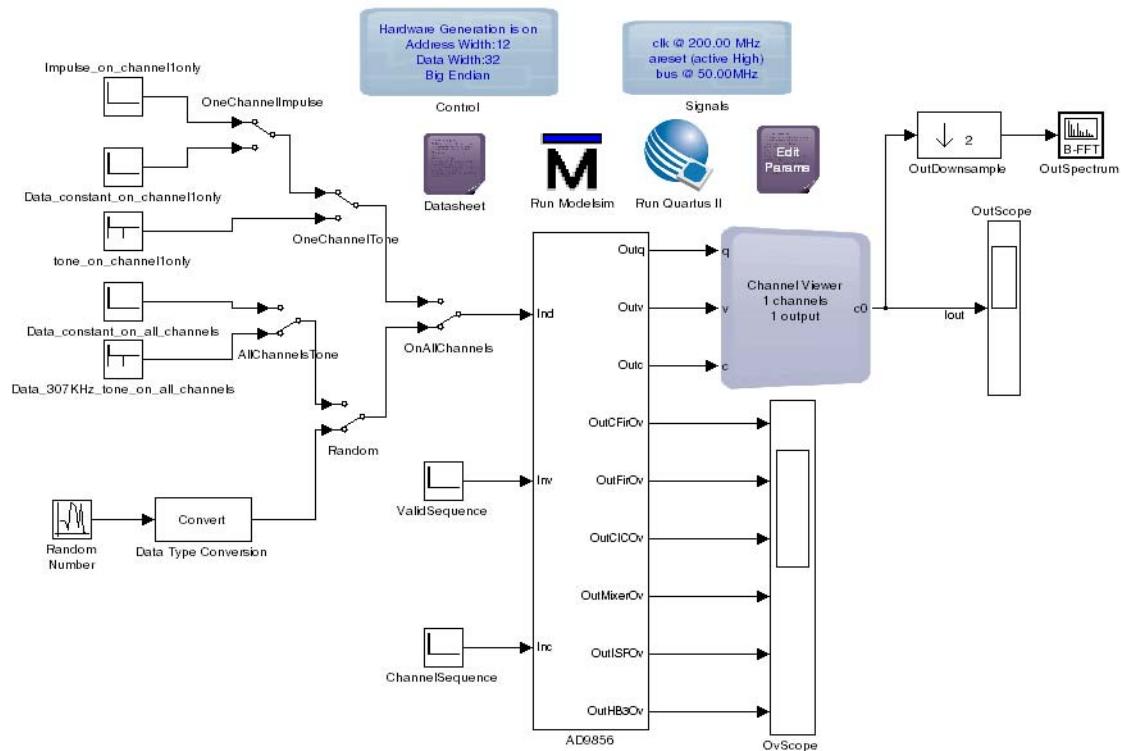
Figure 9-2 shows the 2-Channel Digital Up Converter advanced blockset design example (demo_AD9856).

The synthesizable system in this design is the AD9856 subsystem. Its inputs are Ind, Inv and Inc. It has nine outputs with three feeding a ChanView and six feeding a Scope.

To make the design interoperable with a Signal Compiler block in the top level, perform the following steps:

1. Select the subsystem, AD9856, the scope, OvScope, and the ChanView. Click **Create Subsystem** on the popup menu to create a Subsystem block that contains them.

Figure 9–2. AD9856 Design Example

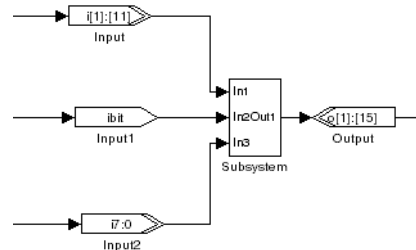


2. Add three Input blocks from the DSP Builder IO & Bus library immediately before the new Subsystem block: Input (with **Signed Fractional [1][11]** type), Input1 (with **Single Bit** type), and Input2 (with **Unsigned Integer 8** type).
3. Add an Output block immediately after the Subsystem block with **Signed Fractional [1][15]** type.



Steps 2 and 3 specify the boundaries between Simulink blocks and DSP Builder blocks.

Figure 9-3. Input, Output and Subsystem Blocks in Top-Level Model



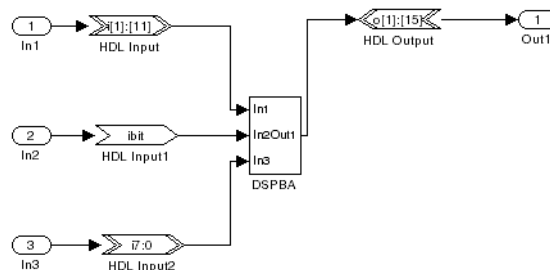
4. Open the Subsystem block and select the AD9856, OvScope, and ChanView blocks inside the subsystem. Click **Create Subsystem** on the popup menu to push these blocks down another level. Rename this subsystem (to for example DSPBA).
5. Add three HDL Input blocks from the DSP Builder **AltLab** library between the Simulink input ports and the DSPBA subsystem.
 - a. These blocks should have the same types as in Step 2: HDL Input (**Signed Fractional [1][11]**), HDL Input1 (**Single Bit**), and HDL Input2 (**Unsigned Integer 8**).
 - b. On the signed fractional HDL Input, set the **External Type** parameter to **Simulink Fixed-point Type**.
6. Add a HDL Output block between the subsystem and the subsystem output port with the same type as in Step 3 (**Signed Fractional [1][15]**).



Steps 5 and 6 specify the boundaries between blocks from the standard and advanced blocksets. The HDL Input and HDL Output blocks must be in a lower level subsystem than the Input and Output blocks. If they are at the same level, a NullPointerException error issues when you run Signal Compiler.

Figure 9-4 shows the new subsystem.

Figure 9-4. HDL Input, HDL Output and DSPBA Blocks in Subsystem



7. Open the DSPBA subsystem and the AD9856 subsystem.

8. Move the Device block from the AD9856 subsystem up a level into the DSPBA subsystem you create by making a copy of the existing Device block and then deleting the old one.


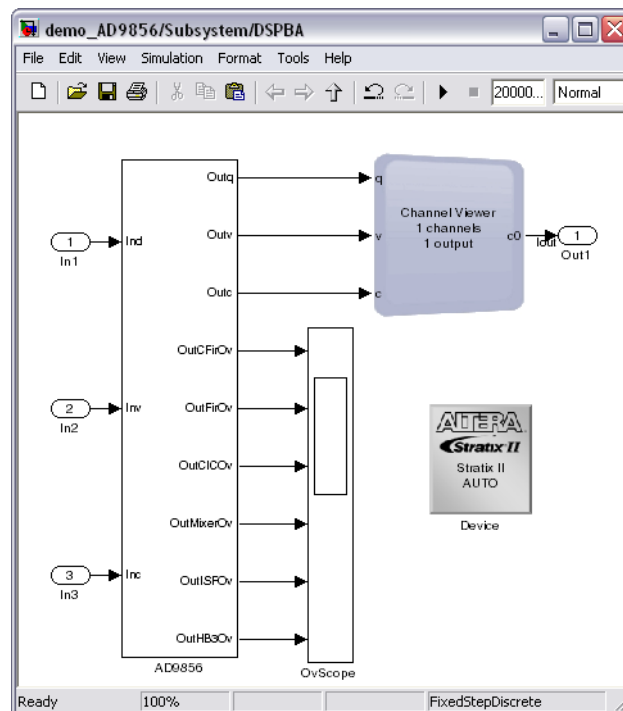

 The Device block detects the presence of a DSP Builder advanced subsystem and should be in the highest level of the advanced blockset design that does not contain any blocks from the standard blockset.

Figure 9-5 shows the DSPBA subsystem.

Figure 9-5. DSPBA Subsystem

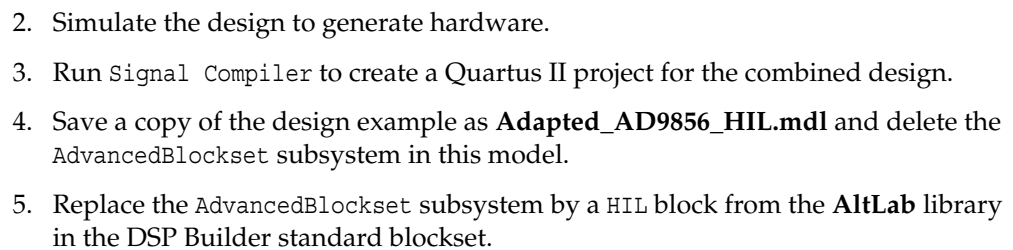


9. Open the Control block in the top level of the design and change the **Hardware Destination Directory** to an absolute path. For example: **C:/rtl**

 You must use forward slashes (/) in this path.

10. Add Signal Compiler, TestBench and Clock blocks from the DSP Builder **AltLab** library to the top-level model.
11. In the Signal Compiler block, set the **Family** to **Stratix II** to match the family specified in the Device block.
12. In the Clock block, set the **Real-World Clock Period** to **50 ns** so that it matches the **Clock Frequency** specified in the Signals block. Set the Reset Name to **aclr Active Low**.

Figure 9–7. Adapted_AD9856 Design Example



- Double-click on the HIL block to open the **Hardware in the loop** dialog box (Figure 9-8). In the first page of the dialog box, select the Quartus II project (**Adapted_AD9856_dspbuilder/Adapted_AD9856.qpf**) that was created in step 3. Select the clock pin (Clock) and reset pin (aclr) names. Set the channelIn signal type to **unsigned 8 bits**, dataIn to **signed [1].[11] bits**, and ChannelOut0 and ChannelOut1 to **signed [1].[15] bits**.

Figure 9-8. Hardware in the Loop Parameter Settings Page 1 for the AD9856 Example

Hardware in the loop - page 1 of 2

1. Select the Quartus II project
Adapted_AD9856.qpf

2. Select the clock pin
Clock

3. Select the reset pin
aclr

4. Identify the signed ports

Input ports

- channelIn (unsigned) 1.11 bits
- Clock (clock)
- dataIn (signed)
- bus_arreset
- aclr (reset)
- validIn
- bus_clk

Output ports

- bus_clk_reset_out 1.15 bits
- ChannelOut0 (signed)
- bus_clk_out
- ChannelOut1 (signed)

5. Select the reset level
Active_High

6. Choose a mode of operation

Burst mode allows faster HIL operation, but introduces latency on the outputs equaled to the burst length selected

☒ Burst Mode
Burst length 1024

Frame mode allows to compensate for the delay by re-synchronizing the outputs to the inputs.

☐ Frame Mode

Input sync channelIn

Output sync bus_clk_reset_out

Sampling period (-1 for inherited) 0

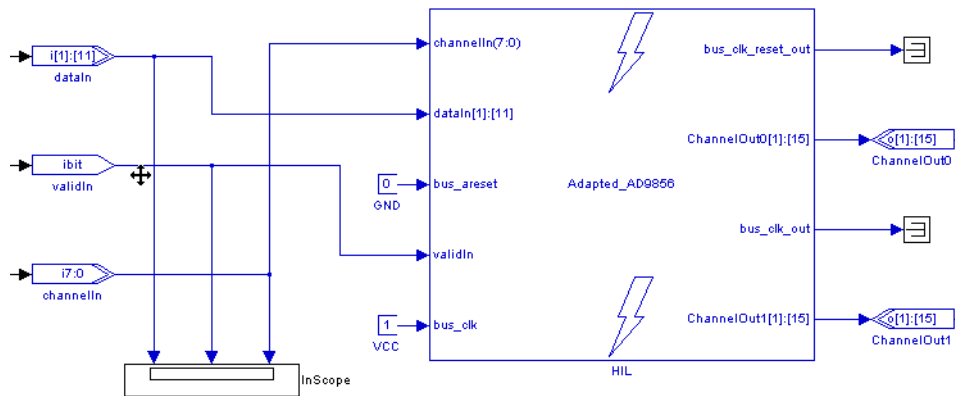
Next page Close

- Close the **Hardware in the loop** dialog box and connect the dataIn, validIn, channelIn, ChannelOut0, and ChannelOut1 ports to your design example (Figure 9-9 on page 9-9).



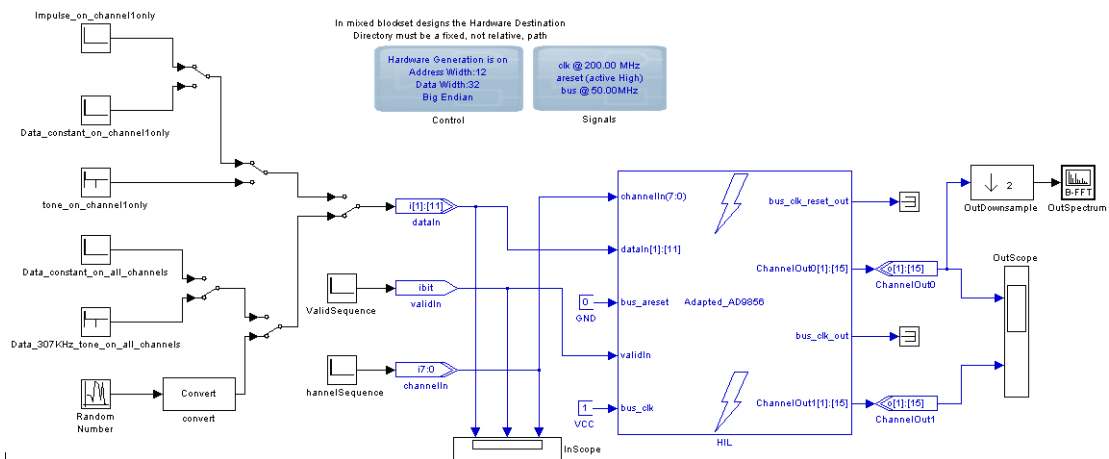
HIL simulation does not use the bus interface. Connect the `bus_aset` signal to a GND block and the `bus_clk` port to a VCC block from the standard blockset **IO & Bus** library. Connect the `bus_clk_out` and `bus_clk_reset_out` signals to Simulink Terminator blocks.

Figure 9-9. HIL Block in the Adapted_AD9856 Model



8. Clean up the HIL design example by removing the blocks that are not related to the HIL (Figure 9-10).

Figure 9-10. Cleaned Up HIL Design



9. Save the **Adapted_AD9856_HIL.mdl** model.
10. Connect the DSP development board and ensure that you switch it on.
11. Re-open the **Hardware in the loop** dialog box and set the reset level as **Active_High**.



The reset level must match the level specified in the Signals block for the original model.

12. Click on **Next** to display the second page of the **Hardware in the loop** dialog box (Figure 9-11). Enter a full device name into the **FPGA device** field. Verify that the device name matches the device on the DSP development board and is compatible with the device family set in the original model.



If you need to change the device family to use a different board, you must repeat steps 2, 3, 6, and 11.

Figure 9-11. Hardware in the Loop Parameter Settings Page 2

Hardware in the loop - page 1 of 2

6. Compile the HIL design

FPGA device: EP2S60F1020C4

Compile with Quartus II

7. Program the FPGA

JTAG Cable: USB-Blaster [USB-0]

Device in chain: @1: EP2S60 (0x120930DD)

Scan JTAG

Configure FPGA

Design Adapted_AD9856_Hil/HIL

Back Close

13. Click **Compile with Quartus II** to compile the HIL model.
14. Click **Scan JTAG** to find all the hardware connected to your computer and select the required **JTAG Cable** and **Device in chain**.
15. Click **Configure FPGA** to download the compiled programming file (.sof) to the DSP development board.
16. Close the **Hardware in the loop** dialog box and save the model.

17. Simulate the HIL model. Compare the displays of the OutputSpectrum and OutScope blocks to the waveforms in the original model, which should be identical.

Figure 9-12 on page 9-11 shows the output spectrum waveform and Figure 9-13 on page 9-11 shows the output scope waveform.

Figure 9-12. Output Spectrum Waveform

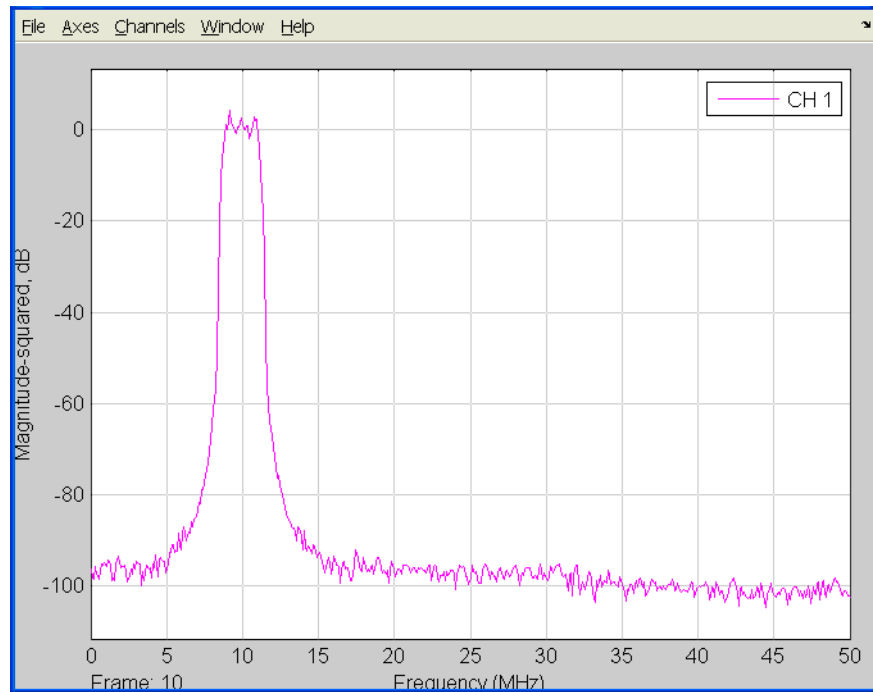
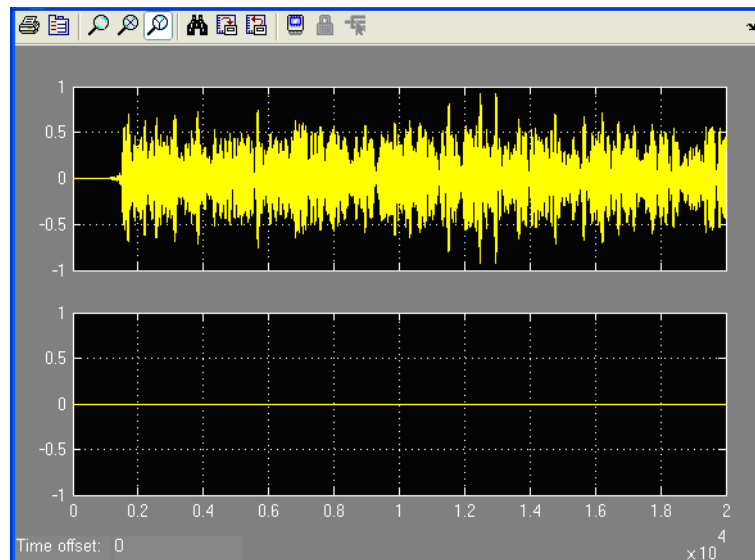


Figure 9-13. Output Scope Waveform





You can speed up the HIL simulation using burst mode. To use burst mode, open the **Hardware in the loop** dialog box and turn on **Burst Mode**. Then repeat step 15 to download an updated programming file into the device on the DSP development board. This action resets the memory and registers to their starting value. When you simulate the HIL design example again the simulation is much faster. The OutputSpectrum display should be identical, but you can observe extra delays (equal to the burst length) on signals in the OutScope display.

Archiving Combined Blockset Designs

To create an archive for a combined design with both the standard and advanced blockset:

1. Simulate the design to generate HDL for the advanced blocks in the design.
2. Open the Signal Compiler interface, select the **Export** tab and click **Export**. This action exports HDL to the specified directory including a copy of all the standard blockset HDL and a Quartus II IP file (.qip), which contains all the assignments and other information to process the standard blockset design example in the Quartus II compiler.
3. This .qip file does not include the HDL files for the advanced blockset. To include these files, add the directories containing the generated HDL for the advanced block as user libraries in the Quartus II software.
4. Select **Add/Remove Files in Project** from the Project menu, select the Libraries category and add all directories that contain generated HDL from the advanced blocks. Separate directories exist for each level of hierarchy in the Simulink design.
5. You can archive this Quartus II project to produce a portable Quartus II archive file (.qar) that contains all the source HDL for the combined standard and advanced blocksets in the DSP Builder system. Select **Archive Project** on the Project menu in the Quartus II software.

Using HIL in Advanced Blockset Designs

To use HIL in a design that uses blocks from the advanced blockset only, perform the following steps:

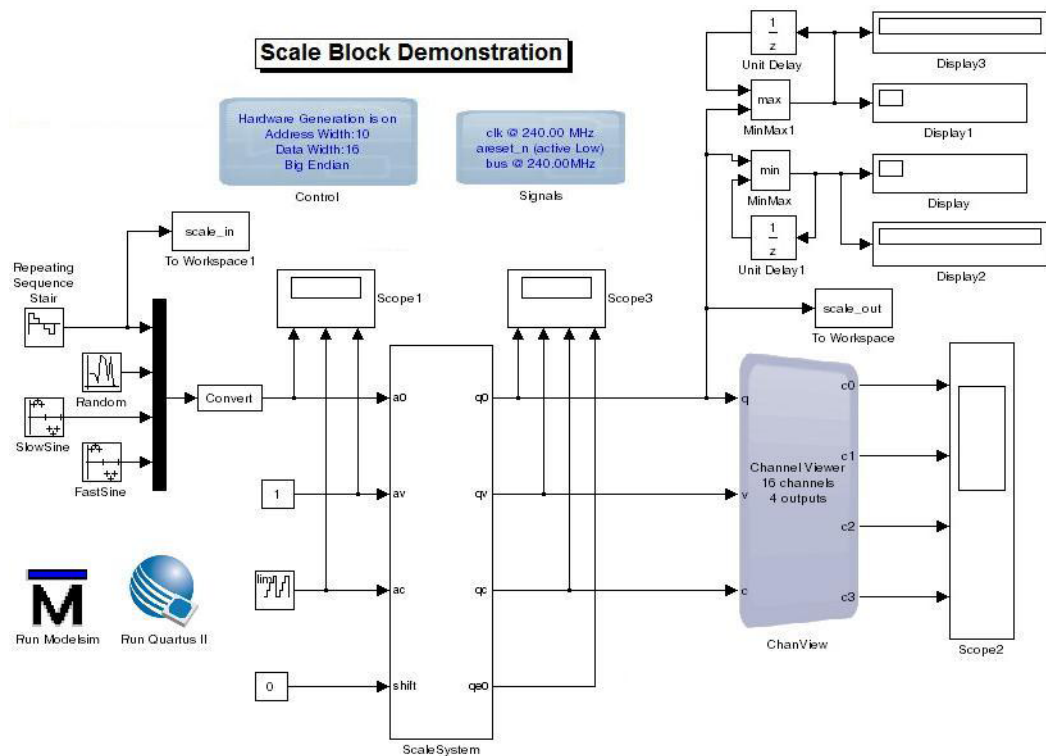
1. Set the following variable in MATLAB to prevent DSP Builder from using virtual pin assignments (virtual pins don't appear on the HIL block).

```
DSPBA_Features.VirtualPins = false
```

Alternatively, add this variable to the `setup_<modelname>.m` file before loading the model.

- Open the **demo_scale.mdl** design example from the **Examples\Baseblocks** directory in the installed design examples for the advanced blockset (Figure 9-14).

Figure 9-14. Scale Block Design Example

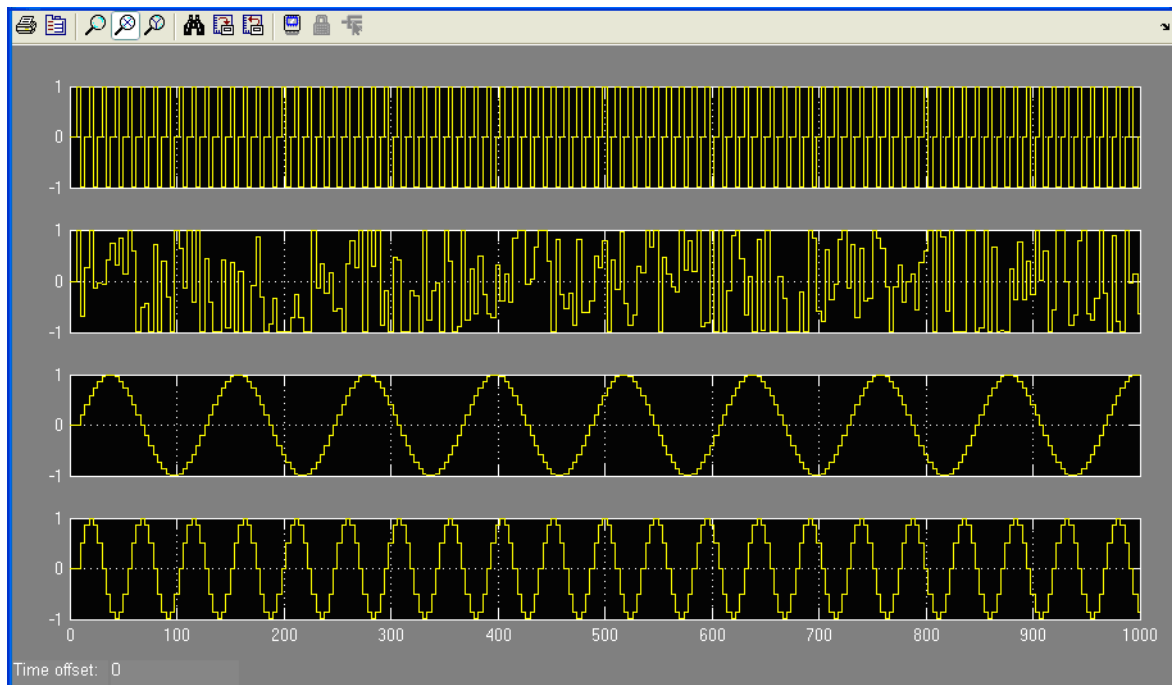


- Simulate the design to generate HDL.


- Ensure that the **Hardware Destination Directory** specified in the Control block uses an absolute, not relative, path.

This design demonstrates the Scale block from the advanced blockset to scale four-channel data from signed fractional [2].[30] to signed fractional [1].[15]. The four inputs are repeating square stairs of [1 0.5 -0.5 -1 0], randomly generated numbers, a slow sine wave, and a fast sine wave (Figure 9-15 on page 9-14).

Figure 9-15. Input Waveforms for the Scaler Example



- Open the Device block inside the ScaleSystem subsystem to verify that the target device family is compatible with the target DSP development board for the HIL simulation.
- Double-click on the Run Quartus II block to start the Quartus II software and create a Quartus II project with the HDL generated in step 4.

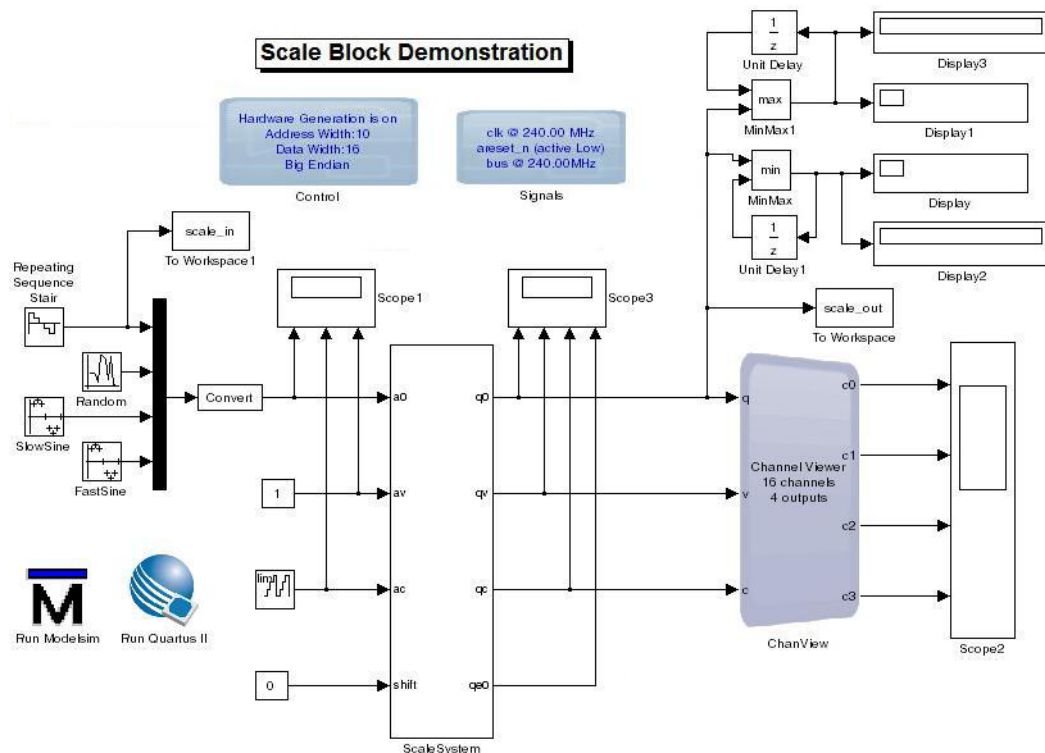
 The Quartus II project obtains its name from the subsystem that contains the Device block, **ScaleSystem.qpf**.

- In the Quartus II Tools menu, click **Start Compilation** and verify that the project compiles successfully.
- Save a copy of the design example as **demo_scale_HIL.mdl** and delete the ScaleSystem subsystem in this model.
- Replace the ScaleSystem subsystem by a HIL block from the **AltLab** library in the DSP Builder standard blockset.

9. Double-click on the HIL block to open the **Hardware in the loop** dialog box (Figure 9–16 on page 9–15). In the first page of the dialog box, select the Quartus II project (**ScaleSystem.qpf**) that you created in step 6. Select the clock pin (clk) and reset pin (areset_n). Set the a0, a1, a2, and a3 input port types as **signed [2].[30]**, the q0, q1, q2, and q3 output port types as **signed [1].[15]**.

Leave the `ac` and `shift` input ports and the `qc` output port as unsigned. The reset level must match the level specified in the `Signals` block for the original model.

Figure 9–16. Hardware in the Loop Parameter Settings Page 1 for the Scale Block Example



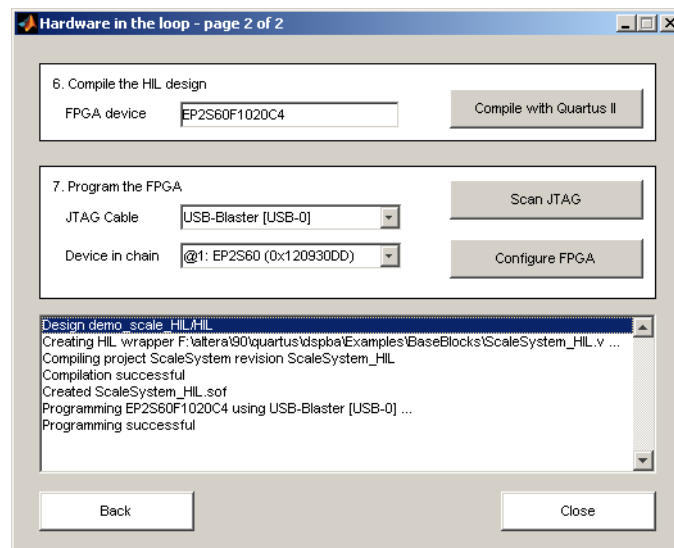
10. Close the **Hardware in the loop** dialog box.
11. Remove the Scope1, Scope3, Unit Delay, Unit Delay1, MinMax, MinMax1, To Workspace, Display, Display1, Display2, and Display3 blocks from the model.
12. Connect all the inputs and outputs to the **HIL** block using Input and Output blocks from the standard blockset. Verify that the data types for all the inputs and outputs are set correctly, matching those set in the **Hardware in the loop** dialog box by step 10.

The HIL block expands the channel data input bus into four individual inputs (a0, a1, a2, and a3) and the channel data output bus into four separate outputs (q0, q1, q2, and q3).

13. Use Simulink Demux and Mux blocks to separate the inputs into individual inputs and multiplex the four outputs together into one data bus.

17. Click on **Next** to display the second page of the **Hardware in the loop** dialog box (Figure 9-18). Enter a full device name into the **FPGA device** field. Verify that the device name matches the device on the DSP development board and is compatible with the device family set in the original model.

Figure 9-18. Hardware in the Loop Parameter Settings Page 2



18. Click **Compile with Quartus II** to compile the HIL model.
19. Click **Scan JTAG** to find all the hardware connected to your computer and select the required **JTAG Cable** and **Device in chain**.
20. Click **Configure FPGA** to download the compiled programming file (.sof) to the DSP development board.
21. Close the **Hardware in the loop** dialog box and save the model.
22. Simulate the HIL model. Compare the waveform of the DataOutScope block to the results for the original model, which should be identical.

This chapter describes the following topics for experienced DSP Builder users:

- Creating a New Model From an Existing Design Example
- Managing your Designs
- Pruning and Twiddle for FFT Blocks
- Hardware Verification
- Using Scripts
- Example Custom Scripts
- Folding
- How to Manage Latency
- Flow Control
- Interfacing with a Processor Bus

Creating a New Model From an Existing Design Example

To create a new design from an existing design example and change the namespaces, perform the following steps, which use the FIR design example as an example:

1. Open the FIR design example (**demo_fir**) from the Filters directory, by typing the following command at the MATLAB command prompt:
`demo_fir`
2. In the `demo_fir` window (the schematic), double-click on the `EditParams` block to open the setup script **setup_demo_fir.m** in the MATLAB Editor.
3. In the Editor on the File menu click **Save As** and save as **setup_mytutorial.m** in a different directory, for example **\myexamples**.
4. In the `demo_fir` window, on the File menu click **Save As** and save as **mytutorial.mdl** in the **\myexamples** directory.
5. In the main MATLAB window, navigate to the **\myexamples** directory.
6. In the Editor on the Edit menu click **Find And Replace**, enter `dspb_fir` in **Find what:** and `my_tutorial` in **Replace with:**. Click **Replace All**. Click **Close**. This step ensures all the setup variables do not interfere with any other workspace variables.
7. Save **setup_mytutorial.m**.
8. On the Debug menu click **Run setup_mytutorial.m** to run the script, which creates the workspace variables to use the schematic design.

9. To ensure MATLAB runs the setup script on opening (so that the design displays correctly) and just before simulation (so that the parameters are up-to-date and reflect any edits made since opening), perform the following steps:
 - a. In the mytutorial window (schematic), on the File menu click **Model Properties**.
 - b. On the Callbacks tab click on **PreLoadFcn** and replace `setup_demo_firi;` with `setup_mytutorial;`.
 - c. Repeat for the **InitFcn**.
 - d. Click **OK**.
10. In the mytutorial window, double-click on the FilterSystem subsystem, then double-click on InterpolatingFIR block. Replace all instances of `dspb_firi` with `mytutorial`. Click **OK**. These parameters set up the FIR filter.
11. Double-click on InChanView, block replace all instances of `dspb_firi` with `mytutorial`, click **OK**.
12. Repeat for the OutChanView and Signals blocks and the following blocks:
 - The input stimulus generation blocks:
 - Sine Wave
 - Const
 - Impulse
 - Random
 - Channel Counter
 - Valid Sequence
 - The downsample blocks:
 - InDownsample
 - OutDownsample.
 - Spectrum analyzers
 - InSpectrum (on the axis properties tab)
 - OutSpectrum (on the axis properties tab)
13. Change the simulation stop time from `20000*dspb_firi.SampleTime` to `20000*mytutorial.SampleTime`.
14. Change the title and save your new design.

Managing your Designs

DSP Builder provides several features that ease managing your designs, including support for parameterization through scripting.

1. To define many of DSP Builder advanced blockset parameters as a MATLAB workspace variables, such as clock rate, sample rate, and bit width, define these variables in a `.m` file.
2. Run this setup script before running your design.

3. Explore different values for various parameters, without having to modify the Simulink design.

For instance, to evaluate the performance impact of varying bit width at different stages of your design:

1. Define the data type and width of ModelPrim blocks in the script
2. Experiment with different values. DSP Builder advanced blockset vector signal and ALU folding support allows you to use the same design file to target single and multiple channels designs.
3. Use a script for device options in your setup script, which eases design migration, whether you are targeting a new device or you are upgrading the design to support more data channels.
4. Use advanced scripting to fine tune Quartus II settings and to build automatic test sweeping, including parameter changes and device changes.

Managing Basic Parameters

DSP Builder supports extensive parameterization of your design. Before you start implementing your design, you should define the following key parameters in a script:

- FPGA clock rate
- Data sample rates at various stages of your design
- Number of channels or data sources
- Bit widths of signals at various stages of your design, including possible bit growth throughout the computational datapath
- Coefficients of filters

Based on the FPGA clock rate and data sample rates, you can derive how many clock cycles are available to process unique data samples. This parameter is called *Period* in many of the design examples. For example, for a *period* of three a new sample for the same channel appears every three clock cycles. For multiplication, you have three clock cycles to compute one multiplication for this channel. In a design with multiple channels, you can accommodate three different channels with just one multiplier. A resource reuse potential exists when *period* is greater than one.

Creating User Libraries

You can group frequently used custom blocks into libraries for future use. To convert a DSP Builder advanced blockset ModelPrim subsystem into a custom block, perform the following steps:

1. Mask the block to hide the block's contents and provide a custom block dialog.
2. Place the block in a library to prohibit modifications and allow you to easily update copies of the block.



This procedure is similar to creating a Simulink custom block and custom library. You can also add a custom library to the Simulink library browser.

Revision Control

You can use Simulink revision control to manage your DSP Builder advanced blockset design revision control.

The Simulink Model Info block displays revision control information about a model as an annotation block in the model's block diagram. It shows revision control information embedded in the model and information maintained by an external revision control or configuration management system.

You can customize some revision control tools to use the Simulink report generator XML comparison, which allows you to compare two versions of the same file.

You must add the following files to revision control:

- Your setup script (.m file)
- Model design files .mdl.
- All the customized library files.



You do not need to archive autogenerated files such as Quartus II project files or synthesizable RTL files.

Pruning and Twiddle for FFT Blocks

DSP Builder FFT blocks (except for the legacy FFT interfaces) allow you to control the widths and types of internal datapaths using the pruning and twiddle specification.

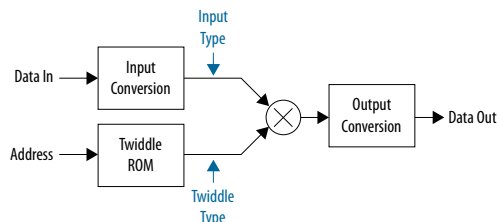
DSP Builder allows you to specify:

- The type of the data values before each twiddle multiplication.
- The type of the twiddle constants.
- The type of the data values after each twiddle multiplication.

For example:

```
dspba.fft.full_wordgrowth(true,false,2,fixdt(1,16,15),fixdt(1,18,17))
```

Figure 10-1. Pruning and Twiddle for FFT Blocks



An FFT with $2N$ points has N radix-2 stages and (conceptually) $N-1$ twiddle multipliers. In practice, DSP Builder optimizes away many of the twiddle multipliers. However, they still need entries in the twiddle specification.

The twiddle and pruning specification for this FFT consists of a $(N-1) \times 3$ array ($N-1$ rows with 3 entries in each row) of strings which specify these types. DSP Builder uses strings because Simulink does not pass raw types into the Simulink GUI.

DSP Builder provides three utility functions to generate twiddle and pruning specifications, each of which implements a different pruning strategy:

- `dspba.fft.full_wordgrowth(complexFFT,radix2,N,input_type,twiddle_type)`
- `dspba.fft.mild_pruning(complexFFT,radix2,N,input_type,twiddle_type)`
- `dspba.fft.prune_to_width(maxWidth,complexFFT,radix2,N,input_type,twiddle_type)`

In addition, DSP Builder provides a fourth function for floating-point FFTs (where no pruning is required)

- `dspba.fft.all_float(N, float_type)`

This function generates a pruning specification where the input, twiddle and output types are all `float_type`.

The legacy FFT interfaces use `dspba.fft.full_wordgrowth()` pruning strategy. It grows the datapath by one bit for each radix-2 FFT stage.

The `dspba.fft.mild_pruning()` grows the datapath by one bit for each two radix-2 FFT stages.

The `dspba.fft.prune_to_width(maxWidth)` grows the datapath by one bit for each radix-2 FFT stage up to the specified maximum width. At that point, it applies drastic pruning to ensure that the data input to the twiddle multiplier is never more than `maxWidth` bits wide.

Altera provides these built-in strategies only for your convenience. If you need a different pruning strategy, you can define and use your own pruning function (or just construct the pruning or twiddle array manually).

Each of these utility functions generate an array in the appropriate format ($N-1$ rows, each containing three entries).

In each case:

- `complexFFT` is a Boolean number (usually true) that indicates whether the FFT's input is complex.
- `radix2` is a Boolean number (usually false) that indicates whether the FFT can have two consecutive twiddle stages.
- `N` is an integer indicating the number of radix-2 stages in the FFT. For example, 10 for a 1,024-point FFT.
- `input_type` is the type of the input signal.
- `twiddle_type` is the type of the twiddle constants.

Hardware Verification

Altera provides the system-in-the-loop flow for hardware verification. In DSP Builder Advanced v14.1 this flow is a beta feature.

System-in-the-loop,

- Automatically generates HW verification system for DSP Builder designs based on your configuration.
- Provides a wizard-based interface to configure, generate, and run HW verification system.
- Provides two separate modes:
 - **Run Test Vectors** loads and runs test vectors with large chunks (based on test memory size on target verification platform)
 - **Data Sample Stepping** loads one set sample at a time while stepping through Simulink simulation

Data Sample Stepping generates a copy of original model and replaces the DSP Builder block with a special block providing connection to the FPGA to process data.

Preparing to Use System-In-The-Loop

Before you use system-in-the-loop:

1. Ensure you have a full installation of the Altera OpenCL SDK.
2. Ensure ALTERAOCLSDKROOT variable points to the installation root directory.
3. For Windows, if you intend to use **Data Sample Stepping**, add the following suffix to PATH environment variable:

```
<YOUR_OPEN_CL_INSTALLATION_ROOT>/host/windows64/bin:<YOUR_DSPBA_INSTALLATION_ROOT>/backend/windows64
```

System-In-The-Loop Supported Blocks

System-in-the-loop only supports DSP Builder device-level blocks.. The block interface may have complex and vector type ports.

All block input and output ports should pass through a single DSP Builder ChannelIn or ChannelOut interface, or be connected to a single ModelIP block. The block may contain memory-mapped registers and memory blocks (accessible through the autogenerated Avalon-MM slave interface). Observe the following limitations:

- The design should use the same clock for system and bus interfaces. Separate clocks are not supported.
- For autogenerated Avalon MM slave interfaces, use the name **bus**.
- Any other combination of DSP Builder block interface, including Avalon-MM master interfaces are not supported.

The overall bitwidth of block input and output ports should not exceed 512 bits (excluding valid signal).

Running HW verification with **Data Sample Stepping** loads a new set of test data to FPGA every simulation step (if the data set is valid), which gives big timing gaps between two consequent cycles for DSP Builder blocks running on hardware. If your DSP Builder block implementation cannot handle such gaps, SIL simulation results may be incorrect.

Building Custom Board Support Packages

This topic is for JTAG-based board support packages.

Setting up Board Support Package for 28 nm Device Families

For 28 nm device families:

1. Copy the <ALTERAOCLSDKROOT>/board/dspb_sil_jtag directory to a location where you have write access (hereinafter CUSTOM_BOARDS).
2. Change <CUSTOM_BOARDS>/dspba_sil_jtag/hardware directory
3. Rename jtag_c5soc directory to desired name (hereinafter jtag_myboard), remove the second directory.
4. Change to the jtag_myboard directory
5. In the top.qsf file, in 'Board specific' section
 - a. Change the device family and name setting according to device in your board
 - b. Change 'clk' and 'resetsn' port location and IO standard assignments according to your board specification
6. In top.sdc file, in the call for 'create_clock' command. Change the clock period according to a clock specification you have set in top.qsf file
7. Open board.qsys file in Qsys
 - a. Update 'REF_CLK_RATE' parameter value for 'kernel_clk_generator' instance according to clock specification you have set in top.qsf/sdc files
 - b. Update the device family setting according to your board specification
8. Open system.qsys file in Qsys. Update device family setting according to your board specification.
9. In the <CUSTOM_BOARDS>/dspba_sil_jtag/board_env.xml file change the default hardware name to your directory (jtag_myboard in this walkthrough).

Setting up Board Support Packages for Other Device Families

For the boards containing other device families, perform these additional steps:

1. In the scripts/post_flow.tcl file remove the following line: source \$::env(ALTERAOCLSDKROOT)/ip/board/bsp/adjust_plls.tcl
2. Open board.qsys file in Qsys
 - a. Remove the kernel_clk_generator instance (the same instance as in step 7.a for 28 nm families).
 - b. Add instance of Altera PLL with one output clock. Set the reference clock frequency according to the same setting used in previous steps.
 - c. Export the refclk Clock Input interface with kernel_pll_refclk name
 - d. Connect outclk0 to the initial source of kernel_clk_generator.kernel_clk output.
 - e. Connect the global_rest_in.out_reset output to reset input of PLL instance.
 - f. Set the generated clock frequency



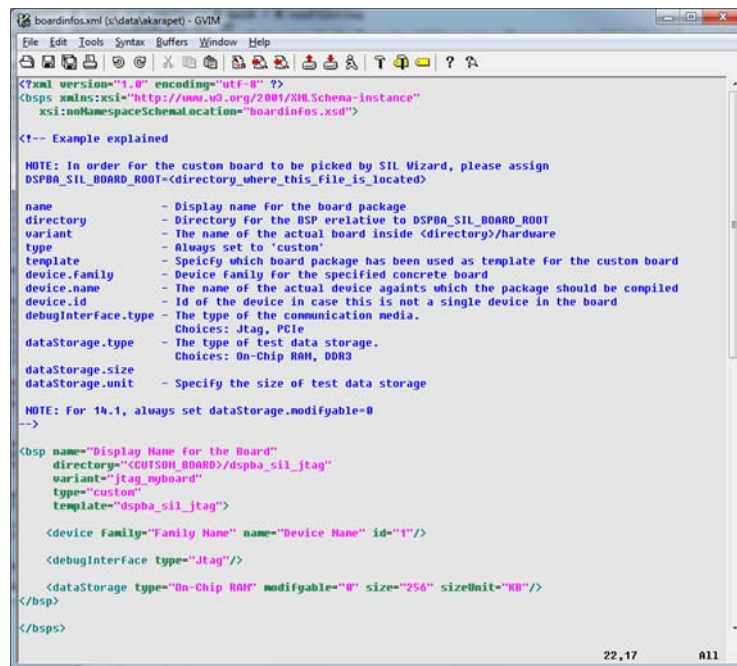
The design must meet timing on this clock domain. Altera advises that you use a low target frequency.

Publishing the Package in the System-In-The-Loop Wizard

To publish the package:

1. Create a file named **boardinfos.xml** file in the directory where you copy the **dspba_sil_jtag** directory.
2. Before you start MATLAB, set the **DSPBA_SIL_BSP_ROOT** variable to point to the directory where this file and your custom board is located.

Figure 10-2. Boardinfo.xml File Content



Third-Party Board Support Packages

System-in-the-loop support the following third-party board support packages that are available for OpenCL:

- Bittware
- Nallatech
- ProcV

These packages are not available in the system-in-the-loop wizard by default. After you install these boards, publish the packages to the system-in-the-loop wizard.

Template Values in the boardinfos.xml File

The following table provides values you should use for the **template** field in the **boardinfos.xml** based on your template.

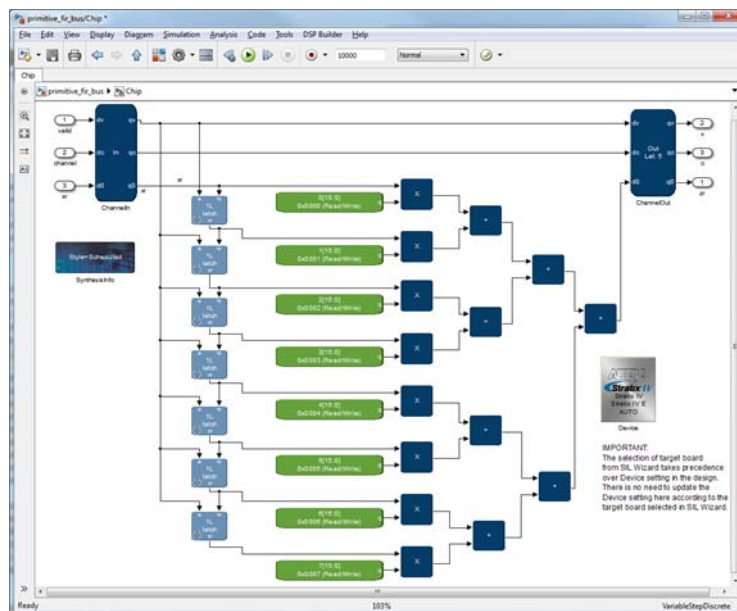
Table 10-1. Template Values in boardinfos.xml File

Board Support Package	Value of 'Template' field
Bittware	bittware_s5phg
Nallatech	nallatech_pcie3x5
Gidel ProcV	gidel_procev
Custom packages based on dspba_sil_jtag	dspba_sil_jtag
Custom packages based on dspba_sil_pcie	dspba_sil_pcie

Running System-In-the-Loop

This walkthrough uses a DSP Builder design that implements a primitive FIR filter with memory-mapped registers for storing coefficients.

Figure 10-3. FIR Filter with Memory-Mapped Registers



To run system-in-the-loop:

1. In the design's Control block ensure you turn on **Generate Hardware**.
2. Simulate the model to generate RTL.
3. Select a device-level sub-system in your design and click **DSP Builder > New SIL Wizard**.
4. On the **Parameters** tab, specify the parameters (Table 10-2).
5. Click the **Run** tab and specify the run parameters (Table 10-3).

System-In-The-Loop Parameters

The design interface settings only generate appropriate adapters between the DSP Builder ChannelIn and ChannelOut interfaces and test Avalon-ST interface. The hardware platform always runs at fixed clock rate.

Table 10-2. System-In-The-Loop Parameters

Section	Option	Description
BSP Settings	BSP	Select the target BSP you want to run the hardware test on
	Device	Device on the selected board.
BSP Memory Allocation	Total Memory Size	Specify the total size for test memory to use. You cannot specify this parameter in v14.1.
	Input Memory Size	Specify the amount of memory (from total memory size) that should be used for storing input test data. The rest are used for storing output data. Several iterations may be required load and process all input test vectors because of memory limitations.
Design Interface	Clock Rate	This value should be the same as specified for DSP Builder design in setup file.
	Sample Rate	The sample described the DSP Builder block sample. This value should be the same as specified in DSP Builder block setup file.
	Number of Channels	The number of channels for the DSP Builder block. This value should be the same as specified in DSP Builder block setup file.
	Frame Size	This value represents a number of valid data samples that you should supply to the DSP Builder block without timing gaps in between. If this value is more than 1, the wizard inserts a specific block in between test data provider and the DSP Builder block. This block enables data transmission to the DSP Builder block only when specified amount of data is already available. An example of such a design is a folded multichannel designs.
-	Destination Directory	Specify the directory where the system In the loop related files should be generated. You should change to this directory to simulate the system-in-the-loop generated model with FPGA proxy.

Table 10-3. System-In-The-Loop Run Settings

Setting	Description
Select SIL Flow	<p>Select the SIL flow to use. The options are:</p> <p>Run Test Vectors runs all test vectors through the hardware verification system. The test vectors are based on simulation data recorded in DSP Builder .stm format files during Simulink simulation.</p> <p>Step Through Simulation allows processing every different set of valid input data on hardware separately, while simulating a design from Simulink. The wizard generates a separate model <model_name>_SIL in the SIL destination directory, which you should use for hardware verification. The original DSP Builder Device level block is replaced with a specific block providing communication with the FPGA.</p> <p>You should change to SIL destination directory before you can simulate this model. If you change the flow, regenerate and recompile the system into a new destination directory.</p>
Generate	Generates the infrastructure, files, and blocks for the hardware verification platform.
Compile	<p>Compiles the entire hardware verification system in the Quartus II software to the generation configuration file.</p> <p>Allow at least 10-15 minutes for this step to run (more time for large DSP Builder designs). During this time the MATLAB input interface is unavailable.</p>
Select JTAG Cable	<p>Press Scan to scan available JTAG connections for programming the board.</p> <p>Chose required JTAG cable from the discovered list.</p>
Program	<p>Program the board through selected JTAG cable.</p> <p>Go directly to this step if you have a pregenerated design with no changes with the flow parameters and in DSP Builder design under test.</p>
Run	<p>Run the test on hardware. Run Test Vectors only.</p> <p>The hardware test automatically detects and executes write requests over the DSP Builder autogenerated Avalon-MM slave interface. As the wizard cannot keep the sequence of transfers for write requests over Avalon-MM slave interface and the DSP Builder data interface on hardware exactly the same as during simulation, you may see data mismatches for a few sets of output samples at points where write requests are issued.</p>
Compare	<p>Compare the hardware verification results with simulation outputs. Run Test Vectors only.</p> <p>The wizard compares only valid output samples.</p>
Simulate <original_model>_SIL system	<p>During simulation, the FPGA proxy block that replaces the original DSP Builder design in the system-in-the-loop:</p> <ul style="list-style-type: none"> ■ Every time you update inputs, it loads data to DSP Builder if valid input is high ■ Every time you request outputs, it populates outputs with data read from hardware if output memory contains valid sample. <p>Step through simulation only.</p> <p>Because FPGA proxy updates its output only with valid samples, you see the same results repeated on the outputs until hardware has a new valid set of data. This behavior may differ from simulation results, where outputs are populated at every simulation cycles with available values.</p>

Using Scripts

This section describes using custom scripts for generating and analyzing your DSP Builder advanced blockset design.

1. Analyze the parts of a design you might wish to keep flexible, and place these parts and derived parameters in a configuration script so that you can change these parameters.
2. Typically use a MATLAB script for the configuration script. The script base addresses for registers and memory are from your design.
3. DSP Builder advanced blockset supports scripting and parameterization of your design. Defining most of the parameters such as clock frequency, data sample rate, number of channels and bit width at various stages of your design in a script, to create a design that can be easily reparameterized without even opening the Simulink model.
4. Create a setup script, which is usually a MATLAB **.m** file with clock rate, bit width, and other important information that you must evaluate before you update your design in Simulink or start a simulation.

Evaluating Setup Scripts

To evaluate the setup script, use the callbacks functionality of Simulink. To configure callbacks, perform the following steps:

1. In a Simulink model file **.mdl**, on the File menu click **Model properties**.
2. Select **Callbacks** tab.
3. Select **PreLoadFcn** and type the setup script name in the window on the right hand side. When you open your Simulink design file, the setup script runs.
4. Select **InitFcn** and type the setup script name in the window on the right hand side. Simulink runs your setup script first at the start of each simulation before it evaluates the model design file **.mdl**.

Callbacks make your design more robust, and ensures all parameters are evaluated before hardware is generated.

Visualization Features

When designing with DSP Builder advanced blockset, use the following visualization features of MATLAB and Simulink:

- **OutScope** block. In addition to exporting data to work space for analysis, you can use the OutScope block to visualize a signal or multiple signals. The OutScope block probes and displays data on a wire or a bus relative to the time samples, which is useful when debugging your design.
- **OutputSpectrum** block. You can also use the OutputSpectrum block, which displays the signal spectrum in real time, when your design has filtering or FFT.
- **Fixed-point toolbox**. When dealing with bit growth and quantization, the fixed-point toolbox can be a valuable tool. You can even visualize the dynamic range of a signal by looking at the histogram of the signal.

Example Custom Scripts

You can write scripts that directly change parameters (such as the hardware destination directory) on the Control and Signals blocks.

For example, in a script that passes the design name (without **.mdl** extension) as *model* you can use:

```
%% Load the model
load_system(model);

%% Get the Signals block
signals = find_system(model, 'type', 'block', 'MaskType', 'DSP
Builder Advanced Blockset Signals Block');

if (isempty(signals))
    error('The design must contain a Signals Block. ');
end;

%% Get the Controls block
control = find_system(model, 'type', 'block', 'MaskType', 'DSP
Builder Advanced Blockset Control Block');

if (isempty(control))
    error('The design must contain a Control Block. ');
end;

%% Example: set the RTL destination directory
dest_dir = ['./rtl' num2str(freq)];
set_param(control{1}, 'destination', dest_dir);
```

Similarly you can get and set other parameters. For example, on the Signals block you can set the target clock frequency:

```
fmax_freq = 300.0;
set_param(signals{1}, 'freq', fmax_freq);
```

You can also change the following threshold values that are parameters on the Control block:

- distRamThresholdBits
- hardMultiplierThresholdLuts
- mlabThresholdBits
- ramThresholdBits

You can loop over changing these values, change the destination directory, run the Quartus II software each time, and perform design space exploration. For example:

```
%% Run a simulation; which also does the RTL generation.
t = sim(model);

%% Then run the Quartus II compilation flow.
[success, details] = run_hw_compilation(<model>, './')

%% where details is a struct containing resource and timing
information
    details.Logic,
```

```
details.Comb_Aluts,  
details.Mem_Aluts,  
details.Reggs,  
details.ALM,  
details.DSP_18bit,  
details.Mem_Bits,  
details.M9K,  
details.M144K,  
details.IO,  
details.FMax,  
details.Slack,  
details.Required,  
details.FMax_unres,  
details.timingpath,  
details.dir,  
details.command,  
details.pwd
```

such that `>> disp(details)` gives output something like:

```
Logic: 4915  
Comb_Aluts: 3213  
Mem_Aluts: 377  
Reggs: 4725  
ALM: 2952  
DSP_18bit: 68  
Mem_Bits: 719278  
M9K: 97  
M144K: 0  
IO: 116  
FMax: 220.1700  
Slack: 0.4581  
Required: 200  
FMax_unres: 220.1700  
timingpath: [1x4146 char]  
dir: '../quartus_demo_iff_t_4096_for_SPR_FFT_4K_n_2'  
command: [1x266 char]  
pwd: 'D:\test\script'
```



The Timing Report is in the *timingpath* variable, which you can display by `disp(details.timingpath)`. Unused resources may appear as -1, rather than 0.

You must previously execute `load_system` before commands such as `find_system` and `run_hw_compilation` work.

A useful set of commands to generate RTL, compile in the Quartus II software and return the details is:

```
load_system(<model>);  
sim(<model>);  
[success, details] = run_hw_compilation(<model>, './')
```

Folding

Folding optimizes hardware usage for low throughput systems, which have many clock cycles between data samples. Low throughput systems often inefficiently use hardware resources. When you map designx that processes data as it arrives every clock cycle to hardware, many hardware resources may be idle for the clock cycles between data.

Folding allows you to create your design and generate hardware that reuses resources to create an efficient implementation.

The folding factor is the number of times you reuse a single hardware resource, such as a multiplier, and it depends on the ratio of the data and clock rates:

Folding factor = clock rate/data rate

DSP Builder offers ALU folding for folding factors greater than 500. With ALU folding, DSP Builder arranges one of each resource in a central arithmetic logic unit (ALU) with a program to schedule the data through the shared operation.

ALU Folding

ALU folding generates an ALU architecture specific to the DSP Builder design. The functional units in the generated ALU architecture depend on the blocks and data types in your design. DSP Builder maps the operations performed by connecting blocks in Simulink to the functional units on the generated architecture.

ALU folding reduces the resource consumption of a design by as much as it can while still meeting the latency constraint. The constraint specifies the maximum number of clock cycles a system with folding takes to process a packet. If ALU folding cannot meet this latency constraint, or if ALU folding cannot meet a latency constraint internal to the DSP Builder system due to a feedback loop, you see an error message stating it is not possible to schedule the design.

ALU Folding Limitations

Avoid using ALU folding with designs that use many data types. ALU folding is ideal for large designs with a uniform data type, such as single-precision floating-point. The design uses less logic when creating a single hardware resource for an operation in the ALU that it can share across the design.

For designs that use more than one data type, a Convert block between two data types uses more resources if the design requires saturation and rounding. An unbiased rounding operation uses more resources than a biased rounding mode.

Some DSP Builder blocks store state, for example:

- Sample Delay
- Counter
- DualMem
- FIFO

With ALU folding, any blocks that store state have a separate state for each channel. DSP Builder only updates the state for a channel when the system processes the channel. Thus, a sample delay delays a signal until processing the next data sample. For 200 clock cycles to a data period, DSP Builder delays the signal for the 200 clock cycles. Also, data associated with one channel cannot affect the state associated with any other channel. Changing the number of channels does not affect the behavior of the design.



For designs without ALU folding, state is associated with a block, which you can update in any clock cycle. Data input with channel 0 can affect state that then affects a computation with data input with channel 1.

ALU Folding Parameters

Table 10-4. ALU Folding Parameters

Parameter	Description
Sample Rate	Data sample rate.
Number of Channels	Supports single or multiple channels
Maximum latency	Maximum latency for the system.
Register outputs	The format of data outputs
Simulation rate	Specify clock rate or data rate to control how Simulink models the system

Simulation Rate

In the ALU folding parameters, you can specify **Data rate** or **Clock rate** for **Simulation rate**. the Simulation rate only controls the simulink simulation; the hardware is identical.

Date rate simulation offers the following features:

- Simulates faster.
- Simulates original unfolded model.
- Each Simulink sample represents a data sample.
- Generates automatic ModelSim testbench (if you turn on in the Control block).

Clock rate simulation offers:

- Simulink sample rates identical to the clock rate.
- Simulation matches the hardware interface.
- Modelling of clock level timings and jitter in the data inputs.

Clock Rate

Figure 10-6. Single Channel Clock Rate Simulation with no Register Outputs

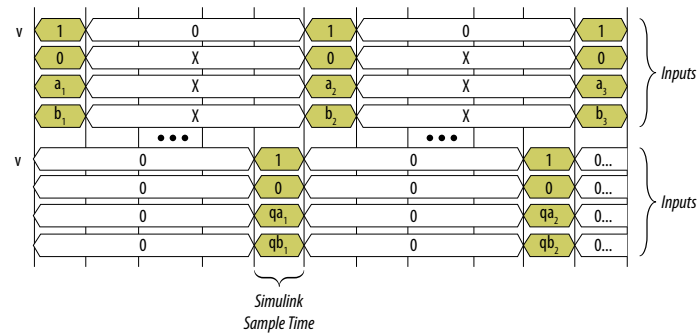


Figure 10-7. Single Channel Clock Rate Simulation with Register Outputs

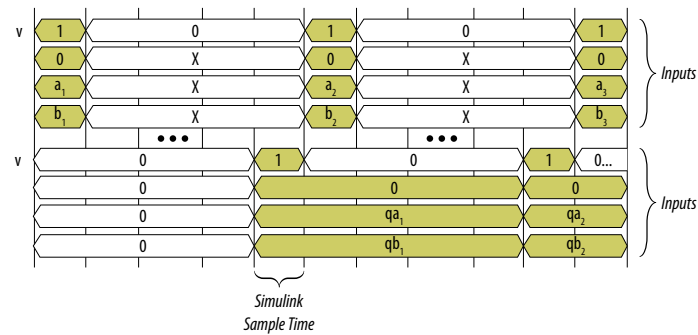


Figure 10-8. Multichannel Clock Rate Simulation with no Register Outputs

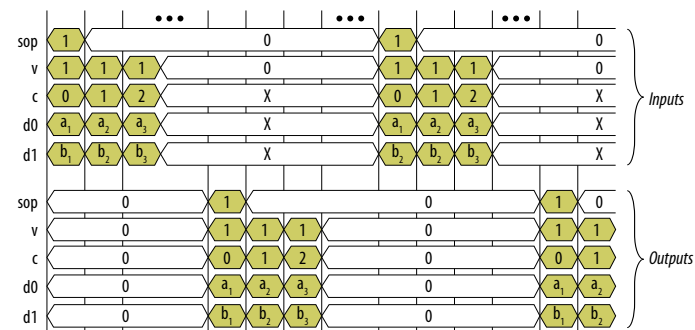
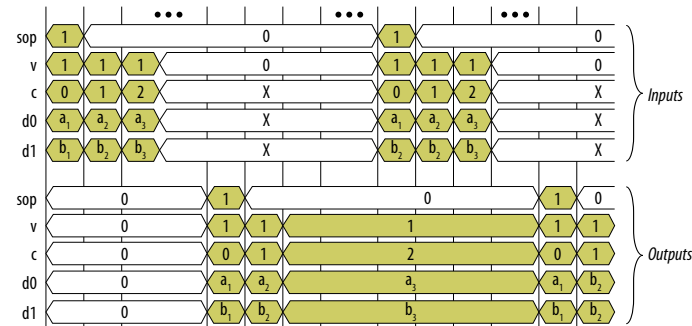



Figure 10-9. Multichannel Clock Rate Simulation with Register Outputs



Using ALU Folding

To use ALU folding, perform the following steps:

1. Open the top-level design that contains the primitive subsystem you want to add ALU folding to.
2. Save a backup of your original design.
3. Replace:
 - Constant multiplier blocks with multipliers blocks.
 - Reciprocal blocks with Divide blocks
 - Sin(?x) blocks with sin(x) blocks.
4. Avoid low-level bit manipulation
5. Open the primitive subsystem (which contains the ChannelIn and ChannelOut blocks) and add a ALU Folding block from the DSP Builder **Utilities** library.

 In the ChannelIn and ChannelOut blocks, before you use ALU folding, ensure you turn off **Folding enabled**.
6. Double click the ALU Folding block to open the **Block Parameters** window.
7. Enter a value for **Sample rate(MHz)**.
8. Enter a value for **Maximum latency(cycles)**.
9. Turn off **Register Outputs** to make the output format the same as the input format. Turn on **Register Outputs**, so that the outputs hold their values until the next data sample output occurs.
10. Select the **Simulation rate**.
11. Simulate your design.

DSP Builder generates HDL for the folded implementation of the subsystem and a testbench. The testbench verifies the sample rate Simulink simulation against a clock rate ModelSim simulation of the generated HDL.

Using Automated Verification

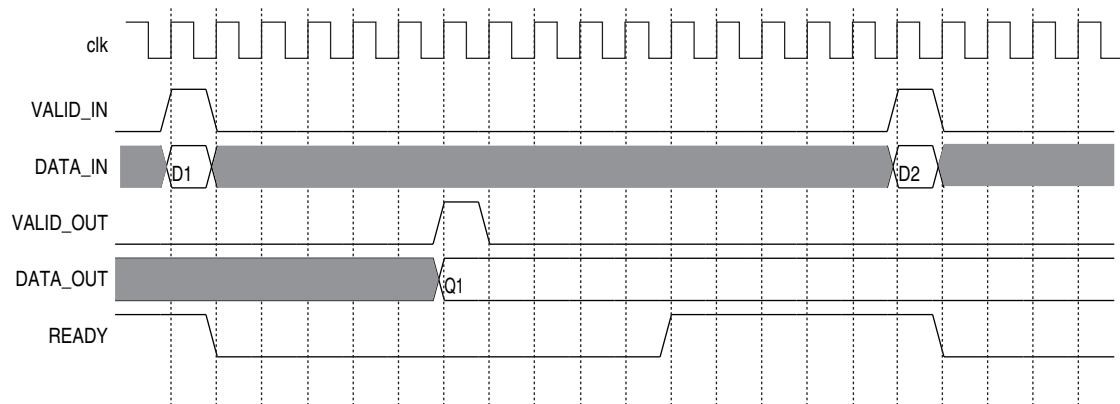
To use automated verification, on the DSP Builder menu click **Verify Design**.

The testbench uses captured test vectors from the Simulink simulation and plays through the clock rate simulation of the generated hardware at the data rate. DSP Builder checks the order and bit-accuracy of the hardware simulation outputs against the Simulink simulation.

Ready Signal

The ready signal is an output that goes high to indicate when you can input data into your design. It provides flow control that allows you to reduce jitter in your design. The ready signal output is high when the internal architecture is idle.

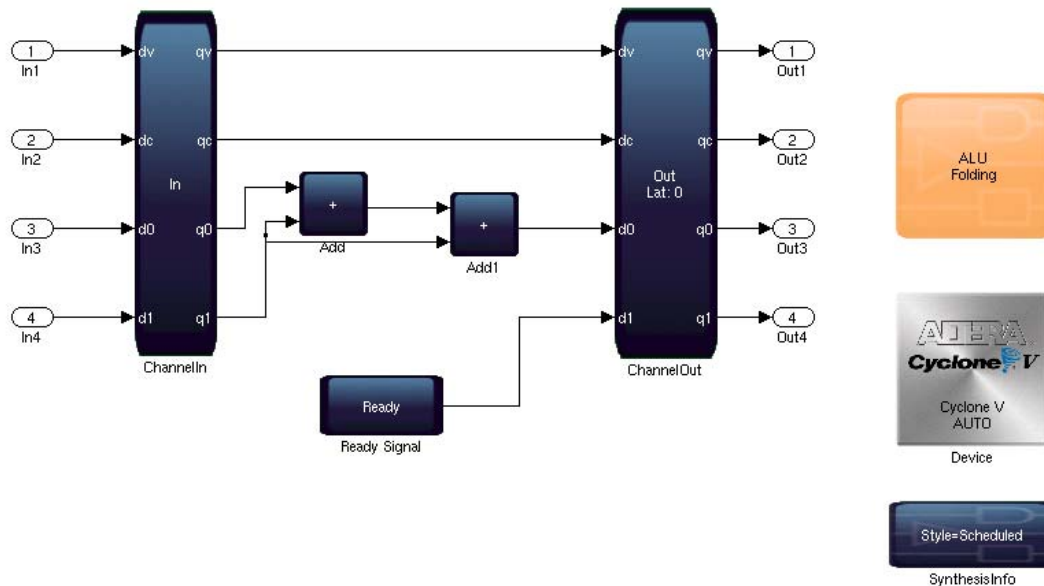
Figure 10-10. Ready Signal Timing



Connecting the Ready Signal

1. Connect a Ready block from the ModelPrim library to the ChannelOut block.

Figure 10-11. Connecting Ready Block



About the Start of Packet Signal

DSP Builder uses a start of packet signal for systems using ALU folding. The start of packet signal is an extra signal on the ChannelIn and ChannelOut blocks. To sue the the start of packet signal turn on **Has Start of Packet Signal** on the ChannelIn and ChannelOut blocks. You must use the start of packet signal for multichannel designs.

With the Start of Packet signal:

- The system is in an idle state after reset and after it finishes processing a data sample.
- The system indicates the first clock cycle of a packet of data when the start of packet signal goes high.
- The system processes the data packet if it is in an idle state when it receives the start of packet signal.
- The system is not idle the clock cycle after the start of packet signal until it finishes processing a data sample.

You may use the valid signal instead of the start of packet signal, which does not allow the folded system to process a non-valid data sample.

Removing Resource Sharing Folding

DSP Builder v14.0 you could use resource sharing folding, which is now removed in v14.1. When you open pre v14.1 designs in v14.1, you must remove resource sharing folding, which you originally selected on the ChannelIn block:

1. Open the design in v14.1.
2. Replace Channel In and ChannelOut blocks with new ChannelIn and ChannelOut blocks.
3. Change any part of your design that uses TDM vectors.
4. Change any aspects of your design that uses a sample delays rather than a clock delays.



If you do not remove resource sharing folding, and you simulate your design you see a MATLAB system error.

How to Manage Latency

The ModelPrim blocks are untimed circuits, so are not cycle accurate. A one-to-one mapping does not exist between the blocks in the Simulink model and the blocks you implement in your design in RTL. This decoupling of design intent from design implementation gives productivity benefits. The ChannelOut block is the boundary between the untimed block and the cycle accurate block. This block models the additional delay that the RTL introduces, so that data going in to the ChannelOut block delays internally, before DSP Builder presents it externally. The latency of the block displays on the ChannelOut mask.

You may want to fix or constrain the latency after you complete part of a design, for example on a ModelIP block or for a ModelPrim subsystem. In other cases, you may want to limit the latency in advance, which allows future changes to other subsystems without causing undesirable effects upon the overall design.

To accommodate extra latency, insert registers. This feature applies only to ModelPrim subsystems. To access, use the ModelPrim Synthesis Info block.

Latency is the number of delays in the valid signal across the subsystem. The DSP Builder advanced blockset balances delays in the valid and channel path with delays that DSP Builder inserts for autoupelining in the datapath.



User-inserted sample delays in the datapath are part of the algorithm, rather than pipelining, and are not balanced. However, any uniform delays that you insert across the entire datapath optimize out. If you want to constrain the latency across the entire datapath, you can specify this latency constraint in the SynthesisInfo block.

Reading Latency

To read the added latency value for a ModelIP block (or for the ChannelOut ModelIP block), select the block and type the following command:

```
get_param(gcb, 'latency')
```

You can also use this command in an M-script. For example when you want to use the returned latency value to balance delays with external circuitry.



If you use an M-script to get this parameter and set latency elsewhere in your design, by the time it updates and sets on the ModelIP block, it is too late to initialize the delays elsewhere. You must run your design twice after any changes to make sure that you have the correct latency. If you are scripting the whole flow, you must run once with end time 0, and then run again immediately with the desired simulation end time.

Using Latency Constraints

To set a latency constraint for a ModelPrim subsystem, use the `SynthesisInfo` block. If you select **Scheduled** synthesis style in the **Block Parameters** dialog box, you can optionally turn on **Constrain Latency** and constrain the latency to $>$, $>=$, $=$, $<=$, or $<$ a specified limit.



You can specify the limit using a workspace variable or expression but it must evaluate to a positive integer.

You can use the `SynthesisInfo` block to specify a latency constraint for any subsystem containing a ModelIP block or ModelPrim subsystem. However, any constraints set by f_{MAX} targets have priority over latency constraints. An error issues if you are setting a latency that is smaller than that required to reach the target f_{MAX} .

For example, if you set a latency constraint of 10, the hardware uses four pipelining stages across the multiplier to reach the target f_{MAX} , and adds an extra six pipelining stages to meet the latency constraint. DSP Builder balances this action on the channel and valid paths so that all signals remain synchronized. The simulation shows the result, and is visible on the ChannelOut block which displays **Lat: 10**.



Do not use latency constraints in subsystems where folding is enabled.

The latency of a subsystem automatically shows on the ChannelOut block after you run a simulation in Simulink.

If you want, you can display the latency that a ModelIP block introduces.

You can get the latency value programmatically by using the following command:

```
eval(get_param(<full path to block>, 'latency'))
```

For example:

```
eval(get_param('design/latencydemo/ChannelOut', 'latency'))
```

DSP Builder calculates the latency in the initialization of the model, at the start of simulation.

DSP Builder calculates all latencies at the same stage, so you cannot use the latency of one block to set the constraint on another and expect to see the changes that propagate in the same simulation. For example, suppose you want to always have a latency of 40 for the CIC block in the **demo_dcic** design. Add a ModelPrim subsystem after the CIC block (Figure 10-12). This subsystem consists only of ChannelIn, ChannelOut, and SynthesisInfo blocks. (Figure 10-13).

Figure 10-12. Decimating CIC Design Example Illustrating Latency Propagation

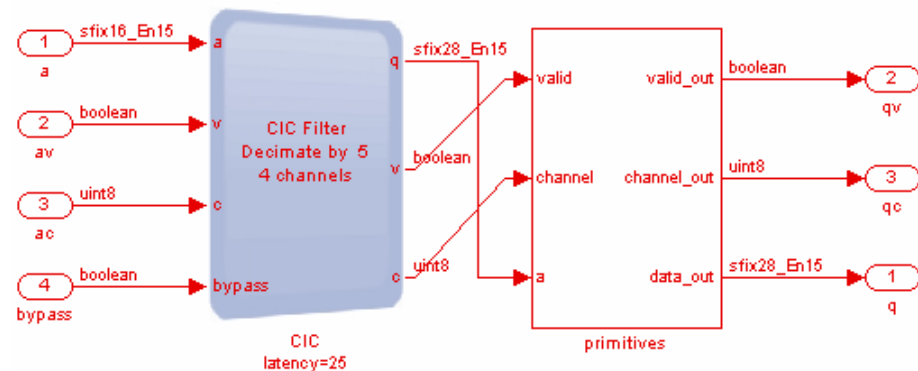
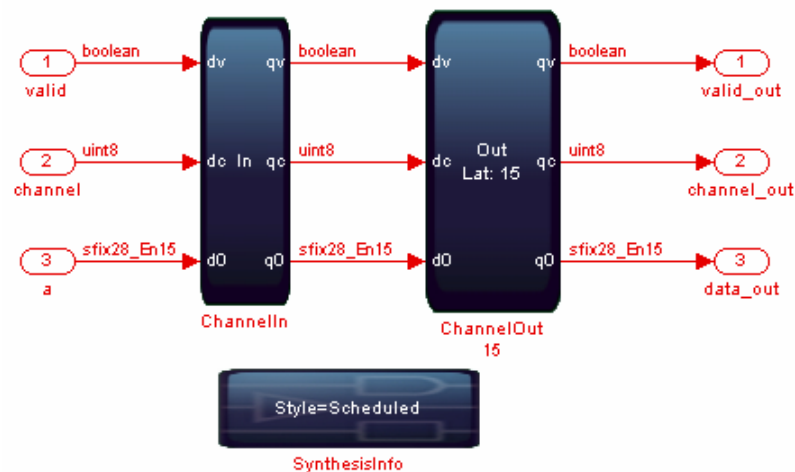


Figure 10-13. ModelPrim Subsystem for the CIC Design Example



On the SynthesisInfo block, set the following latency constraint that DSP Builder derives from the CIC block:

```
40 - eval(get_param('demo_dcic/CICSystem/CIC', 'latency'))
```

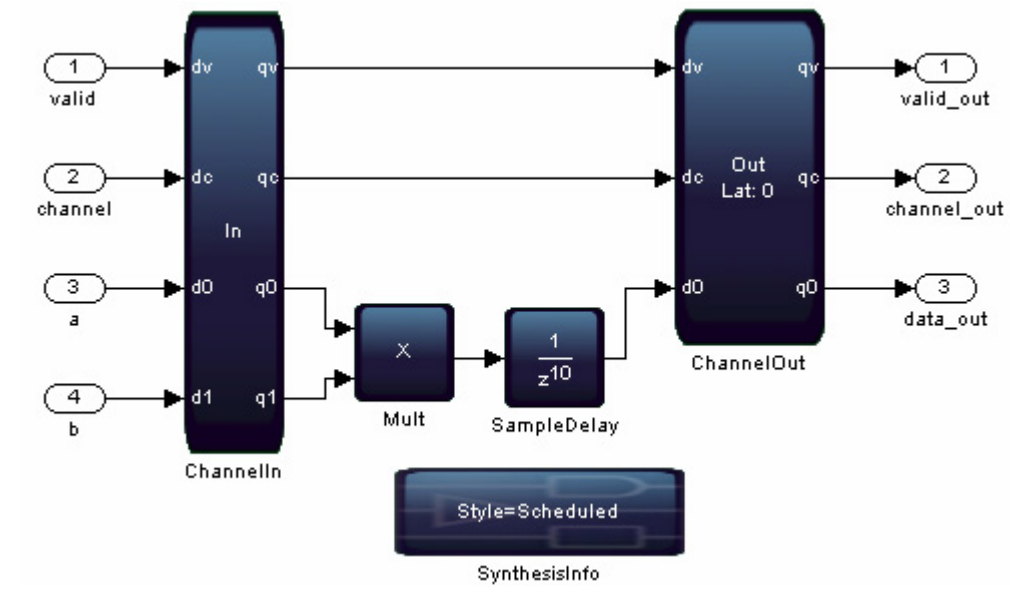
Any changes to the CIC block that change the latency do not display as immediate latency changes in the ModelPrim subsystem; but only on the subsequent simulation and hardware generation cycle. The latency for the CIC block is only available after the initialization step in which it is set in the ModelPrim subsystem.

Thus, you must simulate until after cycle 0, stop and re-start simulation (without making any changes) to guarantee that DSP Builder applies the correct latency in the ModelPrim subsystem. The SynthesisInfo block uses the evaluated CIC block latency from the previous simulation.

Zero Latency Example

In the example (Figure 10-14) sufficient delays in the design ensure that DSP Builder requires no extra automatic pipelining to reach the f_{MAX} target (although DSP Builder distributes this user-added delay through the datapath). Thus, the reported latency is zero. DSP Builder inserts no extra pipelining registers in the datapath to meet f_{MAX} and thus inserts no balancing registers on the channel and valid paths. The delay of the valid signal across the subsystem is zero clock cycles, as the **Lat: 0** latency value on the ChannelOut block shows.

Figure 10-14. Latency Example with a User-Specified Delay

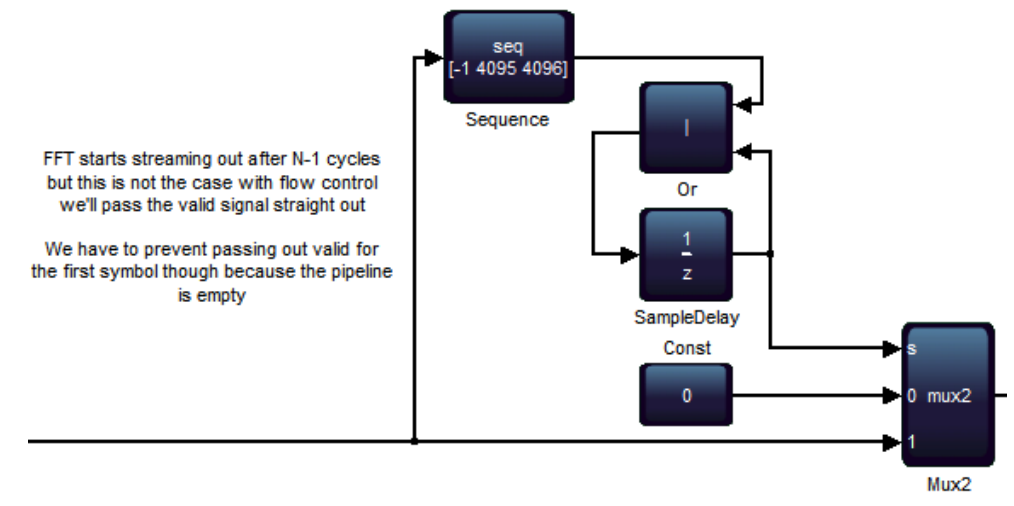


Nonexplicit Delays

The design does not analyze delays to the valid path in ModelPrim subsystems caused by anything other than explicit delays (for example Counter or Sequence blocks) and the design does not count in the valid latency.

For example, the 4K FFT design example uses a Sequence block to drive the valid signal explicitly (Figure 10-15).

Figure 10-15. Sequence Block in the 4K FFT Design Example



The latency that the ChannelOut block reports is therefore not 4096 + the automatic pipelining value, but just the pipelining value.

Distributed Delays

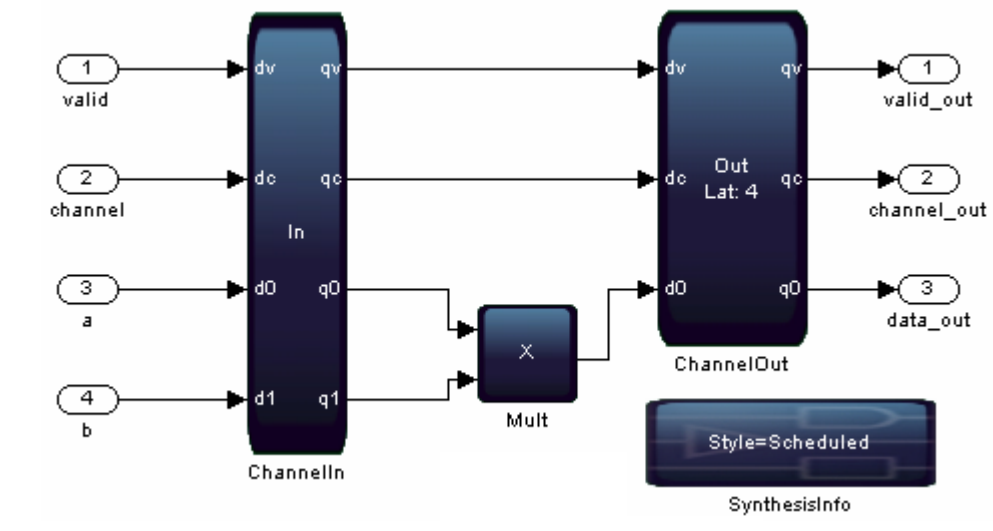
The design example is not cycle-accurate inside a ModelPrim subsystem, because DSP Builder distributes and optimizes the user-specified delay.

For example, in Figure 10-14 on page 10-25 the Mult block has a direct feed-through simulation model, and the following SampleDelay block has a delayed simulation design example with a delay of 10. Thus, there is zero delay on the Mult block in simulation, followed by a delay of 10. In the generated hardware, DSP Builder distributes part of this 10-stage pipelining throughout the multiplier optimally, such that the Mult block has a delay (in this case, four pipelining stages) and the SampleDelay block a delay (in this case, six pipelining stages). The overall result is the same—10 pipelining stages, but if you try to match signals in the ModelPrim subsystem against hardware, you may find DSP Builder shifts them by several cycles.

Similarly, if you have insufficient user-inserted delay to meet the required f_{MAX} , DSP Builder automatically pipelines and balances the delays, and then corrects the cycle-accuracy of the ModelPrim subsystem as a whole, by delaying the output signals in simulation by the appropriate number of cycles at the ChannelOut block.

If there is no user-specified pipelining, the simulation design example for the multiplier is direct-feed-through, and the result appears on the output immediately (Figure 10-16).

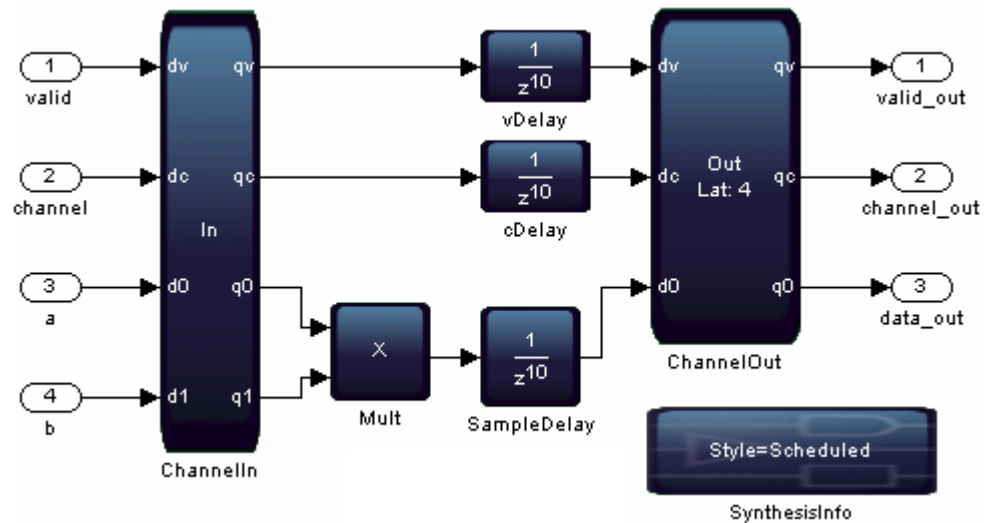
Figure 10-16. Latency Example without a User-Specified Delay



To reach the desired f_{MAX} , DSP Builder then inserts four pipelining stages in the multiplier, and balances these with four registers on the channel and valid paths. To correct the simulation design example to match hardware, the ChannelOut block delays the outputs by four cycles in simulation and displays **Lat: 4** on the block. Thus, if you compare the output of the multiplier simulation with the hardware it is now four cycles early in simulation; but if you compare the ModelPrim subsystem outputs with hardware they match, because the ChannelOut block provides the simulation correction for the automatically inserted pipelining.

If you want a consistent 10 cycles of delay across the valid, channel and datapath, you may need latency constraints (Figure 10-17).

Figure 10-17. latency Example with Consistent Delays



In this example, a consistent line of SampleDelays inserts across the design. However, the algorithm does not use these delays. DSP Builder recognizes that designs do not require them and optimizes them away, leaving only the delay that designs require. In this case, each block requires a delay of four, to balance the four delay stages to pipeline the multiplier sufficiently to reach the target f_{MAX} . The delay of 10 in simulation remains from the non-direct-feed-through SampleDelay blocks. In such cases, you receive the following warning on the MATLAB command line:

Warning:

Some user inserted SampleDelays have been optimized away. The latency on the valid path across primitive subsystem '<design name>' in hardware is 4, which may differ from the simulation model. If you need to preserve extra SampleDelays in this case, use the 'Constraint Latency' option on the SynthesisInfo block.

In summary, if you want to consistently apply extra latency to a primitive subsystem, use latency constraints.

Latency and f_{MAX} Constraint Conflicts

Some blocks need to have a minimum latency, either because of logical or silicon limitations. In these cases, it may be possible to create an abstracted design that cannot be realized in hardware. While these cases can generally be addressed, in some cases like IIRs, find algorithmic alternatives.

Generally, these problems occur in feedback loops. These issues can be addressed by lowering the f_{MAX} target, or by restructuring the feedback loop to reduce the combinatorial logic or increasing the delay. Some control structures that have feedback loops can be redesigned to make them completely feed forward.

You cannot set a latency constraint that conflicts with the constraint that the f_{MAX} target implies. For example, a latency constraint of < 2 for the example (Figure 10–16) conflicts with the f_{MAX} implied pipelining constraint. The multiplier needs four pipelining stages to reach the target f_{MAX} . The simulation fails and issues an error, highlighting the ModelPrim subsystem.

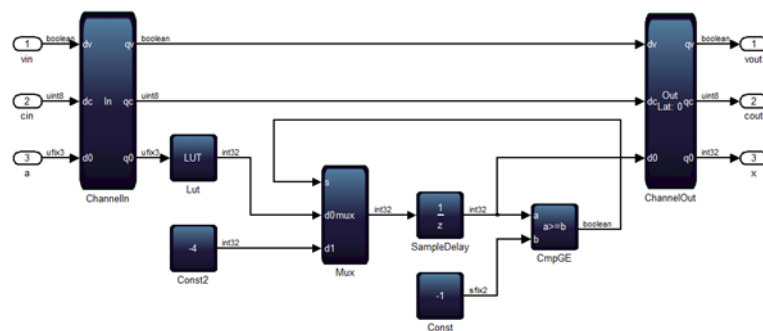
This error issues because you must increase the constraint limit by at least 3 (that is, to < 5) to meet the target f_{MAX} .

Control Units Delays

Commonly, you may use a FSM to design control units. A FSM uses `SampleDelay` blocks to store its internal state. DSP Builder automatically redistributes these `SampleDelay` blocks, which may alter the functional behavior of the control unit subsystem. Then the generated hardware no longer matches the simulation. Also, redistribution of `SampleDelay` blocks throughout the design may change the behavior of the FSM by altering its initial state. Classically, you exploit the reset states of the constituent components to determine the initial state; however this approach may not work. DSP Builder may not record any given component because it automatically pipelines `ModelPrim` subsystems. Also it can leave some components combinatorial based on f_{MAX} target, device family, speed grade, and the locations of registers immediately upstream or downstream.

Figure 10-18 shows a SampleDelay block 32 bits wide. DSP Builder relocates it to save registers to the Boolean signal that drives the s-input of the 2-to-1 Mux block. You may see a mismatch in the first cycle and beyond, depending on the contents of the LUT.

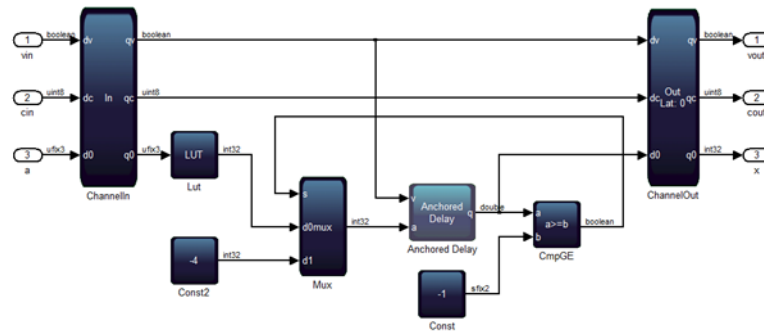
Figure 10–18. SampleDelay Block Example



When you design a control unit as a FSM, the locations of `SampleDelay` blocks specify where DSP Builder expects zero values during the first cycle. In [Figure 10–18](#), DSP Builder expects the first sample that the a-input receives of the `CmpGE` block to be zero. Therefore, the first output value of that compare block is high. Delay redistribution changes this initialization. You cannot rely on the reset state of that block, especially if you embed the `ModelPrim` subsystem within a larger design. Other subsystems may

drive the feedback loop whose pipeline depth adapts to f_{MAX} . The first valid sample may only enter this subsystem after some arbitrary number of cycles that you cannot predetermine. To avoid this problem, always ensure you anchor the `SampleDelay` blocks to the valid signal so that the control unit enters a well-defined state when valid-in first goes high.

Figure 10-19. SampleDelay Block Example 2



To make a control unit design resistant to automated delay redistribution and solve most hardware FSM designs failing to match simulation, replace every `SampleDelay` block with the `Anchored Delay` block from the **Control** folder in the Additional libraries. When the valid-in first goes high, the `Anchored Delay` block outputs one (or more) zeros, otherwise it behaves just like an ordinary `SampleDelay` block.

Synthesizing the example design ($f_{MAX} = 250\text{MHz}$) on Arria V (speedgrade 4), shows that DSP Builder is still redistributing the delays contained inside of the `Anchored Delay` block to minimize register utilization. DSP Builder still inserts a register initialized to zero before the `s`-input of the 2-to-1 Mux block. However, the hardware continues to match Simulink simulation because of the anchoring. If you place highly pipelined subsystems upstream so that the control unit doesn't enter its first state several cycles after device initialization, the FSM still provides correct outputs. Synchronization is maintained because DSP Builder inserts balancing delays on the valid-in wire that drives the `Anchored Delay` and forces the control unit to enter its initial state the correct number of cycles later.

Control units that use this design methodology are also robust to optimizations that alter the latency of components. For example, when a LUT block grows sufficiently large, DSP Builder synthesizes a `DualMem` block in its place that has a latency of at least one cycle. Automated delay balancing inserts a sufficient number of one bit wide delays on the valid signal control path inside every `Anchored Delay`. Hence, even if the `CmpGE` block is registered, its reset state has no influence on the initial state of the control unit when the valid-in first goes high.

Each `Anchored Delay` introduces a 2-to-1 Mux block in the control path. When targeting a high f_{MAX} (or slow device) tight feedback loops may fail to schedule or meet timing. Using `Anchored Delay` blocks in place of `SampleDelay` blocks may also use more registers and can also contribute to routing congestion.

Flow Control

Use DSP Builder `valid` and `channel` signals with data to indicate when data is valid for synchronizing. You should use these signals to process valid data and ignore invalid data cycles in a streaming style to use the FPGA efficiently. You can build designs that run as fast as the data allows and are not sensitive to latency or devices f_{MAX} and which can be responsive to back pressure.

This style uses FIFO buffers for capturing and flow control of valid outputs, loops, and for loops, for simple and complex nested counter structures. Also latches to enable only components with state—thus minimizing enable line fan-out, which can otherwise be a bottleneck to performance.

Flow Control Using Latches

Generally hardware designers avoid latches. However, these subsystems synthesize to flip-flops.

Often designs need to stall or enable signals. Routing an enable signal to all the blocks in the design can lead to high fan-out nets, which become the critical timing path in the design. To avoid this situation, enable only blocks with state, while marking output data as invalid when necessary.

DSP Builder provides the following utility functions in the Additional Blocks Control library, which are masked subsystems.

- “Zero-Latency Latch (`latch_0L`)” on page 12-6
- “Single-Cycle Latency Latch (`latch_1L`)” on page 12-6
- “Reset-Priority Latch (`SRlatch_PS`)” on page 12-6
- “Set-Priority Latch (`SRlatch`)” on page 12-7



Some of these blocks use the Simulink `Data Type Prop Duplicate` block, which takes the data type of a reference signal `ref` and back propagates it to another signal `prop`. Use this feature to match data types without forcing an explicit type that you can use in other areas of your design.

Forward Flow Control Using Latches

The `demo_forward_pressure` example design (“Primitive FIR with Forward Pressure” on page 6-32) shows how to use latches to implement forward flow control.

Flow Control Using FIFO Buffers

You can use FIFO buffers to build flexible, self-timed designs insensitive to latency. They are an essential component in building parameterizable designs with feedback, such as those that implement back pressure.

For more information about the DSP Builder `FIFO` block, refer to “FIFO” on page 17-31.

Flow Control and Backpressure Using FIFO Buffers

The `demo_back_pressure` example design (“Primitive FIR with Back Pressure” on page 6-31) shows how to use latches to implement back pressure flow control.

You must acknowledge reading of invalid output data. Consider a FIFO buffer with the following parameters:

- Depth = 8
- Fill threshold = 2
- Fill period = 7

A three cycle latency exists between the first write and valid going high. The `q` output has a similar latency in response to writes. The latency in response to read acknowledgements is only one cycle for all output ports. The valid out goes low in response to the first read, even though the design writes two items to the FIFO buffer. The second write is not older than three cycles when the read occurs.

With the fill threshold set to a low value, the `t` output can go high even though the `v` out is still zero. Also, the `q` output stays at the last value read when valid goes low in response to a read.

Problems can occur when you use no feedback on the read line, or if you take the feedback from the `t` output instead with fill threshold set to a very low value (< 3). A situation may arise where a read acknowledgement is received shortly following a write but before the valid output goes high. In this situation, the internal state of the FIFO buffer does not recover for many cycles. Instead of attempting to reproduce this behavior, Simulink issues a warning when a read acknowledgement is received while valid output is zero. This intermediate state between the first write to an empty FIFO buffer and the valid going high, highlights that the input to output latency across the FIFO buffer is different in this case. This situation is the only time when the FIFO buffer behaves with a latency greater than one cycle. With other primitive blocks, which have consistent constant latency across each input to output path, you never have to consider these intermediate states.

You can mitigate this issue by taking care when using the FIFO buffer. The model needs to ensure that the read is never high when valid is low using the simple feedback. If you derive the read input from the `t` output, ensure that you use a sufficiently high threshold.

You can set fill threshold to a low number (< 3) and arrive at a state where output `t` is high and output `v` is low, because of differences in latency across different pairs of ports—from `w` to `v` is three cycles, from `r` to `t` is one cycle, from `w` to `t` is one cycle. If this situation arises, do not send a read acknowledgement signal to the FIFO buffer. Ensure that when the `v` output is low, the `r` input is also low. A warning appears in the MATLAB command window if you ever violate this rule. If you derive the read acknowledgement signal with a feedback from the `t` output, ensure that the fill threshold is set to a sufficiently high number (3 or above). Similarly for the `f` output and the full period.

If you supply vector data to the `d` input, you see vector data on the `q` output. DSP Builder does not support vector signals on the `w` or `r` inputs, as the behavior is unspecified. The `v`, `t`, and `f` outputs are always scalar.

Flow Control using Simple Loop

Designs may require counters, or nested counters to implement indexing of multidimensional data. The Loop block provides a simple nested counter—equivalent to a simple software loop.

For more information about the DSP Builder Loop block, refer to “Loop” on page 17-41

The enable input and **demo_kronecker** design example (“Kronecker Tensor Product” on page 6-30) demonstrate flow control using a loop.

Flow Control Using the ForLoop Block

For more information on the DSP Builder ForLoop block, refer to “ForLoop” on page 17-34.

You can use either Loop or ForLoop blocks for building nested loops.

The Loop block has the following advantages:

- A single Loop block can implement an entire stack of nested loops.
- No wasted cycles when the loop is active but the count is not valid.
- The implementation cost is lower because no overhead for the token-passing scheme exists.

The ForLoop block has the following advantages:

- Loops may count either up or down.
- You may specify the initial value and the step, not just the limit value.
- The token-passing scheme allows the construction of control structures that are more sophisticated than just nesting rectangular loops.

When a stack of nested loops is the appropriate control structure (for example, matrix multiplication) use a single Loop block. When a more complex control structure is required, use multiple ForLoop blocks.

Interfacing with a Processor Bus

DSP Builder advanced blockset can interface with a processor bus. You can drag and drop any register in your design without manually creating address decoding logic and memory-mapped switch fabric generation.

Base Addresses

You can add or drop ModelIP or ModelPrim control register fields and memory into your design freely. The only extra work is to keep track of the base address of the modules that hook on Avalon memory-mapped (Avalon-MM) interface.

The base address is a relative address and is expressed as an integer. You can start from address 0 in your model design, or any other arbitrary integer. You only need to keep track of the relative base addresses of modules within your design. When you integrate your model into an Qsys system, Qsys generates a base address for the entire DSP Builder model. Individual modules within **.mdl** design are referenced by Qsys based on the model base address (autogenerated) and relative base address you assigned in the **.mdl** file or its setup script.

To manage base address, consider the following points:

- Bus data width you specify in the Control block
- For ModelIP designs:
 - Number of registers each IP core needs
 - Number of words each register requires
- For ModelPrim subsystems, treat registers independently:
 - Number of words each register requires

The base address depends on data width of the bus and the width of the parameterizable variables (FIR coefficient). For example: a register for a 32-bit FIR coefficient requires two words on a 16-bit bus.

Each ModelIP block and register or memory in a ModelPrim subsystem should have unique base address.

Integrating with Qsys

After you run simulation of your design with **Hardware Generation** turned on in Control block, DSP Builder generates a `<model>_hw.tcl` file for the subsystem containing the Device block. This file marks the boundary of the synthesizable part of your design and ignores the testbench blocks.

1. To add the synthesizable model to Qsys, include `<model>_hw.tcl` at the IP search path.

Qsys native streaming data interface is the Avalon Streaming (Avalon-ST) interface, which DSP Builder advanced blockset does not support. The DSP Builder advanced blockset native interface `<valid, channel, data>` ports are exported to the top-level as conduit signals.

System Components with Avalon-ST Interface Blocks

You may wish to use the DSP Builder design as a subcomponent in a larger system. The DSP Builder component may be one part in a chain of subcomponents that all have Avalon Streaming (Avalon-ST) interfaces, which must interact. You should design the system in Qsys, where you can connect the Avalon-ST interfaces. Hardware Tcl (`_hw.tcl`) files describe the interfaces.

Building system components that Qsys can join together is greatly simplified by defining Avalon-ST interfaces. DSP Builder Advanced builds Avalon Memory-Mapped (Avalon-MM) interfaces, and Avalon-ST input and output blocks allow you to generate the appropriate `hw.tcl` files to define the Avalon-ST interface for the data plane.

This section refers to upstream and downstream components that are part of the system outside of the DSP Builder design.

A design may have multiple Avalon-ST input and output blocks. However, generally all paths across the DSP part must be registered to avoid algebraic loops.

The output of the DSP Builder design is a source of Avalon-ST data for downstream components. It supplies data (and corresponding valid, channel, and start and end of packet information) and accepts a Boolean flag input from the downstream components, which indicates the downstream block is ready to accept data.

The input of the DSP Builder design is a sink of Avalon-ST data for upstream components. It accepts data (and corresponding valid, channel, and start and end of packet information) and provides a Boolean flag output to the upstream component, which indicates the DSPBA component is ready to accept data.

When the system generates the **hw.tcl** file, the name of the Avalon-ST masked subsystem block is the name of the interface.

The Avalon-ST blocks are masked subsystems. You can look under the mask to see the implementation. You can break the library link and extend the definition further, by adding further ports that the **hw.tcl** file declares, or add text that DSP Builder writes unevaluated directly into the interface declaration in the **hw.tcl** file.

These blocks do not enforce Avalon-ST behavior. They encapsulate the common Avalon-ST signals into an interface, add FIFO buffers on the output (and if required on the input) to ease building designs supporting back-pressure, and declare the collected signals as an Avalon-ST interface in the **hw.tcl** file generated for the device level.

DSP Builder offers the following Avalon-ST interface blocks:

- “Avalon-ST Output (AStOutput)” on page 12-2
- “Avalon-ST Input (AStInput)” on page 12-3
- “Avalon-ST Input FIFO Buffer (AStInputFIFO)” on page 12-3

Extending the Interface Definition

DSP Builder provides these Avalon-ST interfaces as masked subsystems.

1. Look under the mask to see the implementation.
2. To edit the mask, you must break the library link. Do not edit the mask type, as DSP Builder uses it to identify the subsystems defining the interfaces.

Adding More Ports to the Avalon-ST Blocks

You can add more ports in the Avalon-ST masked subsystems.

1. Connect these ports internally in the same way as the existing signals. For example, with FIFO buffers.
2. If you add inputs or output ports that you connect to the device level ports, tag these ports with the role the port takes in the Avalon-ST interface. For example, you may want to add error and empty ports.

Adding Custom Text

Any text you write to the description field of the masked subsystem writes with no evaluation into the **hw.tcl** file immediately after the standard parameters for the interface and before the port declarations.

1. You must correctly add the text of any additions.

Restrictions

You can place the Avalon-ST interface blocks in different levels of hierarchy. However, you should never place Simulink, ModelIP or ModelPrim blocks between the interface and the device level ports.

The Avalon-ST interface specification only allows a single data port per interface. Thus you may not add further data ports, or even using a vector through the interface and device-level port (which creates multiple data ports).

To handle multiple data ports through a single Avalon-ST interface, pack them together into a single (not vector or bus) signal, then unpack on the other side of the interface. The maximum width for a data signal is 256 bits.

Use the BitCombine and BitExtract blocks to pack and unpack.

Updating Registers with the Nios II Processor

You can use a processor such as a Nios II processor to read or modify a control register, or to update memory contents. To perform a register update, perform the following steps:

1. Identify the Qsys base address assigned to your DSP Builder design. You can also find the base address information in **system.h** file in your Nios II project, after you have loaded the SOPC library information into your Nios II IDE.
2. Identify the base address for the ModelIP block of interest in your DSP Builder advanced blockset design. It is the base address assigned to your DSP Builder advanced blockset model for step 1, plus the address offset you specified in your ModelIP block or in your setup script. You can also identify the address offset by right clicking on the ModelIP block and selecting **Help**.
3. Identify the base address for the register of interest in your DSP Builder advanced blockset design. It is the base address assigned to your DSP Builder advanced blockset model which you identify in step 1, plus the address offset you specified in your register or in your setup script.
 - a. Identify the address offset in the *<design name>_mmap.h* file, which DSP Builder generates with each design.
 - b. Alternatively, identify the address offset by right clicking on the register and select **Help**.
4. When you identify the base address, use IOWR and IORD commands to write and read registers and memory. For example:

```
IOWR(base_addr_SOPC + base_addr_FIR, coef_x_offset, data)
IORD(base_addr_SOPC + base_addr_FIR, coef_x_offset)
```


Pruning and Twiddle Specification for FFTs

DSP Builder FFT blocks (except for the legacy FFT interfaces) allow you to control the widths and types of internal datapaths using the pruning and twiddle specification.

You specify:

1. The type of the data values before each twiddle multiplication.
2. The type of the twiddle constants.
3. The type of the data values after each twiddle multiplication.

An FFT with 2^N points has N radix-2 stages and (conceptually) $N-1$ twiddle multipliers. In practice, DSP Builder optimizes away many of the twiddle multipliers. However, they still need entries in the twiddle specification.

The twiddle and pruning specification for this FFT consists of a $(N-1) \times 3$ array ($N-1$ rows with 3 entries in each row) of strings which specify these types.

DSP Builder provides three utility functions to generate twiddle and pruning specifications, each of which implements a different pruning strategy:

- `dspba.fft.full_wordgrowth(complexFFT,radix2,N,input_type,twiddle_type)`
- `dspba.fft.mild_pruning(complexFFT,radix2,N,input_type,twiddle_type)`
- `dspba.fft.prune_to_width(maxWidth,complexFFT,radix2,N,input_type,twiddle_type)`

The legacy FFT interfaces use `dspba.fft.full_wordgrowth()` pruning strategy. It grows the datapath by one bit for each radix-2 FFT stage.

The `dspba.fft.mild_pruning()` grows the datapath by one bit for each two radix-2 FFT stages.

The `dspba.fft.prune_to_width(maxWidth)` grows the datapath by one bit for each radix-2 FFT stage up to the specified maximum width. At that point, it applies drastic pruning to ensure that the data input to the twiddle multiplier is never more than `maxWidth` bits wide.

Altera provides these built-in strategies only for your convenience. If you need a different pruning strategy, you can define and use your own pruning function (or just construct the pruning or twiddle array manually).

Each of these utility functions generate an array in the appropriate format ($N-1$ rows, each containing 3 entries).

In each case:

- `complexFFT` is a Boolean number (usually true) that indicates whether the FFT's input is complex.
- `radix2` is a Boolean number (usually false) that indicates whether the FFT can have two consecutive twiddle stages.
- `N` is an integer indicating the number of radix-2 stages in the FFT. For example, 10 for a 1,024-point FFT.
- `input_type` is the type of the input signal.
- `twiddle_type` is the type of the twiddle constants.



Specify `input_type` and `twiddle_type` as actual Simulink types, not as MATLAB strings.

This chapter describes troubleshooting.

Common Problems

You may observe problems with the following areas:

- Timed feedback loops
- Loops, clock cycles, and data cycles

Timed Feedback Loops

Take care with feedback loops generally, in particular provide sufficient delay around the loop.

Designs that have a cycle containing two adders with only a single sample delay are not sufficient. In automatically pipelining designs, DSP Builder creates a schedule of signals through the design. From internal timing models, DSP Builder calculates how fast certain components, such as wide adders, can run and how many pipelining stages they require to run at a specific clock frequency. DSP Builder must account for the required pipelining while not changing the order of the schedule. The single sample delay is not enough to pipeline the path through the two adders at the specific clock frequency. DSP Builder is not free to insert more pipelining, as it changes the algorithm, accumulating every n cycles, rather than every cycle. The scheduler detects this change and gives an appropriate error indicating how much more latency the loop requires for it to run at the specific clock rate. In multiple loops, this error may be hit a few times in a row as DSP Builder balances and resolves each loop.

Loops, Clock Cycles, and Data Cycles

Never confuse clock cycles and data cycles in relation to feedback loops. For example, you may want to accumulate previous data from the same channel. The multichannel IIR filter design example (**demo_iir**) shows feedback accumulators processing multiple channels. In this example, consecutive data samples on any particular channel are 20 clock cycles apart. DSP Builder derives this number from clock rate and sample rate.

The folded IIR filter design example (**demo_iir_fold2**) demonstrates one channel, at a low data rate. This design example implements a single-channel infinite impulse response (IIR) filter with a subsystem built from ModelPrim blocks folded down to a serial implementation.

The design of the IIR is the same as the IIR in the multichannel example, **demo_iir**. As the channel count is one, the lumped delays in the feedback loops are all one. If you run the design at full speed, there is a scheduling problem. With new data arriving every clock cycle, the lumped delay of one cycle is not enough to allow for pipelining around the loops. However, the data arrives at a much slower rate than the clock rate, in this example 32 times slower (the clock rate in the design is 320 MHz, and the sample rate is 10MHz), which gives 32 clock cycles between each sample.

You can set the lumped delays to 32 cycles long—the gap between successive data samples—which is inefficient both in terms of register use and in underused multipliers and adders. Instead, use folding to schedule the data through a minimum set of fully used hardware.

Set the *SampleRate* on both the ChannelIn and ChannelOut blocks to 10 MHz, to inform the synthesis for the ModelPrim subsystem of the schedule of data through the design. Even though the clock rate is 320 MHz, each data sample per channel is arriving only at 10 MHz. The RTL is folded down—in multiplier use—at the expense of extra logic for signal multiplexing and extra latency.

ModelPrim Subsystem Designs to Avoid

You must avoid ModelPrim subsystems with logic that clocked inputs do not drive, because either reset behavior determines hardware behavior or the hardware is inefficient.

Avoid examples that start from reset, as the design simulation in Simulink may not match that of the generated hardware. You should start the counter from the valid signal, rather than the constant. If the counter repeats without stopping after the first valid, add a zero-latency latch into this connection.

Similarly, avoid loops that DSP Builder drives without clocked inputs.



Section II. DSP Builder Advanced Blockset Libraries

This library contains the following libraries:

- Avalon streaming. This library contains the extensible Avalon-ST interface blocks, which are masked subsystems.
- Beta utilities. This library contains the newest blocks, to help with building testbenches.
- Control. This library contains utility masked subsystems that you can use to implement enabled flip-flops.
- Vector utilities. This library contains masked subsystems that help produce vector-parameterizable designs.

The Additional library contains the following blocks in the **Avalon Streaming** folder:

- Avalon-ST Output (AStOutput)
- Avalon-ST Input (AStInput)
- Avalon-ST Input FIFO Buffer (AStInputFIFO)

The Additional library contains the following blocks in the **Beta Utilities** folder:

- Nested Loops (NestedLoop1, NestedLoop2, NestedLoop3)

The Additional library contains the following blocks in the **Control** folder:

- Anchored Delay
- Zero-Latency Latch (latch_0L)
- Single-Cycle Latency Latch (latch_1L)
- Reset-Priority Latch (SRLatch_PS)
- Set-Priority Latch (SRLatch)

The Additional library contains the following blocks in the **Vector Utils** folder:

- Complex to Real-Imag
- Enabled Delay Line
- Enabled Feedback Delay
- Expand Scalar (ExpandScalar)
- Vector Multiplexer (VectorMux)
- Tapped Delay Line (TappedDelayLine)

Avalon-ST Output (AStOutput)

Table 12-1 shows the AStOutput block signals that interface to the external system.

Table 12-1. AStOutput Block External Interface Signals

Name	Direction	Description
source_channel	Output	Channel number.
source_data	Output	The data to be output (which may be, or include control data).
source_eop	Output	Indicates end of packet.
source_ready	Input	Indicates from downstream components that they can accept source_data on this rising clock edge.
source_sop	Output	Indicates start of packet.
source_valid	Output	Indicates that source_data, source_channel, source_sop, and source_eop are valid.

Table 12-2 shows the AStOutput block signals that interface internally with the DSP Builder design component.

Table 12-2. AStOutput Block Internal Interface Signals

Name	Direction	Description
output_channel	input	Channel number.
output_data	input	The output data (which may be, or include control data).
output_eop	input	Indicates end of packet.
output_ready	output	Indicates from the output of the DSP Builder component that it can accept sink_data on this rising clock edge.
output_sop	input	Indicates start of packet.
output_valid	input	Indicates that output_data, output_channel, output_sop, and output_eop are valid.

The downstream system component may not accept data and so may back pressure this block by forcing Avalon ST signal source_ready = 0. However, the DSP Builder design may still have lots of valid outputs in the pipeline. You must store these outputs in memory. DSP Builder writes the output data for the design into a data FIFO buffer, with Avalon-ST signals channel. It writes sop and eop into the respective channel, FIFO buffers.

Connect the back pressure signal (source_ready) from downstream components to port ready in this subsystem. Then DSP Builder reads the FIFO buffers when the downstream block can accept data (read_fifo = 1) and data in FIFO to output (fifo_empty_n = 1) exists.

If the downstream component is continually back pressuring this DSP Builder design, these FIFO buffers start to fill up. If you continue to feed data into the DSP Builder component, eventually the FIFO buffers overflow, which you must not allow to happen. Therefore, when the FIFO buffers reach a certain fill level, they assert signal nearly_full = 1. Use this signal to apply back pressure to upstream component (forcing Avalon ST signal sink_ready = 0). So that upstream components stop sending in more data and so that the FIFO buffer should not overflow, set the fill level at which nearly_full = 1 to a value that depends on the latency of this DSP Builder design. For

example, if the design contains a single ModelPrim subsystem and the ChannelOut block indicates a latency of L, assert the nearly_full flag at the latest point when L free entries are in the FIFO buffer. Setting this threshold is a manual process and full threshold must be greater than or equal to (depth of FIFO buffer – L).

Avalon-ST Input (AStInput)

Table 12–3 shows the AStInput block signals that interface to the external system.

Table 12–3. AStInput Block External Interface Signals

Name	Direction	Description
sink_channel	input	Channel number.
sink_data	input	The data (which may be, or include control data).
sink_eop	input	Indicates end of packet.
sink_ready	output	Indicates to upstream components that the DSPBA component can accept sink_data on this rising clock edge.
sink_sop	input	Indicates start of packet.
sink_valid	input	Indicates that sink_data, sink_channel, sink_sop, and sink_eop are valid.

Table 12–4 shows the AStInput block signals that interface internally with the DSPBA design component.

Table 12–4. AStInput Block Internal Interface Signals

Name	Direction	Description
input_channel	output	Channel number.
input_data	output	The data (which may be, or include control data).
input_eop	output	Indicates end of packet.
input_ready	input	indicates from the output of the DSP Builder component that it can accept sink_data on this rising clock edge.
input_sop	output	Indicates start of packet.
input_valid	output	indicates that input_data, input_channel, input_sop and input_eop are valid.

Avalon-ST Input FIFO Buffer (AStInputFIFO)

The AStInputFIFO block includes FIFO buffers.

Nested Loops (NestedLoop1, NestedLoop2, NestedLoop3)

The NestedLoop1, NestedLoop2, and NestedLoop3 blocks maintain a set of counters that implement the equivalent of a nested for loop in software. They provide more flexibility than the Loop block and greater readability and lower latency than ForLoop blocks.

DSP Builder implements the `NestedLoop` blocks as masked subsystems and use existing DSP Builder ModelPrim blocks. They do not have fixed implementations. DSP Builder generates a new implementation at runtime whenever you change any of the loop specifications.

For each loop in a `NestedLoop` block, you can specify start, increment, and end expressions. Each of these expressions may have one of the following three forms:

- A constant expression that evaluates (in the MATLAB base environment) to an integer. For example, if the MATLAB variable N has the value 256, $(\log_2(N)+1)$ is a legal expression (and evaluates to 9).
- An instance of the loop variable controlling an enclosing loop. For example, you can use "i" (the outer loop variable) as the start expression of the "j" or "k" loops.
- A port name, optionally accompanied by a width specification in angle brackets. For example "p" or "q<4>". If no width is specified, it defaults to 8. This option generates a new input port (with the user-defined name and width) on the `NestedLoop` block.

For a `NestedLoop2` block, with user-supplied start, increment, and end expressions of S1, I1 and E1 (for the outer loop) and S2, I2 and E2 (for the inner loop), the equivalent C++ code is:

```
int i = S1;
do {
    int j = S2;
    do {
        j += I2;
    } while (j != E2);
    i += I1;
} while (i != E1);
```

Each `NestedLoop` block has two fixed input ports (go and en) and a variable number of additional user-defined input ports. DSP Builder regards each user-defined port as a signed input.

Each block also has two fixed output ports (qv and q1) and one (`NestedLoop1`), two (`NestedLoop2`) or three (`NestedLoop3`) output ports for the counter values.

Operation

When the input en signal is low (inactive), the output qv (valid) signal is also set low. The state of the `NestedLoop` block does not change, even if it receives a go signal.

Normal operation occurs when the en signal is high. The `NestedLoop` block can be in the waiting or counting state.

The `NestedLoop` block resets into the waiting state and remains there until it receives a go signal. While in the waiting state, the qv signal is low and the value of the other outputs are undefined.

When the block receives a go signal, the `NestedLoop` block transitions into the counting state. The counters start running and the `qv` output signal is set high. When all the counters eventually reach their final values, the `ql` (last cycle) output becomes high. On the following cycle, the `NestedLoop` block returns to the waiting state until it receives another go signal.

If the block receives a go signal while the `NestedLoop` block is already in the counting state, it remains in the counting state but all its counters are reset to their start values.

Observe the following points:

- All counters in the `NestedLoop` block are signed. To effectively use unsigned counters, zero-extend any unsigned inputs (and correspondingly increase their width specifications) by one bit.
- The end test is an equality test. Each loop continues until the current value of the counter is equal to the end value. However, if the loop counter overflows, the subsequent behavior of the `NestedLoop` block is undefined.
- The end values are inclusive. So setting the start value to 0, the increment to 1 and the end value to 10 actually produces 11 iterations of the loop.
- The previous two factors means that every loop iterates at least once. `NestedLoop` blocks (unlike `ForLoop` blocks) do not support empty loops.
- When you use user-defined ports to supply loop control values, the values on these ports must be held constant while the `NestedLoop` block is in its counting state. Otherwise the block produces undefined behavior.

NestedLoop Block Port Interface

Table 12-5 shows the port interface for the `NestedLoop` blocks.

Table 12-5. NestedLoop Block Port interface

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
go	Input	Boolean	Go.	No	No
en	Input	Boolean	Enable.	No	No
?	Input	Signed integer	Loop control values.	No	No
qv	Output	Boolean	Valid.	No	No
ql	Output	Boolean	Last iteration flag.	No	No
i	Output	Signed integer	Outer loop count.	No	No
j	Output	Signed integer	Middle loop count. (<code>NestedLoop2</code> only).	No	No
k	Output	Signed integer	Inner loop count (<code>NestedLoop3</code> only).	No	No

Anchored Delay

SampleDelay blocks are often not suitable in FSMs, which are common in control unit designs. To ensure that DSP Builder's simulation of FSMs matches the synthesized hardware, use the Anchored Delay block not the SampleDelay block.

The Anchored Delay block has a data input and a valid input port. Connect this to the valid in of the enclosing ModelPrim subsystem to allow DSP Builder to correctly schedule the starting state of your control unit design.



For more information, refer to “Control Units Delays” on page 10-27.

Zero-Latency Latch (latch_0L)

For the latch_0 block the enable signal has an immediate effect on the output. While the enable is high, the data passes straight through. When the enable goes low, the latch_0 block outputs and holds the data input from the previous cycle.

Right-click on the block and select **Look Under Mask**, for the structure.

The e signal is a ufix(1) enable signal. When e is high, the latch_0 block feeds data from input d through to output q. When e is low, the latch_0 block holds the last output.

A switch in e is effective immediately.

Single-Cycle Latency Latch (latch_1L)

For the latch_1 block the enable signal affects the output on the following clock cycle.

These latches work for any data type, and for vector and complex numbers.

Right-click on the block and select **Look Under Mask**, for the structure.

The e signal is a ufix(1) enable signal. When e is high, the latch_1 block delays data from input d by one cycle and feeds through to output q. When e is low, the latch_0 block holds the last output.

A switch in e means the latch_1 block holds the output one cycle later.

Reset-Priority Latch (SRlatch_PS)

DSP Builder offers two single-cycle latency latch subsystems for common operations for the valid signal, latching with set and reset. The SRlatch block gives priority to the reset input signal; the SRlatch_PS block gives priority to the set input signal. In both blocks if set and reset inputs are both zero the current output state is maintained.

Table 12-6 shows the truth table for the SRlatch_PS block.

Table 12-6. Truth Table for SRlatch_PS

S	R	Q
0	0	Q
1	0	1

Table 12–6. Truth Table for SRLatch_PS

S	R	Q
0	1	0
1	1	0

Set-Priority Latch (SRLatch)

DSP Builder offers two single-cycle latency latch subsystems for common operations for the valid signal, latching with set and reset. The SRLatch block gives priority to the reset input signal; the SRLatch_PS block gives priority to the set input signal. In both blocks if set and reset inputs are both zero the current output state is maintained.

Table 12–7 shows the truth table for the SRLatch block

Table 12–7. Truth Table for SRLatch

S	R	q
0	0	q
1	0	1
0	1	0
1	1	1

Complex to Real-Imag

The Complex to Real-Imag handles custom data types that the enhanced precision floating-point types support.

Enabled Delay Line

The Enabled Delay Line block takes a single data signal *a* and an enable signal *e* and implements an enabled delay line, with *q* as the delayed data output.

Enabled Feedback Delay

The Enabled Feedback Delay block takes a single data signal *a* and an enable signal *e* and implements an enabled delay, with *q* as the delayed data output. Internally the implementation is a non-enabled SampleDelay followed by a FIFO buffer. When used in a feedback loop, DSP Builder can distribute the non-enabled SampleDelay around the feedback path, while retaining the right enabled feedback behavior.

Expand Scalar (ExpandScalar)

The ExpandScalar block takes a single connection and replicates it *N* times to form a width *N* vector. The block passes on the width parameter to a Simulink multiplexer under the mask, and uses some standard Simulink commands to add the connections lines.

Tapped Delay Line (TappedDelayLine)

The TappedDelayLine block uses latches from the Control library. This block is an autogenerating subsystem. Right-click on the block, select **View Mask**, and select the **Initialization** tab to see the generation commands.

DSP Builder does not support vector signal input for this block.

Vector Multiplexer (VectorMux)

The VectorMux block dynamically selects a single scalar signal from an input vector of signals. If *in* is a vector of width *N*, *sel* should take the range [0:*N*-1] and the block outputs the (*sel*)th signal from the vector.

This block is an autogenerating masked subsystem that primitive (ModelPrim) blocks build. Internally, it is a demultiplexer and ModelPrim multiplexer, but parameterizable such that you do not have to manually draw and reconnect the connections between the demultiplexer and multiplexer if the vector width parameter changes.

Right-click on the block and select **Look Under Mask** for the structure. Right-click on the block, select **View Mask**, and select the **Initialization** tab to see the generation commands.

This library contains basic blocks that allow you to control your design flow and run external synthesis and simulation tools.

The Base library contains the following blocks:

- Channel Viewer (ChanView)
- Control
- Device
- Edit Params
- LocalThreshold
- Run ModelSim
- Run Quartus II
- Scale
- Signals

Avalon-MM Slave Settings (AvalonMMSlaveSettings)

The AvalonMMSlaveSettings block specifies information about the top-level memory-mapped bus interface widths.



You can either use this block in your design or view the Avalon-MM slave interface settings on the **DSP Builder > Avalon Interface** menu.

Parameters

Table 13–5 shows the parameters for the AvalonMMSlaveSettings block.

Table 13–1. Parameters for the AvalonMMSlaveSettings Block

Parameter	Description
Bus interface name	Specifies the prefix for the address, data and control signals in the generated control bus.
Address width	Specifies the width in bits of the memory-mapped address bus (1–32, default=10).
Data width	Specifies the width in bits of the memory-mapped data bus (8, 16, or 32, default=16).
Bus is:	Specifies whether the memory-mapped address bus is Big Endian or Little Endian .
Separate bus clock	Turn on so any processor-visible control registers are clocked by a separate control bus clock to ease timing closure.
Bus clock frequency (MHz)	Specifies the frequency of the separate processor interface bus clock (when enabled).
Bus clock synchronous with system clock	Turn on so the bus clock is synchronous with the system clock.

Channel Viewer (ChanView)

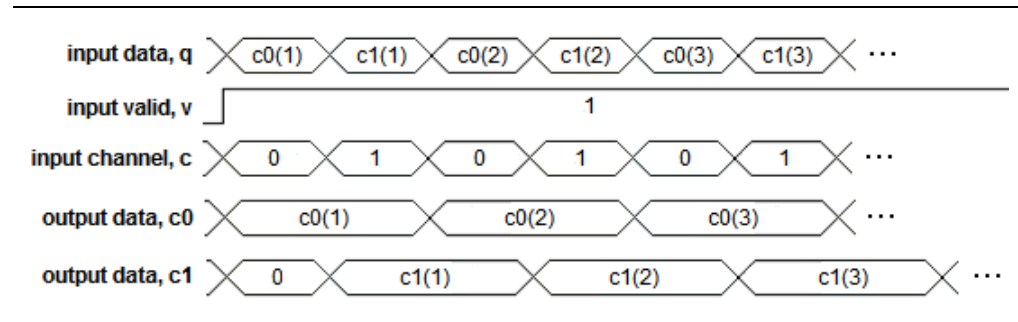
The ChanView block deserializes the bus on its inputs to produce a configurable number of output signals that are not time-division multiplexed (TDM).

You can use a ChanView block in a testbench to visualize the contents of the TDM protocol. It produces synthesizable RTL, so you can use it anywhere in your design.

When a single channel is input, the ChanView block strips out all the non-valid samples, thus cleaning up the display in the Simulink scope.

The channel outputs are not aligned. For example, if you have input channels *c0* and *c1* on a single wire and view both channels, the output is not aligned (Figure 13-1).

Figure 13-1. Channel Viewer Output for Two Channels on a Single Wire



You can add delays after ChanView blocks if you want to realign the output channels.

Parameters

Table 13-2 shows the parameters for the ChanView block.

Table 13-2. Parameters for the ChanView Block

Parameter	Description
Number of input channels	Specifies the number of unique channels the block can process. The design does not use this parameter unless the data bus is a vector or the folding factor is greater than the number of channels. If the data bus is a vector, this value determines which vector element contains the correct channel.
Output channels	A vector that controls the input channels to decode and present as outputs. The number of outputs equals the length of this vector, and each output corresponds to one channel in order.

Port Interface

Table 13-3 shows the port interface for the ChanView block.

Table 13-3. Port Interface for the ChanView Block

Signal	Direction	Description
<i>q</i>	Input	The data input to the block. This signal may be a vector. This block does not support floating-point types.
<i>v</i>	Input	Indicates validity of data input signals. If <i>v</i> is high then the data on the a wire is valid.
<i>c</i>	Input	Indicates channel of data input signals. If <i>v</i> is high, then <i>c</i> indicates which channel the data corresponds to.

Table 13–3. Port Interface for the ChanView Block

Signal	Direction	Description
cn	Output	Each output is a deserialized version of the channel contained in the TDM bus. The output value is updated on each clock cycle that has valid data when the channel matches the required channel.
ov	Output	Optional. Pulses 1 at last cycle of a frame (when all held channel output signals have correct value for the frame) provided valid is high throughout the frame data.

Updated Help

After DSP Builder runs a simulation, it updates the help pages with specific information about each instance of a block. For resource usage, on the DSP Builder menu, point to **Resources**, and click **Design**. Table 13–4 shows some typical help messages that DSP Builder issues for the ChanView block.

Table 13–4. Messages for the ChanView Block

Message Example	Description
Written on Tue Feb 19 11:25:27 2008	Date and time when you ran this file.
Latency is 2	The latency that this block introduces.
Port interface table	Lists the port interfaces to the ChanView block.

Control

The **Control** block specifies information about the hardware generation environment and the top-level memory-mapped bus interface widths.



You must include a **Control** block at the top level of your model.

Parameters

Table 13–5 shows the parameters for the **Control** block.

Table 13–5. Parameters for the Control Block

Parameter	Description
Generate hardware	Turn on to enable output file generation.
Hardware destination directory	Specifies the root directory in which to write the output files. This location can be an absolute path or a relative path (for example, <code>../rtl</code>). A directory tree is created under this root directory that reflects the names of your model hierarchy.
Create automatic testbenches ⁽¹⁾	Turn on this option to generate additional automatic testbench files. These files capture the input and output of each block in a <code>.stm</code> file. DSP Builder creates a test harness (<code>_atb.vhd</code>) that simulates the generated RTL alongside the captured data. DSP Builder generates a script (<code><model>_atb.do</code>) that you can use to simulate the design in ModelSim and ensure bit and cycle accuracy between the Simulink model and the generated RTL.
Signal view depth	A <code>wave.do</code> file generates that you can use to open a Wave window in ModelSim. The <code>wave.do</code> file displays all important signals down to the specified level of hierarchy.
Turn on coverage in testbenches	If you are creating automatic testbenches, this option controls whether ModelSim's code coverage tools enable (if available).

Table 13–5. Parameters for the Control Block

Parameter	Description
Delay: register and RAM block threshold ⁽²⁾	Specifies the RAM block threshold in bits. If the number of logic cells the design requires is greater than the specified value, the design implements the delay with RAM blocks. Any value less than 0 means use the default 20 bits.
DualMem: small and medium threshold ⁽²⁾	Specifies the dual memory RAM threshold in bits. If the number of logic cells that a dual memory block requires is greater than the specified value, the design implements the delay with RAM blocks. Any value less than 0 means use the default 1,280 bits.
DualMem: medium and large threshold ⁽²⁾	Specifies the M-RAM threshold in bits. If the number of bits in memory is greater than the specified value, the design uses an M-RAM. Any value less than 0 means never use M-RAM or M144K.
Multiplier: logic and DSP threshold ⁽²⁾	Specifies the hard multiplier threshold in bits. The number of logic elements you want to use to save a multiplier. If the design exceeds the specified value, it uses hard multipliers. Any value less than 0 means always use hard multipliers.

Notes to Table 13–5:

- (1) For more information about this parameter, refer to *Comparison with RTL* in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.
- (2) For more information about these parameters, refer to *Basic Blocks* in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

From v14.1, the following parameters are in the **DSP Builder > Avalon Interfaces > Avalon-MM slave** menu or in the optional AvalonMMSlavesettings block:

- System address width
- System data width
- System bus is:

Hardware Generation

Options in the **Control** block specify whether hardware generates for your design example and the location of the generated RTL. You can also create automatic RTL testbenches for each subsystem in your design example and specify the depth of signals that DSP Builder includes when your design example simulates in ModelSim.

Memory-Mapped Bus Interface

You can specify the address and data bus widths that the memory-mapped bus interface use and specify whether DSP Builder stores the high-order byte of the address in memory at the lowest address and the low-order byte at the highest address (big endian), or the high-order byte at the highest address and the low-order byte at the lowest address (little endian).

Memory and Multiplier Trade-Off Options

When your design synthesizes to logic, DSP Builder creates delay blocks, whether explicitly from primitive delays, or in the ModelIP blocks. DSP Builder tries to balance the implementation between logic elements (LEs) and block memories (M512, M4K, M9K, or M144K). The trade-off depends on the target FPGA family, but as a guideline the trade-off is set to minimize the absolute silicon area the design uses. For example, if a block of RAM occupies the silicon area of two logic array blocks (LABs), the design implements delay that requires more than 20 LEs (two LABs) as a block of RAM. There may be cases when you want to influence this trade-off.

Table 13-6 describes the memory and multiplier threshold trade-offs and provides some usage examples.

Table 13-6. Memory and Multiplier Trade-Offs

CDelay RAM Block Threshold	
Description	Trade-off between simple delay LEs and small ROM blocks. If any delay's size is such that the number of LEs is greater than this parameter, the design implements the delay as a block RAM.
Default (-1)	20 bits.
Usage	To make more delays use block RAM, enter a lower number. For example a value in the range 20 to 30. To use fewer block memories, enter a larger number. For example 100. To never use block memory for simple delays, set to a very large number, such as 10,000.
Notes	Implement delays of length less than 3 cycles in LEs because of the block RAM behavior. This threshold only applies to implementing simple delays in memory blocks or LEs. You cannot push back dual memories into LEs. DSP Builder builds some ModelIP blocks with dual memories, for example dual and multirate filters; these blocks always use some memory blocks, regardless of the value of this threshold.
CDualMem Dist RAM Threshold	
Description	Trade-off between small and medium RAM blocks. This threshold is similar to the CDelay RAM Block Threshold except that it applies only to the dual-port memories in the ModelPrim library. Implement any dual-port memory in a block memory, rather than LEs, but for some device families there may be different sizes of block memory available. The threshold value determines which medium size RAM memory blocks to use instead of small memory RAM blocks. For example, the threshold that determines whether to use M9K blocks rather than MLAB blocks on Stratix IV devices.
Default (-1)	1,280 bits.
Usage	Use Stratix III devices with the default threshold value (-1), to implement dual memories greater than 1,280 bits as M9Ks and implement dual memories less than or equal to 1,280 bits as MLABs. If you change this threshold to a lower value such as 200, implement dual memories greater than 200 bits as M9Ks and implement dual memories less than or equal to 200 bits as MLABs.
M-RAM Threshold	
Description	Trade-off between medium and large RAM blocks. For larger delays, implement memory in medium block RAM (M4K, M9K) or use larger M-RAM blocks (M512K, M144K).
Default (-1)	1,000,000 bits.
Usage	If the number of bits in a memory or delay is greater than this threshold, DSP Builder uses M-RAM for the implementation. If you set a large value such as the default of 1,000,000 bits, M-RAM blocks are never used.
Hard Multiplier Threshold	
Description	Trade-off between hard and soft multipliers. For devices that support hard multipliers or DSP blocks, use these resources instead of a soft multiplier made from LEs. For example, a 2-bit × 10-bit multiplier consumes very few LEs. The hard multiplier threshold value corresponds to the number of LEs that save a multiplier. If the hard multiplier threshold value is 100, you are allowing 100 LEs. Therefore, an 18×18 multiplier (that requires approximately 182 = 350 LEs) does not transfer to LEs because it requires more LEs than the threshold value. However, DSP Builder implements a 16×4 multiplier that requires approximately 64 LEs as a soft multiplier with this setting.
Default (-1)	

Table 13–6. Memory and Multiplier Trade-Offs

Usage	<p>The default (–1) means always use hard multipliers. With this value, DSP Builder implements 24×18 as two 18×18 multipliers.</p> <p>Set a value of approximately 300 to keep 18×18 multipliers hard, but transform smaller multipliers to LEs. DSP Builder implements A 24×18 multiplier as 6×18 + 18×18, so this setting builds the hybrid multipliers that are required.</p> <p>Set a value of approximately 1000 to implement the multipliers entirely as LEs. Essentially you are allowing a high number (1000) of LEs to save an 18×18 multiplier.</p> <p>Set a values of approximately 10 to implement a 24×16 multiplier as a 36×36 multiplier. With this value, you are not even allowing the adder to combine the two multipliers, therefore the system has to burn a 36×36 multiplier in a single DSP block.</p>
Notes	<p>DSP Builder converts multipliers with a single constant input into balanced adder trees. This occurs automatically where the depth of the tree is not greater than 2. If the depth is greater than 2, DSP Builder compares the hard multiplier threshold with the estimated size of the adder tree, which is generally much lower than the size of a full soft multiplier.</p> <p>If DSP Builder combines two non-constant multipliers followed by an adder into a single DSP block, DSP Builder does not convert the multiplier into LEs, even if a large threshold is present.</p>

Updated Help

After you run a simulation, DSP builder updates the help pages with specific information about each instance of a block.

Device

The Device block marks a particular Simulink subsystem as the top-level design of an FPGA device and specifies a particular device.

DSP Builder generates project files and scripts that relate to this level of hierarchy. All blocks in subsystems below this level become part of the RTL design. All blocks above this level of hierarchy become part of the testbench.

You can insert multiple Device blocks in non-overlapping subsystems to use multiple FPGAs in the same design. You can mix device families freely.

Table 13–7 shows the parameters for the Device block.

Table 13–7. Parameters for the Device Block

Parameter	Description
Device family	Select the required target device family.
Device	Select the specific device.
Family member	Specifies the device member as free-form text or enter <code>AUTO</code> for automatic selection. If you enter free-form text, the name must start with <code>EPxxx</code> . For example: <code>EP2C35F484C6</code> . Click on ... to display the Device Selector .
Speed grade	You can select the speed grade for the FPGA target, which helps DSP Builder balance the hardware size against the resources required to meet the clock frequency set in the <code>Signals</code> block.

Edit Params

The `Edit Params` block is an easy way to open a file called `setup_<model name>.m`, which performs the following actions:

- Runs the script, `setup_<model name>.m`, which defines all the parameters required for the **design** `<model name>.mdl` as workspace variable. Ideally use names as `<model name>.clockrate`, `<model name>.channelcount`, and so on).
- Adds the script to the `PreloadFcn` callbacks of the model, so that it runs when you first open `<model name>.mdl`.
- Adds the script to the `InitFcn` callbacks of the model, so that it runs each time before DSP Builder simulates `<model name>.mdl` (in case you edited the script after opening the model and before simulating it).

The `Edit Params` block is available as a functional block in the Simulink library browser. To view it you must open the library, by right clicking on the **Base Blocks** library in the Simulink Library Browser and selecting **Open Base Blocks Library**.

Alternatively, you can create your own `Edit Params` block as a graphical shortcut to a MATLAB M-script that sets up and initializes your design, or runs at any other stage during the simulation.

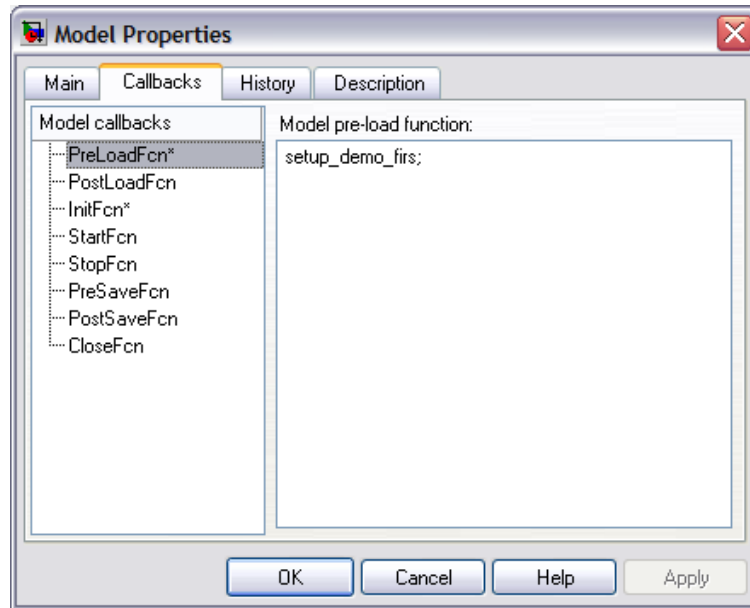
Examples of `Edit Params` blocks are in many of the design examples.

You can create an `Edit Params` block if you create an M-script defining workspace variables that you want to call when your model loads or initializes. Specify this script in the `PreLoadFcn` or `InitFcn` callbacks that you can access by clicking **Model Properties** on the File menu:

- To call your script automatically when your model opens, add a `PreloadFcn` reference to your script in the **Callbacks** tab of your Model Properties in Simulink.
- To call your script automatically at the start of a simulation run, add a `InitFcn` reference to your script in the **Callbacks** tab of your Model Properties in Simulink.

In the design examples, the names of the initialization scripts derive from your model name: `setup_<model name>.m`. For example, `setup_demo_firs.m` (Figure 13-2).

Figure 13-2. Callbacks Tab in the Simulink Model Properties Dialog Box

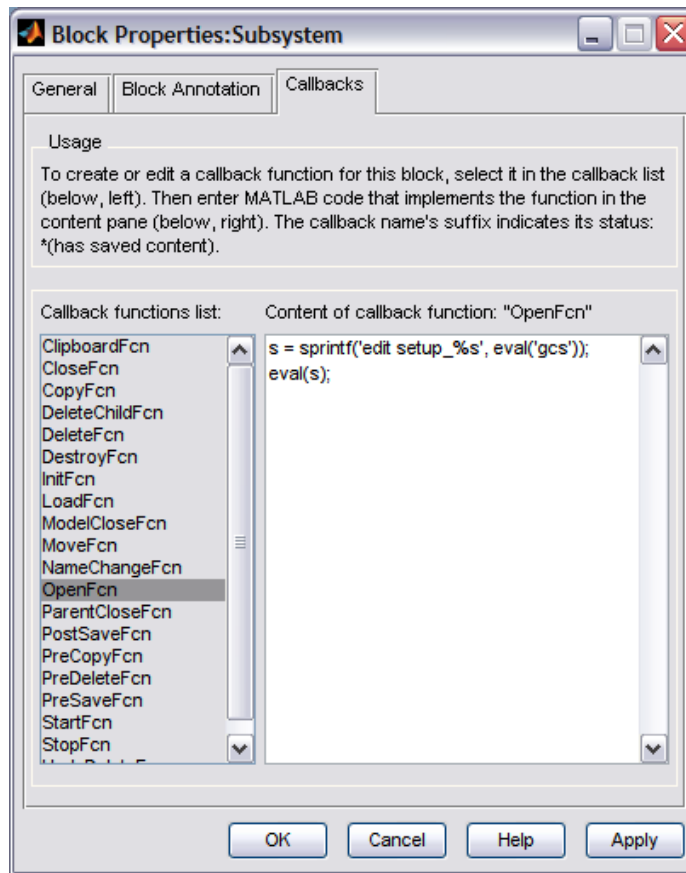


To create an Edit Params block that references this script, perform the following steps:

1. Drag a Subsystem block from the Simulink Commonly Used Blocks library.
2. Double-click on the Subsystem block to open it and delete the default In1 and Out1 ports. Close the Subsystem block.
3. Right-click on the Subsystem block, select **Block Properties** and click the **Callbacks** tab.

4. Select `OpenFcn` in the **Callback functions list** (Figure 13-3).

Figure 13-3. Callbacks in the Subsystem Block Properties Dialog Box




5. If you name your script `setup_<model name>.m`, enter the following functions:

```
s = sprintf('edit setup_%s', eval('gcs'));
eval(s);
```

Alternatively, you can explicitly reference a script such as `my_script.m` by entering a function of the form:

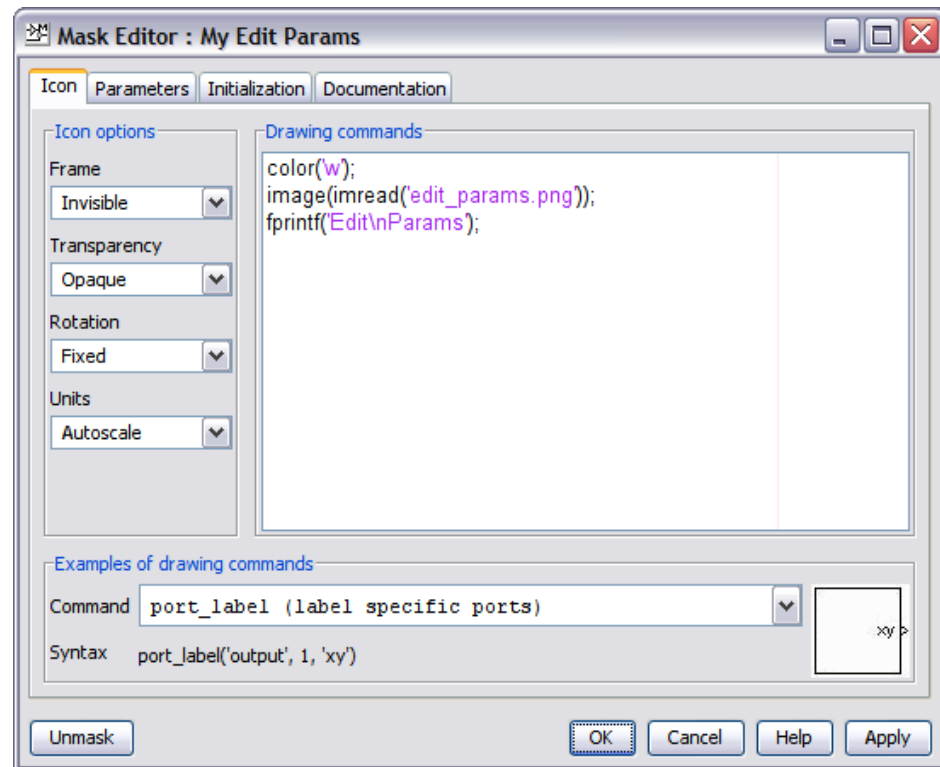
```
eval('edit my_script.m');
```

6. Click **OK** to close the Block Properties dialog box.
7. Rename the Subsystem block `EditParams` (or any name of your choice).

 You can optionally hide the block name by right-clicking and clicking **Hide Name** on the popup menu.

8. If you want to apply a graphic icon to your block, right-click and select **Mask Subsystem** to display the Mask Editor dialog box (Figure 13-4).


Figure 13-4. Mask Editor Dialog Box



9. Click the **Icon** tab and enter drawing commands using the following format:

```
image(imread('edit_params.png'));
color('w');
fprintf('Edit\nParams');
```

These commands reference the default icon for an Edit Params block, set the text color to white and specify the text overlay on the block. You can optionally specify your own custom graphic file, color, and text.

 For more information about drawing commands in the mask editor, refer to the Simulink Help.

10. Select Invisible from the **Frame** icon options.
11. Click on the **Documentation** tab and enter the **Mask type**: DSP Builder Advanced Blockset Ignored Block.
12. Click **OK** to close the Mask Editor.



To create an Edit Params block copy an existing block from one of the design examples, and edit the block and mask properties to customize it for your model.

Usage

Double-click on the `Edit Params` block to open the script in the MATLAB text editor.

Enhanced Precision Support

The `Enhanced Precision Support` block implements a separate code path for variable precision floating-point types. When you place an `enhanced Precision Support` block in a system, DSP Builder processes the system and all of its subsystems using this separate code path.

You should add the block at the lowest level possible in the design. When added, you can specify variable precision output types in the `Math` and `Convert` blocks. You can only specify variable precision output types in subsystems that contain an `Enhanced Precision Support` block. If you use variable precision floating-point types in other subsystems, DSP Builder reports an error.

The following features do not work in a system that has an `Enhanced Precision Support` block, or any subsystems:

- Enabled subsystems
- Bus blocks
- Demux primitive block

LocalThreshold

The `LocalThreshold` block allows hierarchical overrides of the global clock margin and threshold settings set on the `Control` and `Signals` blocks.

You can place the `LocalThreshold` block anywhere in your design to define over-ride values for the margin and threshold settings for that subsystem and any embedded subsystems. You can over-ride these values further down in the hierarchy by implementing more `LocalThreshold` blocks.

For example, you can specify different clock margins for different regions of your design.

Parameters

Table 13-5 shows the parameters for the `Control` block.

Table 13-8. Parameters for the Control Block

Parameter	Description
Clock margin (MHz)	Specifies the margin requested to achieve a high system frequency in the fitter. The specified margin does not affect the folding options because the system runs at the rate specified by the Clock frequency parameter setting. Specify a positive clock margin if you need to pipeline your design more aggressively (or specify a negative clock margin to save resources) when you do not want to change the ratio between the clock speed and the bus speed.
Delay: register and RAM block threshold	Specifies the RAM block threshold in bits. If the number of logic cells the design requires is greater than the specified value, the design implements the delay with RAM blocks. Any value less than 0 means use the default 20 bits.

Table 13–8. Parameters for the Control Block

Parameter	Description
DualMem: small and medium threshold	Specifies the dual memory RAM threshold in bits. If the number of logic cells that a dual memory block requires is greater than the specified value, the design implements the delay with RAM blocks. Any value less than 0 means use the default 1,280 bits.
DualMem: medium and large threshold	Specifies the M-RAM threshold in bits. If the number of bits in memory is greater than the specified value, the design uses an M-RAM. Any value less than 0 means never use M-RAM or M144K.
Multiplier: logic and DSP threshold	Specifies the hard multiplier threshold in bits. The number of logic elements you want to use to save a multiplier. If the design exceeds the specified value, it uses hard multipliers. Any value less than 0 means always use hard multipliers.

This block has no inputs or outputs.

Run ModelSim

The Run ModelSim block works by stimulus capture, in the same way as the automatic testbenches (ATBs), and compares the Simulink simulation result to the ModelSim simulation.



Altera recommends you do not use this block for new designs. To run a simulation, on the DSP Builder menu, click **Run ModelSim**.

Usage

Instantiate the Run ModelSim block into the top-level of any design. Double-click the Run ModelSim block to launch the ModelSim tool with the user system as the top-level entity. ModelSim automatically starts compiling your design and runs a simulation for the same number of clock cycles as the Simulink simulation.

The simulation compiles and loads with the .do scripts that generate when you run a Simulink simulation. DSP Builder creates the .do scripts in the hardware destination directory that you specify in the Control block. DSP Builder creates the generated files and scripts in a hierarchy of directories that match the hierarchy of your design and calls lower level scripts from the scripts in the top-level directory.

Run Quartus II

The Run Quartus II block allows automatic launch of the user system in the Quartus II software, where you can verify your design for performance and logic utilization.



Altera recommends you do not use this block for new designs. To run a the Quartus II software, on the DSP Builder menu, click **Run Quartus II**.

Usage

Instantiate the Run Quartus II block into the top-level of any design containing an Altera Device block. Double-click the Run Quartus II block to launch the Quartus II software with the user system as the top-level entity.

DSP Builder creates the Quartus II project file (**.qpf**), Quartus II settings file (**.qsf**), and Quartus II IP file (**.qip**) for the Quartus II project in your design directory that contains the **.mdl** file. These files contain all required references to the files in the hardware destination directory that the **Control** block specifies and are generated when you run a Simulink simulation. After loading the project in the Quartus II software, you should check the required device settings. You can then compile your design by clicking **Start Compilation** on the Processing menu.

The project compiles with the **.tcl** scripts in the hardware destination directory. DSP Builder creates the generated files and scripts in a hierarchy of directories that match the hierarchy of your design and calls lower level scripts from the scripts in the top-level directory.

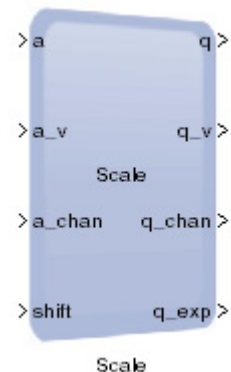
The Run Quartus block allows you to quickly validate the performance of the design and the corresponding resources. The project is created with default settings and is regenerated each time the block is run. Altera does not intend this flow as a means to generate the final deliverable. You should instantiate the top level RTL design directly, and use the **.tcl** or **.qip** flow to incorporate the DSP Builder advanced blockset design into the top-level Quartus II project.

Scale

You can use the **Scale** block to select part of a wide input word, performing various types of rounding, saturation and fixed-point scaling, to produce an output of specified precision.

By default, DSP Builder preserves the binary point so that the fixed-point interpretation of the result has the same value, subject to rounding, as the fixed-point interpretation of the input.

You can dynamically perform additional scaling, by specifying a variable number of bits to shift, allowing you to introduce any power of two gain.



Always use **Scale** blocks to change data types in preference to **Convert** blocks, because they vectorize and automatically balance the delays with the corresponding valid and channel signals.

The **Scale** block provides scaling in addition to rounding and saturation to help you manage bit growth. The basic functional modules of a **Scale** block are shifts followed by rounding and saturation. The multiplication factor is simply a constant scale to apply to the input. Normally this would be 1.

The number of bits to shift left allows you to select the most meaningful bits of a wide word, and discard unused MSBs. You can specify the number of shifts as a scalar or a vector. The block relies on shift input port to decide which value to use if you specified the number of shifts as a vector. The shift input signal selects which gain to use cycle-by-cycle.

In a multichannel design, changing the shift value cycle-by-cycle allows you to use a different scaling factor for different channels.

A positive number of **Number of bits to shift left** indicates that the MSBs are discarded, and the *Scale* block introduces a gain to the input. A negative number means that zeros (or 1 in the signed data case) are padded to the MSBs of the input data signal, and the output signal is attenuated.

Parameters

Table 13–9 shows the parameters for the *Scale* block.

Table 13–9. Parameters for the Scale Block

Parameter	Description
Output data type	The type of the result. For example: <code>sfix(16), uint(8)</code> .
Output scaling value	The scaling of the result if the result type is fixed-point. For example: 2^{-15} .
Rounding method	Specifies one of the following three rounding methods for discarding the least significant bits (LSBs): <ul style="list-style-type: none"> ■ Truncate: truncates the least significant bits. Has the lowest hardware usage, but introduces the worst bias. ■ Biased: rounds up if the discarded bits are 0.5 or above. ■ Unbiased: rounds up if the discarded bits are greater than 0.5, and rounds to even if the discarded bits equal 0.5.
Multiplication factor	Modify the interpreted value by scaling it by this factor. This factor does not affect the hardware generated for the <i>Scale</i> block, but merely affects the interpretation of the result. For example: 1, 2, 3, 4, 8, 0.5.
Saturation method	Specifies one of the following three saturation methods for discarding the most significant bits (MSBs): <ul style="list-style-type: none"> ■ None: no saturation is performed. ■ Asymmetric: the range of the number produced occupies the whole of the two's complement range (for example -1.0 to 0.999). One more negative number is available, which introduces a slight bias. ■ Symmetric: the range of the result is clipped to between symmetrical boundaries (for example -0.999 and 0.999). Ensures no bias enters the dataflow.
Number of bits to shift left	A scalar or a vector that determines the gain of the result. A positive number indicates that the <i>Scale</i> block introduces a gain to the input. A negative number means that the output signal is attenuated. A vector of gains allows the shift input signal to select which gain to use on a cycle per cycle basis. The value of the shift integer performs zero-based indexing of the vector. For example: 2, -4, [0 1 2 3].

Port Interface

Table 13–10 shows the port interface for the *Scale* block.

Table 13–10. Port Interface for the Scale Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit onto a single bus, this signal is a vector. The width in bits is inherited from the input wire.
a_v	Input	Indicates validity of data input signals. If a_v is high then the data on the a wire is valid.
a_chan	Input	Indicates channel of data input signals. If a_v is high, then a_chan indicates to which channel the data corresponds.

Table 13-10. Port Interface for the Scale Block

Signal	Direction	Description
shift	Input	Indicates which element of the zero-based shift vector to use.
q	Output	The scaled fixed-point data output from the block. If you request more channels than can fit onto a single bus, this signal is a vector. The width in bits is calculated as a function of the input width in bits and the parameterization.
q_v	Output	Indicates validity of data output signals.
q_chan	Output	Indicates channel of data output signals.
q_exp	Output	Indicates whether the output sample has saturated or overflowed.

Updated Help

After you run a simulation, DSP Builder updates the help pages with specific information about each instance of a block. [Table 13-11](#) shows some typical help messages that DSP Builder issues for the `Scale` block.

Table 13-11. Messages for the Scale Block

Message Example	Description
Written on Tue Feb 19 11:25:27 2008	Date and time when this file ran.
Number of physical buses: 4	Depending on the input data rate, the number of data wires needed to carry the input data may be more than 1.
Calculated bit width of output stage: 16	The width in bits of the (vectorized) data output.
Latency is 2	The latency introduced by this block.
Parameters table	Lists the current rounding and saturation modes.
Port interface table	Lists the port interfaces to the <code>Scale</code> block.

Signals

The `Signals` block specifies information about the system clock, reset, and memory bus signals that the simulation model and the hardware generation use. DSP Builder uses the names of the signals to generate the RTL.



You must include a `Signals` block at the top level of your model.

Parameters

[Table 13-12](#) shows the parameters for the `Signals` block.

Table 13-12. Parameters for the Signals Block

Parameter	Description
Clock	Specifies the name of the system clock signal that DSP Builder uses in the RTL generation.
Clock frequency (MHz)	Specifies the system clock rate for the system.
Clock margin (MHz)	Specifies the margin requested to achieve a high system frequency in the fitter. The specified margin does not affect the folding options because the system runs at the rate specified by the Clock frequency parameter setting. Specify a positive clock margin if you need to pipeline your design more aggressively (or specify a negative clock margin to save resources) when you do not want to change the ratio between the clock speed and the bus speed.

Table 13–12. Parameters for the Signals Block

Parameter	Description
Reset	Specifies the name of the reset signal that DSP Builder uses in the RTL generation.
Reset active	Specifies whether the logic generated is reset with an active High or active Low reset signal.

From v14.1, the following parameters are in the **DSP Builder > Avalon Interfaces > Avalon-MM slave** menu or in the optional AvalonMMSlavesettings block:

- Bus name
- Separate bus clock
- Bus clock frequency (MHz)
- Bus clock synchronous with system clock

Updated Help

After you run a simulation, DSP Builder updates the help pages with specific information about each instance of a block. [Table 13–13](#) shows a typical help message that DSP Builder issues for the Signals block.

Table 13–13. Messages for the Signals Block

Message Example	Description
Parameters table	Lists the system clock name, system clock rate, fitter target frequency, reset name, and reset active parameters for the current model.

This library contains common blocks that support FFT design. These blocks are masked subsystems of structures built from primitive blocks. The library also includes support for a radix-2² algorithm.

The radix-2² architecture is a serial version of the radix-4 architecture. It computes a radix-4 butterfly over four (not necessarily consecutive) inputs and produces four (not necessarily consecutive) outputs.



For more information about the radix-2² algorithm, refer to *A New Approach to Pipeline FFT Processor* – Shousheng He & Mats Torkleson, Department of Applied Electronics, Lund University, Sweden.

The FFT library contains the following blocks in the **Common** folder:

- Butterfly Unit (BFU)
- Bit Vector Combine (BitVectorCombine)
- Choose Bits (ChooseBits)
- Crossover Switch (XSwitch)
- Dual Twiddle Memory (DualTwiddleMemoryC)
- Edge Detect (EdgeDetect)
- FFT (FFT, FFT_Light, VFFT, VFFT_Light)
- Multiwire Transpose (MultiwireTranspose)
- Pulse Divider (PulseDivider)
- Pulse Multiplier (PulseMultiplier)
- Single-Wire Transpose (Transpose)
- Split Scalar (SplitScalar)
- Twiddle ROM (TwiddleRom, TwiddleMultRom and TwiddleRomF (deprecated))

The FFT library contains the following blocks in the **Floating Point** folder:

- Butterfly Unit (BFU)
- FFT (FFT, FFT_Light, VFFT, VFFT_Light)

The following deprecated blocks are available in the **obsolete** folder:

- Butterfly I C (BFIC) (Deprecated)
- Butterfly II C (BFIIC) (Deprecated)
- Floating-Point Twiddle Generator (TwiddleGenF) (Deprecated)
- Twiddle Generator (TwiddleGenC) (Deprecated)
- Twiddle ROM (TwiddleRom, TwiddleMultRom and TwiddleRomF (deprecated))



Do not use deprecated blocks in new designs.

The following blocks are available in the **parallel** folder:

- Fully-Parallel FFTs (FFT2P, FFT4P, FFT8P, FFT16P, FFT32P, and FFT64P)
- Fully-Parallel FFTs with Flexible Ordering (FFT2X, FFT4X, FFT8X, FFT16X, FFT32X, and FFT64X)
- General Multitwiddle and General Twiddle (GeneralMultiTwiddle, GeneralTwiddle)
- Hybrid FFT (Hybrid_FFT)
- Parallel Pipelined FFT (PFFT_Pipe)

The following blocks are available in the **Radix2** folder:

- Bit Reverse Core C (BitReverseCoreC and VariableBitReverse)
- Twiddle Angle (TwiddleAngle)

Bit Reverse Core C (BitReverseCoreC and VariableBitReverse)

The BitReverseCoreC block performs buffering and bit reversal of incoming FFT frames.

A single synthesis time parameter specifies the length N of the fast Fourier transform.

The bit reversal that this block applies is appropriate only for transform sizes that are an integer power of two. The block is single-buffered to support full streaming operation with minimal overhead.

The VariableBitReverse block performs buffering and bit-reversal of variable-sized FFT frames for designs with a VFFT or FP_VFFT block. A single synthesis-time parameter N specifies the length 2^N of the largest frame that the block handles. The VariableBitReverse block has an additional input: size, which specifies the length 2^{size} of the current frame.

To reconfigure the VariableBitReverse block between frames, observe the following rules:

- Ensure the size input is always in the range $0 \leq \text{size} \leq N$.
- Keep the size input constant while the VariableBitReverse block is processing a frame.
- When you reconfigure the VariableBitReverse, you must completely flush the VariableBitReverse block before changing the value of the size input. You must wait at least 2^{oldSize} (where *oldSize* is the previous value of the size input) cycles before providing valid input to the VFFT.

Parameters

Table 14-1 shows the parameters for the BitReverseCoreC block.

Table 14-1. Parameters for the BitReverseCoreC Block

Parameter	Description
FFT Size	Specifies the size of the FFT.

Table 14-2 shows the parameters for the VariableBitReverse block.

Table 14-2. Parameters for the VariableBitReverse Block

Parameter	Description
N	Logarithm of the maximum frame size.

Port Interface

Table 14-3 shows the port interface for the BitReverseCoreC block.

Table 14-3. Port Interface for the BitReverseCoreC Block

Signal	Direction	Type	Description
v	Input	Boolean	Valid input signal.
c	Input	Unsigned 8-bit integer	Channel input signal.
size	Input	Unsigned integer	Logarithm of the current input frame size. VariableBitReverse only.
x	Input	Any complex fixed-point (BitReverseCoreC); any (VariableBitReverse)	Complex data input signal.
qv	Output	Boolean	Valid output signal.
qsize	Output	Unsigned integer	Logarithm of the current output frame size. VariableBitReverse only.
qc	Output	Unsigned 8-bit integer	Channel output signal.
q	Output	Any	Complex data output signal.

Bit Vector Combine (BitVectorCombine)

The BitVectorCombine block concatenates a vector of bits to form a scalar. The scalar is an unsigned integer of the appropriate width. The first element of the vector becomes the least significant bit of the scalar (little-endian ordering).

Use the BitVectorCombine block to recombine scalars that the SplitScalar block splits.

Parameters

Table 14-4 shows the parameters for the BitVectorCombine block.

Table 14-4. Parameters for the BitVectorCombine Block

Parameter	Description
Width	Width of the input vector (and the output scalar).

Port Interface

Table 14-5 shows the port interface for the BitVectorCombine block.

Table 14-5. Port Interface for the BitVectorCombine Block

Signal	Direction	Type	Description
d	Input	Boolean vector.	Data input.
q	Output	Unsigned integer.	Data output.

Butterfly Unit (BFU)

The BFU, BFU_long, and BFU_short blocks each implement a butterfly unit for use in floating-point streaming FFTs.

The BFU_long block corresponds to a classical radix-2² butterfly I block plus its associated feedback path.

The BFU_short block has exactly the same functionality, but it uses only one floating-point adders. It uses twice as many memory resources as the BFU_long block, but also uses considerably less logic resources.

The BFU block automatically reconfigures to use either BFU_long or BFU_short to minimize the total (memory plus logic) resource usage.

Each BFU block performs a two-point FFT pass over a block of data of size 2^N (where N is a compile-time parameter).

During the first 2^(N-1) cycles, the control signal, s, is 0. During this time, the BFU block stores the first half of the input block.

During the second 2^(N-1) cycles, s is 1. During this time, the BFU block reads the second half of the input block and produces the first result of each of 2^(N-1) two-point FFTs on the output.

During the third 2^(N-1) cycles, s is 0 again. During this time, the BFU unit produces the second result of each of the 2^(N-1) two-point FFTs, while simultaneously storing the first half of the next input block.

Parameters

Table 14-6 shows the parameters for the BFU block.

Table 14-6. Parameters for the BFU Block

Parameter	Description
N	Specifies the input block size to be 2 ^N .

Port Interface

Table 14-7 shows the port interface for the BFU block.

Table 14-7. Port Interface for the BFU Block

Signal	Direction	Type	Description
d	Input	Any floating-point type.	Input samples.
q	Output	Same as d.	Output results.
s	Input	Boolean.	Control pin. Drive with external logic. Ensure it is 0 for $2^{(N-1)}$ cycles and 1 for the next $2^{(N-1)}$ cycles.

Butterfly I C (BFIC) (Deprecated)

The BFIC block implements the butterfly I functionality associated with the radix-2² fully streaming FFT architecture.

You should parameterize this block with the incoming data type to ensure that DSP Builder maintains the necessary data precision. At the output, DSP Builder applies an additional bit of growth.

The s port connects to the control logic. This control logic is the extraction of the appropriate bit of a modulo *N* counter. The value of s determines the signal routing of each sample and the mathematical combination with other samples.

Parameters

Table 14-8 shows the parameters for the BFIC block.

Table 14-8. Parameters for the BFIC Block

Parameter	Description
Input bits	Specifies the number of input bits.
Input scaling exponent	Specifies the fixed-point scaling factor of the input.

Port Interface

Table 14-9 shows the port interface for the BFIC block.

Table 14-9. Port Interface for the BFIC Block

Signal	Direction	Type	Description
s	Input	Boolean or unsigned integer uint(1)	Control pin.
x1	Input	Complex fixed-point data-type determined by parameterization	Complex data input from ComplexSampleDelay.
x2	Input	Complex fixed-point data-type determined by parameterization	Complex data input from previous stage.
z1	Output	Complex fixed-point data-type determined by parameterization	Complex data output to next stage.
z2	Output	Complex fixed-point data-type determined by parameterization	Complex data output to ComplexSampleDelay.

Butterfly II C (BFIIC) (Deprecated)

The BFIIC block implements the butterfly II functionality associated with the radix-2² fully streaming FFT or iFFT architecture.

You should parameterize this block with the incoming data type to ensure that DSP Builder maintains the necessary data precision. At the output, DSP Builder applies an additional bit of growth.

The *s* port connects to the control logic. This control logic is the extraction of the appropriate bit of a modulo *N* counter. The value of *s* determines the signal routing of each sample and the mathematical combination with other samples. The *t* port also connects to the control logic, but the extracted bit is different from the *s* port. The value of *t* determines whether an additional multiplication by $-j$ occurs inside the butterfly unit.

Parameters

Table 14-10 shows the parameters for the BFIIC block.

Table 14-10. Parameters for the BFIIC Block

Parameter	Description
IFFT	Specifies that the design uses the BFIIC block in an IFFT.
Input bits	Specifies the number of input bits.
Input scaling exponent	Specifies the exponent part of the input scaling factor ($2^{-\text{exponent}}$).
Allow output bitwidth growth	Specifies that the output is one bit wider than the input.

Port Interface

Table 14-11 shows the port interface for the BFIIC block.

Table 14-11. Port Interface for the BFIIC Block

Signal	Direction	Type	Description
<i>s</i>	Input	Boolean	Control pin.
<i>t</i>	Input	Boolean	Control pin.
<i>x1</i>	Input	Complex fixed-point data-type determined by parameterization	Complex input from ComplexSampleDelay.
<i>x2</i>	Input	Complex fixed-point data-type determined by parameterization	Complex input from previous stage.
<i>z1</i>	Output	Derived complex fixed-point type	Complex output to next stage.
<i>z2</i>	Output	Derived complex fixed-point type	Complex output to ComplexSampleDelay.

Choose Bits (ChooseBits)

The ChooseBits block selects individual bits from its input (scalar) signal and concatenates them to form its (scalar) output signal.

You specify the bits that occur in the output signal by providing a vector of non-negative integers. Each integer specifies an input bit appears in the output. The block numbers the input bits from 0 (least significant bit) and lists the output bits starting from the least significant bit (little-endian ordering).

The block has no restriction on how many times each input bit may appear in the output. You can omit, reorder, or duplicate bits.

For example, the vector [0,1,4,4,6,5] keeps bits 0 and 1 unchanged, omits bit 3, duplicates bit 4 and swaps the positions of bits 5 and 6.

Parameters

Table 14-12 shows the parameters for the ChooseBits block.

Table 14-12. Parameters for the ChooseBits Block

Parameter	Description
Selected bits	A vector of non-negative integers.

Port Interface

Table 14-13 shows the port interface for the ChooseBits block.

Table 14-13. Port Interface for the ChooseBits Block

Signal	Direction	Type	Description
d	Input	Any scalar.	Data input.
q	Output	Unsigned integer.	Data output.

Crossover Switch (XSwitch)

The XSwitch block a simple crossover switch.

Port Interface

Table 14-14 shows the port interface for the XSwitch block.

Table 14-14. Port Interface for the XSwitch Block

Signal	Direction	Type	Description
swap	Input	Boolean.	Control input. When swap is 0, d0 is routed to q0 and d1 is routed to q1; when swap is 1, d0 is routed to q1 and d1 is routed to q0.
d0	Input	Any.	Data input.
d1	Input	Same as d0.	Data input.
q0	Output	Same as d0.	Data output.
q1	Output	Same as d0.	Data output.

Dual Twiddle Memory (DualTwiddleMemoryC)

The DualTwiddleMemory block calculates the complex twiddle factors associated with the evaluation of $\exp(-2\pi i \cdot k1/N)$ and $\exp(-2\pi i \cdot k2/N)$.

This block uses an efficient dual-port architecture to minimize the size of the internal lookup table while supporting the generation of two complex twiddle factors per clock cycle. The block provides $k1$ and $k2$ at the input and they must be less than or equal to a synthesis time parameter N . Enter the width in bits and fixed-point scaling of the twiddle factors.

A cosine/sine wave has a range of $[-1:1]$, so you must provide at least two integer bits, and as many fractional bits as are appropriate. A good starting point is a twiddle width in bits of 16 bits (enter 16 as the **Precision**), and a scaling of 2^{-14} (enter 14 as the **Scaling** exponent). The resulting fixed-point type is `sfixed16_en14` (2.14 in fixed-point format).

Parameters

Table 14-15 shows the parameters for the DualTwiddleMemoryC block.

Table 14-15. Parameters for the DualTwiddleMemoryC Block

Parameter	Description
Number of points (N)	Specifies the number of points on the unit circle.
Precision	Specifies the precision in bits of the twiddle factors.
Twiddle scaling exponent	Specifies the fixed-point scaling factor of the complex twiddle factor.

Port Interface

Table 14-16 shows the port interface for the DualTwiddleMemoryC block.

Table 14-16. Port Interface for the DualTwiddleMemoryC Block

Signal	Direction	Type	Description
k1	Input	Unsigned integer in range 0 to $(N - 1)$	Desired twiddle factor index.
k2	Input	Unsigned integer in range 0 to $(N - 1)$	Desired twiddle factor index.
q1	Output	Type determined by parameterization	Twiddle factor 1 (complex).
q2	Output	Type determined by parameterization	Twiddle factor 2 (complex).

Edge Detect (EdgeDetect)

The EdgeDetect block implements a simple circuit that detects edges on its input. It outputs 0 if the current input is the same as the previous input and 1 if the inputs are different.

Parameters

The EdgeDetect block has no parameters.

Port Interface

Table 14-17 shows the port interface for the EdgeDetect block.

Table 14-17. Port Interface for the EdgeDetect Block

Signal	Direction	Type	Description
d	Input	Boolean or <code>ufix(1)</code> .	Data input.
q	Output	Same as input	Data output.

FFT (FFT, FFT_Light, VFFT, VFFT_Light)



The following blocks are deprecated and in the **obsolete** directory:

- FFT_Light (replaced with new version in **common** directory)
- FP_FFT_Light
- FP_VFFT_Light
- VFFT_Light (replaced with new version in **common** directory)



Do not use deprecated blocks in new designs.



For new designs use the following blocks in the common directory, which have new parameter interfaces:

- FFT
- FFT_Light
- VFFT
- VFFT_Light



The FP_FFT and FP_VFFT in the radix directory use the pre v14.0 parameters.

The FFT and VFFT blocks support processing multiple interleaved FFTs. The number of interleaved FFTs must be a power of 2. Each FFT is independent except that all the input for all of the FFTs must arrive as a single contiguous block of data. For example, with 8 FFTs each of size 1K each input block must contain 8K points.

For floating-point FFTs, select either correct or faithful rounding. Correct rounding corresponds to the normal IEEE semantics; faithful rounding delivers less accurate results but requires less logic to implement.

FFT

The FFT block provides a full radix-2² streaming FFT or IFFT. Use the FFT block for fixed-point or floating-point data. The blocks is a scheduled subsystem.

The FFT_Light block is a light-weight variant. However, it is not a scheduled subsystem, and it doesn't implement the *c* (channel) signal. The blocks provide an output signal, *g*, which pulses high at the start of each output block.

The FFT blocks all support block-based flow control. You must supply all the input data required for a single FFT iteration (one block) on consecutive clocks cycles, but an arbitrary large (or small) gap can exist between consecutive blocks. The BitReverseCoreC and Transpose blocks produces data in blocks that respect this protocol.

You may provide the input data to any of these block in either natural or bit-reversed order; the output result is in bit-reversed or natural order, respectively.

VFFT

The VFFT block provides a variable-size streaming FFT or IFFT. For these blocks, you statically specify the largest and smallest FFT that the block handles. You can dynamically configure the number of points processed in each FFT iteration using the size signal.

Use the VFFT block for fixed-point or floating-point data. The VFFT blocks is a scheduled subsystem and implements *v* (valid) and *c* (channel) signals.

The VFFT_light blocks is a light-weight variants of the VFFT block. It is not a scheduled subsystem, and it doesn't implement the *c* (channel) signal. Instead, it provides an output *g* signal, which pulses high at the start of each output block.

The VFFT blocks all support block-based flow control. You must supply all the input data required for a single VFFT iteration (one block) on consecutive clocks cycles. If you use two successive FFT iterations that use the same FFT size, the inter-block gap can be as small (or as large) as you like.

However, if you want to reconfigure the VFFT block between FFT iterations, you must use the following rules:

- The size input should always be in the range $minSize \leq size \leq maxSize$.
- The size input must be kept constant while the VFFT block processes an FFT iteration.
- When you reconfigure the VFFT, you must completely flush VFFT pipeline before changing the value of the size input. You must wait at least $2^{oldSize}$ (where *oldSize* is the previous value of the size input) cycles before providing valid input to the VFFT.



The VariableBitReverse block also requires an inter-block gap of $2^{oldSize}$ cycles when you reconfigure its size. If you use both the VariableBitReverse block and the VFFT block, you need to provide an interblock gap of $2 \cdot (2^{oldSize})$ cycles to allow both blocks to reconfigure successfully.

Parameters

Table 14-18 shows the parameters for the FFT and VFFT blocks. Not all parameters are available with all blocks.

Table 14-18. Parameters for the FFT and VFFT Blocks

Parameter	Description
iFFT	true to implement an IFFT, otherwise false.
Number of interleaved subchannels	Enter how many FFTs that DSP Builder interleaves in each block.
Bit-reversed input	true if you expect bit-reversed input, otherwise false.
N	The logarithm of the FFT size. FFT and FFT_Light only
maxSize	The logarithm of the maximum FFT size. VFFT and VFFT_Light only.
minSize	The logarithm of the minimum FFT size. VFFT and VFFT_Light only.
Input type	Input signal type.
Input scaling exponent	The fixed-point scaling factor of the input.
Twiddle/pruning specification(“Pruning and Twiddle Specification for FFTs” on page 10-35.
Use faithful rounding	true if the block uses faithful (rather than correct) rounding for floating-point operations. Fixed-point FFTs ignore this parameter.

Table 14-32 shows the parameters for the FFT blocks before v14.0. Not all parameters are available with all blocks.

Table 14-19. Parameters for the FFT Blocks

Parameter	Description
FFT type	An FFT or an IFFT block.
N	The logarithm of the FFT size.
maxsize	The logarithm of the maximum FFT size. FP_VFFT, FP_VFFT_Light, VFFT and VFFT_Light only.
minsize	The logarithm of the minimum FFT size. FP_VFFT, FP_VFFT_Light, VFFT and VFFT_Light only.
Input Order	Bit-reversed or natural order input.
Input bits	Width of the input signal. (V) FFT and (V) FFT_Light only.
Input scaling exponent	The fixed-point scaling factor of the input. (V) FFT and (V) FFT_Light only.
Twiddle factor bit width	Width of the twiddle factors (V) FFT and (V) FFT_Light only.
Output bit width	Width of the output signal. (V) FFT and (V) FFT_Light only.
Twiddle type	The floating-point type to use for the twiddle factors. FP_(V) FFT and FT_(V) FFT_Light only.

Port Interface

Table 14–33 shows the port interface for the FFT blocks. Not all signals are available with all blocks

Table 14–20. Port Interface for the FFT Blocks

Signal	Direction	Type	Description
v	Input	Boolean.	Valid input signal.
c	Input	Unsigned 8-bit integer.	Channel input signal FFT and VFFT, only.
size	Input	Unsigned integer.	Logarithm of the current FFT size. VFFT and VFFT_Light only.
d	Input	Any complex fixed-point.	Complex data input signal. VFFT and VFFT_Light only.
x	Input	Any complex fixed-point type (FFT and FFT_light). Any floating-point type (FP_FFT or FP_FFT_Light).	Complex data input signal.
qv	Output	Boolean.	Valid output signal.
qc	Output	Unsigned 8-bit integer.	Channel output signal. FFT and VFFT, only.
q	Output	Same as x.	Complex data output signal.
g	Output	Boolean	Start of output block. VFFT_Light only.

Table 14–33 shows the port interface for the FFT blocks before v14.0. Not all signals are available with all blocks

Table 14–21. Port Interface for the FFT Blocks

Signal	Direction	Type	Description
v	Input	Boolean.	Valid input signal.
c	Input	Unsigned 8-bit integer.	Channel input signal (V) FFT and (V) FP_FFT only.
size	Input	Unsigned integer.	Logarithm of the current FFT size. FP_VFFT, FP_VFFT_Light, VFFT and VFFT_Light only.
d	Input	Any complex fixed-point.	Complex data input signal. FP_VFFT, FP_VFFT_Light, VFFT and VFFT_Light only.
x	Input	Any complex fixed-point type (FFT and FFT_light). Any floating-point type (FP_FFT or FP_FFT_Light).	Complex data input signal.
qv	Output	Boolean.	Valid output signal.
qc	Output	Unsigned 8-bit integer.	Channel output signal. (V) FFT and (V) FP_FFT only.
q	Output	Same as x.	Complex data output signal.
g	Output	Boolean	Start of output block. (V) FFT_Light and FP_(V) FFT_Light only.

Floating-Point Twiddle Generator (TwiddleGenF) (Deprecated)

The TwiddleGenF block is the floating-point version of the fixed-point TwiddleGenC block. The TwiddleGenF block generates the appropriate complex coefficients that multiply the streaming data in a radix-2² streaming FFT or IFFT architecture.

Parameters

Table 14-22 shows the parameters for the TwiddleGenF block.

Table 14-22. Parameters for the TwiddleGenC Block

Parameter	Description
FFT type	Specifies whether to generate twiddle factors for an FFT or an IFFT.
Twiddle type	Specifies the floating-point type used for the twiddle factors.

Port Interface

Table 14-23 shows the port interface for the TwiddleGenF block.

Table 14-23. Port Interface for the TwiddleGenF Block

Signal	Direction	Type	Description
counter	Input	Unsigned integer	Counter signal.
w	Output	Complex floating-point type	Complex data output.

Fully-Parallel FFTs (FFT2P, FFT4P, FFT8P, FFT16P, FFT32P, and FFT64P)

The FFT2P, FFT4P, FFT8P, FFT16P, FFT32P, and FFT64P blocks implement fully-parallel FFTs for 2, 4, 8, 16, 32, and 64 points respectively.

The blocks expect bit-reversed input and produce natural-order output.

Parameters

Table 14-24 shows the parameters for the FFT2P blocks. Not all parameters are available with all blocks.

Table 14-24. Parameters for the FFT2P Blocks

Parameter	Description
Twiddle/pruning specification("Pruning and Twiddle Specification for FFTs" on page 10-35.
Use faithful rounding	True if the block uses faithful (rather than correct) rounding for floating-point operations. Fixed-point FFTs ignore this parameter.

Ports

Table 14-25 shows the port interface for the FFT2X blocks.

Table 14-25. Port Interface for the FFT2P Blocks

Signal	Direction	Type	Description
d	Input	Any complex.	Complex data input signal.
q	Output	Determined by pruning specification.	Complex data output signal.

Fully-Parallel FFTs with Flexible Ordering (FFT2X, FFT4X, FFT8X, FFT16X, FFT32X, and FFT64X)

The FFT2X, FFT4X, FFT8X, FFT16X, FFT32X, and FFT64X blocks implement fully-parallel FFTs (or iFFTs) for 2, 4, 8, 16, 32, and 64 points respectively.

Unlike the corresponding P blocks (FFT2P, FFT4P, etc), they implement both FFTs and iFFTs and offer flexible ordering of the input and output wires.

Each block can also be internally parallelized to process several FFTs at once. For example, if there are 16 wires, each FFT8P block can calculate two 8-point FFTs (by specifying the number of spatial bits to be 4). With 32 wires, the same block can calculate four 8-point FFTs (by specifying the number of spatial bits to be 5).

Parameters

Table 14-26 shows the parameters for the FFT2X blocks. Not all parameters are available with all blocks.

Table 14-26. Parameters for the FFT2X Blocks

Parameter	Description
iFFT	true to implement an IFFT, otherwise false.
Number of spatial bits	M for 2^M wires.
Bit-reversed input	true if you expect bit-reversed input, otherwise false.
Bit-reversed output	true if you want bit-reversed output, otherwise false.
Twiddle/pruning specification(“Pruning and Twiddle Specification for FFTs” on page 10-35.
Use faithful rounding	true if the block uses faithful (rather than correct) rounding for floating-point operations. Fixed-point FFTs ignore this parameter.

Ports

Table 14-27 shows the port interface for the FFT2X blocks.

Table 14-27. Port Interface for the FFT2X Blocks

Signal	Direction	Type	Description
v	Input	Boolean.	Valid input signal.
d	Input	Any complex.	Complex data input signal.

Table 14–27. Port Interface for the FFT2X Blocks

Signal	Direction	Type	Description
qv	Output	Boolean.	Valid output signal.
q	Output	Determined by pruning specification.	Complex data output signal.

General Multitwiddle and General Twiddle (GeneralMultiTwiddle, GeneralTwiddle)

Use the GeneralTwiddle and GeneralMultiTwiddle blocks to construct supersampled FFTs. The blocks have the same external interface but use different internal implementations.

The GeneralTwiddle block generates its twiddle factors using the TwiddleRom block; the GeneralMultiTwiddle block uses the TwiddleMultiRom block. The GeneralMultiTwiddle uses approximately twice as many DSP blocks as the GeneralTwiddle block, but (for large FFTs) uses far fewer memory blocks.

Each data sample in the input stream has a unique address. The address consists of the timeslot in which it arrived *tbits* concatenated with the number of wires on which it arrived *sbits*. The *sbits* forms the least significant part of the address; the *tbits* forms the most significant part.

Each data sample is multiplied by a twiddle factor. For an FFT, the twiddle factor is:

$$\text{twiddle} = \exp(-2\pi i \cdot \text{angle} / K)$$

For an IFFT, the twiddle factor is:

$$\text{twiddle} = \exp(2\pi i \cdot \text{angle} / K)$$

where *K* items exist in each block of data.

For each data sample, the twiddle angle is calculated as:

$$\text{angle} = X \cdot Y$$

where *X* and *Y* depend on the position of that data sample in the input stream.

Obtain the value of *X* (or *Y*) by extracting user-specified bits from the address of the data sample, and concatenating them.

Parameters

Table 14–28 shows the parameters for the GeneralTwiddle block.

Table 14–28. Parameters for the GeneralTwiddle Block

Parameter	Description
iFFT	Generate twiddle factors for an FFT or an IFFT.
sbits	The number of spatial address bits i.e. $\log_2(N)$ where there are <i>N</i> wires.
xbits	The vector of bit positions for <i>X</i> .
ybits	The vector of bit positions for <i>Y</i> .
Input type	The type of the input before the twiddle.

Table 14–28. Parameters for the GeneralTwiddle Block

Parameter	Description
Twiddle type	The type of the twiddle factor.
Output type	The type of the output after the twiddle.
Use faithful rounding	Use faithful rather than correct rounding. Only for floating-point twiddle types.

Ports

Table 14–29 shows the port interface for the GeneralTwiddle block.

Table 14–29. Port Interface for the GeneralTwiddle Block

Signal	Direction	Type	Description
v	Input	Boolean.	Input valid signal.
d	Input	Compatible with user-specified input type.	Vector of N data inputs.
qv	Output	Boolean.	Output valid signal.
q	Output	User-specified output type.	Vector of N data outputs.

Hybrid FFT (Hybrid_FFT)

The Hybrid_FFT block implements a hybrid serial or parallel implementation of a supersampled FFT (or IFFT) that processes 2^M points per cycle (with $0 < M$).

The hybrid implementation consists of an optional serial section (built using single-wire streaming FFTs) associated twiddle block, and a parallel section (implemented using the PFFT_Pipe block).

You control the length of the serial section by a user-supplied parameter. For an FFT with 2^N points that processes 2^M points per cycle, this parameter must be no greater than $N-M$.

In general, the serial section is more space-efficient; the parallel section is more multiplier-efficient. So changing the value of this parameter provides a trade-off between DSP usage and memory usage.

Parameters

Table 14–30 shows the parameters for the Hybrid_FFT block.

Table 14–30. Parameters for the Hybrid_FFT Block

Parameter	Description
iFFT	true to implement an IFFT, otherwise false.
N	Log2 of the number of points in the FFT.
Bit-reversed input	true if you expect bit-reversed input, otherwise false.
M	Log2 of the number of input wires.

Table 14–30. Parameters for the Hybrid_FFT Block

Parameter	Description
Number of serial stages	Length of the serial section (in radix-2 stages).
Twiddle/pruning specification	“Pruning and Twiddle Specification for FFTs” on page 10–35.
Optimize twiddle memory usage	true to use GeneralMultTwiddle (rather than GeneralTwiddle) for top-level twiddle.
Use faithful rounding	true if the block uses faithful (rather than correct) rounding for floating-point operations. Fixed-point FFTs ignore this parameter.

Port Interface

[Table 14–31](#) shows the port interface for the Hybrid_FFT block.

Table 14–31. Port Interface for the Hybrid_FFT Block

Signal	Direction	Type	Description
v	Input	Boolean.	Valid input signal.
d	Input	As specified.	Complex data input signal.
qv	Output	Boolean.	Valid output signal.
q	Output	Determined by pruning specification.	Complex data output signal.

Multiwire Transpose (MultiwireTranspose)

The MultiwireTranspose block performs a specialized reordering of a block of data and presents it on multiple wires.

Parameters

[Table 14–32](#) shows the parameters for the MultiwireTranspose block.

Table 14–32. Parameters for the MultiwireTranspose Block

Parameter	Description
Address permutation	A vector of integers that describes how to rearrange the block of data.
N	The number of spatial address bits. The block has 2^N data wires.

Port Interface

[Table 14–33](#) shows the port interface for the MultiwireTranspose block.

Table 14–33. Port Interface for the MultiwireTranspose Block

Signal	Direction	Type	Description
v	Input	Boolean.	Input valid signal.
d	Input	Any type.	Data input.

Table 14–33. Port Interface for the MultiwireTranspose Block

Signal	Direction	Type	Description
qv	Output	Boolean.	Output valid signal.
q	Output	Same as d	Data output.

Parallel Pipelined FFT (PFFT_Pipe)

The PFFT_Pipe block implements a supersampled FFT (or IFFT) that processes 2^M points per cycle (with $0 < M$).

The PFFT_Pipe block uses a pipeline of (small) fully-parallel FFTs, twiddle, and transpose blocks. This FFT uses only a small number of DSP blocks but has a relative high latency (and associated memory usage).

Parameters

Table 14–26 shows the parameters for the PFFT_Pipe blocks. Not all parameters are available with all blocks.

Table 14–34. Parameters for the PFFT_Pipe Blocks

Parameter	Description
iFFT	true to implement an IFFT otherwise false.
N	Log2 of the number of points in the FFT.
Bit-reversed input	true if you expect bit-reversed input, otherwise false. .
Number of spatial bits	M for 2^M wires.
Twiddle/pruning specification(“Pruning and Twiddle Specification for FFTs” on page 10–35.
Use faithful rounding	true if the block uses faithful (rather than correct) rounding for floating-point operations. Fixed-point FFTs ignore this parameter.

Port Interface

Table 14–35 shows the port interface for the PFFT_Pipe blocks.

Table 14–35. Port Interface for the PFFT_Pipe Blocks

Signal	Direction	Type	Description
v	Input	Boolean.	Valid input signal.
d	Input	Any.	Complex data input signal.
qv	Output	Boolean.	Valid output signal.
q	Output	Determined by pruning specification.	Complex data output signal.

Pulse Divider (PulseDivider)

The PulseDivider block generates a single-cycle one on its output for each 2^N ones on its input.

Parameters

Table 14-36 shows the parameters for the PulseDivider block.

Table 14-36. Parameters for the PulseDivider Block

Parameter	Description
N	Specifies the input block size 2^N .

Port Interface

Table 14-37 shows the port interface for the PulseDivider block.

Table 14-37. Port Interface for the PulseDivider Block

Signal	Direction	Type	Description
v	Input	Boolean or <code>uint(1)</code> .	Data valid.
g	Output	<code>uint(1)</code> .	Block containing 2^N elements received.

Pulse Multiplier (PulseMultiplier)

The PulseMultiplier block stretches a single-cycle pulse on its input into a 2^N -cycle pulse on its output. The block ignores any input pulse that arrives within 2^N cycles of the previous one.

The PulseMultiplier is a special version of the StretchPulse block.

Parameters

Table 14-38 shows the parameters for the PulseMultiplier block.

Table 14-38. Parameters for the PulseMultiplier Block

Parameter	Description
N	Specifies the output length pulse size 2^N .

Port Interface

Table 14-39 shows the port interface for the PulseMultiplier block.

Table 14-39. Port Interface for the PulseMultiplier Block

Signal	Direction	Type	Description
g	Input	Boolean or <code>uint(1)</code> .	Start of 2^N data block.
v	Output	<code>uint(1)</code> .	Data valid.

Single-Wire Transpose (Transpose)

The Transpose block performs a specialized reordering of a block of data.

Parameters

Table 14-32 shows the parameters for the Transpose block.

Table 14-40. Parameters for the Transpose Block

Parameter	Description
Address permutation	A vector of integers that describes how to rearrange the block of data.

Port Interface

Table 14-33 shows the port interface for the Transpose block.

Table 14-41. Port Interface for the Transpose Block

Signal	Direction	Type	Description
v	Input	Boolean.	Input valid signal.
d	Input	Any type.	Data input.
qv	Output	Boolean.	Output valid signal.
q	Output	Same as d	Data output.
g	Output	Boolean,	Start of output block.

Split Scalar (SplitScalar)

The SplitScalar block splits its input (typically an unsigned integer) into a vector of Booleans. The least significant bit of the scalar becomes the first entry in the vector (little-endian ordering).

FFT implementations often contain various bit-twiddling operations as part of their control structure. Use the SplitScalar block to make these bit-twiddling operations easier to implement.

Parameters

Table 14-42 shows the parameters for the SplitScalar block.

Table 14-42. Parameters for the SplitScalar Block

Parameter	Description
Width	Width of the scalar (in bits), which is also the width of the output vector.

Port Interface

Table 14-43 shows the port interface for the SplitScalar block.

Table 14-43. Port Interface for the SplitScalar Block

Signal	Direction	Type	Description
d	Input	Any scalar.	Data input.
q	Output	Boolean vector.	Data output.

Streaming FFTs (FFT2, FFT4, VFFT2, and VFFT4)

The FFT2, FFT4, VFFT2, and VFFT4 blocks are low-level blocks that implement streaming FFTs.

Parameters

Table 14-44 shows the parameters for the FFT2, FFT4, VFFT2, and VFFT4 blocks.

Table 14-44. Parameters for the FFT2, FFT4, VFFT2, and VFFT4 Block

Parameter	Description
iFFT	true for an iFFT, otherwise false. FFT4 and VFFT4 only.
Bit reversed input	true for bit-reversed inputs. FFT4 and VFFT4 only.
Stages before this	The number of stages to the left of this FFT.
Stages after this	The number of stages to the right of this FFT.
Input type	The type of the input signal. For example: fixdt(1,16,15).
Use faithful rounding	true to use faithful rather than correct rounding. Fixed-point FFTs ignore this parameter.

Ports

Table 14-45 shows the port interface for the FFT2, FFT4, VFFT2, and VFFT4 blocks.

Table 14-45. Port Interface for the FFT2, FFT4, VFFT2, and VFFT4 Blocks

Signal	Direction	Type	Description
v	Input	Boolean	Valid input signal.
d	Input	Any complex type	Complex data input signal.
drop	Input	uint(k) for some k	Total number of FFT stages to bypass.
qv	Output	Boolean	Valid output signal.
q	Output	Any complex type	Complex data output signal.
qdrop	Output	uint(k) for some k	Total number of FFT stages to bypass.

Twiddle Angle (TwiddleAngle)

The TwiddleAngle block generates FFT twiddle factors when you use it between a counter and the TwiddleRom (or TwiddleRomF) blocks.

The TwiddleAngle block takes the output of the counter and splits it into three parts:

- The channel field (LSBs of the counter)
- The index field
- The pivot field (MSBs of the counter)

It provides $\text{bitreverse}(\text{pivot}) * \text{index}$ at the output. The calculation is optimized to use no multipliers and only a small amount of logic.

The TwiddleAngle block has an additional input: v, which keeps the internal state of the TwiddleAngle block synchronized with the counter. This input should be identical to the enable input to the counter.

Parameters

Table 14-46 shows the parameters for the TwiddleAngle block.

Table 14-46. Parameters for the TwiddleAngle Block

Parameter	Description
K	Width of the channel field.
Pivot width	Width of the pivot field.
Index width	Width of the index field.

Port Interface

Table 14-47 shows the port interface for the TwiddleAngle block.

Table 14-47. Port Interface for the TwiddleAngle Block

Signal	Direction	Type	Description
v	Input	Boolean	Input to upcounter.
c	Input	Unsigned integer	Output of upcounter.
angle	Output	Unsigned integer	Input to TwiddleRom.

Twiddle Generator (TwiddleGenC) Deprecated

The TwiddleGenC block generates the appropriate complex coefficients that multiplies the streaming data in a radix-2² streaming FFT or iFFT architecture.

Feed at the input by a modulo N counter (where N is an integer power of two) and the appropriate complex sequence generates at the output.

To parameterize this block, set the **Counter bit width** parameter with $\log_2(N)$ and enter the width in bits and fixed-point scaling of the twiddle factors. A cosine or sine wave has a range of [-1:1], therefore you must provide at least two integer bits, and as many fractional bits as are appropriate. Starting with a twiddle bit width of 16 bits (enter 16 as the **twiddle bit width**), and a scaling of 2^{-14} (enter 14 as the **Twiddle scaling exponent**). The resulting fixed-point type is `sfixed16_en14` (2.14 fixed-point format).

Parameters

Table 14-48 shows the parameters for the TwiddleGenC block.

Table 14-48. Parameters for the TwiddleGenC Block

Parameter	Description
FFT type	Specifies whether to generate twiddle factors for an FFT or an IFFT.
Counter bit width	Specifies the counter width in bits.
Twiddle bit width	Specifies the twiddle width in bits.
Twiddle scaling exponent value	Specifies the fixed-point scaling factor of the complex twiddle factor.

Port Interface

Table 14-49 shows the port interface for the TwiddleGenC block.

Table 14-49. Port Interface for the TwiddleGenC Block

Signal	Direction	Type	Description
counter	Input	Any fixed-point type	Counter signal.
w	Output	Derived complex fixed-point type	Complex data output.

Twiddle and Variable Twiddle (Twiddle and VTwiddle)

The Twiddle and VTwiddle blocks are low-level blocks that implement streaming FFTs.

Each twiddle block joins two FFTs (the left constituent FFT and the right constituent FFT) to form a larger FFT. For variable-size FFTs, use the VTwiddle block. Each of the constituent FFTs is either a primitive FFT (e.g. an FFT4 block) or is multiple FFT and twiddle blocks.

The Twiddle block FFT may be part of an even larger FFT. In fact, the pipeline is formed by linearizing a binary tree of FFTs (leaf nodes) and twiddle blocks (internal nodes).

Each Twiddle block requires you to specify three types:

- The data signal prior to the twiddle multiplications
- The twiddle factors
- The data signal after the twiddle multiplication.

Parameters

Table 14-50 shows the parameters for the Twiddle and VTwiddle blocks.

Table 14-50. Parameters for the Twiddle and VTwiddle Block

Parameter	Description
iFFT	Generate twiddle factors for an FFT or an IFFT.
Stages before this	The number of stages to the left(*) of this composite FFT.
Left width	The number of stages in the left(*) constituent FFT.
Right width	The number of stages in the right(*) constituent FFT.
Stages after this	The number of stages to the right(*) of this composite FFT.
Input type	The type to which DSP Builder should convert the input. Refers to the type of the input after you apply an explicit type conversion. It doesn't have to exactly match the actual input type to the Twiddle block.
Twiddle type	The type of the twiddle factor.
Output type	The type of the output after the twiddle.
Use faithful rounding	Use faithful rather than correct rounding. Fixed-point FFTs ignore this parameter.



For bit-reversed FFTs, reverse left and right, so left refers to the number of stages to the right of the current block.

Ports

Table 14–51 shows the port interface for the `Twiddle` and `VTwiddle` blocks.

Table 14–51. Port Interface for the Twiddle and VTwiddle Blocks

Signal	Direction	Type	Description
v	Input	Boolean	Valid input signal.
d	Input	Any complex type	Complex data input signal.
drop	Input	uint(k) for some k	Total number of FFT stages to bypass
qv	Output	Boolean	Valid output signal.
q	Output	Any complex type	Complex data output signal.
qdrop	Output	uint(k) for some k	Total number of FFT stages to bypass

Twiddle ROM (`TwiddleRom`, `TwiddleMultRom` and `TwiddleRomF` (deprecated))



`TwiddleRomF` is deprecated; `TwiddleRom` has a new parameters (therefore appears in the obsolete and the common directory).

The `TwiddleRom` and `TwiddleMultRom` block construct FFTs. They map an angle (specified as an unsigned integer) to a complex number (the twiddle factor). For an FFT, the mapping is:

$$\text{twiddle} = \exp(-2\pi i \cdot \text{angle} / N)$$

For an IFFT, the mapping is:

$$\text{twiddle} = \exp(2\pi i \cdot \text{angle} / N)$$

where $N = 2^{\text{anglewidth}}$ and *anglewidth* is the width of the *angle* input signal.

The `TwiddleRom` and `TwiddleMultRom` blocks have the same external interface but different internal implementations. `TwiddleRom` uses a single large memory; `TwiddleMultRom` uses two smaller memories and constructs the twiddle factors using complex multiplication.

`TwiddleMultRom` consumes more DSP blocks but generally uses fewer memory blocks than `TwiddleRom`. `TwiddleMultRom` also produces slightly less accurate results than `TwiddleRom`.

Parameters

Table 14-52 shows the parameters for the TwiddleRom and TwiddleMultRom blocks.

Table 14-52. Parameters for the TwiddleRom and TwiddleMultRom Blocks

Parameter	Description
iFFT	True to generate twiddle factors for an IFFT.
Angle bit width	The width of the angle input signal in bits.
Twiddle type	The type of the twiddle output. For example: fixdt(1,18,17).

Port Interface

Table 14-53 shows the port interface for the TwiddleRom and TwiddleMultRom blocks.

Table 14-53. Port Interface for the TwiddleROM and TwiddleMultRom Blocks

Signal	Direction	Type	Description
angle	Input	Unsigned	Input angle.
twiddle	Output	User specified	Output twiddle factor.

This ModelIP library contains several decimating and interpolating cascaded integrator comb (CIC), and finite impulse response (FIR) filters including single rate multirate and fractional rate FIR filters.

Multirate filters are essential to the up and down conversion tasks that modern radio systems require. Cost effective solutions to many other DSP applications also use multirate filters to reduce the multiplier count.

A memory-mapped interface allows you to read and write coefficients directly, easing system integration.

The Filter library contains the following blocks:




- Decimating CIC
- Decimating FIR
- Fractional Rate FIR
- Interpolating CIC
- Interpolating FIR
- Single-Rate FIR

FIR and CIC Filters

This chapter describes blocks that implement several finite impulse response (FIR) and cascaded integrator comb (CIC) filters. The filters share many common features and use advanced high-level synthesis techniques to generate filters with higher clock speeds, lower logic, multiplier, and memory counts. Using these high clock rates allows you to reduce your costs by choosing smaller FPGAs.

Common Features

- Filter length from 1 to unlimited taps
- Data input width from 2 to 32 bits
- Data output width from 4 to 64 bits
- Multichannel (up to 1024 channels)
- Powerful MATLAB integration
- Simulink fixed-point integration
- Automatic pipelining
- Plug and play connectivity
- Simplified timing closure
- Generates updated help for your parameters

-  Each channel is an independent data source. In an IF Modem design, two channels are required for the complex pair from each antenna.
-  This library does not support complex data.
-  All input data and coefficients must be fixed-point data.

Updated Help

After you run a simulation, DSP Builder updates the help pages with specific information about each instance of a block. This updated help overrides the default help link. To find the updated help click on the help link on the block after simulation. This updated help includes a link back to the help for the general block and the following information about the generated FIR instance:

- Date and time of generation
- The version number and revision for the FIR
- Number of physical input and output data buses
- Bit width of data output.
- Number of different phases
- Implementation folding. The number of times that the design uses each multiplier per sample to reduce the implementation size.
- Filter utilization. For some sample rates, stall the filter internally for several cycles. The number of cycles for active calculation shows with the number of cycles that the sample rate relative to the system clock frequency determines.
- Tap utilization. When some filters are folded, there may be extra unused taps. The extra taps increase the filter length with no hardware resource increase.
- Latency. The depth of pipelining added to the block to meet the target clock frequency on the chosen target device.
- Parameters table that lists the system clock, clock margin, and all FIR input parameters.
- Port interface table.
- Input and output data format. An ASCII rendering of the input and output channelized data ordering.

The updated help includes the following information about the CIC instance:

- Date and time of generation
- The version number and revision for the CIC
- Number of integrators. Depending on the input data rate and interpolation factor the number of integrator stages DSP Builder needs to process the data may be more than 1. In these instances, the integrator sections of the filter duplicate (vectorize) to satisfy the data rate requirement.
- Calculated output bit width. The width in bits of the (vectorized) data output from the filter.

- Calculated stage bit widths. Each stage in the filter has precise width in bits requirements— N comb sections followed by N integrator sections.
- The gain through the CIC filter. CIC filters usually have large gains that you must scale back.
- Comb section utilization. In the comb section, the data rate is lower, so that you can perform more resource sharing. This message indicates the efficiency of the subtractor usage.
- Integrator section utilization. In the integrator section, the data rate is higher, so that you can perform less resource sharing. This message indicates the efficiency of the adder usage.
- The latency that this block introduces.
- Parameters table that lists the decimation rate, number of stages, differential delay, number of channels, clock frequency, and input sample rate parameters.
- Port interface table.
- Input and output data format.

Basic FIR Filter Operation

Equation 15–1 shows the basic convolution operation that a single-rate filter performs.

Equation 15–1.

$$y_k = \sum_{n=0}^{n=1} a_n \times X_{k-n} \quad \text{where } k = 0, 1, 2, \dots, n-1$$

At each sample time k , the new output y , is calculated by multiplying coefficients a , by the recent past values of the input x .

Half-Band and L-Band Nyquist FIR Filters

Some filtering functions can use a half-band filter where nearly half of the coefficients are zero. The half-band support uses these extra zeros to further reduce the number of multipliers, and thereby reduce the filter cost.

The generalized form of these filters is L-band Nyquist filters, in which every L th coefficient is zero counting out from the center tap. DSP Builder also supports these structures and can often reduce the number of multipliers required in a filter.

Automatic Pipelining

The required system clock frequency, and the device family and speed grade determine the maximum logic depth permitted in the output RTL. Functions such as adders are pipelined by splitting them into multiple sections with a registered carry between them. This pipelining decreases the logic depth allowing higher frequency operation.

High Speed Operation

The filter generator is responsive to the system clock frequency, therefore timing closure is much easier to achieve. The generator uses heuristics that ensure the logic is capable of running at the desired system clock frequency on the FPGA. You can help timing closure by adding more clock margin, resulting in additional pipelining that shortens the critical paths.

The FPGA structures such as internal multiplier and memory delays determine the maximum clock frequencies.

Scalability

In some cases, the aggregate sample rate for all channels may be higher than the system clock rate. In these cases, the filter has multiple input or output buses to carry the additional data. This requirement is implemented in the Simulink block by increasing the vector width of the data signals.

Parameterization

The system specification, including such factors as the channel count and sample rates, determines the main parameters for a filter. The remaining parameters such as data widths and system clock rates are inferred from the enclosing Simulink model. Any changes to these parameters ripple through your design, changing the system performance without you having to update all the components. You can express any of the parameters as MATLAB expressions, to rapidly parameterize a whole system.

The hardware generation techniques create efficient filters with combinations of parameters, such as a symmetric 3-band FIR filter with 7 channels and 100 cycles to process a sample from each channel. Hardware generation is fast and can run on-the-fly with every Simulink simulation, so that the edit simulation loop time is much reduced, improving productivity.

Coefficient Generation

You can generate filter coefficients using a MATLAB function that reloads at run time with the memory-mapped interface registers. For example, the Simulink fixed-point object `fi(fir1(49, 0.3), 1, 18, 19)`.

Channelization

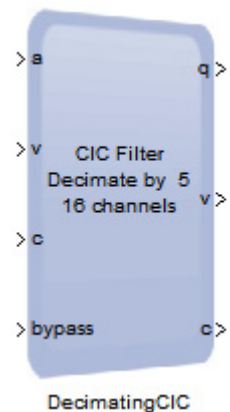
The generated help page for the block shows the input channel data format and output data channel format that a FIR or CIC filter uses, after you have run a Simulink simulation.

Decimating CIC

The DecimatingCIC block implements a highly efficient multichannel cascaded integrator-comb filter across a broad range of parameters directly from a Simulink model.

The DecimatingCIC block performs filtering on a stream of multichannel input data and produces a stream of output data with decreased sampling frequency.

You can use the DecimatingCIC block in a digital down converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full-precision fixed-point type. You can reduce the precision with a separate Scale block, which can perform rounding and saturation to provide the required output precision.



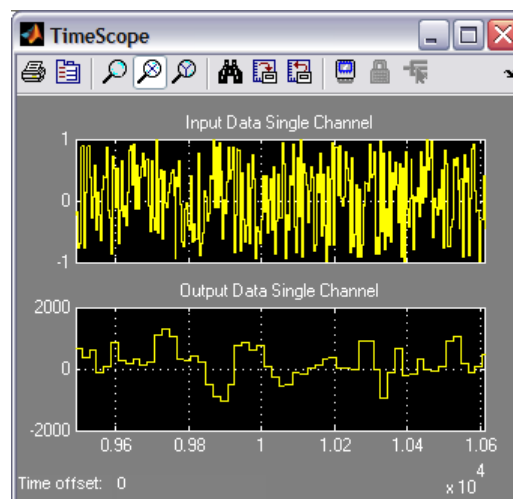
Features

The DecimatingCIC block supports rate changes from two upwards.

Operation

The DecimatingCIC has a lower output sample rate than the input sample rate by a factor D , where D is the decimation factor. Usually, the DecimatingCIC discards $(D-1)$ out of D output samples thus lowering the sample rate by a factor D . The physical implementation avoids performing additions leading to these discarded samples, reducing the filter cost. Figure 15–1 shows how decimating by 5 decreases the sample rate of a random noise input.

Figure 15–1. Decimate by 5 Filter Decreasing Sample Rate of a Random Noise Input



Parameters

Table 15-1 shows the parameters for the DecimatingCIC block.

Table 15-1. Parameters for the DecimatingCIC Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Number of channels	Specifies the number of unique channels to process.
Number of stages	Specifies the number of comb and integrator stages.
Decimation factor	Specifies the decimation factor 1/(integer). (An integer greater than 1 implies interpolation.)
Differential delay	Specifies the differential delay.

Port Interface

Table 15-2 shows the port interface for the DecimatingCIC block.

Table 15-2. Port Interface for the DecimatingCIC Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of the data input signals. If v is high, the data on the a wire is valid.
c	Input	Indicates channel of data input signals. If v is high, c indicates which channel the data corresponds to.
bypass	Input	When this input asserts, the input data is zero-stuffed and scaled by the gain of the filter, which is useful during hardware debugging.
q	Output	The data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾

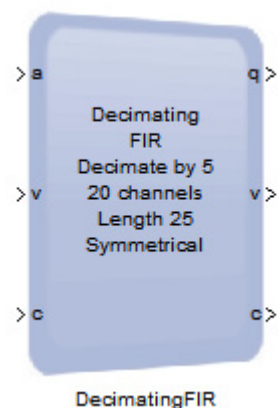
Note to Table 15-2:

(1) The output data can be non-zero when v is low.

Decimating FIR

The DecimatingFIR block implements a highly efficient multichannel finite impulse response filter across a broad range of parameters directly from a Simulink model. A memory-mapped interface allows you to read and write coefficients directly, easing system integration.

The Decimating FIR block performs filtering on a stream of multichannel input data and produces a stream of output data with increased sampling frequency.



Use the Decimating FIR block in a digital down converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type. You can reduce the precision by using a separate Scale block, which can perform rounding and saturation to provide the required output precision.

Features

The Decimating FIR block supports rate changes from two upwards, coefficient width in bits from 2 to 32 bits, half-band and L-band Nyquist filters, real and complex filters, symmetry and anti(negative)-symmetry.

Operation

Equation 15-1 on page 15-3 shows the basic convolution operation that a single-rate filter performs.

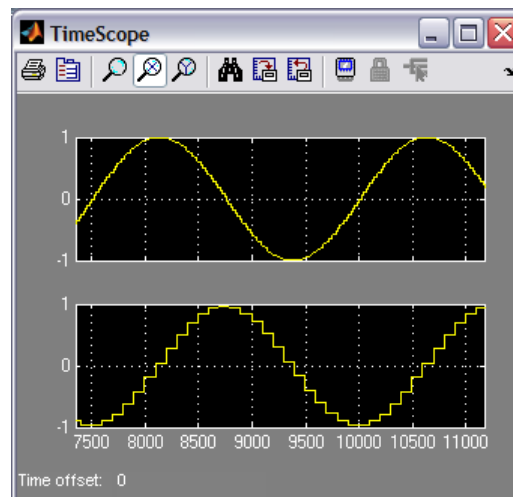
At each sample time k , the new output y , is calculated by multiplying coefficients a , by the recent past values of the input x .

The Decimating FIR has a lower output sample rate than the input sample rate by a factor, D , the decimation factor. The decimating FIR discards $D-1$ out of D output samples, thus lowering the sample rate by a factor D .

The physical implementation avoids performing multiplications with these zero samples, reducing the filter cost.

Figure 15-2 on page 15-7 shows how decimating by 5 decreases the sample rate of a sine wave input.

Figure 15-2. Decimating by 5 Filter Decreasing Sample Rate of a Sine Wave Input



Parameters

Table 15-3 shows the parameters for the DecimatingFIR block.

Table 15-3. Parameters for the DecimatingFIR Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Decimation	Specifies the decimation rate. Must be an integer.
Number of channels	Specifies the number of unique channels to process.
Symmetry	You can select Symmetrical or Anti-Symmetrical coefficients. Symmetrical coefficients can result in hardware resource savings over the unsymmetrical version.
Coefficients	You can specify the filter coefficients using a Simulink fixed-point object <code>fi(0)</code> . The data type of the fixed-point object determines the width and format of the coefficients. The length of the array determines the length of the filter. For example, <code>fi(fir1(49, 0.3), 1, 18, 19)</code>
Base address	You can memory map the filter's coefficients into the address space of the system. This field determines the starting address for the coefficients. It is specified as a MATLAB <code>double</code> type (decimal integer) but you can use a MATLAB expression to specify a hexadecimal or octal type if required.
Read/Write mode	You can map the coefficients as Read , Write , Read/Write , or Constant . This field determines the type of address decode to build.
Filter structure	You can select Use All Taps , Half Band , or other specified band (from 3rd Band to 46th Band).
Expose Avalon-MM Slave in Simulink	Allows you to reconfigure coefficients without Qsys. Also, it also allows you to reprogram multiple FIR filters simultaneously. When you turn on this parameter, the following three additional input ports and two output ports appear in Simulink. <ul style="list-style-type: none"> ■ data, address, write ■ readdata, valid

Filter Coefficients

You can change filter coefficients at run time by writing to the memory-mapped interface. You do not need to create custom logic to deal with awkward update schemes.

You can set the base address of the memory-mapped coefficients with the **Base address** parameter and set the filter coefficients by entering a Simulink fixed-point array into the **Coefficients** parameter. You can generate a vector of coefficients either by entering an array of numbers, or using one of the many MATLAB functions to build the required coefficients.



For more information about Simulink fixed-point objects and MATLAB functions, refer to the MATLAB Help.

Port Interface

Table 15-4 shows the port interface for the DecimatingFIR block.

Table 15-4. Port Interface for the DecimatingFIR Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of the data input signals. If v is high, the data on the a wire is valid.
c	Input	Indicates channel of data input signals. If v is high, then c indicates which channel the data corresponds to.
data	Input	The data port has the same type as the coefficients.
address	Input	Only available when you turn on Expose Avalon-MM Slave in Simulink . The address port is the same width as the system address width that you configure in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu. Also the base address is the same.
write	Input	Deassert the write port to make a read occur.
q	Output	The fixed-point filtered data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾
readdata	Output	The data port has the same type as the coefficients.
valid	Output	Indicates a valid output.

Note to Table 15-4:

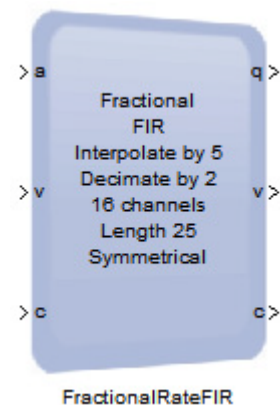
(1) The output data can be non-zero when v is low.

Fractional Rate FIR

The FractionalRateFIR block implements a highly efficient multichannel finite impulse response filter across a broad range of parameters directly from a Simulink model. A memory-mapped interface allows you to read and write coefficients directly, easing system integration.

The FractionalRateFIR block performs filtering on a stream of multichannel input data and produces a stream of output data with increased sampling frequency.

You can use the FractionalRateFIR block in a digital down converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type. You can reduce the precision by using a separate Scale block, which can perform rounding and saturation to provide the required output precision.



Features

The FractionalRateFIR block supports:

- Interpolation rate changes and decimation rate changes from two upwards
- Rational fractional rate changes
- Coefficient width in bits from 2 to 32 bits
- Half-band and L-band Nyquist filters
- Symmetry and anti(negative)-symmetry.

Operation

Equation 15-1 on page 15-3 shows the basic convolution operation that a single-rate filter performs.

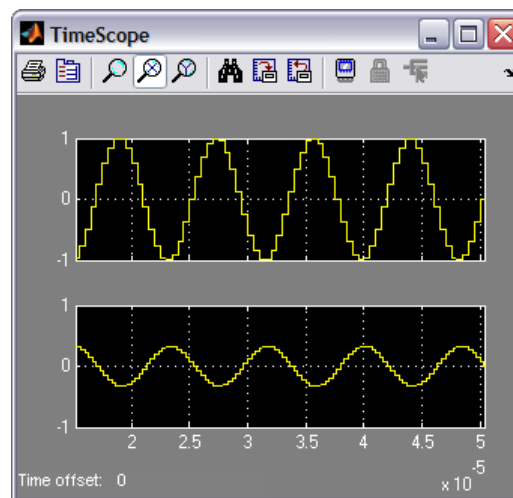
At each sample time, k , the new output y , is calculated by multiplying coefficients a , by the recent past values of the input x .

The FractionalRateFIR has a modified output sample rate that differs from the input sample rate by a factor, I/D , where I is the interpolation rate and D is the decimation factor. Usually, the fractional rate interpolates by a factor I by inserting $(I-1)$ zeros before performing the filter operation. Then the FIR discards $D-1$ out of D output samples, thus lowering the sample rate by a factor D .

The filtering is performed as in Equation 15-1. The physical implementation avoids performing multiplications with these zero samples, reducing the filter cost.

Figure 15-3 on page 15-10 shows how interpolating by 3 and decimating by 2 changes the sample rate of a sine wave input.

Figure 15-3. Sample Rate of a Sine Wave Input Interpolated by 3 and Decimated by 2



Parameters

Table 15-5 shows the parameters for the FractionalRateFIR block.

Table 15-5. Parameters for the FractionalRateFIR Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Interpolation	Specifies the interpolation rate. Must be an integer.
Decimation	Specifies the decimation rate. Must be an integer.
Number of channels	Specifies the number of unique channels to process.
Symmetry	You can select Symmetrical or Anti-Symmetrical coefficients. Symmetrical coefficients can result in hardware resource savings over the unsymmetrical version.
Coefficients	You can specify the filter coefficients using a Simulink fixed-point object <code>fi(0)</code> . The data type of the fixed-point object determines the width and format of the coefficients. The length of the array determines the length of the filter. For example, <code>fi(fir1(49, 0.3),1,18,19)</code> .
Base address	You can memory map the filter's coefficients into the address space of the system. This field determines the starting address for the coefficients. It is specified as a MATLAB <code>double</code> type (decimal integer) but you can use a MATLAB expression to specify a hexadecimal or octal type if required.
Read/Write mode	You can map the coefficients as Read , Write , Read/Write , or Constant . This field determines the type of address decode to build.
Filter structure	You can select Use All Taps , Half Band , or a specified band (from 3rd Band to 46th Band).
Expose Avalon-MM Slave in Simulink	Allows you to reconfigure coefficients without Qsys. Also, it also allows you to reprogram multiple FIR filters simultaneously. When you turn on this parameter, the following three additional input ports and two output ports appear in Simulink. <ul style="list-style-type: none"> ■ data, address, write ■ readdata, valid

Filter Coefficients

You can change filter coefficients by writing to the memory-mapped coefficients. You do not need to create custom logic to deal with awkward update schemes.

You can set the base address of the memory-mapped coefficients with the **Base address** parameter and set the filter coefficients by entering a Simulink fixed-point array into the **Coefficients** parameter. You can generate a vector of coefficients either by entering an array of numbers, or by using one of the many MATLAB functions to build the required coefficients.



For more information about Simulink fixed-point objects and MATLAB functions, refer to the MATLAB Help.

Port Interface

Table 15-6 shows the port interface for the FractionalRateFIR block.

Table 15-6. Port Interface for the FractionalRateFIR Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of the data input signals. If v is high, the data on the a wire is valid.
c	Input	Indicates channel of data input signals. If v is high, then c indicates which channel the data corresponds to.
data	Input	The data port has the same type as the coefficients.
address	Input	Only available when you turn on Expose Avalon-MM Slave in Simulink . The address port is the same width as the system address width that you configure in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu. Also the base address is the same.
write	Input	Deassert the write port to make a read occur.
q	Output	The fixed-point filtered data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾
readdata	Output	The data port has the same type as the coefficients.
valid	Output	Indicates a valid output.

Note to Table 15-6:

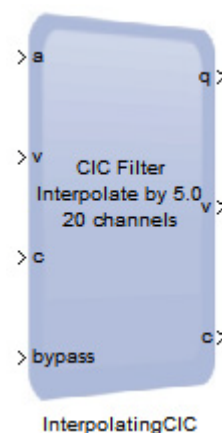
(1) The output data can be non-zero when v is low.

Interpolating CIC

The InterpolatingCIC block implements a highly efficient multichannel cascaded integrator-comb filter across a broad range of parameters directly from a Simulink model.

The InterpolatingCIC block performs filtering on a stream of multichannel input data and produces a stream of output data with increased sampling frequency.

You can use the InterpolatingCIC block in a digital up converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type. You can reduce the precision by using a separate Scale block, which can perform rounding and saturation to provide the required output precision.



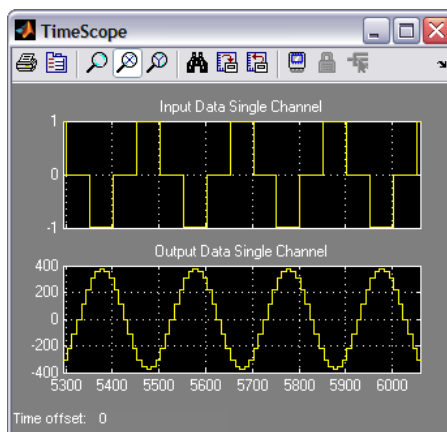
Features

The InterpolatingCIC block supports rate changes from two upwards.

Operation

The InterpolatingCIC has a higher output sample rate than the input sample rate by a factor I , where I is the interpolation rate. Usually, the InterpolatingCIC inserts $(I-1)$ zeros for every input sample, thus raising the sample rate by a factor I . Figure 15-4 shows how interpolating by 5 increases the sample rate of a sine wave input.

Figure 15-4. Interpolate by 5 Filter Increasing Sample Rate of a Sine Wave Input



Parameters

Table 15-7 shows the parameters for the InterpolatingCIC block.

Table 15-7. Parameters for the InterpolatingCIC Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Number of channels	Specifies the number of unique channels to process.
Number of stages	Specifies the number of comb and integrator stages.
Interpolation factor	Specifies the interpolation factor. Must be an integer.
Differential delay	Specifies the differential delay.
Final decimation	You can optionally specify a final decimation by 2 to allow interpolation rates which are multiples of 0.5. The decimation works by simply throwing away data value. Only use this option to reduce the number of unique outputs the CIC generates.

Port Interface

Table 15-8 shows the port interface for the InterpolatingCIC block.

Table 15-8. Port Interface for the InterpolatingCIC Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of the data input signals. If v is high, the data on the a wire is valid.

Table 15–8. Port Interface for the InterpolatingCIC Block

Signal	Direction	Description
c	Input	Indicates channel of data input signals. If v is high, then c indicates which channel the data corresponds to.
bypass	Input	When this input is asserted, the input data is zero-stuffed and scaled by the gain of the filter. This option can be useful during hardware debug.
q	Output	The fixed-point filtered data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾

Note to Table 15–8:

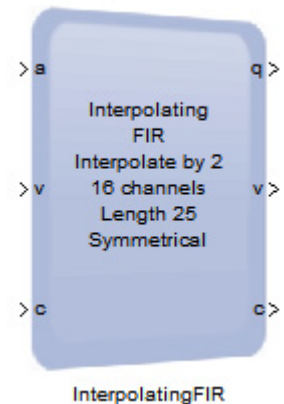
(1) The output data can be non-zero when v is low.

Interpolating FIR

The InterpolatingFIR block implements a highly efficient multichannel finite impulse response filter across a broad range of parameters directly from a Simulink model. A memory-mapped interface allows you to read and write coefficients directly, easing system integration.

The InterpolatingFIR block performs filtering on a stream of multichannel input data and produces a stream of output data with increased sampling frequency.

You can use the InterpolatingFIR block in a digital up converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type. You can reduce the precision by using a separate Scale block, which can perform rounding and saturation to provide the required output precision.



Features

The InterpolatingFIR block supports:

- Rate changes from two upwards
- Coefficient width in bits from 2 to 32 bits
- Data output width in bits from 4 to 64 bits
- Half-band and L-band Nyquist filters
- Symmetry and anti(negative)-symmetry
- Real filters

Operation

Equation 15–1 on page 15–3 shows the basic convolution operation that a single-rate filter performs.

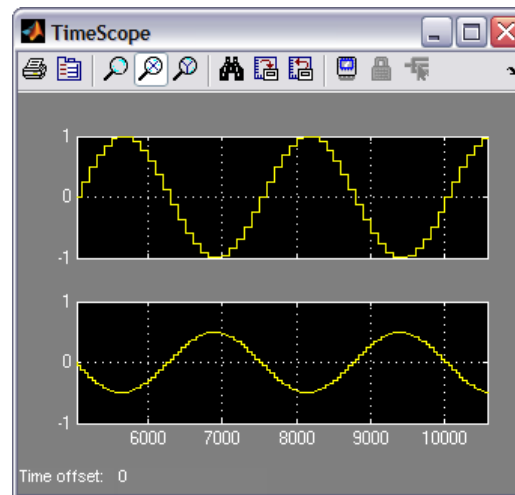
At each sample time k , the new output y , is calculated by multiplying coefficients a , by the recent past values of the input x .

The InterpolatingFIR has a higher output sample rate than the input sample rate by a factor, I , the interpolation factor. Usually, the interpolating FIR inserts $I-1$ zeroes for every input sample, thus raising the sample rate by a factor I .

The filtering is performed as in Equation 15-1. The physical implementation avoids performing multiplications with these zero samples, reducing the filter cost.

Figure 15-5 shows how interpolating by 2 increases the sample rate of a sine wave input.

Figure 15-5. Interpolate by 2 Filter Increasing Sample Rate of a Sine Wave Input



Parameters

Table 15-9 shows the parameters for the InterpolatingFIR block.

Table 15-9. Parameters for the InterpolatingFIR Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Interpolation	Specifies the interpolation rate. Must be an integer.
Number of channels	Specifies the number of unique channels to process.
Symmetry	You can select Symmetrical or Anti-Symmetrical coefficients. Symmetrical coefficients can result in hardware resource savings over the unsymmetrical version.
Coefficients	You can specify the filter coefficients using a Simulink fixed-point object <code>fi(0)</code> . The data type of the fixed-point object determines the width and format of the coefficients. The length of the array determines the length of the filter. For example, <code>fi(fir1(49, 0.3), 1, 18, 19)</code> .
Base address	You can memory map the filter's coefficients into the address space of the system. This field determines the starting address for the coefficients. It is specified as a MATLAB <code>double</code> type (decimal integer) but you can use a MATLAB expression to specify a hexadecimal or octal type if required.

Table 15–9. Parameters for the InterpolatingFIR Block

Parameter	Description
Read/Write mode	You can map the coefficients as Read , Write , Read/Write , or Constant . This field determines the type of address decode to build.
Filter structure	You can select Use All Taps , Half Band , or a specified band (from 3rd Band to 46th Band).
Expose Avalon-MM Slave in Simulink	Allows you to reconfigure coefficients without Qsys. Also, it also allows you to reprogram multiple FIR filters simultaneously. When you turn on this parameter, the following three additional input ports and two output ports appear in Simulink. <ul style="list-style-type: none"> ■ data, address, write ■ readdata, valid

Filter Coefficients

You can change filter coefficients at run time by writing to the memory-mapped interface. You do not need to create custom logic to deal with awkward update schemes.

You can set the base address of the memory-mapped coefficients with the **Base address** parameter and set the filter coefficients by entering a Simulink fixed-point array into the **Coefficients** parameter. You can generate a vector of coefficients either by entering an array of numbers, or using one of the many MATLAB functions to build the required coefficients.



For more information about Simulink fixed-point objects and MATLAB functions, refer to the MATLAB Help.

Port Interface

Table 15–10 shows the port interface for the InterpolatingFIR block.

Table 15–10. Port Interface for the InterpolatingFIR Block

Signal	Direction	Description
a	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of the data input signals. If v is high, the data on the a wire is valid.
c	Input	Indicates channel of data input signals. If v is high, then c indicates which channel the data corresponds to.
data	Input	The data port has the same type as the coefficients.
address	Input	Only available when you turn on Expose Avalon-MM Slave in Simulink . The address port is the same width as the system address width that you configure in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu. Also the base address is the same.
write	Input	Deassert the write port to make a read occur.
q	Output	The fixed-point filtered data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾
readdata	Output	The data port has the same type as the coefficients.

Table 15-10. Port Interface for the InterpolatingFIR Block

Signal	Direction	Description
valid	Output	Indicates a valid output.

Note to Table 15-10:

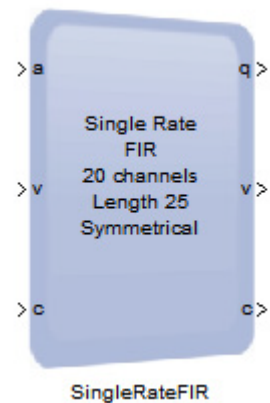
- (1) The output data can be non-zero when v is low.

Single-Rate FIR

The SingleRateFIR block implements a highly efficient multichannel finite impulse response filter across a broad range of parameters directly from a Simulink model. A memory-mapped interface allows you to read and write coefficients directly, easing system integration.

The SingleRateFIR block performs filtering on a stream of multichannel input data and produces a stream of output data with increased sampling frequency.

You can use the SingleRateFIR block in a digital up converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type. You can reduce the precision by using a separate Scale block, which can perform rounding and saturation to provide the required output precision.



Features

The SingleRateFIR block supports sample rates from 1 to 500, coefficient width in bitss from 2 to 32 bits, half-band and L-band Nyquist filters, real and complex filters, and symmetry and anti(negative)-symmetry.

Operation

Equation 15-1 on page 15-3 shows the basic convolution operation that a single-rate filter performs.

Parameters

Table 15-11 shows the parameters for the SingleRateFIR block.

Table 15-11. Parameters for the Single-Rate FIR Block

Parameter	Description
Input rate per channel	Specifies the sampling frequency of the input data per channel measured in millions of samples per second (MSPS).
Number of channels	Specifies the number of unique channels to process.
Symmetry	You can select Symmetrical or Anti-Symmetrical coefficients. Symmetrical coefficients can result in hardware resource savings over the unsymmetrical version.

Table 15–11. Parameters for the Single-Rate FIR Block

Parameter	Description
Coefficients	You can specify the filter coefficients using a Simulink fixed-point object <code>fi(0)</code> . The data type of the fixed-point object determines the width and format of the coefficients. The length of the array determines the length of the filter. For example, <code>fi(fir1(49, 0.3), 1, 18, 19)</code>
Base address	You can memory map the filter's coefficients into the address space of the system. This field determines the starting address for the coefficients. It is specified as a MATLAB <code>double</code> type (decimal integer) but you can use a MATLAB expression to specify a hexadecimal or octal type if required.
Read/Write mode	You can map the coefficients as Read , Write , Read/Write , or Constant . This field determines the type of address decode to build.
Expose Avalon-MM Slave in Simulink	Allows you to reconfigure coefficients without Qsys. Also, it also allows you to reprogram multiple FIR filters simultaneously. When you turn on this parameter, the following three additional input ports and two output ports appear in Simulink. <ul style="list-style-type: none"> ■ <code>data</code>, <code>address</code>, <code>write</code> ■ <code>readdata</code>, <code>valid</code>

Filter Coefficients

You can change filter coefficients at run time by writing to the memory-mapped interface. You do not need to create custom logic to deal with awkward update schemes.

You can set the base address of the memory-mapped coefficients with the **Base address** parameter and set the filter coefficients by entering a Simulink fixed-point array into the **Coefficients** parameter. You can generate a vector of coefficients either by entering an array of numbers or by using one of the many MATLAB functions to build the required coefficients.



For more information about Simulink fixed-point objects and MATLAB functions, refer to the MATLAB Help.

Port Interface

Table 15–12 shows the port interface for the `SingleRateFIR` block.

Table 15–12. Port Interface for the Single-Rate FIR Block

Signal	Direction	Description
<code>a</code>	Input	The fixed-point data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
<code>v</code>	Input	Indicates validity of the data input signals. If <code>v</code> is high, the data on the <code>a</code> wire is valid.
<code>c</code>	Input	Indicates channel of data input signals. If <code>v</code> is high, then <code>c</code> indicates the channel to which the data corresponds.
<code>data</code>	Input	The data port has the same type as the coefficients.
<code>address</code>	Input	Only available when you turn on Expose Avalon-MM Slave in Simulink . The address port is the same width as the system address width that you configure in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu. Also the base address is the same.
<code>write</code>	Input	Deassert the write port to make a read occur.

Table 15-12. Port Interface for the Single-Rate FIR Block

Signal	Direction	Description
q	Output	The fixed-point filtered data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
v	Output	Indicates validity of data output signals. ⁽¹⁾
c	Output	Indicates channel of data output signals. ⁽¹⁾
readdata	Output	The data port has the same type as the coefficients.
valid	Output	Indicates a valid output.

Note to Table 15-12:

(1) The output data can be non-zero when v is low.

This library provides memories and registers that you can access in your DSP datapath and with an external interface. You can use these blocks to configure coefficients or run-time parameters and read calculated values.

This library also provides blocks that you can use to simulate the bus interface in the Simulink environment.

The ModelBus library contains the following blocks:

- Bus Slave (BusSlave)
- Bus Stimulus (BusStimulus)
- Bus Stimulus File Reader (Bus StimulusFileReader)
- Register Bit (RegBit)
- Register Field (RegField)
- Register Out (RegOut)
- Shared Memory (SharedMem)

The ModelBus library does not support the Simulink complex data-type or vectors.

Bus Slave (BusSlave)

The BusSlave block provides direct access to the signals on the processor interface bus.



When you use BusSlave block in a design, DSP Builder disables all Avalon-MM interface pipelined reads for the whole design.

Any accesses to the memory region encapsulated by the base address (described by the **Memory Name** parameter) and size (described by the **Number of Words** parameter) over the bus is output over the a, d and w outputs.

You must provide logic to generate any appropriate response connected to the rd and rv inputs, which then returns over the processor interface. You must also add your own decoding logic to work with this block



All signals connected to the BusSlave block are within the bus clock domain. You must implement appropriate clock-crossing logic (such as a DualMem block).

Parameters

Table 16-1 shows the parameters for the BusSlave block.

Table 16-1. Parameters for the BusSlave Block

Parameter	Description
Memory Name	Specifies the memory region. Can be an expression but must evaluate to an integer address.
Read/Write Mode	Specifies the mode of the memory as viewed from the processor: <ul style="list-style-type: none"> ■ Read: processor can only read over specified address range. ■ Write: processor can only write over specified address range. ■ Read/Write: processor can read or write over specified address range. ■ Constant: processor cannot access specified address range. This option continues to reserve space in the memory map.
Number of Words to Address	Specifies the address range that this block accesses.
Description	Text describing what is at the specified address.
Evaluated Address Expression	Displays the evaluated value of the Memory Name expression when you click Apply .
Sample Time	Specifies the Simulink sample time.

Port Interface

Table 16-2 shows the port interface for the BusSlave block.

Table 16-2. Port Interface for the BusSlave Block

Signal	Direction	Type	Description
rd	Input	16-bit or 32-bit unsigned integer	Read data.
rv	Input	Boolean	Read data valid.
a	Output	Derived fixed-point type	Bus address.
d	Output	16-bit or 32-bit unsigned integer	Write data.
w	Output	Boolean	Write enable.

Bus Stimulus (BusStimulus)

Use the **Bus Stimulus File Reader (BusStimulusFileReader)** block with the BusStimulus block to simulate accesses over the processor interface in the Simulink environment.

The BusStimulus block performs hidden accesses to the registers and SharedMem blocks in the memory hierarchy of your model. It is an interface that allows another block to read and write to any address. The address and data ports act as though an external processor reads and writes to your system.

The BusStimulus block transmits data from its input ports (address, writedata and write) over the processor interface, and thus modifies the internal state of the memory-mapped registers and memories as appropriate. Any response from the simulated processor interface is output on the readdata and readvalid output ports.

For example, to use the `BusStimulus` block connect constants to the address and data inputs. A pulse on the write port then writes the data to any register mapped to the specified address. Put a counter on the address input to provide all the data in every memory location on the readdata port. DSP Builder asserts the `readdatavalid` output when a valid read data is on the readdata port.

Parameters

Table 16-3 shows the parameters for the `BusStimulus` block.

Table 16-3. Parameters for the BusStimulus Block

Parameter	Description
Sample Time	Specifies the Simulink sample time.
Show read enable	Turn on to show read enable port. If you use the <code>BusStimulus</code> with the <code>BusStimulusFileReader</code> blocks in a design, ensure this parameter is turned on or turned off in both blocks.

Port Interface

Table 16-4 shows the port interface for the `BusStimulus` block.

Table 16-4. Port Interface for the BusStimulus Block

Signal	Direction	Type	Description
address	Input	Unsigned integer	Address to access.
writedata	Input	16-, 32-, or 64-bit unsigned integer	Write data.
write	Input	Boolean	Write enable.
read	Input	Boolean	Read enable.
readdata	Output	16-, 32-, or 64-bit unsigned integer	Read data.
readdatavalid	Output	Boolean	Read data valid.

Bus Stimulus File Reader (Bus StimulusFileReader)

Using the `Bus Stimulus (BusStimulus)` block with the `BusStimulusFileReader` block allows you to simulate accesses over the processor interface in the Simulink environment.

The `BusStimulusFileReader` block reads a stimulus file (`.stm`) and generates signals that match the `BusStimulus` block.

A bus stimulus file describes a sequence of transactions to occur over the processor interface, together with expected read back values. This block reads such files and produces outputs for each entry in the file.

Bus stimulus files automatically write to any blocks that have processor mapped registers when you simulate a design. Any design with useful register files generates a bus stimulus file that you can use to bring your design out of reset (all registers 0). You can also write your own bus stimulus files with the following format:

```
MemSpace Address WriteData WE ExpReadData Mask
```

or

```
MemSpace Address WriteData WE RE ExpReadData Mask
```

where:

MemSpace specifies the memory space (the format supports multiple memory spaces).

Address is the word address.

WriteData is the data to write if any.

WE performs a write when 1.

ExpReadData is the expected read data. The value that is read from a location is checked against this value to allow self checking tests.

Mask specifies when the expected read data is checked, only the bits in this mask are checked, to allows you to read, write, or check specified bits in a register.

RE performs a read when 1. If RE is not present, assume RE is 1 when WE is 0.

The mask also masks the written data and performs a read-update-write cycle if you write to certain bits (i.e. not overwrite all of them).

During simulation, any mismatch between the expected read data (as the bus stimulus file describes) and the incoming read data (as the BusStimulus block provides) highlights and DSP Builder issues a warning.

Parameters

Table 16-5 shows the parameters for the BusStimulusFileReader block.

Table 16-5. Parameters for the BusStimulusFileReader Block

Parameter	Description
Enabled	Turn on to enable reading of the bus stimulus file data. You must turn on Has read enable in the BusStimulusFileReader block if you turn on Show read enable in the BusStimulus block.
Stimulus File Name	Specifies the file from which to read bus stimulus data.
Log File Name	Specifies the file to store a log of all attempted bus stimulus accesses.
Space Width	Specifies the width of the memory space as described in the bus stimulus file—must be the same as the width specified in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu.
Addr Width	Specifies the width of the address space as described in the bus stimulus file—must be the same as the width specified in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu.
Data Width	Specifies the width of the data as described in the bus stimulus file—must be the same as the width specified in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu.
Sample Time	Specifies the Simulink sample time.
Has read enable	Turn on to show read enable port. If you use the BusStimulusFileReader with the BusStimulus block in a design, ensure this parameter is turned on or turned off in both blocks.

Port Interface

Table 16–6 shows the port interface for the BusStimulusFileReader block.

Table 16–6. Port Interface for the BusStimulusFileReader Block

Signal	Direction	Type	Description
address	Output	Unsigned integer	Address from file.
checkstrobe	Output	Boolean	Indicates when the readexpected and mask signals should be checked against readdata.
endofstimulus	Output	Boolean	Generated signal to indicate when the end of the bus stimulus file is reached.
read	Output	Boolean	Read signal from file.
readdatavalid	Input	Boolean	Read data valid.
readdata	Input	16-bit or 32-bit unsigned integer	Read data.
readexpected	Output	16-bit or 32-bit unsigned integer	Expected read data from file.
space	Output	Unsigned integer	Memory space from file.
write	Output	Boolean	Write signal from file.
writedata	Output	16-bit or 32-bit unsigned integer	Data from file.
mask	Output	16-bit or 32-bit unsigned integer	Mask value from file.

Register Bit (RegBit)

The RegBit block provides a register bit that you can read in your model and read or write with the processor interface.

The **Register Name** field specifies the address of the register and must evaluate to an unsigned integer. The **Bit** parameter determines the location of the register bit in the memory-mapped word.

Parameters

Table 16–7 shows the parameters for the RegBit block.

Table 16–7. Parameters for the RegBit Block

Parameter	Description
Register Name	Specifies the address of the register. Must evaluate to an integer address.
Read/Write Mode	Specifies the mode of the memory as viewed from the processor: <ul style="list-style-type: none"> ■ Read: processor can only read over specified address range. ■ Write: processor can only write over specified address range. ■ Read/Write: processor can read or write over specified address range. ■ Constant: processor cannot access specified address range. This option continues to reserve space in the memory map.
Bit	Specifies the bit location of the memory-mapped register in a processor word (allows different registers to share same address).
Initial Value	Specifies the initial state of the register.

Table 16-7. Parameters for the RegBit Block

Parameter	Description
Description	Text describing the register. The description is propagated to the generated memory map.
Sample Time	Specifies the Simulink sample time.

Port Interface

Table 16-8 shows the port interface for the RegBit block.

Table 16-8. Port Interface for the RegBit Block

Signal	Direction	Type	Description
q	Output	Boolean	Data.

Register Field (RegField)

The RegField block provides a register field that you can read in your model and read or write with the processor interface.

The **Register Name** field specifies the address of the register and must evaluate to an unsigned integer.

The **Most Significant Bit** and **Least Significant Bit** parameters determine the size and location of the register bits in the memory-mapped word.

Parameters

Table 16-9 shows the parameters for the RegField block.

Table 16-9. Parameters for the RegField Block

Parameter	Description
Register Name	Specifies the address of the register. Must evaluate to an integer address.
Read/Write Mode	Specifies the mode of the memory as viewed from the processor: <ul style="list-style-type: none"> ■ Write: processor can only write over specified address range. ■ Read/Write: processor can read or write over specified address range. ■ Constant: processor cannot access specified address range. This option continues to reserve space in the memory map.
Most Significant Bit	Specifies the MSB of the memory-mapped register in a processor word (allows different registers to share same address).
Least Significant Bit	Specifies the LSB of the memory-mapped register in a processor word (allows different registers to share same address).
Register Output Type	Specifies the width and sign of the data type that the register stores. The size should equal (MSB – LSB + 1).
Register Output Scale	Specifies the scaling of data type that the register stores. For example, 2^{-15} for 15 of the above bits as fractional bits.
Initial Value	Specifies the initial state of the register.
Description	Text describing the register. The description is propagated to the generated memory map.
Sample Time	Specifies the Simulink sample time.

Port Interface

Table 16-10 shows the port interface for the RegField block.

Table 16-10. Port Interface for the RegField Block

Signal	Direction	Type	Description
q	Output	Any fixed-point type.	Data.

Register Out (RegOut)

The RegOut block provides a register field that you can read in your model and read with the processor interface.

The **Register Name** field specifies the address of the register and must evaluate to an unsigned integer.

The **Most Significant Bit** and **Least Significant Bit** parameters determine the size and location of the register bits in the memory-mapped word.

Parameters

Table 16-11 shows the parameters for the RegOut block.

Table 16-11. Parameters for the RegOut Block

Parameter	Description
Register Name	Specifies the address of the register. Must evaluate to an integer address.
Most Significant Bit	Specifies the MSB of the memory-mapped register in a processor word (allows different registers to share same address).
Least Significant Bit	Specifies the LSB of the memory-mapped register in a processor word (allows different registers to share same address).
Description	Text describing the register. The description is propagated to the generated memory map.
Sample Time	Specifies the Simulink sample time.

Port Interface

Table 16-12 shows the port interface for the RegOut block.

Table 16-12. Port Interface for the RegOut Block

Signal	Direction	Type	Description
d	Input	Any fixed-point type	Data.
w	Input	Boolean	Write data.

Shared Memory (SharedMem)

The SharedMem block provides a memory block that you can read in your model and read or write with the processor interface.



If you set the **Read/write** mode to **Read** only, you can write to it from your DSP Builder model using the **we** and **wd** signals. In any other mode, DSP Builder ignores these signals.

The length of the **Initial Data** parameter, 1-D array, determines the size of the memory. You can optionally initialize the generated HDL with this data.

Parameters

Table 16-13 shows the parameters for the SharedMem block.

Table 16-13. Parameters for the SharedMem Block

Parameter	Description
Memory-Mapped Address	Specifies the address of the memory block. Must evaluate to an integer address.
Read/Write Mode	Specifies the mode of the memory as viewed from the processor: <ul style="list-style-type: none"> ■ Read: processor can only read over specified address range. ■ Write: processor can only write over specified address range. ■ Read/Write: processor can read or write over specified address range. ■ Constant: processor cannot access specified address range. This option continues to reserve space in the memory map.
Initial Data	Specifies the initialization data. The size of the 1-D array determines the memory size.
Initialize Hardware Memory Blocks with Initial Data Contents	Turn on when you want to initialize the generated HDL with the specified initial data.
Description	Text describing the memory block. The description is propagated to the generated memory map.
Memory Output Type	Specifies the data type that the memory block stores.
Memory Output Scale	Specifies the scale factor to apply to the data stored in the memory block.
Sample Time	Specifies the Simulink sample time.

Port Interface

Table 16-14 shows the port interface for the SharedMem block.

Table 16-14. Port Interface for the SharedMem Block

Signal	Direction	Type	Description
a	Input	Unsigned integer	Address.
wd	Input	Any fixed-point type	Write data.
we	Input	Boolean	Write enable.
rd	Output	Any fixed-point type	Read data.

This library contains primitive operators such as add, multiply, and delay. It also includes functions to manipulate signal types that support building hardware functions that use MATLAB fixed-point types. You do not need to understand the details of the underlying FPGA architecture, as the primitive blocks are automatically mapped into efficient FPGA constructs.

You can design and debug your model quickly using zero-latency blocks, without having to track block latencies around your circuit, decreasing design time and reducing bugs.

You can let the synthesis tool pipeline your logic to give you the system clock frequency you need and take care of pipeline balancing. Arithmetic operators, such as adders, are pipelined to increase operating frequency. Register blocks and shared memories permit synthesis of the processor-datapath interface.

Your design remains portable between supported FPGA families, allowing you to future-proof your design investment.



Some primitives support Simulink 1-D vector and complex types.

The ModelPrim library contains the following primitive blocks:

- Absolute Value (Abs)
- Accumulator (Acc)
- Add
- Add SLoad (AddSLoad)
- AddSub
- AddSubFused
- AND Gate (And)
- Bit Combine (BitCombine)
- Bit Extract (BitExtract)
- Bit Reverse (BitReverse)
- Channel In (ChannelIn)
- Channel Out (ChannelOut)
- Compare (CmpCtrl)
- Compare Equality (CmpEQ)
- Compare Greater Than (CmpGE)
- Compare Less Than (CmpLT)
- Compare Not Equal (CmpNE)
- Complex Conjugate (ComplexConjugate)

- Constant (Const)
- Constant Multiply (Const Mult)
- Convert
- CORDIC
- Counter
- Count Leading Zeros, Ones, or Sign Bits (CLZ)
- Dual Memory (DualMem)
- Demultiplexer (Demux)
- Divide
- FIFO
- Floating-point Classifier (FloatClass)
- Floating-point Multiply Accumulate (MultAcc)
- General Purpose Input (GPIn)
- General Purpose Output (GPOut)
- Load Exponent (LdExp)
- Left Shift (LShift)
- Look-Up Table (Lut)
- Loop
- Math
- Minimum and Maximum (MinMax)
- MinMaxCtrl
- Multiply (Mult)
- Multiplexer (Mux)
- NAND Gate (Nand)
- Negate
- NOR Gate (Nor)
- NOT Gate (Not)
- OR Gate (Or)
- Polynomial
- Round
- Sample Delay (SampleDelay)
- Scalar Product
- Select
- Sequence
- Shift

- Sqrt
- Subtract (Sub)
- Synthesis Information (SynthesisInfo)
- Trig
- XNOR Gate (Xnor)
- XOR Gate (Xor)



Blocks from the ModelPrim library are often referred to in this document as primitive blocks, as they are simple low-level functions. A subsystem where each input routes through a ChannelIn or GPIn block, each output routes through a ChannelOut or GPOut, and that contains a SynthesisInfo block is referred to as a ModelPrim subsystem.

Vector and Complex Type Support

The ModelPrim and ModelVectorPrim libraries provide automatic support for arrays and complex types.

These modes of operation engage with type propagation, and provide a convenient automatic method for generating repeated design elements to operate on all the data elements within vector and complex signals.

Blocks automatically determine whether the data they process is in scalar or vector format and operate accordingly.

Using complex data (where it is supported) automatically causes DSP Builder to generate blocks internally, which processes both real and imaginary data elements.

The hardware elements that these processes generate fully incorporate into the optimization schemes available within DSP Builder advanced blockset.

No restrictions on the combination of vector and complex modes exist.

Vector Type Support

This topic describes element by element mode and mathematical vector mode.

Element by Element Mode

The blocks in the primitive library exhibit an element by element mode of operation when you use with vector types.

This mode provides a convenient way to generate a uniform array to handle each element of data in a vector signal, without having to manually instantiate multiple blocks.

Internally, DSP Builder generates identical block instantiations, one for each element in the vector signal. The vector width propagates through the Simulink system.

Change this mode of operation in one of the following two ways:

- Drive the block with a vector signal.

- Initialize the block with a vector of values. This option is only available for blocks that you can initialize with a user-specified value.

The following restrictions exist on the vectors:

- Vector signals must be of uniform type.
- Signals associated with a block must either be vectors of identical width, or scalar.

When you use a scalar value with vectors, DSP Builder uses a copy of the single scalar value with each data element in the vector signal.

This behavior is analogous to the scalar expansion that occurs with Simulink blocks.

Mathematical Vector Mode

The blocks in the primitive vector library perform mathematical operations with vector data.

The outputs of these blocks are potentially a function of any or all of the inputs. Vector width does not necessarily propagate.

The `SumOfElements` block exhibits this behavior.

Interactions with Simulink

You can use Simulink `Mux` and `Demux` to manipulate signals within DSP Builder advanced blockset designs.

Complex Support

Some blocks can automatically process complex data, which provides a convenient way to simultaneously generate data and control pathways for the real and imaginary components of such data.

For each complex value, two identical block instantiations generate internally, for the real and imaginary components.

The complex nature of the data propagates. Strictly real signals expand to provide a value for the imaginary component with complex data. The exact behavior depends on the nature of the port associated with the real signal. The real value is duplicated for control or address signals. The real and imaginary parts of complex data are subject to identical control signals. A zero imaginary value generates for real data signals in a complex data context. Real data values, x , expand, when required, to $x + 0i$.

Restrictions

Not all primitive blocks support complex data. Data signals are the only signal type permitted to be complex. DSP Builder issues an error message if an attempt is made to drive control or address signals with complex values.

Interactions with Simulink

The following elements of the Simulink environment are available for use with the primitive blocks:

- Simulink Complex to Real-Imag and Real-Imag to Complex blocks may manipulate complex signals within DSP Builder advanced blockset designs.
- Simulink Scope blocks can display signals, but they do not directly support complex data. Attempting to view complex data generates a type propagation error.

Use a Complex to Real-Imag block to convert the complex signal.

You can use the complex Simulink function `complex(x, y)` to generate initialization values. Use this function to ensure DSP Builder always treats data as complex.

Simulink automatically converts complex values of form $(x + 0i)$ to real values, which can cause type propagation errors. The `complex()` function can resolve this problem.

Use `complex(x, 0)` to ensure such data is treated as complex.

Absolute Value (Abs)

The Abs block outputs the absolute value of the input:

$$q = \text{abs}(a)$$



Parameters

Table 17–1 shows the parameters for the Abs block.

Table 17–1. Parameters for the Abs Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that the number of bits of the input can store. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17–1:

(1) If you use the Abs block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-2 shows the port interface for the Abs block.

Table 17-2. Port Interface for the Abs Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand	Yes	No
q	Output	Derived fixed- or floating-point type	Result	Yes	No

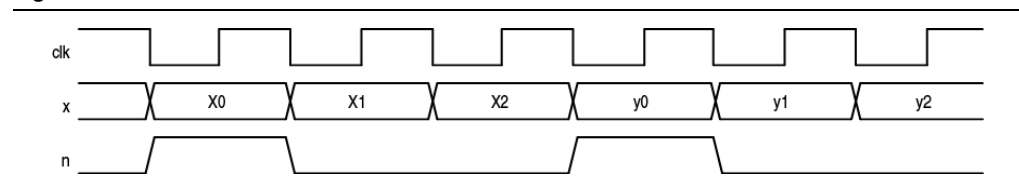
Accumulator (Acc)

The Acc block implements an application-specific floating-point accumulator.

$$r = \text{acc}(x, n)$$

The acc block allows accumulating data sets of variable lengths. The block indicates a new data set by setting *n* high with the first element of the accumulation (see Figure 17-1).

Figure 17-1. New Data Set



This example accumulates $x_0 + x_1 + x_2$ and $y_0 + y_1 + y_2$.

Parameters

The acc block has single and double-precision floating-point data inputs and outputs.

Table 17-5 shows the parameters for the Add block.

Table 17-3. Parameters for the Add Block

Parameter	Description
LSBA	This parameter defines the weight of the accumulator's LSB, and therefore the accuracy of the accumulation. This value and the maximum number of terms to be accumulated sets the accuracy of the accumulator. The maximum number of terms the design can accumulate can invalidate the $\log_2(N)$ lower bits of the accumulator. For instance, if an accuracy of 2^{-30} is enough, and you add 1k of numbers, $LSBA = -30 - \log_2(1k)$, which is approximately -40.
MSBA	The weight of the MSB of the accumulation result. Adding a few guard bits to the value has little impact on the implementation size. You can set this parameter in one of the following ways: <ul style="list-style-type: none"> ■ For a stock simulation, to limit the value of any stock to \$100k before the simulation is invalid, use a value of $\text{ceil}(\log_2(100K)) \sim \text{ceil}(16.6) = 17$ ■ For a simulation where the implemented circuit adds numbers ≤ 1, for one year, at 400MHz, use $\text{ceil}(\log_2(365 * 86400 * 400 * 10^6)) \sim 54$
maxMSBX	The maximum weight of the inputs. When adding probabilities ≤ 1 set this weight to 0. When adding data from sensors, set bounds on the input ranges. Alternatively, set $\text{MaxMSBX} = \text{MSBA}$. However, the size of the architecture may increase.

Port Interface

Table 17-4 shows the port interface for the Acc block.

Table 17-4. Port Interface for the Acc Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Single or double	Operand	Yes	No
n	Input	Boolean	Control	Yes	No
r	Output	Single or double	Output	Yes	No
xO	Output	Boolean	This flag goes high when the input value has a weight larger than selected value for MaxMSBX . The result of the accumulation is then invalid.	Yes	No
xU	Output	Boolean	If this flag goes high, an input value is completely shifted out of the accumulator. This flag warns that the value of $LSBA$ is possibly too large.	Yes	No
aO	Output	Boolean	This flag goes high when the accumulated value has a weight larger than $MSBA$. The result of the accumulation is then invalid.	Yes	No

Add

The Add block outputs the sum of the inputs:

$$q = a + b$$

For two or more inputs, the Add block outputs the sum of the inputs:

$$q = a + b + \dots$$

For a single vector input, the Add block outputs the sum of elements:

$$q = \sum a_n$$

For a single scalar input, the Add block outputs the input value:

$$q = a$$



Parameters

Table 17-5 shows the parameters for the Add block.

Table 17-5. Parameters for the Add Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that you can store in the number of bits of the input. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Number of Inputs	Specifies the number of inputs.

Note to Table 17-5:

(1) If you use the Add block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-6 shows the port interface for the Add block.

Table 17-6. Port Interface for the Add Block

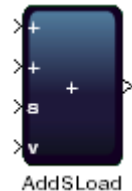
Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes (scalar output in one input case).	Yes

Add SLoad (AddSLoad)

The AddSLoad block performs the following function:

$$q = s ? v : (a + b)$$

If the *s* input is low, output the sum of the first 2 inputs, *a* + *b*, else if *s* is high, then output the value *v*.



Parameters

Table 17-7 shows the parameters for the AddSLoad block.

Table 17-7. Parameters for the AddSLoad Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that you can store in the number of bits of the input. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-7:

(1) If you use the AddSLoad block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-8 shows the port interface for the AddLoad block.

Table 17-8. Port Interface for the AddLoad Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
s	Output	Any fixed- or floating-point type	Synchronous load	Yes	No
v	Output	Any fixed- or floating-point type	Value to load if s is true	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

AddSub

The AddSub block produces either the sum ($a + b$) or the difference ($a - b$) depending on the input you select (1 for add; 0 for subtract).



For single-precision inputs and designs targeting Arria 10 devices, the block uses a mixture of resources including the Arria10 DSP blocks in floating-point mode.

Parameters

None.

Port Interface

Table 17-9 shows the port interface for the AddSub block.

Table 17-9. Port Interface for the AddSub Block

Signal	Direction	Type	Description	Vector	Complex
a	Input	Single or double	Operand 1	Yes	Yes
b	Input	Single or double	Operand 2	Yes	Yes
add	Input	Boolean	Select input	Yes	Yes
q	Output	Single or double	Result	Yes	Yes

AddSubFused

The AddSubFused block produces both the sum and the difference of the IEEE floating-point signals that arrive on the input ports.

Parameters

None.

Port Interface

Table 17-10 shows the port interface for the AddSubFused block.

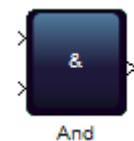
Table 17-10. Port Interface for the AddSubFused Block

Signal	Direction	Type	Description	Vector	Complex
	Input	Single or double	Operand 1	Yes	Yes
b	Input	Single or double	Operand 2	Yes	Yes
+	Output	Single or double	Result 1	Yes	Yes
-	Output	Single or double	Result 2	Yes	Yes

AND Gate (And)

The And block outputs the logical AND of the input values.

If the number of inputs is set to 1, then the logical and of all the individual bits of the input word is output.



Parameters

Table 17-11 shows the parameters for the And block.

Table 17-11. Parameters for the And Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-11:

(1) If you use the And block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-12 shows the port interface for the And block.

Table 17-12. Port Interface for the And Block

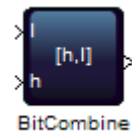
Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
unnamed	Input	Any fixed-point type	Operands 1 to n	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Bit Combine (BitCombine)

The BitCombine block outputs the bit concatenation of the input values:

```
((h << bitwidth(i)) | i)
```

You can change the number of inputs on the BitCombine block according to your requirements. When Boolean vectors are input on multiple ports, DSP Builder combines corresponding components from each vector and outputs a vector of signals. The widths of all input vectors must match. However, the widths of the signals arriving on different inputs do not have to be equal. The one input BitCombine block is a special case that concatenates all the components of the input vector, so that one wide scalar signal is output. Use with logical operators to apply a 1-bit reducing operator to Boolean vectors.



Parameters

Table 17-13 shows the parameters for the BitCombine block.

Table 17-13. Parameters for the BitCombine Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-13:

(1) If you use the BitCombine block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-14 shows the port interface for the BitCombine block.

Table 17-14. Port Interface for the BitCombine Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
i	Input	Any fixed-point type	Operand	Yes	No
h	Input	Any fixed-point type	Operand	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Bit Extract (BitExtract)

The BitExtract block outputs the bits extracted from the input, and recast as the specified data type:

$$q = (a \gg \text{LSB})$$

If bit position is a negative number, the bit position is an offset from the MSB instead of LSB.



If the BitExtract block initialization parameter is a vector of LSB positions, the output is a vector of matching width, even if the input is a scalar signal. Use this feature to split a wide data line into a vector of Boolean signals. The components are in the same order as you specify in the initialization parameter. If the input to the BitCombine block is a vector, the width of any vector initialization parameter must match, and then a different bit can be selected from each component in the vector. The output data type does not always have to be Boolean signals. For example, setting to uint8 provides a simple way to split one wide signal into a vector of unsigned 8-bit data lines.

Parameters

Table 17-15 shows the parameters for the BitExtract block.

Table 17-15. Parameters for the BitExtract Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Table 17-15. Parameters for the BitExtract Block

Parameter	Description
Least Significant Bit Position from Input Word	Specifies the bit position from the input word as the LSB in the output word.

Note to Table 17-15:

- (1) If you use the BitExtract block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-16 shows the port interface for the BitExtract block.

Table 17-16. Port Interface for the BitExtract Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Bit Reverse (BitReverse)

The BitReverse primitive block reverses the bits at the input. The MSB is output as the LSB.



Parameters

The BitReverse block has no parameters.

Port Interface

Table 17-17 shows the port interface for the BitReverse block.

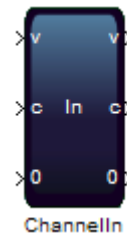
Table 17-17. Port Interface for the BitReverse Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Channel In (Channelln)

The ChannelIn block delineates the input boundary of a synthesizable ModelPrim subsystem.

The ChannelIn block passes its input through to the outputs unchanged, with types preserved. This block indicates to DSP Builder that these signals arrive synchronized from their source, so that the synthesis tool can interpret them.



Parameters

Table 17-18 shows the parameters for the ChannelIn block.

Table 17-18. Parameters for the ChannelIn Block

Parameter	Description
Number of data signals	Specifies the number of data signals on this block.

Port Interface

Table 17-19 shows the port interface for the ChannelIn block.

Table 17-19. Port Interface for the ChannelIn Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
v	Input	Boolean	Valid input signal	No	No
c	Input	uint(8)	Channel input signal	No	No
0, 1, 2, ...	Input	Any fixed-or floating-point type	A number of input data signals	Yes	Yes
v	Output	Boolean	Valid signal	No	No
c	Output	uint(8)	Channel signal	No	No
0, 1, 2, ...	Output	Any fixed- or floating-point type	A number of data signals	Yes	Yes

Channel Out (ChannelOut)

The ChannelOut block delineates the output boundary of a synthesizable ModelPrim subsystem.

The ChannelOut block passes its input through to the outputs unchanged, with types preserved. This block indicates to DSP Builder that these signals must synchronize, which the synthesis tool can ensure.

When you run a simulation in Simulink, DSP Builder adds additional latency from the balanced pipelining stages to meet the specified timing constraints for your model. The block accounts for this additional latency. This latency does not include any delay explicitly added to your model, by for example a [Sample Delay \(SampleDelay\)](#) block, just added pipelining for timing closure.



You can also access the value of the latency parameter by typing a command of the following form on the MATLAB command line:

```
get_param(gcf, 'latency')
```

Parameters

Table 17-20 shows the parameters for the ChannelOut block.

Table 17-20. Parameters for the ChannelOut Block

Parameter	Description
Number of data signals	Specifies the number of data signals on this block.

Port Interface

Table 17-21 shows the port interface for the ChannelOut block.

Table 17-21. Port Interface for the ChannelOut Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
v	Input	Boolean	Valid output signal	No	No
c	Input	8-bit unsigned integer	Channel output signal	No	No
0, 1, 2, ...	Input	Any fixed-or floating-point type	A number of output data signals	Yes	Yes
v	Output	Boolean	Valid signal	No	No
c	Output	8-bit unsigned integer	Channel signal	No	No
0, 1, 2, ...	Output	Any fixed-or floating-point type	A number of data signals	Yes	Yes

Compare (CmpCtrl)

The CmpCtrl block produces the Boolean result of comparing two IEEE floating-point input signals. A select line controls the comparison. The select line is at least three-bits wide to select from five different comparison operators.

Table 17-22 shows the comparison operators for the CmpCtrl block.

Table 17-22. Comparison Operators

s	Comparison Operator
0	Less than.
1	Less than or equal to.
2	Equal.
3	Greater than or equal to.
4	Greater than.
5	Not equal.

Parameters

None.

Port Interface

Table 17-23 shows the port interface for the CmpCtrl block.

Table 17-23. Port Interface for the CmpCtrl Block

Signal	Direction	Type	Description	Vector	Complex
	Input	Single or double	Operand 1	Yes	No
b	Input	Single or double	Operand 2	Yes	No
s	Input	Fixed-point (unsigned)	Select input	Yes	No
q	Output	Boolean	Result	Yes	No

Complex Conjugate (ComplexConjugate)

The ComplexConjugate block outputs the complex conjugate of its input value.

If $a = x + yi$,

then $q = x - yi$

If the input value is real, an unchanged real value is output

$q = a$

Parameters

Table 17-5 shows the parameters for the ComplexConjugate block.

Table 17-24. Parameters for the ComplexConjugate Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that you can store in the number of bits of the input. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Number of Inputs	Specifies the number of inputs.

Port Interface

Table 17-25 shows the port interface for the ComplexConjugate block.

Table 17-25. Port Interface for the ComplexConjugate Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

Compare Equality (CmpEQ)

The CmpEQ block outputs true if and only if the two inputs have the same value:

$$a == b$$


Parameters

The CmpEQ block has no parameters.

Port Interface

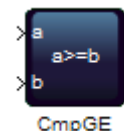
Table 17-26 shows the port interface for the CmpEQ block.

Table 17-26. Port Interface for the CmpEQ Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Boolean	Result	Yes	Yes

Compare Greater Than (CmpGE)

The CmpGE block outputs true if and only if the first input is greater than or equal to the second input:

$$a >= b$$


Parameters

The CmpGE block has no parameters.

Port Interface

Table 17-27 shows the port interface for the CmpGE block.

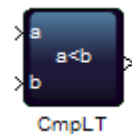
Table 17-27. Port Interface for the CmpGE Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	No
b	Input	Any fixed- or floating-point type	Operand 2	Yes	No
q	Output	Boolean	Result	Yes	No

Compare Less Than (CmpLT)

The CmpLT block outputs true if and only if the first input is less than the second input:

$$a < b$$



Parameters

The CmpLT block has no parameters.

Port Interface

Table 17-28 shows the port interface for the CmpLT block.

Table 17-28. Port Interface for the CmpLT Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	No
b	Input	Any fixed- or floating-point type	Operand 2	Yes	No
q	Output	Boolean	Result	Yes	No

Compare Not Equal (CmpNE)

The CmpNE block outputs true if the two inputs do not have the same value:

$$a \neq b$$



Parameters

The CmpNE block has no parameters.

Port Interface

Table 17–29 shows the port interface for the CmpNE block.

Table 17–29. Port Interface for the CmpNE Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Boolean	Result	Yes	Yes

Constant (Const)

The Const block outputs a specified constant value.



Parameters

Table 17–30 shows the parameters for the Const block. Every block in the ModelPrim

Table 17–30. Parameters for the Const Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via back projection: a downstream block that this block drives determines the output data type. If the driven block does not propagate a data type to the driver, you must use a Simulink SameDT block to copy the required data type to the output wire. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Single: single-precision floating-point data. ■ Double: double-precision floating-point data. ■ Variable precision floating point: variable precision floating-point output type
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Value	Specifies the constant value. This parameter may also be a fi object when specifying data of arbitrarily high precision.
Floating point precision	Specifies the floating-point precision. For example, <code>float32_m23</code> .

Note to Table 17–30:

(1) If you use the Const block in a loop, you must specify the output type for at least one block in the loop.

library accepts double-precision floating-point values when specifying mask parameters. This format limits precision to no more than 53 bits, which is more than sufficient for most of the blocks. For higher precision, the Const, DualMem, or LUT blocks optionally accept values using Simulink's Fixed Point data type. For example:

```
constValue = fi(0.142, 1, 16, 15)
vectorValue = fi(sin([0:10]'), 1, 18, 15)
```

To configure a Const, DualMem, or LUT with data of precision higher than IEEE double precision, create a MATLAB fi object of the required precision that contains the high precision data. Avoid truncation when creating this object. Use the fi object to specify the **Value** of the Const, the **Initial Contents** of the DualMem block, or the **Output value map** of the LUT block.

Port Interface

Table 17-31 shows the port interface for the Const block.

Table 17-31. Port Interface for the Const Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
q	Output	Any fixed- or floating-point type	Constant value.	Yes	Yes

Constant Multiply (Const Mult)

The Const Mult block scales the input by a user configurable coefficient and outputs the result.

Parameters

The **Value** parameter is a floating-point scaling factor that is multiplied by the input signal. If this parameter is a vector, the output is a vector. If both the input and the **Value** parameter are vectors, they must have the same length. If the **Value** parameter is complex, the block performs a complex multiply and the output is complex.

Port Interface

Table 17-32 shows the port interface for the Const Mult block.

Table 17-32. Port Interface for the Const Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Any floating-point type	Operand 1	Yes	Yes
q	Output	Any fixed- or floating-point type	Result	Yes	Yes

Convert

The Convert block performs a type conversion of the input, and outputs the new data type.

You can optionally perform truncation, biased, or unbiased rounding if the output data type is smaller than the input. The LSB must be a value in the width in bits of the input type.



Parameters

Table 17-33 shows the parameters for the Convert block.

Table 17-33. Parameters for the Convert Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via back projection: a downstream block that this block drives determines the output data type. If the driven block does not propagate a data type to the driver, you must use a Simulink SameDT block to copy the required data type to the output wire. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Single: single-precision floating-point data. ■ Double: double-precision floating-point data. ■ Variable precision floating point: variable precision floating-point output type
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Rounding method	<p>Determines the rounding mode:</p> <ul style="list-style-type: none"> ■ Truncate: Discard any bits that fall below the new least significant bit. ■ Biased: Add 0.5 LSB and then truncate. This rounds towards infinity. ■ Unbiased: If the discarded bits equal 0.5 LSB of the new value then round towards the even integer, otherwise perform add 0.5 LSB and then truncate. This prevents the rounding operation introducing a DC bias where 0.5 always rounds towards positive infinity.
Saturation	The Convert block allows saturation, which has an optional clip detect output that outputs 1 if any clipping has occurred. Saturation choices are none , symmetric , or asymmetric .
Floating point precision	Specifies the floating-point precision. For example, <code>float32_m23</code> .

Note to Table 17-33:

(1) If you use the Convert block in a loop, you must specify the output type for at least one block in the loop.

For example, for an Add or Mult block, you can select the output word-length and fractional part using dialog.

Specifying the output type is a casting operation, which does not preserve the numerical value, only the underlying bits. This method never adds hardware to a block— just changes the interpretation of the output bits.

For example, for a multiplier with both input data-types, `sfix16_En15` has output type `sfix32_En30`. If you select output format `sfix32_En28`, the output numerical value multiplies by four. For example, `1*1` input gives an output value of 4.

If the you select output format `sfix32_En31`, the output numerical value is divided by two. For example `1*1` input gives an output value of 0.5.

If you want to change data-type format in a way that preserves the numerical value, use a `convert` block, which adds the corresponding hardware. Adding a `convert` block directly after a `primitive` block lets you specify the data-type to preserve the numerical value.

For example, a `Mult` block followed by a `Convert` block, with input values `1*1` always give output value 1.

Port Interface

Table 17-34 shows the port interface for the `Convert` block.

Table 17-34. Port Interface for the Convert Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Data	Yes	Yes
q	Output	Specified fixed-point type	Data	Yes	Yes

CORDIC

The `CORDIC` block performs a coordinate rotation using the coordinate rotation digital computer algorithm.

The CORDIC algorithm is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions. It calculates the trigonometric functions of sine, cosine, magnitude and phase (arctangent) to any desired precision. A CORDIC algorithm is useful when you do not want to use a hardware multiplier, because the only operations it requires are addition, subtraction, bit shift and lookup.

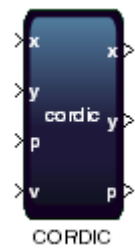
The CORDIC algorithm is generally faster than other approaches when you do not want to use a hardware multiplier, or you want to minimize the number of gates required. Alternatively, when a hardware multiplier is available, table-lookup and power series methods are generally faster than CORDIC.

CORDIC is based on rotating the phase of a complex number, by multiplying it by a succession of constant values. The multiplications can all be powers of 2, which you can perform with just shifts and adds in binary arithmetic. Therefore you need no actual multiplier function.

During each multiplication, a gain occurs equal to:

$$K_i = \sqrt{1 + 2^{-2i}}$$

where i represents the i th iterative step.



The total gain of the successive multiplications has a value of:

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}}$$

where n is the number of iterations.

You can calculate this total gain in advance and stored in a table. Additionally:

$$K = \lim_{n \rightarrow \infty} K(n) \approx 1.64676$$

The CORDIC block implements the these iterative steps using a set of shift-add algorithms to perform a coordinate rotation.

The CORDIC block takes four inputs, where the x and y inputs represent the (x, y) coordinates of the input vector, the p input represents the angle input, and the v represents the mode of the CORDIC block. It supports the following modes:

- The first mode rotates the input vector by a specified angle.
- The second mode rotates the input vector to the x -axis while recording the angle required to make that rotation.

The x and y inputs must have the same width in bits. The input width in bits of the x and y inputs determines the number of stages (iterations) inside the CORDIC block, unless you explicitly specify an output width in bits smaller than the input width in bits in the block parameters.

The CORDIC gain is completely ignored to save time and resource. The width in bits of the x and y inputs automatically grows by two bits inside the CORDIC block to account for the gaining factor of the CORDIC algorithm. Hence the x and y outputs are two bits wider than the input and you must handle the extra two bits in your design, if you have not specified the output width in bits explicitly through the block parameters. You can compensate for the CORDIC gain outside the CORDIC block.

The p input is the angular value and has a range between $-\pi$ and $+\pi$, which requires at least three integer bits to fully represent the range. The v input determines the mode. You can trade accuracy for size (and efficiency) by specifying a smaller output data width to reduce the number of stages inside the CORDIC block.

Parameters

Table 17-35 shows the parameters for the CORDIC block.

Table 17-35. Parameters for the CORDIC Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-35:

(1) If you use the CORDIC block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-36 shows the port interface for the CORDIC block.

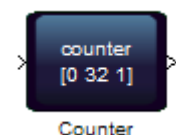
Table 17-36. Port Interface for the CORDIC Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Any fixed-point type	x coordinate of the input vector.	Yes	Yes
y	Input	Any fixed-point type	y coordinate of the input vector.	Yes	Yes
p	Input	Any fixed-point type	Required angle of rotation in the range between $-\pi$ and $+\pi$.	Yes	Yes
v	Input	Any fixed-point type	Selects the mode of operation (0 = rotate by angle, 1 = rotate to x-axis).	Yes	Yes
x	Output	Any fixed-point type	x coordinate of the output vector.	Yes	Yes
y	Output	Any fixed-point type	y coordinate of the output vector.	Yes	Yes
p	Output	Any fixed-point type	Angle through which the coordinates rotate	Yes	Yes

Counter

The Counter block maintains a counter and outputs the counter value each cycle.

The input is a counter enable and allows you to implement irregular counters. The counter initializes to the value that you provide, and counts with the modulo, with the step size you provide:



```
count = _pre_initialization_value;
while (1) { if (en) count = (count + _step_size) % _modulo}
```



If you create a counter with a preinitialization value of 0 and with a step of 1, it outputs the value 1 (not 0) on its first enabled cycle. If you want the counter to output 0 on its first valid output, initialize with:

```
[<(modulo - step size)> <modulo> <step size>]
```



Modulo and step size cannot be coprime—the step size must exactly divide into the modulo value.

Parameters

Table 17-37 shows the parameters for the Counter block.

Table 17-37. Parameters for the Counter Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Counter setup	<p>A vector that specifies the counter in the format:</p> <pre>[<pre_initialization_value> <modulo> <step size>]</pre> <p>For example, <code>[0 32 1]</code></p>

Note to Table 17-37:

(1) If you use the Counter block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-38 shows the port interface for the Counter block.

Table 17-38. Port Interface for the Counter Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
en	Input	Boolean	Count enable	Yes	No
q	Output	Specified fixed-point type	Result	Yes	No

Count Leading Zeros, Ones, or Sign Bits (CLZ)

The CLZ block counts the leading zeroes, ones, or sign bits of the input, and outputs that count.



Parameters

Table 17-39 shows the parameters for the CLZ block.

Table 17-39. Parameters for the CLZ Block

Parameter	Description
Mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Count Leading Zeroes: returns the count of the leading zeros in the input ■ Count Leading Ones: returns the count of the leading ones in the input ■ Count Leading Digits: returns the count of the leading sign digits in the input

Port Interface

Table 17-40 shows the port interface for the CLZ block.

Table 17-40. Port Interface for the CLZ Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand	Yes	No
q	Output	Derived fixed-point type	Number of consecutive zero bits in input word starting from the MSB	Yes	No

Dual Memory (DualMem)

The DualMem block models a dual interface memory structure. You can read or write the first data interface (inputs d, a, and w). The second data interface (specified on the second address input) is read only. The memory size is inferred from the size of the initialization array.

The behavior of read during write cycles of the memories depends on the interface to which you read:

- Reading from q1 while writing to interface 1 outputs the new data on q1 (write first behavior).
- Reading from q2 while writing to interface 1 outputs the old data on q2 (read first behavior).

Turning on **DONT_CARE** may give a higher f_{MAX} for your design, especially if you implement the memory as a MLAB. When this option is on, the output is not double-registered (and therefore, in the case of MLAB implementation, uses fewer external registers), and you gain an extra half-cycle on the output. The word don't care overlaid on the block symbol indicates the current setting is **DON'T CARE**. The default is off, which outputs old data for read-during-write.



For more information about this option, refer to the *Read-During-Write Output Behavior* section in the *RAM Megafunction User Guide*.

Parameters

Table 17-41 shows the parameters for the DualMem block.

Table 17-41. Parameters for the DualMem Block

Parameter	Description
Output data type mode	Determines how the block sets its output data type: <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>.⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16), uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Initial contents	Specifies the initialization data. The size of the 1-D array determines the memory size. This parameter may also be a <code>fi</code> object when specifying data of arbitrarily high precision.
Use DONT_CARE when reading from and writing to the same address	Turn this option on to produce faster hardware (a higher f_{MAX}) but with uncertain read data in hardware if you are simultaneously reading from and writing to the same address. Ensure that you do not read from or write to the same address at the same time to guarantee valid read data.

Note to Table 17-41:

(1) If you use the DualMem block in a loop, you must specify the output type for at least one block in the loop.

You can specify the contents of the DualMem block in one of the following ways:

- Use a single row or column vector to specify table contents. The length of the 1D row or column vector determines the number of addressable entries in the table. If DSP Builder reads vector data from the table, all components of a given vector share the same value.
- When a look-up table contains vector data, you can provide a matrix to specify the table contents. The number of rows in the matrix determines the number of addressable entries in the table. Each row specifies the vector contents of the corresponding table entry. The number of columns must match the vector length, otherwise DSP Builder issues an error.

Every block in the ModelPrim library accepts double-precision floating-point values when specifying mask parameters. This format limits precision to no more than 53 bits, which is more than sufficient for most of the blocks. For higher precision, the Const, DualMem, or LUT blocks optionally accept values using Simulink's Fixed Point data type. For example:

```
constValue = fi(0.142, 1, 16, 15)
vectorValue = fi(sin([0:10]'), 1, 18, 15)
```

To configure a Const, DualMem, or LUT with data of precision higher than IEEE double precision, create a MATLAB fi object of the required precision that contains the high precision data. Avoid truncation when creating this object. Use the fi object to specify the **Value** of the Const, the **Initial Contents** of the DualMem block, or the **Output value map** of the LUT block.

Port Interface

Table 17-42 shows the port interface for the DualMem block.

Table 17-42. Port Interface for the DualMem Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
d	Input	Any fixed-point type	Data to write for interface 1	Yes	Yes
a	Input	Unsigned integer	Address to read/write from for interface 1	Yes	No
w	Input	Boolean	Write is enabled for interface 1 when 1, read is enabled for interface 1 when 0	Yes	No
a	Input	Unsigned integer	Address to read from for interface 2	Yes	No
q1	Output	Fixed-point type	Data out from interface 1. (1)	Yes	Yes
q2	Output	Fixed-point type	Data out from interface 2. (1)	Yes	Yes
Note: (1) If the address for interface 1 exceeds the memory size, q1 is not defined. If the address for interface 2 exceeds the memory size, q2 is not defined. To write to the same location as DSP Builder reads from with q2, you must provide the same address on both interfaces.					

Demultiplexer (Demux)

The Demux block deserializes the DSP Builder protocol bus on its inputs to produce a configurable number of output signals without TDM.

The Demux block is a primitive version of the ChannelViewer block

Parameters

Table 17-43 shows the parameters for the Demux block.

Table 17-43. Parameters for the Demux Block

Parameter	Description
Number of output channels	A vector of the channel number you want to see for example [0 1 3].
Number of input channels	The number of input channels. The block takes valid, channel, and (vector) data inputs. The channel is the normal channel count, which varies across 0 to NumberOfChannelsPerWire.

Divide

The Divide block outputs the first input, a, divided by the second input, b.

$$q = a/b$$

Parameters

Table 17-44 shows the parameters for the Divide block.

Table 17-44. Parameters for the Divide Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: if the input data types are floating-point, the output data type is the same floating-point data type. Mixing different precisions is not allowed. If the input data types are fixed-point, the output data type is fixed point with bitwidth equal to the sum of the bitwidths of the input data types. The fraction width is equal to the sum of the fraction width of the a-input data type, and the integer bitwidth of the b-input data type. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option type casts the output to the chosen fixed-point type. Attempting to type cast floating-point input is disallowed. You can only use this option to trim bits off the least significant end of the output data type that is otherwise inherited.
Output data type	Specifies the output data type. For example, <code>fixdt(1,16,15)</code>
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Float point rounding	<p>This option only has an effect for floating-point inputs.:</p> <ul style="list-style-type: none"> ■ Correct: the result is correctly rounded IEEE ■ Faithful: the result is may be rounded up or may be rounded down

Table 17-46 shows the data-type inheritance for fixed-point inputs.

Table 17-45. Data-Type Inheritance for Fixed-Point Inputs

Port	Fixed-Point Data Type	Integer Bits	Fraction Bits
a	<code>sfix16_en10</code>	6	10

Table 17-45. Data-Type Inheritance for Fixed-Point Inputs

Port	Fixed-Point Data Type	Integer Bits	Fraction Bits
b	sfix12_en7	5	7
q (Inherit via internal rule)	sfix28_en15	$6 + 7 = 13$	$10 + 5 = 15$

If you specify **Specify via dialog** for the **Output data type mode**, the block restricts the allowed data types to: sfix28_en15, sfix27_en14, sfix26_en13, etc.

Port Interface

Table 17-46 shows the port interface for the Divide block.

Table 17-46. Port Interface for the Divide Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type.	Numerator	Yes	No
b	Input	Any fixed- or floating-point type.	Denominator	Yes	No
q	Output	Any fixed- or floating-point type.	Result	Yes	No

FIFO

The FIFO block models a FIFO memory. DSP Builder writes data through the *d* input when the write-enable input *w* is high. After some implementation-specific number of cycles, DSP Builder presents data at output *q* and the valid output *v* goes high. DSP Builder holds this data at output *q* until the read acknowledge input *r* is set high.

The FIFO block wraps the Altera single clock FIFO (SCFIFO) megafunction operating in show-ahead mode. That is, the read input, *r*, is a read acknowledgement which means the DSP Builder has read the output data, *q*, from the FIFO buffer, so you can delete it and show the next data output on *q*. The data you present on *q* is only valid if the output valid signal, *v*, is high

Parameters

Table 17-47 shows the parameters for the FIFO block.

Table 17-47. Parameters for the FIFO Block

Parameter	Description
FIFO Setup	<p>A vector of three non-zero integers in the format: [<i><depth></i> <i><fill_threshold></i> <i><full_period></i>]</p> <ul style="list-style-type: none"> ■ <i>depth</i> specifies the maximum number of data values that the FIFO can store. ■ <i>fill_threshold</i> specifies a low-threshold for empty-detection. If the number of data items in the memory is greater than the low-threshold, the <i>t</i> output is 1 (otherwise it is 0). ■ <i>full_period</i> specifies a high-threshold for full-detection. If the number of data items is greater than the high-threshold, output <i>f</i> is 1 (otherwise it is 0).

If the inputs *w* or *r* is a vector, the FIFO setup parameter must be a three column matrix with the number of rows equal to the number of components in the vector. Each row in the matrix independently configures the *depth*, *fill_threshold*, and *full_period* of the FIFO buffer for the corresponding vector component.

Port Interface

Table 17-48 shows the port interface for the FIFO block.

Table 17-48. Port Interface for the FIFO Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
w	Input	Boolean	Write enable.	Yes	No
d	Input	Fixed-point	Data.	Yes	Yes
r	Input	Boolean	Read acknowledge.	Yes	No
v	Output	Boolean	Valid.	Yes	No
q	Output	Fixed-point	Data.	Yes	Yes
t	Output	Boolean	Fill threshold.	Yes	No
f	Output	Boolean	Fullness.	Yes	No

You can to set *fill_threshold* to a low number (<3) and arrive at a state such that output *t* is high and output *v* is low, because of differences in latency across different pairs of ports—from *w* to *v* is 3 cycles, from *r* to *t* is 1 cycle, from *w* to *t* is 1 cycle. If this situation arises, do not send a read acknowledgement to the FIFO buffer. Ensure that when the *v* output is low, the *r* input is also low, otherwise a warning appears in the MATLAB command window. If the read acknowledgement is derived from a feedback from the *t* output, ensure that the *fill_threshold* is set to a sufficiently high number (3 or above). Likewise for the *f* output and the *full_period*.

You may supply vector data to the *d* input, and vector data on the *q* output is the result. DSP Builder does not support vector signals on the *w* or *r* inputs, and the behavior is unspecified. The *v*, *t*, and *f* outputs are always scalar.

Floating-point Classifier (FloatClass)

The *FloatClass* block indicates whether a floating-point input is equal to zero, is signed (negative), is infinity, or is equal to not a number.

Table 17-49 shows the port interface.

Table 17-49. Port Interface for the FloatClass Block

Signal	Direction	Vector Data Support	Complex Data Support	Description
x	Input	—	—	—
inf	Output	—	—	Infinity output.
nan	Output	—	—	Not a number output.
sig	Output	—	—	Signed negative output.
zer	Output	—	—	Zero output.

Floating-point Multiply Accumulate (MultAcc)

The MultAcc block instantiates an Arria 10 DSP block in multiply-accumulate mode. This block only works on Arria10 devices and supports a hardware single-precision multiply accumulate structure. The block latency is 4 cycles.

Parameters

Table 17-50 shows the parameters .

Table 17-50. Parameters for the MultAcc Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Function	Accumulate (fpAcc) or multiply accumulate (fpMultAcc). When you select fpMultAcc , the block flushes denormalized numbers to zero on inputs and output; when you select fpAcc , the block flushes subnormal numbers to zero on inputs and output.

The fpMultAcc function implements the following equation:

$$q_n = (acc \& q_{(n-1)}) + x * y$$

- When acc is high (1) the result is equal to the sum between the previous accumulated result and the product $x*y$.
- When acc is low (0) the output value is the product value $x*y$

The fpAcc function implements the following equation:

$$q_n = (acc \& q_{(n-1)}) + x$$

- When acc is high the result consists of the sum between the previous accumulated result and the input x .
- When acc is low the x input is forwarded to the output q .
- Subnormal numbers are flushed to zero on inputs and output

Port Interface

Table 17-51 shows the port interface.

Table 17-51. Port Interface for the MultAcc Block

Signal	Direction	Vector Data Support	Complex Data Support	Description
x	Input	—	—	—
y	Input	—	—	Multiply accumulate only. Tied to 1 for the accumulate parameter.

Table 17–51. Port Interface for the MultAcc Block

Signal	Direction	Vector Data Support	Complex Data Support	Description
acc	Input	—	—	—
q	Output	—	—	—

ForLoop

The ForLoop block extends the basic loop, providing a more flexible structure that implements all common loop structures—for example, triangular loops, parallel loops, and sequential loops.

Each ForLoop block manages a single counter with a token-passing scheme that allows you to link these counters in a variety of ways.

Each ForLoop block has a static loop test parameter, which may be \leq , $<$, $>$ or \geq . Loops that count up should use \leq or $<$, depending on whether you consider the limit value, supplied by the limit signal, is within the range of the loop. Loops that count down should use \geq or $>$.

Port Interface

Table 17–65 shows the port interface for the ForLoop block.

The latency of the ForLoop block is non-zero. At loop end detection there are some cycles that may be invalid overhead required to build nested loop structures. The second activation of an inner loop does not necessarily begin immediately after the end of the first activation.

Table 17–52. Port Interface for the ForLoop Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
bs	Output	Boolean	Token-passing inputs and outputs. The four signals <i>ls</i> (loop start), <i>bs</i> (body start), <i>bd</i> (body done) and <i>ld</i> (loop done) pass a control token between different ForLoop blocks, to create a variety of different control structures.	Yes	No
bd	Input	Boolean		Yes	No
ld	Output	Boolean		Yes	No
ls	Input	Boolean	<p>When the <i>ls</i> port receives a token, the ForLoop block initializes. The loop counter is set to its initial value (that the <i>i</i> signal specifies). When the <i>bd</i> port receives a token, the step value (<i>s</i>) increments the loop counter. In either case, the new value of the counter is compared with the limit value (<i>l</i>) with the statically-configured loop test.</p> <p>If the loop test passes, the ForLoop block outputs the control token on the <i>bs</i> port to initiate the loop body and the valid signal, <i>v</i>, becomes active. If the loop test fails, the ForLoop block outputs the control token on <i>ld</i> port to indicate that the loop is complete and <i>v</i> becomes inactive.</p> <p>The ForLoop block becomes active when it receives a token on its <i>ls</i> port, and remains active until it finally outputs a token on its <i>ld</i> port. Changing any of the loop parameterization inputs (<i>i</i>, <i>s</i>, or <i>l</i>) while the loop is active, is not supported and produces unpredictable results.</p>	Yes	No
c	Output	Derived unsigned fixed-point type	The signal <i>c</i> is the count output from the loop. Its value is reliable only when the valid signal, <i>v</i> , is active	Yes	No
e	Input	Boolean	Use the enable input, <i>e</i> , to suspend and resume operation of the ForLoop block. When you disable the loop, the valid signal, <i>v</i> , goes low but DSP Builder makes no changes to the internal state of the block. When you re-enable the block, it resumes counting from the state at which you suspended it.	Yes	No

Table 17–52. Port Interface for the ForLoop Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
i	Input	Any unsigned fixed-point type	Loop parameterization inputs. The signals i, s, and l set the initial value, step and limit value (respectively) of the loop. Use with the loop test parameter, to control the operation of the loop. The loop parameter signals must be held constant while the loop is active, but you may them when the loop is inactive. Different activations of a ForLoop block can have different start or end points, which is useful for creating nested triangular loops, for example.	Yes	No
s	Input	Any unsigned fixed-point type		Yes	No
l	Input	Any unsigned fixed-point type		Yes	No
e1	Output	Boolean	Auxiliary loop outputs: the signals f1 and l1 are active on the first loop iteration and last loop iteration, respectively. The signal e1 is active when the ForLoop block is processing an empty loop.	Yes	No
f1	Output	Boolean		Yes	No
l1	Output	Boolean		Yes	No

General Purpose Input (GPIn)

The GPIn block models a general purpose input to a synthesizable subsystem. It is similar to the [Channel In \(ChannelIn\)](#) block but has no valid or channel inputs.

If the signal width is greater than one, you can assume the multiple inputs are synchronized.



Parameters

[Table 17–53](#) shows the parameters for the GPIn block.

Table 17–53. Parameters for the GPIn Block

Parameter	Description
Number of data signals	Specifies the number of input and output signals.

Port Interface

[Table 17–54](#) shows the port interface for the GPIn block.

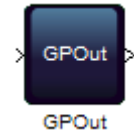
Table 17–54. Port Interface for the GPIn Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a, b, ...	Input	Any fixed-point type	Operands 1 to n	Yes	Yes
a, b, ...	Output	Same type as input	Data is passed through unchanged.	Yes	Yes

General Purpose Output (GPOut)

The GPOut block models a general purpose output to a synthesizable subsystem. It is similar to the [Channel Out \(ChannelOut\)](#) block but has no valid or channel inputs.

If the width is greater than one, the multiple outputs generate and are synchronized.



Parameters

[Table 17-55](#) shows the parameters for the GPOut block.

Table 17-55. Parameters for the GPOut Block

Parameter	Description
Number of data signals	Specifies the number of input and output signals.

Port Interface

[Table 17-56](#) shows the port interface for the GPOut block.

Table 17-56. Port Interface for the GPOut Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a, b, ...	Input	Any fixed-point type	Operands 1 to n	Yes	Yes
a, b, ...	Output	Same type as input	Data is passed through unchanged.	Yes	Yes

Load Exponent (LdExp)

[Table 17-57](#) shows the functions for the LdExp block.

Table 17-57. Functions for the LdExp Block

Function	Description
ikogb(a)	Outputs the integer logarithm of input, a, to the base 2. $q = \text{floor}(\log_2 a)$.
ldexp(a,b)	outputs the first input, a, scaled by 2 raised to the power of the second input, b. $q = a \cdot 2^b$.

Parameters

The **Function mask** parameter selects either ldexp or ilogb. The number of input ports on the block change according to the number of operands.

Port Interface

Table 17-58 shows the port interface for the LdExp block.

Table 17-58. Port Interface for the LdExp Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Single or double	Operand 1	Yes	No
b	Input	Integer	Operand 2	Yes	No
q	Output	floating-point for ldexp; Integer for ilogb	Result	Yes	No

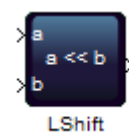
Left Shift (LShift)

The LShift block outputs the left shifted version of the input value.

The shift is specified by the input b:

$$q = (a \ll b)$$

The width of the data type a determines the maximum size of the shift. Shifts of more than the input word width result in an output of 0.



Parameters

Table 17-59 shows the parameters for the LShift block.

Table 17-59. Parameters for the LShift Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-59:

(1) If you use the LShift block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-60 shows the port interface for the LShift block.

Table 17-60. Port Interface for the LShift Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Loadable Counter (LoadableCounter)

The LoadableCounter block maintains a counter that you can reload with new parameters as needed in-circuit. The value of the counter increments by the step value every cycle for which the enable input is high. If the counter exceeds or equals the modulo value, or underflows in the case of a negative step value, it wraps around to zero or the value minus the modulo value as applicable. The current counter value is always available from the block's only output.

Internal registers hold the value, modulo, and step size of the counter. The values of these registers on reset are parameters that you can set on the block. Additionally, you can reload these registers with new values in-circuit by raising the ld signal high. While ld is high, DSP Builder writes the values of the i, s, and m input signals into the value, step, and modulo registers, respectively. The value of i passes through to the counter output. When ld falls low again, the counter resumes its normal operation starting from these new values.

If the initial or step values exceed the modulo value, the behavior is undefined. Using signed step values increases logic usage in hardware.

Table 17-61 lists the parameters for the LoadableCounter block.

Table 17-61. Parameters

Parameter	Description
Counter setup	A vector that specifies the counter settings on reset in the following format: [<initial value> <modulo> <step size>]

Table 17-62 lists the signals for the LoadableCounter block.

Table 17-62. Signals

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
en	Input	Boolean	Enable the counter.	Yes	No
ld	Input	Boolean	Load the counter.	Yes	No
i	Input	Any unsigned integer	New initial value to load.	Yes	No
s	Input	Any integer	New step value to load.	Yes	No
m	Input	Any non-zero unsigned integer	New modulo value to load.	Yes	No
q	Output	Unsigned integer	Counter value.	Yes	No

Look-Up Table (Lut)

The `Lut` block outputs the contents of a look-up table, indexed by the input.

$$q = \text{LUT}[a]$$

The size of the table determines the size of the initialization arrays.



Parameters

Table 17-63 shows the parameters for the `Lut` block.

Table 17-63. Parameters for the Lut Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via back projection: a downstream block that this block drives determines the output data type. If the driven block does not propagate a data type to the driver, you must use a Simulink SameDT block to copy the required data type to the output wire. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Single: single-precision floating-point data. ■ Double: double-precision floating-point data. ■ Variable precision floating point: variable precision floating-point output type
Output data type	Specifies the output data type. For example, sfix(16) , uint(8) .
Output scaling value	Specifies the output scaling value. For example, 2⁻¹⁵ .
Output value map	Specifies the location of the output values. For example, round([0;254]/17) . This parameter may also be a fi object when specifying data of arbitrarily high precision.
Floating point precision	Specifies the floating-point precision. For example, float32_m23 .

Note to Table 17-63:

(1) If you use the `Lut` block in a loop, you must specify the output type for at least one block in the loop.

You can specify the contents of the `Lut` block in one of the following ways:

- Specify table contents with a single row or column vector. The length of the 1D row or column vector determines the number of addressable entries in the table. If DSP Builder reads vector data from the table, all components of a given vector share the same value.
- When a look-up table contains vector data, you can provide a matrix to specify the table contents. The number of rows in the matrix determines the number of addressable entries in the table. Each row specifies the vector contents of the corresponding table entry. The number of columns must match the vector length, otherwise DSP Builder issues an error.



The default initialization of the LUT is a row vector `round([0:255]/17)`. This vector is inconsistent with the default for the DualMem block, which is a column vector `[zeros(16, 1)]`. The latter form is consistent with the new matrix initialization form in which the number of rows determines the addressable size.

Every block in the ModelPrim library accepts double-precision floating-point values when specifying mask parameters. This format limits precision to no more than 53 bits, which is more than sufficient for most of the blocks. For higher precision, the Const, DualMem, or LUT blocks optionally accept values using Simulink's Fixed Point data type. For example:

```
constValue = fi(0.142, 1, 16, 15)
vectorValue = fi(sin([0:10]'), 1, 18, 15)
```

To configure a Const, DualMem, or LUT with data of precision higher than IEEE double precision, create a MATLAB fi object of the required precision that contains the high precision data. Avoid truncation when creating this object. Use the fi object to specify the **Value** of the Const, the **Initial Contents** of the DualMem block, or the **Output value map** of the LUT block.

Port Interface

Table 17-64 shows the port interface for the Lut block.

Table 17-64. Port Interface for the Lut Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand	Yes	No
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes (with output value map)

Loop

The Loop block maintains a set of counters that implement the equivalent of a nested for loop in software. The counted values range from 0 to limit values provided with an input signal.

When the go signal is asserted on the g input, limit-values are read into the block with the c input. The dimension of the vector determines the number of counters (nested loops). When DSP Builder enables the block with the e input, it presents the counter values as a vector value at the q output each cycle. The valid output is set to 1 to indicate that a valid output is present.

There are vectors of flags indicating when first values (output f) and last values (output l) occur.

A particular element in these vector outputs is set to 1 when the corresponding loop counter is set at 0 or at *count-1* respectively.

Use the Loop block to drive datapaths that operate on regular data either from an input port or data stored in a memory. The enable input, and corresponding valid output, facilitate forward flow control.

For a two dimensional loop the equivalent C++ code to describe the general loop is:

```

for (int i = 0; i < c[0]; i++)
    for (int j = 0; j < c[1]; j++) {
        q[0] = i;
        q[1] = j;
        f[0] = (i==0);
        f[1] = (j==0);
        l[0] = (i==(c[0]-1));
        l[1] = (j==(c[1]-1));
    }

```

Port Interface

Table 17-65 shows the port interface for the Loop block.

Table 17-65. Port Interface for the Loop Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
g	Input	Boolean	Go.	Yes	No
c	Input	Unsigned integer	Counter limit values.	Yes	No
e	Input	Boolean	Enable.	Yes	No
v	Output	Boolean	Valid.	Yes	No
q	Output	Unsigned integer	Counter output values.	Yes	No
f	Output	Boolean	First value flags.	Yes	No
l	Output	Boolean	Last value flags.	Yes	No

Math

The Math block applies a mathematical operation to its floating-point inputs and outputs the floating-point result. A mask parameter popup menu selects the required elementary mathematical function that DSP Builder applies.

Table 17-66 shows the functions for the Math block.

Table 17-66. Functions for the Math Block

Function	Description
exp(x) ⁽¹⁾	<i>e</i> raised to the exponent x.
exp2(x)	2 raised to the exponent x.
exp10(x)	10 raised to the exponent x.
expm1(x)	exp(x) – 1.
inverse(x)	The reciprocal of x. For Floating-point rounding , select either correct or faithful .
hypot(x,y)	Hypotenuse of right-angled triangle with other two sides length x and y.
hypot3d	The Euclidean norm of a vector in a three-dimensional space.
log(x) ⁽¹⁾	Natural logarithm.
log2(x)	Logarithm of x to the base 2.
log10(x)	Logarithm of x to the base 10.
log1p(x)	log(x+1).

Table 17-66. Functions for the Math Block

Function	Description
pow(x,y)	x raised to the power of y.
powr(x,y)	x raised to the power of y, where l is non-negative.
mod(x,y)	$(x - n \times y)$ where $n = y/x$ rounded toward zero.
Note: (1) For single-precision input and designs targeting Arria 10 devices, the block uses a mixture of resources including the Arria10 DSP blocks in floating-point mode. This implementation uses fewer ALMs at the expense of more DSP blocks.	

Parameters

The **Function** mask parameter selects one of five elementary functions. The number of input ports on the block change as required by the semantics of the function that you select:

- One-input function: exp(x), exp2(x), exp10(x), expm1(x), log(x), log2(x), log10(x), log1p(x), inverse(x)
- Two-input functions: hypot(x,y), mod(x,y), pow(x,y), powr(x,y)
- Three input function: hypot3d(x,y,z)

Port Interface

Table 17-67 shows the port interface for the Math block.

Table 17-67. Port Interface for the Math Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Single or double	Operand 1	Yes	No
y	Input	Single or double	Operand 2 (hypot, mod, pow, and powr only)	Yes	No
z	Input	Single or double	Hypot3 only.	Yes	No
q	Output	Single or double	Result	Yes	No

Minimum and Maximum (MinMax)

The MinMax block allows you to select a bounding function to apply to the inputs.

Table 17-68 shows the functions for the MinMax block.

Table 17-68. Functions for the MinMax Block

Function	Data types	Description
max	Fixed- or floating-point	Outputs a if $a > b$, otherwise outputs b.
min	Fixed or floating-point	Outputs a if $a < b$, otherwise outputs b.
maxmag	Floating-point	Outputs a if $ a > b $, b if $ b > a $, otherwise max(a,b).

Table 17-68. Functions for the MinMax Block

Function	Data types	Description
minmag	Floating-point	Outputs a if $ a < b $, b if $ b < a $, otherwise $\min(a,b)$.
dim	Floating-point	Outputs $(a - b)$ if $a > b$, otherwise outputs 0.
sat	Floating-point	Saturate input a to interval $[c,b]$.

Parameters

The **Function** mask parameter selects one of six bounding functions. The number of input ports on the block change as required by the semantics of the function that you select:

- Two-input functions: max, min, maxmag, minmag, dim
- Three-input functions: sat

The **Output data type mode** mask parameter applies only if the input is fixed-point format.

Port Interface

Table 17-69 shows the port interface for the MinMax block.

Table 17-69. Port Interface for the MinMax Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
	Input	Fixed-point, single or double	Operand 1	Yes	No
b	Input	Fixed-point, single or double	Operand 2	Yes	No
c	Input	Single or double	Operand 3 (saturate only)	Yes	No
q	Output	Single or double	Result	Yes	No

MinMaxCtrl

The MinMaxCtrl block applies a minimum or maximum operator to the inputs depending on the Boolean signal it receives on the control port.

Table 17-70 shows the functions for the MinMaxCtrl block.

Table 17-70. Functions of the MinMaxCtrl Block

s	Function	Description
0	Minimum	Output a if $a < b$, otherwise output b.
1	Maximum	Output a if $a > b$, otherwise output b.

Parameters

The **Output data type mode** mask parameter applies only if the inputs a and b are fixed-point format.

Table 17-71 shows the parameters for the MinMaxCtrl block.

Table 17-71. Parameters for the MinMaxCtrl Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available. This option reinterprets the output bit pattern from the LSB up according to the specified type. ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, sfix(16), uint(8).
Output scaling value	Specifies the output scaling value. For example, 2 ⁻¹⁵ .

Port Interface

Table 17-72 shows the port interface for the MinMaxCtrl block.

Table 17-72. Port Interface for the MinMaxCtrl Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
	Input	Fixed-point, single or double	Operand 1	Yes	No
b	Input	Fixed-point, single or double	Operand 2	Yes	No
s	Input	Boolean	Select input	Yes	No
q	Output	Fixed-point, single or double	Result	Yes	No

Multiply (Mult)

The Mult block outputs the product of the inputs:

$$q = a \times b$$



For single-precision inputs and designs targeting Arria 10 devices, the block uses a mixture of resources including the Arria10 DSP blocks in floating-point mode.

Parameters

Table 17-73 shows the parameters for the `Mult` block.

Table 17-73. Parameters for the Mult Block

Parameter	Description
Output data type mode	Determines how the block sets its output data type: <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Variable precision floating point: variable precision floating-point output type
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Floating point precision	Specifies the floating-point precision. For example, <code>float32_m23</code> .

Note to Table 17-73:

(1) If you use the `Mult` block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-74 shows the port interface for the `Mult` block.

Table 17-74. Port Interface for the Mult Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

Multiplexer (Mux)

The `Mux` block allows a variable number of inputs and outputs the selected input, or zero if the select value is invalid (outside the number of data signals).



You can make a multiple input multiplexer by combining more than one `mux2` blocks in a tree or by using a [Select](#) block.

Parameters

Table 17-75 shows the parameters for the Mux block.

Table 17-75. Parameters for the Mux Block

Parameter	Description
Number of data signals	The input type for s is an unsigned integer of width $\log_2(\text{number of data signals})$. Boolean is also allowed in the case of two data inputs.
Output data type mode	Determines how the block sets its output data type: <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-75:

(1) If you use the Mux block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-76 shows the port interface for the Mux block.

Table 17-76. Port Interface for the Mux Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
s	Input	Any fixed- or floating-point type	Select input	Yes	No
0	Input	Any fixed- or floating-point type	Input 0	Yes	Yes
1	Input	Any fixed- or floating-point type	Input 1	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

NAND Gate (Nand)

The Nand block outputs the logical NAND of the input values:

$$q = \sim(a \ \& \ b)$$

If the number of inputs is set to 1, then output the logical NAND of all the individual bits of the input word.



Parameters

Table 17-77 shows the parameters for the Nand block.

Table 17-77. Parameters for the Nand Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-77:

(1) If you use the Nand block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-77 shows the port interface for the Nand block.

Table 17-78. Port Interface for the Nand Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Negate

The Negate block outputs the negation of the input value.

Parameters

The **Output datatype mode** determines how the block infers its output data type:

- **Inherit via internal rule.** The output data type is the same as the input data type.
- **Inherit via internal rule with word growth.** The output data type is the same as the input data type. If the input data type is fixed-point, word growth is applied to the output data type.

Port Interface

Table 17-83 shows the port interface for the Negate block.

Table 17-79. Port Interface for the Not Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand	Yes	Yes
q	Output	Any fixed- or floating-point type	Result	Yes	No

NOR Gate (Nor)

The Nor block outputs the logical NOR of the input values:

$$q = \sim(a \mid b)$$

Set the number of inputs to 1, to output the logical NOR of all the individual bits of the input word.



Parameters

Table 17-80 shows the parameters for the Nor block.

Table 17-80. Parameters for the Nor Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-80:

(1) If you use the Nor block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17–81 shows the port interface for the Nor block.

Table 17–81. Port Interface for the Nor Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

NOT Gate (Not)

The Not block outputs the logical NOT of the input value:

$$q = \sim a$$



Parameters

Table 17–82 shows the parameters for the Not block.

Table 17–82. Parameters for the Not Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17–82:

(1) If you use the Not block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17–83 shows the port interface for the Not block.

Table 17–83. Port Interface for the Not Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

OR Gate (Or)

The Or block outputs the logical OR of the input values:

$$q = a \mid b$$

Set the number of inputs to 1, to output the logical OR of all the individual bits of the input word.



Parameters

Table 17-84 shows the parameters for the Or block.

Table 17-84. Parameters for the Or Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-84:

(1) If you use the Or block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-85 shows the port interface for the Or block.

Table 17-85. Port Interface for the Or Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Polynomial

The Polynomial block takes input x , and provides the result of evaluating a polynomial of degree, n :

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

Parameters

Table 17-86 shows the parameters for the Polynomial block.

Table 17-86. Parameters for the Polynomial Block

Parameter	Description
Coefficient banks	<p>A vector of $(n + 1)$ components. Specify the coefficients in the order $a_0, a_1, a_2, \dots, a_n$.</p> <p>If input x is driven by a vector signal, then a matrix with $(n+1)$ columns, and one row per vector component can be specified. Each output component will be the result of evaluating an independently defined polynomial of degree, n.</p> <p>If there is more than one coefficient bank, the number of rows in the matrix should be $v * u$, for v vector components, and u banks. The coefficients for a given bank are ordered contiguously in the matrix.</p>
Number of coefficient banks	<ul style="list-style-type: none"> ■ Set to the default value of 1, for only one input, x. ■ Set to greater than 1, for a second input, b, to specify which bank of coefficients DSP Builder uses to evaluate the polynomial.

Port Interface

Table 17-87 shows the port interface for the Polynomial block.

Table 17-87. Port Interface for the Polynomial Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Floating-point	Data	Yes	No
b	Input	Integer	Bank selector	No	No
q	Output	Floating-point	Data	Yes	No

Ready

Use the Ready block in designs with ALU folding. The Ready block adds a ready signal to your design.

Reinterpret Cast (ReinterpretCast)

The ReinterpretCast block outputs the same bit pattern that it reads on its input port, but casts it to a data type that you specify with the block parameters. This data type should use the same number of bits as the bit width of the input signal.

Parameters

Table 17-88 shows the parameters for the `ReinterpretCast` block.

Table 17-88. Parameters for the ReinterpretCast Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via back projection: a downstream block that this block drives determines the output data type. If the driven block does not propagate a data type to the driver, you must use a Simulink SameDT block to copy the required data type to the output wire. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Single: single-precision floating-point data. ■ Double: double-precision floating-point data. ■ Variable precision floating point: variable precision floating-point output type
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Floating point precision	Specifies the floating-point precision. For example, <code>float32_m23</code> .

Note to Table 17-88:

(1) If you use the `ReinterpretCast` block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-89 shows the port interface for the `SampleDelay` block.

Table 17-89. Port Interface for the ReinterpretCast Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand	Yes	Yes
q	Output	User specified	Result	Yes	Yes

Round

The Round block applies a rounding operation to the floating-point input. A mask parameter popup menu selects the required rounding function that you apply.

Table 17-90 shows the functions for the Round block.

Table 17-90. Functions for the Round Block

Function	Description
ceil(x)	Lowest integer not less than input x.
floor(x)	Highest integer not exceeding input x.
rint(x)	Round to nearest integer; halfway cases rounded to even number.
round(x)	Round to nearest integer; halfway cases rounded away from zero.

Parameters

The **Function** mask parameter selects one of four rounding functions.

Port Interface

Table 17-91 shows the port interface for the Round block.

Table 17-91. Port Interface for the Round Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Single or double	Operand	Yes	Yes
q	Output	Single or double	Result	Yes	Yes

Sample Delay (SampleDelay)

The SampleDelay block outputs a delayed version of the input.



Parameters

Table 17-92 shows the parameters for the SampleDelay block.

Table 17-92. Parameters for the SampleDelay Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Single: single floating-point data. ■ Double: double floating-point data.
Output data type	Specifies the output data type. For example, <code>sfixed(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Number of delays	Specifies the number of samples to delay.
Minimum delay	Checks if the delay can grow as needed, so that the specified length becomes the lower bound.
Equivalence group	Sample delays that share the same equivalence group string grow by the same increment.

Note to Table 17-92:

(1) If you use the SampleDelay block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-93 shows the port interface for the SampleDelay block.

Table 17-93. Port Interface for the SampleDelay Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Data input	Yes	Yes
q	Output	Derived fixed- or floating-point type	Data output	Yes	Yes

Scalar Product

The Scalar Product block accepts two vector inputs of the same dimension and produces the inner product on the output. If one or more inputs are complex, the output is complex. If one of the inputs is a scalar signal, the same factor scales all vector components of the other input port.



For single-precision inputs and designs targeting Arria 10 devices, the block uses a mixture of resources including the Arria10 DSP blocks in floating-point mode.

Parameters

Table 17-96 shows the parameters for the Scalar Product block.

Table 17-94. Parameters for the Select Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean. ■ Variable precision floating point: variable precision floating-point output type.
Output data type	Specifies the output data type. For example, <code>fixdt(1, 16, 15)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Floating-point precision	Specifies a predefined floating-point type.
Fused datapath	<p>Determines what evaluation strategy the generated HDL uses:</p> <ul style="list-style-type: none"> ■ None: evaluates the result using IEEE floating-point operations. The result is closest to the Simulink simulation model as it also uses IEEE operations. ■ Opt1, Opt2, Opt3, Opt4: evaluates the result using fused datapath internal types, which have significantly more bits of precision, but the result deviates more from the simulation model.

Port Interface

Table 17-97 shows the port interface for the Scalar Product block.

Table 17-95. Port Interface for the Select Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	No	Yes

Performance Tradeoffs

The following graphs show the effect of synthesizing scalar products with different fused datapath values. The column labelled correct is correctly rounded IEEE floating-point values.

Figure 17-2 shows the trade-off between logic usage and accuracy for the different fused datapath options. The scalar product is applied to single-precision vectors of length 8. The design targets a Stratix V (C2) device and an f_{MAX} of 300 MHz. The input data is from a Gaussian distribution with mean of zero and variance of 1.0. The accuracy is the root mean squared error when compared against a scalar product result that used 1,000 bits of precision.

Figure 17-2. Logic versus Accuracy

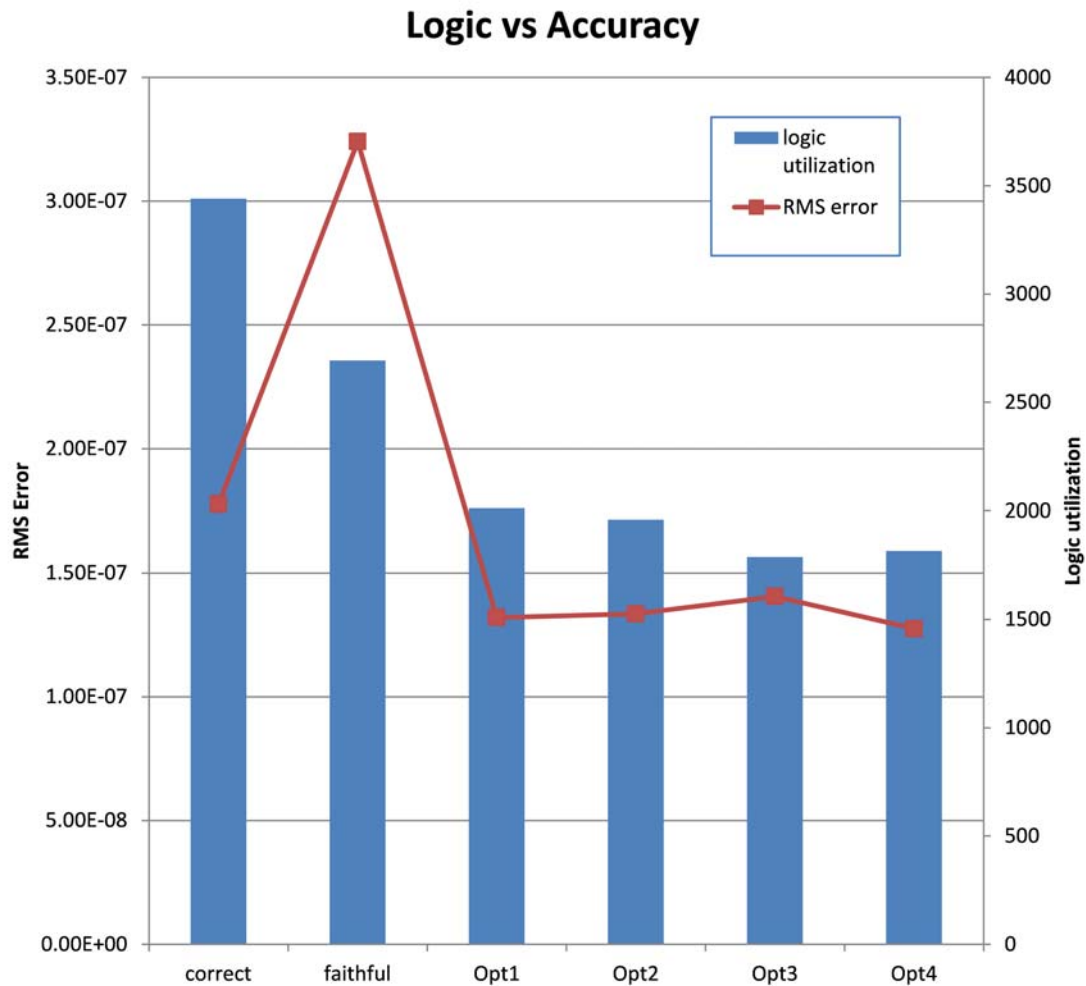
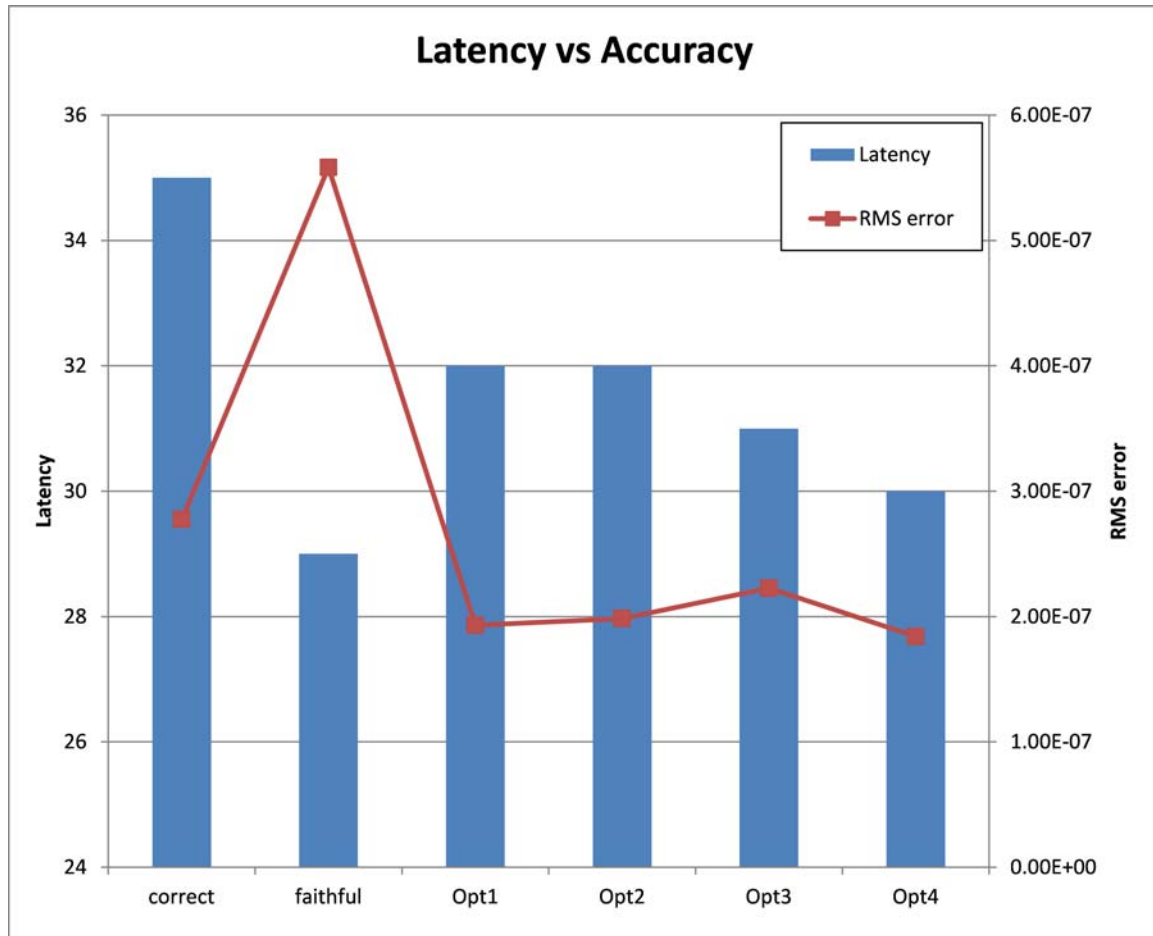


Figure 17-2 shows the trade-off between latency across the block and accuracy for the different fused datapath options. The scalar product is applied to single-precision vectors of length 16. The Stratix V device (C2) targets an f_{MAX} of 300 MHz.

Figure 17-3. Latency versus Accuracy



Select

The Select block outputs one of the data signals (a, b, ...) if its paired select input (0, 1, ...) has a non-zero value.

$$q = 0 ? a : (1 ? b : d)$$

If all select inputs are 0, the Select block outputs the default value d. At most one select input should be high at a time.



Parameters

Table 17-96 shows the parameters for the Select block.

Table 17-96. Parameters for the Select Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Number of cases	Specifies the number of non-default data inputs.

Note to Table 17-96:

(1) If you use the Select block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-97 shows the port interface for the Select block.

Table 17-97. Port Interface for the Select Block

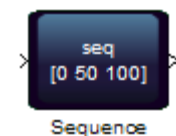
Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
d	Input	Any fixed- or floating-point type	Default input	Yes	Yes
0, 1, 2, ...	Input	Boolean	One-hot select inputs	Yes	No
a, b, c, ...	Input	Any fixed- or floating-point type	Data input	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

Sequence

The Sequence block outputs a Boolean pulse of configurable duration and phase.

The input acts as an enable for this sequence. Usually, this block initializes with an array of Boolean pulses of length *period*. The first *step_value* entries are zero, and the remaining values are one.

A counter steps along this array, one entry at a time, and indexes the array. The output value is the contents of the array. The counter is initialized to *initial_value*. The counter wraps at step *period*, back to zero, to index the beginning of the array.



Parameters

Table 17-98 shows the parameters for the Sequence block.

Table 17-98. Parameters for the Sequence Block

Parameter	Description
Output data type mode	Determines how the block sets its output data type: <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Sequence setup	A vector that specifies the counter in the format: [<i>initial_value</i> < <i>step_value</i> > < <i>period</i> >] For example, [0 50 100]

Note to Table 17-98:

(1) If you use the Sequence block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-99 shows the port interface for the Sequence block.

Table 17-99. Port Interface for the Sequence Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Boolean	Sequence enable	Yes	No
q	Output	Boolean	Result	Yes	No

Shift

The Shift block outputs the logical right shifted version of the input value if unsigned, or outputs the arithmetic right shifted version of the input value if signed. The shift is specified by the input b:

$$q = (a \gg b)$$

The width of the data type b determines the maximum size of the shift.

Shifts of more than the input word width result in an output of 0 for non-negative numbers and $(0 - 2^{-F})$ for negative numbers (where F is the fraction length).



Parameters

Table 17-100 shows the parameters for the Shift block.

Table 17-100. Parameters for the Shift Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-100:

(1) If you use the Shift block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-101 shows the port interface for the Shift block.

Table 17-101. Port Interface for the Shift Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Unsigned integer	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

Sqrt

The Sqrt block applies a numerical root operation to its floating-point input and produces the floating-point result. The mask parameter popup menu selects the required root function that you apply.

Table 17-102 shows the functions for the Sqrt block.

Table 17-102. Functions for the Sqrt Block

Function	Description
<code>cbrt(x)</code>	Cube root.
<code>recipsqrt(x)</code>	Reciprocal square root.
<code>sqrt(x)</code>	Square root.

Parameters

The **Function** mask parameter selects one of the three numerical root functions. For **floating-point rounding**, select either **correct** or **faithful**.

Port Interface

Table 17-103 shows the port interface for the Sqrt block.

Table 17-103. Port Interface for the Sqrt Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Single or double	Operand	Yes	No
q	Output	Single or double	Result	Yes	No

Subtract (Sub)

The Sub block outputs the difference between the inputs:

$$q = a - b.$$



For single-precision inputs and designs targeting Arria 10 devices, the block uses a mixture of resources including the Arria10 DSP blocks in floating-point mode.

Parameters

Table 17-104 shows the parameters for the Sub block.

Table 17-104. Parameters for the Sub Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that you can store in the number of bits of the input. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter <i>Volume 3: DSP Builder Advanced Blockset</i> of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-104:

(1) If you use the Sub block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-105 shows the port interface for the Sub block.

Table 17-105. Port Interface for the Sub Block


Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed- or floating-point type	Operand 1	Yes	Yes
b	Input	Any fixed- or floating-point type	Operand 2	Yes	Yes
q	Output	Derived fixed- or floating-point type	Result	Yes	Yes

Synthesis Information (SynthesisInfo)

The SynthesisInfo block controls the synthesis flow for the current model.

The inputs and outputs to this subsystem become the primary inputs and outputs of the RTL entity that DSP Builder creates. After you run a Simulink simulation, the online Help page for the SynthesisInfo block updates to show the latency, and port interface for the current ModelPrim subsystem.




 The SynthesisInfo block can be at the same level as the Device block (if the synthesizable subsystem is the same as the generated hardware subsystem). However, it is often convenient to create a separate subsystem level that contains the Device block. Refer to the design examples for some examples of design hierarchy.

Two styles of operation exist during synthesis: **WYSIWYG** and **Scheduled**.

WYSIWYG Style

WYSIWYG is the default style of operation. Use when you want full control over the pipelining in a system. A SampleDelay ModelPrim block must follow every ModelPrim block that requires registering (such as an adder or multiplier). The preceding block absorbs the delay ModelPrim block to satisfy its delay requirements. DSP Builder issues error messages if insufficient delay is present.

 The ModelPrim logic blocks (And, Or, Nand, Not, Nor, Xnor, Xor) and the Simulink Mux and Demux blocks do not require a register.

Scheduled Style

The **Scheduled** style of operation uses a pipelining and delay distribution algorithm that creates fast hardware implementations from an easily described untimed block diagram. This style takes full advantage of the automatic pipelining capability.

 Altera recommends the **Scheduled** style of operation.

The algorithm performs the following operations:

1. Reads in and flattens your design example for any subsystem that contains a `SynthesisInfo` block.
2. Builds an internal graph to represent the logic.
3. Based on the absolute clock frequency requested, adds enough pipeline stages to meet that clock frequency. For example, you may pipeline long adders into several shorter adders. This additional pipelining helps reach high clock frequencies.

Parameters

Table 17-106 shows the parameters for the `SynthesisInfo` block.

Table 17-106. Parameters for the `SynthesisInfo` Block

Parameter	Description
Synthesis Style (1)	<p>You can select the following synthesis styles:</p> <ul style="list-style-type: none"> ■ Scheduled ■ WYYSIWYG
Constrain Latency (2)	<p>This option is available when the Scheduled synthesis style is selected and allows you to select the type of constraint and to specify its value. The specified value can be a workspace variable or an expression but must evaluate to a positive integer.</p> <p>You can select the following types of constraint:</p> <ul style="list-style-type: none"> ■ >: Greater than ■ >=: Greater than or equal to ■ =: Equal to ■ <=: Less than or equal to ■ <: Less than

Notes to Table 17-106:

- (1) For more information about these styles, refer to “ModelPrim Blocks in” *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.
- (2) For more information about constraining latency, refer to “Using latency Constraints” in *Volume 3: DSP Builder Advanced Blockset* of the *DSP Builder Handbook*.

Port Interface

The `SynthesisInfo` block has no inputs or outputs.

Updated Help

After you run a simulation, DSP Builder updates the help pages with specific information about each instance of a block. This updated help overrides the default help link. To find the updated help, click on the help link on the block after simulation. This updated help includes a link back to the help for the general block and the following information about the generated instance:

- Date and time of generation
- The latency introduced by this block.
- Port interface table.

Trig

The Trig block applies a trigonometric operation to its floating-point inputs and produces the floating-point result.

Table 17-107 shows the functions for the Trig block.



Your design may use up to 50% less resources if you use the pi functions.

Table 17-107. Functions for the Trig Block

Function	Description
acos(a)	Arc cosine (inverse cosine) output in radians.
asin(a)	Arc sine (inverse sine) output in radians.
atan(a)	Arc tangent (inverse tangent) output in radians.
acos(x)/pi	Arc cosine (inverse cosine) output as fraction of half circle.
asin(x)/pi	Arc sine (inverse sine) output as fraction of half circle.
atan(x)/pi	Arc tangent (inverse tangent) output as fraction of half circle.
atan2(y,x)	Four quadrant inverse tangent, output angle in interval $[-\pi; +\pi]$ radians.
cos(a)	Cosine of input in radians.
cos(pi*x)	Cosine of input angle specified as fraction of half circle.
cot(a)	Cotangent of input in radians.
cot(pi*x)	Cotangent of input angle specified as fraction of half circle.
sin(a)	Sine of input in radians.
sin(pi*x)	Sine of input angle specified as fraction of half circle.
sincos(a)	Outputs both sine and cosine of input a in radians.
tan(a)	Tangent of input in radians.
tan(pi*x)	Tangent of input angle specified as fraction of half circle.

Parameters

The **Function** parameter selects one of the 16 trigonometric functions. The number of input ports and output ports on the block change as required by the semantics of the function that you select:

- One-input and one-output: sin, cos, tan, cot, asin, acos, atan
- Two-inputs and one-output: atan2
- One-input and two-outputs: sincos

If you reduce the input range for the sin(x) and cos(x) functions to the interval $[-2\pi, 2\pi]$, and you target Arria10 devices, in **Advanced Options** set `struct('rangeReduction', 0)`. The design then uses the floating-point mode of the DSP blocks to build more efficient architectures.

Port Interface

Table 17-108 shows the port interface for the Trig block.

Table 17-108. Port Interface for the Trig Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
x	Input	Single or double	Operand 1 (Operand 2 of atan2)	Yes	No
y	Input	Single or double	Operand 1 of atan2	Yes	No
q	Output	Single or double	Result 1	Yes	No
r	Output	Single or double	Result 2 (Cosine output of sincos)	Yes	No

XNOR Gate (Xnor)

The Xnor block outputs the logical XNOR of the input values:

$$q = \sim(a \text{ XOR } b)$$

Set the number of inputs to 1, to output the logical XNOR of all the individual bits of the input word.



Parameters

Table 17-109 shows the parameters for the Xnor block.

Table 17-109. Parameters for the Xnor Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter in Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-109:

(1) If you use the Xnor block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-110 shows the port interface for the Xnor block.

Table 17-110. Port Interface for the Xnor Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

XOR Gate (Xor)

The Xor block outputs the logical XOR of the input values:

$$q = (a \text{ XOR } b)$$

Set the number of inputs to 1, to output the logical XOR of all the individual bits of the input word.



Parameters

Table 17-111 shows the parameters for the Xor block.

Table 17-111. Parameters for the Xor Block

Parameter	Description
Number of inputs	Specifies the number of inputs.
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. This option reinterprets the output bit pattern from the LSB up according to the specified type. For more information, refer to the <i>Design Flow</i> chapter Volume 3: DSP Builder Advanced Blockset of the <i>DSP Builder Handbook</i>. ⁽¹⁾ ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .

Note to Table 17-111:

(1) If you use the Xor block in a loop, you must specify the output type for at least one block in the loop.

Port Interface

Table 17-112 shows the port interface for the Xor block.

Table 17-112. Port Interface for the Xor Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand 1	Yes	No
b	Input	Any fixed-point type	Operand 2	Yes	No
q	Output	Derived fixed-point type	Result	Yes	No

The ModelPrim library contains the SumOfElements block that specifically operates on vector signals, rather than replicates functionality based on vector widths as do other primitive blocks. ModelPrim subsystems use this block with ModelPrim library blocks.

Sum of Elements (SumOfElements)

The SumOfElements block outputs the sum of the elements within its single data input.

$$q = \sum a_n$$

If the input is a scalar, the SumOfElements block outputs an unchanged value.

$$q = a$$



Parameters

Table 18–1 shows the parameters for the SumOfElements block.

Table 18–1. Parameters for the SumOfElements Block

Parameter	Description
Output data type mode	<p>Determines how the block sets its output data type:</p> <ul style="list-style-type: none"> ■ Inherit via internal rule: the number of integer and fractional bits is the maximum of the number of bits in the input data types. ■ Inherit via internal rule with word growth: the number of fractional bits is the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0, which exceeds the maximum positive number that you can store in the number of bits of the input. ■ Specify via dialog: you can set the output type of the block explicitly using additional fields that are available when this option is selected. ■ Boolean: the output type is Boolean.
Output data type	Specifies the output data type. For example, <code>sfix(16)</code> , <code>uint(8)</code> .
Output scaling value	Specifies the output scaling value. For example, 2^{-15} .
Number of Inputs	Specifies the number of inputs.

Port Interface

Table 18–2 shows the port interface for the SumOfElements block.

Table 18–2. Port Interface for the SumOfElements Block

Signal	Direction	Type	Description	Vector Data Support	Complex Data Support
a	Input	Any fixed-point type	Operand	Yes	Yes
q	Output	Derived fixed-point type	Result	No (scalar output only).	Yes

This ModelIP library contains a numerically controlled oscillator (NCO), complex mixer, and real mixer blocks.

Direct digital synthesis of accurate sine and cosine signals using a NCO is an essential part of radio receivers and transmitters converting to and from intermediate frequencies (IF). The NCO block can generate any number of sine and cosine signals at any fraction of the system clock frequency. Memory-mapped phase increments and offsets allow you to use the NCO flexibly in a system configuration. You can parameterize the NCO with arbitrary precision, to give over 150dB spurious-free dynamic range (SFDR) with 26-bit outputs.

The mixer blocks perform multiplications with real and complex inputs.

The Waveform Synthesis library contains the following blocks:

- **Complex Mixer (ComplexMixer)**
- **Real Mixer (Mixer)**
- **NCO**

The ComplexMixer block does not support the Simulink complex data-type. The in phase (real), and quadrature (imaginary) signals are input and output as separate signals.

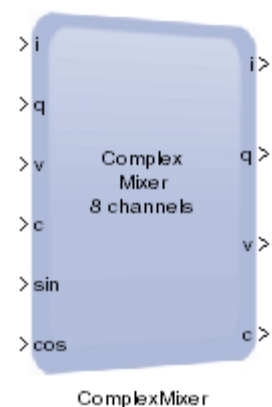
After you run a simulation, DSP Builder updates the help pages with specific information about each instance of a block. This updated help overrides the default help link. To find the updated help, click on the help link on the block after simulation. This updated help includes a link back to the help for the general block and the following information about the generated instance:

- Date and time of generation
- The latency introduced by this block.
- Port interface table.

Complex Mixer (ComplexMixer)

The ComplexMixer block performs a complex by complex multiply on streams of data and it splits the inputs and outputs into their real and imaginary components. This function frequency can shift a data stream in a digital up converter, where the first complex data is the *i* and *q* data and the second complex data is the cosine and sine data provided by an NCO.

The ComplexMixer block multiplies a complex input stream by a synchronized complex data stream, sample by sample.



You can use this block in a digital up converter for a radio system or a general purpose DSP application. The data has fixed-point types, and the output is the implied full precision fixed-point type.



You can easily replicate the ComplexMixer block with a ModelPrim Multiply block that takes complex inputs within a ModelPrim subsystem.

Parameters

The ComplexMixer performs element-by-element multiplication on n channels and m frequencies.

The system specification, including such factors as the channel count and sample rates, determines the main parameters for this block. The input sample rate of the block determines the number of channels present on each input wire and the number of wires:

$$\text{Number of Channels per wire} = \text{Clock_Rate} / \text{Sample_Rate}$$

$$\text{Number of Wires} = \text{ceiling}(\text{Chan_Count} \times \text{Sample_Rate} / \text{Clock_Rate})$$

For example, a sample rate of 60 MSPS and system clock rate of 240 MHz gives four samples to be TDM on to each input wire.

If there are more channels than TDM slots available on a wire, the input wire is a vector of sufficient width to hold all the samples. Similarly, the number of frequencies (the number of complex numbers) determines the width of the sine and cosine inputs. The number of results produced by the ComplexMixer is the product of the sample input vector and the frequency vector. The results are TDM on to the i and q outputs in a similar manner to the inputs.

Table 19–1 shows the parameters for the ComplexMixer block.

Table 19–1. Parameters for the ComplexMixer Block

Parameter	Description
Input Rate Per Channel (MSPS)	The data rate per channel measured in millions of samples per second.
Number of Complex Channels	The number of complex input channels.
Number of Frequencies	The number of complex frequencies in the multiplier.

Port Interface

Table 19–2 shows the port interface for the ComplexMixer block.

Table 19–2. Port Interface for the ComplexMixer Block

Signal	Direction	Description
i	Input	The real (in phase) half of the complex data input. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits inherits from the input wire.
q	Input	The imaginary (quadrature phase) half of the complex data input. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits inherits from the input wire.
v	Input	Indicates validity of data input signals. If v is high, the data on the a wire is valid.
c	Input	Indicates channel of data input signals. If v is high, c indicates the data channel data.
\sin	Input	The imaginary part of the complex number. For example, the NCO's sine output.

Table 19–2. Port Interface for the ComplexMixer Block

Signal	Direction	Description
cos	Input	The real part of the complex number. For example, the NCO's cosine output.
i	Output	The in-phase (real) output of the mixer, which is $(i \times \cos - q \times \sin)$. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is wide enough for the full precision result.
q	Output	The quadrature phase (imaginary) output of the mixer, which is $(i \times \sin + q \times \cos)$. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is wide enough for the full precision result.
v	Output	Indicates validity of data output signals.
c	Output	Indicates channel of data output signals.

The **ComplexMixer** block performs the multiplication on corresponding components; the **RealMixer** block does not. The **ComplexMixer** block uses modulo indexing if one vector is shorter than another. Hence, the output vector width is the maximum of the widths of the input vectors. The **RealMixer** block performs a full outer product on the input vectors. The number of components in the output vector is the product of the width of the input vectors for sin and cos (must be the same) and the width of the input vector for a.

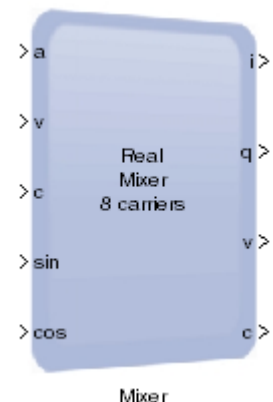
Real Mixer (Mixer)

The **Mixer** block performs a real by complex multiply on streams of data. This function creates quadrature data from an antenna input, where the real data is the antenna data and the complex data is the cosine and sine data provided by an NCO.

The **Mixer** block multiplies a real input stream by a synchronized complex data stream, sample by sample.

You can use the **Mixer** block in a digital down converter for a radio system or a general purpose DSP application. The data has fixed-point types, and the output is the implied full precision fixed-point type.

Also refer to the **ComplexMixer** block ("**Complex Mixer (ComplexMixer)**" on page 19–1).



You can easily replicate the **Mixer** block with a **ModelPrim Multiply** block that takes one real and one complex input within a **ModelPrim** subsystem.

Parameters

The **Mixer** performs element-by-element multiplication on n channels and m frequencies.

The system specification, including such factors as the channel count and sample rates, determines the main parameters for this block. The input sample rate of the block determines the number of channels present on each input wire and the number of wires:

$$\text{Number of Channels per wire} = \text{Clock_Rate} / \text{Sample_Rate}$$

$$\text{Number of Wires} = \text{ceiling}(\text{Chan_Count} \times \text{Sample_Rate} / \text{Clock_Rate})$$

For example, a sample rate of 60 MSPS and system clock rate of 240 MHz gives four samples to be TDM on to each input wire:

If there are more channels than TDM slots available on a wire, the input wire is a vector of sufficient width to hold all the samples. Similarly, the number of frequencies (the number of complex numbers) determines the width of the sine and cosine inputs. The number of results that the Mixer produces is the product of the sample input vector and the frequency vector. The results are TDM on to the *i* and *q* outputs in a similar way to the inputs.

Table 19-3 shows the parameters for the Mixer block.

Table 19-3. Parameters for the Mixer Block

Parameter	Description
Input Rate Per Channel (MSPS)	The data rate per channel measured in millions of samples per second.
Number of Channels	The number of real input channels.
Number of Frequencies	The number of real frequencies in the multiplier.

Port Interface

Table 19-4 shows the port interface for the Mixer block.

Table 19-4. Port Interface for the Mixer Block

Signal	Direction	Description
a	Input	The real data input to the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is inherited from the input wire.
v	Input	Indicates validity of data input signals. If <i>v</i> is high, the data on the <i>a</i> wire is valid.
c	Input	Indicates channel of data input signals. If <i>v</i> is high, <i>c</i> indicates the data channel.
sin	Input	The imaginary part of the complex number. For example, the NCO's sine output.
cos	Input	The real part of the complex number. For example, the NCO's cosine output.
i	Output	The in-phase (real) output of the mixer, which is (<i>a</i> × <i>cos</i>). If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is wide enough for the full precision result.
q	Output	The quadrature phase (imaginary) output of the mixer, which is (<i>a</i> × <i>sin</i>). If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is wide enough for the full precision result.
v	Output	Indicates validity of data output signals.
c	Output	Indicates channel of data output signals.

NCO

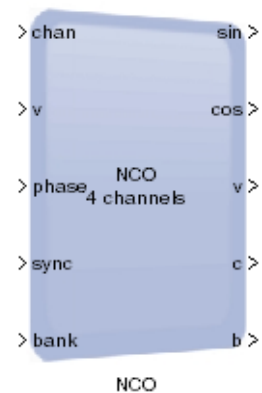
A numerically controlled oscillator (NCO) or digitally controlled oscillator (DCO) is an electronic system for synthesizing a range of frequencies from a fixed time base. NCOs are useful when a continuous phase sinusoidal signal with variable frequency is required, such as when receiving the signal from a NCO-based transmitter in a communications system.

The NCO block uses an octant-based algorithm with trigonometric interpolation.

The basic operation the NCO performs is to accumulate a phase angle in an accumulator. DSP Builder uses this angle as a lookup into sine and cosine tables to find a coarse sine and cosine approximation. DSP Builder implements the tables with a ROM. A Taylor series expansion of the small angle errors refines this coarse approximation to produce accurate sine and cosine values. The NCO block uses folding to produce multiple sine and cosine values if the sample rate is an integer fraction of the system clock rate.

You can use this block in a digital up or down converter for a radio system or a general purpose DSP application. The coefficients and input data are fixed-point types, and the output is the implied full precision fixed-point type.

An NCO sometimes needs to synchronize its phase to an exact cycle. It uses the phase and sync inputs for this purpose. The sync input is a write enable for the channel (address) specified by the chan input when the new phase value (data) is available on the phase input. You may need some external logic (which you can implement as a ModelPrim subsystem) to drive these signals. For example, you can prepare a sequence of new phase values in a shared memory and then write all the values to the NCO on a synchronization pulse. This option is particularly useful if you want an initial phase offset in the upper sinusoid. You can also use this option to implement efficient phase-shift keying (PSK) modulators in which the input to the phase modulator varies according to a data stream.



Parameters

The system specification, including such factors as the channel count, sample rates, and noise floor, determines the main parameters for this block. You can express all the parameters as MATLAB expressions, making it easy to parameterize a complete system.

The hardware generation techniques create very efficient NCOs, which are fast enough to update with every Simulink simulation, so the edit-simulation loop time is much reduced, improving productivity.

Table 19-5 shows the specification parameters for the NCO block.

Table 19-5. Specification Parameters for the NCO Block

Parameter	Description
Output Rate Per Channel (MSPS)	The sine and cosine output rate per channel measured in millions of samples per second.
Output Data Type	The output width in bits of the NCO. The bit width controls the internal precision of the NCO. The spurious-free dynamic range (SFDR) of the waves produced is approximately $6.02 \times \text{bit width}$. The 6.02 factor comes from the definition of decibels with each added bit of precision increasing the SFDR by a factor of $20 \times \log_{10}(2)$.
Output Scaling Value	This value interprets the output data in the Simulink environment. The power of 2 scaling provided lets you specify the range of the output value.
Accumulator Bit Width	Specifies the width of the memory-mapped accumulator bit width, which governs the precision you can control the NCO frequency. The width is limited to the range 15–30 for use with a 32-bit memory map (shared by other applications such as a Nios II processor). The top two bits in the 32-bit width are reserved to control the inversion of the sine and cosine outputs. A width of 30 bits gives an accumulator precision of 0.02794 Hz. However, you can select Constant for the Read/Write Mode to allow the width to increase to 40 bits. This width results in an accumulator precision of 0.000027 MHz.
Phase Increment and Inversion	A vector that represents the step in phase between each sample. This vector controls the frequencies generated during simulation. The length of the vector determines how many channels (frequencies) of data are generated from the NCO. The unit of the vector is one (sine or cosine) cycle. For information about using this parameter, refer to “ Phase Increment and Inversion ”.
Phase Increment and Inversion Memory Map	Specifies where in the memory-mapped space the NCO registers are mapped. For information about using this parameter, refer to “ Phase Increment Memory Registers ”.
Read/Write Mode	Specifies whether the NCO phase increment and inversion registers are mapped as Read , Write , Read/Write , or Constant .
Expose Avalon-MM Slave in Simulink	Allows you to reconfigure increments without Qsys. Also, it also allows you to reprogram multiple NCOs simultaneously. When you turn on this parameter, the following three additional input ports and two output ports appear in Simulink. <ul style="list-style-type: none"> ■ data, address, write ■ readdata, valid

Phase Increment and Inversion

The **Phase Increment and Inversion** parameter allows you to specify the phase increment values that control the frequencies of the sinusoidal wave signals generated during simulation. You can also specify whether to invert the generated sinusoidal signals. This parameter is closely related to the **Output Rate per Channel** and the **Accumulator Bit Width** parameters. To achieve a desired frequency (in MHz) from the NCO block, you must specify a phase increment value defined by:

$$\text{Phase Increment Value} = (\text{Frequency}) * 2^{(\text{Accumulator Bit Width})} / (\text{Output Data Rate})$$

This value must fall within the range specified by the **Accumulator Bit Width** parameter. For example, for an accumulator bit width of 24 bits, you can specify a phase increment value less than 2^{24} .

You can specify the phase increment values in a vector format that generates multichannel sinusoidal signals. The length of the vector determines how many channels (frequencies) of data are generated from the NCO block. For example, a length of 4 implies that four channels of data are generated.

When the design uses the NCO for super-rate applications (NCO frequency is higher than output data rate), for example direct RF DUC, use multiple channels (in evenly distributed phases). The phase increment value is:

$$\text{Phase increment value} = \text{mod}((\text{frequency})/(\text{output data rate}), 1) \times 2(\text{accumulator bit width})$$

The modulus function limits the phase value to less than 1 and prevents interfering with the inversion bits.

When the input is in matrix format (with multiple rows of vectors), the design configures the NCO block as a multi-bank NCO for frequency hopping for multicarrier designs. The number of rows in the matrix represents the number of banks of frequencies (of sine and cosine waves) that generate for a given channel. An additional bank input and b output port automatically add to the NCO block.



No upper limit to the number of rows exists in the matrix and you can specify any number of frequency banks. However, you should carefully monitor the resource usage to ensure that the specified design fits into the target device.

You can also use the **Phase Increment and Inversion** parameter to indicate whether the generated sinusoidal signals are inverted. For an accumulator width in bits of 24 bits, you can add two bits (the 25th and 26th bits) to the phase increment value for a given frequency. These bits indicate if the sine (26th bit) and cosine (25th bit) are inverted.

Phase Increment Memory Registers

You can use the **Phase Increment and Inversion Memory Map** parameter to specify the base address of the memory-mapped space where the NCO registers are mapped. The **System Data Width** specified in the **DSP Builder > Avalon Interfaces > Avalon MM Slave** menu and the **Accumulator Bit Width** specified in the NCO block determines the number of registers required for each phase increment value. You can specify the **System Data Width** to be either 8, 16, or 32 bits. If the **Accumulator Bit Width** is larger than the **System Data Width**, two registers are required to store each phase increment value.

The NCO block only supports one or two registers for each phase increment value. If one register is required for each phase increment value, the phase increment value for the first frequency is written into the base address, the second value into the next address (base address + 1) and so on. If you require two registers, the design uses the base address and the next address (base address + 1) for the first value with each address storing part of the value. The next pair of addresses store the next value and so on.

For example, for a **System Data Width** of 16, **Accumulator Bit Width** of 24 and **Phase Increment and Inversion Memory Map** base address of 1000, addresses 1000 and 1001 store the phase increment value for the first frequency. Address 1001 stores the lower 16 bits (15 .. 0) and address 1000 stores the remaining 8 bits (23 .. 16). If DSP Builder generates four channels of sinusoidal signals, it uses addresses 1002 and 1003 for the second channel, addresses 1004 and 1005 for the third channel, addresses 1006 and 1007 for the fourth channel.

In summary:

$$\text{<total addresses required>} = \text{<number of registers per value>} \times \text{<number of channels>}$$

When the phase increment and inversion memory map registers (in write mode) write, the new value takes effect immediately.

If the application is a super-rate operation (like direct RF DUC) and multiple channels NCO are configured for a new center frequency, first configure the phase increment value for each channel. DSP Builder then synchronizes the phases offsets of all channels at the same time by asserting the `sync` pulse.

To minimize the duration of disruption, you may use two banks of phase increment registers. The new phase increment registers bank switches first. Then, you can apply the `sync` pulse to synchronize the new phase offsets.

Frequency Hopping

You can use the NCO block to configure multiple banks of predefined frequencies for frequency hopping. If you specify a matrix comprising multiple rows of vectors as the **Phase Increment and Inversion** values, DSP Builder configures the NCO for multiple banks and defines the number of banks by the number of rows of vectors specified by inputs to the **Phase Increment and Inversion** parameter. A bank input and b output are automatically added to the NCO block. It also allocates phase increment memory registers for the multiple banks of frequencies automatically.

You can use the Avalon-MM interface to access (read or write) the phase increment memory registers in the same way as for a single bank with the register address for the *i*th bank frequencies starting from:

$$\langle \text{base address} \rangle + (i - 1) \times \langle \text{number of registers per value} \rangle \times \langle \text{number of channels} \rangle.$$

You can use the bank input as the index to switch the generated sinusoidal waves to the specified set (bank) of predefined frequencies.



Ensure you constrain the bank input to the range (0 .. $\langle \text{number of banks} \rangle - 1$). You can expect unreliable outputs from the NCO block if the bank input exceeds the number of banks.

When using an Avalon-MM interface to access (read or write) the phase increment memory registers, ensure that you only write to the inactive banks (banks which are not equal to the index specified by the input bank port). The dual-port memory that the NCO block uses is in DONT_CARE mode when reading and writing to the same address. The NCO block uses the active bank to read the phase increment value. Writing to the active bank may cause unreliable values to read out and the active bank may pass out unexpected sinusoidal signals through the memory interface.

The read data, from the address to which you write the new values to, may also be unreliable because of the memory type that the NCO block uses. Only use read data from banks where they do not write to at the same time.

Results Tab

The **Results** tab shows the implications of your parameter settings. [Table 19–6](#) describes the parameters in the **Results** tab.

Table 19–6. Results Tab Parameters for the NCO Block

Parameter	Description
Expected SFDR	The SFDR in decibels relative to the carrier (dBc): $(\text{Output Data Type Width}) \times 20 \times \log_{10}(2)$.
Accumulator precision	Accumulator precision in Hz: $10^6 \times (\text{output rate}) / 2^{(\text{accumulator width in bits}+1)}$.
Frequency	Frequency in MHz: $(\text{output rate}) \times (\text{phase increment and inversion}) / 2^{(\text{accumulator width in bits})}$.
# outputs per cycle	The number of outputs per cycle is the width of the vector of output signals: physical channels out = $\text{ceil}(\text{length}(\text{phase increment and inversion}) / ((\text{system clock frequency}) / (\text{output rate})))$
log2 of look-up table	The number of address bits in the internal look-up tables.

Port Interface

[Table 19–7](#) shows the port interface for the NCO block.

Table 19–7. Port Interface for the NCO Block

Signal	Direction	Description
chan	Input	Indicates the channel. If <i>v</i> is high, then <i>chan</i> indicates which channel the data corresponds to.
v	Input	Indicates validity. If <i>v</i> is high, then new data generates.
phase	Input	Specifies the phase offset. The size of this port should match the wire count of the NCO. ⁽¹⁾
sync	Input	Specifies the phase synchronization. The size of this port should match the wire count of the NCO output. When asserted, the phase offsets of all channels synchronize to the phase inputs. This signal has no effect to the phase increment and inversion registers. When you use this signal, you may need to initialize the offsets upon system power-up or reset. ⁽¹⁾
bank	Input	This input is available when you specify a matrix of predefined vectors for the phase increment values. You can use this input to switch to the bank of predefined frequencies. For information about using this signal, refer to “ Frequency Hopping ” on page 19–8.
data	Input	The data port has unsigned integers with a width equal to the width of the accumulator plus two for the inversion bits.
address	Input	Only available when you turn on Expose Avalon-MM Slave in Simulink . The address port is the same width as the system address width that you configure in the DSP Builder > Avalon Interfaces > Avalon MM Slave menu. Also the base address is the same.
write	Input	Deassert the write port to make a read occur.
sin	Output	The sine data output from the block. If you request more channels than can fit on a single bus, this signal is a vector. The width in bits is a function of the input width in bits and the parameterization.
cos	Output	The cosine data output from the block. If you request more channels than can fit on a single bus, this signal is vector. The width in bits is a function of the input width in bits and the parameterization. ⁽¹⁾
v	Output	Indicates validity of the data output signals.
c	Output	Indicates channel of the data output signals.
b	Output	Indicates the bank that the output signals use. This output is available when you specify a matrix of predefined vectors for the phase increment values.

Table 19-7. Port Interface for the NCO Block

Signal	Direction	Description
readdata	Output	The data port has unsigned integers with a width equal to the width of the accumulator plus two for the inversion bits.
valid	Output	Indicates a valid output.

Note to Table 19-7:

- (1) The number of sines/cosines per cycle is limited to 1–16 outputs. Use multiple NCO blocks if more outputs are required.

This chapter lists the blocks in each of the libraries in the Altera DSP Builder advanced blockset.

Additional

The additional library the following blocks in the **Avalon Streaming** folder:

- Avalon-ST Output (AStOutput)
- Avalon-ST Input (AStInput)
- Avalon-ST Input FIFO Buffer (AStInputFIFO)

The Additional library contains the following blocks in the **Beta Utilities** folder:

- Nested Loops (NestedLoop1, NestedLoop2, NestedLoop3)

The Additional library contains the following blocks in the **Control** folder:

- Zero-Latency Latch (latch_0L)
- Single-Cycle Latency Latch (latch_1L)
- Reset-Priority Latch (SRlatch_PS)
- Set-Priority Latch (SRlatch)

The Additional library contains the following blocks in the **Vector Utils** folder:

- Expand Scalar (ExpandScalar)
- Vector Multiplexer (VectorMux)
- Tapped Delay Line (TappedDelayLine)

Base

The Base library contains the following blocks:

- Channel Viewer (ChanView)
- Control
- Device
- Edit Params
- LocalThreshold
- Run ModelSim
- Run Quartus II
- Scale
- Signals

FFT

The FFT library contains the following blocks in the **Common** folder:

- Bit Vector Combine (BitVectorCombine)
- Choose Bits (ChooseBits)
- Crossover Switch (XSwitch)
- Dual Twiddle Memory (DualTwiddleMemoryC)
- Edge Detect (EdgeDetect)
- Multiwire Transpose (MultiwireTranspose)
- Pulse Divider (PulseDivider)
- Pulse Multiplier (PulseMultiplier)
- Single-Wire Transpose (Transpose)
- Split Scalar (SplitScalar)

The following blocks are available in the **Floating Point** folder:

- Butterfly Unit (BFU)
- FFT (FFT, FFT_Light, VFFT, VFFT_Light)
- Floating-Point Twiddle Generator (TwiddleGenF) (Deprecated)

The following blocks are available in the **Radix2** folder:

- Butterfly I C (BFIC) (Deprecated)
- Butterfly II C (BFIC) (Deprecated)
- Bit Reverse Core C (BitReverseCoreC and VariableBitReverse)
- FFT (FFT, FFT_Light, VFFT, VFFT_Light)
- Twiddle Angle (TwiddleAngle)
- Twiddle Generator (TwiddleGenC) (Deprecated)
- Twiddle ROM (TwiddleRom, TwiddleMultRom and TwiddleRomF (deprecated))

Filter

The Filter library contains the following blocks:

- Decimating CIC
- Decimating FIR
- Fractional Rate FIR
- Interpolating CIC
- Interpolating FIR
- Single-Rate FIR

ModelBus

The ModelBus library includes the following blocks:

- Bus Slave (BusSlave)
- Bus Stimulus (BusStimulus)
- Bus Stimulus File Reader (Bus StimulusFileReader)
- Register Bit (RegBit)
- Register Field (RegField)
- Register Out (RegOut)
- Shared Memory (SharedMem)

ModelPrim

The ModelPrim library contains the following primitive blocks:

- Absolute Value (Abs)
- Accumulator (Acc)
- Add
- Add SLoad (AddSLoad)
- AddSub
- AddSubFused
- AND Gate (And)
- Bit Combine (BitCombine)
- Bit Extract (BitExtract)
- Bit Reverse (BitReverse)
- Channel In (ChannelIn)
- Channel Out (ChannelOut)
- Compare (CmpCtrl)
- Complex Conjugate (ComplexConjugate)
- Compare Equality (CmpEQ)
- Compare Greater Than (CmpGE)
- Compare Less Than (CmpLT)
- Compare Not Equal (CmpNE)
- Constant (Const)
- Constant Multiply (Const Mult)
- Convert
- CORDIC
- Counter

- Count Leading Zeros, Ones, or Sign Bits (CLZ)
- Dual Memory (DualMem)
- Demultiplexer (Demux)
- Divide
- FIFO
- Floating-point Classifier (FloatClass)
- Floating-point Multiply Accumulate (Acc and MultAcc)
- ForLoop
- General Purpose Input (GPIIn)
- General Purpose Output (GPOut)
- Load Exponent (LdExp)
- Left Shift (LShift)
- Loadable Counter (LoadableCounter)
- Look-Up Table (Lut)
- Loop
- Math
- Minimum and Maximum (MinMax)
- MinMaxCtrl
- Multiply (Mult)
- Multiplexer (Mux)
- NAND Gate (Nand)
- Negate
- NOR Gate (Nor)
- NOT Gate (Not)
- OR Gate (Or)
- Reinterpret Cast (ReinterpretCast)
- Round
- Sample Delay (SampleDelay)
- Scalar Product
- Select
- Sequence
- Shift
- Sqrt
- Subtract (Sub)
- Synthesis Information (SynthesisInfo)

- XNOR Gate (Xnor)
- XOR Gate (Xor)

ModelVectorPrim

- Sum of Elements (SumOfElements)

Waveform Synthesis

The Waveform Synthesis library includes the following blocks:

- Complex Mixer (ComplexMixer)
- Real Mixer (Mixer)
- NCO

Symbols

.xml file 2-28

A

- Abs block 17-5
- Acc block 17-6
- Add block 17-8
- Add SLoad block 17-9
- AddSub block 17-10
- AddSubFused block 17-10
- Anchored Delay block 12-6
- And block 17-11
- AStInput block 12-3
- AStInputFIFO block 12-3
- AStOutput block 12-2
- Automatic testbench
 - Enabling 13-4
- Avalon-MM interface
 - Bus clock 1-4
- Avalon-MM Slave Settings block 13-1

B

- Base Blocks
 - Example designs 6-2
- BFIC block 14-5
- BFIIC block 14-6
- BFU block 14-4
- BFU_long block 14-4
- BFU_short block 14-4
- Bit Combine block 17-12
- Bit Extract block 17-13
- Bit Reverse primitive block 17-14
- BitReverseCoreC block 14-2
- BitVectorCombine block 14-3
- Bus clock 1-6
 - Avalon-MM interface 1-4
- Bus Slave block 16-1
- Bus Stimulus block 16-2
- Bus Stimulus File Reader block 16-3

C

- CDelay RAM Block threshold 13-5
- CDualMem Dist RAM threshold 13-5
- Channel In block 17-14
- Channel Out block 17-15
- ChanView block 13-2
- ChooseBits block 14-6
- Clock rate 1-4

- CLZ block 17-27
- CmpCtrl block 17-16
- CmpEQ block 17-18
- CmpGE block 17-18
- CmpLT block 17-19
- CmpNE block 17-19
- Complex Mixer block 19-1
- Complex to Real-Imag block 12-7
- Const block 17-20
- Const Mult block 17-21
- Control block 13-3
- Convert block 17-22
- CORDIC block 17-23
- Counter block 17-25

D

- Decimating CIC block 15-5
- Decimating FIR block 15-6
- Demux block 17-29
- Design example
 - Running 6-2
- Device block 13-6
- Divide block 17-30
- DualMem block 17-27
- DualTwiddleMemoryC block 14-8

E

- EdgeDetect block 14-8
- Edit Params Block 13-7
- Enabled Delay Line block 12-7
- Enabled Feedback Delay block 12-7
- Enhanced Precision Support block 13-11
- Example designs
 - 16-Channel DDC 6-35
 - 16-Channel DUC 6-35
 - 1K Floating-Point FFT 6-4
 - 2-Antenna DUC for WiMAX 6-36
 - 2-Channel DUC 6-36
 - Automatic Gain Control 6-38
 - Avalon-ST Interface (Input and Output FIFO Buffer) with Backpressure 6-30
 - Avalon-ST Interface (Output FIFO Buffer) with Backpressure 6-30
 - Bit Combine for Boolean Vectors 6-39
 - Bit Extract for Boolean Vectors 6-39
 - Black-Scholes Floating Point 6-24
 - Color Space Converter (Resource Sharing Folding) 6-39

- Complex FIR 6-11
- Complex Mixer 6-62
- Decimating CIC Filter 6-11
- Decimating FIR Filter 6-12
- Digital Predistortion Forward Path 6-40
- Double-Precision Real Floating-Point Matrix Multiply 6-24
- FFT 6-5, 6-6
- FFT without BitReverseCoreC Block 6-5
- Fibonacci Series 6-40
- Filter Chain with Forward Flow Control 6-12
- Fine Doppler Estimator 6-24
- FIR Filter with Exposed Bus 6-12
- Fixed-Point Math Functions 6-41
- Floating Point Mandelbrot Set 6-25
- Floating Point Matrix Multiply 6-28
- Floating-Point FFT without BitReverseCoreC Block 6-6
- Floating-Point IFFT 6-7
- Floating-Point IFFT without BitReverseCoreC Block 6-7
- Folded Primitive FIR Filter 6-23
- Fractional FIR Filter Chain 6-13
- Fractional Rate FIR Filter 6-13
- Fractional Square Root Using CORDIC 6-41
- Half Band FIR Filter 6-13
- Hello World 6-42
- Hybrid Direct Form and Transpose Form FIR Filter 6-42
- IFFT 6-5
- IFFT without BitReverseCoreC Block 6-6
- Interpolating CIC Filter 6-13
- Interpolating FIR Filter 6-14
- Inverse Discrete Cosine Transform 6-38
- Kronecker Tensor Product 6-30
- Loadable Counter 6-42
- Local threshold 6-3
- Matrix Initialization of LUT 6-43
- Matrix Initialization of Vector Memories 6-43
- Memory-Mapped Registers 6-34
- Multi-Channel IIR Filter 6-44
- Multiple Coefficient Banks Interpolating FIR Filter 6-14
- Multiwire Transpose 6-7
- NCO 6-66
- NCO with Exposed Bus 6-66
- NCO, 4 channel, 16 banks 6-65
- NCO, 4 channel, 2 banks 6-62
- NCO, 4 channel, 4 banks 6-63
- NCO, 4 channel, 8 banks, 2 wires 6-64
- Newton Root Finding Tutorial Step 1—Iteration 6-27
- Newton Root Finding Tutorial Step 2—Convergence 6-27
- Newton Root Finding Tutorial Step 3—Valid 6-27
- Newton Root Finding Tutorial Step 4—Control 6-27
- Newton Root Finding Tutorial Step 5—Final 6-28
- Normalizer 6-28
- Parallel Loops 6-31
- Position, Speed, and Current Control for AC Motors 6-17
- Position, Speed, and Current Control for AC Motors (with ALU Folding) 6-22
- Primitive FIR with Back Pressure 6-31
- Primitive FIR with Forward Pressure 6-32
- Primitive Systolic FIR using Forward Flow Control 6-33
- Quadrature Amplitude Modulation 6-44
- Radix 2 Streaming FFT 6-8
- Radix 4 Streaming FFT 6-8
- Real Mixer 6-66
- Rectangular Nested Loop 6-33
- Reinterpret Cast for Bit Packing and Unpacking 6-44
- Root Raised Cosine FIR Filter 6-15
- Run-time Configurable Decimating and interpolating Half-rate FIR Filter 6-45
- Scale 6-3
- Sequential Loops 6-34
- Simple Nonadaptive 2D Beamformer 6-29
- Single Rate FIR Filter 6-16
- Single-Precision Complex Floating-Point Matrix Multiply 6-28
- Single-Precision Real Floating-Point Matrix Multiply 6-29
- Single-Wire Transpose 6-8
- Square Root Using CORDIC 6-45
- Super-Sample Decimating FIR Filter 6-16
- Super-Sample Fractional FIR Filter 6-16
- Super-Sample Interpolating FIR Filter 6-16
- Switchable FFT 6-9
- Test CORDIC Using the CORDIC Block 6-45
- Triangular Nested Loop 6-34
- Updating Coefficient Banks Interpolating FIR Filter 6-15
- Variable-Rate CIC Filter 6-17
- Variable-Size Fixed-Point FFT 6-9
- Variable-Size Fixed-Point FFT without BitReverseCoreC Block 6-9
- Variable-Size Fixed-Point iFFT without BitReverseCoreC Block 6-9
- Variable-Size Floating-Point FFT 6-9, 6-10
- Variable-Size Floating-Point FFT without BitReverseCoreC Block 6-9
- Variable-Size Floating-Point iFFT without Bit-

- ReverseCoreC Block 6-10
- Variable-Size Low-Resource FFT 6-10
- Variable-Size Low-Resource Real-Time FFT 6-10
- Vector Initialization of Sample Delay 6-47
- Wide-Single Channel Accumulators 6-47
- ExpandScalar block 12-7

F

- FFT
 - Example designs 6-2
- FFT block 14-9
- FFT_Light block 14-9
- FIFO block 17-31
- Filters
 - Example designs 6-2
- Floating Point
 - Example designs 6-2
- Flow Control
 - Example designs 6-2
- Folding
 - Example designs 6-2
- ForLoop block 17-32
- Fractional Rate FIR block 15-9

G

- Generated files
 - DDC design example 7-15
- GPIIn block 17-34
- GPOut block 17-35

H

- Hard multiplier threshold 13-5
- Hardware generation
 - Enabling 13-4
- Host Interface
 - Example designs 6-2

I

- Interpolating CIC block 15-12
- Interpolating FIR block 15-14

L

- latch_0 block 12-6
- latch_1 block 12-6
- Latency parameter
 - Displaying for a primitive subsystem 2-3
 - Reading 10-20
- LdExp block 17-35
- Library
 - Additional 12-1
 - Base 13-1
 - FFT 14-1

- Filter 15-1
- ModelBus 16-1
- ModelPrim 17-1
- ModelVectorPrim 18-1
- Waveform Synthesis 19-1
- LoadableCounter block 17-37
- LocalThreshold block 13-11
- Loop block 17-39
- LShift block 17-36
- Lut block 17-38

M

- Math block 17-40
- Memory and multiplier thresholds 13-4
- Memory-Mapped bus interface
 - Address and bus width 13-4
- MinMax block 17-41
- MinMaxCtrl block 17-42
- ModelBus
 - Example designs 6-2
- ModelIP
 - Protocol 2-8
- ModelPrim
 - Example designs 6-2
- M-RAM threshold 13-5
- Mult block 17-43
- Multichannel systems 2-16
- MultiwireTranspose block 14-17
- Mux2 block 17-44

N

- Nand block 17-45
- NCO block 19-5
- Negate block 17-46
- NestedLoop1 block 12-3
- Nor block 17-47
- Not block 17-48

O

- Or block 17-49

P

- Platforms
 - Example designs 6-2
- Polynomial block 17-49
- Primitive blocks
 - Example designs 6-2
 - Output data type 2-3
 - Synthesis styles 17-61
- PulseDivider block 14-18
- PulseMultiplier block 14-19

R

Real Mixer block 19-3

Reference designs 6-2

1-Antenna WiMAX DDC 6-49

1-Antenna WiMAX DUC 6-50

1-Carrier, 2-Antenna W-CDMA DDC 6-52

1-Carrier, 2-Antenna W-CDMA DUC 6-53

2-Antenna WiMAX DDC 6-49

2-Antenna WiMAX DUC 6-50

4-Carrier, 2-Antenna W-CDMA DDC 6-51

4-Carrier, 2-Antenna W-CDMA DUC 6-52

4-Carrier, 2-Antenna W-CDMA DUC at 307.2
MHz with Total Rate Change 40 6-554-Carrier, 2-Antenna WDCMA DUC at 368.64
MHz with Total Rate Change 32 6-544-Carrier, 2-Antenna WDCMA DUC at 368.64
MHz with Total Rate Change 48 6-54

Cholesky Solver Multiple Channels 6-57

Cholesky Solver Single Channel 6-56

Dynamic Decimating FIR 6-58

Multichannel QR Decomposition 6-58

QR Decomposition 6-58

Reconfigurable Decimation Filter 6-59

Single-Channel 10-MHz LTE Transmitter 6-59

STAP Radar Forward and Backward Substitu-
tion 6-60

STAP Radar QR Decomposition 192x204 6-61

STAP Radar Steering Generation 6-61

time delay beamformer 6-61

Variable Integer Rate Decimation Filter 6-61

Register Bit block 16-5

Register Field block 16-6

Register Out block 16-7

Reinterpret cast block 17-50

Round block 17-51

Run ModelSim block 13-12

Run Quartus II block 13-12

S

Sample Delay block 17-52

Sample rate 1-4

Scalar Product block 17-53

Scale block 13-13

Select block 17-56

Sequence block 17-57

Shared Memory block 16-7

Shift block 17-58

Signals block 13-15

Single Rate FIR block 15-17

SplitScalar block 14-20

Sqrt block 17-59

SRLatch block 12-6

SRLatch_PS block 12-7

Standard blockset interoperability 1-5

Sub block 17-60

SynthesisInfo block 17-61

System clock 1-6

T

TappedDelayLine block 12-8

Transpose block 14-19

Trig block 17-63

Troubleshooting 11-1

Tutorials

Primitive Library 5-1

TwiddleAngle block 14-21

TwiddleGenC block 14-22

TwiddleGenF block 14-13

TwiddleROM block 14-24

V

Vectorized inputs 2-19

VectorMux block 12-8

W

Walkthrough

Primitive Library tutorial 5-1

Waveform Synthesis

Example designs 6-2

X

Xnor block 17-64

Xor block 17-65

XSwitch block 14-7

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for the user guide section.

Date	Version	Changes Made
December 2014	8.1	<ul style="list-style-type: none"> ■ Added step about disabling virtual pins in the Quartus II software when using HIL with advanced blockset designs/ ■ Added information on <code>_mmap.h</code> file, which contains register information on your design ■ Corrected Has read enable and Show read enable descriptions in BusStimulus and BusStimulusFileReader blocks ■ Added BusStimulus and BusStimulusFileReader blocks to memory-mapped registers design example. ■ Added AvalonMMSlaveSettings block and DSP Builder > Avalon Interfaces > Avalon-MM slave menu option ■ Removed bus parameters from Control and Signal blocks ■ Removed the following design examples: <ul style="list-style-type: none"> ■ Color Space Converter (Resource Sharing Folding) ■ Interpolating FIR Filter with Updating Coefficients ■ Primitive FIR Filter (Resource Sharing Folding) ■ Single-Stage IIR Filter (Resource Sharing Folding) ■ Three-stage IIR Filter (Resource Sharing Folding) ■ Added system-in-the-loop support ■ Added new blocks: <ul style="list-style-type: none"> ■ Floating-point classifier ■ Floating-point multiply accumulate ■ Added hypotenuse function to math block ■ Added design examples: <ul style="list-style-type: none"> ■ Color space converter ■ Complex FIR design example ■ CORDIC from Primitive Blocks ■ Crest factor reduction ■ Folding FIR ■ Variable Integer Rate Decimation Filter ■ Vector sort - sequential and iterative ■ Added reference designs: <ul style="list-style-type: none"> ■ Crest factor reduction ■ Direct RF with Synthesizable Testbench ■ Dynamic Decimation Filter ■ Reconfigurable Decimation Filter ■ Variable Integer Rate Decimation Filter ■ Changed directory structure ■ Added correct floating-point rounding for reciprocal and square root blocks. ■ Corrected signal descriptions for LoadableCounter block ■ Removed resource sharing folder ■ Added new ALU folder information: <ul style="list-style-type: none"> ■ Start of packet signal ■ Clock-rate mode

Date	Version	Changes Made
June 2014	8.0	<ul style="list-style-type: none"> ■ Added new blocks: <ul style="list-style-type: none"> ■ Enabled Delay Line ■ Enabled Feedback Delay ■ FFT2P, FFT4P, FFT8P, FFT16P, FFT32P, and FFT64P ■ FFT2X, FFT4X, FFT8X, FFT16X, FFT32X, and FFT64X ■ FFT2, FFT4, VFFT2, and VFFT4 ■ General Multitwiddle and General Twiddle (GeneralMultiTwiddle, GeneralTwiddle) ■ Hybrid FFT (Hybrid_FFT) ■ Parallel Pipelined FFT (PFFT_Pipe) ■ Ready ■ Updated port descriptions for <ul style="list-style-type: none"> ■ Convert ■ Const ■ LUT ■ Reinterpretcast ■ New parameter interfaces for: <ul style="list-style-type: none"> ■ FFT ■ FFT_Light ■ VFFT ■ VFFT_Light ■ Added new design examples: <ul style="list-style-type: none"> ■ Avalon-ST Interface (Input and Output FIFO Buffer) with Backpressure ■ Avalon-ST Interface (Output FIFO Buffer) with Backpressure ■ Fixed-point maths functions ■ Fractional square root using CORDIC ■ Normalizer ■ Square root using CORDIC ■ Switchable FFT/iFFT ■ Variable-Size Fixed-Point FFT ■ Variable-Size Fixed-Point FFT without BitReverseCoreC Block ■ Variable-Size Fixed-Point iFFT ■ Variable-Size Fixed-Point iFFT without BitReverseCoreC Block ■ Variable-Size Floating-Point FFT ■ Variable-Size Floating-Point FFT without BitReverseCoreC Block ■ Variable-Size Floating-Point iFFT ■ Variable-Size Floating-Point iFFT without BitReverseCoreC Block ■ Added new <code>ready</code> signal for ALU folding.

Date	Version	Changes Made
June 2014 cont	14.0	<ul style="list-style-type: none"> ■ Deprecated the following FFT blockset blocks <ul style="list-style-type: none"> ■ BFIC ■ BFICF ■ BFIIC ■ BFIICF ■ BFTC ■ FFT_Light (replaced with new version that has new parameters) ■ FP_FFT_Light ■ FP_VFFT_Light ■ TwiddleGenC ■ TwiddleGenF ■ TwiddleROM (replaced with new version that has new parameters) ■ TwiddleROMF ■ VFFT_Light (replaced with new version that has new parameters) ■ Updated instructions on how to list demos. ■ Updated descriptions for BusStimulus, BusStimulusFileReader, RegField, and RegOut blocks ■ Removed test CORDIC functions with primitive blocks design example. ■ Added limit of 50 testbenches to DSP Builder > Run ModelSim list. ■ Updated <i>Floating-point Data Types</i> chapter ■ Revised ALU folding information
November 2013	7.0	<ul style="list-style-type: none"> ■ Added delays in control units topic. ■ Added new functions to Math block. ■ Added more information to FIFO block and LdExp block ■ Added Simulink fi block option to Const, DualMem, and LUT blocks ■ Added new design examples: <ul style="list-style-type: none"> ■ Variable-precision real-time FFT ■ Interpolating FIR Filter with updating coefficients ■ Time-delay beamformer ■ Added new blocks: <ul style="list-style-type: none"> ■ Anchored Delay ■ Polynomial ■ TwiddleAngle ■ TwiddleROM and TwiddleROMF ■ VariableBitReverse ■ VFFT

Date	Version	Changes Made
May 2013	6.0	<ul style="list-style-type: none"> ■ Updated device block description with new Device Selector menu. ■ Updated floating-point data types ■ Added new ModelPrim blocks: <ul style="list-style-type: none"> ■ Const Mult ■ Divide ■ MinMax ■ Negate ■ Scalar Product ■ Added nine new FFT blocks ■ Added ten new FFT demonstrations ■ Removed deprecated blocks ■ Renamed bit-reverse FFT with natural output (FFT_BR_Natural) to FFT block
November 2012	5.0	<ul style="list-style-type: none"> ■ Updated description for the Sqrt block ■ Updated descriptions for FFT design examples. ■ Added descriptions for the following new ModelPrim blocks: <ul style="list-style-type: none"> ■ AddSub ■ AddSubFused ■ CmpCtrl ■ Math ■ Maximum and Minimum ■ MinMaxCtrl ■ Round ■ Trig ■ Added descriptions for the following new FFT blocks: <ul style="list-style-type: none"> ■ Edge Detect (EdgeDetect) ■ Pulse Divider (PulseDivider) ■ Pulse Multiplier (PulseMultiplier) ■ Bit-Reverse FFT with Natural Output (FFT_BR_Natural) ■ Added descriptions for the following new FIR design examples: <ul style="list-style-type: none"> ■ Super-Sample Decimating FIR Filter ■ Super-sample Fractional FIR Filter ■ Added the position, speed, and current control for AC motors (with ALU folding) design example. ■ Added ALU folding description ■ Added memory map description.

Date	Version	Changes Made
June 2012	4.0	<ul style="list-style-type: none"> ■ Added new DSP Builder starting procedure to tutorials—you must start from the Windows Start menu and click Start in MATLAB. ■ Added description of new DSP Builder menu in Simulink designs that has the following options: <ul style="list-style-type: none"> ■ New model wizard ■ Design checker ■ Verify design ■ Resource usage ■ View memory map ■ Run Quartus II block ■ Run ModelSim block ■ Renamed WCDMA models ■ Updated ModelIP tutorial ■ Updated System tutorial instructions ■ Updated opening design examples instructions ■ Added new design examples: <ul style="list-style-type: none"> ■ FIR filter with exposed bus ■ NCO with exposed bus ■ Reinterpret cast ■ Added new QR decomposition reference design ■ Added new blocks: <ul style="list-style-type: none"> ■ Accumulator ■ Floor ■ Modulus ■ Nested loop ■ Reinterpret cast ■ Square root ■ Updated block descriptions to add new signals and parameters: <ul style="list-style-type: none"> ■ Reinterpret cast ■ Decimating FIR ■ Fractional Rate FIR ■ Interpolating FIR ■ Single-Rate FIR ■ NCO ■ Corrected loadable counter block description

Date	Version	Changes Made
November 2011	3.0	<ul style="list-style-type: none"> ■ Reordered design examples into alphabetical order ■ Added new design examples and reference designs: <ul style="list-style-type: none"> ■ Bit combine and bit extract ■ Cholesky ■ Reloadable counter ■ Simple adaptive 2D beamformer ■ STAP radar ■ Wide-single channel accumulators
April 2011	2.1	<ul style="list-style-type: none"> ■ Reordered some of the information in the chapters ■ Added <i>Getting Started</i> chapter ■ Added <i>Floating-Point Data Types</i> chapter ■ Updated description for multiple coefficient banks interpolating FIR filter ■ Added the following two examples: <ul style="list-style-type: none"> ■ Fine Doppler Estimator ■ Position, Speed, and Current Control for AC Motors
December 2010	2.0	<ul style="list-style-type: none"> ■ Added the following design example descriptions: <ul style="list-style-type: none"> ■ Rectangular nested loop ■ Triangular nested loop ■ Sequential loops ■ Partial loops ■ Folded color space converter ■ Folded single-stage IIR filter ■ Folded 3-stage IIR filter ■ Folded primitive FIR filter ■ Hybrid direct form and transpose form FIR filter ■ Digital predistortion forward path ■ Run-time configurable decimating and interpolating half-rate FIR filter ■ Matrix initialization of vector memories ■ Matrix initialization of LUT ■ Vector initialization of sample delay ■ Added <i>General Primitive Library Guidelines</i>
June 2010	1.0	First published. Replaces <i>DSP Builder Advanced Blockset User Guide</i>

The following table shows the revision history for the library section.

Date	Version	Changes Made
November 2011	3.0	<ul style="list-style-type: none"> ■ Added LoadableCounter block description ■ Separated Additional and Base library chapters ■ Removed Delay block
April 2011	2.1	<ul style="list-style-type: none"> ■ Corrected input vector information for FIFO block ■ Updated BitCombine and BitExtract block descriptions ■ Added floating-point data types to the following ModelPrim blocks: <ul style="list-style-type: none"> ■ Abs ■ Add ■ AddSload ■ Const ■ CmpEQ ■ CmpGE ■ CmpLT ■ CmpNE ■ ComplexConjugate ■ Lut ■ Min ■ Max ■ Mult ■ Mux ■ SampleDelay ■ Select ■ Sub ■ Added the following block descriptions: <ul style="list-style-type: none"> ■ ACos ■ ASin ■ ATan ■ Cos ■ Divide ■ Exp ■ LdExp ■ Log ■ Reciprocal ■ RecipSqRt ■ Sin ■ Tan

Date	Version	Changes Made
December 2010	2.0	<ul style="list-style-type: none"> ■ Corrected categorized block list ■ Added the following block descriptions: <ul style="list-style-type: none"> ■ AStInput ■ AStInputFIFO ■ AStOutput ■ ExpandScalar ■ trappedDelayLine ■ VectorMux
June 2010	1.0	First published. Replaces <i>DSP Builder Advanced Blockset Reference Manual</i> .

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact ⁽¹⁾	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Nontechnical support (general) (software licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com










Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, \qdesigns directory, D: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicate document titles. For example, <i>Stratix IV Design Guidelines</i> .
<i>italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>.pof file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.

Visual Cue	Meaning
“Subheading Title”	Quotation marks indicate references to sections in a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”
Courier type	<p>Indicates signal, port, register, bit, block, and primitive names. For example, <code>data1</code>, <code>tdi</code>, and <code>input</code>. The suffix <code>n</code> denotes an active-low signal. For example, <code>resetn</code>.</p> <p>Indicates command line commands and anything that must be typed exactly as it appears. For example, <code>c:\qdesigns\tutorial\chiptrip.gdf</code>.</p> <p>Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword <code>SUBDESIGN</code>), and logic function names (for example, <code>TRI</code>).</p>
	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	The question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	The multimedia icon directs you to a related multimedia presentation.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.
	The feedback icon allows you to submit feedback to Altera about the document. Methods for collecting feedback vary as appropriate for each document.

