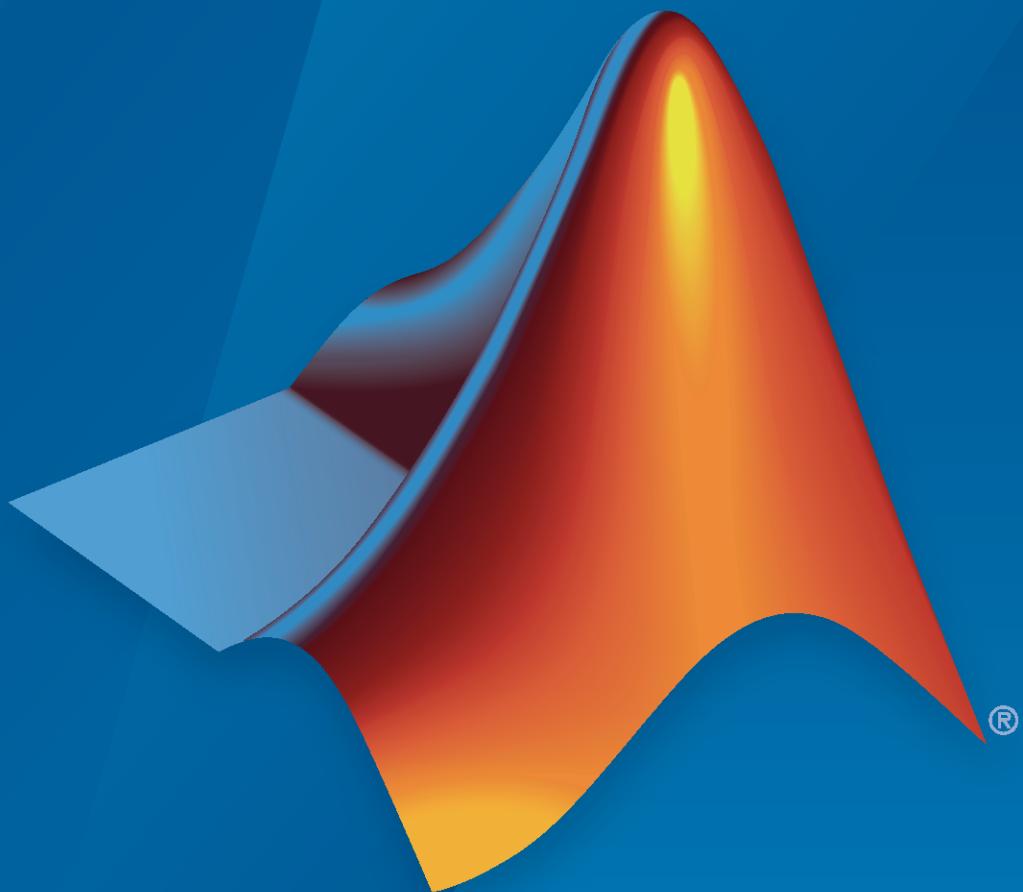


**HDL Coder™**

User's Guide



**MATLAB® & SIMULINK®**

R2020b

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)  
Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

*HDL Coder™ User's Guide*

© COPYRIGHT 2012-2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

**FEDERAL ACQUISITION:** This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

March 2012	Online only	New for Version 3.0 (R2012a)
September 2012	Online only	Revised for Version 3.1 (R2012b)
March 2013	Online only	Revised for Version 3.2 (R2013a)
September 2013	Online only	Revised for Version 3.3 (R2013b)
March 2014	Online only	Revised for Version 3.4 (R2014a)
October 2014	Online only	Revised for Version 3.5 (R2014b)
March 2015	Online only	Revised for Version 3.6 (R2015a)
September 2015	Online only	Revised for Version 3.7 (R2015b)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (R2016a)
September 2016	Online only	Revised for Version 3.9 (R2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)
September 2017	Online only	Revised for Version 3.11 (R2017b)
March 2018	Online only	Revised for Version 3.12 (Release 2018a)
September 2018	Online only	Revised for Version 3.13 (Release 2018b)
March 2019	Online only	Revised for Version 3.14 (Release 2019a)
September 2019	Online only	Revised for Version 3.15 (Release 2019b)
March 2020	Online only	Revised for Version 3.16 (Release 2020a)
September 2020	Online only	Revised for Version 3.17 (Release 2020b)

## HDL Code Generation from MATLAB

### MATLAB Algorithm Design

---

### 1

<b>Functions Supported for HDL Code Generation</b> . . . . .	1-2
Supported MATLAB and Fixed Point Runtime Library Functions . . . . .	1-2
Fixed-Point Function Limitations . . . . .	1-2
<b>Supported MATLAB Data Types, Operators, and Control Flow Statements</b> . . . . .	1-4
Supported Data Types . . . . .	1-4
Supported Operators . . . . .	1-5
Control Flow Statements . . . . .	1-7
<b>Persistent Variables and Persistent Array Variables</b> . . . . .	1-9
Persistent Variables . . . . .	1-9
Persistent Array Variables . . . . .	1-9
<b>Complex Data Type Support</b> . . . . .	1-11
Declaring Complex Signals . . . . .	1-11
Conversion Between Complex and Real Signals . . . . .	1-12
Support for Vectors of Complex Numbers . . . . .	1-12
<b>HDL Code Generation for System Objects</b> . . . . .	1-14
Why Use System Objects? . . . . .	1-14
Predefined System Objects . . . . .	1-14
User-Defined System Objects . . . . .	1-14
Limitations of HDL Code Generation for System Objects . . . . .	1-14
System object Examples for HDL Code Generation . . . . .	1-15
<b>HDL Code Generation from System Objects</b> . . . . .	1-16
<b>HDL Code Generation for Streaming Matrix Inverse System Object</b> . . . . .	1-20
<b>HDL Code Generation for Streaming Matrix Multiply System Object</b> . . . . .	1-29
<b>HDL Code Generation from hdl.RAM System Object</b> . . . . .	1-37
<b>HDL Code Generation from A Non-Restoring Square Root System Object</b> . . . . .	1-41

<b>HDL Code Generation from Viterbi Decoder System Object . . . . .</b>	<b>1-46</b>
<b>Predefined System Objects Supported for HDL Code Generation . . . . .</b>	<b>1-50</b>
Predefined System Objects in MATLAB Code . . . . .	1-50
Predefined System Objects in the MATLAB System Block . . . . .	1-51
<b>Load constants from a MAT-File . . . . .</b>	<b>1-52</b>
<b>Generate Code for User-Defined System Objects . . . . .</b>	<b>1-53</b>
How To Create A User-Defined System object . . . . .	1-53
User-Defined System object Example . . . . .	1-53
<b>Map Matrices to ROM . . . . .</b>	<b>1-55</b>
<b>Model State with Persistent Variables and System Objects . . . . .</b>	<b>1-56</b>
<b>Bitwise Operations in MATLAB for HDL Code Generation . . . . .</b>	<b>1-59</b>
Bit Shifting and Rotation . . . . .	1-59
Bit Slicing and Bit Concatenation . . . . .	1-60
<b>Guidelines for Writing MATLAB Code to Generate Efficient HDL Code . . . . .</b>	<b>1-62</b>
MATLAB Design Requirements for HDL Code Generation . . . . .	1-62
Guidelines for Writing MATLAB code . . . . .	1-62
<b>For-Loop Best Practices for HDL Code Generation . . . . .</b>	<b>1-64</b>
Monotonically Increasing Loop Counters . . . . .	1-64
Persistent Variables in Loops . . . . .	1-64
Persistent Arrays in Loops . . . . .	1-65
<b>MATLAB Test Bench Requirements and Best Practices for HDL Code Generation . . . . .</b>	<b>1-66</b>
What Is a MATLAB Test Bench? . . . . .	1-66
MATLAB Test Bench Requirements . . . . .	1-66
MATLAB Test Bench Best Practices . . . . .	1-66

## **MATLAB to HDL Examples for Communications and Signal Processing Applications**

**2**

<b>HDL Code Generation for LMS Filter . . . . .</b>	<b>2-2</b>
<b>Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number . . . . .</b>	<b>2-9</b>
<b>Timing Offset Estimation . . . . .</b>	<b>2-14</b>
<b>Data Packetization . . . . .</b>	<b>2-18</b>
<b>Transmit and Receive FIFO . . . . .</b>	<b>2-25</b>

<b>HDL Code Generation for Harris Corner Detection Algorithm</b> . . . . .	<b>2-32</b>
<b>HDL Code Generation for Adaptive Median Filter</b> . . . . .	<b>2-39</b>
<b>Contrast Adjustment</b> . . . . .	<b>2-47</b>
<b>Image Enhancement by Histogram Equalization</b> . . . . .	<b>2-56</b>
<b>HDL Code Generation for Image Format Conversion from RGB to YUV</b> . . . . .	<b>2-58</b>
<b>High Dynamic Range Imaging</b> . . . . .	<b>2-63</b>
<b>Accelerate a Pixel-Streaming Design Using MATLAB Coder</b> . . . . .	<b>2-68</b>
<b>Enhanced Edge Detection from Noisy Color Video</b> . . . . .	<b>2-71</b>
<b>Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow</b> . . . . .	<b>2-74</b>

## **MATLAB Best Practices and Design Patterns for HDL Code Generation**

**3**

<b>Model a Counter for HDL Code Generation</b> . . . . .	<b>3-2</b>
MATLAB Counter . . . . .	<b>3-2</b>
MATLAB Code for the Counter . . . . .	<b>3-2</b>
<b>Model a State Machine for HDL Code Generation</b> . . . . .	<b>3-4</b>
MATLAB Code for the Mealy State Machine . . . . .	<b>3-4</b>
MATLAB Code for the Moore State Machine . . . . .	<b>3-5</b>
<b>Generate Hardware Instances For Local Functions</b> . . . . .	<b>3-8</b>
MATLAB Local Functions . . . . .	<b>3-8</b>
MATLAB Code for mlhdlc_two_counters.m . . . . .	<b>3-8</b>
<b>Implement RAM Using MATLAB Code</b> . . . . .	<b>3-10</b>
Implement RAM Using a Persistent Array or System object Properties . . . . .	<b>3-10</b>
Implement RAM Using hdl.RAM . . . . .	<b>3-11</b>

## **Fixed-Point Conversion**

**4**

<b>Specify Type Proposal Options</b> . . . . .	<b>4-2</b>
<b>Log Data for Histogram</b> . . . . .	<b>4-5</b>

<b>View and Modify Variable Information</b> .....	<b>4-7</b>
View Variable Information .....	4-7
Modify Variable Information .....	4-7
Revert Changes .....	4-8
Promote Sim Min and Sim Max Values .....	4-8
<b>Automated Fixed-Point Conversion</b> .....	<b>4-9</b>
License Requirements .....	4-9
Automated Fixed-Point Conversion Capabilities .....	4-9
Code Coverage .....	4-10
Proposing Data Types .....	4-12
Locking Proposed Data Types .....	4-13
Viewing Functions .....	4-14
Viewing Variables .....	4-14
Histogram .....	4-19
Function Replacements .....	4-20
Validating Types .....	4-21
Testing Numerics .....	4-21
Detecting Overflows .....	4-21
<b>Custom Plot Functions</b> .....	<b>4-23</b>
<b>Visualize Differences Between Floating-Point and Fixed-Point Results</b> .....	<b>4-24</b>
<b>Inspecting Data Using the Simulation Data Inspector</b> .....	<b>4-29</b>
What Is the Simulation Data Inspector? .....	4-29
Import Logged Data .....	4-29
Export Logged Data .....	4-29
Group Signals .....	4-29
Run Options .....	4-29
Create Report .....	4-30
Comparison Options .....	4-30
Enabling Plotting Using the Simulation Data Inspector .....	4-30
Save and Load Simulation Data Inspector Sessions .....	4-30
<b>Enable Plotting Using the Simulation Data Inspector</b> .....	<b>4-31</b>
From the UI .....	4-31
From the Command Line .....	4-31
<b>Replacing Functions Using Lookup Table Approximations</b> .....	<b>4-32</b>
<b>Replace a Custom Function with a Lookup Table</b> .....	<b>4-33</b>
Using the HDL Coder App .....	4-33
From the Command Line .....	4-36
<b>Replace the <code>exp</code> Function with a Lookup Table</b> .....	<b>4-39</b>
From the UI .....	4-39
From the Command Line .....	4-42
<b>Data Type Issues in Generated Code</b> .....	<b>4-45</b>
Enable the Highlight Option in a Project .....	4-45
Enable the Highlight Option at the Command Line .....	4-45
Stowaway Doubles .....	4-45
Stowaway Singles .....	4-45

Expensive Fixed-Point Operations . . . . .	<b>4-45</b>
<b>Working with Fixed-Point Code . . . . .</b>	<b>4-47</b>
<b>Floating-Point to Fixed-Point Conversion . . . . .</b>	<b>4-49</b>
<b>Fixed-Point Type Conversion and Refinement . . . . .</b>	<b>4-59</b>
<b>Working with Generated Fixed-Point Files . . . . .</b>	<b>4-66</b>
<b>Fixed-Point Type Conversion and Derived Ranges . . . . .</b>	<b>4-72</b>
<b>Generate HDL-compatible lookup table function replacements using 'coder.approximate'</b> . . . . .	<b>4-77</b>

## Code Generation

# 5

---

<b>Create and Set Up Your Project . . . . .</b>	<b>5-2</b>
Create a New Project . . . . .	5-2
Open an Existing Project . . . . .	5-3
Add Files to the Project . . . . .	5-3
<b>Specify Properties of Entry-Point Function Inputs . . . . .</b>	<b>5-4</b>
When to Specify Input Properties . . . . .	5-4
Why You Must Specify Input Properties . . . . .	5-4
Properties to Specify . . . . .	5-4
Rules for Specifying Properties of Primary Inputs . . . . .	5-5
Methods for Defining Properties of Primary Inputs . . . . .	5-5
<b>Code Generation Reports . . . . .</b>	<b>5-7</b>
Report Generation . . . . .	5-7
Report Location . . . . .	5-7
Errors and Warnings . . . . .	5-8
Files and Functions . . . . .	5-8
MATLAB Source . . . . .	5-8
MATLAB Variables . . . . .	5-9
Additional Reports . . . . .	5-10
Report Limitations . . . . .	5-10
<b>Generate Instantiable Code for Functions . . . . .</b>	<b>5-11</b>
How to Generate Instantiable Code for Functions . . . . .	5-11
Generate Code Inline for Specific Functions . . . . .	5-11
Limitations for Instantiable Code Generation for Functions . . . . .	5-11
<b>Integrate Custom HDL Code Into MATLAB Design . . . . .</b>	<b>5-12</b>
Define the <code>hdl.BlackBox</code> System object . . . . .	5-12
Use System object In MATLAB Design Function . . . . .	5-13
Generate HDL Code . . . . .	5-14
Limitations for <code>hdl.BlackBox</code> . . . . .	5-16

<b>Enable MATLAB Function Block Generation</b> . . . . .	<b>5-17</b>
Requirements for MATLAB Function Block Generation . . . . .	5-17
Enable MATLAB Function Block Generation . . . . .	5-17
Restrictions for MATLAB Function Block Generation . . . . .	5-17
Results of MATLAB Function Block Generation . . . . .	5-17
<b>System Design with HDL Code Generation from MATLAB and Simulink</b> . . . . .	<b>5-18</b>
<b>Specify the Clock Enable Rate</b> . . . . .	<b>5-21</b>
Why Specify the Clock Enable Rate? . . . . .	5-21
How to Specify the Clock Enable Rate . . . . .	5-21
<b>Specify Test Bench Clock Enable Toggle Rate</b> . . . . .	<b>5-23</b>
When to Specify Test Bench Clock Enable Toggle Rate . . . . .	5-23
How to Specify Test Bench Clock Enable Toggle Rate . . . . .	5-23
<b>Generate an HDL Coding Standard Report from MATLAB</b> . . . . .	<b>5-25</b>
Using the HDL Workflow Advisor . . . . .	5-25
Using the Command Line . . . . .	5-27
<b>Generate an HDL Lint Tool Script</b> . . . . .	<b>5-28</b>
How To Generate an HDL Lint Tool Script . . . . .	5-28
<b>Generate HDL code from MATLAB functions using automated lookup table generation</b> . . . . .	<b>5-30</b>
<b>Generate Board-Independent IP Core from MATLAB Algorithm</b> . . . . .	<b>5-35</b>
Requirements and Limitations for IP Core Generation . . . . .	5-35
Generate Board-Independent IP Core . . . . .	5-35
<b>Minimize Clock Enables</b> . . . . .	<b>5-37</b>
Using the GUI . . . . .	5-37
Using the Command Line . . . . .	5-37
Limitations . . . . .	5-38

## Verification

# 6

<b>Verify Code with HDL Test Bench</b> . . . . .	<b>6-2</b>
<b>Test Bench Generation</b> . . . . .	<b>6-5</b>
How Test Bench Generation Works . . . . .	6-5
Test Bench Data Files . . . . .	6-5
Test Bench Data Type Limitations . . . . .	6-5
Use Constants Instead of File I/O . . . . .	6-5

Generate Synthesis Scripts . . . . .	7-2
--------------------------------------	-----

<b>RAM Mapping for MATLAB Code . . . . .</b>	<b>8-2</b>
<b>Map Matrices to Block RAMs to Reduce Area . . . . .</b>	<b>8-3</b>
<b>Map Persistent Arrays and dsp.Delay to RAM . . . . .</b>	<b>8-8</b>
How To Enable RAM Mapping . . . . .	8-8
RAM Mapping Requirements for Persistent Arrays and System object Properties . . . . .	8-8
RAM Mapping Requirements for dsp.Delay System Objects . . . . .	8-10
<b>RAM Mapping Comparison for MATLAB Code . . . . .</b>	<b>8-11</b>
<b>Pipelining MATLAB Code . . . . .</b>	<b>8-12</b>
Port Registers . . . . .	8-12
Input and Output Pipeline Registers . . . . .	8-12
Operation Pipelining . . . . .	8-12
<b>Pipeline MATLAB Expressions . . . . .</b>	<b>8-13</b>
How To Pipeline a MATLAB Expression . . . . .	8-13
Limitations of Pipelining for MATLAB Expressions . . . . .	8-13
<b>Distributed Pipelining . . . . .</b>	<b>8-15</b>
What is Distributed Pipelining? . . . . .	8-15
Benefits and Costs of Distributed Pipelining . . . . .	8-15
Selected Bibliography . . . . .	8-15
<b>Distributed Pipelining for Clock Speed Optimization . . . . .</b>	<b>8-16</b>
<b>Optimize MATLAB Loops . . . . .</b>	<b>8-20</b>
Loop Streaming . . . . .	8-20
Loop Unrolling . . . . .	8-20
How to Optimize MATLAB Loops . . . . .	8-20
Limitations for MATLAB Loop Optimization . . . . .	8-21
<b>Constant Multiplier Optimization . . . . .</b>	<b>8-22</b>
What is Constant Multiplier Optimization? . . . . .	8-22
Specify Constant Multiplier Optimization . . . . .	8-22
<b>Resource Sharing of Multipliers to Reduce Area . . . . .</b>	<b>8-24</b>
<b>Loop Streaming to Reduce Area . . . . .</b>	<b>8-31</b>
<b>Constant Multiplier Optimization to Reduce Area . . . . .</b>	<b>8-36</b>

<b>HDL Workflow Advisor</b> .....	<b>9-2</b>
Overview .....	9-2
<b>MATLAB to HDL Code and Synthesis</b> .....	<b>9-6</b>
MATLAB to HDL Code Conversion .....	9-6
Code Generation: Target Tab .....	9-6
Code Generation: Coding Style Tab .....	9-7
Code Generation: Clocks and Ports Tab .....	9-8
Code Generation: Test Bench Tab .....	9-10
Code Generation: Optimizations Tab .....	9-11
Simulation and Verification .....	9-12
Synthesis and Analysis .....	9-13

**HDL Code Generation from Simulink****Model Design for HDL Code Generation**

<b>Signal and Data Type Support</b> .....	<b>10-2</b>
Buses .....	10-2
Enumerations .....	10-3
Matrices .....	10-3
Unsupported Signal and Data Types .....	10-6
<b>Use Simulink Templates for HDL Code Generation</b> .....	<b>10-7</b>
Create Model Using HDL Coder Model Template .....	10-7
HDL Coder Model Templates .....	10-7
<b>Generate DUT Ports for Tunable Parameters</b> .....	<b>10-17</b>
Prerequisites .....	10-17
Create and Add Tunable Parameter That Maps to DUT Ports .....	10-17
Generated Code .....	10-18
Limitations .....	10-18
Use Tunable Parameter in Other Blocks .....	10-18
<b>Generate Parameterized Code for Referenced Models</b> .....	<b>10-20</b>
Parameterize Referenced Model for HDL Code Generation .....	10-20
Restrictions .....	10-20
<b>Generating HDL Code for Subsystems with Array of Buses</b> .....	<b>10-21</b>
How HDL Coder Generates Code for Array of Buses .....	10-21
Array of Buses Limitations .....	10-23
<b>Implement Control Signals Based Mathematical Functions using HDL Coder</b> .....	<b>10-24</b>

<b>Using ForEach Subsystems in HDL Coder . . . . .</b>	<b>10-47</b>
<b>Generate HDL Code for Blocks Inside For Each Subsystem . . . . .</b>	<b>10-51</b>
<b>Field-Oriented Control of a Permanent Magnet Synchronous Machine . . . . .</b>	<b>10-55</b>
<b>Model and Debug Test Point Signals with HDL Coder . . . . .</b>	<b>10-59</b>
<b>Allocate Sufficient Delays for Floating-Point Operations . . . . .</b>	<b>10-67</b>
Problem . . . . .	10-67
Cause . . . . .	10-67
Solution . . . . .	10-68
<b>Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials . . . . .</b>	<b>10-72</b>
Issue . . . . .	10-72
Description . . . . .	10-72
Recommendations . . . . .	10-74
<b>Getting Started with HDL Coder Native Floating-Point Support . . . . .</b>	<b>10-80</b>
Key Features . . . . .	10-80
Numeric Considerations and IEEE-754 Standard Compliance . . . . .	10-80
Floating Point Types . . . . .	10-81
Data Type Considerations . . . . .	10-82
<b>Numeric Considerations with Native Floating-Point . . . . .</b>	<b>10-84</b>
Round to Nearest Rounding Mode . . . . .	10-84
Denormal Numbers . . . . .	10-84
Exception Handling . . . . .	10-85
Relative Accuracy and ULP Considerations . . . . .	10-85
<b>ULP Considerations of Native Floating-Point Operators . . . . .</b>	<b>10-88</b>
Adherence of Native Floating Point Operators to IEEE-754 Standard . . . . .	10-88
ULP Values of Floating Point Operators . . . . .	10-88
Considerations . . . . .	10-89
<b>Latency Values of Floating Point Operators . . . . .</b>	<b>10-91</b>
Math Operations . . . . .	10-91
Trigonometric and Exponential Operations . . . . .	10-93
Comparisons and Conversions . . . . .	10-94
<b>Latency Considerations with Native Floating Point . . . . .</b>	<b>10-96</b>
<b>Generate Target-Independent HDL Code with Native Floating-Point . . . . .</b>	<b>10-103</b>
How HDL Coder Generates Target-Independent HDL Code . . . . .	10-103
Enable Native Floating Point and Generate Code . . . . .	10-104
View Code Generation Report . . . . .	10-105
Analyze Results . . . . .	10-106
Limitation . . . . .	10-108
<b>Floating Point Support: Field-Oriented Control Algorithm . . . . .</b>	<b>10-109</b>

<b>Verify the Generated Code from Native Floating-Point .....</b>	<b>10-116</b>
Specify the Tolerance Strategy .....	10-116
Verify the Generated Code with HDL Test Bench .....	10-117
Verify the Generated Code with Cosimulation .....	10-117
Limitation .....	10-119
<b>Simulink Blocks Supported with Native Floating-Point .....</b>	<b>10-120</b>
HDL Floating Point Operations Library .....	10-120
Supported Simulink Blocks in Math Operations Library .....	10-120
Supported Simulink Blocks in Other Libraries .....	10-121
Simulink Block Restrictions .....	10-123
<b>Supported Data Types and Scope .....</b>	<b>10-125</b>
Supported Data Types .....	10-125
Unsupported Data Types .....	10-126
Scope for Variables .....	10-126
<b>Import Verilog Code and Generate Simulink Model .....</b>	<b>10-127</b>
HDL Import .....	10-127
HDL Import Requirements .....	10-127
How to Import HDL Code .....	10-127
Model Location .....	10-128
Errors and Warnings .....	10-128
Limitations of Verilog HDL Import .....	10-128
<b>Supported Verilog Constructs for HDL Import .....</b>	<b>10-130</b>
Module Definition and Instantiations .....	10-130
Data Types and Vectors .....	10-131
Identifiers and Comments .....	10-131
Assignments .....	10-132
Operators .....	10-132
Conditional and Looping Statements .....	10-133
Procedural Blocks and Events .....	10-133
Other Constructs .....	10-133
<b>Verilog Dataflow Modeling with HDL Import .....</b>	<b>10-135</b>
Supported Verilog Dataflow Patterns .....	10-135
Unsupported Verilog Dataflow Patterns .....	10-137
<b>Simulate and Generate HDL Code for the Float Typecast Block .....</b>	<b>10-146</b>
<b>Generate Simulink® Model From CORDIC Atan2 Verilog® Code .....</b>	<b>10-148</b>

## Simulink to HDL Examples for Communication and Signal Processing Applications

# 11

<b>Programmable FIR Filter for FPGA .....</b>	<b>11-2</b>
<b>Multichannel FIR Filter for FPGA .....</b>	<b>11-8</b>

<b>Implement FFT for FPGA Using FFT HDL Optimized Block . . . . .</b>	<b>11-11</b>
<b>High Throughput Channelizer for FPGA . . . . .</b>	<b>11-15</b>
<b>HDL Implementation of a Digital Down-Converter for LTE . . . . .</b>	<b>11-23</b>
<b>HDL Optimized QPSK Transmitter . . . . .</b>	<b>11-42</b>
<b>HDL Optimized QPSK Receiver with Captured Data . . . . .</b>	<b>11-49</b>
<b>HDL Optimized QAM Transmitter and Receiver . . . . .</b>	<b>11-59</b>
<b>Airplane Tracking with ADS-B Captured Data . . . . .</b>	<b>11-78</b>
<b>HDL Code Generation for Viterbi Decoder . . . . .</b>	<b>11-85</b>
<b>Design Video Processing Algorithms for HDL in Simulink . . . . .</b>	<b>11-91</b>
<b>Edge Detection and Image Overlay . . . . .</b>	<b>11-98</b>
<b>Lane Detection . . . . .</b>	<b>11-103</b>

## Code Generation Options in the HDL Coder Dialog Boxes

# 12

---

<b>Set HDL Code Generation Options . . . . .</b>	<b>12-2</b>
HDL Code Generation Options in the Configuration Parameters Dialog Box . . . . .	12-2
HDL Code Tab in Simulink Toolbar . . . . .	12-3
HDL Code Options in the Block Context Menu . . . . .	12-4
The HDL Block Properties Dialog Box . . . . .	12-5
<b>HDL Code Generation Options in Configuration Parameters Dialog Box . . . . .</b>	<b>12-6</b>
HDL Code Generation Pane: Target . . . . .	12-7
HDL Code Generation Pane: Optimization . . . . .	12-7
HDL Code Generation Pane: Floating Point . . . . .	12-7
HDL Code Generation Pane: Global Settings . . . . .	12-7
HDL Code Generation Pane: Report . . . . .	12-9
HDL Code Generation Pane: Testbench . . . . .	12-9
HDL Code Generation Pane: EDA Tool Scripts . . . . .	12-9
<b>Generate HDL Code from Simulink Model Using Configuration Parameters . . . . .</b>	<b>12-11</b>
FIR Filter Model . . . . .	12-11
Create a Folder and Copy Relevant Files . . . . .	12-12
Open HDL Code Generation Pane of Configuration Parameters Dialog Box . . . . .	12-13
Generate HDL Code . . . . .	12-13
<b>Generate HDL Code from Simulink Model from Command Line . . . . .</b>	<b>12-15</b>
FIR Filter Model . . . . .	12-15

Create a Folder and Copy Relevant Files .....	12-16
Generate HDL Code .....	12-17

## HDL Code Generation Pane: General

**13**

<b>HDL Code Generation Top-Level Pane Overview</b> .....	13-2
Buttons in the HDL Code Generation Top-Level Pane .....	13-2
<b>Target Language and Folder Selection Parameters</b> .....	13-3
Generate HDL for .....	13-3
Language .....	13-3
Folder .....	13-4
Restore Model Defaults .....	13-5
Run Compatibility Checker .....	13-5
Generate .....	13-6

## HDL Code Generation Pane: Target

**14**

<b>Target Overview</b> .....	14-2
<b>Tool and Device Parameters</b> .....	14-3
Synthesis Tool .....	14-3
Family .....	14-4
Device .....	14-5
Package .....	14-5
Speed .....	14-6
<b>Target Frequency Parameter</b> .....	14-8
Settings .....	14-8
Command-Line Information .....	14-9
See Also .....	14-9

## HDL Code Generation Pane: Optimization

**15**

<b>Optimization Overview</b> .....	15-2
<b>Delay Balancing and General Optimization Parameters</b> .....	15-3
Balance delays .....	15-3
Transform non zero initial value delay .....	15-4
Multiplier partitioning threshold .....	15-5
Remove Unused Ports .....	15-6

<b>RAM Mapping Parameters</b> . . . . .	<b>15-7</b>
Map pipeline delays to RAM . . . . .	<b>15-7</b>
RAM mapping threshold (bits) . . . . .	<b>15-8</b>
<b>Pipelining Parameters</b> . . . . .	<b>15-9</b>
Hierarchical distributed pipelining . . . . .	<b>15-9</b>
Distributed pipelining priority . . . . .	<b>15-10</b>
Clock-rate pipelining . . . . .	<b>15-11</b>
Allow clock-rate pipelining of DUT output ports . . . . .	<b>15-12</b>
Adaptive pipelining . . . . .	<b>15-13</b>
Preserve design delays . . . . .	<b>15-14</b>
<b>Resource Sharing Parameters for Adders and Multipliers</b> . . . . .	<b>15-15</b>
Share Adders . . . . .	<b>15-15</b>
Adder sharing minimum bitwidth . . . . .	<b>15-16</b>
Share Multipliers . . . . .	<b>15-17</b>
Multiplier sharing minimum bitwidth . . . . .	<b>15-18</b>
Multiplier promotion threshold . . . . .	<b>15-19</b>
Share Multiply-Add blocks . . . . .	<b>15-20</b>
Multiply-Add block sharing minimum bitwidth . . . . .	<b>15-21</b>
<b>Resource Sharing Parameters for Subsystems and Floating-Point IPs</b> . . . . .	<b>15-23</b>
Share Atomic subsystems . . . . .	<b>15-23</b>
Share MATLAB Function blocks . . . . .	<b>15-24</b>
Share Floating-Point IPs . . . . .	<b>15-25</b>
<b>Multicycle Path Constraints Parameters</b> . . . . .	<b>15-27</b>
Enable based constraints . . . . .	<b>15-27</b>
Register-to-register path info . . . . .	<b>15-28</b>

## HDL Code Generation Pane: Floating Point

# 16

<b>Floating Point Overview</b> . . . . .	<b>16-2</b>
<b>Floating Point IP Library</b> . . . . .	<b>16-3</b>
Settings . . . . .	<b>16-3</b>
Command-Line Information . . . . .	<b>16-3</b>
See Also . . . . .	<b>16-3</b>
<b>Native Floating Point Parameters</b> . . . . .	<b>16-4</b>
Latency Strategy . . . . .	<b>16-4</b>
Handle Denormals . . . . .	<b>16-5</b>
Mantissa Multiplier Strategy . . . . .	<b>16-6</b>
<b>FPGA Floating-Point Library Targeting Parameters</b> . . . . .	<b>16-8</b>
Initialize IP Pipelines To Zero . . . . .	<b>16-8</b>
Latency Strategy . . . . .	<b>16-9</b>
Objective . . . . .	<b>16-9</b>
IP Settings . . . . .	<b>16-10</b>

<b>Global Settings Overview</b> . . . . .	17-3
<b>Clock Settings and Timing Controller Postfix Parameters</b> . . . . .	17-4
Clock input port . . . . .	17-4
Clock inputs . . . . .	17-5
Clock edge . . . . .	17-5
Clocked process postfix . . . . .	17-6
Timing controller postfix . . . . .	17-7
<b>Reset Settings and Parameters</b> . . . . .	17-8
Reset type . . . . .	17-8
Reset asserted level . . . . .	17-9
Reset input port . . . . .	17-10
<b>Clock Enable Settings and Parameters</b> . . . . .	17-12
Clock enable input port . . . . .	17-12
Enable prefix . . . . .	17-13
<b>Oversampling factor</b> . . . . .	17-15
Settings . . . . .	17-15
Dependency . . . . .	17-15
Command-Line Information . . . . .	17-15
See Also . . . . .	17-16
<b>Comment in header</b> . . . . .	17-17
Settings . . . . .	17-17
Command-Line Information . . . . .	17-17
See Also . . . . .	17-17
<b>Language-Specific File Extension Parameters</b> . . . . .	17-19
Verilog file extension . . . . .	17-19
VHDL file extension . . . . .	17-19
<b>Language-Specific Identifiers and Postfix Parameters</b> . . . . .	17-21
Entity conflict postfix . . . . .	17-21
Package postfix . . . . .	17-21
Reserved word postfix . . . . .	17-22
Module name prefix . . . . .	17-23
Pipeline postfix . . . . .	17-23
<b>Split entity and architecture Parameters</b> . . . . .	17-25
Split entity file postfix . . . . .	17-25
Split arch file postfix . . . . .	17-25
Split entity and architecture . . . . .	17-26
<b>Complex Signals Postfix Parameters</b> . . . . .	17-28
Complex real part postfix . . . . .	17-28
Complex imaginary part postfix . . . . .	17-28

<b>VHDL Architecture and Library Name and Code for Model Reference</b>	
<b>Parameters</b>	17-30
VHDL architecture name	17-30
VHDL library name	17-30
Generate VHDL code for model references into a single library	17-31
<b>Generate Statement and Vector and Component Instance Label</b>	
<b>Parameters</b>	17-32
Block generate label	17-32
Output generate label	17-32
Instance generate label	17-32
Vector prefix	17-33
Instance postfix	17-33
Instance prefix	17-33
Map file postfix	17-34
<b>Input and Output Port and Clock Enable Output Type Parameters</b>	
Input data type	17-35
Output data type	17-35
Clock Enable output port	17-36
<b>Minimize Clock Enables and Reset Signal Parameters</b>	17-37
Minimize clock enables	17-37
Minimize global resets	17-38
<b>Using Trigger Signals and Scalarization and Test Point DUT Port Generation Parameters</b>	17-41
Use trigger signal as clock	17-41
Enable HDL DUT port generation for test points	17-41
Scalarize ports	17-42
<b>RTL Annotation Parameters</b>	17-44
Use Verilog `timescale directives	17-44
Verilog timescale specification	17-44
Inline VHDL configuration	17-45
Concatenate type safe zeros	17-45
Generate obfuscated HDL code	17-46
Emit time/date stamp in header	17-47
Include requirements in block comments	17-47
<b>RTL Customization Parameters for Constants and MATLAB Function Blocks</b>	17-49
Inline MATLAB Function block code	17-49
Represent constant values by aggregates	17-49
<b>RTL Customization Parameters for RAMs</b>	17-51
Initialize all RAM blocks	17-51
RAM Architecture	17-51
<b>No-reset registers initialization</b>	17-53
Settings	17-53
Usage Notes	17-53
Command-Line Information	17-54
See Also	17-54

<b>RTL Style Parameters</b> . . . . .	<b>17-55</b>
Use “rising_edge/falling_edge” style for registers . . . . .	<b>17-55</b>
Minimize intermediate signals . . . . .	<b>17-56</b>
Unroll for Generate Loops in VHDL code . . . . .	<b>17-56</b>
Generate parameterized HDL code from masked subsystem . . . . .	<b>17-57</b>
Enumerated Type Encoding Scheme . . . . .	<b>17-58</b>
<b>Timing Controller Settings</b> . . . . .	<b>17-60</b>
Optimize timing controller . . . . .	<b>17-60</b>
Timing controller architecture . . . . .	<b>17-60</b>
<b>File Comment Customization Parameters</b> . . . . .	<b>17-62</b>
Custom File Header Comment . . . . .	<b>17-62</b>
Custom File Footer Comment . . . . .	<b>17-62</b>
<b>Choose Coding Standard and Report Option Parameters</b> . . . . .	<b>17-64</b>
HDL coding standard . . . . .	<b>17-64</b>
Do not show passing rules in coding standard report . . . . .	<b>17-65</b>
<b>Basic Coding Practices Parameters</b> . . . . .	<b>17-66</b>
Check for duplicate names . . . . .	<b>17-66</b>
Check for HDL keywords in design names . . . . .	<b>17-67</b>
Check module, instance, entity name length . . . . .	<b>17-67</b>
Check signal, port, and parameter name length . . . . .	<b>17-69</b>
<b>RTL Description Rules for clock enables and resets Parameters</b> . . . . .	<b>17-71</b>
Check for clock enable signals . . . . .	<b>17-71</b>
Detect usage of reset signals . . . . .	<b>17-72</b>
Detect usage of asynchronous reset signals . . . . .	<b>17-73</b>
<b>RTL Description Rules for Conditional Parameters</b> . . . . .	<b>17-74</b>
Check for conditional statements in processes . . . . .	<b>17-74</b>
Check if-else statement chain length . . . . .	<b>17-75</b>
Check if-else statement nesting depth . . . . .	<b>17-76</b>
<b>Other RTL Description Rule Parameters</b> . . . . .	<b>17-77</b>
Minimize use of variables . . . . .	<b>17-77</b>
Check for initial statements that set RAM initial values . . . . .	<b>17-78</b>
Check multiplier width . . . . .	<b>17-78</b>
<b>RTL Design Rule Parameters</b> . . . . .	<b>17-80</b>
Check for non-integer constants . . . . .	<b>17-80</b>
Check line length . . . . .	<b>17-81</b>
<b>Model Generation Parameters for HDL Code</b> . . . . .	<b>17-82</b>
Generated model . . . . .	<b>17-82</b>
Validation model . . . . .	<b>17-83</b>
<b>Naming and Layout Options for Model Generation</b> . . . . .	<b>17-85</b>
Prefix for generated model name . . . . .	<b>17-85</b>
Suffix for validation model name . . . . .	<b>17-85</b>
Auto block placement . . . . .	<b>17-86</b>
Auto signal routing . . . . .	<b>17-87</b>
Inter-block horizontal scaling . . . . .	<b>17-87</b>
Inter-block vertical scaling . . . . .	<b>17-88</b>

<b>Diagnostic Parameters for Optimizations</b> .....	<b>17-89</b>
Highlight feedback loops inhibiting delay balancing and optimizations .....	<b>17-89</b>
Highlight blocks inhibiting clock-rate pipelining .....	<b>17-90</b>
Highlight blocks inhibiting distributed pipelining .....	<b>17-91</b>
<b>Diagnostic Parameters for Reals and Black Box Interfaces</b> .....	<b>17-92</b>
Check for name conflicts in black box interfaces .....	<b>17-92</b>
Check for presence of reals in generated HDL code .....	<b>17-93</b>
<b>Code Generation Output Parameter</b> .....	<b>17-94</b>
Generate HDL code .....	<b>17-94</b>

## HDL Code Generation Pane: Report

**18**

<b>Report Pane Overview</b> .....	<b>18-2</b>
See Also .....	<b>18-2</b>
<b>Code Generation Report Parameters</b> .....	<b>18-3</b>
Generate traceability report .....	<b>18-3</b>
Traceability style .....	<b>18-4</b>
Generate model Web view .....	<b>18-5</b>
Generate resource utilization report .....	<b>18-6</b>
Generate high-level timing critical path report .....	<b>18-7</b>
Generate optimization report .....	<b>18-8</b>

## HDL Code Generation Pane: Test Bench

**19**

<b>Test Bench Overview</b> .....	<b>19-2</b>
Generate Test Bench Button .....	<b>19-2</b>
<b>Test Bench Generation Output Parameters</b> .....	<b>19-3</b>
HDL test bench .....	<b>19-3</b>
Cosimulation model .....	<b>19-4</b>
SystemVerilog DPI test bench .....	<b>19-5</b>
Simulation tool .....	<b>19-6</b>
HDL code coverage .....	<b>19-7</b>
<b>Test Bench Postfix Parameters</b> .....	<b>19-8</b>
Test bench name postfix .....	<b>19-8</b>
Test bench reference postfix .....	<b>19-8</b>
Test bench data file name postfix .....	<b>19-9</b>
<b>Clock and Reset Input Parameters for Testbench</b> .....	<b>19-10</b>
Force clock .....	<b>19-10</b>
Clock high time (ns) .....	<b>19-10</b>
Clock low time (ns) .....	<b>19-11</b>

Force clock enable .....	<b>19-12</b>
Clock enable delay (in clock cycles) .....	<b>19-13</b>
Force reset .....	<b>19-14</b>
Reset length (in clock cycles) .....	<b>19-14</b>
<b>Setup and Hold Time Parameters for Testbench</b> .....	<b>19-16</b>
Hold time (ns) .....	<b>19-16</b>
Setup time (ns) .....	<b>19-17</b>
<b>Test Bench Stimulus and Output Parameters</b> .....	<b>19-18</b>
Hold input data between samples .....	<b>19-18</b>
Initialize test bench inputs .....	<b>19-19</b>
Ignore output data checking (number of samples) .....	<b>19-19</b>
Use file I/O to read/write test bench data .....	<b>19-21</b>
<b>Multi-File Testbench and Simulation Library Path Parameters</b> ..	<b>19-23</b>
Multi-file test bench .....	<b>19-23</b>
Simulation library path .....	<b>19-24</b>
<b>Floating-Point Tolerance Parameters</b> .....	<b>19-26</b>
Floating point tolerance check based on .....	<b>19-26</b>
Tolerance Value .....	<b>19-26</b>

## HDL Code Generation Pane: EDA Tool Scripts

# 20

---

<b>EDA Tool Scripts Overview</b> .....	<b>20-2</b>
<b>Generate EDA scripts</b> .....	<b>20-3</b>
Settings .....	<b>20-3</b>
Command-Line Information .....	<b>20-3</b>
See Also .....	<b>20-3</b>
<b>Compilation Script Parameters</b> .....	<b>20-4</b>
Compile file postfix .....	<b>20-4</b>
Compile initialization .....	<b>20-4</b>
Compile command for VHDL .....	<b>20-5</b>
Compile command for Verilog .....	<b>20-5</b>
Compile termination .....	<b>20-6</b>
<b>Simulation Script Parameters</b> .....	<b>20-7</b>
Simulation file postfix .....	<b>20-7</b>
Simulation initialization .....	<b>20-7</b>
Simulation command .....	<b>20-8</b>
Simulation waveform viewing command .....	<b>20-8</b>
Simulation termination .....	<b>20-9</b>
Simulator flags .....	<b>20-9</b>
<b>Synthesis Script Parameters</b> .....	<b>20-11</b>
Choose synthesis tool .....	<b>20-11</b>
Synthesis file postfix .....	<b>20-12</b>
Synthesis initialization .....	<b>20-13</b>

Synthesis command .....	20-14
Synthesis termination .....	20-14
Additional files to add to synthesis project .....	20-15
<b>Lint Script Parameters .....</b>	<b>20-16</b>
Choose HDL lint tool .....	20-16
Lint initialization .....	20-17
Lint command .....	20-17
Lint termination .....	20-17

## Modeling Guidelines

# 21

<b>HDL Modeling Guidelines Severity Levels .....</b>	<b>21-2</b>
<b>Model Design and Compatibility Guidelines - By Numbered List .....</b>	<b>21-3</b>
Guidelines 1.1: Basic Settings .....	21-3
Guidelines 1.2: DUT Subsystem and Hierarchical Modeling .....	21-4
Guidelines 1.3: Guidelines for Vectors and Buses .....	21-4
Guidelines 1.4: Guidelines for Clock Bundle Signals .....	21-5
Guidelines 1.5: Modeling Guidelines for Native Floating Point .....	21-5
<b>Guidelines for Supported Blocks and Data Types - By Numbered List .....</b>	<b>21-6</b>
Guidelines 2.1: Blocks in HDL RAMs and HDL Operations Library .....	21-6
Guidelines 2.2: Blocks in Logic and Bit Operations Library .....	21-6
Guidelines 2.3: Lookup Table Blocks .....	21-6
Guidelines 2.4: Ports and Subsystems .....	21-7
Guideline 2.5: Rate Change and Constant Blocks .....	21-7
Guideline 2.6: Delay Blocks .....	21-7
Guideline 2.7: Blocks for Multiplication and Accumulation Operations .....	21-8
Guideline 2.8: MATLAB Function Blocks .....	21-8
Guideline 2.9: Stateflow Charts .....	21-8
Guidelines 2.10: Data Types .....	21-9
<b>Guidelines for Speed and Area Optimizations - By Numbered List .....</b>	<b>21-10</b>
Guidelines 3.1: Resource Sharing .....	21-10
Guidelines 3.2: Clock Rate Pipelining and Distributed Pipelining .....	21-11
<b>Basic Guidelines for Modeling HDL Algorithm in Simulink .....</b>	<b>21-12</b>
Use HDL-Supported Blocks .....	21-12
Partition Model into DUT and Test Bench .....	21-13
Avoid Using Double-Byte Characters .....	21-15
Document Model Features and Attributes .....	21-15
<b>Guidelines for Model Setup and Checking Model Compatibility .....</b>	<b>21-18</b>
Customize hdlsetup Function Based on Target Application .....	21-18
Check Subsystem for HDL Compatibility .....	21-19

Run Model Checks for HDL Coder . . . . .	<b>21-19</b>
<b>Modeling with Simulink, Stateflow, and MATLAB Function Blocks</b>	
. . . . .	<b>21-22</b>
Guideline ID . . . . .	<b>21-22</b>
Severity . . . . .	<b>21-22</b>
Description . . . . .	<b>21-22</b>
<b>Terminate Unconnected Block Outputs and Usage of Commenting Blocks</b> . . . . .	<b>21-25</b>
Terminate Unconnected Block Outputs . . . . .	<b>21-25</b>
Using Comment Out and Comment Through of Blocks . . . . .	<b>21-26</b>
<b>Identify and Programmatically Change and Display HDL Block Parameters</b> . . . . .	<b>21-29</b>
Adjust Sizes of Constant and Gain Blocks for Identifying Parameters . . . . .	<b>21-29</b>
Display Parameters that Affect HDL Code Generation . . . . .	<b>21-29</b>
Change Block Parameters by Using <code>find_system</code> and <code>set_param</code> . . . . .	<b>21-33</b>
<b>DUT Subsystem Guidelines</b> . . . . .	<b>21-34</b>
DUT Subsystem Considerations . . . . .	<b>21-34</b>
Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks . . . . .	<b>21-34</b>
Insert Handwritten Code into Simulink Modeling Environment . . . . .	<b>21-36</b>
<b>Hierarchical Modeling Guidelines</b> . . . . .	<b>21-38</b>
Avoid Constant Block Connections to Subsystem Port Boundaries . . . . .	<b>21-38</b>
Generate Parameterized HDL Code for Constant and Gain Blocks . . . . .	<b>21-39</b>
Place Physical Signal Lines Inside a Subsystem . . . . .	<b>21-41</b>
<b>Design Considerations for Matrices and Vectors</b> . . . . .	<b>21-44</b>
Modeling Requirements for Matrices . . . . .	<b>21-44</b>
Avoid Generating Ascending Bit Order in HDL Code From Vector Signals . . . . .	<b>21-46</b>
<b>Use Bus Signals to Improve Readability of Model and Generate HDL Code</b> . . . . .	<b>21-49</b>
<b>Guidelines for Clock and Reset Signals</b> . . . . .	<b>21-55</b>
Use Global Oversampling to Create Frequency-Divided Clock . . . . .	<b>21-55</b>
Create Multirate Model with Integer Clock Multiples by Clock Division . . . . .	<b>21-55</b>
Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times . . . . .	<b>21-57</b>
Asynchronous Clock Modeling in HDL Coder . . . . .	<b>21-58</b>
Use Global Reset Type Setting Based on Target Hardware . . . . .	<b>21-60</b>
<b>Modeling with Native Floating Point</b> . . . . .	<b>21-62</b>
Guideline ID . . . . .	<b>21-62</b>
Severity . . . . .	<b>21-62</b>
Description . . . . .	<b>21-62</b>

<b>Design Considerations for RAM Blocks and Blocks in HDL Operations Library</b> . . . . .	<b>21-65</b>
RAM Block Access Considerations . . . . .	21-65
Serial to Parallel Conversion . . . . .	21-67
<b>Usage of Blocks in Logic and Bit Operations Library</b> . . . . .	<b>21-69</b>
Logical and Arithmetic Bit Shift Operations . . . . .	21-69
Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks . . . . .	21-71
Use Boolean Output for Compare to Constant and Relational Operator Blocks . . . . .	21-73
<b>Generate FPGA Block RAM from Lookup Tables</b> . . . . .	<b>21-74</b>
<b>Usage of Different Subsystem Types</b> . . . . .	<b>21-77</b>
Virtual Subsystem: Use as DUT . . . . .	21-77
Atomic Subsystem: Generate Reusable HDL Files . . . . .	21-77
Variant Subsystem: Using Variant Subsystems for HDL Code Generation . . . . .	21-78
Model References: Build Model Design Using Smaller Partitions . . . . .	21-79
Block Settings of Enabled and Triggered Subsystems . . . . .	21-80
<b>Usage of Rate Change and Constant Blocks</b> . . . . .	<b>21-82</b>
Usage of Rate Conversion Blocks . . . . .	21-82
Use Discrete and Finite Sample Time for Constant Block . . . . .	21-83
<b>Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation</b> . . . . .	<b>21-85</b>
Appropriate Usage of Delay Blocks as Registers . . . . .	21-85
Required HDL Settings for Goto and From Blocks . . . . .	21-85
<b>Modeling Efficient Multiplication and Division Operations for FPGA Targeting</b> . . . . .	<b>21-87</b>
Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA . . . . .	21-87
Use ShiftAdd Architecture of Divide Block for Fixed-Point Types . . . . .	21-91
<b>Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation</b> . . . . .	<b>21-92</b>
Update Persistent Variables at End of MATLAB Function . . . . .	21-92
Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks . . . . .	21-93
Use hdlfimath Setting and Specify fi Objects inside MATLAB Function Block . . . . .	21-95
<b>Guidelines for HDL Code Generation Using Stateflow Charts</b> . . . . .	<b>21-98</b>
Choose State Machine Type based on HDL Implementation Requirements . . . . .	21-98
Specify Block Configuration Settings of Stateflow Chart . . . . .	21-98
Insert Unconditional Transition State for Else Statement in HDL Code . . . . .	21-99
<b>Simulink Data Type Considerations</b> . . . . .	<b>21-103</b>
Use Boolean for Logical Data and Ufix1 for Numerical Data . . . . .	21-103
Specify Data Type of Gain Blocks . . . . .	21-103

Enumerated Data Type Restrictions . . . . .	<b>21-104</b>
<b>Resource Sharing Settings for Various Blocks . . . . .</b>	<b>21-105</b>
Resource Sharing of Add Blocks . . . . .	21-105
Resource Sharing of Gain Blocks . . . . .	21-106
Resource Sharing of Product Blocks . . . . .	21-107
Resource Sharing of Multiply-Add Blocks . . . . .	21-107
<b>Resource Sharing of Subsystems and Floating-Point IPs . . . . .</b>	<b>21-109</b>
General Considerations for Sharing of Subsystems . . . . .	21-109
Use MATLAB Datapath Architecture for Sharing with MATLAB	
Function Blocks . . . . .	21-110
Sharing of Atomic Subsystems . . . . .	21-110
Resource Sharing of Floating-Point IPs . . . . .	21-112
<b>Distributed Pipelining and Clock-Rate Pipelining Guidelines . . . . .</b>	<b>21-114</b>
Clock-Rate Pipelining Guidelines . . . . .	21-114
Recommended Distributed Pipelining Settings . . . . .	21-114
<b>Insert Distributed Pipeline Registers for Blocks with Vector Data</b>	
<b>Type Inputs . . . . .</b>	<b>21-117</b>
Guideline ID . . . . .	21-117
Severity . . . . .	21-117
Description . . . . .	21-117

## Supported Blocks Library and Block Properties

**22**

---

<b>View HDL-Supported Blocks and HDL-Specific Block Documentation . . . . .</b>	<b>22-2</b>
View HDL-Supported Blocks and Documentation . . . . .	22-2
View HDL-Specific Block Documentation . . . . .	22-2
<b>HDL Block Properties: General . . . . .</b>	<b>22-3</b>
Overview . . . . .	22-3
AdaptivePipelining . . . . .	22-4
BalanceDelays . . . . .	22-5
ClockRatePipelining . . . . .	22-5
CodingStyle . . . . .	22-6
ConstMultiplierOptimization . . . . .	22-7
ConstrainedOutputPipeline . . . . .	22-8
DistributedPipelining . . . . .	22-8
DotProductStrategy . . . . .	22-9
DSPStyle . . . . .	22-10
FlattenHierarchy . . . . .	22-12
InputPipeline . . . . .	22-13
InstantiateFunctions . . . . .	22-13
InstantiateStages . . . . .	22-14
LoopOptimization . . . . .	22-14
LUTRegisterResetType . . . . .	22-15
MapPersistentVarsToRAM . . . . .	22-16
OutputPipeline . . . . .	22-17

RAMDirective . . . . .	22-18
ResetType . . . . .	22-21
SerialPartition . . . . .	22-22
SharingFactor . . . . .	22-23
SoftReset . . . . .	22-23
StreamingFactor . . . . .	22-24
UsePipelines . . . . .	22-24
UseRAM . . . . .	22-25
VariablesToPipeline . . . . .	22-28
<b>HDL Block Properties: Native Floating Point</b> . . . . .	22-29
Overview . . . . .	22-29
CheckResetToZero . . . . .	22-30
DivisionAlgorithm . . . . .	22-30
HandleDenormals . . . . .	22-31
InputRangeReduction . . . . .	22-32
LatencyStrategy . . . . .	22-33
CustomLatency . . . . .	22-34
NFPCustomLatency . . . . .	22-35
MantissaMultiplyStrategy . . . . .	22-36
MaxIterations . . . . .	22-37
<b>HDL Filter Block Properties</b> . . . . .	22-39
AdderTreePipeline . . . . .	22-39
AddPipelineRegisters . . . . .	22-39
ChannelSharing . . . . .	22-40
CoeffMultipliers . . . . .	22-40
DALUTPartition . . . . .	22-40
DARadix . . . . .	22-41
FoldingFactor . . . . .	22-42
MultiplierInputPipeline . . . . .	22-42
MultiplierOutputPipeline . . . . .	22-42
NumMultipliers . . . . .	22-43
ReuseAccum . . . . .	22-43
SerialPartition . . . . .	22-43
<b>HDL Filter Architectures</b> . . . . .	22-45
Fully Parallel Architecture . . . . .	22-45
Serial Architectures . . . . .	22-46
Frame-Based Architecture . . . . .	22-47
<b>Distributed Arithmetic for HDL Filters</b> . . . . .	22-50
Requirements and Considerations for Generating Distributed Arithmetic Code . . . . .	22-50
Further References . . . . .	22-51
<b>Set and View HDL Model and Block Parameters</b> . . . . .	22-52
Set HDL Block Parameters . . . . .	22-52
Set HDL Block Parameters for Multiple Blocks Programmatically . . . . .	22-52
View All HDL Block Parameters . . . . .	22-54
View Non-Default HDL Block Parameters . . . . .	22-54
View HDL Model Parameters . . . . .	22-54

<b>Pass through, No HDL, and Cascade Implementations</b> .....	<b>22-56</b>
Pass-through and No HDL Implementations .....	22-56
Cascade Architecture Best Practices .....	22-56
<b>Build a ROM Block with Simulink Blocks</b> .....	<b>22-57</b>
<b>Getting Started with RAM and ROM in Simulink®</b> .....	<b>22-58</b>
<b>Wireless Communications Design for FPGAs and ASICs</b> .....	<b>22-61</b>
From Mathematical Algorithm to Hardware Implementation .....	22-61
HDL-Optimized Blocks .....	22-63
Reference Applications .....	22-63
Generate HDL Code and Prototype on FPGA .....	22-64

## 23

### Generating HDL Code for Multirate Models

<b>Code Generation from Multirate Models</b> .....	<b>23-2</b>
Clock Enable Generation for a Multirate DUT .....	23-2
<b>Timing Controller for Multirate Models</b> .....	<b>23-4</b>
<b>Generate Reset for Timing Controller</b> .....	<b>23-5</b>
Requirements for Timing Controller Reset Port Generation .....	23-5
How To Generate Reset for Timing Controller .....	23-5
Limitations for Timing Controller Reset Port Generation .....	23-5
<b>Multirate Model Requirements for HDL Code Generation</b> .....	<b>23-6</b>
Model Configuration Parameters .....	23-6
Sample Rate .....	23-6
Blocks To Use For Rate Transitions .....	23-6
<b>Generate a Global Oversampling Clock</b> .....	<b>23-8</b>
Why Use a Global Oversampling Clock? .....	23-8
Requirements for the Oversampling Factor .....	23-8
Specifying the Oversampling Factor From the GUI .....	23-8
Specifying the Oversampling Factor From the Command Line .....	23-9
Resolving Oversampling Rate Conflicts .....	23-9
<b>Using Multiple Clocks in HDL Coder</b> .....	<b>23-12</b>
<b>Using Triggered Subsystems for HDL Code Generation</b> .....	<b>23-16</b>
Best Practices .....	23-16
Using the Signal Builder Block .....	23-16
Using Trigger As Clock .....	23-16
Requirements .....	23-17
Specify Trigger As Clock .....	23-17
Limitations .....	23-17
<b>Generate Multicycle Path Information Files</b> .....	<b>23-19</b>
Overview .....	23-19
Format and Content of a Multicycle Path Information File .....	23-20

File Naming and Location Conventions . . . . .	23-23
Generating Multicycle Path Information Files Using the GUI . . . . .	23-23
Generating Multicycle Path Information Files Using the Command Line . . . . .	23-23
Limitations . . . . .	23-24
<b>Meet Timing Requirements Using Enable-Based Multicycle Path Constraints . . . . .</b>	<b>23-26</b>
How Enable-Based Multicycle Path Constraints Work . . . . .	23-26
Specify Enable-Based Constraints . . . . .	23-27
Benefits of Using Enable-Based Constraints . . . . .	23-28
Modeling Guidelines . . . . .	23-29
Multicycle Path Constraints for Various Synthesis Tools . . . . .	23-29
Caveats and Limitations . . . . .	23-30
<b>Use Multicycle Path Constraints to Meet Timing for Slow Paths . . . . .</b>	<b>23-32</b>

## 24

### Optimization

<b>Speed and Area Optimizations in HDL Coder . . . . .</b>	<b>24-2</b>
Optimizations in MATLAB HDL Code Generation . . . . .	24-2
Optimizations in Simulink HDL Code Generation . . . . .	24-2
General Optimizations . . . . .	24-4
Speed Optimizations . . . . .	24-4
Area Optimizations . . . . .	24-5
<b>Automatic Iterative Optimization . . . . .</b>	<b>24-7</b>
How Automatic Iterative Optimization Works . . . . .	24-7
Automatic Iterative Optimization Output . . . . .	24-7
Automatic Iterative Optimization Report . . . . .	24-8
Requirements for Automatic Iterative Optimization . . . . .	24-8
Limitations of Automatic Iterative Optimization . . . . .	24-8
<b>Generated Model and Validation Model . . . . .</b>	<b>24-10</b>
Generated Model . . . . .	24-10
Validation Model . . . . .	24-11
<b>Locate Numeric Differences After Speed Optimization . . . . .</b>	<b>24-13</b>
<b>Simplify Constant Operations and Reduce Design Complexity in HDL Coder™ . . . . .</b>	<b>24-17</b>
<b>Optimization with Constrained Overclocking . . . . .</b>	<b>24-22</b>
Why Constrain Overclocking? . . . . .	24-22
Optimizations that Overclock Resources . . . . .	24-22
How to Use Constrained Overclocking . . . . .	24-22
Constrained Overclocking Limitations . . . . .	24-23
<b>Resolve Numerical Mismatch with Delay Balancing . . . . .</b>	<b>24-24</b>

<b>Streaming</b> . . . . .	<b>24-29</b>
What Is Streaming? . . . . .	24-29
Specify Streaming . . . . .	24-29
How to Determine Streaming Factor and Sample Time . . . . .	24-30
Determine Blocks That Support Streaming . . . . .	24-30
Requirements for Streaming Subsystems . . . . .	24-30
Streaming Report . . . . .	24-31
<b>Resource Sharing</b> . . . . .	<b>24-32</b>
How Resource Sharing Works . . . . .	24-32
Benefits and Costs of Resource Sharing . . . . .	24-32
Shareable Resources in Different Blocks . . . . .	24-33
Specify Resource Sharing . . . . .	24-33
Limitations for Resource Sharing . . . . .	24-33
Block Requirements for Resource Sharing . . . . .	24-34
Resource Sharing Report . . . . .	24-34
<b>Streaming: Area Optimization</b> . . . . .	<b>24-36</b>
<b>Resource Sharing For Area Optimization</b> . . . . .	<b>24-40</b>
<b>Single-rate Resource Sharing Architecture</b> . . . . .	<b>24-50</b>
<b>Improve Resource Sharing with Design Modifications</b> . . . . .	<b>24-53</b>
<b>Improve Resource Sharing with Clone Detection and Replacement</b> . . . . .	<b>24-58</b>
<b>Delay Balancing</b> . . . . .	<b>24-63</b>
Why Use Delay Balancing? . . . . .	24-63
Specify Delay Balancing . . . . .	24-63
Delay Balancing Limitations . . . . .	24-64
Delay Balancing Report . . . . .	24-66
<b>Delay Balancing and Validation Model Workflow In HDL Coder™</b> . . . . .	<b>24-68</b>
<b>Control the Scope of Delay Balancing</b> . . . . .	<b>24-75</b>
<b>Delay Balancing on multi-rate designs</b> . . . . .	<b>24-82</b>
<b>Find Feedback Loops</b> . . . . .	<b>24-90</b>
Specify Highlighting of Feedback Loops . . . . .	24-90
Remove Highlighting . . . . .	24-90
Limitations . . . . .	24-91
<b>Hierarchy Flattening</b> . . . . .	<b>24-92</b>
What Is Hierarchy Flattening? . . . . .	24-92
When to Flatten Hierarchy . . . . .	24-92
Considerations . . . . .	24-92
How to Flatten Hierarchy . . . . .	24-93
Limitations for Hierarchy Flattening . . . . .	24-93
Hierarchy Flattening Report . . . . .	24-94
<b>RAM Mapping for Simulink Models</b> . . . . .	<b>24-95</b>

<b>RAM Mapping With the MATLAB Function Block</b> . . . . .	<b>24-96</b>
<b>Distributed Pipelining</b> . . . . .	<b>24-101</b>
What Is Distributed Pipelining? . . . . .	24-101
Benefits and Costs of Distributed Pipelining . . . . .	24-102
Requirements for Distributed Pipelining . . . . .	24-102
Specify Distributed Pipelining . . . . .	24-103
Limitations of Distributed Pipelining . . . . .	24-103
Distributed Pipelining Report . . . . .	24-104
Selected Bibliography . . . . .	24-104
<b>Hierarchical Distributed Pipelining</b> . . . . .	<b>24-105</b>
What Is Hierarchical Distributed Pipelining? . . . . .	24-105
How Hierarchical Distributed Pipelining Works . . . . .	24-105
Benefits of Hierarchical Distributed Pipelining . . . . .	24-106
Specify Hierarchical Distributed Pipelining . . . . .	24-107
Limitations of Hierarchical Distributed Pipelining . . . . .	24-107
Hierarchical Distributed Pipelining Report . . . . .	24-107
Selected Bibliography . . . . .	24-107
<b>Distributed Pipelining: Speed Optimization</b> . . . . .	<b>24-108</b>
<b>Constrained Output Pipelining</b> . . . . .	<b>24-112</b>
What Is Constrained Output Pipelining? . . . . .	24-112
When to Use Constrained Output Pipelining . . . . .	24-112
Requirements for Constrained Output Pipelining . . . . .	24-112
Specify Constrained Output Pipelining . . . . .	24-112
Limitations of Constrained Output Pipelining . . . . .	24-113
<b>Clock-Rate Pipelining</b> . . . . .	<b>24-114</b>
Rationale for Clock-Rate Pipelining . . . . .	24-114
How Clock-Rate Pipelining Works . . . . .	24-114
Clock-Rate Pipelining and Hierarchy Flattening . . . . .	24-115
Clock-Rate Pipelining for DUT Output Ports . . . . .	24-115
Best Practices for Clock-Rate Pipelining . . . . .	24-116
Specify Clock-Rate Pipelining . . . . .	24-116
Limitations for Clock-Rate Pipelining . . . . .	24-116
<b>Clock Rate Pipelining</b> . . . . .	<b>24-118</b>
<b>Adaptive Pipelining</b> . . . . .	<b>24-130</b>
Requirements . . . . .	24-130
Specify Adaptive Pipelining . . . . .	24-130
Supported Blocks . . . . .	24-131
Pipeline Insertion for Lookup Tables . . . . .	24-131
Pipeline Insertion for Rate Transition and Downsample Blocks . . . . .	24-132
Pipeline Insertion for Product and Gain Blocks . . . . .	24-132
Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks . . . . .	24-133
Pipeline Insertion for MATLAB Function Blocks . . . . .	24-135
Adaptive Pipelining Report . . . . .	24-136
<b>Critical Path Estimation Without Running Synthesis</b> . . . . .	<b>24-137</b>
How Critical Path Estimation Works . . . . .	24-137
How to Use Critical Path Estimation . . . . .	24-138

Characterized Blocks .....	<b>24-140</b>
Caveats .....	<b>24-143</b>
<b>HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture .....</b>	<b>24-146</b>
<b>Subsystem Optimizations for Filters .....</b>	<b>24-156</b>
Sharing .....	<b>24-156</b>
Streaming .....	<b>24-156</b>
Pipelining .....	<b>24-156</b>
Area Reduction of Multichannel Filter Subsystem .....	<b>24-157</b>
Area Reduction of Filter Subsystem .....	<b>24-162</b>
<b>Remove Redundant Logic and Unused Blocks in Generated HDL Code .....</b>	<b>24-166</b>
<b>Optimize Unconnected Ports in Generated HDL Code for Simulink Models .....</b>	<b>24-188</b>

## **Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation**

**25**

<b>Create and Use Code Generation Reports .....</b>	<b>25-2</b>
Report Generation .....	<b>25-2</b>
Code Generation Report .....	<b>25-2</b>
Summary .....	<b>25-2</b>
Code Interface Report .....	<b>25-2</b>
Timing and Area Report .....	<b>25-3</b>
Optimization Report .....	<b>25-3</b>
<b>Navigate Between Simulink Model and HDL Code by Using Traceability .....</b>	<b>25-4</b>
How Traceability Works .....	<b>25-4</b>
Generate Traceability Report .....	<b>25-5</b>
Report Location .....	<b>25-5</b>
View the Traceability Report .....	<b>25-6</b>
Code-to-Model Navigation .....	<b>25-6</b>
Model-to-Code Navigation .....	<b>25-8</b>
Traceability Report Limitations .....	<b>25-9</b>
<b>Web View of Model in Code Generation Report .....</b>	<b>25-10</b>
About Model Web View .....	<b>25-10</b>
Generate HTML Code Generation Report with Model Web View ..	<b>25-10</b>
Model Web View Limitations .....	<b>25-11</b>
<b>Generate Code with Annotations or Comments .....</b>	<b>25-13</b>
Simulink Annotations .....	<b>25-13</b>
Signal Descriptions .....	<b>25-13</b>
Text Comments .....	<b>25-13</b>
Requirements Comments and Hyperlinks .....	<b>25-14</b>

<b>Check Your Model for HDL Compatibility</b> . . . . .	<b>25-16</b>
<b>Show Blocks Supported for HDL Code Generation</b> . . . . .	<b>25-18</b>
Show Supported Blocks in Library Browser . . . . .	25-18
Reset Library Browser to Show All Blocks . . . . .	25-19
Generate a Supported Blocks Report . . . . .	25-20
<b>Trace Code Using the Mapping File</b> . . . . .	<b>25-21</b>
<b>Add or Remove the HDL Configuration Component</b> . . . . .	<b>25-24</b>
What Is the HDL Configuration Component? . . . . .	25-24
Adding the HDL Coder Configuration Component To a Model . . . . .	25-24
Removing the HDL Coder Configuration Component From a Model . . . . .	25-24

## HDL Coding Standards

26

<b>HDL Coding Standard Report</b> . . . . .	<b>26-2</b>
Rule Summary . . . . .	26-2
Rule Hierarchy . . . . .	26-2
Rule and Report Customization . . . . .	26-3
How to Fix Warnings and Errors . . . . .	26-3
<b>HDL Coding Standards</b> . . . . .	<b>26-4</b>
<b>Generate an HDL Coding Standard Report from Simulink</b> . . . . .	<b>26-5</b>
Using the HDL Workflow Advisor . . . . .	26-5
Using the Command Line . . . . .	26-7
<b>Basic Coding Practices</b> . . . . .	<b>26-9</b>
1.A General Naming Conventions . . . . .	26-10
1.B General Guidelines for Clocks and Resets . . . . .	26-15
1.C Guidelines for Initial Reset . . . . .	26-15
1.D Guidelines for Clocks . . . . .	26-16
1.F Guidelines for Hierarchical Design . . . . .	26-17
<b>RTL Description Techniques</b> . . . . .	<b>26-18</b>
2.A Guidelines for Combinational Logic . . . . .	26-18
2.B Guidelines for “Always” Constructs of Combinational Logic . . . . .	26-23
2.C Guidelines for Flip-Flop Inference . . . . .	26-25
2.D Guidelines for Latch Description . . . . .	26-28
2.E Guidelines for Tristate Buffer . . . . .	26-29
2.F Guidelines for Always/Process Construct with Circuit Structure into Account . . . . .	26-30
2.G Guidelines for “IF” Statement Description . . . . .	26-30
2.H Guidelines for “CASE” Statement Description . . . . .	26-32
2.I Guidelines for “FOR” Statement Description . . . . .	26-35
2.J Guidelines for Operator Description . . . . .	26-36
2.K Guidelines for Finite State Machine Description . . . . .	26-39

<b>RTL Design Methodology Guidelines</b> . . . . .	<b>26-41</b>
3.A Guidelines for Creating Function Libraries . . . . .	<b>26-41</b>
3.B Guidelines for Using Function Libraries . . . . .	<b>26-42</b>
3.C Guidelines for Test Facilitation Design . . . . .	<b>26-43</b>
<b>Generate an HDL Lint Tool Script</b> . . . . .	<b>26-45</b>
How to Generate an HDL Lint Tool Script . . . . .	<b>26-45</b>

# 27

## Interfacing Subsystems and Models to HDL Code

<b>Model Referencing for HDL Code Generation</b> . . . . .	<b>27-2</b>
Benefits of Model Referencing for Code Generation . . . . .	<b>27-2</b>
How To Generate Code for a Referenced Model . . . . .	<b>27-2</b>
Generate Code for Model Arguments . . . . .	<b>27-3</b>
Generate Comments . . . . .	<b>27-3</b>
Limitations . . . . .	<b>27-3</b>
<b>Generate Black Box Interface for Subsystem</b> . . . . .	<b>27-4</b>
What Is a Black Box Interface? . . . . .	<b>27-4</b>
Requirements . . . . .	<b>27-4</b>
Generate a Black Box Interface for a Subsystem . . . . .	<b>27-4</b>
Generate Code for a Black Box Subsystem Implementation . . . . .	<b>27-6</b>
<b>Generate Black Box Interface for Referenced Model</b> . . . . .	<b>27-8</b>
When to Generate a Black Box Interface . . . . .	<b>27-8</b>
How to Generate a Black Box Interface . . . . .	<b>27-8</b>
Caveats and Limitations . . . . .	<b>27-8</b>
<b>Integrate Custom HDL Code Using DocBlock</b> . . . . .	<b>27-10</b>
When To Use DocBlock to Integrate Custom Code . . . . .	<b>27-10</b>
How To Use DocBlock to Integrate Custom Code . . . . .	<b>27-10</b>
Restrictions . . . . .	<b>27-10</b>
Example . . . . .	<b>27-11</b>
<b>Customize Black Box or HDL Cosimulation Interface</b> . . . . .	<b>27-12</b>
Interface Parameters . . . . .	<b>27-12</b>
<b>Specify Bidirectional Ports</b> . . . . .	<b>27-15</b>
Requirements . . . . .	<b>27-15</b>
How To Specify a Bidirectional Port . . . . .	<b>27-15</b>
Limitations . . . . .	<b>27-15</b>
<b>Generate Reusable Code for Atomic Subsystems</b> . . . . .	<b>27-17</b>
Requirements for Generating Reusable Code for Atomic Subsystems . . . . .	<b>27-17</b>
Generate Reusable Code for Atomic Subsystems . . . . .	<b>27-17</b>
Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters . . . . .	<b>27-19</b>
<b>Scalarization of Vector Ports in Generated VHDL Code</b> . . . . .	<b>27-24</b>

<b>Create a Xilinx System Generator Subsystem</b>	27-27
Why Use Xilinx System Generator Subsystems?	27-27
Requirements for Xilinx System Generator Subsystems	27-27
How to Create a Xilinx System Generator Subsystem	27-28
Limitations for Code Generation from Xilinx System Generator Subsystems	27-28
<b>Create an Altera DSP Builder Subsystem</b>	27-29
Why Use Altera DSP Builder Subsystems?	27-29
Requirements for Altera DSP Builder Subsystems	27-29
How to Create an Altera DSP Builder Subsystem	27-29
Determine Clocking Requirements for Altera DSP Builder Subsystems	27-30
Limitations for Code Generation from Altera DSP Builder Subsystems	27-30
<b>Using Altera DSP Builder Advanced Blockset with HDL Coder</b>	27-31
<b>Using Xilinx® System Generator for DSP with HDL Coder™</b>	27-36
<b>Choose a Test Bench for Generated HDL Code</b>	27-39
<b>Generate a Cosimulation Model</b>	27-41
Requirements	27-41
What Is A Cosimulation Model?	27-41
Generating a Cosimulation Model from the GUI	27-42
Structure of the Generated Model	27-45
Launching a Cosimulation	27-50
The Cosimulation Script File	27-52
Complex and Vector Signals in the Generated Cosimulation Model	27-54
Generating a Cosimulation Model from the Command Line	27-55
Naming Conventions for Generated Cosimulation Models and Scripts	27-55
Limitations for Cosimulation Model Generation	27-56
<b>HDL Verifier Cosimulation Model Generation in HDL Coder™</b>	27-57
<b>Verify HDL Design Using SystemVerilog DPI Test Bench</b>	27-79
<b>Pass-Through and No-Op Implementations</b>	27-84
<b>Synchronous Subsystem Behavior with the State Control Block</b>	27-85
What Is a State Control Block?	27-85
State Control Block Modes	27-85
Synchronous Badge for Subsystems by Using Synchronous Mode	27-86
Generate HDL Code with the State Control Block	27-87
Enable and Reset Hardware Simulation Behavior	27-89
<b>Using the State Control block to generate more efficient code with HDL Coder™</b>	27-91
<b>Resettable Subsystem Support in HDL Coder™</b>	27-97

<b>Introduction to Stateflow HDL Code Generation</b> .....	<b>28-2</b>
Overview .....	28-2
Comments .....	28-2
Tunable Parameters .....	28-2
Example .....	28-2
Restrictions .....	28-3
<b>Hardware Realization of Stateflow Semantics</b> .....	<b>28-6</b>
<b>Generate HDL for Mealy and Moore Finite State Machines</b> .....	<b>28-7</b>
Overview .....	28-7
Generating HDL Code for a Moore Finite State Machine .....	28-7
Generating HDL for a Mealy Finite State Machine .....	28-10
<b>Design Patterns Using Advanced Chart Features</b> .....	<b>28-14</b>
Temporal Logic .....	28-14
Graphical Function .....	28-15
Hierarchy and Parallelism .....	28-16
Stateless Charts .....	28-17
Truth Tables .....	28-18
<b>Initialize Persistent Variables in MATLAB Functions</b> .....	<b>28-22</b>
MATLAB Function Block With No Direct Feedthrough .....	28-22
State Control Block in Synchronous Mode .....	28-24
Stateflow Chart Implementing Moore Semantics .....	28-25

**Generating HDL Code with the MATLAB Function Block**

<b>HDL Applications for the MATLAB Function Block</b> .....	<b>29-2</b>
Structure of Generated HDL Code .....	29-2
HDL Applications .....	29-2
<b>Viterbi Decoder with the MATLAB Function Block</b> .....	<b>29-4</b>
<b>Code Generation from a MATLAB Function Block</b> .....	<b>29-5</b>
Counter Model Using the MATLAB Function block .....	29-5
Setting Up .....	29-7
Creating the Model and Configuring General Model Settings .....	29-7
Adding a MATLAB Function Block to the Model .....	29-8
Set Fixed-Point Options for the MATLAB Function Block .....	29-8
Programming the MATLAB Function Block .....	29-10
Constructing and Connecting the DUT_eML_Block Subsystem .....	29-11
Compiling the Model and Displaying Port Data Types .....	29-13
Simulating the eml_hdl_incremoter_tut Model .....	29-13
Generating HDL Code .....	29-14

<b>Generate Instantiable Code for Functions</b> . . . . .	<b>29-17</b>
How To Generate Instantiable Code for Functions . . . . .	29-17
Generate Code Inline for Specific Functions . . . . .	29-17
Limitations for Instantiable Code Generation for Functions . . . . .	29-17
<b>MATLAB Function Block Design Patterns for HDL</b> . . . . .	<b>29-19</b>
The eml_hdl_design_patterns Library . . . . .	29-19
Efficient Fixed-Point Algorithms . . . . .	29-21
Model State Using Persistent Variables . . . . .	29-23
Creating Intellectual Property with the MATLAB Function Block . . . . .	29-24
Nontunable Parameter Arguments . . . . .	29-24
Modeling Control Logic and Simple Finite State Machines . . . . .	29-24
Modeling Counters . . . . .	29-26
Modeling Hardware Elements . . . . .	29-26
Decimal to Binary Conversion . . . . .	29-27
<b>Design Guidelines for the MATLAB Function Block</b> . . . . .	<b>29-29</b>
Use Compiled External Functions With MATLAB Function Blocks . . . . .	29-29
Build the MATLAB Function Block Code First . . . . .	29-29
Use the hdlfimath Utility for Optimized FIMATH Settings . . . . .	29-29
Use Optimal Fixed-Point Option Settings . . . . .	29-30
Set the Output Data Type of MATLAB Function Blocks Explicitly . . . . .	29-30
Using Tunable Parameters . . . . .	29-30
Run HDL Model Check for MATLAB Function Blocks . . . . .	29-30
Use MATLAB Datapath Architecture for Enhanced HDL Optimizations . . . . .	29-30
<b>CORDIC Algorithm Using the MATLAB® Function Block</b> . . . . .	<b>29-32</b>
<b>Hardware Design Patterns Using the MATLAB Function Block</b> . . . . .	<b>29-33</b>
<b>Distributed Pipeline Insertion for MATLAB Function Blocks</b> . . . . .	<b>29-37</b>

## Generating Scripts for HDL Simulators and Synthesis Tools

**30**

<b>Generate Scripts for Compilation, Simulation, and Synthesis</b> . . . . .	<b>30-2</b>
<b>Structure of Generated Script Files</b> . . . . .	<b>30-3</b>
<b>Properties for Controlling Script Generation</b> . . . . .	<b>30-4</b>
Enabling and Disabling Script Generation . . . . .	30-4
Customizing Script Names . . . . .	30-4
Customizing Script Code . . . . .	30-4
Examples . . . . .	30-6
<b>Configure Compilation, Simulation, Synthesis, and Lint Scripts</b> . . . . .	<b>30-7</b>
Compilation Script Options . . . . .	30-8
Simulation Script Options . . . . .	30-9
Synthesis Script Options . . . . .	30-11

<b>Add Synthesis Attributes</b>	<b>30-14</b>
---------------------------------	--------------

<b>Configure Synthesis Project Using Tcl Script</b>	<b>30-15</b>
-----------------------------------------------------	--------------

## Using the HDL Workflow Advisor

# 31

<b>Workflows in HDL Workflow Advisor</b>	<b>31-2</b>
Set Up HDL Workflow Advisor in MATLAB	31-2
Set Up HDL Workflow Advisor in Simulink	31-2
Generic ASIC/FPGA	31-3
FPGA Turnkey	31-4
IP Core Generation	31-4
Simulink Real-Time FPGA I/O	31-5
FPGA-in-the-Loop	31-5
<b>Getting Started with the HDL Workflow Advisor</b>	<b>31-6</b>
Open the HDL Workflow Advisor	31-6
Run Tasks in the HDL Workflow Advisor	31-7
Fix HDL Workflow Advisor Warnings or Failures	31-8
Save and Restore the HDL Workflow Advisor State	31-8
View and Save HDL Workflow Advisor Reports	31-9
<b>Generate Code and Synthesize on FPGA Using HDL Workflow Advisor</b>	<b>31-12</b>
FIR Filter Model	31-12
Create a Folder and Copy Relevant Files	31-13
Set Up Tool Path	31-14
Open the HDL Workflow Advisor	31-14
Generate HDL Code and Synthesize on FPGA	31-15
Run Workflow at Command Line with a Script	31-15
<b>Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor</b>	<b>31-17</b>
<b>Generate HDL Code for FPGA Floating-Point Target Libraries</b>	<b>31-20</b>
Setup for FPGA Floating-Point Library Mapping	31-20
Map to an FPGA Floating-Point Library	31-20
View Code Generation Reports of Floating-Point Library Mapping	31-22
Analyze Results of Floating-Point Library Mapping	31-24
<b>FPGA Floating-Point Library IP Mapping</b>	<b>31-27</b>
<b>Customize Floating-Point IP Configuration</b>	<b>31-39</b>
Customize the IP Latency with Target Frequency	31-40
Customize the IP Latency with Latency Strategy	31-43
<b>HDL Coder Support for FPGA Floating-Point Library Mapping</b>	<b>31-47</b>
Supported Blocks That Map to FPGA Floating-Point Target IP	31-47
Supported Blocks That Do Not Need to Map to FPGA Floating-Point Target IP	31-49

Limitations for FPGA Floating-Point Library Mapping .....	<b>31-50</b>
<b>Synthesis Objective to Tcl Command Mapping</b> .....	<b>31-51</b>
Altera Quartus II .....	31-51
Xilinx Vivado 2014.4 .....	31-51
Xilinx ISE 14.7 with PlanAhead .....	31-52
<b>Run HDL Workflow with a Script</b> .....	<b>31-53</b>
Export an HDL Workflow Script .....	31-54
Specify Verbosity of Workflow Script .....	31-54
Enable or Disable Tasks in HDL Workflow Script .....	31-54
Run a Single Workflow Task .....	31-54
Import an HDL Workflow Script .....	31-55
Generic ASIC/FPGA Workflow Script Example .....	31-55
FPGA-in-the-Loop Script Example .....	31-56
FPGA Turnkey Workflow Script Example .....	31-58
IP Core Generation Workflow Script Example .....	31-60
Simulink Real-Time FPGA I/O Workflow Example .....	31-62
<b>Getting Started with the HDL Workflow Command-Line Interface</b> .....	<b>31-65</b>
<b>Getting Started with FPGA Turnkey Workflow</b> .....	<b>31-78</b>

## Simscape to HDL Workflow

# 32

<b>Get Started with Simscape Hardware-in-the-Loop Workflow</b> .....	<b>32-2</b>
Simscape Example Models for HDL Code Generation .....	32-2
Guidelines for Modeling Simscape for HDL Compatibility .....	32-2
Restrictions for HDL Code Generation from Simscape Models .....	32-3
<b>Modeling Guidelines for Simscape Subsystem Replacement</b> .....	<b>32-5</b>
Enclose Simscape Blocks Inside a Subsystem .....	32-5
Multiple Simscape Network Considerations .....	32-6
Avoid Using Certain Blocks in Simscape Utilities Library .....	32-7
<b>Generate HDL Code for Simscape Models</b> .....	<b>32-9</b>
<b>Generate Optimized HDL Implementation Model from Simscape</b> .....	<b>32-17</b>
<b>Generate Simulink Real-Time Interface Subsystem for Simscape Two-Level Converter Model</b> .....	<b>32-25</b>
<b>Deploy Simscape Buck Converter Model to Speedgoat IO Module Using HDL Workflow Script</b> .....	<b>32-33</b>
<b>Partition Simscape Models Containing a Large Network into Multiple Smaller Networks</b> .....	<b>32-47</b>

<b>Generate HDL Code for Simscape Models with Multiple Networks</b> .....	<b>32-54</b>
<b>Troubleshoot Conversion of Simscape DC Motor Control to HDL-Compatible Simulink Model</b> .....	<b>32-63</b>
<b>Troubleshoot Conversion of Simscape Permanent Magnet Synchronous Motor to HDL-Compatible Simulink Model</b> .....	<b>32-70</b>
<b>Replacing Variable Resistors</b> .....	<b>32-86</b>
<b>Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules</b> .....	<b>32-90</b>
<b>Validate HDL Implementation Model to Simscape Algorithm</b> .....	<b>32-97</b>
Bridge Rectifier Model .....	32-97
Increase Validation Logic Tolerance .....	32-99
Increase Number of Solver Iterations .....	32-100
Use Larger Floating-Point Precision .....	32-101
<b>Improve Sampling Rate of HDL Implementation Model Generated from Simscape Algorithm</b> .....	<b>32-104</b>
Sampling Frequency .....	32-104
Boost Converter Model .....	32-104
Reducing Number of Solver Iterations .....	32-106
Using Oversampling Factor and Latency Strategy .....	32-106

## 33

### Simscape HDL Workflow Advisor Tasks

<b>Simscape HDL Workflow Advisor Tasks</b> .....	<b>33-2</b>
Simscape HDL Workflow Advisor folder .....	33-2
Code generation compatibility folder .....	33-2
Check solver configuration task .....	33-2
Check switched linear task .....	33-3
State-space conversion folder .....	33-3
Extract Equations .....	33-4
Discretize Equations .....	33-4
Implementation model generation folder .....	33-4
Generate implementation model task .....	33-5
<b>Simscape HDL Workflow Advisor Tips and Guidelines</b> .....	<b>33-6</b>
Estimating Resource Consumption Using Algebraic and Differential Variables .....	33-6
Setting Simulation Stop Time for Extracting Equations .....	33-7
Changing Sample Time for Discretizing Equations .....	33-8
Using Number of Solver Iterations .....	33-9
Floating-Point Precision and Numerical Accuracy .....	33-10

<b>Create Protected Models to Conceal Contents and Generate HDL Code</b> . . . . .	<b>34-2</b>
How Model Protection Works . . . . .	34-2
How to Create a Protected Model . . . . .	34-2
General Protected Model Requirements and Limitations . . . . .	34-3
Protected Model Restrictions for HDL Code Generation . . . . .	34-3
Prepare the Parent Model . . . . .	34-4
Protect the Referenced Model . . . . .	34-5
Protected Model Report . . . . .	34-6
Generate HDL Code for Models Referencing Protected Model . . . . .	34-8
<b>Test Protected Models</b> . . . . .	<b>34-9</b>
<b>Package and Share Protected Models</b> . . . . .	<b>34-11</b>
Harness Model . . . . .	34-11
MAT-File with Base Workspace Definitions . . . . .	34-11
Simulink Data Dictionary . . . . .	34-12
Protected Model File Contents . . . . .	34-12
<b>Obfuscate Generated HDL Code from Simulink Models</b> . . . . .	<b>34-14</b>
How to Generate Obfuscated HDL Code . . . . .	34-14
Generated HDL Code with Obfuscation . . . . .	34-14
Code Obfuscation Report . . . . .	34-15
HDL Model Parameters Incompatible with Code Obfuscation . . . . .	34-15
Code Obfuscation Considerations and Restrictions . . . . .	34-16

<b>Verify Generated Code Using HDL Test Bench from Configuration Parameters</b> . . . . .	<b>35-2</b>
FIR Filter Model . . . . .	35-2
Create a Folder and Copy Relevant Files . . . . .	35-4
What is a HDL Test Bench? . . . . .	35-5
How to Verify the Generated Code . . . . .	35-5
Generate HDL Test Bench . . . . .	35-5
View HDL Test Bench Files . . . . .	35-6
Run Simulation and Verify Generated HDL Code . . . . .	35-7
<b>Verify Generated Code Using HDL Test Bench at Command Line</b> . . . . .	<b>35-9</b>
FIR Filter Model . . . . .	35-9
Create a Folder and Copy Relevant Files . . . . .	35-11
What is a HDL Test Bench? . . . . .	35-12
How to Verify the Generated Code . . . . .	35-12
Generate HDL Test Bench . . . . .	35-12
View HDL Test Bench Files . . . . .	35-13
Run Simulation and Verify Generated HDL Code . . . . .	35-13

<b>Test Bench Generation</b> .....	35-15
How Test Bench Generation Works .....	35-15
Test Bench Data Files .....	35-15
Test Bench Data Type Limitations .....	35-15
Use Constants Instead of File I/O .....	35-15
<b>Test Bench Block Restrictions</b> .....	35-17

## FPGA Board Customization

**36**

<b>FPGA Board Customization</b> .....	36-2
Feature Description .....	36-2
Custom Board Management .....	36-2
FPGA Board Requirements .....	36-2
<b>Create Custom FPGA Board Definition</b> .....	36-6
<b>Create Xilinx KC705 Evaluation Board Definition File</b> .....	36-7
Overview .....	36-7
What You Need to Know Before Starting .....	36-7
Start New FPGA Board Wizard .....	36-7
Provide Basic Board Information .....	36-8
Specify FPGA Interface Information .....	36-9
Enter FPGA Pin Numbers .....	36-10
Run Optional Validation Tests .....	36-12
Save Board Definition File .....	36-13
Use New FPGA Board .....	36-14
<b>FPGA Board Manager</b> .....	36-18
Introduction .....	36-18
Filter .....	36-19
Search .....	36-19
FIL Enabled/Turnkey Enabled .....	36-20
Create Custom Board .....	36-20
Add Board from File .....	36-20
Get More Boards .....	36-20
View/Edit .....	36-20
Remove .....	36-20
Clone .....	36-20
Validate .....	36-20
<b>New FPGA Board Wizard</b> .....	36-21
Basic Information .....	36-22
Interfaces .....	36-22
FIL I/O .....	36-25
Turnkey I/O .....	36-27
Validation .....	36-30
Finish .....	36-31
<b>FPGA Board Editor</b> .....	36-32
General Tab .....	36-32

**HDL Workflow Advisor Tasks**

<b>HDL Workflow Advisor Tasks</b> .....	<b>37-2</b>
HDL Workflow Advisor Tasks Overview .....	37-3
Set Target Overview .....	37-4
Set Target Device and Synthesis Tool .....	37-4
Set Target Reference Design .....	37-5
Set Target Interface .....	37-5
Set Target Frequency .....	37-6
Set Target Interface .....	37-6
Set Target Interface .....	37-7
Prepare Model For HDL Code Generation Overview .....	37-8
Check Global Settings .....	37-9
Check Algebraic Loops .....	37-9
Check Block Compatibility .....	37-9
Check Sample Times .....	37-10
Check FPGA-In-The-Loop Compatibility .....	37-10
HDL Code Generation Overview .....	37-10
Set Code Generation Options Overview .....	37-11
Set Basic Options .....	37-11
Set Report Options .....	37-11
Set Advanced Options .....	37-12
Set Optimization Options .....	37-12
Set Testbench Options .....	37-12
Generate RTL Code .....	37-12
Generate RTL Code and Testbench .....	37-12
Verify with HDL Cosimulation .....	37-13
Generate RTL Code and IP Core .....	37-13
FPGA Synthesis and Analysis Overview .....	37-14
Create Project .....	37-15
Perform Synthesis and P/R Overview .....	37-15
Perform Logic Synthesis .....	37-16
Perform Mapping .....	37-16
Perform Place and Route .....	37-16
Run Synthesis .....	37-17
Run Implementation .....	37-17
Annotate Model with Synthesis Result .....	37-17
Download to Target Overview .....	37-18
Generate Programming File .....	37-19
Program Target Device .....	37-19
Generate Simulink Real-Time Interface .....	37-19
Save and Restore HDL Workflow Advisor State .....	37-19
FPGA-In-The-Loop Implementation .....	37-19
Set FPGA-In-The-Loop Options .....	37-19
Build FPGA-In-The-Loop .....	37-20
Check USRP Compatibility .....	37-20
Generate FPGA Implementation .....	37-20
Check SDR Compatibility .....	37-20
SDR FPGA Implementation .....	37-21
Set SDR Options .....	37-21

Build SDR .....	37-22
Embedded System Integration .....	37-22
Create Project .....	37-22
Generate Software Interface .....	37-23
Build FPGA Bitstream .....	37-23
Program Target Device .....	37-23

## HDL Code Advisor

# 38

<b>HDL Coder Checks in Model Advisor / HDL Code Advisor Overview</b> .....	38-3
<b>Model configuration checks overview</b> .....	38-4
<b>Check for model parameters suited for HDL code generation</b> .....	38-5
Description .....	38-5
Results and Recommended Actions .....	38-6
See Also .....	38-6
<b>Check for global reset setting for Xilinx and Altera devices</b> .....	38-7
Description .....	38-7
Results and Recommended Actions .....	38-7
See Also .....	38-7
<b>Check inline configurations setting</b> .....	38-8
Description .....	38-8
Results and Recommended Actions .....	38-8
<b>Check algebraic loops</b> .....	38-9
Description .....	38-9
Results and Recommended Actions .....	38-9
See Also .....	38-9
<b>Check for visualization settings</b> .....	38-10
Description .....	38-10
Results and Recommended Actions .....	38-10
See Also .....	38-10
<b>Check delay balancing setting</b> .....	38-11
Description .....	38-11
Results and Recommended Actions .....	38-11
See Also .....	38-11
<b>Check for ports and subsystems overview</b> .....	38-12
<b>Check for invalid top level subsystem</b> .....	38-13
Description .....	38-13
Results and Recommended Actions .....	38-13
<b>Check initial conditions of enabled and triggered subsystems</b> .....	38-14
Description .....	38-14

Results and Recommended Actions .....	<b>38-14</b>
See Also .....	<b>38-14</b>
<b>Check for blocks and block settings overview .....</b>	<b>38-15</b>
<b>Check for infinite and continuous sample time sources .....</b>	<b>38-16</b>
Description .....	<b>38-16</b>
Results and Recommended Actions .....	<b>38-16</b>
See Also .....	<b>38-16</b>
<b>Check for unsupported blocks .....</b>	<b>38-17</b>
Description .....	<b>38-17</b>
Results and Recommended Actions .....	<b>38-17</b>
<b>Check for large matrix operations .....</b>	<b>38-18</b>
Description .....	<b>38-18</b>
Results and Recommended Actions .....	<b>38-18</b>
See Also .....	<b>38-18</b>
<b>Check for MATLAB Function block settings .....</b>	<b>38-19</b>
Description .....	<b>38-19</b>
Results and Recommended Actions .....	<b>38-19</b>
See Also .....	<b>38-19</b>
<b>Check for Stateflow chart settings .....</b>	<b>38-20</b>
Description .....	<b>38-20</b>
Results and Recommended Actions .....	<b>38-20</b>
See Also .....	<b>38-20</b>
<b>Check for obsolete Unit Delay Enabled/Resettable Blocks .....</b>	<b>38-21</b>
Description .....	<b>38-21</b>
Results and Recommended Actions .....	<b>38-21</b>
<b>Check for blocks that have nonzero output latency .....</b>	<b>38-22</b>
Description .....	<b>38-22</b>
Results and Recommended Actions .....	<b>38-22</b>
See Also .....	<b>38-22</b>
<b>Check for unsupported storage class for signal objects .....</b>	<b>38-23</b>
Description .....	<b>38-23</b>
Results and Recommended Actions .....	<b>38-23</b>
<b>Native Floating Point Checks Overview .....</b>	<b>38-24</b>
<b>Check for single datatypes in the model .....</b>	<b>38-25</b>
Description .....	<b>38-25</b>
Results and Recommended Actions .....	<b>38-25</b>
See Also .....	<b>38-25</b>
<b>Check for double datatypes in the model with Native Floating Point .....</b>	<b>38-26</b>
Description .....	<b>38-26</b>
Results and Recommended Actions .....	<b>38-26</b>
See Also .....	<b>38-26</b>

<b>Check for Data Type Conversion blocks with incompatible settings</b>	38-27
Description	38-27
Results and Recommended Actions	38-27
See Also	38-27
<b>Check for HDL Reciprocal block usage</b>	38-28
Description	38-28
Results and Recommended Actions	38-28
See Also	38-28
<b>Check for Relational Operator block usage</b>	38-29
Description	38-29
Results and Recommended Actions	38-29
See Also	38-29
<b>Check for unsupported blocks with Native Floating Point</b>	38-30
Description	38-30
Results and Recommended Actions	38-30
See Also	38-30
<b>Check blocks with nonzero ulp error</b>	38-31
Description	38-31
Results and Recommended Actions	38-31
See Also	38-31
<b>Industry standard checks overview</b>	38-32
<b>Check VHDL file extension</b>	38-33
Description	38-33
Results and Recommended Actions	38-33
See Also	38-33
<b>Check naming conventions</b>	38-34
Description	38-34
Results and Recommended Actions	38-34
See Also	38-34
<b>Check top-level subsystem/port names</b>	38-35
Description	38-35
Results and Recommended Actions	38-35
See Also	38-35
<b>Check module/entity names</b>	38-36
Description	38-36
Results and Recommended Actions	38-36
See Also	38-36
<b>Check signal and port names</b>	38-37
Description	38-37
Results and Recommended Actions	38-37
See Also	38-37
<b>Check package file names</b>	38-38
Description	38-38

Results and Recommended Actions .....	38-38
See Also .....	38-38
<b>Check generics</b> .....	38-39
Description .....	38-39
Results and Recommended Actions .....	38-39
See Also .....	38-39
<b>Check clock, reset, and enable signals</b> .....	38-40
Description .....	38-40
Results and Recommended Actions .....	38-40
See Also .....	38-40
<b>Check architecture name</b> .....	38-41
Description .....	38-41
Results and Recommended Actions .....	38-41
See Also .....	38-41
<b>Check entity and architecture</b> .....	38-42
Description .....	38-42
Results and Recommended Actions .....	38-42
See Also .....	38-42
<b>Check clock settings</b> .....	38-43
Description .....	38-43
Results and Recommended Actions .....	38-43
See Also .....	38-43

# 39

## Using the HDL Code Advisor

<b>Check HDL Compatibility of Simulink Model Using HDL Code Advisor</b> .....	39-2
Open the HDL Code Advisor .....	39-2
Run Checks In the HDL Code Advisor .....	39-3
Fix HDL Code Advisor Warnings or Failures .....	39-3
View and Save HDL Code Advisor Reports .....	39-4
<b>Run Model Advisor Checks for HDL Coder</b> .....	39-6
Open the Model Advisor Checks .....	39-6
Run Checks in the Model Advisor .....	39-6
Run Checks In Background .....	39-7
Display Check Results in the Model Advisor Report .....	39-7
Fix Warnings or Failures .....	39-8
Save and Restore Model Advisor State .....	39-9
<b>HDL Code Advisor Checks</b> .....	39-11
Model configuration checks .....	39-12
Checks for ports and subsystems .....	39-12
Checks for blocks and block settings .....	39-12
Native Floating Point checks .....	39-13
industry standard checks .....	39-14

### Hardware-Software Co-Design Basics

40

<b>Hardware-Software Co-Design Workflow for SoC Platforms . . . . .</b>	<b>40-2</b>
<b>Speedgoat FPGA Support with HDL Workflow Advisor . . . . .</b>	<b>40-8</b>
Speedgoat Simulink-Programmable I/O Module Support . . . . .	40-8
Prepare for FPGA Workflow . . . . .	40-8
<b>Custom IP Core Generation . . . . .</b>	<b>40-10</b>
Custom IP Core Architectures . . . . .	40-10
Target Platform Interfaces . . . . .	40-10
Processor/FPGA Synchronization . . . . .	40-11
Custom IP Core Generated Files . . . . .	40-11
Restrictions . . . . .	40-11
<b>Custom IP Core Report . . . . .</b>	<b>40-13</b>
Summary . . . . .	40-13
Target Interface Configuration . . . . .	40-13
Register Address Mapping . . . . .	40-14
IP Core User Guide . . . . .	40-15
IP Core File List . . . . .	40-18
<b>Generate Board-Independent HDL IP Core from Simulink Model . . . . .</b>	<b>40-19</b>
Generate Board-Independent IP Core . . . . .	40-19
IP Core without AXI4 Slave Interfaces . . . . .	40-21
Requirements and Limitations for IP Core Generation . . . . .	40-22
<b>Processor and FPGA Synchronization . . . . .</b>	<b>40-23</b>
Free Running Mode . . . . .	40-23
Coprocessing - Blocking Mode . . . . .	40-23
Coprocessing - Nonblocking With Delay Mode . . . . .	40-24
<b>Synchronization of Global Reset Signal to IP Core Clock Domain . . . . .</b>	<b>40-25</b>
<b>IP Caching for Faster Reference Design Synthesis . . . . .</b>	<b>40-29</b>
Requirements for Using IP Caching . . . . .	40-29
What Is an IP Cache? . . . . .	40-29
How IP Caching Works . . . . .	40-30
Enable IP Caching . . . . .	40-30
IP Caching in HDL Coder Reference Designs . . . . .	40-31
IP Caching in Custom Reference Designs . . . . .	40-32
<b>Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows . . . . .</b>	<b>40-34</b>
Step 1: Identify the Timing Failure . . . . .	40-34
Step 2: Find the Critical Path . . . . .	40-37
Step 3: Resolve Timing Failures . . . . .	40-41

<b>Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface</b> .....	<b>40-45</b>
Vivado-Based Reference Designs .....	40-45
Qsys-Based Reference Designs .....	40-47
<b>Program Target FPGA Boards or SoC Devices</b> .....	<b>40-49</b>
How to Program Target Device .....	40-49
Programming Methods .....	40-50
<b>Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core</b> .....	<b>40-53</b>
Prerequisites .....	40-53
Generate Software Interface .....	40-53
Software Interface Model .....	40-54
Software Interface Script .....	40-57
<b>Create Software Interface Script to Control and Rapidly Prototype HDL IP Core</b> .....	<b>40-60</b>
Software Interface Script .....	40-60
Customizing Software Interface Script .....	40-60
Develop Software Interface Script .....	40-60
<b>Getting Started with Targeting Xilinx Zynq Platform</b> .....	<b>40-65</b>
<b>Getting Started with Targeting Zynq UltraScale+ MPSoC Platform</b> .....	<b>40-84</b>
<b>Getting Started with Targeting Intel SoC Devices</b> .....	<b>40-104</b>
<b>Getting Started with Targeting Intel Quartus Pro based Devices</b> .....	<b>40-122</b>
<b>Save Target Hardware Settings in Model</b> .....	<b>40-137</b>
<b>Using IP Core Generation Workflow from MATLAB: LED Blinking</b> .....	<b>40-143</b>
<b>IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705</b> .....	<b>40-153</b>
<b>IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit</b> .....	<b>40-162</b>
<b>IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705</b> .....	<b>40-172</b>

<b>Model Design for AXI4 Slave Interface Generation</b> .....	<b>41-3</b>
Considerations .....	41-3
Map Scalar Ports to AXI4 Slave Interface .....	41-3

Map Vector Ports to AXI4 Slave Interface .....	41-4
Initial Value of AXI4 Slave Registers .....	41-5
Read Back Value of AXI4 Slave Interfaces .....	41-6
Optimize AXI4 Slave Read Back Logic .....	41-9
<b>Model Design for AXI4-Stream Interface Generation .....</b>	<b>41-10</b>
Simplified Streaming Protocol .....	41-10
Map Scalar Ports To AXI4-Stream Interface .....	41-11
Map Vector Ports To AXI4-Stream Interface .....	41-14
Model Designs with Multiple Streaming Channels .....	41-15
Model Designs with Multiple Sample Rates .....	41-15
Restrictions .....	41-15
<b>Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces .....</b>	<b>41-17</b>
Why Use Multiple AXI4 Interfaces .....	41-17
Specify Multiple AXI4 Interfaces in Generic IP Core Generation Workflow .....	41-17
Specify Multiple AXI4 Interfaces in Custom Reference Designs .....	41-18
Ready Signal Mapping for Multiple Streaming Interfaces .....	41-20
Restrictions .....	41-20
<b>Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard .....</b>	<b>41-22</b>
<b>Multirate IP Core Generation .....</b>	<b>41-35</b>
<b>Board and Reference Design Registration System .....</b>	<b>41-39</b>
Board, IP Core, and Reference Design Definitions .....	41-39
Board Registration Files .....	41-39
Reference Design Registration Files .....	41-40
Predefined Board and Reference Design Examples .....	41-41
<b>Register a Custom Board .....</b>	<b>41-42</b>
Define a Board .....	41-42
Create a Board Plugin .....	41-43
Define a Board Registration Function .....	41-43
<b>Register a Custom Reference Design .....</b>	<b>41-45</b>
Define a Reference Design .....	41-45
Create a Reference Design Plugin .....	41-46
Define a Reference Design Registration Function .....	41-46
<b>Define Custom Parameters and Callback Functions for Custom Reference Design .....</b>	<b>41-48</b>
Define Custom Parameters and Register Callback Function Handle .....	41-48
Define Custom Callback Functions .....	41-52
<b>Customize Reference Design Dynamically Based on Reference Design Parameters .....</b>	<b>41-54</b>
Why Customize the Reference Design .....	41-54
How Reference Design Customization Works .....	41-54
Customizable Reference Design Parameters .....	41-55

Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs .....	<b>41-56</b>
<b>Define and Add IP Repository to Custom Reference Design .....</b>	<b>41-59</b>
Create an IP Repository Folder Structure .....	41-59
Define IP List Function .....	41-60
Add IP List Function to Reference Design Project .....	41-61
<b>FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules .....</b>	<b>41-63</b>
<b>Model Design for AXI4-Stream Video Interface Generation .....</b>	<b>41-69</b>
Streaming Pixel Protocol .....	41-69
Protocol Signals and Timing Diagrams .....	41-69
Model Data and Control Bus Signals .....	41-71
Map DUT Ports to Multiple Channels .....	41-75
Model Designs with Multiple Sample Rates .....	41-75
Video Porch Insertion Logic .....	41-75
Default Video System Reference Design .....	41-76
Restrictions .....	41-77
<b>Model Design for AXI4 Master Interface Generation .....</b>	<b>41-78</b>
Simplified AXI4 Master Protocol - Write Channel .....	41-78
Simplified AXI4 Master Protocol - Read Channel .....	41-80
Base Address Register Calculation .....	41-81
Modeling for AXI4 Master Interfaces .....	41-81
Map Vector Ports to AXI4 Master Interfaces .....	41-84
Model Designs with Multiple Sample Rates .....	41-86
Reference Designs for IP Core Integration .....	41-87
Restrictions .....	41-88
<b>IP Core Generation Workflow for Standalone FPGA Devices .....</b>	<b>41-89</b>
Targeting FPGA Reference Designs with AXI4 Interface .....	41-90
Targeting FPGA Reference Designs Without AXI4 Interface .....	41-92
Board Support .....	41-92
Restrictions .....	41-92
<b>IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules .....</b>	<b>41-93</b>
Supported I/O Modules .....	41-93
IP Core Generation Workflow .....	41-93
Restrictions .....	41-95
<b>IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip .....</b>	<b>41-96</b>
<b>Running an Audio Filter on Live Audio Input using Intel Board .....</b>	<b>41-116</b>
<b>Running an Audio Filter on Live Audio Input Using a Zynq Board .....</b>	<b>41-126</b>
<b>Getting Started with AXI4-Stream Interface in Zynq Workflow .....</b>	<b>41-137</b>
<b>Getting Started with AXI4-Stream Video Interface in Zynq Workflow .....</b>	<b>41-152</b>

<b>Performing Large Matrix Operation on FPGA using External Memory</b>	<b>41-162</b>
<b>Authoring a Reference Design for Audio System on a Zynq Board</b>	<b>41-170</b>
<b>Authoring a Reference Design for Audio System on a ZYBO Board</b>	<b>41-180</b>
<b>Authoring a Reference Design for Audio System on Intel board</b>	<b>41-186</b>
<b>Define Custom Board and Reference Design for Zynq Workflow</b>	<b>41-196</b>
<b>Define Custom Board and Reference Design for Intel SoC Workflow</b>	<b>41-215</b>
<b>Dynamically Create Master Only or Slave Only or Both Slave and Master Reference Designs</b>	<b>41-229</b>
<b>Using JTAG MATLAB as AXI Master to control HDL Coder generated IP Core</b>	<b>41-242</b>
<b>Debug a Zynq Design Using HDL Coder and Embedded Coder</b>	<b>41-248</b>
<b>Debug IP Core Using FPGA Data Capture</b>	<b>41-253</b>

# HDL Code Generation from MATLAB



# MATLAB Algorithm Design

---

- “Functions Supported for HDL Code Generation” on page 1-2
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Persistent Variables and Persistent Array Variables” on page 1-9
- “Complex Data Type Support” on page 1-11
- “HDL Code Generation for System Objects” on page 1-14
- “HDL Code Generation from System Objects” on page 1-16
- “HDL Code Generation for Streaming Matrix Inverse System Object” on page 1-20
- “HDL Code Generation for Streaming Matrix Multiply System Object” on page 1-29
- “HDL Code Generation from `hdl.RAM` System Object” on page 1-37
- “HDL Code Generation from A Non-Restoring Square Root System Object” on page 1-41
- “HDL Code Generation from Viterbi Decoder System Object” on page 1-46
- “Predefined System Objects Supported for HDL Code Generation” on page 1-50
- “Load constants from a MAT-File” on page 1-52
- “Generate Code for User-Defined System Objects” on page 1-53
- “Map Matrices to ROM” on page 1-55
- “Model State with Persistent Variables and System Objects” on page 1-56
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL Code” on page 1-62
- “For-Loop Best Practices for HDL Code Generation” on page 1-64
- “MATLAB Test Bench Requirements and Best Practices for HDL Code Generation” on page 1-66

# Functions Supported for HDL Code Generation

## In this section...

"Supported MATLAB and Fixed Point Runtime Library Functions" on page 1-2

"Fixed-Point Function Limitations" on page 1-2

You can generate efficient HDL code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code.

## Supported MATLAB and Fixed Point Runtime Library Functions

The supported functions for HDL code generation are listed in the following tables. In these tables, a

 icon before the name of a function indicates that there are specific usage notes and limitations related to HDL code generation for that function. To view these usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **HDL Code Generation** section.

The table shows HDL code generation support for both MATLAB and fixed-point run-time library functions from the Fixed-Point Designer™ functions.

HDL code generation support for the functions is summarized in the following tables.

- Functions Supported for HDL Code Generation (Category List)
- Functions Supported for HDL Code Generation (Alphabetical List)

## Fixed-Point Function Limitations

In addition to function-specific limitations listed in the table, the following general limitations apply to the use of Fixed-Point Designer functions in generated HDL code:

- `fipref` and `quantizer` objects are not supported.
- Slope and bias scaling are not supported.
- Dot notation is only supported for getting the values of `fimath` and `numerictype` properties. Dot notation is not supported for `fi` objects, and it is not supported for setting properties.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numerictype` of a given variable after that variable has been created.
- The `boolean` and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- General limitations of C/C++ code generated from MATLAB apply. See "MATLAB Language Features That Code Generation Does Not Support".

## See Also

`codegen` | `coder.HdlConfig`

## More About

- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59
- “Functions for Programming and Data Types”

# Supported MATLAB Data Types, Operators, and Control Flow Statements

## In this section...

["Supported Data Types" on page 1-4](#)

["Supported Operators" on page 1-5](#)

["Control Flow Statements" on page 1-7](#)

When you generate HDL code from your MATLAB algorithm, use the data types, operators, and control flow statements that HDL Coder supports.

## Supported Data Types

HDL Coder does not support cell arrays and `Inf` data types. This table shows the supported subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	<ul style="list-style-type: none"> <li><code>uint8</code>, <code>uint16</code>, <code>uint32</code>, <code>uint64</code></li> <li><code>int8</code>, <code>int16</code>, <code>int32</code>, <code>int64</code></li> </ul>	In Simulink®, MATLAB Function block ports must use numeric types <code>sfix64</code> or <code>ufix64</code> for 64-bit data.
Real	<ul style="list-style-type: none"> <li><code>double</code></li> <li><code>single</code></li> </ul>	<p>HDL code generated with <code>double</code> or <code>single</code> data types in your MATLAB code can be used for simulation, but is not synthesizable. You can generate synthesizable code when you use these data types in your Simulink model. For more information, see:</p> <ul style="list-style-type: none"> <li><a href="#">"Simulink Blocks Supported with Native Floating-Point" on page 10-120</a></li> <li><a href="#">"Generate Target-Independent HDL Code with Native Floating-Point" on page 10-103</a></li> <li><a href="#">"Signal and Data Type Support" on page 10-2</a></li> </ul>
Character	<code>char</code>	-
Logical	<code>logical</code>	-
Fixed point	<ul style="list-style-type: none"> <li>Scaled (binary point only) fixed-point numbers</li> <li>Custom integers (zero binary point)</li> </ul>	<p>Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported.</p> <p>Maximum word size for fixed-point numbers is 128 bits.</p>
Vectors	<ul style="list-style-type: none"> <li><code>unordered {N}</code></li> <li><code>row {1, N}</code></li> <li><code>column {N, 1}</code></li> </ul>	<p>The maximum number of vector elements allowed is <math>2^{32}</math>.</p> <p>Before a variable is subscripted, it must be fully defined.</p>

Types	Supported Data Types	Restrictions
Matrices	{N, M}	Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.  Do not use matrices in the testbench.
Structures	struct	Arrays of structures are not supported.  For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.
Enumerations	enumeration	Enumeration values must be monotonically increasing.  If your target language is Verilog®, all enumeration member names must be unique within the design.  Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods: <ul style="list-style-type: none"> <li>• IP Core Generation workflow</li> <li>• FPGA Turnkey workflow</li> <li>• FPGA-in-the-Loop</li> <li>• HDL Cosimulation</li> </ul>

Global variables are not supported for HDL code generation.

## Supported Operators

---

**Note** HDL code generated for large vector and matrix inputs to arithmetic operations can result in inefficient code. The code for these operators is not automatically pipelined.

---

## Arithmetic Operators

Operation	Operator Syntax	Equivalent Function	Restrictions
Binary addition	A+B	plus(A,B)	Neither A nor B can be data type logical.
Matrix multiplication	A*B	mtimes(A,B)	HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.
Arraywise multiplication	A.*B	times(A,B)	Neither A nor B can be data type logical.
Matrix power	A^B	mpower(A,B)	A and B must be scalar, and B must be an integer.  HDL code generated for matrix arithmetic operations is not pipelined, and can result in inefficient code.
Arraywise power	A.^B	power(A,B)	A and B must be scalar, and B must be an integer.
Complex transpose	A'	ctranspose(A)	-
Matrix transpose	A.'	transpose(A)	-
Matrix concat	[A B]	None	-
Matrix index	A(r c)	None	Before you use a variable, you must fully define it.

## Logical Operators

Operation	Operator Syntax	M Function Equivalent	Notes
Logical And	A&B	and(A,B)	-
Logical Or	A B	or(A,B)	-
Logical Xor	A xor B	xor(A,B)	-
Logical And (short circuiting)	A&&B	N/A	Use short circuiting logical operators within conditionals.
Logical Or (short circuiting)	A  B	N/A	Use short circuiting logical operators within conditionals.
Element complement	~A	not(A)	-

## Relational Operators

Relation	Operator Syntax	Equivalent Function
Less than	A<B	lt(A,B)
Less than or equal to	A<=B	le(A,B)
Greater than or equal to	A>=B	ge(A,B)
Greater than	A>B	gt(A,B)
Equal	A==B	eq(A,B)
Not equal	A~=B	ne(A,B)

## Control Flow Statements

HDL Coder supports the following control flow statements and constructs with restrictions.

Control Flow Statement	Restrictions
for	<p>Do not use <code>for</code> loops without static bounds.</p> <p>Do not use the <code>&amp;</code> and <code> </code> operators within conditions of a <code>for</code> statement. Instead, use the <code>&amp;&amp;</code> and <code>  </code> operators.</p> <p>HDL Coder does not support nonscalar expressions in the conditions of <code>for</code> statements. Instead, use the <code>all</code> or <code>any</code> functions to collapse logical vectors into scalars.</p>
if	<p>Do not use the <code>&amp;</code> and <code> </code> operators within conditions of an <code>if</code> statement. Instead, use the <code>&amp;&amp;</code> and <code>  </code> operators.</p> <p>HDL Coder does not support nonscalar expressions in the conditions of <code>if</code> statements. Instead, use the <code>all</code> or <code>any</code> functions to collapse logical vectors into scalars.</p>
switch	<p>The conditional expression in a <code>switch</code> or <code>case</code> statement must use only:</p> <ul style="list-style-type: none"> <li>• <code>uint8</code>, <code>uint16</code>, <code>uint32</code>, <code>int8</code>, <code>int16</code>, or <code>int32</code> data types</li> <li>• Scalar data</li> </ul> <p>If multiple <code>case</code> statements make assignments to the same variable, the numeric type and <code>fimath</code> specification for that variable must be the same in every <code>case</code> statement.</p>

The following control flow statements are not supported:

- `while`
- `break`
- `continue`
- `return`
- `parfor`

Avoid using the following vector functions, as they may generate loops containing `break` statements:

- `isequal`
- `bitrevorder`

## See Also

`codegen` | `coder.HdlConfig`

## More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59
- “Functions for Programming and Data Types”

# Persistent Variables and Persistent Array Variables

## Persistent Variables

Persistent variables enable you to model registers. If you need to preserve state between invocations of your MATLAB algorithm, use persistent variables.

Before you use a persistent variable, you must initialize it with a statement specifying its size and type. You can initialize a persistent variable with either a constant value or a variable, as in the following examples:

```
% Initialize with a constant
persistent p;
if isempty(p)
    p = fi(0,0,8,0);
end

% Initialize with a variable
initval = fi(0,0,8,0);

persistent p;
if isempty(p)
    p = initval;
end
```

Use a logical expression that evaluates to a constant to test whether a persistent variable has been initialized, as in the preceding examples. Using a logical expression that evaluates to a constant ensures that the generated HDL code for the test is executed only once, as part of the reset process.

You can initialize multiple variables within a single logical expression, as in the following example:

```
% Initialize with variables
initval1 = fi(0,0,8,0);
initval2 = fi(0,0,7,0);

persistent p;
if isempty(p)
    x = initval1;
    y = initval2;
end
```

---

**Note** If persistent variables are not initialized as described above, extra sentinel variables can appear in the generated code. These sentinel variables can translate to inefficient hardware.

---

## Persistent Array Variables

Persistent array variables enable you to model RAM.

By default, the HDL Coder software optimizes the area of your design by mapping persistent array variables to RAM. If persistent array variables are not mapped to RAM, they map to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

To learn how persistent array variables map to RAM, see “Map Persistent Arrays and `dsp.Delay` to RAM” on page 8-8.

## See Also

`codegen` | `coder.HdlConfig`

## More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59
- “Functions for Programming and Data Types”

# Complex Data Type Support

## In this section...

- “Declaring Complex Signals” on page 1-11
- “Conversion Between Complex and Real Signals” on page 1-12
- “Support for Vectors of Complex Numbers” on page 1-12

## Declaring Complex Signals

The following MATLAB code declares several local complex variables. `x` and `y` are declared by complex constant assignment; `z` is created using the `complex()` function.

```
function [x,y,z] = fcn

% create 8 bit complex constants
x = uint8(1 + 2i);
y = uint8(3 + 4j);
z = uint8(complex(5, 6));
```

The following code example shows VHDL® code generated from the previous MATLAB code.

```
ENTITY complex_decl IS
  PORT (
    clk : IN std_logic;
    clk_enable : IN std_logic;
    reset : IN std_logic;
    x_re : OUT std_logic_vector(7 DOWNTO 0);
    x_im : OUT std_logic_vector(7 DOWNTO 0);
    y_re : OUT std_logic_vector(7 DOWNTO 0);
    y_im : OUT std_logic_vector(7 DOWNTO 0);
    z_re : OUT std_logic_vector(7 DOWNTO 0);
    z_im : OUT std_logic_vector(7 DOWNTO 0));
END complex_decl;

ARCHITECTURE fsm_SFHDl OF complex_decl IS

BEGIN
  x_re <= std_logic_vector(to_unsigned(1, 8));
  x_im <= std_logic_vector(to_unsigned(2, 8));
  y_re <= std_logic_vector(to_unsigned(3, 8));
  y_im <= std_logic_vector(to_unsigned(4, 8));
  z_re <= std_logic_vector(to_unsigned(5, 8));
  z_im <= std_logic_vector(to_unsigned(6, 8));
END fsm_SFHDl;
```

As shown in the example, complex inputs, outputs and local variables declared in MATLAB code expand into real and imaginary signals. The naming conventions for these derived signals are:

- Real components have the same name as the original complex signal, suffixed with the default string '`_re`' (for example, `x_re`). To specify a different suffix, set the **Complex real part postfix** option (or the corresponding `ComplexRealPostfix` CLI property).
- Imaginary components have the same name as the original complex signal, suffixed with the string '`_im`' (for example, `x_im`). To specify a different suffix, set the **Complex imaginary part postfix** option (or the corresponding `ComplexImagPostfix` CLI property).

A complex variable declared in MATLAB code remains complex during the entire length of the program.

## Conversion Between Complex and Real Signals

The MATLAB code provides access to the fields of a complex signal via the `real()` and `imag()` functions, as shown in the following code.

```
function [Re_part, Im_part]=fcn(c)
% Output real and imaginary parts of complex input signal

Re_part = real(c);
Im_part = imag(c);
```

HDL Coder supports these constructs, accessing the corresponding real and imaginary signal components in generated HDL code. In the following Verilog code example, the MATLAB complex signal variable `c` is flattened into the signals `c_re` and `c_im`. Each of these signals is assigned to the output variables `Re_part` and `Im_part`, respectively.

```
module Complex_To_Real_Imag (clk, clk_enable, reset, c_re, c_im, Re_part, Im_part );

    input clk;
    input clk_enable;
    input reset;
    input [3:0] c_re;
    input [3:0] c_im;
    output [3:0] Re_part;
    output [3:0] Im_part;

    // Output real and imaginary parts of complex input signal
    assign Re_part = c_re;
    assign Im_part = c_im;
```

## Support for Vectors of Complex Numbers

You can generate HDL code for vectors of complex numbers. Like scalar complex numbers, vectors of complex numbers are flattened down to vectors of real and imaginary parts in generated HDL code.

For example in the following script `t` is a complex vector variable of base type `ufix4` and size `[1, 2]`.

```
function y = fcn(u1, u2)
t = [u1 u2];
y = t+1;
```

In the generated HDL code the variable `t` is broken down into real and imaginary parts with the same two-element array.

```
VARIABLE t_re : vector_of_unsigned4(0 TO 3);
VARIABLE t_im : vector_of_unsigned4(0 TO 3);
```

The real and imaginary parts of the complex number have the same vector of type `ufix4`, as shown in the following code.

```
TYPE vector_of_unsigned4 IS ARRAY (NATURAL RANGE <>) OF unsigned(3 DOWNTO 0);
```

Complex vector-based operations (+, -, \*, etc.) are similarly broken down to vectors of real and imaginary parts. Operations are performed independently on the elements of such vectors, following MATLAB semantics for vectors of complex numbers.

In both VHDL and Verilog code generated from MATLAB code, complex vector ports are always flattened. If complex vector variables appear on inputs and outputs, real and imaginary vector components are further flattened to scalars.

In the following code, `u1` and `u2` are scalar complex numbers and `y` is a vector of complex numbers.

```
function y = fcn(u1, u2)
```

```
t = [u1 u2];
y = t+1;
```

This generates the following port declarations in a VHDL entity definition.

```
ENTITY _MATLAB_Function IS
  PORT (
    clk : IN std_logic;
    clk_enable : IN std_logic;
    reset : IN std_logic;
    u1_re : IN vector_of_std_logic_vector4(0 TO 1);
    u1_im : IN vector_of_std_logic_vector4(0 TO 1);
    u2_re : IN vector_of_std_logic_vector4(0 TO 1);
    u2_im : IN vector_of_std_logic_vector4(0 TO 1);
    y_re : OUT vector_of_std_logic_vector32(0 TO 3);
    y_im : OUT vector_of_std_logic_vector32(0 TO 3));
END _MATLAB_Function;
```

## See Also

[codegen](#) | [coder.HdlConfig](#)

## More About

- “Hexadecimal and Binary Values”
- “Functions Supported for HDL Code Generation” on page 1-2
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59

## HDL Code Generation for System Objects

### In this section...

- “Why Use System Objects?” on page 1-14
- “Predefined System Objects” on page 1-14
- “User-Defined System Objects” on page 1-14
- “Limitations of HDL Code Generation for System Objects” on page 1-14
- “System object Examples for HDL Code Generation” on page 1-15

HDL Coder supports both predefined and user-defined System objects for code generation.

### Why Use System Objects?

System objects provide a design advantage because:

- You can save time during design and testing by using existing System object components.
- You can design and qualify custom System objects for reuse in multiple designs.
- You can define your algorithm in a System object once, and reuse multiple instances of it in a single MATLAB design.

This idiom cannot be used with MATLAB functions that have state. For example, if the algorithm has state and requires the use of persistent variables, that function cannot be instantiated multiple times in a design. Instead, you would need to copy and rename the function for each instance.

- HDL code that you generate from System objects is modular and more readable.

### Predefined System Objects

Predefined System objects that are available with MATLAB, DSP System Toolbox™, and Communications Toolbox™ are supported for HDL code generation. For a list, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

### User-Defined System Objects

You can create user-defined System objects for HDL code generation. For an example, see “Generate Code for User-Defined System Objects” on page 1-53.

### Limitations of HDL Code Generation for System Objects

The following limitations apply to HDL code generation for all System objects:

- Your design can call the `step` method only once per System object.
- `step` must not be inside a nested conditional statement, such as a nested loop, `if` statement, or `switch` statement.
- `step` must not be inside a conditional statement that contains a matrix indexing operation.
- A System object must be declared persistent if it has state.

A System object has state when it has a tunable private or public property, or a property with the `DiscreteState` attribute.

- You can use the `dsp.Delay` System object only in feed-forward delay modeling.
- Enumerations are not supported.
- Global variables are not supported.

### **Supported Methods**

For predefined System Objects, `step` is the only method supported for HDL code generation.

For user-defined System Objects, either the `step` method, or the `output` and `update` methods, are supported for HDL code generation.

### **Additional Restrictions for Predefined System Objects**

Predefined System objects are not supported for HDL code generation from within a MATLAB System block.

### **Additional Restrictions for User-Defined System Objects**

In addition to the limitations for all System objects, the following restrictions apply to user-defined System objects for HDL code generation:

- In the `setupImpl` and `resetImpl` methods, if you assign values to properties or variables, the values must be constants.
- If your design uses the `output` and `update` methods, it can call each method only once per System object.
- Initial and reset values for properties must be compile-time constant.
- User-defined System objects must not be public properties.
- A `step` method with multiple outputs cannot be called within a conditional statement.

## **System object Examples for HDL Code Generation**

To learn how to use System objects for HDL code generation, view the MATLAB designs in the following examples:

- “HDL Code Generation from System Objects” on page 1-16
- “Model State with Persistent Variables and System Objects” on page 1-56
- “Generate Code for User-Defined System Objects” on page 1-53
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-12

## HDL Code Generation from System Objects

This example shows how to generate HDL code from MATLAB® code that contains System objects.

### MATLAB Design

The MATLAB code used in this example implements a simple symmetric FIR filter and uses the `dsp.Delay` System object to model state. This example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_sysobj_ex';
testbench_name = 'mlhdlc_sysobj_ex_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Design pattern covered in this example:
% Filter states modeled using DSP System object (dsp.Delay)
% Filter coefficients passed in as parameters to the design
%%%%%%%%%%%%%

% Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, delayed_xout] = mlhdlc_sysobj_ex(x_in, h_in1, h_in2, h_in3, h_in4)
% Symmetric FIR Filter

persistent h1 h2 h3 h4 h5 h6 h7 h8;
if isempty(h1)
    h1 = dsp.Delay;
    h2 = dsp.Delay;
    h3 = dsp.Delay;
    h4 = dsp.Delay;
    h5 = dsp.Delay;
    h6 = dsp.Delay;
    h7 = dsp.Delay;
    h8 = dsp.Delay;
end

h1p = step(h1, x_in);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);
h5p = step(h5, h4p);
h6p = step(h6, h5p);
h7p = step(h7, h6p);
h8p = step(h8, h7p);

a1 = h1p + h8p;
a2 = h2p + h7p;
a3 = h3p + h6p;
a4 = h4p + h5p;
```

```

m1 = h_in1 * a1;
m2 = h_in2 * a2;
m3 = h_in3 * a3;
m4 = h_in4 * a4;

a5 = m1 + m2;
a6 = m3 + m4;

% filtered output
y_out = a5 + a6;
% delayout input signal
delayed_xout = h8p;

end

type(testbench_name);

%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%

% Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sysobj_ex;

x_in = cos(2.*pi.* (0:0.001:2).* (1+(0:0.001:2).*75)).';

h1 = -0.1339;
h2 = -0.0838;
h3 = 0.2026;
h4 = 0.4064;

len = length(x_in);
y_out_sysobj = zeros(1,len);
x_out_sysobj = zeros(1,len);
a = 10;

for ii=1:len
    data = x_in(ii);
    % call to the design 'sfir' that is targeted for hardware
    [y_out_sysobj(ii), x_out_sysobj(ii)] = mlhdlc_sysobj_ex(data, h1, h2, h3, h4);
end

figure('Name', [filename, '_plot']);
subplot(2,1,1);
plot(1:len,x_in); title('Input signal with noise');
subplot(2,1,2);
plot(1:len,y_out_sysobj); title('Filtered output signal');

```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];

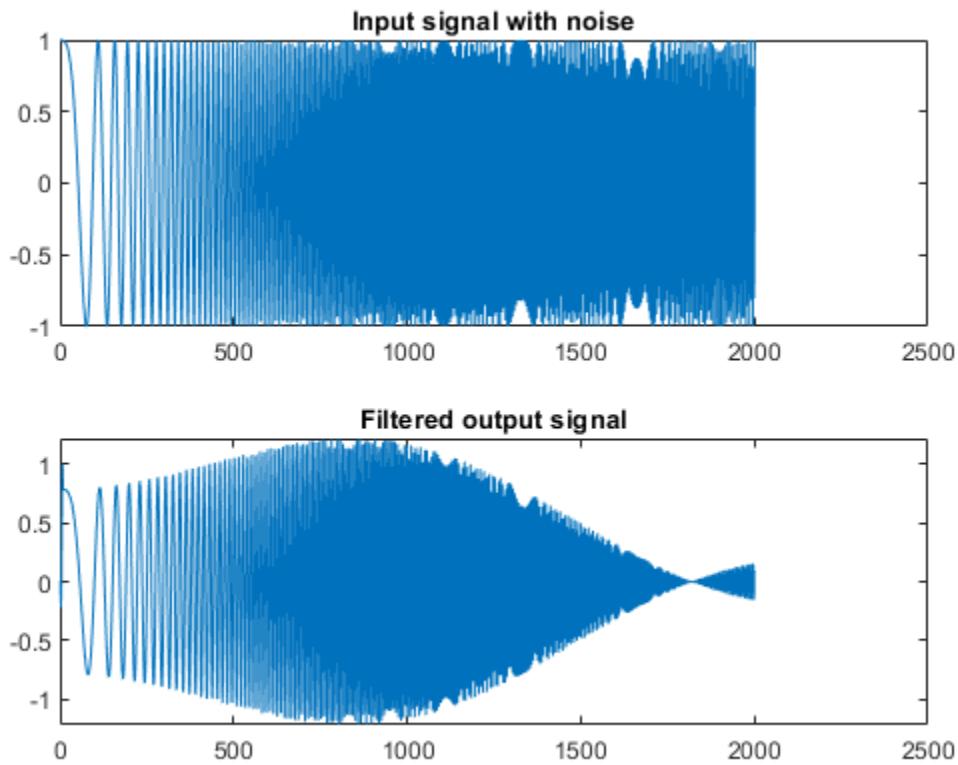
```

```
% Create a temporary folder and copy the MATLAB files.  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');  
mkdir(mlhdlc_temp_dir);  
cd(mlhdlc_temp_dir);  
  
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);  
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sysobj_ex_tb
```



### Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sysobj_prj
```

Next, add the file 'mlhdlc\_sysobj\_ex.m' to the project as the MATLAB Function and 'mlhdlc\_sysobj\_ex\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

## Supported System objects

For a list of System objects supported for HDL code generation, see "Predefined System Objects Supported for HDL Code Generation" on page 1-50.

## Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## HDL Code Generation for Streaming Matrix Inverse System Object

This example shows how HDL Coder™ implements a streaming mode of matrix inverse operation with configurable sizes.

### What is inverse of a matrix

A matrix X is invertible if there exists a matrix Y of the same size such that  $XY = YX = I$ , where I is the Identity matrix. The matrix Y is called inverse of X. A matrix that has no inverse is singular. A square matrix is singular only when its determinant is exactly zero.

Matrix inverse computation involves following steps:

- 1 Cofactor matrix calculation
- 2 Transpose of cofactor matrix
- 3 Multiply reciprocal of determinant of input matrix with transpose of cofactor matrix

### Example:

```
A = [4 12 -16;12 37 -43;-16 -43 98];
Cofactors of 'A' will be calculated from matrix of minors
cof(A) = [1777 488 76;488 136 20;76 20 4];

Transpose of cofactor matrix will be
(cof(A))' = [1777 488 76;488 136 20;76 20 4];

Multiply reciprocal of determinant of 'A' with transpose of cofactor matrix
Ainv = (1/det(A)) * (cof(A))'
      = [49.3611 -13.5556 2.1111;-13.5556 3.7778 -0.5556;2.1111 -0.5556 0.1111];
```

### Matrix Inverse: Gauss-Jordan elimination

To find the inverse of matrix A using Gauss-Jordan elimination, we must find elementary row operations that reduce A to identity matrix(I) and then perform the same operations on Identity matrix(I) to obtain Ainv.

Computation of Matrix Inverse using Gauss-Jordan elimination: Let start with matrix A, and write it down with an Identity matrix next to it:  $[A | I]$

The goal is to make A an identity matrix by applying row transformations and right hand side matrix I also participated in the row transformations, finally reduced to Ainv.

Computation of Ainv involves following steps:

- 1 swapping rows
- 2 make the diagonal elements as 1
- 3 make the non-diagonal elements as 0

### Example:

$$[A \mid I] = \begin{array}{ccc|ccc} & (A) & & (I) & & & \\ & 1 & 2 & 3 & 1 & 0 & 0 \\ & 2 & 5 & 3 & 0 & 1 & 0 \\ & 1 & 0 & 8 & 0 & 0 & 1 \end{array}$$

Find the element with maximum value in the first column and swap the current row with maximum element row  
 swap R1 and R2 rows as R2 contains the largest values.

$$\begin{array}{ccc|ccc} 2 & 5 & 3 & 0 & 1 & 0 \\ = & 1 & 2 & 3 & 1 & 0 & 0 \\ & 1 & 0 & 8 & 0 & 0 & 1 \end{array}$$

Make the diagonal element in the first column as '1'  
 $R1 \rightarrow R1/2$

$$\begin{array}{ccc|ccc} 1 & 2.5 & 1.5 & 0 & 0.5 & 0 \\ = & 1 & 2 & 3 & 1 & 0 & 0 \\ & 1 & 0 & 8 & 0 & 0 & 1 \end{array}$$

Make the non-diagonal elements in the first column as '0'

$$\begin{array}{l} R2 \rightarrow R2 - R1 \\ R3 \rightarrow R3 - R1 \end{array}$$

$$\begin{array}{ccc|ccc} 1 & 2.5 & 1.5 & 0 & 0.5 & 0 \\ = & 0 & -0.5 & 1.5 & 1 & -0.5 & 0 \\ & 0 & -2.5 & 6.5 & 0 & -0.5 & 1 \end{array}$$

Now column 1 has diagonal elements '1' and other elements as '0'. This procedure is repeated for remaining columns and matrix A will be reduced to identity matrix, Identity matrix will be reduced to Ainv.

$$\begin{array}{ccc|ccc} 1 & 0 & 0 & -40 & 16 & 9 \\ = & 0 & 1 & 0 & 13 & -5 & -3 \\ & 0 & 0 & 1 & 5 & -2 & -1 \\ & & & & (I) & & (Ainv) \end{array}$$

### Matrix Inverse: Cholesky decomposition

Matrix Inverse using cholesky decomposition supports only symmetric positive definite matrices. Positive definite means all the eigen values of the matrix should be positive.

Given a symmetric positive definite matrix A:

$$A = L * L', \quad L \text{ is the lower triangular matrix} \\ L' \text{ is the transpose of } L$$

$$\begin{aligned} \text{inv}(A) &= \text{inv}(L * L') \\ &= \text{inv}(L') * \text{inv}(L) \\ &= (\text{inv}(L))' * \text{inv}(L) \end{aligned}$$

$$\text{Ainv} = \text{Linv}' * \text{Linv}, \quad \text{Linv is the inverse of lower triangular matrix} \\ \text{Ainv is the inverse of input matrix}$$

Computation of Ainv involves following steps:

- 1** Lower triangular matrix computation(L)
- 2** Inverse of lower triangular matrix(Linv)
- 3** Multiplication of transpose of Linv with Linv

### Example:

$$A = [4 12 -16; 12 37 -43; -16 -43 98];$$

Lower triangular matrix(L) will be computed using cholesky decomposition  
 $L = [2 \ 0 \ 0; 6 \ 1 \ 0; -8 \ 5 \ 3];$

Linv will be computed using forward substitution method  
 $Linv = [0.5 \ 0 \ 0; -3 \ 1 \ 0; 6.3333 \ -1.6667 \ 0.3333];$

Multiply transpose of Linv with Linv  
 $Ainv = Linv' * Linv$   
 $= [49.3611 \ -13.5556 \ 2.1111; -13.5556 \ 3.7778 \ -0.5556; 2.1111 \ -0.5556 \ 0.1111];$

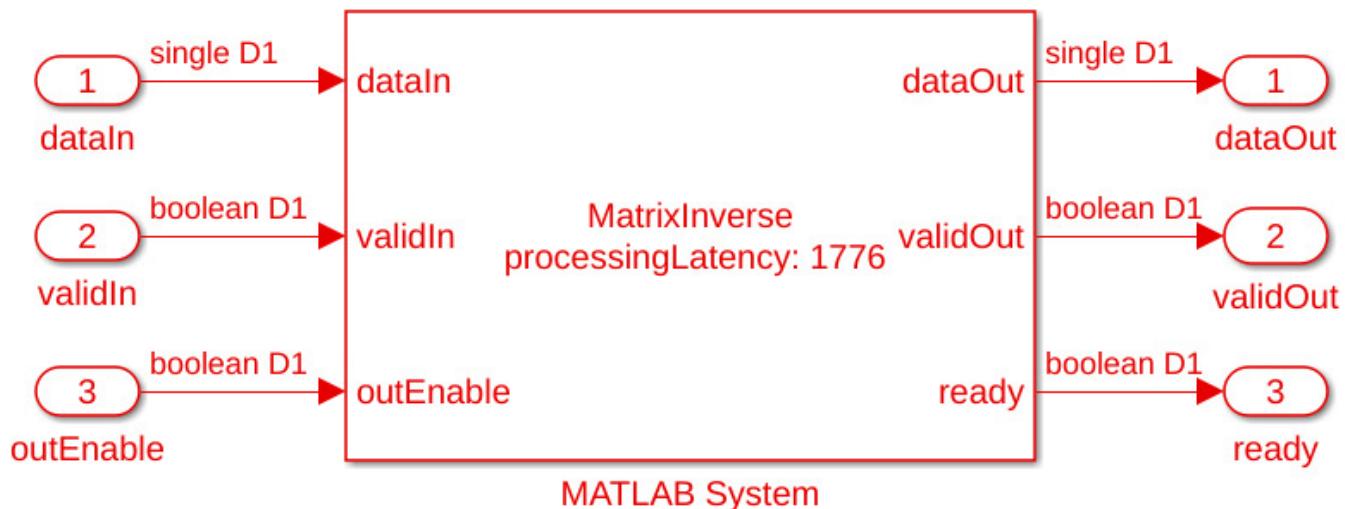
### Benefits of using Gauss-Jordan Elimination

- Gauss-Jordan Elimination supports all square matrices.
- Gauss-Jordan Elimination supports both single and double data types.

### Restrictions for Cholesky implementation

- Matrices for which inverse is to be computed must be symmetric positive-definite.
- Input data types of the matrices must be single and block must be used in the Native Floating Point mode.
- Input matrices must not be larger than 64-by-64 in size.

### Matrix Inverse Subsystem Interface:

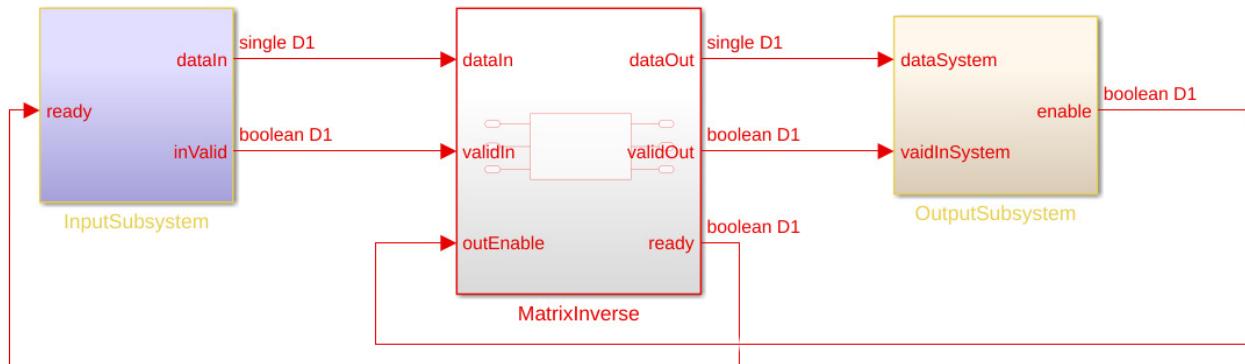


### Matrix Inverse ports description:

Input ports		Output ports	
dataIn	Input data to the module	dataOut	Output data from the module
validIn	Valid signal for input data	validOut	Valid signal for output data
outEnable	Input signal that indicates downstream module is ready to take the output data from processing module	ready	Output signal that indicates processing module is ready to accept the input data in row major from upstream module

## Matrix Inverse Implementation

This example shows streaming matrix inverse implementation

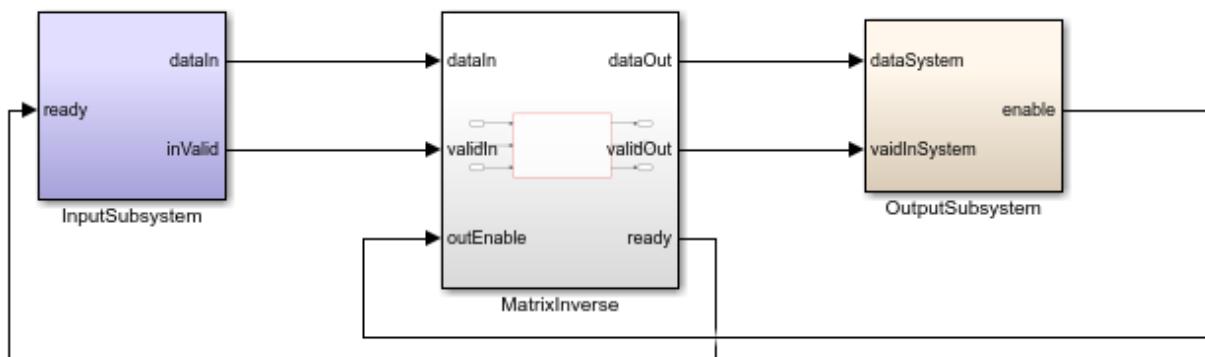


This example model contains three subsystems: **InputSubsystem**, **MatrixInverse**, and **OutputSubsystem**. The **InputSubsystem** is the upstream module that serializes the matrix input to the processing module when the **ready** signal is enabled. The **OutputSubsystem** is the downstream module that deserializes the data from the processing module to a matrix output when the **outEnable** signal is enabled. The **MatrixInverse** is a processing module that implements the matrix inverse operation.

```
open_system('hdlcoder_streaming_mat_inv_max_lat_cholesky');
open_system('hdlcoder_streaming_mat_inv_max_lat_gauss_jordan');
```

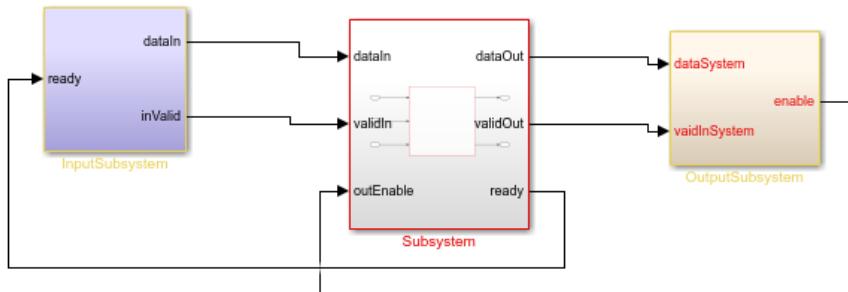
Copyright 2017-2019 The MathWorks, Inc.

This example shows streaming matrix implementation using Cholesky decomposition



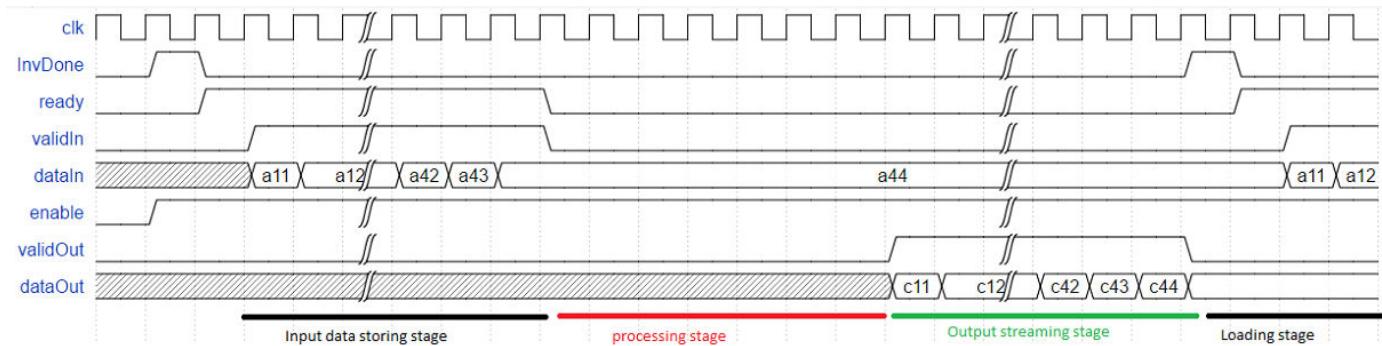
Copyright 2017-2019 The MathWorks, Inc.

This example shows streaming matrix implementation using Gauss-Jordan elimination

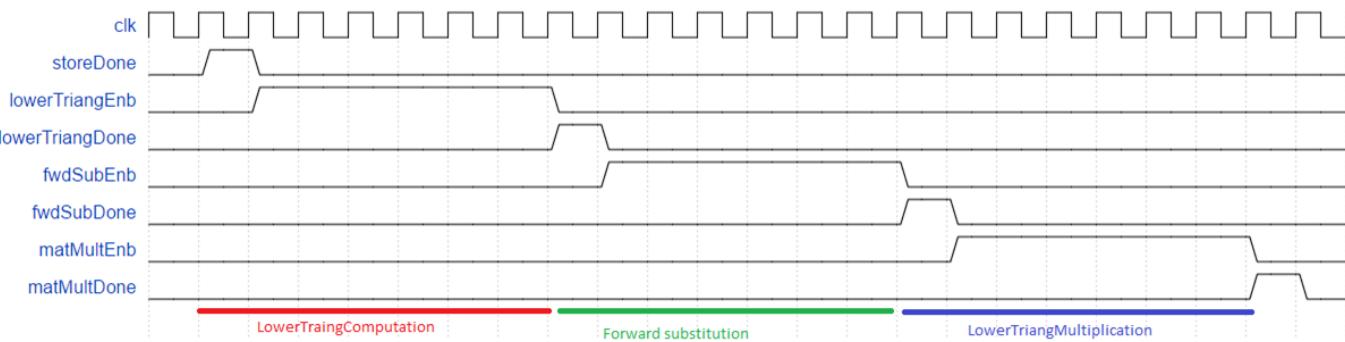


### Matrix Inverse Timing diagrams:

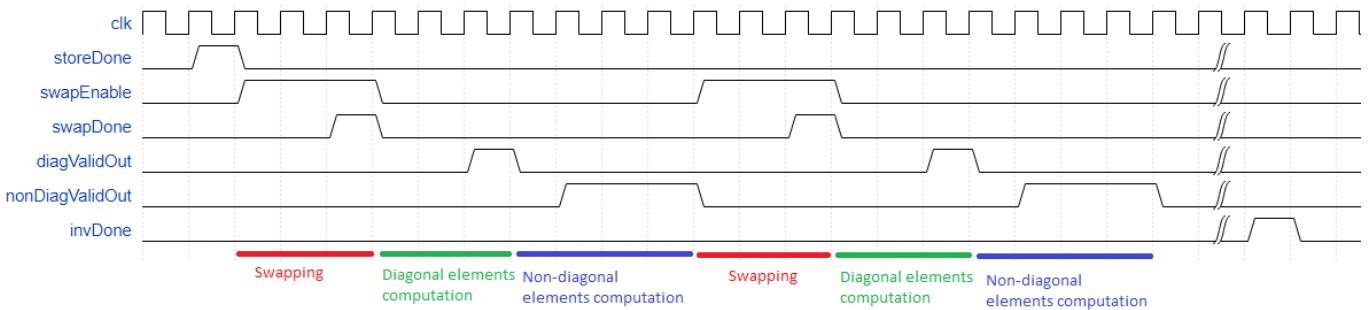
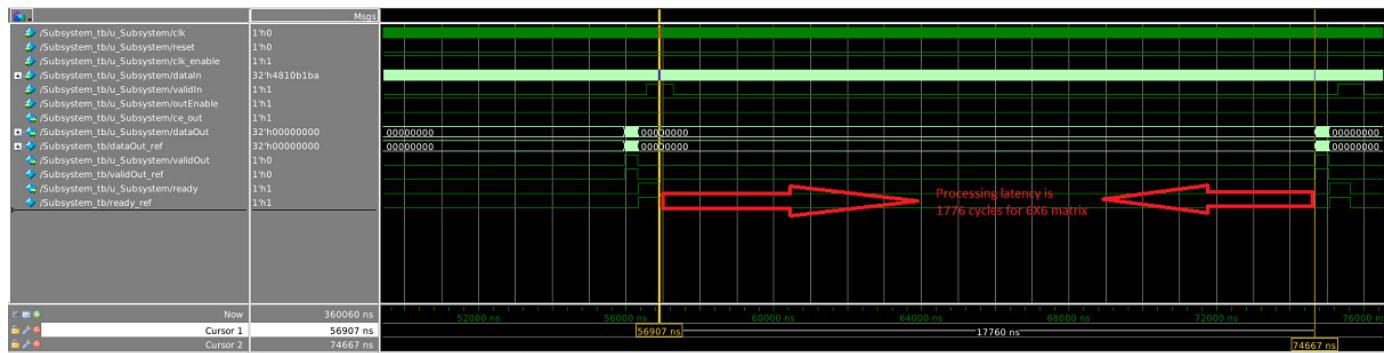
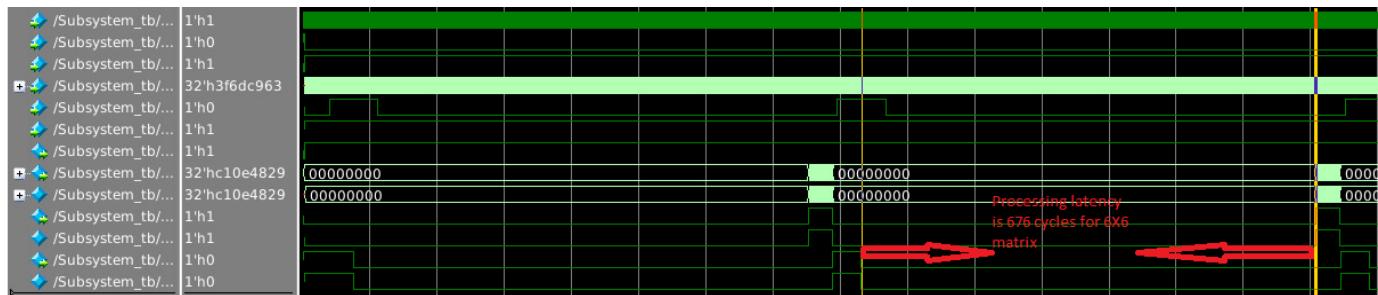
#### Timing diagram for complete system:



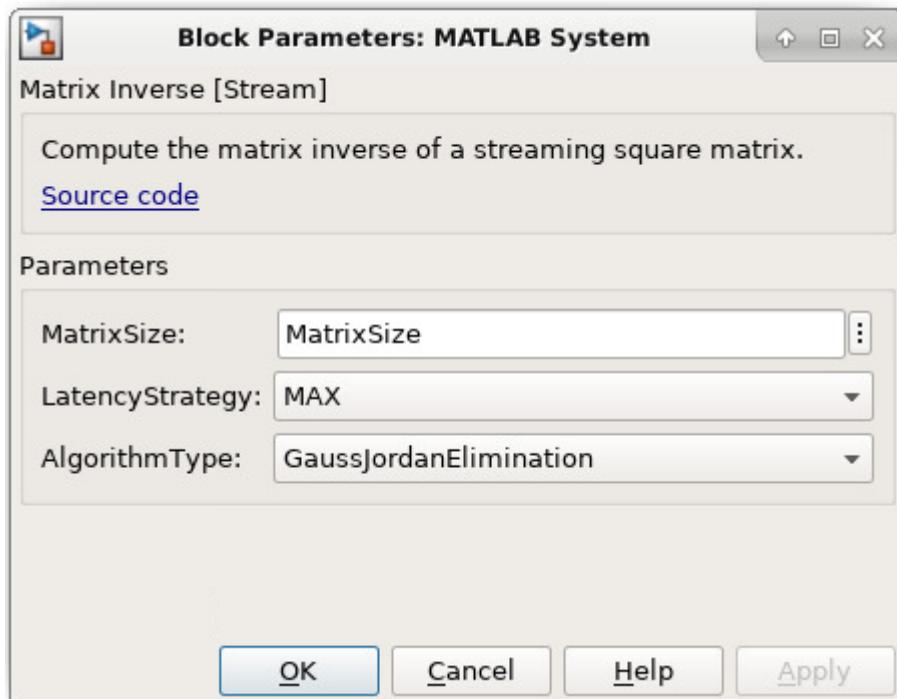
#### Timing diagram for processing stage(Cholesky decomposition):



#### Timing diagram for processing stage(Gauss-Jordan elimination):

**Modelsim result waveform(Cholesky decomposition)****Modelsim result waveform(Gauss-Jordan elimination)**

## Matrix Inverse Block parameters:



MatrixSize : Enter size of input matrix as a positive integer.

LatencyStrategy : Select latency strategy from drop down menu  
({'ZERO', 'MIN', 'MAX'}) which should be same as HDL coder latency strategy. User can see processing latency based on the latency strategy.

AlgorithmType : Select algorithm type from drop down menu({'CholeskyDecomposition', 'GaussJordanElimination'})

## Matrix Inverse Block usage

- 1 Set block parameters of Matlab System block.
- 2 Select input matrix based on the matrix size.
- 3 Generate HDL code for MatrixInverse subsystem.

## Generated code and Generated model

After running code generation for MatrixInverse subsystem, generated code will be

```

>> makehdl(gcb, 'nat', 'on')
### Generating HDL for 'hdlcoder_streaming_matrix_inverse_max_latency/MatrixInverse'.
### Using the config set for model hdlcoder streaming matrix inverse max latency for HDL code generation parameters.
### Starting HDL check.
### One or more feedback loops in the model are inhibiting optimizations. To highlight these loops in your model, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: hdlsrc/hdlcoder streaming matrix inverse max latency/clearhighlighting.m
### Begin Verilog Code Generation for 'hdlcoder_streaming_matrix_inverse_max_latency'.
### Working on RowColCounter as hdlsrc/hdlcoder streaming matrix inverse max latency/RowColCounter.v.
### Working on ReadySignalGenerator as hdlsrc/hdlcoder streaming matrix inverse max latency/ReadySignalGenerator.v.
### Working on StoringDone as hdlsrc/hdlcoder streaming matrix inverse max latency/StoringDone.v.
### Working on WrEnbStore as hdlsrc/hdlcoder streaming matrix inverse max latency/WrEnbStore.v.
### Working on WrAddrStore as hdlsrc/hdlcoder streaming matrix inverse max latency/WrAddrStore.v.
### Working on WrDataStore as hdlsrc/hdlcoder streaming matrix inverse max latency/WrDataStore.v.
### Working on InputDataStoreMemoryControl as hdlsrc/hdlcoder streaming matrix inverse max latency/InputDataStoreMemoryControl.v.
### Working on InputMatrixStoreControl as hdlsrc/hdlcoder streaming matrix inverse max latency/InputMatrixStoreControl.v.
### Working on LowerTriangEnable as hdlsrc/hdlcoder streaming matrix inverse max latency/LowerTriangEnable.v.
### Working on LowerTriangDataValidIn as hdlsrc/hdlcoder streaming matrix inverse max latency/LowerTriangDataValidIn.v.
### Working on LTMemReadControl as hdlsrc/hdlcoder streaming matrix inverse max latency/LTMemReadControl.v.
### Working on LTRowCounter as hdlsrc/hdlcoder streaming matrix inverse max latency/LTRowCounter.v.
### Working on LTColumnCounter as hdlsrc/hdlcoder streaming matrix inverse max latency/LTColumnCounter.v.
### Working on LTRowColCounter as hdlsrc/hdlcoder streaming matrix inverse max latency/LTRowColCounter.v.
### Working on LTProcessController as hdlsrc/hdlcoder streaming matrix inverse max latency/LTProcessController.v.
### Working on DiagDataSelector as hdlsrc/hdlcoder streaming matrix inverse max latency/DiagDataSelector.v.
### Working on DiagDataComputation/nfp_mul_single as hdlsrc/hdlcoder streaming matrix inverse max latency/nfp mul single.v.
### Working on DiagDataComputation/nfp_add_single as hdlsrc/hdlcoder streaming matrix inverse max latency/nfp add single.v.
### Working on DiagDataComputation/nfp_sqrt_single as hdlsrc/hdlcoder streaming matrix inverse max latency/nfp sqrt single.v.
### Working on DiagDataComputation/nfp_relop_single as hdlsrc/hdlcoder streaming matrix inverse max latency/nfp relop single.v.
### Working on DiagDataComputation/nfp_sub_single as hdlsrc/hdlcoder streaming matrix inverse max latency/nfp sub single.v.
### Working on DiagDataComputation as hdlsrc/hdlcoder streaming matrix inverse max latency/DiagDataComputation.v.

```

Generated model contains the MatrixInverse Matlab System block. During modelsim simulation code generation outputs are compared with Matlab System block outputs.

## Synthesis statistics

### Cholesky decomposition

SynthesisTool : Altera Quartus II 16.0  
 SynthesisToolChipFamily : Stratix V  
 SynthesisToolDeviceName : 5SEE9F45C2

Size	Fmax	ALMs	LABs	DSPs	Comb ALUTs	RAMs	Latency
4X4	316.36	7240	1223	12	11386	45	958
8X8	267.59	12889	2193	24	20415	76	2803
16X16	237.47	24219	4190	48	37884	138	9122
24X24	223.31	35738	6133	72	56826	205	18961
32X32	198.97	46949	8028	96	73918	266	32309

SynthesisTool : Xilinx Vivado 2017.4  
 SynthesisToolChipFamily : xc7v2000t  
 SynthesisToolDeviceName : ffg1761

Size	Fmax	Slices	LUTs	DSPs	Latency
4X4	233.81	4783	13968	24	958
8X8	261.85	8700	24362	48	2803
16X16	169.69	15466	44077	96	9122
24X24	221.29	25543	79260	144	18961
32X32	161.29	26371	80258	192	32309

### Gauss-Jordan elimination

SynthesisTool : Altera Quartus II 18.1  
 SynthesisToolChipFamily : Stratix V  
 SynthesisToolDeviceName : 5SEE9F45C2

Size	Fmax	ALMs	LABs	DSPs	Comb ALUTs	RAMs	Latency
4X4	354.99	3509	650	2	4752	29	356
8X8	334.67	4793	826	2	6665	35	1156
16X16	288.77	7119	1191	2	9213	51	5636
24X24	250.88	9356	1617	2	11686	67	16516
32X32	237.98	12007	2072	2	14562	83	36868

SynthesisTool : Xilinx Vivado 2018.3  
 SynthesisToolChipFamily : xc7v2000t  
 SynthesisToolDeviceName : ffg1761

Size	Fmax	Slices	LUTs	DSPs	Latency
4X4	254.97	2462	6657	2	356
8X8	208.72	2885	7573	2	1156
16X16	151.03	4234	10648	2	5636
24X24	154.14	4828	11291	2	16516
32X32	113.39	6375	13130	2	36868

## References

- `inv`
- `chol`
- `eig`
- `rref`

LocalWords: eigen Ainv Linv serial deserializer Lowertriangular YX Cofactors LocalWords:  
Matrixmultiplication validout muxing modelsim cof gauss

# HDL Code Generation for Streaming Matrix Multiply System Object

This example shows how HDL Coder™ implements a streaming mode of matrix multiplication with configurable sizes.

## How to Multiply Matrices

Let A, B be two matrices then  $C = A * B$  is the matrix multiplication of A and B. If A is an m-by-p and B is an p-by-n matrix, then C is an m-by-n matrix defined by

$$C(i,j) = A(i,1)B(1,j) + A(i,2)B(2,j) + \dots + A(i,p)B(p,j)$$

This inner definition says that  $C(i,j)$  is the inner product of ith row of A with the jth column of B

For non scalar A and B, the number of columns of A must equal to the number of rows of B

### Example:

```
A = [1 3 5;2 4 7];           (2 X 3 matrix)
B = [-5 8 11;3 9 21;4 0 8];   (3 X 3 matrix)
```

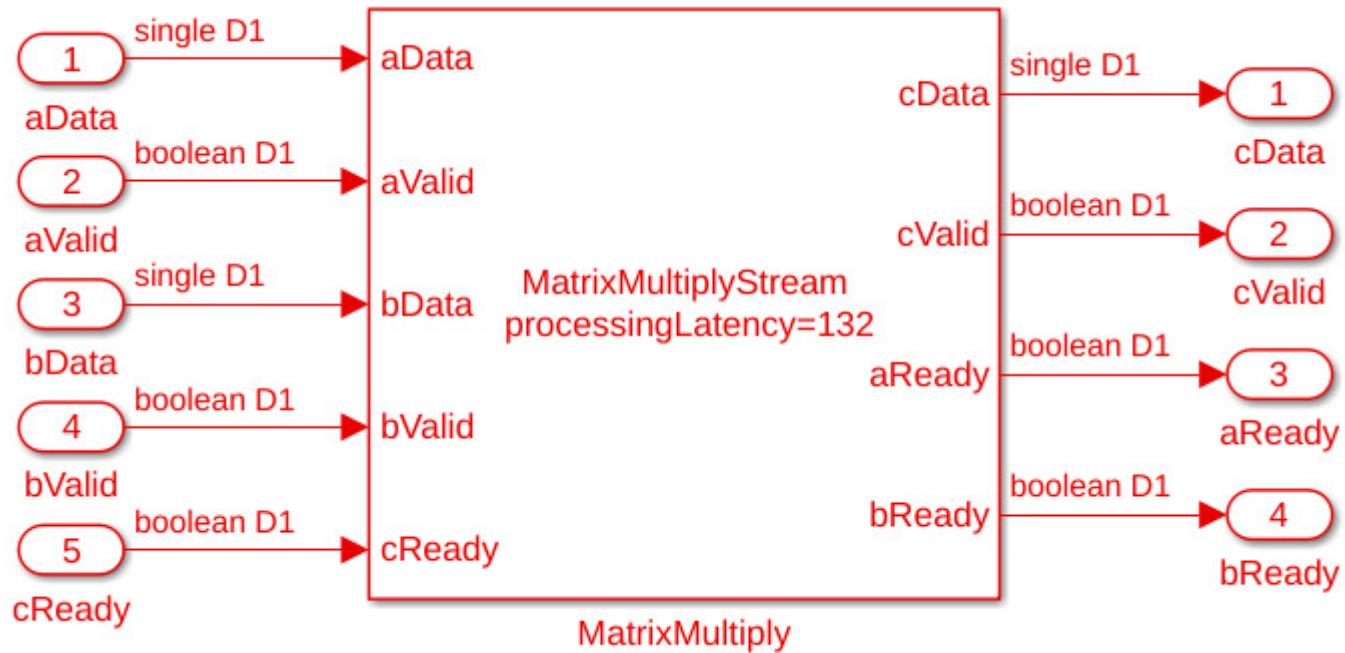
After calculating inner product of rows of A with columns of B

```
C = [24 35 114;30 52 162];   (2 X 3 matrix)
```

## Introduction

Streaming Matrix Multiply supports multiplication of two matrices with configurable matrix sizes and dot product size. Dot product size is equal to the number of multipliers used for computation. This block can accept serialized input data from matrix in row major or column major format.

### Matrix Multiply Interface:

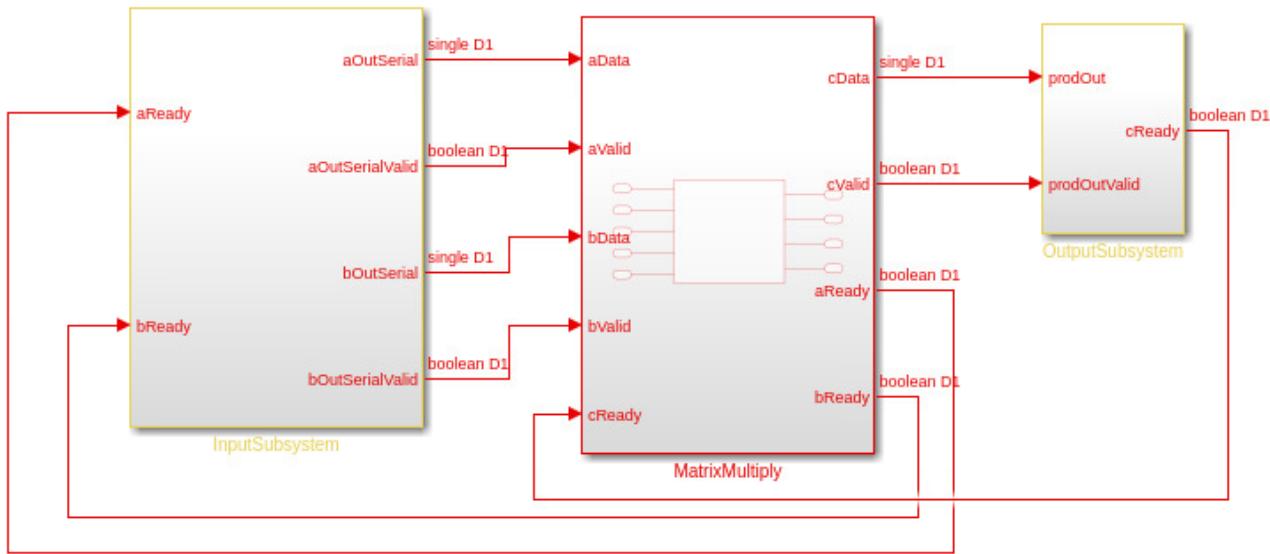


### Matrix Multiply ports description:

Input ports		Output ports	
aData	Matrix-A input data to the module	cData	Matrix-C output data from processing module
aValid	Valid signal for matrix-A input data	cValid	Valid signal for matrix-C output data
bData	Matrix-B input data to the module	aReady	Output signal that indicates the processing module is ready to accept the matrix-A input data from upstream module
bValid	Valid signal for matrix-B input data	bReady	Output signal that indicates the processing module is ready to accept the matrix-B input data from upstream module
cReady	Input signal that indicates downstream module is ready to take output data from processing module		

## MatrixMultiply Implementation

This example shows streaming matrix multiplication operation

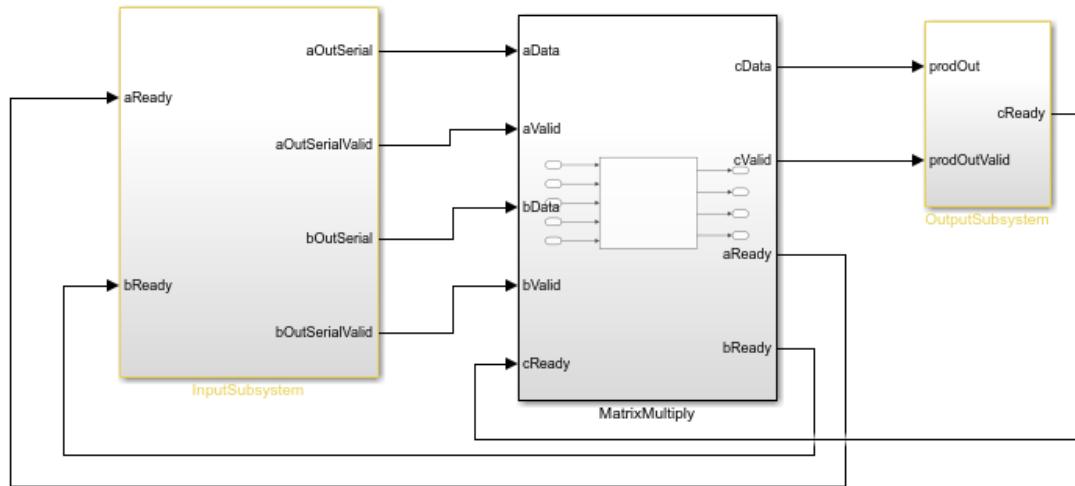


This example model contains three subsystems: **InputSubsystem**, **MatrixMultiply** and **OutputSubsystem**. The **InputSubsystem** is the upstream module that serializes the matrix inputs(A,B) to the processing module when **aReady** and **bReady** signals are enabled. The **OutputSubsystem** is the downstream module that deserializes the data from the processing module to matrix output(C) when the **cReady** signal is enabled. The **MatrixMultiply** is a processing module that implements the matrix multiplication.

```
open_system('hdlcoder_streaming_matrix_multiply_max_latency');
```

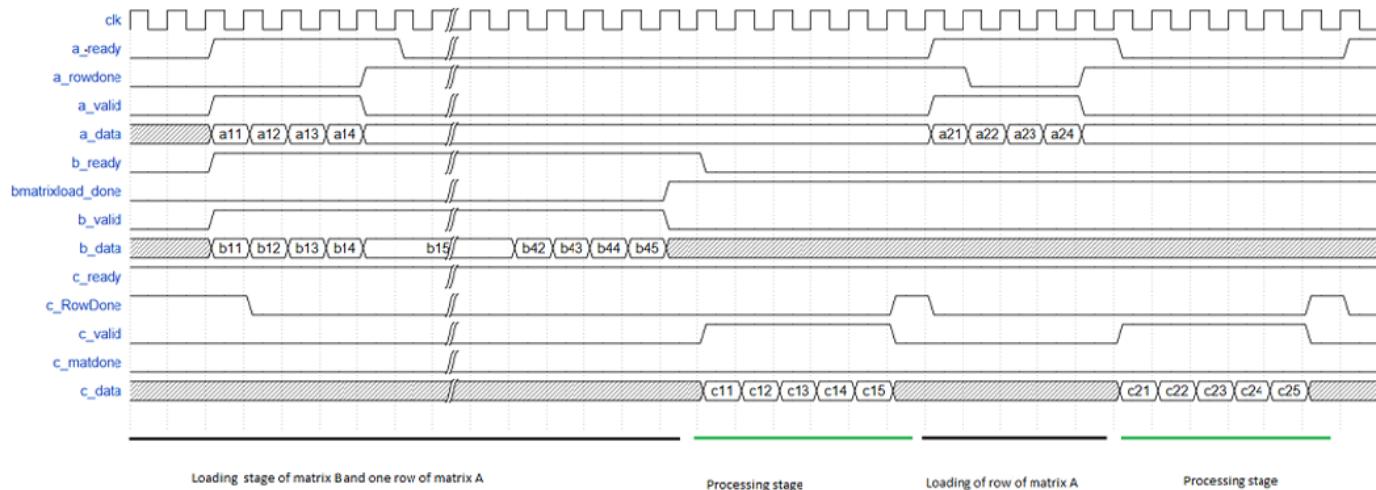
This example shows streaming matrix multiplication operation

Copyright 2018-2019 The MathWorks, Inc.

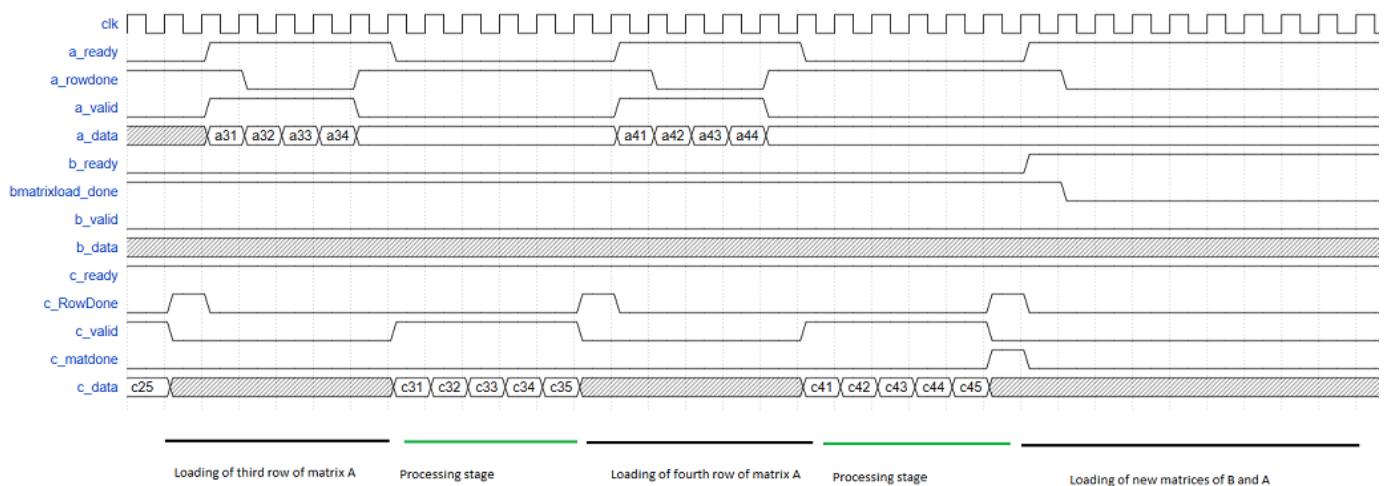


### Matrix Multiply Timing diagrams

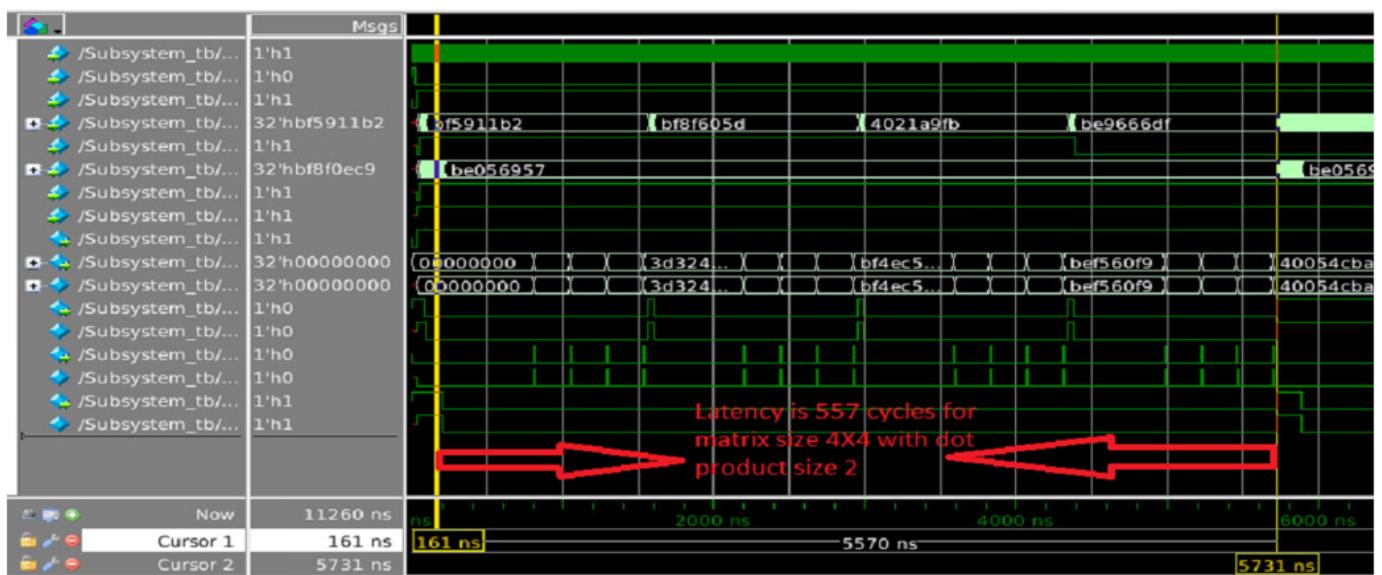
#### Timing diagram

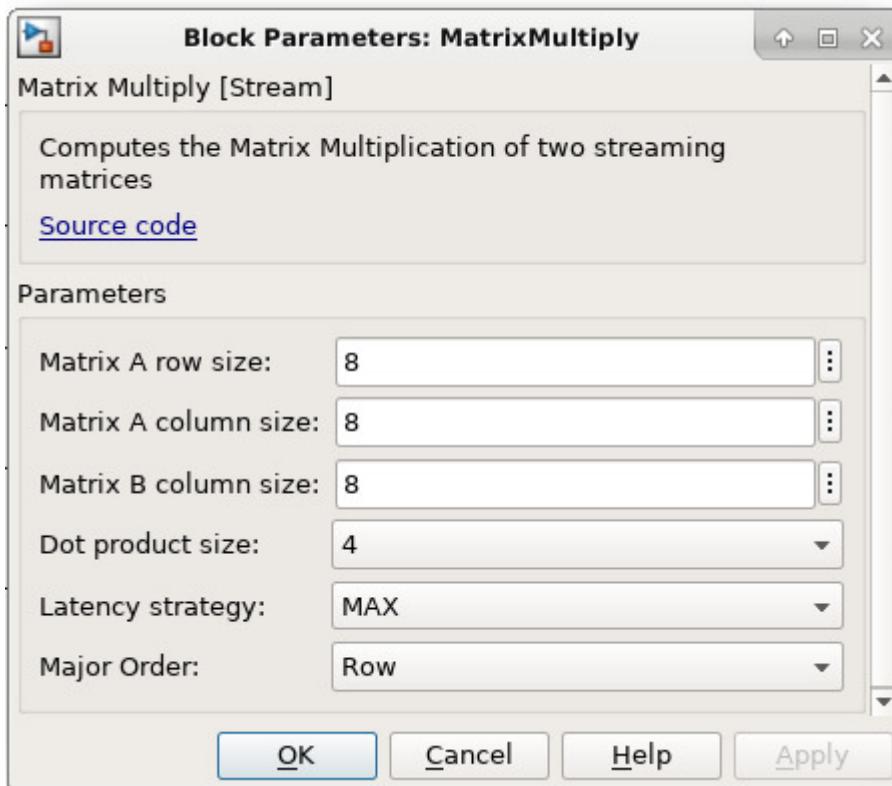


#### Timing diagram



### Modelsim results for streaming matrix multiply



**Matrix Multiply Block parameters:**

Matrix-A Row Size	: Enter row size of input matrix A as a positive integer.
Matrix-A Column Size	: Enter column size of input matrix B as a positive integer which is equals to input matrix B row size.
Matrix-B Column Size	: Enter column size of input matrix B as a positive integer.
Dot product size	: Select dot product size from drop down menu(1,2,4,8,16,32,64) which should be less than input matrix A column size.
LatencyStrategy	: Select latency strategy from drop down menu ({'ZERO', 'MIN', 'MAX'}) which should be same as HDL coder latency strategy. Processing latency depends on the latency strategy.
Major Order	: Select row major or column major based on the input data streaming.

**Matrix Multiply Block usage**

- 1 Set block parameters of Matlab System block.
- 2 Select input matrix sizes based on the values set in block parameters.
- 3 Generate HDL code for MatrixMultiply subsystem.

**Generated code and Generated model**

After running code generation for MatrixMultiply subsystem, generated code will be

```

>> makehdligcb, 'on'
### Generating HDL for 'hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply'.
### Using the config set for model hdlcoder\_streaming\_matrix\_multiply\_max\_latency for HDL code generation parameters.
### Starting HDL check.
### One or more feedback loops in the model are inhibiting optimizations. To highlight these loops in your model, click the following MATLAB script: hdsrc/hdlcoder\_stre
### To clear highlighting, click the following MATLAB script: hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/clearhighlighting.m
### Begin VHDL Code Generation for 'hdlcoder_streaming_matrix_multiply_max_latency'.
### Working on matrixAStoreControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixAStoreControl.vhd.
### Working on matrixBStoreControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBStoreControl.vhd.
### Working on matrixAMemoryReadAddress as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixAMemoryReadAddress.vhd.
### Working on matrixBMemoryReadAddress as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemoryReadAddress.vhd.
### Working on readAddressValid as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/readAddressValid.vhd.
### Working on memoryReadAddressControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/memoryReadAddressControl.vhd.
### Working on matrixASubColumnControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixASubColumnControl.vhd.
### Working on matrixAMemoryWriteEnableDecoder as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixAMemoryWriteEnableDecoder.vhd.
### Working on matrixAMemory/ SingleDualPortRAM_generic as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/alphaSingleDualPortRAM\_generic.vhd.
### Working on matrixAMemory as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixAMemory.vhd.
### Working on matrixAMemoryController as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixAMemoryController.vhd.
### Working on matrixBMemory/writeControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemoryWriteControl.vhd.
### Working on matrixBMemory/writeEnableDecoder as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemoryWriteEnableDecoder.vhd.
### Working on matrixBMemory as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemory.vhd.
### Working on matrixBMemoryController as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemoryController.vhd.
### Working on matrixBMemoryReadDataDecoder as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixBMemoryReadDataDecoder.vhd.
### Working on memoryController as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/memoryController.vhd.
### Working on dotProduct/nfp_rul_single as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/nfp\_rul\_single.vhd.
### Working on dotProduct/nfp_add_single as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/nfp\_add\_single.vhd.
### Working on dotProduct as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/dotProduct.vhd.
### Working on accumulator as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/accumulator.vhd.
### Working on processingSystem as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/processingSystem.vhd.
### Working on matrixMultiplyOutputControl as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixMultiplyOutputControl.vhd.
### Working on matrixMultiplyStream as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/matrixMultiplyStream.vhd.
### Working on hdlcoder_streaming_matrix_multiply_max_latency/MatrixMultiply as hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/MatrixMultiply.vhd.
### Generating package file hdsrc/hdlcoder\_streaming\_matrix\_multiply\_max\_latency/MatrixMultiply\_pkq.vhd.
### Creating HDL Code Generation Check Report MatrixMultiply\_report.html
### HDL check for 'hdlcoder_streaming_matrix_multiply_max_latency' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.

```

Generated model contains the MatrixMultiply Matlab System block. During modelsim simulation code generation outputs are compared with Matlab System block outputs.

### Synthesis statistics

#### Altera Stratix V 5SEE9F45C2:

Matrix size	Dot Product size	Frequency (MHz)	Latency	Utilization		
				ALMs	LABs	DSP slices
A (4,4),B(4,4)	2	500	557	1016	213	2
A (4,4),B(4,4)	4	426.62	200	1597	299	4
A (8,8),B(8,8)	2	500	3481	1061	209	2
A (8,8),B(8,8)	4	500	2057	1934	353	4
A (8,8),B(8,8)	8	465.77	585	3626	669	8

#### Xilinx Virtex 7 xc7v2000t:

Matrix size	Dot Product Size	Frequency (MHz)	Latency	Utilization		
				LUTS	Slices	DSP slices
A (4,4),B(4,4)	2	376.08	557	1960	745	4
A (4,4),B(4,4)	4	409.17	200	2788	1106	8
A (8,8),B(8,8)	2	343.29	3481	2073	818	4
A (8,8),B(8,8)	4	389.56	2057	3798	1548	8
A (8,8),B(8,8)	8	412.88	585	6124	2792	16

## Restrictions

- Matrix dot product sizes can be 1 or a power of 2. The allowed sizes are 1, 2, 4, 8, 16, 32 and 64.
- Input data types of the matrices must be `single` and block must be used in the `Native Floating Point` mode.
- Input matrices must not be larger than 64-by-64 in size.

## References

- `mtimes`

LocalWords: serail deserializer AStore BStore Cmatrix muxing LocalWords: Amemory Bmemory  
modelsim

# HDL Code Generation from hdl.RAM System Object

This example shows how to generate HDL code from MATLAB® code from hdl.RAM system object in MATLAB and infer RAM in generated hardware.

## MATLAB Design

This example shows implementation of a line delay that uses a memory in a ring structure, where data is written in one position and read from another position in such a way that the data written will be read after a specific number of cycles. An efficient implementation of this architecture on Virtex FPGAs uses the on-chip Dual Port Block RAMs and an address counter. The Block RAMs can be configured as 512x8 or 256x9 synchronous Dual Port RAMs. To parameterize the delay length, the RAM write address is generated by a counter and the read address is generated by adding a constant K to the write address. If the memory size is M, the input will be read M-K clock cycles after it was written to the memory, hence implementing M-K word shift behaviour.

```
design_name = 'mlhdlc_hdram';
testbench_name = 'mlhdlc_hdram_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%#codegen
function data_out = mlhdlc_hdram(data_in)
%
% This example shows implementation of a line delay that uses a memory in a
% ring structure, where data is written in one position and read from
% another position in such a way that the data written will be read after a
% specific number of cycles. An efficient implementation of this
% architecture on Virtex FPGAs uses the on-chip Dual Port Block RAMs and an
% address counter. The Block RAMs can be configured as 512x8 or 256x9
% synchronous Dual Port RAMs. To parameterize the delay length, the RAM
% write address is generated by a counter and the read address is generated
% by adding a constant K to the write address. If the memory size is M, the
% input will be read M-K clock cycles after it was written to the memory,
% hence implementing M-K word shift behaviour.

% Copyright 2012-2015 The MathWorks, Inc.

persistent hRam;
if isempty(hRam)
    hRam = hdl.RAM('RAMType', 'Dual port');
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 0;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
```

```
ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM
[~, ramRdDout] = step(hRam, ramWriteData, ramWriteAddr, ramWriteEnable, ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;

type(testbench_name);

function mlhdlc_hdram_tb
%
% Copyright 2012-2015 The MathWorks, Inc.

clear test_hdram;

data = 100:200;
ring_out = zeros(1, length(data));

for ii=1:100
    ring_in = data(ii);
    ring_out(ii) = mlhdlc_hdram(ring_in);
end

figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(1:100,data(1:100));
title('Input data to the ring counter')

subplot(2,1,2);
plot(1:100,ring_out(1:100));
title('Output data')

end
```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];

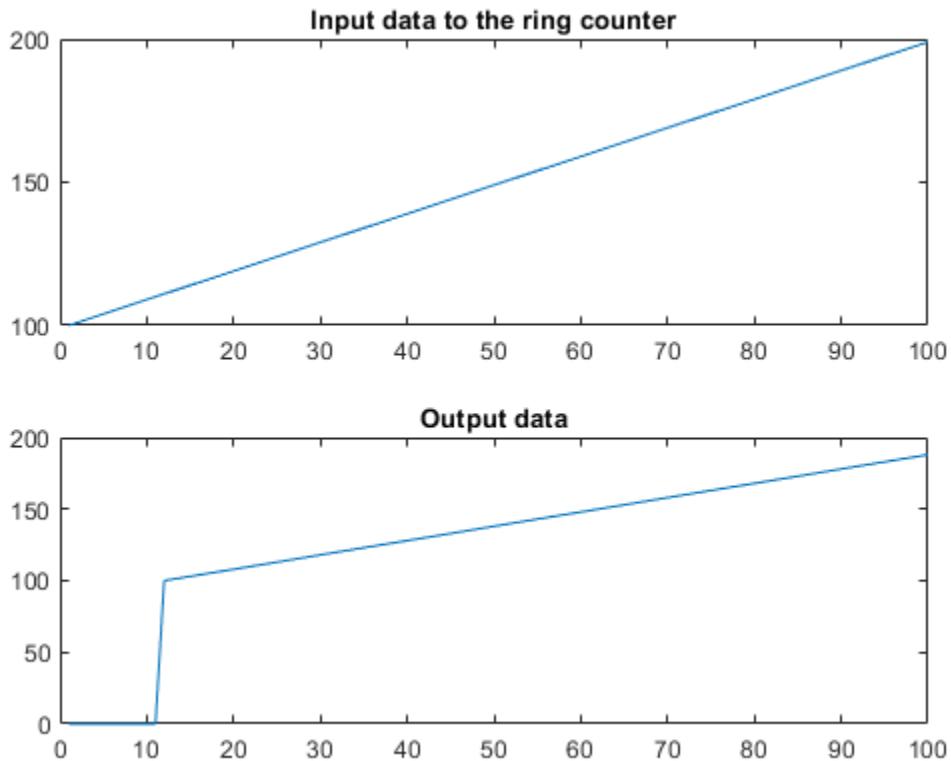
% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_hdlram_tb
```



## Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sysobj_prj
```

Next, add the file 'mlhdlc\_hdlram.m' to the project as the MATLAB Function and 'mlhdlc\_hdlram\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

## Supported System objects

For a list of System objects supported for HDL code generation, see “Predefined System Objects Supported for HDL Code Generation” on page 1-50.

## Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sysobj_intro'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# HDL Code Generation from A Non-Restoring Square Root System Object

This example shows how to check, generate and verify HDL code from MATLAB® code that instantiates a non-restoring square root system object.

## MATLAB Design

The MATLAB code used in this example is a non-restoring square root engine suitable for implementation in an FPGA or ASIC. The engine uses a multiplier-free minimal area implementation based on [1] decision convolutional decoding, implemented as a System object. This example also shows a MATLAB test bench that tests the engine.

```
design_name = 'mlhdlc_sysobj_nonrestsqrt.m';
testbench_name = 'mlhdlc_sysobj_nonrestsqrt_tb.m';
sysobj_name = 'mlhdlc_msyobj_nonrestsqrt.m';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Non-restoring Square Root
%
% Key Design pattern covered in this example:
% (1) Using a user-defined system object
% (2) The 'step' method can be called only per system object in a design iteration
%%%%%%%%%%%%%
function [Q_o,Vld_o] = mlhdlc_sysobj_nonrestsqrt(D_i, Init_i)

% Copyright 2014-2015 The MathWorks, Inc.

    persistent hSqrt;

    if isempty(hSqrt)
        hSqrt = mlhdlc_msyobj_nonrestsqrt();
    end

    [Q_o,Vld_o] = step(hSqrt, D_i,Init_i);
end

type(testbench_name);

% Nonrestoring Squareroot Testbench

% Copyright 2014-2015 The MathWorks, Inc.

% Generate some random data
rng('default'); % set the random number generator to a consistent state
nsamp = 100; %number of samples
nbits = 32; % fixed-point word length
nfrac = 31; % fixed-point fraction length

data_i = fi(rand(1,nsamp), numerictype(0,nbits,nfrac));
```

```
% clear any persistent variables in the HDL function
clear mlhdlc_sysobj_nonrestsqrt

% Determine the "golden" sqrt results
data_go = sqrt(data_i);

% Commands for the sqrt engine
LOAD_DATA = true;
CALC_DATA = false;

% Pre-allocate the result array
data_o = zeros(1,nsamp, 'like', data_go);
% Load in a sample, then iterate until the results are ready
cyc_cnt = 0;
for i = 1:nsamp
    % Load the new sample into the sqrt engine
    [~, vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),LOAD_DATA);
    cyc_cnt = cyc_cnt + 1;
    while(vld == false)
        % Iterate until the result has been found
        [data_o(i), vld] = mlhdlc_sysobj_nonrestsqrt(data_i(i),CALC_DATA);
        cyc_cnt = cyc_cnt + 1;
    end
end

% find the integer representation of the result data
idt = numerictype(0,ceil(nbits/2),0);
% find the error in terms of integer bits
ierr = abs(double(reinterpretcast(data_o,idt))-double(reinterpretcast(data_go,idt)));
% find the error in terms of real-world values
derr = abs(double(data_o)- double(data_go));
pct_err = 100*derr ./ double(data_go);

fprintf('Maximum Error: %d (%0.3f %%)\n', max(derr), max(pct_err));
fprintf('Maximum Error (as unsigned integer): %d\n', max(ierr));
fprintf('Number of cycles: %d (%d per sample)\n', cyc_cnt, cyc_cnt / nsamp);

%EOF
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_so_nonrestsqrt'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, sysobj_name), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

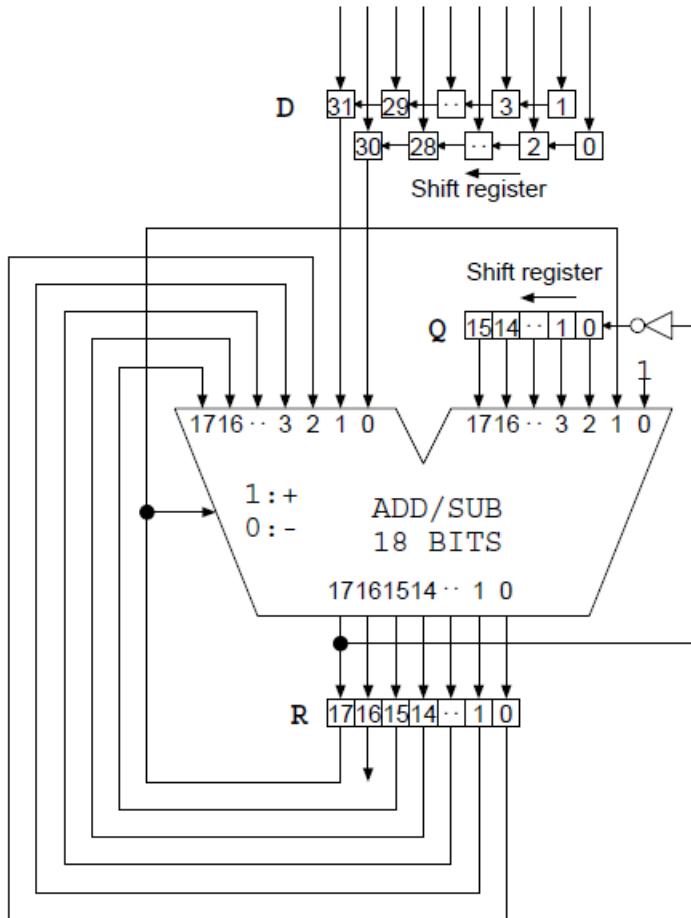
```
mlhdlc_sysobj_nonrestsqrt_tb
```

```
Maximum Error: 3.051758e-05 (0.028 %)
Maximum Error (as unsigned integer): 1
Number of cycles: 2000 ( 20 per sample)
```

## Hardware Implementation of the Non-Restoring Square Root Algorithm

This algorithm implements the square root operation in a minimal area by using a single adder/subtractor with no mux (compared to a restoring algorithm that requires a mux). The square root is calculated using a series of shifts and adds/subs, so uses no multipliers (compared to other implementations which require a multiplier).

The overall architecture of the algorithm is shown below, as described in [1].



This implementation of the algorithm uses a minimal area approach that requires multiple cycles to calculate each result. The overall calculation time can be approximated as [Input Word Length / 2], with a few cycles of overhead to load the incoming data.

## Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_nonrestsqrt
```

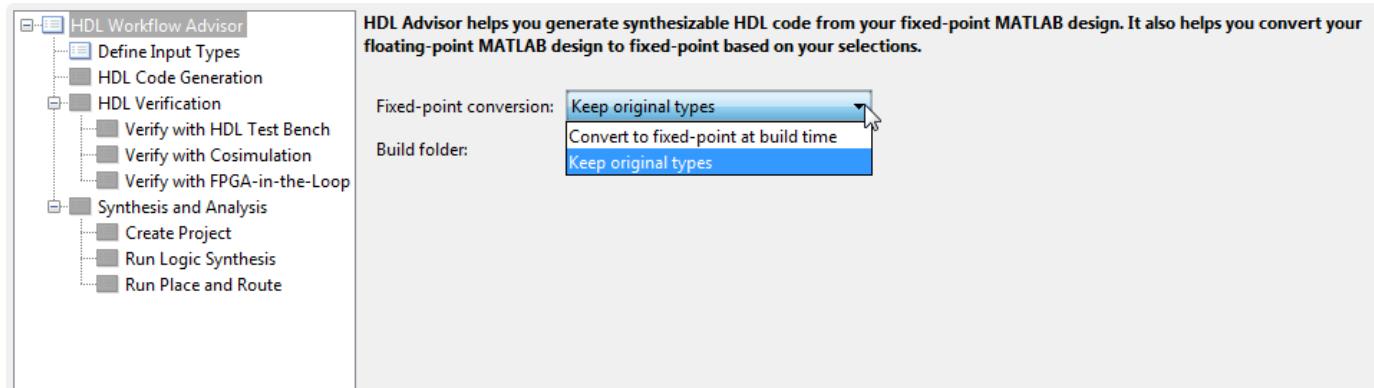
Next, add the file 'mlhdlc\_sysobj\_nonrestsqrt.m' to the project as the MATLAB function and 'mlhdlc\_sysobj\_nonrestsqrt\_tb.m' as the MATLAB test bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Skip Fixed-Point Conversion

As the design is already in fixed-point, we do not need to perform automatic conversion.

Launch the HDL Advisor and choose 'Keep original types' on the option 'Fixed-point conversion':



## Run HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

## Supported System objects

For a list of System objects supported for HDL code generation, see "Predefined System Objects Supported for HDL Code Generation" on page 1-50.

## Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_so_nonrestsqrt'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## References

- [1] Li, Y. and Chu, W. (1996) "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations". IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '96 Austin, Texas USA (7-9 October, 1996), pp. 538-544. doi: 10.1109/ICCD.1996.563604

## HDL Code Generation from Viterbi Decoder System Object

This example shows how to check, generate and verify HDL code from MATLAB® code that instantiates a Viterbi Decoder System object.

### MATLAB Design

The MATLAB code used in this example is a Viterbi Decoder used in hard decision convolutional decoding, implemented as a System object. This example also shows a MATLAB test bench that tests the decoder.

```
design_name = 'mlhdlc_sysobj_viterbi';
testbench_name = 'mlhdlc_sysobj_viterbi_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Viterbi Decoder
%
% Key Design pattern covered in this example:
% (1) Using comm system toolbox ViterbiDecoder function
% (2) The 'step' method can be called only per system object in a design iteration
%%%%%%%%%%%%%

% Copyright 2011-2015 The MathWorks, Inc.

function decodedBits = mlhdlc_sysobj_viterbi(inputSymbol)

persistent hVitDec;

if isempty(hVitDec)
    hVitDec = comm.ViterbiDecoder('InputFormat','Hard', 'OutputDataType', 'Logical');
end

decodedBits = step(hVitDec, inputSymbol);

type(testbench_name);

% Viterbi_tb - testbench for Viterbi_dut
%
% Copyright 2011-2015 The MathWorks, Inc.

numErrors = 0;
% rand stream
original_rs = RandStream.getGlobalStream;
rs = RandStream.create('mrg32k3a', 'seed', 25);
%RandStream.getGlobalStream(rs);
rs.reset;
% convolutional encoder
hConvEnc = comm.ConvolutionalEncoder;
% BER
hBER = comm.ErrorRate;
hBER.ReceiveDelay = 34;
reset(hBER);
```

```
% clear persistent variables in the design between runs of the testbench
clear mlhdlc_msobj_viterbi;

for numSymbols = 1:10000
    % generate a random bit
    inputBit = logical(randi([0 1], 1, 1));

    % encode it with the Convolutional Encoder - rate 1/2
    encodedSymbol = step(hConvEnc, inputBit);

    % optional - add noise

%%%%%%%%%%%%%
% call Viterbi Decoder DUT to decode the symbol
%%%%%%%%%%%%%
vitdecOut = mlhdlc_msobj_viterbi(encodedSymbol);

ber = step(hBER, inputBit, vitdecOut);
end

fprintf('%s\n', repmat('%', 1, 38));
fprintf('%%%%%%%%%%%%% %s %%%%%%%%%%%%%%\n', 'Viterbi Decoder Output');
fprintf('%s\n', repmat('%', 1, 38));
fprintf('Number of bits %d, BER %g\n', numSymbols, ber(1));
fprintf('%s\n', repmat('%', 1, 38));

% EOF
```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderden');
mlhdlc_temp_dir = [tempdir 'mlhdlc_so_viterbi'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

`mlhdlc_msobj_viterbi_tb`

```
%%%%%%%%%%%%%
%%%%%% Viterbi Decoder Output %%%%%%
%%%%%%%%%%%%%
Number of bits 10000, BER 0
%%%%%%%%%%%%%
```

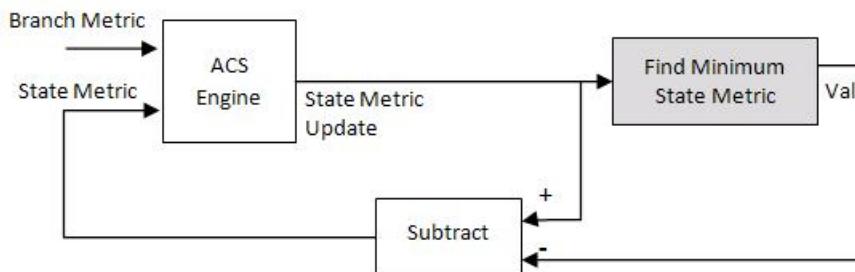
## Hardware Implementation of Viterbi Decoding Algorithm

There are three main components in the Viterbi decoding algorithm. They are the branch metric computation (BMC), add-compare-select (ACS), and traceback decoding. The following diagram illustrates the three units in the Viterbi decoding algorithm.



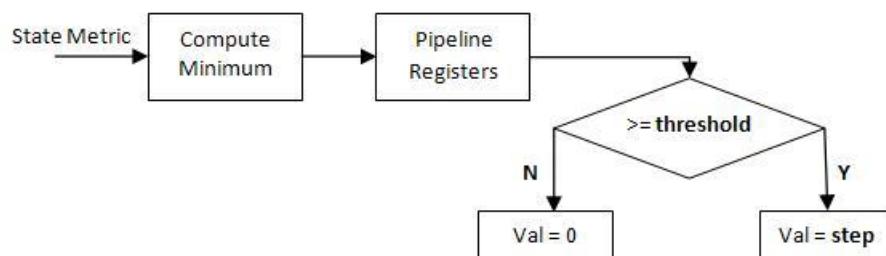
## The Renormalization Method

The Viterbi decoder prevents the overflow of the state metrics in the ACS component by subtracting the minimum value of the state metrics at each time step, as shown in the following figure.



Obtaining the minimum value of all the state metric elements in one clock cycle results in a poor clock frequency for the circuit. The performance of the circuit may be improved by adding pipeline registers. However, simply subtracting the minimum value delayed by pipeline registers from the state metrics may still lead to overflow.

The hardware architecture modifies the renormalization method and avoids the state metric overflow in three steps. First, the architecture calculates values for the threshold and step parameters, based on the trellis structure and the number of soft decision bits. Second, the delayed minimum value is compared to the threshold. Last, if the minimum value is greater than or equal to the threshold value, the implementation subtracts the step value from the state metric; otherwise no adjustment is performed. The following figure illustrates the modified renormalization method.



## Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_viterbi
```

Next, add the file 'mlhdlc\_sysobj\_viterbi.m' to the project as the MATLAB function and 'mlhdlc\_sysobj\_viterbi\_tb.m' as the MATLAB test bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

### Supported System objects

For a list of System objects supported for HDL code generation, see "Predefined System Objects Supported for HDL Code Generation" on page 1-50.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde  
mlhdlc_temp_dir = [tempdir 'mlhdlc_so_viterbi'];  
clear mex;  
cd (mlhdlc_demo_dir);  
rmdir(mlhdlc_temp_dir, 's');
```

# Predefined System Objects Supported for HDL Code Generation

## In this section...

["Predefined System Objects in MATLAB Code" on page 1-50](#)

["Predefined System Objects in the MATLAB System Block" on page 1-51](#)

## Predefined System Objects in MATLAB Code

HDL Coder supports the following MATLAB System objects for HDL code generation:

- `hdl.RAM`
- `hdl.BlackBox`

HDL Coder supports the following Communications Toolbox System objects for HDL code generation:

- `comm.BPSKModulator`, `comm.BPSKDemodulator`
- `comm.PSKModulator`, `comm.PSKDemodulator`
- `comm.QPSKModulator`, `comm.QPSKDemodulator`
- `comm.ConvolutionalInterleaver`, `comm.ConvolutionalDeinterleaver`
- `comm.ViterbiDecoder`
- `comm.HDLCRCDetector`, `comm.HDLCRCGenerator`
- `comm.HDLRSDecoder`, `comm.HDLRSEncoder`

HDL Coder supports the following DSP System Toolbox System objects for HDL code generation:

- `dsp.Delay`
- `dsp.BiquadFilter`
- `dsp.DCBlocker`
- `dsp.HDLComplexToMagnitudeAngle`
- `dsp.HDLFIRRateConverter`
- `dsp.HDLFFT`, `dsp.HDLIFFT`
- `dsp.HDLChannelizer`
- `dsp.HDLNCO`
- `dsp.FIRFilter`
- `dsp.HDLFIRFilter`

HDL Coder supports the following Vision HDL Toolbox™ System objects for HDL code generation:

- `visionhdl.BilateralFilter`
- `visionhdl.BirdsEyeView`
- `visionhdl.ChromaResampler`
- `visionhdl.ColorSpaceConverter`
- `visionhdl.DemosaicInterpolator`

- `visionhdl.EdgeDetector`
- `visionhdl.GammaCorrector`
- `visionhdl.LookupTable`
- `visionhdl.Histogram`
- `visionhdl.ImageStatistics`
- `visionhdl.ROISelector`
- `visionhdl.LineBuffer`
- `visionhdl.PixelStreamAligner`
- `visionhdl.ImageFilter`
- `visionhdl.MedianFilter`
- `visionhdl.Closing`
- `visionhdl.Dilation`
- `visionhdl.Erosion`
- `visionhdl.Opening`
- `visionhdl.GrayscaleClosing`
- `visionhdl.GrayscaleDilation`
- `visionhdl.GrayscaleErosion`
- `visionhdl.GrayscaleOpening`

## Predefined System Objects in the MATLAB System Block

A subset of these predefined System objects are supported for code generation when you use them in a MATLAB System block. To learn more, see “HDL Code Generation” on the MATLAB System page.

## Load constants from a MAT-File

You can load compile-time constants from a MAT-file with the `coder.load` function in your MATLAB design.

For example, you can create a MAT-file, `sinvals.mat`, that contains fixed-point values of `sin` by entering the following commands in MATLAB:

```
sinvals = sin(fi(-pi:0.1:pi, 1, 16,15));
save sinvals.mat sinvals;
```

You can then generate HDL code from the following MATLAB code, which loads the constants from `sinvals.mat` into a persistent variable, `pConstStruct`, and assigns the values to a variable that is not persistent, `sv`.

```
persistent pConstStruct;
if isempty(pConstStruct)
    pConstStruct = coder.load('sinvals.mat');
end
sv = pConstStruct.sinvals;
```

### See Also

`codegen` | `coder.HdlConfig`

### More About

- “Functions Supported for HDL Code Generation” on page 1-2
- “Complex Data Type Support” on page 1-11
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

# Generate Code for User-Defined System Objects

## In this section...

["How To Create A User-Defined System object" on page 1-53](#)

["User-Defined System object Example" on page 1-53](#)

## How To Create A User-Defined System object

To create a user-defined System object and generate code:

- 1 Create a class that subclasses from `matlab.System`.
- 2 Define one of the following sets of methods:

- `setupImpl` and `stepImpl`
- `setupImpl`, `outputImpl`, and `updateImpl`

To use the `outputImpl` and `updateImpl` methods, your System object must also inherit from the `matlab.system.mixin.Nondirect` class.

- 3 Optionally, if your System object has private state properties, define the `resetImpl` method to initialize them to zero.
- 4 Write a top-level design function that creates an instance of your System object and calls the `step` method, or the `output` and `update` methods.

---

**Note** The `resetImpl` method runs automatically during System object initialization. For HDL code generation, you cannot call the public `reset` method.

- 5 Write a test bench function that exercises the top-level design function.
- 6 Generate HDL code.

## User-Defined System object Example

This example shows how to generate HDL code for a user-defined System object that implements the `setupImpl` and `stepImpl` methods.

- 1 In a writable folder, create a System object, `CounterSysObj`, which subclasses from `matlab.System`. Save the code as `CounterSysObj.m`.

```
classdef CounterSysObj < matlab.System

    properties (Nontunable)
        Threshold = int32(1)
    end
    properties (Access=private)
        State
        Count
    end
    methods
        function obj = CounterSysObj(varargin)
            setProperties(obj,nargin,varargin{:});
        end
    end
end
```

```
end

methods (Access=protected)
    function setupImpl(obj, ~)
        % Initialize states
        obj.Count = int32(0);
        obj.State = int32(0);
    end
    function y = stepImpl(obj, u)
        if obj.Threshold > u(1)
            obj.Count(:) = obj.Count + int32(1); % Increment count
        end
        y = obj.State; % Delay output
        obj.State = obj.Count; % Put new value in state
    end
end
end
```

The `stepImpl` method implements the System object functionality. The `setupImpl` method defines the initial values for the persistent variables in the System object.

- 2 Write a function that uses this System object and save it as `myDesign.m`. This function is your DUT.

```
function y = myDesign(u)

persistent obj
if isempty(obj)
    obj = CounterSysObj('Threshold',5);
end

y = step(obj, u);

end
```

- 3 Write a test bench that calls the DUT function and save it as `myDesign_tb.m`.

```
clear myDesign
for ii=1:10
    y = myDesign(int32(ii));
end
```

- 4 Generate HDL code for the DUT function as you would for any other MATLAB code, but skip fixed-point conversion.

## See Also

## More About

- “HDL Code Generation for System Objects” on page 1-14

## Map Matrices to ROM

To map a matrix constant to ROM:

- Read one matrix element at a time.
- The matrix size must be greater than or equal to the **RAM Mapping Threshold** value.

To learn how to set the RAM mapping threshold in Simulink, see the **RAM mapping threshold (bits)** section in “RAM Mapping Parameters” on page 15-7. To learn how to set the RAM mapping threshold in MATLAB, see “How To Enable RAM Mapping” on page 8-8.

- Read accesses to the matrix must not be within a feedback loop.

If your MATLAB code meets these requirements, HDL Coder inserts a no-reset register at the output of the matrix in the generated code. Many synthesis tools infer a ROM from this code pattern.

# Model State with Persistent Variables and System Objects

This example shows how to use persistent variables and System objects to model state and delays in a MATLAB® design for HDL code generation.

## Introduction

Using System objects to model delay results in concise generated code.

In MATLAB, multiple calls to a function having persistent variables do not result in multiple delays. Instead, the state in the function gets updated multiple times.

```
% In order to reuse code implemented in a function with states,  
% you need to duplicate functions multiple times to create multiple  
% instances of the algorithm with delay.
```

## Examine the MATLAB Code

Let us take a quick look at the implementation of the Sobel algorithm.

Examine the design to see how the delays and line buffers are modeled using:

- Persistent variables: mlhdlc\_sobel
- System objects: mlhdlc\_sysobj\_sobel

Notice that the 'filterdelay' function is duplicated with different function names in 'mlhdlc\_sobel' code to instantiate multiple versions of the algorithm in MATLAB for HDL code generation.

The delay line implementation is more complicated when done using MATLAB persistent variables.

Now examine the simplified implementation of the same algorithm using System objects in 'mlhdlc\_sysobj\_sobel'.

When used within the constraints of HDL code generation, the `dsp.Delay` objects always map to registers. For persistent variables to be inferred as registers, you have to be careful to read the variable before writing to it to map it to a register.

## MATLAB Design

```
demo_files = {...  
    'mlhdlc_sysobj_sobel', ...  
    'mlhdlc_sysobj_sobel_tb', ...  
    'mlhdlc_sobel', ...  
    'mlhdlc_sobel_tb'  
};
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderden'  
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];  
  
% create a temporary folder and copy the MATLAB files  
cd(tempdir);  
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
```

```

mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

for ii=1:numel(demo_files)
    copyfile(fullfile(mlhdlc_demo_dir, [demo_files{ii}, '.m*']), mlhdlc_temp_dir);
end

```

### Known Limitations

For predefined System Objects, HDL Coder™ only supports the 'step' method and does not support 'output' and 'update' methods.

With support for only the step method, delays cannot be used in modeling feedback paths. For example, the following piece of MATLAB code cannot be supported using the `dsp.Delay` System object.

```

%#codegen
function y = accumulate(u)
persistent p;
if isempty(p)
    p = 0;
end
y = p;
p = p + u;

```

### Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new mlhdlc_sobel
```

Next, add the file 'mlhdlc\_sobel.m' to the project as the MATLAB Function and 'mlhdlc\_sobel\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the hyperlinks in the Code Generation Log window.

Now, create a new project for the system object design:

```
coder -hdlcoder -new mlhdlc_sysobj_sobel
```

Add the file 'mlhdlc\_sysobj\_sobel.m' to the project as the MATLAB Function and 'mlhdlc\_sysobj\_sobel\_tb.m' as the MATLAB Test Bench.

Repeat the code generation steps and examine the generated fixed-point MATLAB and HDL code.

### Additional Notes:

You can model integer delay using `dsp.Delay` object by setting the 'Length' property to be greater than 1. These delay objects will be mapped to shift registers in the generated code.

If the optimization option 'Map persistent array variables to RAMs' is enabled, delay System objects will get mapped to block RAMs under the following conditions:

- 'InitialConditions' property of the `dsp.Delay` is set to zero.
- Delay input data type is not floating-point.
- RAMSize (DelayLength \* InputWordLength) is greater than or equal to the 'RAM Mapping Threshold'.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_delay_modeling'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Bitwise Operations in MATLAB for HDL Code Generation

HDL Coder supports bit shift, bit rotate, bit slice operations that mimic HDL-specific operators without saturation and rounding logic.

## Bit Shifting and Rotation

The following code implements a barrel shifter/rotator that performs a selected operation (based on the mode argument) on a fixed-point input operand.

```
function y = fcn(u, mode)
% Multi Function Barrel Shifter/Rotator

% fixed width shift operation
fixed_width = uint8(3);

switch mode
    case 1
        % shift left logical
        y = bitsll(u, fixed_width);
    case 2
        % shift right logical
        y = bitsrl(u, fixed_width);
    case 3
        % shift right arithmetic
        y = bitsra(u, fixed_width);
    case 4
        % rotate left
        y = bitrol(u, fixed_width);
    case 5
        % rotate right
        y = bitror(u, fixed_width);
    otherwise
        % do nothing
        y = u;
end
```

This table shows the generated VHDL and Verilog code.

Generated VHDL code	Generated Verilog code
<p>In VHDL code generated for this function, the shift and rotate functions map directly to shift and rotate instructions in VHDL.</p> <pre> CASE mode IS     WHEN "00000001" =&gt;         -- shift left logical         --&lt;S2&gt;:1:8'         cr := signed(u) sll 3;         y &lt;= std_logic_vector(cr);     WHEN "00000010" =&gt;         -- shift right logical         --&lt;S2&gt;:1:11'         b_cr := signed(u) srl 3;         y &lt;= std_logic_vector(b_cr);     WHEN "00000011" =&gt;         -- shift right arithmetic         --&lt;S2&gt;:1:14'         c_cr := SHIFT_RIGHT(signed(u) , 3);         y &lt;= std_logic_vector(c_cr);     WHEN "00000100" =&gt;         -- rotate left         --&lt;S2&gt;:1:17'         d_cr := signed(u) rol 3;         y &lt;= std_logic_vector(d_cr);     WHEN "00000101" =&gt;         -- rotate right         --&lt;S2&gt;:1:20'         e_cr := signed(u) ror 3;         y &lt;= std_logic_vector(e_cr);     WHEN OTHERS =&gt;         -- do nothing         --&lt;S2&gt;:1:23'         y &lt;= u; END CASE; </pre>	<p>The corresponding Verilog code is similar, except that Verilog does not have native operators for rotate instructions.</p> <pre> case ( mode )     1 :         begin             // shift left logical             //&lt;S2&gt;:1:8'             cr = u &lt;&lt; 3;             y = cr;         end     2 :         begin             // shift right logical             //&lt;S2&gt;:1:11'             b_cr = u &gt;&gt; 3;             y = b_cr;         end     3 :         begin             // shift right arithmetic             //&lt;S2&gt;:1:14'             c_cr = u &gt;&gt;&gt; 3;             y = c_cr;         end     4 :         begin             // rotate left             //&lt;S2&gt;:1:17'             d_cr = {u[12:0], u[15:13]};             y = d_cr;         end     5 :         begin             // rotate right             //&lt;S2&gt;:1:20'             e_cr = {u[2:0], u[15:3]};             y = e_cr;         end     default :         begin             // do nothing             //&lt;S2&gt;:1:23'             y = u;         end endcase </pre>

## Bit Slicing and Bit Concatenation

The `bitsliceget` and `bitconcat` functions map directly to slice and concatenate operators in both VHDL and Verilog.

You can use the functions `bitsliceget` and `bitconcat` to access and manipulate bit slices (fields) in a fixed-point or integer word. As an example, consider the operation of swapping the upper and

lower 4-bit nibbles of an 8-bit byte. The following example accomplishes this task without resorting to traditional mask-and-shift techniques.

```
function y = fcn(u)
% NIBBLE SWAP
y = bitconcat( ...
    bitsliceget(u, 4, 1),
    bitsliceget(u, 8, 5));
```

This table shows the generated VHDL and Verilog code.

Generated VHDL code	Generated Verilog code
<p>The following listing shows the corresponding generated VHDL code.</p> <pre>ENTITY fcn IS   PORT (     clk : IN std_logic;     clk_enable : IN std_logic;     reset : IN std_logic;     u : IN std_logic_vector(7 DOWNTO 0);     y : OUT std_logic_vector(7 DOWNTO 0));   END nibble_swap_7b;  ARCHITECTURE fsm_SFHDl OF fcn IS  BEGIN   -- NIBBLE SWAP   y &lt;= u(3 DOWNTO 0) &amp; u(7 DOWNTO 4); END fsm_SFHDl;</pre>	<p>The following listing shows the corresponding generated Verilog code.</p> <pre>module fcn (clk, clk_enable, reset, u, y );   input clk;   input clk_enable;   input reset;   input [7:0] u;   output [7:0] y;   // NIBBLE SWAP   assign y = {u[3:0], u[7:4]};  endmodule</pre>

## See Also

[codegen](#) | [coder.HdlConfig](#)

## More About

- “Functions for Programming and Data Types”
- “Fixed-Point Function Limitations” on page 1-2
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

# Guidelines for Writing MATLAB Code to Generate Efficient HDL Code

## MATLAB Design Requirements for HDL Code Generation

When you generate HDL code from your MATLAB design, you are converting an algorithm into an architecture that must meet hardware area and speed requirements.

Your MATLAB design has the following requirements:

- MATLAB code within the design must be supported for HDL code generation.
- Inputs and outputs must not be matrices or structures.

If you are generating code from the command line, verify your code readiness for code generation with the following command:

```
coder.screener('design_function_name')
```

If you use the HDL Workflow Advisor to generate code, this check runs automatically.

For a MATLAB language support reference, including supported functions from the Fixed-Point Designer, see “Functions Supported for HDL Code Generation” on page 1-2.

## Guidelines for Writing MATLAB code

For better HDL code and faster code generation, design your MATLAB code according to the following best practices:

- Serialize your input and output data. Parallel data processing structures require more hardware resources and a higher pin count.
- Use add and subtract algorithms instead of algorithms that use functions like sine, divide, and modulo. Add and subtract operations use fewer hardware resources.
- Avoid large arrays and matrices. Large arrays and matrices require more registers and RAM for storage.
- Convert your code from floating-point to fixed-point. Floating-point data types are inefficient for hardware realization. HDL Coder provides an automated workflow for floating-point to fixed-point conversion.
- Unroll loops to increase speed at the cost of higher area; unroll fewer loops and enable the loop streaming optimization to conserve area at the cost of lower throughput.

## See Also

**Apps**  
HDL Coder

**Classes**  
`coder.HdlConfig` | `coder.hdl.loopspec` | `coder.hdl.pipeline`

## More About

- “Optimize MATLAB Loops” on page 8-20
- “For-Loop Best Practices for HDL Code Generation” on page 1-64
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

## For-Loop Best Practices for HDL Code Generation

When you generate HDL code from your MATLAB design, you are converting an algorithm into an architecture that must meet hardware area and speed requirements. Some best practices for using loops in MATLAB code for HDL code generation are:

- Use monotonically increasing loop counters, with increments of 1, to minimize the amount of hardware generated in the HDL code.
- If you want to use the loop streaming optimization:
  - When assigning new values to persistent variables inside a loop, do not use other persistent variables on the right side of the assignment. Instead, use an intermediate variable.
  - If a loop modifies any elements in a persistent array, the loop should modify all of the elements in the persistent array.

### Monotonically Increasing Loop Counters

By using monotonically increasing loop counters with increments of 1, you can reduce the amount of hardware in the generated HDL code. The following loop is an example of a monotonically increasing loop counter with increments of 1.

```
a=1;  
for i=1:10  
    a=a+1;  
end
```

If a loop counter increases by an increment other than 1, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;  
for i=1:2:10  
    a=a+1;  
end
```

If a loop counter decreases, the generated HDL code can require additional adders. Due to this additional hardware, do not use the following type of loop.

```
a=1;  
for i=10:-1:1  
    a=a+1;  
end
```

### Persistent Variables in Loops

If a loop contains multiple persistent variables, when you assign values to persistent variables, use intermediate variables that are not persistent on the right side of the assignment. This practice makes dependencies clear to the compiler and assists internal optimizations during the HDL code generation process. If you want to use the loop streaming optimization to reduce the amount of generated hardware, this practice is recommended.

In the following example, `var1` and `var2` are persistent variables. `var1` is used on the right side of the assignment. Because a persistent variable is on the right side of an assignment, do not use this type of loop:

```
for i=1:10
    var1 = 1 + i;
    var2 = var1 * 2;
end
```

Instead of using `var1` on the right side of the assignment, use an intermediate variable that is not persistent. This example demonstrates this with the intermediate variable `var_intermediate`.

```
for i=1:10
    var_intermediate = 1 + i;
    var1 = var_intermediate;
    var2 = var_intermediate * 2;
end
```

## Persistent Arrays in Loops

If a loop modifies elements in a persistent array, make sure that the loop modifies all of the elements in the persistent array. If all elements of the persistent array are not modified within the loop, HDL Coder cannot perform the loop streaming optimization.

In the following example, `a` is a persistent array. The first element is modified outside of the loop. Do not use this type of loop.

```
for i=2:10
    a(i)=1+i;
end
a(1)=24;
```

Rather than modifying the first element outside the loop, modify all of the elements inside the loop.

```
for i=1:10
    if i==1
        a(i)=24;
    else
        a(i)=1+i;
    end
end
```

## See Also

**Apps**  
**HDL Coder**

**Classes**  
`coder.HdlConfig` | `coder.hdl.loopspec` | `coder.hdl.pipeline`

## More About

- “Optimize MATLAB Loops” on page 8-20
- “Guidelines for Writing MATLAB Code to Generate Efficient HDL Code” on page 1-62
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

# MATLAB Test Bench Requirements and Best Practices for HDL Code Generation

## What Is a MATLAB Test Bench?

A test bench is a MATLAB script or function that you write to test the algorithm in your MATLAB design function. The test bench varies the input data to the design to simulate real world conditions. It can also check that the output data meets design specifications.

HDL Coder uses the data it gathers from running your test bench with your design to infer fixed-point data types for floating-point to fixed-point conversion. The coder also uses the data to generate HDL test data for verifying your generated code. For more information on how to write your test bench for the best results, see “MATLAB Test Bench Requirements and Best Practices for HDL Code Generation” on page 1-66.

## MATLAB Test Bench Requirements

You can use any MATLAB data type and function in your test bench.

A MATLAB test bench has the following requirements:

- For floating-point to fixed-point conversion, the test bench must be a script or a function with no inputs.
- The inputs and outputs in your MATLAB design interface must use the same data types, sizes, and complexity in each call site in your test bench.
- If you enable the **Accelerate test bench for faster simulation** option in the Float-to-Fixed Workflow, the MATLAB constructs in your test bench loop must be compilable.

## MATLAB Test Bench Best Practices

Use the following MATLAB test bench best practices:

- *Design your test bench to cover the full numeric range of data that the design must handle.* HDL Coder uses the data that it accumulates from running the test bench to infer fixed-point data types during floating-point to fixed-point conversion.

If you call the design function multiple times from your test bench, the coder uses the accumulated data from each instance to infer fixed-point types. Both the design and the test bench can call local functions within the file or other functions on the MATLAB path. The call to the design function can be at any level of your test bench hierarchy.

- *Before trying to generate code, run your test bench in MATLAB.* If simulation is slow, accelerate your test bench. To learn how to accelerate your simulation, see “Accelerate MATLAB Algorithms”.
- If you have a loop that calls your design function, use only compilable MATLAB constructs within the loop and enable the **Accelerate test bench for faster simulation** option.
- Before each test bench simulation run, use the `clear variables` command to reset your persistent variables.

To see an example of a test bench, enter this command:

```
showdemo mlhdlc_tutorial_float2fixed_files
```

## See Also

### Apps

### HDL Coder

### Classes

`coder.HdlConfig` | `coder.hdl.loopspec` | `coder.hdl.pipeline`

## More About

- “Guidelines for Writing MATLAB Code to Generate Efficient HDL Code” on page 1-62
- “For-Loop Best Practices for HDL Code Generation” on page 1-64
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4



# MATLAB to HDL Examples for Communications and Signal Processing Applications

---

- “HDL Code Generation for LMS Filter” on page 2-2
- “Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number” on page 2-9
- “Timing Offset Estimation” on page 2-14
- “Data Packetization” on page 2-18
- “Transmit and Receive FIFO” on page 2-25
- “HDL Code Generation for Harris Corner Detection Algorithm” on page 2-32
- “HDL Code Generation for Adaptive Median Filter” on page 2-39
- “Contrast Adjustment” on page 2-47
- “Image Enhancement by Histogram Equalization” on page 2-56
- “HDL Code Generation for Image Format Conversion from RGB to YUV” on page 2-58
- “High Dynamic Range Imaging” on page 2-63
- “Accelerate a Pixel-Streaming Design Using MATLAB Coder” on page 2-68
- “Enhanced Edge Detection from Noisy Color Video” on page 2-71
- “Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow” on page 2-74

## HDL Code Generation for LMS Filter

This example shows how to generate HDL code from a MATLAB® design that implements an LMS filter. The example also illustrates how to design a test bench that cancels out the noise signal by using this filter.

### LMS Filter MATLAB Design

The MATLAB design used in the example is an implementation of an LMS (Least Mean Squares) filter. The LMS filter is a class of adaptive filter that identifies an FIR filter signal that is embedded in the noise. The LMS filter design implementation in MATLAB consists of a top-level function `mlhdlc_lms_fcn` that calculates the optimal filter coefficients to reduce the difference between the output signal and the desired signal.

```
design_name = 'mlhdlc_lms_fcn';
testbench_name = 'mlhdlc_lms_noise_canceler_tb';
```

Review the MATLAB design:

```
open(design_name);

%%%%%%%%%%%%%
% MATLAB Design: Adaptive Noise Canceler algorithm using Least Mean Square
% (LMS) filter implemented in MATLAB
%
% Key Design pattern covered in this example:
% (1) Use of function calls
% (2) Function inlining vs instantiation knobs available in the coder
% (3) Use of system objects in the testbench to stream test vectors into the design
%%%%%%%%%%%%%

##codegen
function [filtered_signal, y, fc] = mlhdlc_lms_fcn(input, ...
                                                     desired, step_size, reset_weights)
% 'input' : The signal from Exterior Mic which records the ambient noise.
% 'desired': The signal from Pilot's Mic which includes
%             original music signal and the noise signal
% 'err_sig': The difference between the 'desired' and the filtered 'input'
%             It represents the estimated music signal (output of this block)
%
% The LMS filter is trying to retrieve the original music signal('err_sig')
% from Pilot's Mic by filtering the Exterior Mic's signal and using it to
% cancel the noise in Pilot's Mic. The coefficients/weights of the filter
% are updated(adapted) in real-time based on 'input' and 'err_sig'.

% register filter coefficients
persistent filter_coeff;
if isempty(filter_coeff)
    filter_coeff = zeros(1, 40);
end

% Variable Filter: Call 'tapped_delay_fcn' function on path to create
% 40-step tapped delay
delayed_signal = mtapped_delay_fcn(input);

% Apply filter coefficients
```

```

weight_applied = delayed_signal .* filter_coeff;

% Call treesum function on matlab path to sum up the results
filtered_signal = mtreesum_fcn(weight_applied);

% Output estimated Original Signal
td = desired;
tf = filtered_signal;
esig = td - tf;
y = esig;

% Update Weights: Call 'update_weight_fcn' function on MATLAB path to
% calculate the new weights
updated_weight = update_weight_fcn(step_size, esig, delayed_signal, ...
                                    filter_coeff, reset_weights);

% update filter coefficients register
filter_coeff = updated_weight;
fc = filter_coeff;

function y = mtreesum_fcn(u)
%Implement the 'sum' function without a for-loop
% y = sum(u);

% The loop based implementation of 'sum' function is not ideal for
% HDL generation and results in a longer critical path.
% A tree is more efficient as it results in
% delay of log2(N) instead of a delay of N delay

% This implementation shows how to explicitly implement the vector sum in
% a tree shape to enable hardware optimizations.

% The ideal way to code this generically for any length of 'u' is to use
% recursion but it is not currently supported by MATLAB Coder

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

% This implementation is hardwired for a 40tap filter.

level1 = vsum(u);
level2 = vsum(level1);
level3 = vsum(level2);
level4 = vsum(level3);
level5 = vsum(level4);
level6 = vsum(level5);
y = level6;

function output = vsum(input)

coder.inline('always');

vt = input(1:2:end);

for i = int32(1:numel(input)/2)
    k = int32(i*2);

```

```
    vt(i) = vt(i) + input(k);
end

output = vt;

function tap_delay = mtapped_delay_fcn(input)
% The Tapped Delay function delays its input by the specified number
% of sample periods, and outputs all the delayed versions in a vector
% form. The output includes current input

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

persistent u_d;
if isempty(u_d)
    u_d = zeros(1,40);
end

u_d = [u_d(2:40), input];

tap_delay = u_d;

function weights = update_weight_fcn(step_size, err_sig, ...
    delayed_signal, filter_coeff, reset_weights)
% This function updates the adaptive filter weights based on LMS algorithm

% Copyright 2007-2015 The MathWorks, Inc.

% NOTE: To instruct MATLAB Coder to compile an external function,
% add the following compilation directive or pragma to the function code
%#codegen

step_sig = step_size .* err_sig;
correction_factor = delayed_signal .* step_sig;
updated_weight = correction_factor + filter_coeff;

if reset_weights
    weights = zeros(1,40);
else
    weights = updated_weight;
end
```

The MATLAB function is modular and uses functions:

- `mtapped_delay_fcn` to calculate delayed versions of the input signal in vector form.
- `mtreesum_fcn` to calculate the sum of the applied weights in a tree structure. The individual sum is calculated by using a `vsum` function.
- `update_weight_fcn` to calculate the updated filter weights based on the least mean square algorithm.

### LMS Filter MATLAB Test Bench

Review the MATLAB test bench:

```

open(testbench_name)

% Returns an adaptive FIR filter System object,
% HLMS, that computes the filtered output, filter error and the filter
% weights for a given input and desired signal using the Least Mean
% Squares (LMS) algorithm.

% Copyright 2011-2019 The MathWorks, Inc.
clear('mlhdlc_lms_fcn');

hfilt2 = dsp.FIRFilter(...
    'Numerator', fir1(10, [.5, .75]));
rng('default'); % always default to known state
x = randn(1000,1); % Noise
d = step(hfilt2, x) + sin(0:.05:49.95)'; % Noise + Signal

stepSize = 0.01;
reset_weights =false;

hSrc = dsp.SignalSource(x);
hDesiredSrc = dsp.SignalSource(d);

hOut = dsp.SignalSink;
hErr = dsp.SignalSink;
%%%%%%%%%%%%%
%Call to the design
%%%%%%%%%%%%%
while (~isDone(hSrc))
    [y, e] = mlhdlc_lms_fcn(step(hSrc), step(hDesiredSrc), ...
        stepSize, reset_weights);
    step(hOut, y);
    step(hErr, e);
end

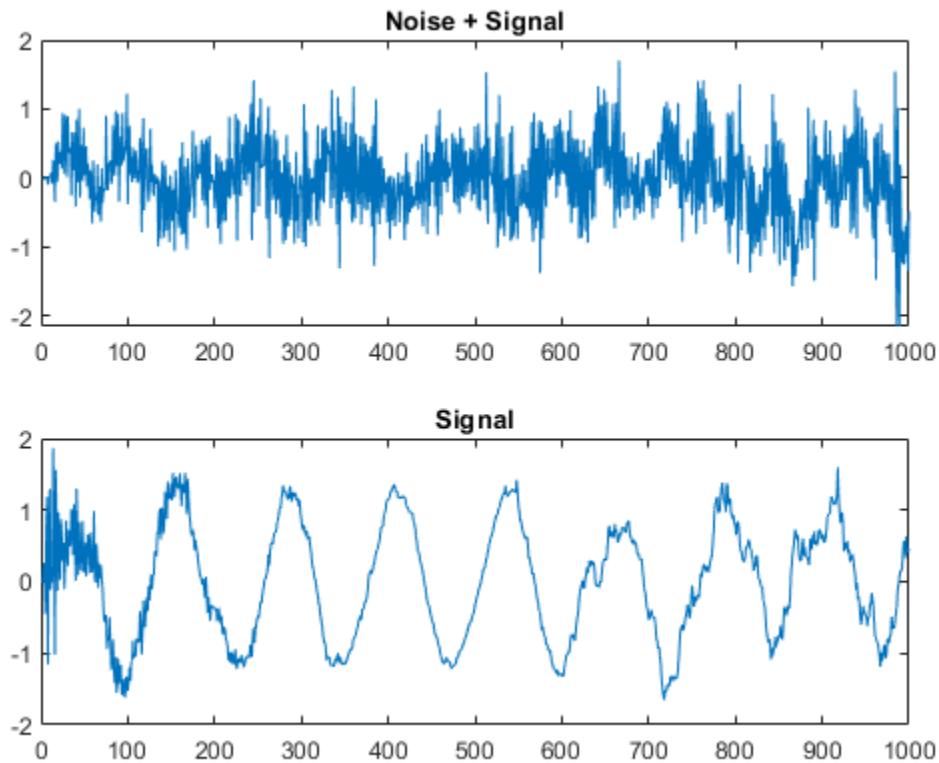
figure('Name', [mfilename, '_signal_plot']);
subplot(2,1,1), plot(hOut.Buffer), title('Noise + Signal');
subplot(2,1,2),plot(hErr.Buffer), title('Signal');

```

### Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_lms_noise_canceler_tb
```



### Create a Folder and Copy Relevant Files

Before you generate HDL code for the MATLAB design, copy the design and test bench files to a writeable folder. These commands copy the files to a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdesign');
mlhdlc_temp_dir = [tempdir 'mlhdlc_lms_nc'];
```

create a temporary folder and copy the MATLAB files.

```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Create an HDL Coder Project

To generate HDL code from a MATLAB design:

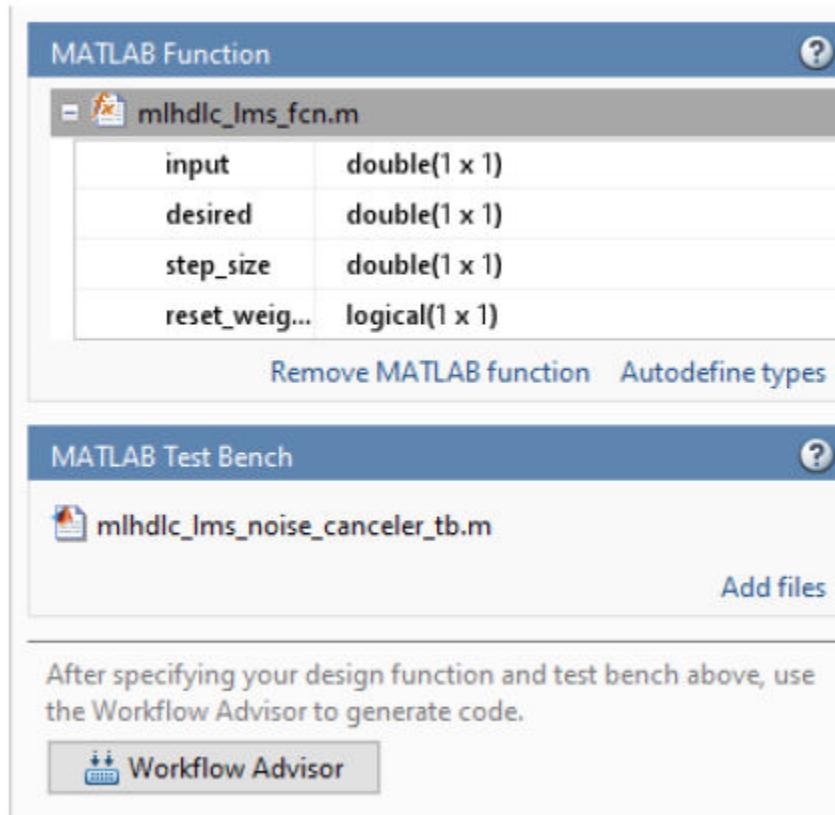
1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_lms_nc
```

2. Add the file `mlhdlc_lms_fcn.m` to the project as the **MATLAB Function** and `mlhdlc_lms_noise_canceler_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_lms_fcn`.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.



### Run Fixed-Point Conversion and HDL Code Generation

- 1 Click the **Workflow Advisor** button to start the Workflow Advisor.
- 2 Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_lms_fcn_FixPt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also “Generate Instantiable Code for Functions” on page 5-11.

### Clean Up Generated Files

To clean up the temporary project folder, run these commands:

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_lms_nc'];
clear mex;
```

```
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Bisection Algorithm to Calculate Square Root of an Unsigned Fixed-Point Number

This example shows how to generate HDL code from MATLAB® design implementing an bisection algorithm to calculate the square root of a number in fixed point notation.

Same implementation, originally using n-multipliers in HDL code, for wordlength n, under sharing and streaming optimizations, can generate HDL code with only 1 multiplier demonstrating the power of MATLAB® HDL Coder optimizations.

The design of the square-root algorithm shows the pipelining concepts to achieve a fast clock rate in resulting RTL design. Since this design is already in fixed point, you don't need to run fixed-point conversion.

## MATLAB Design

```
% Design Sqrt
design_name = 'mlhdlc_sqrt';

% Test Bench for Sqrt
testbench_name = 'mlhdlc_sqrt_tb';
```

Lets look at the Sqrt Design

```
dbtype(design_name)
```

```

1      %%%%%%
2      % MATLAB design: Pipelined Bisection Square root algorithm
3
4      % Introduction:
5
6      % Implement SQRT by the bisection algorithm in a pipeline, for unsigned fixed
7      % point numbers (also why you don't need to run fixed-point conversion for this design).
8      % The demo illustrates the usage of a pipelined implementation for numerical algorithms.
9
10     % Key Design pattern covered in this example:
11     % (1) State of the bisection algorithm is maintained with persistent variables
12     % (2) Stages of the bisection algorithm are implemented in a pipeline
13     % (3) Code is written in a parameterized fashion, i.e. word-length independent, to work for
14
15     % Ref. 1. R. W. Hamming, "Numerical Methods for Scientists and Engineers," 2nd, Ed, pp 67-
16     %       2. Bisection method, http://en.wikipedia.org/wiki/Bisection\_method, (accessed 02/18)
17
18
19     % Copyright 2013-2015 The MathWorks, Inc.
20
21     %#codegen
22     function [y,z] = mlhdlc_sqrt( x )
23         persistent sqrt_pipe
24         persistent in_pipe
25         if isempty(sqrt_pipe)
26             sqrt_pipe = fi(zeros(1,x.WordLength),numerictype(x));
27             in_pipe = fi(zeros(1,x.WordLength),numerictype(x));
28         end
29
```

```
30      % Extract the outputs from pipeline
31      y = sqrt_pipe(x.WordLength);
32      z = in_pipe(x.WordLength);
33
34      % for analysis purposes you can calculate the error between the fixed-point bisection ro
35      %Q = [double(y).^2, double(z)];
36      %[Q, diff(Q)]
37
38      % work the pipeline
39      for itr = x.WordLength-1:-1:1
40          % move pipeline forward
41          in_pipe(itr+1) = in_pipe(itr);
42          % guess the bits of the square-root solution from MSB to the LSB of word length
43          sqrt_pipe(itr+1) = guess_and_update( sqrt_pipe(itr), in_pipe(itr+1), itr );
44      end
45
46      %% Prime the pipeline
47      % with new input and the guess
48      in_pipe(1) = x;
49      sqrt_pipe(1) = guess_and_update( fi(0,numerictype(x)), x, 1 );
50
51      %% optionally print state of the pipeline
52      %disp('***** State of Pipeline *****')
53      %double([in_pipe; sqrt_pipe])
54
55      return
56  end
57
58 % Guess the bits of the square-root solution from MSB to the LSB in
59 % a binary search-fashion.
60 function update = guess_and_update( prev_guess, x, stage )
61     % Key step of the bisection algorithm is to set the bits
62     guess = bitset( prev_guess, x.WordLength - stage + 1);
63     % compare if the set bit is a candidate solution to retain or clear it
64     if ( guess*guess <= x )
65         update = guess;
66     else
67         update = prev_guess;
68     end
69     return
70 end
```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde
mlhdlc_temp_dir = [tempdir 'mlhdlc_sqrt'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

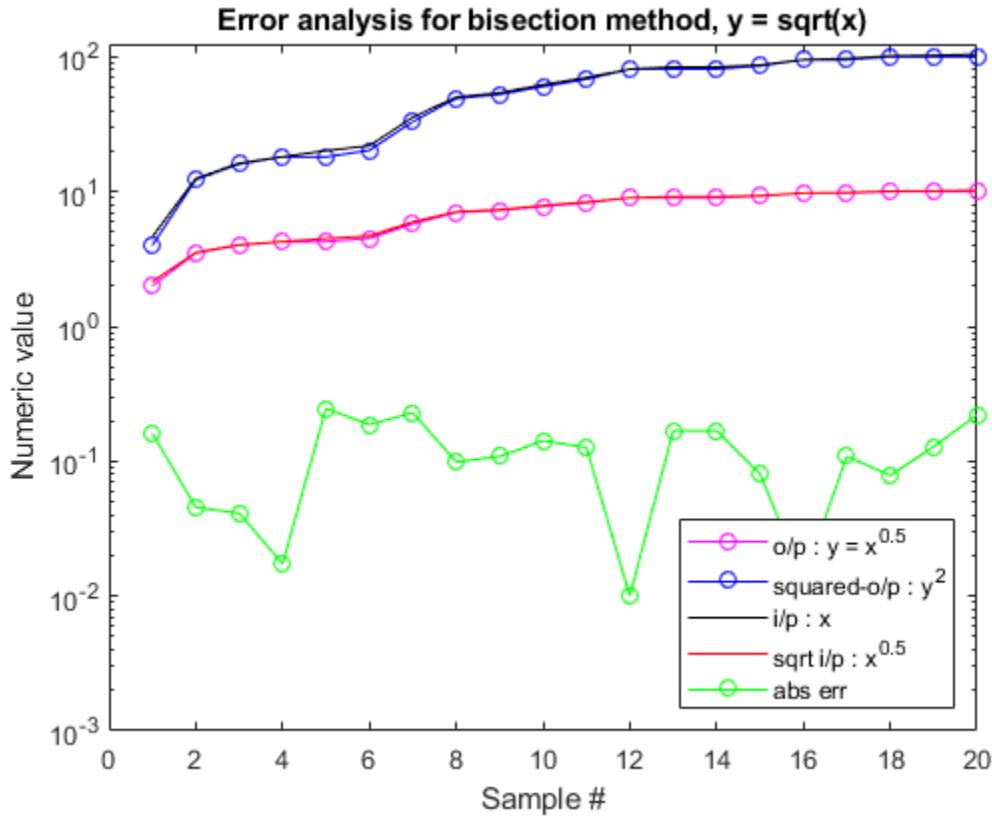
% copy files to the temp dir
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

`mlhdlc_sqrt_tb`

Iter	Input	Output	actual	abserror
Iter = 01	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 02	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 03	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 04	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 05	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 06	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 07	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 08	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 09	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 10	0.000	0000000000 (0.00)	0.000000	0.000000
Iter = 11	4.625	0000010000 (2.00)	2.150581	0.150581
Iter = 12	12.500	0000011100 (3.50)	3.535534	0.035534
Iter = 13	16.250	0000100000 (4.00)	4.031129	0.031129
Iter = 14	18.125	0000100010 (4.25)	4.257347	0.007347
Iter = 15	20.125	0000100010 (4.25)	4.486090	0.236090
Iter = 16	21.875	0000100100 (4.50)	4.677072	0.177072
Iter = 17	35.625	0000101110 (5.75)	5.968668	0.218668
Iter = 18	50.250	0000111000 (7.00)	7.088723	0.088723
Iter = 19	54.000	0000111010 (7.25)	7.348469	0.098469
Iter = 20	62.125	0000111110 (7.75)	7.881941	0.131941
Iter = 21	70.000	0001000010 (8.25)	8.366600	0.116600
Iter = 22	81.000	0001001000 (9.00)	9.000000	0.000000
Iter = 23	83.875	0001001000 (9.00)	9.158330	0.158330
Iter = 24	83.875	0001001000 (9.00)	9.158330	0.158330
Iter = 25	86.875	0001001010 (9.25)	9.320676	0.070676
Iter = 26	95.125	0001001110 (9.75)	9.753205	0.003205
Iter = 27	97.000	0001001110 (9.75)	9.848858	0.098858
Iter = 28	101.375	0001010000 (10.00)	10.068515	0.068515
Iter = 29	102.375	0001010000 (10.00)	10.118053	0.118053
Iter = 30	104.250	0001010000 (10.00)	10.210289	0.210289



### Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_sqrt_prj
```

Next, add the file 'mlhdlc\_sqrt.m' to the project as the MATLAB Function and 'mlhdlc\_sqrt\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run HDL Code Generation

This design is already in fixed point and suitable for HDL code generation. It is not desirable to run floating point to fixed point advisor on this design.

- 1 Launch Workflow Advisor
- 2 Under 'Define Input Types' Choose 'Keep original types' for the option 'Fixed-point conversion'
- 3 Under 'Optimizations' tab in 'RAM Mapping' box uncheck 'MAP persistent variables to RAMs'. We don't want the pipeline to be inferred as a RAM.
- 4 Optionally you may want to choose, under 'Optimizations' tab, 'Area Optimizations' and set 'Resource sharing factor' equal to wordlength (10 here), select 'Stream Loops' under the 'Loop Optimizations' tab. Also don't forget to check 'Distributed Pipelining' when you enable the optimizations.
- 5 Click on the 'Code Generation' step and click 'Run'

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

### Examine the Synthesis Results

- 1 Run the logic synthesis step with the following default options if you have ISE installed on your machine.
- 2 In the synthesis report, note the clock frequency reported by the synthesis tool without any optimization options enabled.
- 3 Typically **timing performance** of this design using Xilinx ISE synthesis tool for the 'Virtex7' chip family, device 'xc7v285t', speed grade -3, to be around **229MHz**, and a maximum combinatorial path delay: **0.406ns**.
- 4 Optimizations for this design (loop streaming and multiplier sharing) work to reduce resource usage, with a moderate trade-off on timing. For the particular word-length size in test bench you will see a reduction of **n** multipliers to **1**.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_sqrt'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Timing Offset Estimation

This example shows how to generate HDL code from a basic lead-lag timing offset estimation algorithm implemented in MATLAB® code.

### Introduction

In wireless communication systems, receive data is oversampled at the RF front end. This serves several purposes, including providing sufficient sampling rates for receive filtering.

However, one of the most important functions is to provide multiple sampling points on the received waveform such that data can be sampled near the maximum amplitude point in the received waveform. This example illustrates a basic lead-lag time offset estimation core, operating recursively.

The generated hardware core for this design operates at  $1/\text{os\_rate}$  where  $\text{os\_rate}$  is the oversampled rate. That is, for 8 oversampled clock cycles this core iterates once. The output is at the symbol rate.

```
design_name = 'mlhdlc_comms_toe';
testbench_name = 'mlhdlc_comms_toe_tb';
```

Let us take a look at the MATLAB® design.

```
type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Time Offset Estimation
%
%% Introduction:
%
% The generated hardware core for this design operates at 1/os_rate
% where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles
% this core iterates once. The output is at the symbol rate.
%
% Key design pattern covered in this example:
% (1) Data is sent in a vector format, stored in a register and accessed
% multiple times
% (2) The core also illustrates basic mathematical operations
%

% Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [tauh,q] = mlhdlc_comms_toe(r,mu)

persistent tau
persistent rBuf

os_rate = 8;
if isempty(tau)
    tau = 0;
    rBuf = zeros(1,3*os_rate);
end

rBuf = [rBuf(1+os_rate:end) r];
taur = round(tau);
```

```
% Determine lead/lag values and compute offset error
zl = rBuf(os_rate+taur-1);
zo = rBuf(os_rate+taur);
ze = rBuf(os_rate+taur+1);
offsetError = zo*(ze-zl);

% update tau
tau = tau + mu*offsetError;

tauh = tau;

q = zo;

type(testbench_name);

function mlhdlc_comms_toe_tb
%
% Copyright 2011-2015 The MathWorks, Inc.

os_rate = 8;
Ns = 128;
SNR = 100;
mu = .5; % smoothing factor for time offset estimates

% create simulated signal
rng('default'); % always default to known state
b = round(rand(1,Ns));
d = reshape(repmat(b*2-1,os_rate,1),1,Ns*os_rate);

x = [zeros(1,Ns*os_rate) d zeros(1,Ns*os_rate)];
y = awgn(x,SNR);

w = fir1(3*os_rate+1,1/os_rate)';
z = filter(w,1,y);
r = z(4:end); % give it an offset to make things interesting

%tau = 0;
Nsym = floor(length(r)/os_rate);
tauh = zeros(1,Nsym-1); q = zeros(1,Nsym-1);
for il = 1:Nsym
    rVec = r(1+(il-1)*os_rate:il*os_rate);

    % Call to the Timing Offset Estimation Algorithm
    [tauh(il),q(il)] = mlhdlc_comms_toe(rVec,mu);
end

indexes = 1:os_rate:length(tauh)*os_rate;
indexes = indexes+tauh+os_rate-1-os_rate*2;

Fig1Loc=figposition([5 50 90 40]);
H_f1=figure(1); clf;
set(H_f1,'position',Fig1Loc);
subplot(2,1,1)
plot(r,'b');
hold on
```

```
plot(indexes,q,'ro');
axis([indexes(1) indexes(end) -1.5 1.5]);
title('Received Signal with Time Correct Detections');
subplot(2,1,2)
plot(tauh);
title('Estimate of Time Offset');

function y=figposition(x)
%FIGPOSITION Positions figure window irrespective of the screen resolution
% Y=FIGPOSITION(X) generates a vector the size of X.
% This specifies the location of the figure window in pixels
%
screenRes=get(0,'ScreenSize');
% Convert x to pixels
y(1,1)=(x(1,1)*screenRes(1,3))/100;
y(1,2)=(x(1,2)*screenRes(1,4))/100;
y(1,3)=(x(1,3)*screenRes(1,3))/100;
y(1,4)=(x(1,4)*screenRes(1,4))/100;
```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder'
mlhdlc_temp_dir = [tempdir 'mlhdlc_toe'];

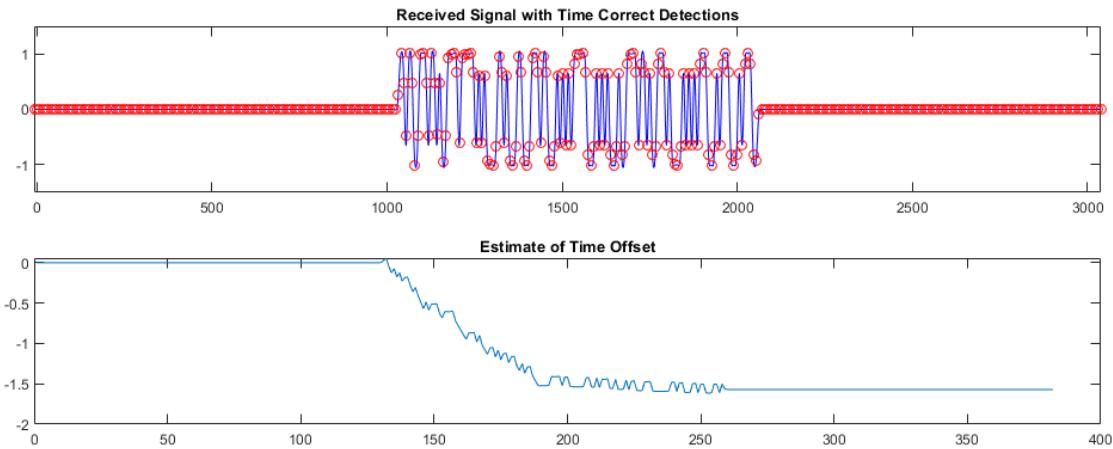
% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_comms_toe_tb
```



### Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_toe
```

Next, add the file 'mlhdlc\_comms\_toe.m' to the project as the MATLAB Function and 'mlhdlc\_comms\_toe\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Build tab and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_toe'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Data Packetization

This example shows how to generate HDL code from a MATLAB® design that packetizes a transmit sequence.

### Introduction

In wireless communication systems receive data is oversampled at the RF front end. This serves several purposes, including providing sufficient sampling rates for receive filtering.

```
% However, one of the most important
% functions is to provide multiple sampling points on the received
% waveform such that data can be sampled near the maximum amplitude point
% in the received waveform. This example illustrates a basic lead-lag time
% offset estimation core, operating recursively.

% The generated hardware core for this design operates at 1/os_rate
% where os_rate is the oversampled rate. That is, for 8 oversampled clock cycles
% this core iterates once. The output is at the symbol rate.

design_name = 'mlhdlc_comms_data_packet';
testbench_name = 'mlhdlc_comms_data_packet_tb';

Let us take a look at the MATLAB design.

type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Data packetization
%
% Introduction:
%
% This core is meant to illustrate packetization of a transmit sequence.
% There is a "pad" data section, which allows for the transmit amplifier to
% settle. This is then followed by a 65-bit training sequence. This is
% followed by the number of symbols beginning encoded into two bytes or
% 16-bits. This is then followed by a variable length data sequence and a
% CRC. All bits can optionally be differentially encoded.
%
% Key design pattern covered in this example:
% (1) Design illustrates the use of binary operands, such as bitxor
% (2) Shows how to properly segment persistent variables for register and
% BRAM access
% (3) Illustrates the use of fi math
% (4) Shows how to properly format and store ROM data, e.g., padData

% Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [symbolOut, reByte] = ...
    mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad)

persistent trainBits1 padData
persistent valueCRC crcVector bitPrev
persistent inPacketFlag bit0fByteIndex symbolCount
```

```

fm = hdlfimath;
if isempty(symbolCount)
    symbolCount = 1;
    inPacketFlag = 0;
    valueCRC = fi(1, 0, 16, 0, fm);
    bit0fByteIndex = 1;
    bitPrev = fi(1, 0, 1, 0, fm);
    crcVector = zeros(1, 16);
end
if isempty(trainBits1)
    % data-set already exists
    trainBits1 = TRAIN_DATA;
    padData = PAD_DATA;
end

%genPoly = 69665;
genPoly = fi(65535, 0, 16, 0, fm);
byteUint8 = uint8(byteValue);

reByte = 0;
symbolOut = fi(0, 0, 1, 0, fm);

%the first condition is whether or not we're currently processing a packet
if inPacketFlag == 1
    bitOut = fi(0, 0, 1, 0, fm);
    if symbolCount <= Npad
        bitOut(:) = padData(symbolCount);
    elseif symbolCount <= Npad+Nts
        bitOut(:) = trainBits1(symbolCount-Npad);
    elseif symbolCount <= Npad+Nts+numberSymbols
        bitOut(:) = bitget(byteUint8, 9-bit0fByteIndex);
        bit0fByteIndex = bit0fByteIndex + 1;
        if bit0fByteIndex == 9 && symbolCount < Npad+Nts+numberSymbols
            bit0fByteIndex = 1;
            reByte = 1; % we've exhausted this one so pop new one off
        end
    elseif symbolCount <= Npad+Nts+numberSymbols+16
        bitOut(:) = 0;
    elseif symbolCount <= Npad+Nts+numberSymbols+32
        bitOut(:) = crcVector(symbolCount-(Npad+Nts+numberSymbols+16));
    else
        inPacketFlag = 0; %we're done
    end

%leadValue = 0;
% here we have the bit going out so if past Nts+Npad then form CRC.
% Note that we throw 16 zeros on the end in order to flush the CRC
if symbolCount > Npad+Nts && symbolCount <= Npad+Nts+numberSymbols+16

    valueCRCsh1 = bitsll(valueCRC, 1);
    valueCRCadd1 = bitor(valueCRCsh1, fi(bitOut, 0, 16, 0, fm));
    leadValue = bitget(valueCRCadd1, 16);
    if leadValue == 1
        valueCRCxor = bitxor(valueCRCadd1, genPoly);
    else
        valueCRCxor = valueCRCadd1;
    end
    valueCRC = valueCRCxor;

```

```
if symbolCount == Npad+Nts+numberSymbols+16
    crcVector(:) = bitget( valueCRC, 16:-1:1);
end

if diffOn == 0 || symbolCount <= Npad+Nts
    symbolOut(:) = bitOut;
else
    if bitPrev == bitOut
        symbolOut(:) = 1;
    else
        symbolOut(:) = 0;
    end
end
bitPrev(:) = symbolOut;

symbolCount = symbolCount + 1; %total number of symbols transmitted
else
    % we're not processing a packet and waiting for a new packet to arrive
    if emptyFlag == 0
        % reset everything
        inPacketFlag = 1;
        % toggle re to grab data
        reByte = 1;
        symbolCount = 1;
        bit0fByteIndex = 1;
        valueCRC(:) = 65535;
        bitPrev(:) = 0;
    end
end
end

type(testbench_name);

function mlhdlc_comms_data_packet_tb
%
% Copyright 2011-2015 The MathWorks, Inc.

% generate transmit data, note the first two bytes are the data length
numberBytes = 8; % this is total number of symbols
numberSymbols = numberBytes*8;
rng(1); % always default to known state
data = [floor(numberBytes/2^8) mod(numberBytes,2^8) ...
    round(rand(1,numberBytes-2)*255)];

% generate training data helper function
make_train_data('TRAIN_DATA');

% make sure training data is generated
pause(2)
[~] = which('TRAIN_DATA');

trainBits1 = TRAIN_DATA;
Nts = length(trainBits1);
```

```

make_pad_data('PAD_DATA');
pause(2)
[~] = which('PAD_DATA');
Npad = 2^9;

% Give number of samples, where the start of the sequence flag will be
% (indicated by a zero), as well as an output buffer for generated symbols
Nsamp = 1000;
Noffset = 20;
emptyFlagHold = ones(1,Nsamp); emptyFlagHold(Noffset) = 0;
symbolOutHold = zeros(1,Nsamp);

dataIndex = 1;
byteValue = 0;
diffOn = 1; % 0 - regular encoding, 1 - differential encoding
for i1 = 1:Nsamp
    emptyFlag = emptyFlagHold(i1);

    %%%%%%
    % Call to the design
    %%%%%%
    [symbolOut, reByte] = ...
        mlhdlc_comms_data_packet(emptyFlag, byteValue, numberSymbols, diffOn, Nts, Npad);

    % This set of code emulates the external FIFO interface
    if reByte == 1 % when high, pop a value off the input FIFO
        byteValue = data(dataIndex);
        dataIndex = dataIndex + 1;
    end
    symbolOutHold(i1) = symbolOut;
end

%%%%%%%%%%%%%
% This is all code to verify we did the encoding properly
%%%%%%%%%%%%%

% grad training data - not differentially encoded
symbolTrain = symbolOutHold(1+Noffset+Npad:Noffset+Npad+Nts);

% grab user data and decode if necessary
symbolEst = zeros(1,numberSymbols);
symbolPrev = trainBits1(end);
if diffOn == 0
    symbolData = ...
        symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols); %#ok<NASGU>
else
    % decoding is simply comparing adjacent received symbols
    symbolTemp = ...
        symbolOutHold(1+Noffset+Npad+Nts:Noffset+Npad+Nts+numberSymbols+32);
    for i1 = 1:length(symbolTemp)
        if symbolTemp(i1) == symbolPrev
            symbolEst(i1) = 1;
        else
            symbolEst(i1) = 0;
        end
        symbolPrev = symbolTemp(i1);
    end
end

```

```
% training data
trainDataEst = symbolTrain(1:Nts);
trainDiff = abs(trainDataEst-trainBits1');

% user data
userDataEst = symbolEst(1:numberSymbols);
dataEst = zeros(1,numberBytes);
for i1 = 1:numberBytes
    y = userDataEst((i1-1)*8+1:i1*8);
    dataEst(i1) = bin2dec(char(y+48));
end
userDiff = abs(dataEst-data);

disp(['Training Difference: ',num2str(sum(trainDiff)), ...
    ' User Data Difference: ',num2str(sum(userDiff))]);

% run it through and check CRC
genPoly = 69665;
c = symbolEst;
cEst = c(1,:);
cEst2 = [cEst(1:end-32) cEst(end-15:end)];
cEst = cEst2;

valueCRCc = 65535;
for i1 = 1:length(cEst)
    valueCRCsh1 = bitsll(uint16(valueCRCc), 1);
    valueCRCadd1 = bitor(uint16(valueCRCsh1), cEst(i1));
    leadValue = bitget( valueCRCadd1, 16);
    if (leadValue == 1)
        valueCRCxor = bitxor(uint16(valueCRCadd1), uint16(genPoly));
    else
        valueCRCxor = bitxor(uint16(valueCRCadd1), 0);
    end
    valueCRCc = valueCRCxor;
end
if valueCRCc == 0
    disp('CRC decoded correctly');
else
    disp('CRC check failed');
end

function make_train_data(filename)
x = load('mlhdlc_dpack_train_data.txt');
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
fprintf(fid,'y = [\n');
fprintf(fid,'%1.0e\n',x);
fprintf(fid,'];\n');
fclose(fid);

function make_pad_data(filename)
rng(1);
x = round(rand(1,2^9));
fid = fopen([filename,'.m'],'w+');
fprintf(fid,['function y = ' filename '\n']);
fprintf(fid,'%%#codegen\n');
```

```
fprintf(fid,'y = [\n');
fprintf(fid,'%1.0e\n',x);
fprintf(fid,']);\n');
fclose(fid);
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_dpack'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_comms_data_packet_tb
```

```
Training Difference: 0 User Data Difference: 0
CRC decoded correctly
```

## Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_dpack
```

Next, add the file 'mlhdlc\_comms\_data\_packet.m' to the project as the MATLAB Function and 'mlhdlc\_comms\_data\_packet\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Build tab and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

## Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_dpack'];
clear mex;
```

```
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Transmit and Receive FIFO

This example shows how to generate HDL code from MATLAB® code modeling transfer data between transmit and receive FIFO.

Let us take a look at the MATLAB design for the transmit and receive FIFO and a testbench that exercises both designs.

```

design_core1 = 'mlhdlc_rx_fifo';
design_core2 = 'mlhdlc_tx_fifo';
testbench_name = 'mlhdlc_fifo_tb';

type('mlhdlc_rx_fifo');

function [dout, empty, byte_ready, full, bytes_available] = ...
    mlhdlc_rx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%
% Copyright 2014-2015 The MathWorks, Inc.

%
% First In First Out (FIFO) structure.
% This FIFO stores integers.
% The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
end

full = 0;
empty = 0;
byte_ready = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

% handshaking logic
if get_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_ready = 1;

```

```
end

if (fifo_enable == 1)
    %%%%%%%%get%%%%%%%put%%%%%%
    if (get_byte && ~empty && handshake == 0 )
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
        byte_ready = 1;
        handshake = 1;
    end
    %%%%%%%%put%%%%%%
    if (store_byte && ~full)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
end

type('mlhdlc_tx_fifo');

function [dout, empty, byte_received, full, bytes_available, dbg_fifo_enable] = ...
    mlhdlc_tx_fifo(get_byte, store_byte, byte_in, reset_fifo, fifo_enable)
%
% Copyright 2014-2015 The MathWorks, Inc.

%
% First In First Out (FIFO) structure.
% This FIFO stores integers.
% The FIFO is actually a circular buffer.
%
persistent head tail fifo byte_out handshake

if (reset_fifo || isempty(head))
    head = 1;
    tail = 2;
    byte_out = 0;
    handshake = 0;
end

if isempty(fifo)
    fifo = zeros(1,1024);
```

```

end

full = 0;
empty = 0;
byte_received = 0;

% Section for checking full and empty cases
if ((tail == 1 && head == 1024) || ((head + 1) == tail))
    empty = 1;
end
if ((head == 1 && tail == 1024) || ((tail + 1) == head))
    full = 1;
end

% handshaking logic
if store_byte == 0
    handshake = 0;
end
if handshake == 1
    byte_received = 1;
end

if (fifo_enable == 1)
    %%%%%%%get%%%%%
    if (get_byte && ~empty)
        head = head + 1;
        if head == 1025
            head = 1;
        end
        byte_out = fifo(head);
    end
    %%%%%%put%%%%%
    if (store_byte && ~full && handshake == 0)
        fifo(tail) = byte_in;
        tail = tail + 1;
        if tail == 1025
            tail = 1;
        end
        byte_received = 1;
        handshake = 1;
    end
end

% Section for calculating num bytes in FIFO
if (head < tail)
    bytes_available = (tail - head) - 1;
else
    bytes_available = (1024 - head) + tail - 1;
end

dout = byte_out;
dbg_fifo_enable = fifo_enable;
end

type('mlhdlc_fifo_tb');

%%%%%%%%%%%%%

```

```
% simulation parameters
%%%%%%%%%%%%%
% data payload creation

% Copyright 2014-2015 The MathWorks, Inc.

messageASCII = 'Hello World!';
message = double(unicode2native(messageASCII));
msgLength = length(message);
%%%%%%%%%%%%%
% TX_FIFO core
%%%%%%%%%%%%%
numBytesToFifo = 1;
tx_get_byte = 0;
tx_full = 0;
tx_byte_received = 0;
i1 = 1;

while (numBytesToFifo <= msgLength && ~tx_full)
    % first thing the processor does is clear the internal tx fifo
    if i1 == 1
        tx_reset_fifo = 1;
        mlhdlc_tx_fifo(0, 0, 0, tx_reset_fifo, 1);
    else
        tx_reset_fifo = 0;
    end
    if (i1 > 1)
        tx_data_in = message(numBytesToFifo);
        numBytesToFifo = numBytesToFifo + 1;
        tx_store_byte = 1;
        while (tx_byte_received == 0)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
        tx_store_byte = 0;
        while (tx_byte_received == 1)
            [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
                mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);
        end
    end
    i1 = i1 + 1;
end
%%%%%%%%%%%%%
% Transfer Bytes from TX FIFO to RX FIFO
%%%%%%%%%%%%%
i1 = 1;

tx_get_byte = 0;
tx_store_byte = 0;
tx_data_in = 0;
tx_reset_fifo = 0;

rx_get_byte = 0;
rx_data_in = 0;
rx_reset_fifo = 0;

while (tx_bytes_available > 0)
    if i1 == 1
```

```

        rx_reset_fifo = 1;
        mlhdlc_rx_fifo(0, 0, 0, rx_reset_fifo, 1);
    else
        rx_reset_fifo = 0;
    end
    if (il > 1)
        tx_get_byte = 1;
        rx_store_byte = 1;
        [tx_data_out, tx_empty, tx_byte_received, tx_full, tx_bytes_available] = ...
            mlhdlc_tx_fifo(tx_get_byte, tx_store_byte, tx_data_in, tx_reset_fifo, 1);

        rx_data_in = tx_data_out;

        [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
            mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
    end
    il = il + 1;
end
%%%%%%%%%%%%%
% RX_FIFO core
%%%%%%%%%%%%%
numBytesFromFifo = 1;
rx_store_byte = 0;
rx_byte_recieved = 0;
il = 1;
msgBytes = zeros(1,msgLength);

while (~rx_empty)
    % first thing the processor does is clear the internal rx fifo
    if (il > 1)
        rx_get_byte = 1;
        while (rx_byte_ready == 0)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
        msgBytes(il-1) = rx_data_out;
        rx_get_byte = 0;
        while (rx_byte_ready == 1)
            [rx_data_out, rx_empty, rx_byte_ready, rx_full, rx_bytes_available] = ...
                mlhdlc_rx_fifo(rx_get_byte, rx_store_byte, rx_data_in, rx_reset_fifo, 1);
        end
    end
    il = il + 1;
end
%%%%%%%%%%%%%

numRecBytes = numBytesFromFifo;
if sum(msgBytes-message) == 0
    disp('Received message correctly');
else
    disp('Received message incorrectly');
end
native2unicode(msgBytes)

```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemos');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fifo'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_fifo_tb.m*'), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_rx_fifo.m*'), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_tx_fifo.m*'), mlhdlc_temp_dir);

% Additional test files
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_rx_fifo_tb.m*'), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_tx_fifo_tb.m*'), mlhdlc_temp_dir);
```

## Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_fifo_tb

Received message correctly

ans =

'Hello World!'
```

## Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_fifo
```

Next, add the file 'mlhdlc\_fifo.m' to the project as the MATLAB Function and 'mlhdlc\_fifo\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Build tab and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

## Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemos');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fifo'];
```

```
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## HDL Code Generation for Harris Corner Detection Algorithm

This example shows how to generate HDL code from a MATLAB® design that computes the corner metric by using Harris' technique.

### Corner Detection Algorithm

A corner is a point in an image where two edges of the image intersect. The corners are robust to image rotation, translation, and illumination. Corners contain important features that you can use in many applications such as restoring image information, image registration, and object tracking.

Corner detection algorithms identify the corners by using a corner metric. This metric corresponds to the likelihood of pixels located at the corner of certain objects. Peaks of corner metric identify the corners. See also Corner Detection (Computer Vision Toolbox) in the Computer Vision Toolbox documentation. The corner detection algorithm:

1. Reads the input image.

```
Image_in = checkerboard(10);
```

2. Finds the corners.

```
cornerDetector = detectHarrisFeatures(Image_in);
```

3. Displays the results.

```
[~,metric] = step(cornerDetector,image_in);
figure;
subplot(1,2,1);
imshow(image_in);
title('Original');
subplot(1,2,2);
imshow(imadjust(metric));
title('Corner metric');
```

### Corner Detection MATLAB Design

```
design_name = 'mlhdlc_corner_detection';
testbench_name = 'mlhdlc_corner_detection_tb';
```

Review the MATLAB design:

```
edit(design_name);

%#codegen
function [valid, ed, xfo, yfo, cm] = mlhdlc_corner_detection(data_in)
% Copyright 2011-2019 The MathWorks, Inc.

[~, ed, xfo, yfo] = mlhdlc_sobel(data_in);

cm = compute_corner_metric(xfo, yfo);

% compute valid signal
persistent cnt
if isempty(cnt)
    cnt = 0;
```

```

end
cnt = cnt + 1;
valid = cnt > 3*80+3 && cnt <= 80*80+3*80+3;
end

%%%%%
function bm = compute_corner_metric(gh, gv)

cmh = make_buffer_matrix_gh(gh);
cmv = make_buffer_matrix_gv(gv);
bm = compute_harris_metric(cmh, cmv);

end

%%%%%
function bm = make_buffer_matrix_gh(gh)

persistent b1 b2 b3 b4;
if isempty(b1)
    b1 = dsp.Delay('Length', 80);
    b2 = dsp.Delay('Length', 80);
    b3 = dsp.Delay('Length', 80);
    b4 = dsp.Delay('Length', 80);
end

b1p = step(b1, gh);
b2p = step(b2, b1p);
b3p = step(b3, b2p);
b4p = step(b4, b3p);

cc = [b4p b3p b2p b1p gh];

persistent h1 h2 h3 h4;
if isempty(h1)
    h1 = dsp.Delay();
    h2 = dsp.Delay();
    h3 = dsp.Delay();
    h4 = dsp.Delay();
end

h1p = step(h1, cc);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);

bm = [h4p h3p h2p h1p cc];
end

%%%%%
function bm = make_buffer_matrix_gv(gv)

persistent b1 b2 b3 b4;
if isempty(b1)
    b1 = dsp.Delay('Length', 80);
    b2 = dsp.Delay('Length', 80);
    b3 = dsp.Delay('Length', 80);

```

```
b4 = dsp.Delay('Length', 80);
end

b1p = step(b1, gv);
b2p = step(b2, b1p);
b3p = step(b3, b2p);
b4p = step(b4, b3p);

cc = [b4p b3p b2p b1p gv];

persistent h1 h2 h3 h4;
if isempty(h1)
    h1 = dsp.Delay();
    h2 = dsp.Delay();
    h3 = dsp.Delay();
    h4 = dsp.Delay();
end

h1p = step(h1, cc);
h2p = step(h2, h1p);
h3p = step(h3, h2p);
h4p = step(h4, h3p);

bm = [h4p h3p h2p h1p cc];
end

%%%%%
function cm = compute_harris_metric(gh, gv)

[g1, g2, g3] = gaussian_filter(gh, gv);
[s1, s2, s3] = reduce_matrix(g1, g2, g3);

cm = (((s1*s3) - (s2*s2)) - (((s1+s3) * (s1+s3)) * 0.04));
end

%%%%%
function [g1, g2, g3] = gaussian_filter(gh, gv)

%g=fspecial('gaussian',[5 5],1.5);
g = [0.0144    0.0281    0.0351    0.0281    0.0144
      0.0281    0.0547    0.0683    0.0547    0.0281
      0.0351    0.0683    0.0853    0.0683    0.0351
      0.0281    0.0547    0.0683    0.0547    0.0281
      0.0144    0.0281    0.0351    0.0281    0.0144];

g1 = (gh .* gh) .* g(:)';
g2 = (gh .* gv) .* g(:)';
g3 = (gv .* gv) .* g(:)';
end

%%%%%
function [s1, s2, s3] = reduce_matrix(g1, g2, g3)

s1 = sum(g1);
s2 = sum(g2);
```

```
s3 = sum(g3);
end
```

The MATLAB function is modular and uses several functions to compute the corners of the image. The function:

- `compute_corner_metric` computes the corner metric matrix by instantiating the function `compute_harris_metric`.
- `compute_harris_metric` detects the corner features in the input image by instantiating functions `gaussian_filter` and `reduce_matrix`. The function takes outputs of `make_buffer_matrix_gh` and `make_buffer_matrix_gv` as the inputs.

### Corner Detection MATLAB Test Bench

Review the MATLAB test bench:

```
edit(testbench_name);

clear mlhdlc_corner_detection;
clear mlhdlc_sobel;

% Copyright 2011-2019 The MathWorks, Inc.

image_in = checkerboard(10);
[image_height, image_width] = size(image_in);

% Pre-allocating y for simulation performance
y_cm = zeros(image_height, image_width);
y_ed = zeros(image_height, image_width);
gradient_hori = zeros(image_height,image_width);
gradient_vert = zeros(image_height,image_width);

dataValidOut = y_cm;

idx_in = 1;
idx_out = 1;
for i=1:image_width+3
    for j=1:image_height+3
        if idx_in <= image_width * image_height
            u = image_in(idx_in);
        else
            u = 0;
        end
        idx_in = idx_in + 1;
        [valid, ed, gh, gv, cm] = mlhdlc_corner_detection(u);
        if valid
            y_cm(idx_out) = cm;
            y_ed(idx_out) = ed;
            gradient_hori(idx_out) = gh;
            gradient_vert(idx_out) = gv;
        end
    end
    idx_out = idx_out + 1;
end
```

```
        idx_out = idx_out + 1;
    end
end
end

padImage = y_cm;
findLocalMaxima = vision.LocalMaximaFinder('MaximumNumLocalMaxima',100, ...
    'NeighborhoodSize', [11 11], ...
    'Threshold', 0.0005);
Corners = step(findLocalMaxima, padImage);
drawMarkers = vision.MarkerInserter('Size', 2); % Draw circles at corners
ImageCornersMarked = step(drawMarkers, image_in, Corners);

% Display results
% ...
%

nplots = 4;

scrsz = get(0,'ScreenSize');
figure('Name', [mfilename, '_plot'], 'Position',[1 300 700 200])

subplot(1,nplots,1);
imshow(image_in,[min(image_in(:)) max(image_in(:))]);
title('Checker Board')
axis square

subplot(1,nplots,2);
imshow(gradient_hori(3:end,3:end),[min(gradient_hori(:)) max(gradient_hori(:))]);
title(['Vertical',newline, ' Gradient'])
axis square

subplot(1,nplots,3);
imshow(gradient_vert(3:end,3:end),[min(gradient_vert(:)) max(gradient_vert(:))]);
title(['Horizontal',newline, ' Gradient'])
axis square

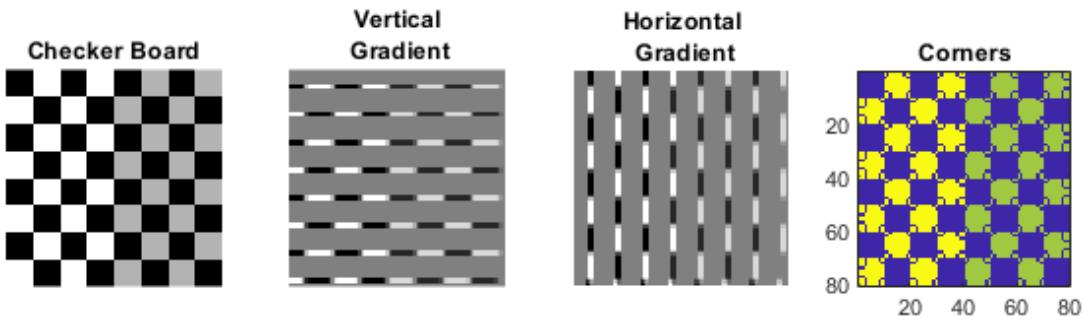
% subplot(1,nplots,4);
% imshow(y_ed);
% title('Edges')

subplot(1,nplots,4);
imagesc(ImageCornersMarked)
title('Corners');
axis square
```

### Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

`mlhdlc_corner_detection_tb`



### Create a Folder and Copy Relevant Files

Before you generate HDL code for the MATLAB design, copy the design and test bench files to a writeable folder. These commands copy the files to a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_cdetect'];
```

Create a temporary folder and copy the MATLAB files.

```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

Copy the design files to the temporary directory.

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_sobel.m*'), mlhdlc_temp_dir);
```

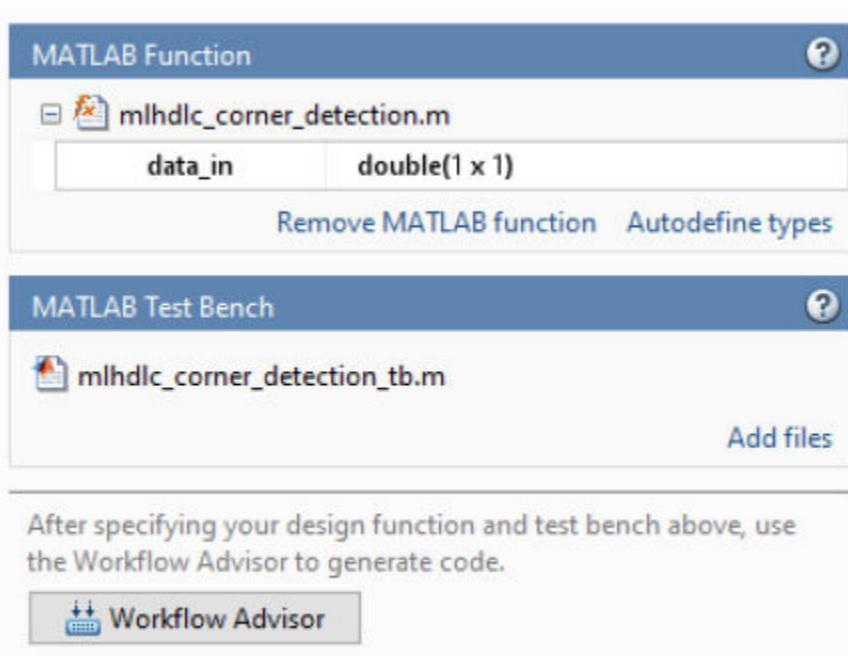
### Create an HDL Coder™ Project

1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_corner_detect_prj
```

2. Add the file `mlhdlc_corner_detection.m` to the project as the **MATLAB Function** and `mlhdlc_corner_detection_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefines types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_corner_detection.m`.



Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

- 1 Click the **Workflow Advisor** button to start the Workflow Advisor.
- 2 Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_corner_detection_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also “Generate Instantiable Code for Functions” on page 5-11.

### Clean Up Generated Files

To clean up the temporary project folder, run these commands:

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_cdetect'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# HDL Code Generation for Adaptive Median Filter

This example shows how to generate HDL code from a MATLAB® design that implements an adaptive median filter algorithm and generates HDL code.

## Adaptive Filter MATLAB Design

An adaptive median filter performs spatial processing to reduce noise in an image. The filter compares each pixel in the image to the surrounding pixels. If one of the pixel values differ significantly from majority of the surrounding pixels, the pixel is treated as noise. The filtering algorithm then replaces the noise pixel by the median values of the surrounding pixels. This process repeats until all noise pixels in the image are removed.

```
design_name = 'mlhdlc_median_filter';
testbench_name = 'mlhdlc_median_filter_tb';
```

Review the MATLAB design:

```
edit(design_name);

%#codegen
function [pixel_val, pixel_valid] = mlhdlc_median_filter(c_data, c_idx)
% Copyright 2011-2019 The MathWorks, Inc.

smax = 9;
persistent window;
if isempty(window)
    window = zeros(smax, smax);
end

cp = ceil(smax/2); % center pixel;

w3 = -1:1;
w5 = -2:2;
w7 = -3:3;
w9 = -4:4;

r3 = cp + w3;      % 3x3 window
r5 = cp + w5;      % 5x5 window
r7 = cp + w7;      % 7x7 window
r9 = cp + w9;      % 9x9 window

d3x3 = window(r3, r3);
d5x5 = window(r5, r5);
d7x7 = window(r7, r7);
d9x9 = window(r9, r9);

center_pixel = window(cp, cp);

% use 1D filter for 3x3 region
outbuf = get_median_1d(d3x3(:));
[min3, med3, max3] = getMinMaxMed_1d(outbuf);

% use 2D filter for 5x5 region
outbuf = get_median_2d(d5x5);
```

```
[min5, med5, max5] = getMinMaxMed_2d(outbuf);

% use 2D filter for 7x7 region
outbuf = get_median_2d(d7x7);
[min7, med7, max7] = getMinMaxMed_2d(outbuf);

% use 2D filter for 9x9 region
outbuf = get_median_2d(d9x9);
[min9, med9, max9] = getMinMaxMed_2d(outbuf);

pixel_val = get_new_pixel(min3, med3, max3, ...
    min5, med5, max5, ...
    min7, med7, max7, ...
    min9, med9, max9, ...
    center_pixel);

% we need to wait until 9 cycles for the buffer to fill up
% output is not valid every time we start from col1 for 9 cycles.
persistent datavalid
if isempty(datavalid)
    datavalid = false;
end
pixel_valid = datavalid;
datavalid = (c_idx >= smax);

% build the 9x9 buffer
window(:,2:smax) = window(:,1:smax-1);
window(:,1) = c_data;

end

%%%%%%%%%%%%%
function [min, med, max] = getMinMaxMed_1d(inbuf)

max = inbuf(1);
med = inbuf(ceil(numel(inbuf)/2));
min = inbuf(numel(inbuf));

end

%%%%%%%%%%%%%
function [min, med, max] = getMinMaxMed_2d(inbuf)

[nrows, ncols] = size(inbuf);
max = inbuf(1, 1);
med = inbuf(ceil(nrows/2), ceil(ncols/2));
min = inbuf(nrows, ncols);

end

%%%%%%%%%%%%%
function new_pixel = get_new_pixel...
    min3, med3, max3, ...
    min5, med5, max5, ...
    min7, med7, max7, ...
```

```

min9, med9, max9, ...
center_data)

if (med3 > min3 && med3 < max3)
    new_pixel = get_center_data(min3, med3, max3,center_data);
elseif (med5 > min5 && med5 < max5)
    new_pixel = get_center_data(min5, med5, max5,center_data);
elseif (med7 > min7 && med7 < max7)
    new_pixel = get_center_data(min7, med7, max7,center_data);
elseif (med9 > min9 && med9 < max9)
    new_pixel = get_center_data(min9, med9, max9,center_data);
else
    new_pixel = center_data;
end
end

%%%%%%%%%%%%%
function [new_data] = get_center_data(min,med,max,center_data)
if center_data <= min || center_data >= max
    new_data = med;
else
    new_data = center_data;
end
end

%%%%%%%%%%%%%
% perform median 1d computation
%%%%%%%%%%%%%
function outbuf = get_median_1d(inbuf)

numpixels = length(inbuf);

tbuf = inbuf;

for ii=coder.unroll(1:numpixels)
    if bitand(ii,uint32(1)) == 1
        tbuf = compare_stage1(tbuf);
    else
        tbuf = compare_stage2(tbuf);
    end
end
outbuf = tbuf;
end

function outbuf = compare_stage1(inbuf)
numpixels = length(inbuf);
tbuf = compare_stage(inbuf(1:numpixels-1));
outbuf = [tbuf(:)' inbuf(numpixels)];
end

function outbuf = compare_stage2(inbuf)
numpixels = length(inbuf);
tbuf = compare_stage(inbuf(2:numpixels));
outbuf = [inbuf(1)' tbuf(:)'];
end

```

```
function [outbuf] = compare_stage(inbuf)

step = 2;
numpixels = length(inbuf);

outbuf = inbuf;

for ii=coder.unroll(1:step:numpixels)
    t = compare_pixels([inbuf(ii), inbuf(ii+1)]);
    outbuf(ii) = t(1);
    outbuf(ii+1) = t(2);
end

end

function outbuf = compare_pixels(inbuf)
if (inbuf(1) > inbuf(2))
    outbuf = [inbuf(1), inbuf(2)];
else
    outbuf = [inbuf(2), inbuf(1)];
end
end

%%%%%%%%%%%%%
% perform median 2d computation
%%%%%%%%%%%%%
function outbuf = get_median_2d(inbuf)

outbuf = inbuf;
[nrows, ncols] = size(inbuf);
for ii=coder.unroll(1:ncols)
    colData = outbuf(:, ii)';
    colDataOut = get_median_1d(colData)';
    outbuf(:, ii) = colDataOut;
end
for ii=coder.unroll(1:nrows)
    rowData = outbuf(ii, :);
    rowDataOut = get_median_1d(rowData);
    outbuf(ii, :) = rowDataOut;
end
end
```

The MATLAB function is modular and uses several functions to filter the noise in the image.

### Adaptive Filter MATLAB Test Bench

A MATLAB test bench `mlhdlc_median_filter_tb` exercises the filter design by using a representative input range.

Review the MATLAB test bench:

```
edit(testbench_name);
```

```
I = imread('mlhdlc_img_pattern_noisy.tif');
```

```
J = I;

% Copyright 2011-2019 The MathWorks, Inc.

smax = 9;
[nrows, ncols] = size(I);
ll = ceil(smax/2);
ul = floor(smax/2);

for ii=1:ncols-smax
    for jj=1:nrows-smax

        c_idx = ii;
        c_data = double(I(jj:jj+smax-1, ii));

        [pixel_val, pixel_valid] = mlhdlc_median_filter(c_data, c_idx);

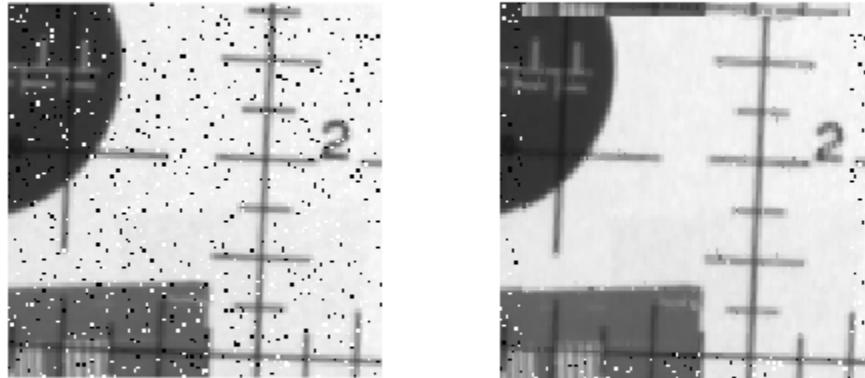
        if pixel_valid
            J(jj, ii) = pixel_val;
        end
    end
end

h = figure;
set( h, 'Name', [ mfilename, '_plot' ] );
subplot( 1, 2, 1 );
imshow( I, [ ] );
subplot( 1, 2, 2 );
imshow( J, [ ] );
```

### Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

```
mlhdlc_median_filter_tb
```



### Create a Folder and Copy Relevant Files

Before you generate HDL code for the MATLAB design, copy the design and test bench files to a writeable folder. These commands copy the files to a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_med_filt'];
```

Create a temporary folder and copy the MATLAB files.

```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

Copy files to the temporary directory.

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_img_pattern_noisy.tif'), mlhdlc_temp_dir);
```

### Accelerating the Design for Faster Simulation

To simulate the test bench faster:

1. Create a MEX file by using MATLAB Coder™. The HDL Workflow Advisor automates these steps when running fixed-point simulations of the design.

```
codegen -o mlhdlc_median_filter -args {zeros(9,1), 0} mlhdlc_median_filter
[~, tbn] = fileparts(testbench_name);
```

2. Simulate the design by using the MEX file. When you run the test bench, HDL Coder uses the MEX file and runs the simulation faster.

```
mlhdlc_median_filter_tb
```

3. Clean up the MEX file.

```
clear mex;
rmdir('codegen', 's');
delete(['mlhdlc_median_filter', '.', mexext]);
```

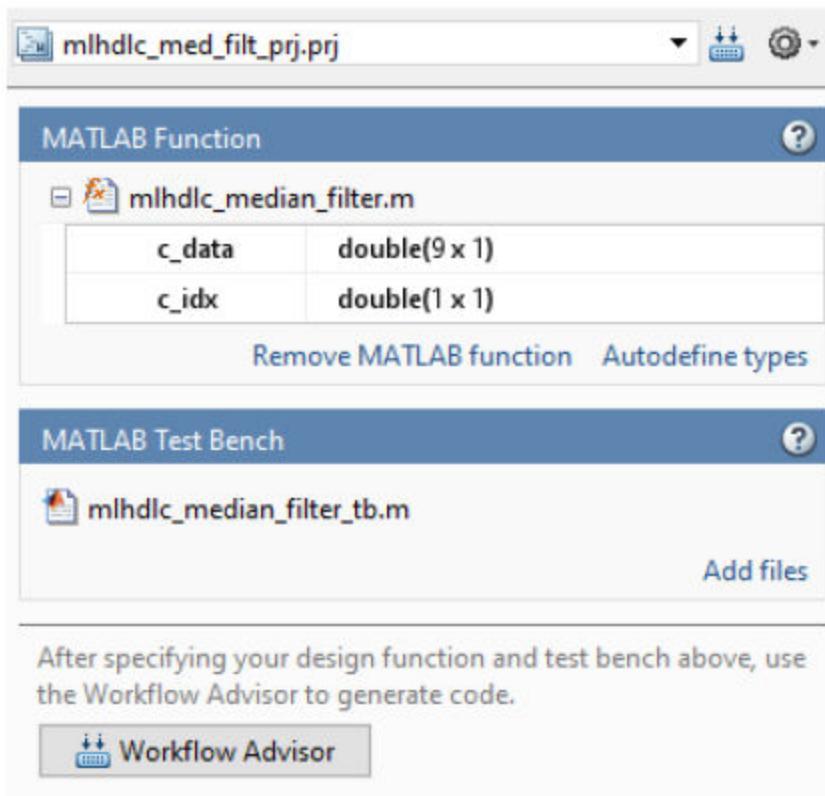
### Create an HDL Coder Project

1. Create an HDL Coder project:

```
coder -hdlcoder -new mlhdlc_med_filt_prj
```

2. Add the file `mlhdlc_median_filter.m` to the project as the **MATLAB Function** and `mlhdlc_median_filter_tb.m` as the **MATLAB Test Bench**.

3. Click **Autodefine types** and use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_median_filter`.



Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

- 1** Click the **Workflow Advisor** button to start the Workflow Advisor.
- 2** Right click the **HDL Code Generation** task and select **Run to selected task**.

A single HDL file `mlhdlc_median_filter_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlinks in the Code Generation Log window.

If you want to generate a HDL file for each function in your MATLAB design, in the **Advanced** tab of the **HDL Code Generation** task, select the **Generate instantiable code for functions** check box. See also “Generate Instantiable Code for Functions” on page 5-11.

### Clean Up Generated Files

To clean up the temporary project folder, run these commands:

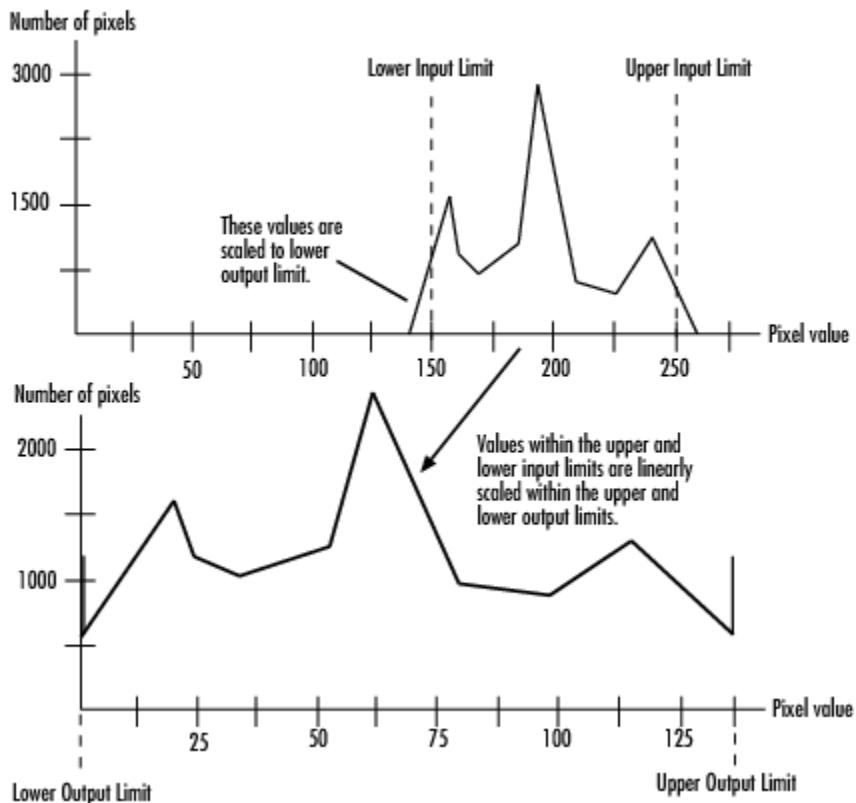
```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemos');
mlhdlc_temp_dir = [tempdir 'mlhdlc_med_filt'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Contrast Adjustment

This example shows how to generate HDL code from a MATLAB® design that adjusts image contrast by linearly scaling pixel values.

## Algorithm

Contrast adjustment adjusts the contrast of an image by linearly scaling the pixel values between upper and lower limits. Pixel values that are above or below this range are saturated to the upper or lower limit value, respectively.



## MATLAB Design

```
design_name = 'mlhdlc_image_scale';
testbench_name = 'mlhdlc_image_scale_tb';
```

Let us take a look at the MATLAB design

```
type(design_name);
```

```
%%%%%%%%%%%%%
% scale.m
%
% Adjust image contrast by linearly scaling pixel values.
%
% The input pixel value range has 14bits and output pixel value range is
% 8bits.
```

```
%  
%%%%%%%%%%%%%%  
function [x_out, y_out, pixel_out] = ...  
    mlhdlc_image_scale(x_in, y_in, pixel_in, ...  
        damping_factor_in, dynamic_range_in, ...  
        tail_size_in, max_gain_in, ...  
        width, height)  
  
% Copyright 2011-2015 The MathWorks, Inc.  
  
persistent histogram1 histogram2  
persistent low_count  
persistent high_count  
persistent offset  
persistent gain  
persistent limits_done  
persistent damping_done  
persistent reset_hist_done  
persistent scaling_done  
persistent hist_ind  
persistent tail_high  
persistent min_hist_damped %Damped lower limit of populated histogram  
persistent max_hist_damped %Damped upper limit of populated histogram  
persistent found_high  
persistent found_low  
  
DR_PER_BIN      = 8;  
SF              = 1./(1:(2^14/8)); % be nice to fix this  
NR_OF_BINS      = (2^14/DR_PER_BIN) - 1;  
MAX_DF          = 255;  
  
if isempty(offset)  
    offset          = 1;  
    gain           = 1;  
    limits_done    = 1;  
    damping_done   = 1;  
    reset_hist_done = 1;  
    scaling_done   = 1;  
    hist_ind       = 1;  
    tail_high      = NR_OF_BINS;  
    low_count      = 0;  
    high_count     = 0;  
    min_hist_damped = 0;  
    max_hist_damped = (2^14/DR_PER_BIN) - 1;  
    found_high     = 0;  
    found_low      = 0;  
end  
if isempty(histogram1)  
    histogram1     = zeros(1, NR_OF_BINS+1);  
    histogram2     = zeros(1, NR_OF_BINS+1);  
end  
  
if y_in < height  
    frame_valid = 1;  
    if x_in < width  
        line_valid = 1;  
    else  
        line_valid = 0;  
    end  
end
```

```

        end
    else
        frame_valid = 0;
        line_valid = 0;
    end

% initialize at beginning of frame
if x_in == 0 && y_in == 0
    limits_done = 0;
    damping_done = 0;
    reset_hist_done = 0;
    scaling_done = 0;
    low_count = 0;
    high_count = 0;
    hist_ind = 1;
end

max_gain_frac = max_gain_in/2^4;
pix11 = floor(pixel_in/DR_PER_BIN);
pix_out_temp = pixel_in;

*****%
%Check if valid part of frame. If pixel is valid remap pixel to desired
%output dynamic range (dynamic_range_in) by subtracting the damped offset
%(min_hist_damped) and applying the calculated gain calculated from the
%previous frame histogram statistics.
*****%

% histogram read
histReadIndex1 = 1;
histReadIndex2 = 1;
if frame_valid && line_valid
    histReadIndex1 = pix11+1;
    histReadIndex2 = pix11+1;
elseif ~limits_done
    histReadIndex1 = hist_ind;
    histReadIndex2 = NR_OF_BINS - hist_ind;
end
histReadValue1 = histogram1(histReadIndex1);
histReadValue2 = histogram2(histReadIndex2);
histWriteIndex1 = NR_OF_BINS+1;
histWriteIndex2 = NR_OF_BINS+1;
histWriteValue1 = 0;
histWriteValue2 = 0;
if frame_valid
    if line_valid
        temp_sum = histReadValue1 + 1;
        ind = min(pix11+1, NR_OF_BINS);
        val = min(temp_sum, tail_size_in);
        histWriteIndex1 = ind;
        histWriteValue1 = val;
        histWriteIndex2 = ind;
        histWriteValue2 = val;

        %Scale pixel
        pix_out_offs_corr = pixel_in - min_hist_damped*DR_PER_BIN;
        pix_out_scaled = pix_out_offs_corr * gain;
        pix_out_clamp = max(min(dynamic_range_in, pix_out_scaled), 0);
        pix_out_temp = pix_out_clamp;
    end
end

```

```
    end
else
%*****Ignore tail_size_in pixels and find lower and upper limits of the
%histogram.
%*****
if ~limits_done
    if hist_ind == 1
        tail_high = NR_OF_BINS-1;
        offset = 1;
        found_high = 0;
        found_low = 0;
    end

    low_count = low_count + histReadValue1;
    hist_ind_high = NR_OF_BINS - hist_ind;
    high_count = high_count + histReadValue2;

    %Found enough high outliers
    if high_count > tail_size_in && ~found_high
        tail_high = hist_ind_high;
        found_high = 1;
    end

    %Found enough low outliers
    if low_count > tail_size_in && ~found_low
        offset = hist_ind;
        found_low = 1;
    end

    hist_ind = hist_ind + 1;
    %All bins checked so limits must already be found
    if hist_ind >= NR_OF_BINS
        hist_ind = 1;
        limits_done = 1;
    end
%*****
%Damp the limit change to avoid image flickering. Code below equivalent
%to: max_hist_damped = damping_factor_in*max_hist_dampedOld +
%(1-damping_factor_in)*max_hist_dampedNew;
%*****
elseif ~damping_done
    min_hist_weighted_old = damping_factor_in*min_hist_damped;
    min_hist_weighted_new = (MAX_DF-damping_factor_in+1)*offset;
    min_hist_weighted = (min_hist_weighted_old + ...
        min_hist_weighted_new)/256;
    min_hist_damped = max(0, min_hist_weighted);
    max_hist_weighted_old = damping_factor_in*max_hist_damped;
    max_hist_weighted_new = (MAX_DF-damping_factor_in+1)*tail_high;
    max_hist_weighted = (max_hist_weighted_old + ...
        max_hist_weighted_new)/256;
    max_hist_damped = min(NR_OF_BINS, max_hist_weighted);
    damping_done = 1;
    hist_ind = 1;
%*****
%Reset all bins to zero. More than one bin can be reset per function
%call if blanking time is too short.
%*****
```

```

elseif ~reset_hist_done
    histWriteIndex1 = hist_ind;
    histWriteValue1 = 0;
    histWriteIndex2 = hist_ind;
    histWriteValue2 = 0;
    hist_ind = hist_ind+1;
    if hist_ind == NR_OF_BINS
        reset_hist_done = 1;
    end
%*****
%The gain factor is determined by comparing the measured damped actual
%dynamic range to the desired user specified dynamic range. Input
%dynamic range is measured in bins over DR_PER_BIN space.
%*****
elseif ~scaling_done
    dr_in = round(max_hist_damped - min_hist_damped);
    gain_temp = dynamic_range_in*SF(dr_in);
    gain_scaled = gain_temp/DR_PER_BIN;
    gain = min(max_gain_frac, gain_scaled);
    scaling_done = 1;
    hist_ind = 1;
end
histogram1(histWriteIndex1) = histWriteValue1;
histogram2(histWriteIndex2) = histWriteValue2;

x_out = x_in;
y_out = y_in;
pixel_out = pix_out_temp;

type(testbench_name);

%Test bench for scaling, analogous to automatic gain control (AGC)

% Copyright 2011-2018 The MathWorks, Inc.

testFile = 'mlhdlc_img_peppers.png';
imgOrig = imread(testFile);
[height, width] = size(imgOrig);
imgOut = zeros(height,width);
hBlank = 20;
% make sure we have enough vertical blanking to filter the histogram
vBlank = ceil(2^14/(width+hBlank));

%df - Temporal damping factor of rescaling
%dr - Desired output dynamic range
df = 0;
dr = 255;
nrOfOutliers = 248;
maxGain = 2*2^4;

for frame = 1:2
    disp(['frame: ', num2str(frame)]);
    for y_in = 0:height+vBlank-1
        %disp(['frame: ', num2str(frame), ' of 2, row: ', num2str(y_in)]);
        for x_in = 0:width+hBlank-1
            if x_in < width && y_in < height

```

```
    pixel_in = double(imgOrig(y_in+1, x_in+1));
else
    pixel_in = 0;
end

[x_out, y_out, pixel_out] = ...
    mlhdlc_image_scale(x_in, y_in, pixel_in, df, dr, ...
        nrOfOutliers, maxGain, width, height);

if x_out < width && y_out < height
    imgOut(y_out+1,x_out+1) = pixel_out;
end
end
end

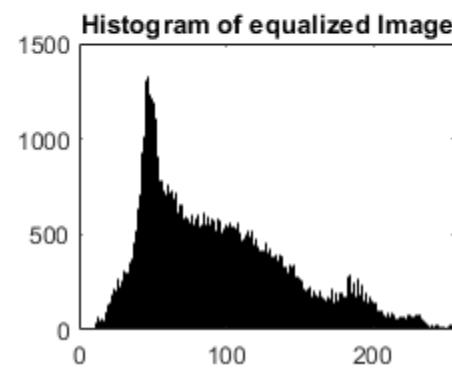
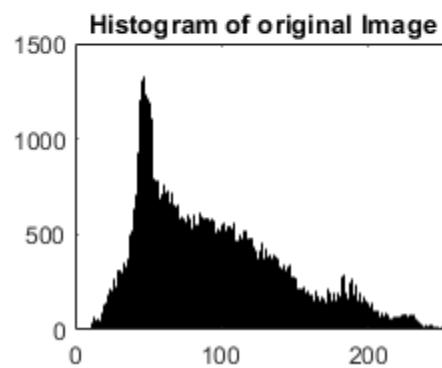
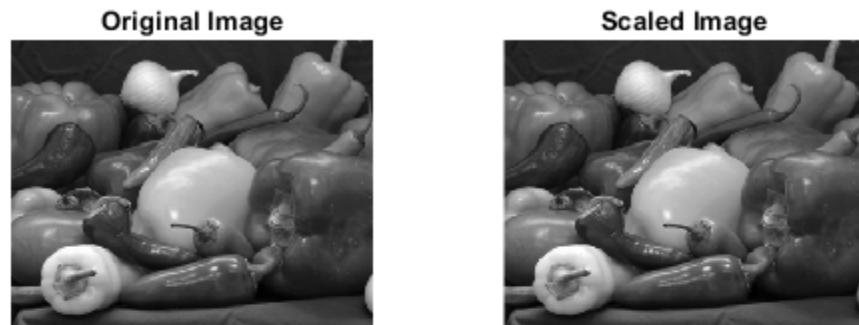
figure('Name', [mfilename, '_scale_plot']);
imgOut = round(255*imgOut/max(max(imgOut)));
subplot(2,2,1); imshow(imgOrig, []);
title('Original Image');
subplot(2,2,2); imshow(imgOut, []);
title('Scaled Image');
subplot(2,2,3); histogram(double(imgOrig(:)),2^14-1);
axis([0, 255, 0, 1500]);
title('Histogram of original Image');
subplot(2,2,4); histogram(double(imgOut(:)),2^14-1);
axis([0, 255, 0, 1500]);
title('Histogram of equalized Image');
end
```

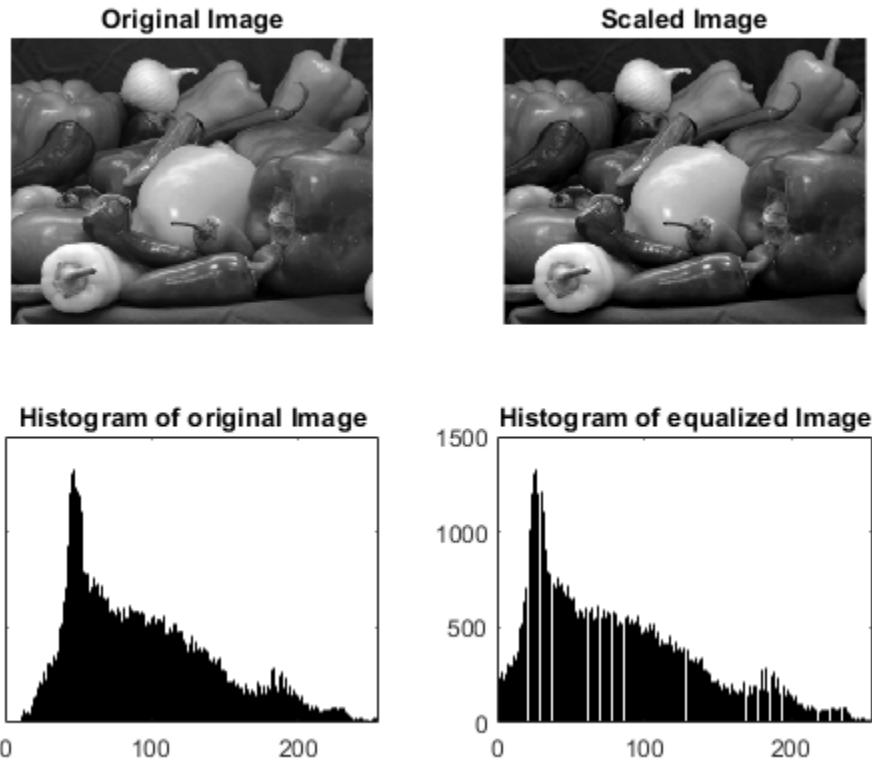
### Simulate the Design

It is a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_image_scale_tb

frame: 1
frame: 2
```





### Setup for the Example

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_scale'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy files to the temp dir
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_img_peppers.png'), mlhdlc_temp_dir);
```

### Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_scale_prj
```

Next, add the file 'mlhdlc\_image\_scale.m' to the project as the MATLAB Function and 'mlhdlc\_image\_scale\_tb.m' as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor from the Build tab and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

## Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_scale'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

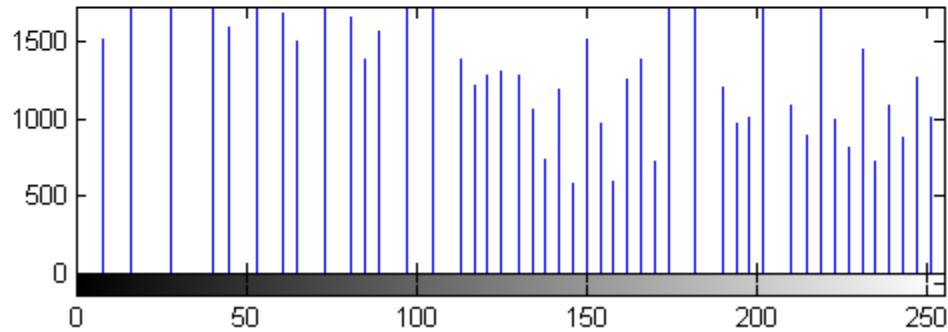
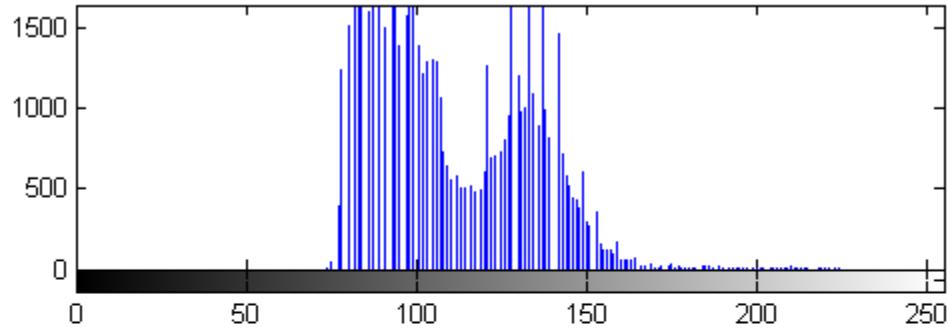
## Image Enhancement by Histogram Equalization

This example shows how to generate HDL code from a MATLAB® design that does image enhancement using histogram equalization.

### Algorithm

The Histogram Equalization algorithm enhances the contrast of images by transforming the values in an intensity image so that the histogram of the output image is approximately flat.

```
I = imread('pout.tif');
J = histeq(I);
subplot(2,2,1);
imshow( I );
subplot(2,2,2);
imhist(I)
subplot(2,2,3);
imshow( J );
subplot(2,2,4);
imhist(J)
```



### MATLAB Design

```
design_name = 'mlhdlc_heq';
testbench_name = 'mlhdlc_heq_tb';
```

Let us take a look at the MATLAB design

```
type(design_name);
type(testbench_name);
```

### Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_heq_tb
```

### Setup for the Example

Executing the following lines copies the necessary files into a temporary folder

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_heq'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy files to the temp dir
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_img_peppers.png'), mlhdlc_temp_dir);
```

### Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_heq_prj
```

Next, add the file 'mlhdlc\_heq.m' to the project as the MATLAB Function and 'mlhdlc\_heq\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

Launch HDL Advisor and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_heq'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## HDL Code Generation for Image Format Conversion from RGB to YUV

This example shows how to generate HDL code from a MATLAB® design that converts the image format from RGB to YUV.

### MATLAB Design and Test Bench

```
design_name = 'mlhdlc_rgb2yuv';
testbench_name = 'mlhdlc_rgb2yuv_tb';

Review the MATLAB design:

open(design_name)

function [x_out, y_out, y_data_out, u_data_out, v_data_out] = ...
    mlhdlc_rgb2yuv(x_in, y_in, r_in, g_in, b_in)
 %#codegen

% Copyright 2011-2019 The MathWorks, Inc.

persistent RGB_Reg YUV_Reg
persistent x1 x2 y1 y2

if isempty(RGB_Reg)
    RGB_Reg = zeros(3,1);
    YUV_Reg = zeros(3,1);
    x1 = 0; x2 = 0; y1 = 0; y2 = 0;
end

D = [.299 .587 .144; -.147 -.289 .436; .615 -.515 -.1];
C = [0; 128; 128];

RGB = [r_in; g_in; b_in];

YUV_1 = D*RGB_Reg;
YUV_2 = YUV_1 + C;
RGB_Reg = RGB;

y_data_out = round(YUV_Reg(1));
u_data_out = round(YUV_Reg(2));
v_data_out = round(YUV_Reg(3));
YUV_Reg = YUV_2;

x_out = x2; x2 = x1; x1 = x_in;
y_out = y2; y2 = y1; y1 = y_in;
```

Review the MATLAB test bench:

```
open(testbench_name);
```

```
FRAMES = 1;
WIDTH = 752;
```

```

HEIGHT = 480;
HBLANK = 10;%748;
VBLANK = 10;%120;

% Copyright 2011-2019 The MathWorks, Inc.

vidData = double(imread('mlhdlc_img_yuv.tif'));

for f = 1:FRAMES
    vidOut = zeros(HEIGHT, WIDTH, 3);

    for y = 0:HEIGHT+VBLANK-1
        for x = 0:WIDTH+HBLANK-1
            if y >= 0 && y < HEIGHT && x >= 0 && x < WIDTH
                b = vidData(y+1,x+1,1);
                g = vidData(y+1,x+1,2);
                r = vidData(y+1,x+1,3);
            else
                b = 0; g = 0; r = 0;
            end

            [xOut, yOut, yData, uData, vData] = ...
                mlhdlc_rgb2yuv(x, y, r, g, b);

            if yOut >= 0 && yOut < HEIGHT && xOut >= 0 && xOut < WIDTH
                vidOut(yOut+1,xOut+1,:) = [yData vData uData];
            end
        end
    end

    figure(1);
    subplot(1,2,1);
    imshow(uint8(vidData));
    subplot(1,2,2);
    imshow(ycbcr2rgb(uint8(vidOut)));
    drawnow;
end

```

### Test the MATLAB Algorithm

To avoid run-time errors, simulate the design with the test bench.

`mlhdlc_rgb2yuv_tb`



### Create a Folder and Copy Relevant Files

Before you generate HDL code for the MATLAB design, copy the design and test bench files to a writeable folder. These commands copy the files to a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_rgb2yuv'];
```

Create a temporary folder and copy the MATLAB files.

```
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);
```

Copy files to the temporary directory.

```
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

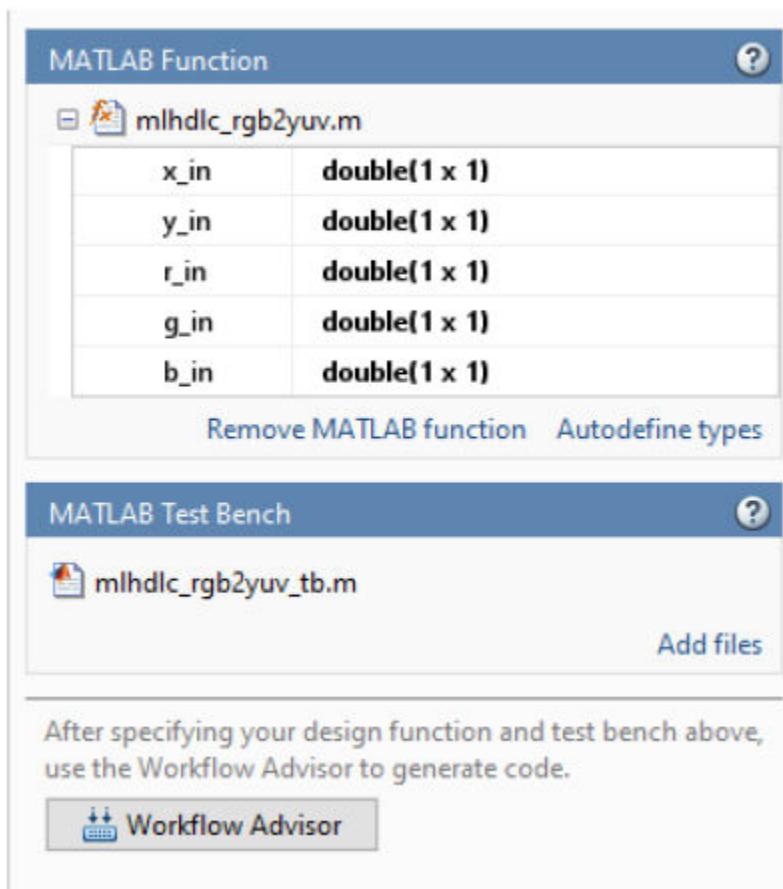
### Create an HDL Coder™ Project

To generate HDL code from a MATLAB design:

1. Create a HDL Coder project:

```
coder -hdlcoder -new mlhdlc_rgb_prj
```

2. Add the file `mlhdlc_rgb2yuv.m` to the project as the **MATLAB Function** and `mlhdlc_rgb2yuv_tb.m` as the **MATLAB Test Bench**.
3. Click **Autodefines types** to use the recommended types for the inputs and outputs of the MATLAB function `mlhdlc_rgb2yuv`.



Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Run Fixed-Point Conversion and HDL Code Generation

- 1 Click the **Workflow Advisor** button to start the Workflow Advisor.
- 2 Right click the **HDL Code Generation** task and select **Run to selected task**.

A HDL file `mlhdlc_rgb2yuv_fixpt.vhd` is generated for the MATLAB design. To examine the generated HDL code for the filter design, click the hyperlink to the HDL file in the Code Generation Log window.

### Clean Up Generated Files

To clean up the temporary project folder, run these commands:

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_rgb2yuv'];
clear mex;
```

```
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# High Dynamic Range Imaging

This example shows how to generate HDL code from a MATLAB® design that implements a high dynamic range imaging algorithm.

## Algorithm

High Dynamic Range Imaging (HDRI or HDR) is a set of methods used in imaging and photography to allow a greater dynamic range between the lightest and darkest areas of an image than current standard digital imaging methods or photographic methods. HDR images can represent more accurately the range of intensity levels found in real scenes, from direct sunlight to faint starlight, and is often captured by way of a plurality of differently exposed pictures of the same subject matter.

## MATLAB Design

```
design_name = 'mlhdlc_hdr';
testbench_name = 'mlhdlc_hdr_tb';
```

Let us take a look at the MATLAB design

```
dbtype(design_name);
```

```

1 function [valid_out, x_out, y_out, ...
2     HDR1, HDR2, HDR3] = mlhdlc_hdr(YShort1, YShort2, YShort3, ...
3     YLong1, YLong2, YLong3, ...
4     plot_y_short_in, plot_y_long_in, ...
5     valid_in, x, y)
6 %
7 %
8 % Copyright 2013-2015 The MathWorks, Inc.
9 %
10 % This design implements a high dynamic range imaging algorithm.
11 %
12 plot_y_short = plot_y_short_in;
13 plot_y_long = plot_y_long_in;
14 %
15 %% Apply Lum(Y) channels LUTs
16 y_short = plot_y_short(uint8(YShort1)+1);
17 y_long = plot_y_long(uint8(YLong1)+1);
18 %
19 y_HDR = (y_short+y_long);
20 %
21 %% Create HDR Chorm channels
22 % HDR per color
23 %
24 HDR1 = y_HDR * 2^-8;
25 HDR2 = (YShort2+YLong2) * 2^-1;
26 HDR3 = (YShort3+YLong3) * 2^-1;
27 %
28 %% Pass on valid signal and pixel location
29 %
30 valid_out = valid_in;
31 x_out = x;
32 y_out = y;
33 %
34 end
```

```
1 dbtype(testbench_name);
2 %
3 %
4 % Copyright 2013-2015 The MathWorks, Inc.
5 %
6 % Clean screen and memory
7 close all
8 clear mlhdlc_hdr
9 set(0,'DefaultFigureWindowStyle','docked')
10 %
11 %
12 %% Read the two exposed images
13 short = imread('mlhdlc_hdr_short.tif');
14 long = imread('mlhdlc_hdr_long.tif');
15 %
16 % define HDR output variable
17 HDR = zeros(size(short));
18 [height, width, color] = size(HDR);
19 %
20 figure('Name', [mfilename, '_plot']);
21 subplot(1,3,1);
22 imshow(short, 'InitialMagnification','fit'), title('short');
23 %
24 subplot(1,3,2);
25 imshow(long, 'InitialMagnification','fit'), title('long');
26 %
27 %
28 %% Create the Lum(Y) channels LUTs
29 % Pre-process
30 % Luminance short LUT
31 ShortLut.x = [0    16    45    96    255];
32 ShortLut.y = [0    20    38    58    115];
33 %
34 % Luminance long LUT
35 LongLut.x = [ 0 255];
36 LongLut.y = [ 0 140];
37 %
38 % Take the same points to plot the joined Lum LUT
39 plot_x = 0:1:255;
40 plot_y_short = interp1(ShortLut.x,ShortLut.y,plot_x); %LUT short
41 plot_y_long = interp1(LongLut.x,LongLut.y,plot_x); %LUT long
42 %
43 %subplot(4,1,3);
44 %plot(plot_x, plot_y_short, plot_x, plot_y_long, plot_x, (plot_y_long+plot_y_short)), grid
45 %
46 %
47 %% Create the HDR Lum channel
48 % The HDR algorithm
49 % read the Y channels
50 %
51 YIQ_short = rgb2ntsc(short);
52 YIQ_long = rgb2ntsc(long);
53 %
54 %% Stream image through HDR algorithm
```

```

56
57     for x=1:width
58         for y=1:height
59             YShort1 = round(YIQ_short(y,x,1)*255); %input short
60             YLong1 = round(YIQ_long(y,x,1)*255); %input long
61
62             YShort2 = YIQ_short(y,x,2); %input short
63             YLong2 = YIQ_long(y,x,2); %input long
64
65             YShort3 = YIQ_short(y,x,3); %input short
66             YLong3 = YIQ_long(y,x,3); %input long
67
68             valid_in = 1;
69
70             [valid_out, x_out, y_out, HDR1, HDR2, HDR3] = mlhdlc_hdr(YShort1, YShort2, YShort3);
71
72             % use x and y to reconstruct image
73             if valid_out == 1
74                 HDR(y_out,x_out,1) = HDR1;
75                 HDR(y_out,x_out,2) = HDR2;
76                 HDR(y_out,x_out,3) = HDR3;
77             end
78         end
79     end
80
81 %% plot HDR
82 HDR_rgb = ntsc2rgb(HDR);
83 subplot(1,3,3);
84 imshow(HDR_rgb, 'InitialMagnification','fit'), title('hdr ');

```

### Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

`mlhdlc_hdr_tb`



### Setup for the Example

Executing the following lines copies the necessary files into a temporary folder

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_hdr'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy files to the temp dir
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_hdr_long.tif'), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_hdr_short.tif'), mlhdlc_temp_dir);
```

### Create a New HDL Coder™ Project

```
coder -hdlcoder -new mlhdlc_hdr_prj
```

Next, add the file 'mlhdlc\_hdr.m' to the project as the MATLAB Function and 'mlhdlc\_hdr\_tb.m' as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### **Creating constant parameter inputs**

This example shows to use pass constant parameter inputs.

In this design the input parameters 'plot\_y\_short\_in' and 'plot\_y\_long\_in' are constant input parameters. You can define them accordingly by modifying the input types as 'constant(double(1x256))'

'plot\_y\_short\_in' and 'plot\_y\_short\_in' are LUT inputs. They are constant folded as double inputs to the design. You will not see port declarations for these two input parameters in the generated HDL code.

Note that inside the design 'mlhdlc\_hdr.m' these variables are reassigned so that they get properly fixed-point converted. This is not necessary if these are purely used as constants for defining sizes of variables for example and not part of the logic.

### **Run Fixed-Point Conversion and HDL Code Generation**

Launch HDL Advisor and right click on the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

### **Convert the design to fixed-point and generate HDL code**

The following script converts the design to fixed-point, and generate HDL code with a test bench.

```
exArgs = {0,0,0,0,0,0,coder.Constant(ones(1,256)),coder.Constant(ones(1,256)),0,0,0};
fc = coder.config('fixpt');
fc.TestBenchName = 'mlhdlc_hdr_tb';
hc = coder.config('hdl');
hc.GenerateHDLTestBench = true;
hc.SimulationIterationLimit = 1000; % Limit number of testbench points
codegen -float2fixed fc -config hc -args exArgs mlhdlc_hdr
```

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

### **Clean up the Generated Files**

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_hdr'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Accelerate a Pixel-Streaming Design Using MATLAB Coder

This example shows how to accelerate a pixel-stream video processing algorithm in MATLAB by using MATLAB Coder™.

You must have a MATLAB Coder license to run this example.

Acceleration with MATLAB Coder enables you to simulate large frame sizes, such as 1080p video, at practical speeds. Use this acceleration workflow after you have debugged the algorithm using a small frame size. Testing a design with a small image is demonstrated in the “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox) example.

### How MATLAB Coder Works

MATLAB Coder generates C code from MATLAB® code. Code generation accelerates simulation by locking-down the sizes and data types of variables. This process removes the overhead of the interpreted language checking for size and data type in every line of code. This example compiles both the test bench file `DesignAccelerationHDLTestBench.m` and the design file `DesignAccelerationHDLDesign.m` into a MEX function, and uses the resulting MEX file to speed up the simulation.

The directive (or pragma) **%#codegen** beneath the function signature indicates that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during code generation. The directive **%#codegen** does not affect interpreted simulation.

### Best Practices

Debugging simulations with large frame sizes is impractical in interpreted mode due to long simulation time. However, debugging a MEX simulation is challenging due to lack of debug access into the code.

To avoid these scenarios, a best practice is to develop and verify the algorithm and test bench using a thumbnail frame size. In most cases, the HDL-targeted design can be implemented with no dependence on frame size. Once you are confident that the design and test bench are working correctly, then increase the frame size in the test bench, and use MATLAB Coder to accelerate the simulation. To increase the frame size, test bench only requires minor changes, as you can see by comparing `DesignAccelerationHDLTestBench.m` with the `PixelStreamingDesignHDLTestBench.m` in “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox).

### Test Bench

In the test bench `DesignAccelerationHDLTestBench.m`, the **videoIn** object reads each frame from a video source, and the **scaler** object interpolates this frame from 240p to 1080p. This 1080p image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function `DesignAccelerationHDLDesign` is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. The **DesignAccelerationHDLViewer** function displays the output and original images side-by-side.

The workflow above is implemented in the following lines of `DesignAccelerationHDLTestBench.m`.

```

...
for f = 1:numFrm
    frmFull = step(videoIn);           % Get a new frame
    frmIn = step(scaler,frmFull);    % Enlarge the frame

    [pixInVec,ctrlInVec] = step(frm2pix,frmIn);
    for p = 1:numPixPerFrm
        [pixOutVec(p),ctrlOutVec(p)] = ...
            visionhdlobel_design(pixInVec(p),ctrlInVec(p));
    end
    frmOut = step(pix2frm,pixOutVec,ctrlOutVec);

    DesignAccelerationHDLViewer(actPixPerLine,actLine,[frmIn uint8(255*frmOut)]);
end
...

```

The data type of `frmIn` is `uint8` while that of `frmOut`, the edge detection output, is logical. Matrices of different data types cannot be concatenated, so `uint8(255*frmOut)` maps logical false and true to `uint8(0)` and `uint8(255)`, respectively.

Both `frm2pix` and `pix2frm` are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing (i.e., `videoIn`, `scaler`, and `viewer` inside the `DesignAccelerationHDLViewer` function).

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

Not all functions used in the test bench support C code generation. For those that do not, such as `tic`, `toc`, `fprintf`, use `coder.extrinsic` to declare them as extrinsic functions. Extrinsic functions are excluded from MEX generation. The simulation executes them in the regular interpreted mode.

## Pixel-Stream Design

The function defined in `DesignAccelerationHDLDesign.m` accepts a pixel stream and five control signals, and returns a modified pixel stream and control signals. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see “Streaming Pixel Interface” (Vision HDL Toolbox).

In this example, the function contains the Edge Detector System object.

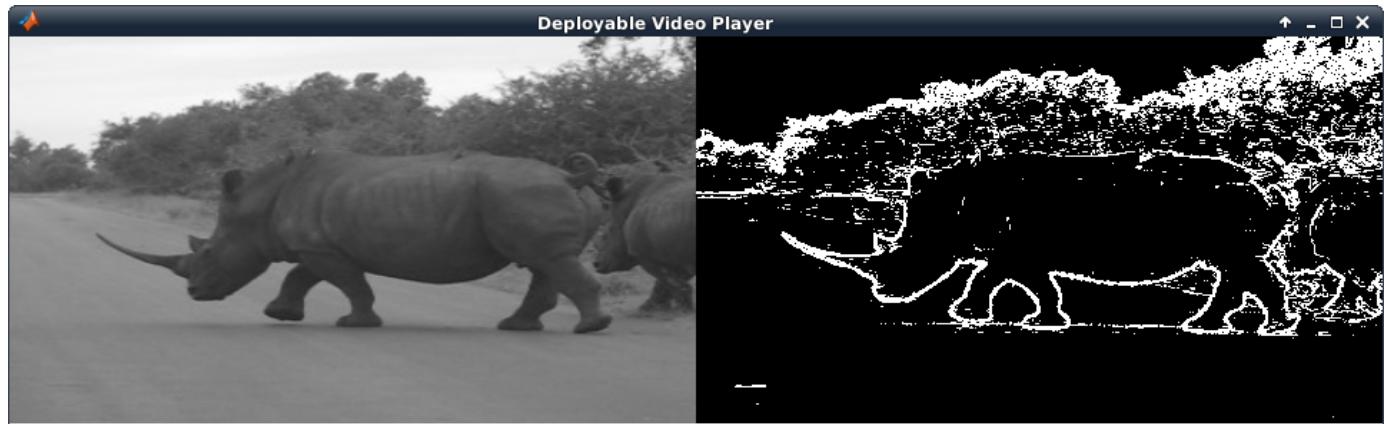
The focus of this example is the workflow, not the algorithm design itself. Therefore, the design code is quite simple. Once you are familiar with the workflow, it is straightforward to implement advanced video algorithms by taking advantage of the functionality provided by the System objects from Vision HDL Toolbox.

## Create MEX File and Simulate the Design

Generate and execute the MEX file.

```
codegen('DesignAccelerationHDLTestBench');
DesignAccelerationHDLTestBench_mex;
```

10 frames have been processed in 11.43 seconds.  
Average frame rate is 0.87 frames/second.



The **viewer** displays the original video on the left, and the output on the right.

### HDL Code Generation

Enter the following command to create a new HDL Coder™ project in the temporary folder

```
coder -hdlcoder -new DesignAccelerationProject
```

Then, add the file `DesignAccelerationHDLDesign.m` to the project as the MATLAB Function and `DesignAccelerationHDLTestBench.m` as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the ‘Code Generation’ step. Choose the option ‘Run to selected task’ to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

# Enhanced Edge Detection from Noisy Color Video

This example shows how to develop a complex pixel-stream video processing algorithm, accelerate its simulation using MATLAB Coder™, and generate HDL code from the design. The algorithm enhances the edge detection from noisy color video.

You must have a MATLAB Coder license to run this example.

This example builds on the “Pixel-Streaming Design in MATLAB” (Vision HDL Toolbox) and the “Accelerate a Pixel-Streaming Design Using MATLAB Coder” (Vision HDL Toolbox) examples.

## Test Bench

In the EnhancedEdgeDetectionHDLTestBench.m file, the **videoIn** object reads each frame from a color video source, and the **imnoise** function adds salt and pepper noise. This noisy color image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function EnhancedEdgeDetectionHDLDesign.m is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. A full-frame reference design EnhancedEdgeDetectionHDLReference.m is also called to process the noisy color image. Its output is compared with that of the pixel-stream design. The function EnhancedEdgeDetectionHDLViewer.m is called to display video outputs.

The workflow above is implemented in the following lines of EnhancedEdgeDetectionHDLTestBench.m.

```
...
frmIn = zeros(actLine,actPixPerLine,3,'uint8');
for f = 1:numFrm
    frmFull = step(videoIn); % Get a new frame
    frmIn = imnoise(frmFull,'salt & pepper'); % Add noise

    % Call the pixel-stream design
    [pixInVec,ctrlInVec] = step(frm2pix,frmIn);
    for p = 1:numPixPerFrm
        [pixOutVec(p),ctrlOutVec(p)] = EnhancedEdgeDetectionHDLDesign(pixInVec(p,:),ctrlInVec(p));
    end
    frmOut = step(pix2frm,pixOutVec,ctrlOutVec);

    % Call the full-frame reference design
    [frmGray,frmDenoise,frmEdge,frmRef] = visionhdlenhancededge_reference(frmIn);

    % Compare the results
    if nnz(imabsdiff(frmRef,frmOut))>20
        printf('frame %d: reference and design output differ in more than 20 pixels.\n',f);
        return;
    end

    % Display the results
    EnhancedEdgeDetectionHDLViewer(actPixPerLine,actLine,[frmGray frmDenoise uint8(255*[frmEdge
end
...
```

Since **frmGray** and **frmDenoise** are **uint8** data type while **frmEdge** and **frmOut** are logical, **uint8(255\*[frmEdge frmOut])** maps logical false and true to **uint8(0)** and **uint8(255)**, respectively, so that matrices can be concatenated.

Both **frm2pix** and **pix2frm** are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing.

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

For the functions that do not support C code generation, such as **tic**, **toc**, **imnoise**, and **fprintf** in this example, use **coder.extrinsic** to declare them as extrinsic functions. Extrinsic functions are excluded from MEX generation. The simulation executes them in the regular interpreted mode. Since **imnoise** is not included in the C code generation process, the compiler cannot infer the data type and size of **frmIn**. To fill in this missing piece, we add the statement **frmIn = zeros(actLine,actPixPerLine,3,'uint8')** before the outer for-loop.

### Pixel-Stream Design

The function defined in **EnhancedEdgeDetectionHDLDesign.m** accepts a pixel stream and a structure consisting of five control signals, and returns a modified pixel stream and control structure. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see the “Streaming Pixel Interface” (Vision HDL Toolbox).

In this example, the **rgb2gray** object converts a color image to grayscale, **medfil** removes the salt and pepper noise, **sobel** highlights the edge. Finally, the **mclose** object performs morphological closing to enhance the edge output. The code is shown below.

```
[pixGray,ctrlGray] = step(rgb2gray,pixIn,ctrlIn); % Convert RGB to grayscale
[pixDenoise,ctrlDenoise] = step(medfil,pixGray,ctrlGray); % Remove noise
[pixEdge,ctrlEdge] = step(sobel,pixDenoise,ctrlDenoise); % Detect edges
[pixClose,ctrlClose] = step(mclose,pixEdge,ctrlEdge); % Apply closing
```

### Full-Frame Reference Design

When designing a complex pixel-stream video processing algorithm, it is a good practice to develop a parallel reference design using functions from the Image Processing Toolbox™. These functions process full image frames. Such a reference design helps verify the implementation of the pixel-stream design by comparing the output image from the full-frame reference design to the output of the pixel-stream design.

The function **EnhancedEdgeDetectionHDLReference.m** contains a similar set of four functions as in the **EnhancedEdgeDetectionHDLDesign.m**. The key difference is that the functions from Image Processing Toolbox process full-frame data.

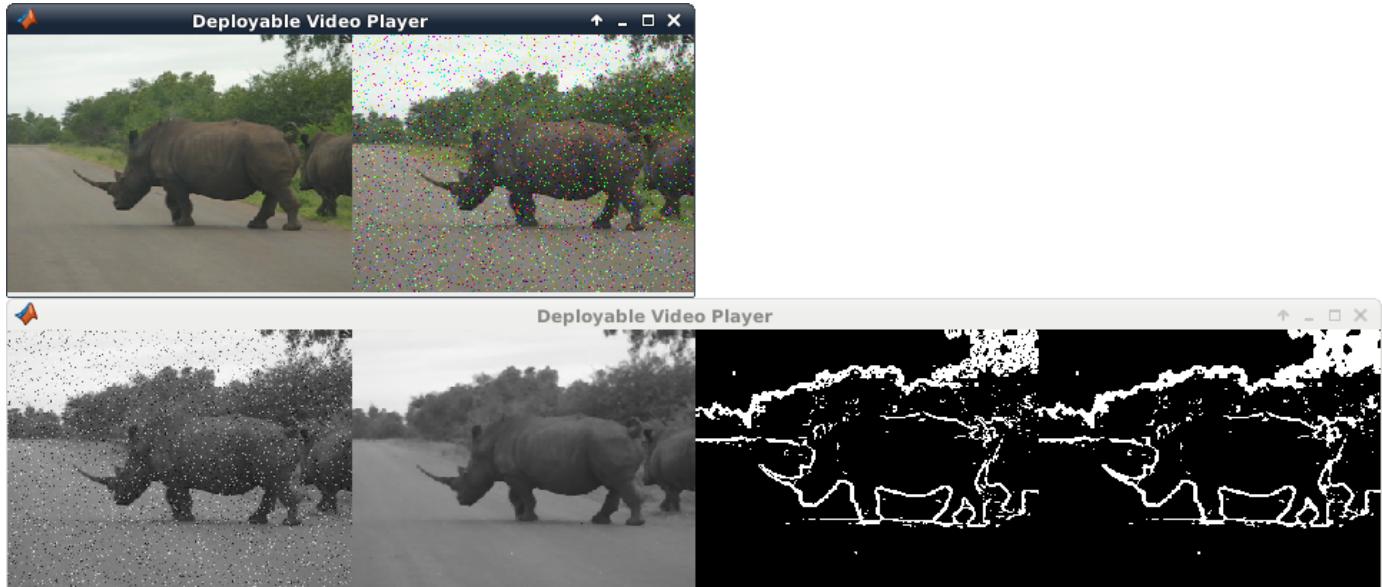
Due to the implementation difference between **edge** function and **visionhdl.EdgeDetector** System object, reference and design output are considered matching if **frmOut** and **frmRef** differ in no greater than 20 pixels.

### Create MEX File and Simulate the Design

Generate and execute the MEX file.

```
codegen( 'EnhancedEdgeDetectionHDLTestBench' );
EnhancedEdgeDetectionHDLTestBench_mex;
```

```
frame 1: reference and design output differ in more than 20 pixels.
```



The upper video player displays the original color video on the left, and its noisy version after adding salt and pepper noise on the right. The lower video player, from left to right, represents: the grayscale image after color space conversion, the de-noised version after median filter, the edge output after edge detection, and the enhanced edge output after morphological closing operation.

Note that in the lower video chain, only the enhanced edge output (right-most video) is generated from pixel-stream design. The other three are the intermediate videos from the full-frame reference design. To display all of the four videos from the pixel-stream design, you would have written the design file to output four sets of pixels and control signals, and instantiated three more **visionhdl.PixelsToFrame** objects to convert the three intermediate pixel streams back to frames. For the sake of simulation speed and the clarity of the code, this example does not implement the intermediate pixel-stream displays.

### HDL Code Generation

To create a new project, enter the following command in the temporary folder

```
coder -hdlcoder -new EnhancedEdgeDetectionProject
```

Then, add the file 'EnhancedEdgeDetectionHDLDesign.m' to the project as the MATLAB Function and 'EnhancedEdgeDetectionHDLTestBench.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

## Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow

This example shows how to generate HDL code from a MATLAB design implementing the Sobel edge detection algorithm.

### Set Up Example

Run the following code to set up the design:

```
design_name = 'mlhdlc_sobel.m';
testbench_name = 'mlhdlc_sobel_tb.m';

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];

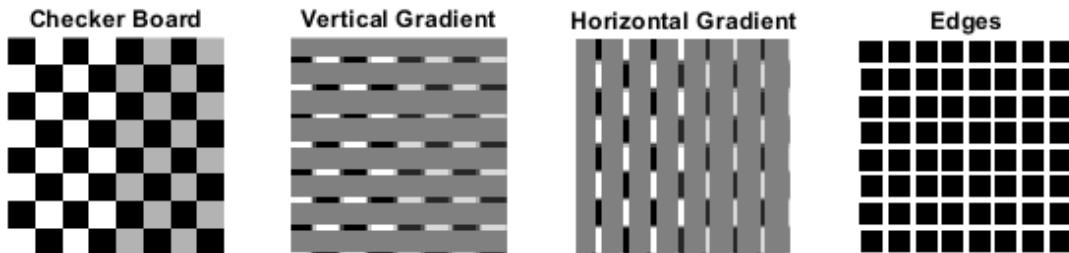
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, design_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, testbench_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_img_stop_sign.gif'), mlhdlc_temp_dir);
```

### Simulate the Design

It is a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sobel_tb;
```



### Create a New HDL Coder Project

Run the following command to create the HDL code generation project.

```
coder -hdlcoder -new cosim_fil_sobel
```

## Specify the Design and the Test Bench

- 1 Drag the file "mlhdlc\_sobel.m" from the Current Folder Browser into the Entry Points tab of the HDL Coder UI, under the "MATLAB Function" section.
- 2 Under the newly added "mlhdlc\_sobel\_tb.m" file, specify the data type of input argument "data\_in" as "double (1 x 1)"
- 3 Drag the file 'mlhdlc\_sobel\_tb.m' into the HDL Coder UI, under "MATLAB Test Bench" section.



## Generate HDL Code

- 1 Click "Workflow Advisor".
- 2 Right click on the "Code Generation" step in Workflow Advisor.
- 3 Choose option "Run to selected task" to run all steps from the beginning of the workflow through to HDL code generation.

## Verify Generated HDL Code with Cosimulation

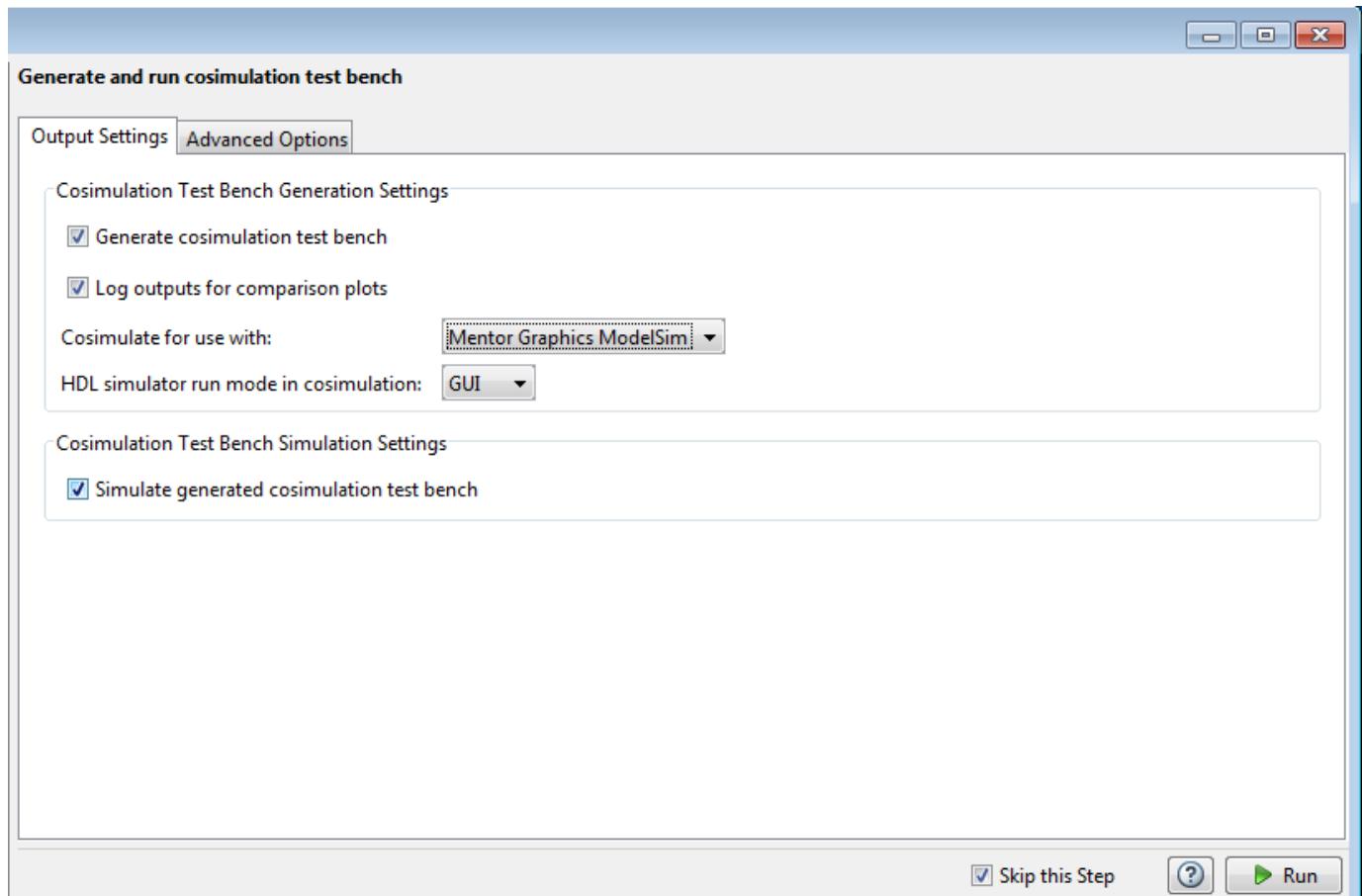
To run this step, you must have one of the HDL simulators supported by HDL Verifier. See {Supported EDA Tools}. You may skip this step if you do not.

1. Select the "Generate cosimulation test bench" option.
2. Select the "Log outputs for comparison plots" option. This option generates the plotting of the HDL simulator output, the reference MATLAB algorithm output, and the differences between them.
3. For "Cosimulate for use with:", select your HDL simulator. The HDL simulator executable must be on your system path.
4. To view the waveform in the HDL simulator, select "GUI" mode in the "HDL simulator run mode in cosimulation" list.

5. Select "Simulate generated cosimulation test bench".

6. Click "Run".

When the simulation is complete, check the comparison plots. There should be no mismatch between the HDL simulator output and the reference MATLAB algorithm output.



### Verify Generated HDL Code with FPGA-in-the-Loop

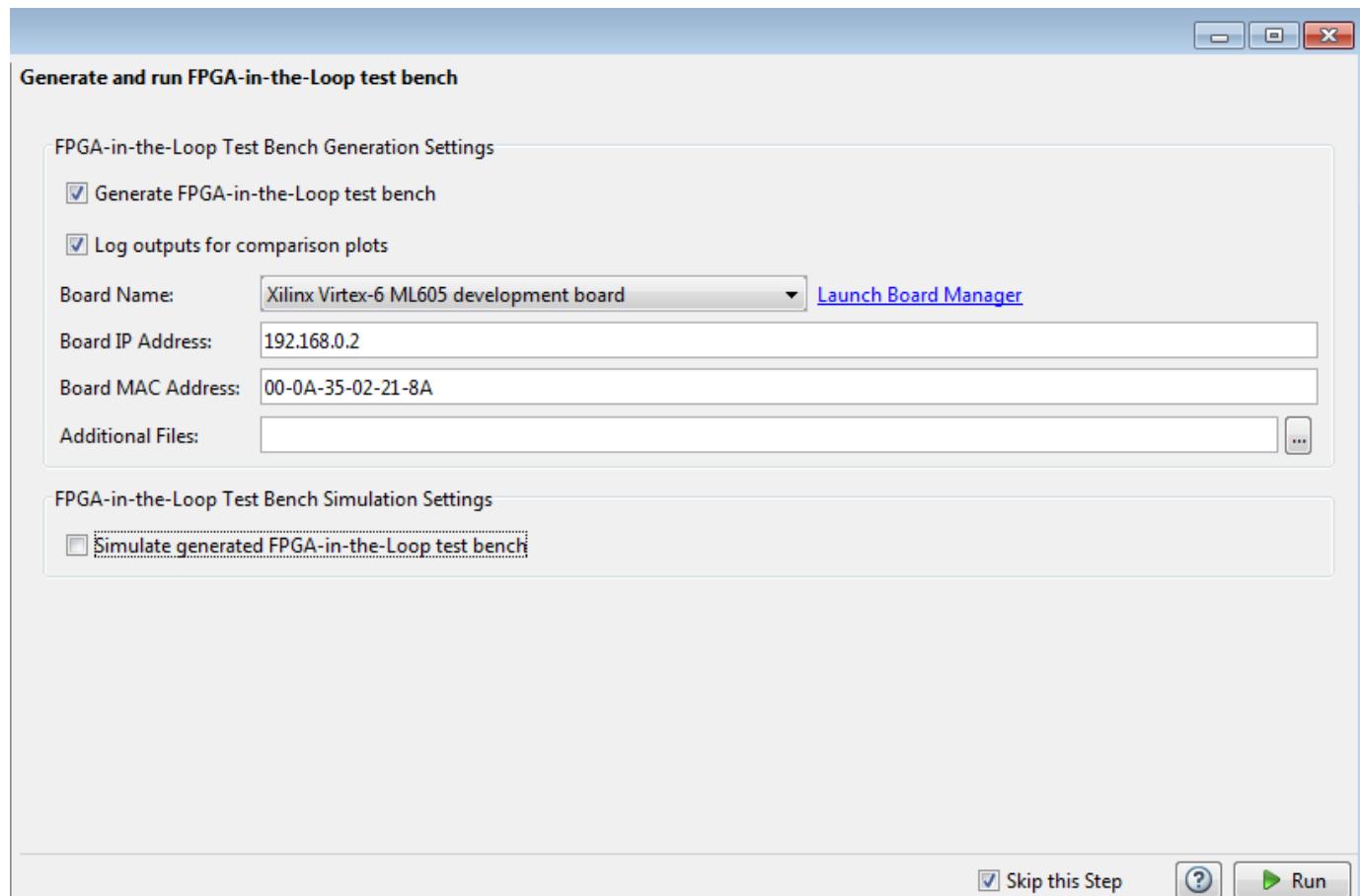
To run this step, you must have one of the supported FPGA boards (see {Supported EDA Tools}). Refer to here for additional setup instructions required for FPGA-in-the-Loop.

In the "Verify with FPGA-in-the-Loop" step, perform the following steps:

1. Select the "Generate FPGA-in-the-Loop test bench" option.
2. Select the "Log outputs for comparison plots" option. This option generates the plotting of the FPGA output, the reference MATLAB algorithm output, and the differences between them.
3. Select your FPGA board from the "Cosimulate for use with:" list. If your board is not on the list, select one of the following options:
  - "Get more boards..." to download the FPGA board support package(s) (this option starts the Support Package Installer)

- "Create custom board..." to create the FPGA board definition file for your particular FPGA board (this option starts the New FPGA Board Manager).
4. Ethernet connection only: Enter your Ethernet connection information in the "Board IP Address" and "Board MAC Address:" fields. Leave the "Additional Files" field empty.
5. Select "Simulate generated FPGA-in-the-Loop test bench".
6. Click "Run".

When the simulation is complete, check the comparison plots. There should be no mismatch between the FPGA output and the reference MATLAB algorithm output.



This ends the Verify Sobel Edge Detection Algorithm in MATLAB-to-HDL Workflow example.



# MATLAB Best Practices and Design Patterns for HDL Code Generation

---

- “Model a Counter for HDL Code Generation” on page 3-2
- “Model a State Machine for HDL Code Generation” on page 3-4
- “Generate Hardware Instances For Local Functions” on page 3-8
- “Implement RAM Using MATLAB Code” on page 3-10

# Model a Counter for HDL Code Generation

## In this section...

[“MATLAB Counter” on page 3-2](#)

[“MATLAB Code for the Counter” on page 3-2](#)

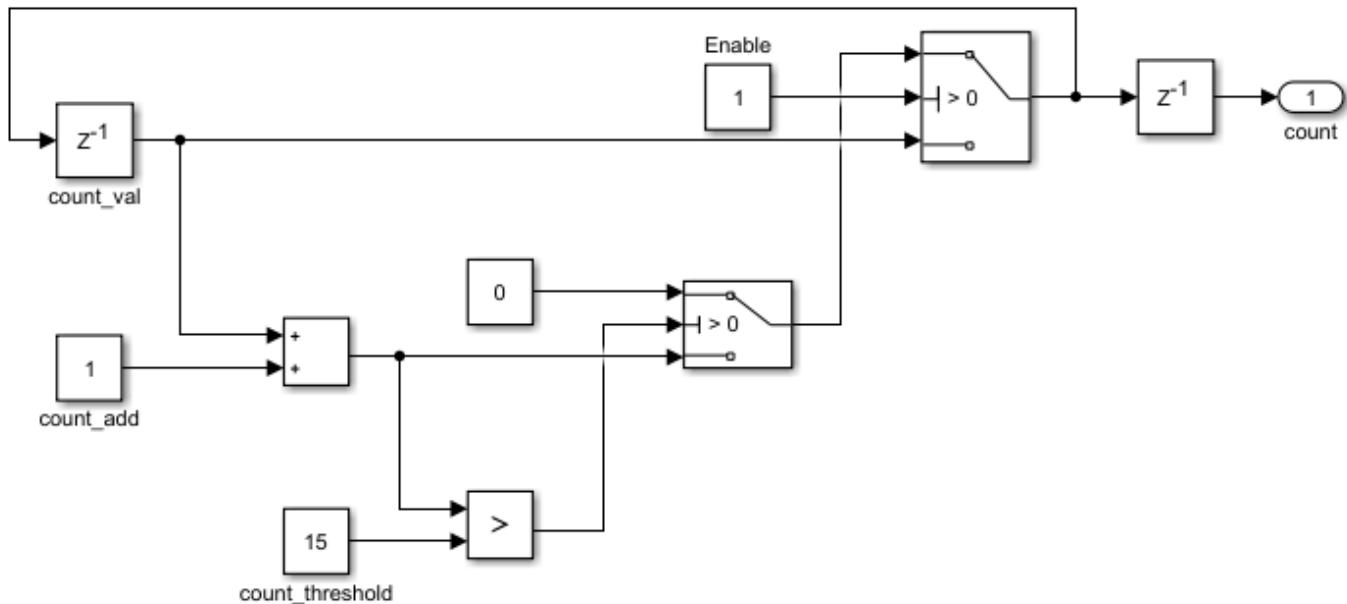
This design pattern shows a MATLAB example of a counter, which is suitable for HDL code generation.

## MATLAB Counter

This design pattern demonstrates two best practices for writing MATLAB code for HDL code generation:

- Initialize persistent variables to a specific value. In this example, an `if` statement and the `isempty` function initialize the persistent variable. If the persistent variable is not initialized then HDL code cannot be generated.
  - Inside a function, read persistent variables before they are modified, in order for the persistent variables to be inferred as registers.

This Simulink model illustrates the counter modeled in this example.



To learn how to model the counter in Simulink, see “Create HDL-Compatible Simulink Model”.

## MATLAB Code for the Counter

The function `mlhdlc_counter` is a behavioral model of a four bit synchronous up counter. The input signal, `enable_ctr`, triggers the value of the count register, `count_val`, to increase by one. The counter continues to increase by one each time the input is nonzero, until the count reaches a limit of

15. After the counter reaches this limit, the counter returns to zero. A persistent variable, which is initialized to zero, represents the current value of the count. Two `if` statements determine the value of the count based on the input.

The following section of code defines the `mldhlc_counter` function.

```
%#codegen
function count = mldhlc_counter(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;

    %limit to four bits
    if count_val>15
        count_val=0;
    end
end

count=count_val;

end
```

## See Also

`codegen` | `coder.HdlConfig`

## More About

- “Model a State Machine for HDL Code Generation” on page 3-4
- “Implement RAM Using MATLAB Code” on page 3-10
- “Supported MATLAB Data Types, Operators, and Control Flow Statements” on page 1-4

## Model a State Machine for HDL Code Generation

### In this section...

"MATLAB Code for the Mealy State Machine" on page 3-4

"MATLAB Code for the Moore State Machine" on page 3-5

The following design pattern shows MATLAB examples of Mealy and Moore state machines which are suitable for HDL code generation.

The MATLAB code in these models demonstrates best practices for writing MATLAB models for HDL code generation.

- With a `switch` block, use the `otherwise` statement to ensure that the model accounts for all conditions. If the model does not cover all conditions, the generated HDL code can contain errors.
- To designate the states in a state machine, use variables with numerical values.

### MATLAB Code for the Mealy State Machine

In a Mealy state machine, the output depends on the state and the input. In a Moore state machine, the output depends only on the state.

The following MATLAB code defines the `mlhdlc_fsm_mealy` function. A persistent variable represents the current state. A `switch` block uses the current state and input to determine the output and new state. In each `case` in the `switch` block, an `if-else` statement calculates the new state and output.

```
%#codegen
function Z = mlhdlc_fsm_mealy(A)
% Mealy State Machine

% y = f(x,u) :
% all actions are condition actions and
% outputs are function of state and input

% define states
S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;

persistent current_state;
if isempty(current_state)
    current_state = S1;
end

% switch to new state based on the value state register
switch (current_state)

    case S1,
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S1;
        else
            Z = false;
            current_state = S2;
        end
    case S2,
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S2;
        else
            Z = false;
            current_state = S3;
        end
    case S3,
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S3;
        else
            Z = false;
            current_state = S4;
        end
    case S4,
        % value of output 'Z' depends both on state and inputs
        if (A)
            Z = true;
            current_state = S4;
        else
            Z = false;
            current_state = S1;
        end
end
```

```

else
    Z = false;
    current_state = S2;
end

case S2,
    if (A)
        Z = false;
        current_state = S3;
    else
        Z = true;
        current_state = S2;
    end

case S3,
    if (A)
        Z = false;
        current_state = S4;
    else
        Z = true;
        current_state = S1;
    end

case S4,
    if (A)
        Z = true;
        current_state = S1;
    else
        Z = false;
        current_state = S3;
    end

otherwise,
    Z = false;
end

```

## MATLAB Code for the Moore State Machine

The following MATLAB code defines the `mlhdlc_fsm_moore` function. A persistent variable represents the current state, and a switch block uses the current state to determine the output and new state. In each case in the switch block, an if-else statement calculates the new state and output. The value of the state is represented by numerical variables.

```

%#codegen
function Z = mlhdlc_fsm_moore(A)
% Moore State Machine

% y = f(x) :
% all actions are state actions and
% outputs are pure functions of state only

% define states
S1 = 0;

```

```
S2 = 1;
S3 = 2;
S4 = 3;

% using persistent keyword to model state registers in hardware
persistent curr_state;
if isempty(curr_state)
    curr_state = S1;
end

% switch to new state based on the value state register
switch (curr_state)

    case S1,
        % value of output 'Z' depends only on state and not on inputs
        Z = true;

        % decide next state value based on inputs
        if (~A)
            curr_state = S1;
        else
            curr_state = S2;
        end

    case S2,
        Z = false;

        if (~A)
            curr_state = S1;
        else
            curr_state = S3;
        end

    case S3,
        Z = false;

        if (~A)
            curr_state = S2;
        else
            curr_state = S4;
        end

    case S4,
        Z = true;
        if (~A)
            curr_state = S3;
        else
            curr_state = S1;
        end

    otherwise,
        Z = false;
end
```

**See Also**

`codegen` | `coder.HdlConfig`

**More About**

- “Functions Supported for HDL Code Generation” on page 1-2
- “Model a Counter for HDL Code Generation” on page 3-2
- “Implement RAM Using MATLAB Code” on page 3-10

## Generate Hardware Instances For Local Functions

### In this section...

"MATLAB Local Functions" on page 3-8

"MATLAB Code for mlhdlc\_two\_counters.m" on page 3-8

The following example shows how to use local functions in MATLAB, so that each execution of a local function corresponds to a separate hardware module in the generated HDL code.

### MATLAB Local Functions

This example demonstrates best practices for writing local functions in MATLAB code that is suitable for HDL code generation.

- If your MATLAB code executes a local function multiple times, the generated HDL code does not necessarily instantiate multiple hardware modules. Rather than instantiating multiple hardware modules, multiple calls to a function typically update the state variable.
- If you want the generated HDL code to contain multiple hardware modules corresponding to each execution of a local function, specify two different local functions with the same code but different function names. If you want to avoid code duplication, consider using System objects to implement the behavior in the function, and instantiate the System object multiple times.
- If you want to specify a separate HDL file for each local function in the MATLAB code, in the Workflow Advisor, on the **Advanced** tab in the HDL Code Generation section, select **Generate instantiable code for functions**.

### MATLAB Code for `mlhdlc_two_counters.m`

This function creates two counters and adds the output of these counters. To create two counters, there are two local functions with identical code, `counter` and `counter2`. The main method calls each of these local functions once. If the function were to call the `counter` function twice, separate hardware modules for the counters would not be generated in the HDL code.

```
%#codegen
function total_count = mlhdlc_two_counters(a,b)

%This function contains two different local functions with identical
%counters and calls each counter once.

total_count1=counter(a);
total_count2=counter2(b);
total_count=total_count1+total_count2;

function count = counter(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
```

```
end

%counting up
if enable_ctr
    count_val=count_val+1;
end

%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;

function count = counter2(enable_ctr)
%four bit synchronous up counter

%persistent variable for the state
persistent count_val;
if isempty(count_val)
    count_val = 0;
end

%counting up
if enable_ctr
    count_val=count_val+1;
end

%limit from four bits
if count_val>15
    count_val=0;
end

count=count_val;
```

## See Also

[codegen](#) | [coder.HdlConfig](#)

## More About

- “Functions Supported for HDL Code Generation” on page 1-2
- “Model a Counter for HDL Code Generation” on page 3-2
- “Model a State Machine for HDL Code Generation” on page 3-4
- “RAM Mapping Comparison for MATLAB Code” on page 8-11

## Implement RAM Using MATLAB Code

### In this section...

"Implement RAM Using a Persistent Array or System object Properties" on page 3-10

"Implement RAM Using `hdl.RAM`" on page 3-11

You can write MATLAB code that maps to RAM during HDL code generation by using:

- Persistent arrays or private properties in a user-defined System object.
- `hdl.RAM` System objects.

The following examples model the same line delay in MATLAB. The line delay uses memory in a ring structure. Data is written to one location and read from another location in such a way that the data written is read after a delay of a specific number of cycles. The RAM read address is generated by a counter. The write address is generated by adding a constant value to the read address.

### Implement RAM Using a Persistent Array or System object Properties

This example shows a line delay that implements the RAM behavior using a persistent array with the function `mlhdlc_hdrlam_persistent`. Changing a specific value in the persistent array is equivalent to writing to the RAM. Accessing a specific value in the array is equivalent to reading from the RAM.

You can implement RAM by using user-defined System object private properties in the same way.

```
%#codegen
function data_out = mlhdlc_hdrlam_persistent(data_in)

persistent hRam;
if isempty(hRam)
    hRam = zeros(128,1);
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 1;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
%ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM

hRam(ramWriteAddr)=ramWriteData;
ramRdDout=hRam(ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;
```

```
data_out = ramRdDout;
```

## Implement RAM Using `hdl.RAM`

This example shows a line delay that implements the RAM behavior using `hdl.RAM` with the function, `mlhdlc_hdlram_sysobj`. In this function, the `step` method of the `hdl.RAM` System object reads and writes to specific locations in `hRam`. Code generation from `hdl.RAM` has the same restrictions as code generation from other System objects. For details, see “Limitations of HDL Code Generation for System Objects” on page 1-14.

```
%#codegen
function data_out = mlhdlc_hdlram_sysobj(data_in)
persistent hRam;
if isempty(hRam)
    hRam = hdl.RAM('RAMType', 'Dual port');
end

% read address counter
persistent rdAddrCtr;
if isempty(rdAddrCtr)
    rdAddrCtr = 0;
end

% ring counter length
ringCtrLength = 10;
ramWriteAddr = rdAddrCtr + ringCtrLength;

ramWriteData = data_in;
ramWriteEnable = true;

ramReadAddr = rdAddrCtr;

% execute single step of RAM
[~,ramRdDout] = step(hRam,ramWriteData,ramWriteAddr, ...
    ramWriteEnable,ramReadAddr);

rdAddrCtr = rdAddrCtr + 1;

data_out = ramRdDout;
```

## See Also

`codegen` | `coder.HdlConfig`

## More About

- “Functions Supported for HDL Code Generation” on page 1-2
- “Model a Counter for HDL Code Generation” on page 3-2
- “Model a State Machine for HDL Code Generation” on page 3-4
- “RAM Mapping Comparison for MATLAB Code” on page 8-11
- “Map Persistent Arrays and `dsp.Delay` to RAM” on page 8-8



# Fixed-Point Conversion

---

- “Specify Type Proposal Options” on page 4-2
- “Log Data for Histogram” on page 4-5
- “View and Modify Variable Information” on page 4-7
- “Automated Fixed-Point Conversion” on page 4-9
- “Custom Plot Functions” on page 4-23
- “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-24
- “Inspecting Data Using the Simulation Data Inspector” on page 4-29
- “Enable Plotting Using the Simulation Data Inspector” on page 4-31
- “Replacing Functions Using Lookup Table Approximations” on page 4-32
- “Replace a Custom Function with a Lookup Table” on page 4-33
- “Replace the `exp` Function with a Lookup Table” on page 4-39
- “Data Type Issues in Generated Code” on page 4-45
- “Working with Fixed-Point Code” on page 4-47
- “Floating-Point to Fixed-Point Conversion” on page 4-49
- “Fixed-Point Type Conversion and Refinement” on page 4-59
- “Working with Generated Fixed-Point Files” on page 4-66
- “Fixed-Point Type Conversion and Derived Ranges” on page 4-72
- “Generate HDL-compatible lookup table function replacements using ‘`coder.approximate`’” on page 4-77

## Specify Type Proposal Options

Basic Type Proposal Settings	Values	Description
Fixed-point type proposal mode	Propose fraction lengths for specified word length	Use the specified word length for data type proposals and propose the minimum fraction lengths to avoid overflows.
	Propose word lengths for specified fraction length (default)	Use the specified fraction length for data type proposals and propose the minimum word lengths to avoid overflows.
Default word length	14 (default)	Default word length to use when <b>Fixed-point type proposal mode</b> is set to Propose fraction lengths for specified word lengths
Default fraction length	4 (default)	Default fraction length to use when <b>Fixed-point type proposal mode</b> is set to Propose word lengths for specified fraction lengths

Advanced Type Proposal Settings	Values	Description
<b>Note</b> Manually-entered static ranges always take precedence over simulation ranges.	ignore simulation ranges	Propose data types based on derived ranges.
	ignore derived ranges	Propose data types based on simulation ranges.
	use all collected data (default)	Propose data types based on both simulation and derived ranges.
Propose target container types	Yes	Propose data type with the smallest word length that can represent the range and is suitable for C code generation ( 8,16,32, 64 ... ). For example, for a variable with range [0..7], propose a word length of 8 rather than 3.
	No (default)	Propose data types with the minimum word length needed to represent the value.
Optimize whole numbers	No	Do not use integer scaling for variables that were whole numbers during simulation.
	Yes (default)	Use integer scaling for variables that were whole numbers during simulation.

<b>Advanced Type Proposal Settings</b>	<b>Values</b>	<b>Description</b>
Signedness	Automatic (default)	Proposes signed and unsigned data types depending on the range information for each variable.
	Signed	Propose signed data types.
	Unsigned	Propose unsigned data types.
Safety margin for sim min/max (%)	0 (default)	<p>Specify safety factor for simulation minimum and maximum values.</p> <p>The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. For example, a value of 55 specifies that you want a range at least 55 percent larger. A value of -15 specifies that a range up to 15 percent smaller is acceptable.</p>
Search paths	' ' (default)	Add paths to the list of paths to search for MATLAB files. Separate list items with a semicolon.

<b>fimath Settings</b>	<b>Values</b>	<b>Description</b>
Rounding method	Ceiling	Specify the <b>fimath</b> properties for the generated fixed-point data types.
	Convergent	
	Floor (default)	
	Nearest	
	Round	
	Zero	
Overflow action	Saturate	The default fixed-point math properties use the <b>Floor</b> rounding and <b>Wrap</b> overflow. These settings generate the most efficient code but might cause problems with overflow.
	Wrap (default)	
Product mode	FullPrecision (default)	After code generation, if required, modify these settings to optimize the generated code, or example, avoid overflow or eliminate bias, and then rerun the verification.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	
Sum mode	FullPrecision (default)	For more information on <b>fimath</b> properties, see “ <b>fimath Object Properties</b> ”.
	KeepLSB	
	KeepMSB	
	SpecifyPrecision	

<b>Generated File Settings</b>	<b>Value</b>	<b>Description</b>
Generated fixed-point file name suffix	_fixpt (default)	Specify the suffix to add to the generated fixed-point file names.

<b>Plotting and Reporting Settings</b>	<b>Values</b>	<b>Description</b>
Custom plot function	' ' (default)	Specify the name of a custom plot function to use for comparison plots.
Plot with Simulation Data Inspector	No (default)	Specify whether to use the Simulation Data Inspector for comparison plots.
	Yes	
Highlight potential data type issues	No (default)	Specify whether to highlight potential data types in the generated html report. If this option is turned on, the report highlights single-precision, double-precision, and expensive fixed-point operation usage in your MATLAB code.
	Yes	

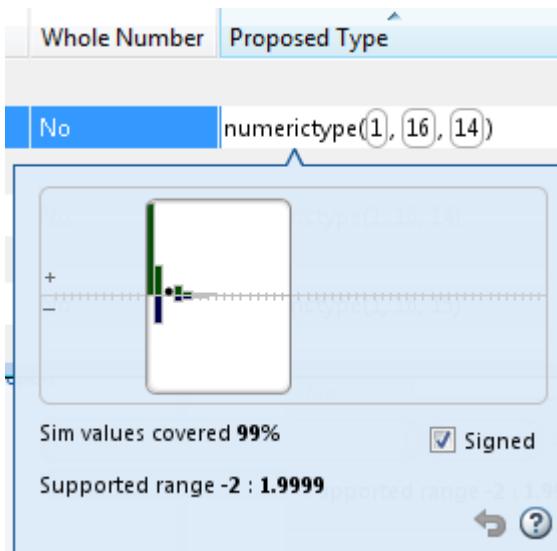
## Log Data for Histogram

To log data for histograms:

- 1 In the Fixed-Point Conversion window, click **Run Simulation** and select Log data for histogram, and then click the Run Simulation button.

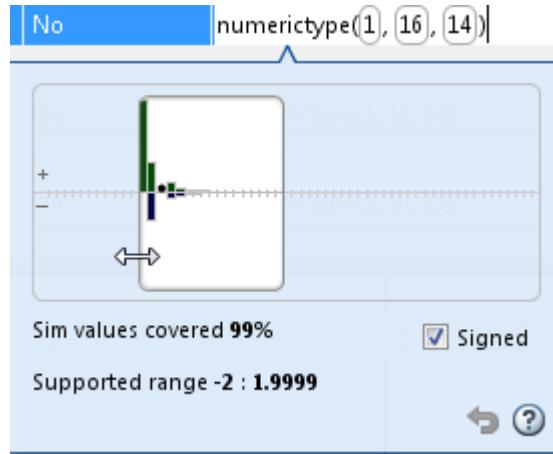
The simulation runs and the simulation minimum and maximum ranges are displayed on the **Variables** tab. Using the simulation range data, the software proposes fixed-point types for each variable based on the default type proposal settings, and displays them in the **Proposed Type** column.

- 2 To view a histogram for a variable, click the variable's **Proposed Type** field.



- 3 You can view the effect of changing the proposed data types by:

- Selecting and dragging the white bounding box in the histogram window. This action does not change the word length of the proposed data type, but modifies the position of the binary point within the word so that the fraction length of the proposed data type changes.
- Selecting and dragging the left edge of the bounding box to increase or decrease the word length. This action does not change the fraction length or the position of the binary point.



- Selecting and dragging the right edge to increase or decrease the fraction length of the proposed data type. This action does not change the position of the binary point. The word length changes to accommodate the fraction length.
- Selecting or clearing **Signed**. Clear **Signed** to ignore negative values.

Before committing changes, you can revert to the types proposed by the automatic conversion by clicking .

# View and Modify Variable Information

## View Variable Information

In the Fixed-Point Conversion tool, you can view information about the variables in the MATLAB functions. To view information about the variables for the selected function, use the **Variables** tab or pause over a variable in the code window. For more information, see “Viewing Variables” on page 4-14.

You can view the variable information:

- **Variable**

Variable name. Variables are classified and sorted as inputs, outputs, persistent, or local variables.

- **Type**

The original size, type, and complexity of each variable.

- **Sim Min**

The minimum value assigned to the variable during simulation.

- **Sim Max**

The maximum value assigned to the variable during simulation.

To search for a variable in the MATLAB code window and on the **Variables** tab, use **Ctrl+F**.

## Modify Variable Information

If you modify variable information, the app highlights the modified values using bold text. You can modify the following fields:

- **Static Min**

You can enter a value for **Static Min** into the field or promote **Sim Min** information. See “Promote Sim Min and Sim Max Values” on page 4-8.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Static Max**

You can enter a value for **Static Max** into the field or promote **Sim Max** information. See “Promote Sim Min and Sim Max Values” on page 4-8.

Editing this field does not trigger static range analysis, but the app uses the edited values in subsequent analyses.

- **Whole Number**

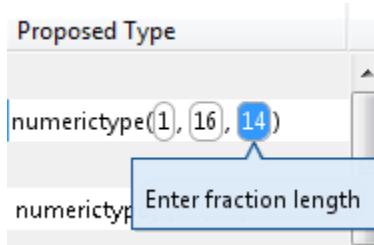
The app uses simulation data to determine whether the values assigned to a variable during simulation were always integers. You can manually override this field.

Editing this field does not trigger static range analysis, but the app uses the edited value in subsequent analyses.

- **Proposed Type**

You can modify the signedness, word length, and fraction length settings individually:

- On the **Variables** tab, modify the value in the **ProposedType** field.



- In the code window, select a variable, and then modify the **Proposed Type** field.

If you selected to log data for a histogram, the histogram dynamically updates to reflect the modifications to the proposed type. You can also modify the proposed type in the histogram, see "Histogram" on page 4-19.

## Revert Changes

- To clear results and revert edited values, right-click the **Variables** tab and select **Reset entire table**.
- To revert the type of a selected variable to the type computed by the app, right-click the field and select **Undo changes**.
- To revert changes to variables, right-click the field and select **Undo changes for all variables**.
- To clear a static range value, right-click an edited field and select **Clear this static range**.
- To clear manually entered static range values, right-click anywhere on the **Variables** tab and select **Clear all manually entered static ranges**.

## Promote Sim Min and Sim Max Values

With the app, you can promote simulation minimum and maximum values to static minimum and maximum values. This capability is useful if you have not specified static ranges and you have simulated the model with inputs that cover the full intended operating range.

To copy:

- A simulation range for a selected variable, select a variable, right-click, and then select **Copy sim range**.
- Simulation ranges for top-level inputs, right-click the Static Min or Static Max column, and then select **Copy sim ranges for all top-level inputs**.
- Simulation ranges for persistent variables, right-click the Static Min or Static Max column, and then select **Copy sim ranges for all persistent variables**.

# Automated Fixed-Point Conversion

## In this section...

- "License Requirements" on page 4-9
- "Automated Fixed-Point Conversion Capabilities" on page 4-9
- "Code Coverage" on page 4-10
- "Proposing Data Types" on page 4-12
- "Locking Proposed Data Types" on page 4-13
- "Viewing Functions" on page 4-14
- "Viewing Variables" on page 4-14
- "Histogram" on page 4-19
- "Function Replacements" on page 4-20
- "Validating Types" on page 4-21
- "Testing Numerics" on page 4-21
- "Detecting Overflows" on page 4-21

## License Requirements

Fixed-point conversion requires the following licenses:

- Fixed-Point Designer
- MATLAB Coder™

## Automated Fixed-Point Conversion Capabilities

You can convert floating-point MATLAB code to fixed-point code using the Fixed-Point Conversion tool in HDL Coder projects. You can choose to propose data types based on simulation range data, derived (also known as static) range data, or both.

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see "Locking Proposed Data Types" on page 4-13.

For a list of supported MATLAB features and functions, see "MATLAB Language Features Supported for Automated Fixed-Point Conversion".

During fixed-point conversion, you can:

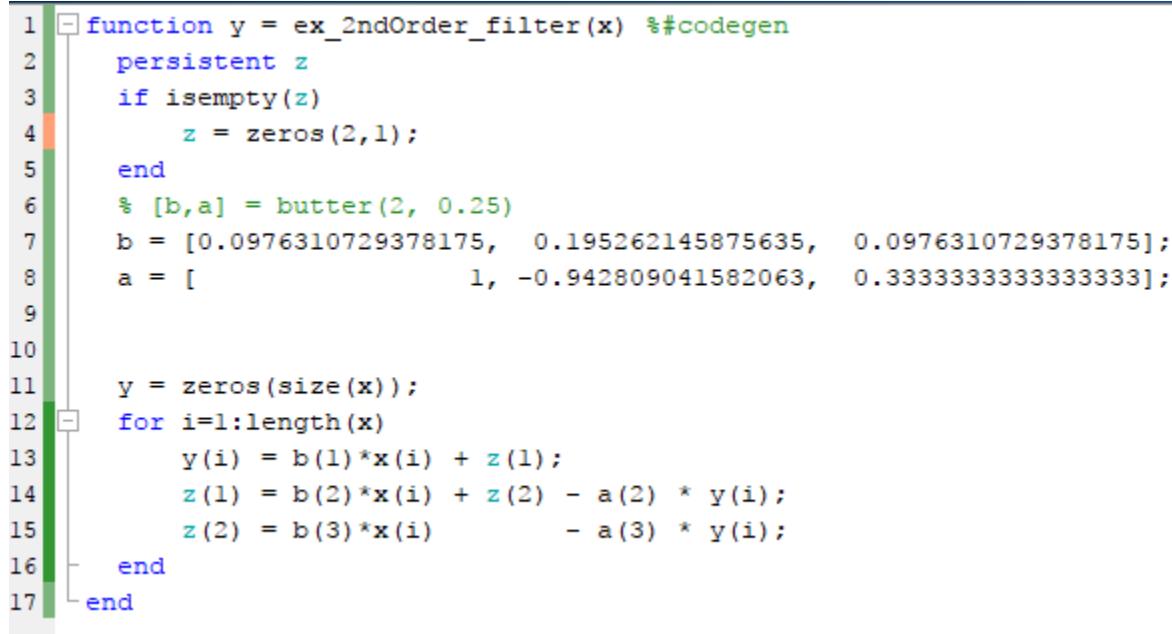
- Verify that your test files cover the full intended operating range of your algorithm using code coverage results.
- Propose fraction lengths based on default word lengths.
- Propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.

- Validate that you can build your project with the proposed data types.
- Test numerics by running the test bench with the fixed-point types applied.
- View a histogram of bits used by each variable.
- Detect overflows.

## Code Coverage

By default, the Fixed-Point Conversion tool shows code coverage results. Your test files must exercise the algorithm over its full operating range so that the simulation ranges are accurate. The quality of the proposed fixed-point data types depends on how well the test files cover the operating range of the algorithm with the accuracy that you want. Reviewing code coverage results helps you verify that your test files are exercising the algorithm adequately. If the code coverage is inadequate, modify the test files or add more test files to increase coverage. If you simulate multiple test files in one run, the tool displays cumulative coverage. However, if you specify multiple test files but run them one at a time, the tool displays the coverage of the file that ran last.

The tool displays a color-coded coverage bar to the left of the code.



A screenshot of a MATLAB code editor showing the function `ex_2ndOrder_filter`. The code is annotated with color-coded bars on the left margin to indicate coverage status. The bars are green for fully covered lines, orange for partially covered lines, and grey for unexecuted lines. The code itself is as follows:

```
1 function y = ex_2ndOrder_filter(x) %#codegen
2 persistent z
3 if isempty(z)
4     z = zeros(2,1);
5 end
6 % [b,a] = butter(2, 0.25)
7 b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8 a = [
9                 1, -0.942809041582063, 0.3333333333333333];
10
11 y = zeros(size(x));
12 for i=1:length(x)
13     y(i) = b(1)*x(i) + z(1);
14     z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15     z(2) = b(3)*x(i) - a(3) * y(i);
16 end
17 end
```

This table describes the color coding.

Coverage Bar Color	Indicates
Green	<p>One of the following situations:</p> <ul style="list-style-type: none"> <li>The entry-point function executes multiple times and the code executes more than one time.</li> <li>The entry-point function executes one time and the code executes one time.</li> </ul> <p>Different shades of green indicate different ranges of line execution counts. The darkest shade of green indicates the highest range.</p>
Orange	The entry-point function executes multiple times, but the code executes one time.
Red	Code does not execute.

When you pause over the coverage bar, the color highlighting extends over the code. For each section of code, the app displays the number of times that section executes.

```

1 function y = ex_2ndOrder_filter(x) %#codegen
2 persistent z
3 if isempty(z)
4     z = zeros(2,1);                                1 calls
5 end
6 % [b,a] = butter(2, 0.25)
7 b = [0.0976310729378175, 0.195262145875635, 0.0976310729378175];
8 a = [1, -0.942809041582063, 0.3333333333333333];
9
10
11 y = zeros(size(x));
12 for i=1:length(x)                                768 calls
13     y(i) = b(1)*x(i) + z(1);
14     z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
15     z(2) = b(3)*x(i) - a(3) * y(i);
16 end
17

```

To verify that your test files are testing your algorithm over the intended operating range, review the code coverage results.

Coverage Bar Color	Action
Green	If you expect sections of code to execute more frequently than the coverage shows, either modify the MATLAB code or the test files.
Orange	This behavior is expected for initialization code, for example, the initialization of persistent variables. If you expect the code to execute more than one time, either modify the MATLAB code or the test files.

Coverage Bar Color	Action
Red	If the code that does not execute is an error condition, this behavior is acceptable. If you expect the code to execute, either modify the MATLAB code or the test files. If the code is written conservatively and has upper and lower boundary limits, and you cannot modify the test files to reach this code, add static minimum and maximum values. See “Computing Derived Ranges” on page 4-13.

Code coverage is on by default. Turn it off only after you have verified that you have adequate test file coverage. Turning off code coverage can speed up simulation. To turn off code coverage, in the Fixed-Point Conversion tool:

- 1 Click **Run Simulation**.
- 2 Clear `Show code coverage`.

## Proposing Data Types

The Fixed-Point Conversion tool proposes fixed-point data types based on computed ranges and the word length or fraction length setting. The computed ranges are based on simulation range data, derived range data, or both. If you run a simulation and compute derived ranges, the conversion tool merges the simulation and derived ranges.

---

**Note** You cannot propose data types based on derived ranges for MATLAB classes.

---

You can manually enter static ranges. These manually-entered ranges take precedence over simulation ranges and the tool uses them when proposing data types. In addition, you can modify and lock the proposed type so that the tool cannot change it. For more information, see “Locking Proposed Data Types” on page 4-13.

### Running a Simulation

When you open the Fixed-Point Conversion tool, the tool generates an instrumented MEX function for your MATLAB design. If the build completes without errors, the tool displays compiled information (type, size, complexity) for functions and variables in your code. To navigate to local functions, click the **Functions** tab. If build errors occur, the tool provides error messages that link to the line of code that caused the build issues. You must address these errors before running a simulation. Use the link to navigate to the offending line of code in the MATLAB editor and modify the code to fix the issue. If your code uses functions that are not supported for fixed-point conversion, the tool displays them on the **Function Replacements** tab. See “Function Replacements” on page 4-20.

Before running a simulation, specify the test bench that you want to run. When you run a simulation, the tool runs the test bench, calling the instrumented MEX function. If you modify the MATLAB design code, the tool automatically generates an updated MEX function before running the test bench.

If the test bench runs successfully, the simulation minimum and maximum values and the proposed types are displayed on the **Variables** tab. If you manually enter static ranges for a variable, the manually-entered ranges take precedence over the simulation ranges. If you manually modify the proposed types by typing or using the histogram, the data types are locked so that the tool cannot modify them.

If the test bench fails, the errors are displayed on the **Simulation Output** tab.

The test bench should exercise your algorithm over its full operating range. The quality of the proposed fixed-point data types depends on how well the test bench covers the operating range of the algorithm with the desired accuracy.

Optionally, you can select to log data for histograms. After running a simulation, you can view the histogram for each variable. For more information, see “Histogram” on page 4-19.

### **Computing Derived Ranges**

The advantage of proposing data types based on derived ranges is that you do not have to provide test files that exercise your algorithm over its full operating range. Running such test files often takes a very long time.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values or proposed data types for all input variables. To improve the analysis, enter as much static range information as possible for other variables. You can manually enter ranges or promote simulation ranges to use as static ranges. Manually-entered static ranges always take precedence over simulation ranges.

If you know what data type your hardware target uses, set the proposed data types to match this type. Manually-entered data types are locked so that the tool cannot modify them. The tool uses these data types to calculate the input minimum and maximum values and to derive ranges for other variables. For more information, see “Locking Proposed Data Types” on page 4-13.

When you select **Compute Derived Ranges**, the tool runs a derived range analysis to compute static ranges for variables in your MATLAB algorithm. When the analysis is complete, the static ranges are displayed on the **Variables** tab. If the run produces  $+\text{-Inf}$  derived ranges, consider defining ranges for all persistent variables.

Optionally, you can select **Quick derived range analysis**. With this option, the conversion tool performs faster static analysis. The computed ranges might be larger than necessary. Select this option in cases where the static analysis takes more time than you can afford.

If the derived range analysis for your project is taking a long time, you can optionally set a timeout. The tool aborts the analysis when the timeout is reached.

### **Locking Proposed Data Types**

You can lock proposed data types against changes by the Fixed-Point Conversion tool using one of the following methods:

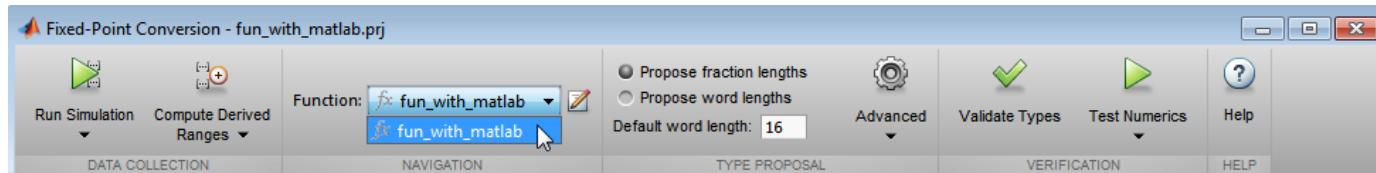
- Manually setting a proposed data type in the Fixed-Point Conversion tool.
- Right-clicking a type proposed by the tool and selecting **Lock computed value**.

The tool displays locked data types in bold so that they are easy to identify. You can unlock a type using one of the following methods:

- Manually overwriting it.
- Right-clicking it and selecting **Undo changes**. This action unlocks only the selected type.
- Right-clicking and selecting **Undo changes for all variables**. This action unlocks all locked proposed types.

## Viewing Functions

You can view a list of functions in your project on the **Navigation** pane. This list also includes function specializations and class methods. When you select a function from the list, the MATLAB code for that function or class method is displayed in the Fixed-Point Conversion tool code window.



After conversion, the left pane also displays a list of output files including the fixed-point version of the original algorithm. If your function is not specialized, the conversion retains the original function name in the fixed-point filename and appends the fixed-point suffix. For example, the fixed-point version of `fun_with_matlab.m` is `fun_with_matlab_fixpt.m`.

## Viewing Variables

The **Variables** tab provides the following information for each variable in the function selected in the **Navigation** pane:

- **Type** — The original data type of the variable in the MATLAB algorithm.
- **Sim Min** and **Sim Max** — The minimum and maximum values assigned to the variable during simulation.

You can edit the simulation minimum and maximum values. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Static Min** and **Static Max** — The static minimum and maximum values.

To compute derived ranges and propose data types based on these ranges, provide static minimum and maximum values for all input variables. To improve the analysis, enter as much static range information as possible for other variables.

When you compute derived ranges, the Fixed-Point Conversion tool runs a static analysis to compute static ranges for variables in your code. When the analysis is complete, the static ranges are displayed. You can edit the computed results. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- **Whole Number** — Whether all values assigned to the variable during simulation are integers.

The Fixed-Point Conversion tool determines whether a variable is always a whole number. You can modify this field. Edited fields are shown in bold. Editing these fields does not trigger static range analysis, but the tool uses the edited values in subsequent analyses. You can revert to the types proposed by the tool.

- The proposed fixed-point data type for the specified word (or fraction) length. Proposed data types use the `numerictype` notation. For example, `numerictype(1,16,12)` denotes a signed fixed-point type with a word length of 16 and a fraction length of 12. `numerictype(0,16,12)` denotes an unsigned fixed-point type with a word length of 16 and a fraction length of 12.

Because the tool does not apply data types to expressions, it does not display proposed types for them. Instead, it displays their original data types.

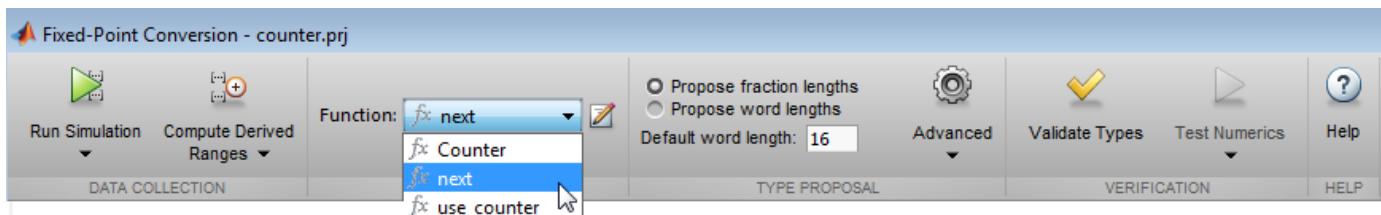
You can also view and edit variable information in the code pane by placing your cursor over a variable name.

You can use **Ctrl+F** to search for variables in the MATLAB code and on the **Variables** tab. The tool highlights occurrences in the code and displays only the variable with the specified name on the **Variables** tab.

## Viewing Information for MATLAB Classes

The tool displays:

- Code for MATLAB classes and code coverage for class methods in the code window. Use the **Function** list in the Navigation bar to select which class or class method to view.



This screenshot shows the MATLAB code editor with the code for the Counter class. The code is as follows:

```

1 classdef Counter < handle
2 properties
3     Value;
4 end
5
6 methods(Static)
7     function t = MAX_VALUE()
8         t = 128;
9     end
10
11
12 methods
13     function this = Counter()
14         this.Value = 0;
15     end
16     function out = next(this)
17         out = this.Value;
18         if this.Value == this.MAX_VALUE
19             this.Value = 0;
20         else
21             this.Value = this.Value + 1;
22         end
23     end
24 end
25 end

```

The code editor has a "NAVIGATION" tab selected. The "Function" dropdown in the top bar is set to "fx next". The "TYPE PROPOSAL" section shows "Validate Types" and "Test Numerics" buttons. The "VERIFICATION" and "HELP" buttons are also visible.

- Information about MATLAB classes on the **Variables** tab.

Variables	Function Replacements	Simulation Output	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
<b>Input</b>								
<b>this</b>								
this	Counter		Unknown	Unknown			No	
this.Value	double		0	1024			Yes	numerictype(0, 11, 0)
<b>Output</b>								
v	double		0	1024			Yes	numerictype(0, 11, 0)

## Specializations

If a function is specialized, the tool lists each specialization and numbers them sequentially. For example, consider a function, `dut`, that calls subfunctions, `foo` and `bar`, multiple times with different input types.

```

function y = dut(u, v)

tt1 = foo(u);
tt2 = foo([u v]);
tt3 = foo(complex(u,v));

ss1 = bar(u);
ss2 = bar([u v]);
ss3 = bar(complex(u,v));

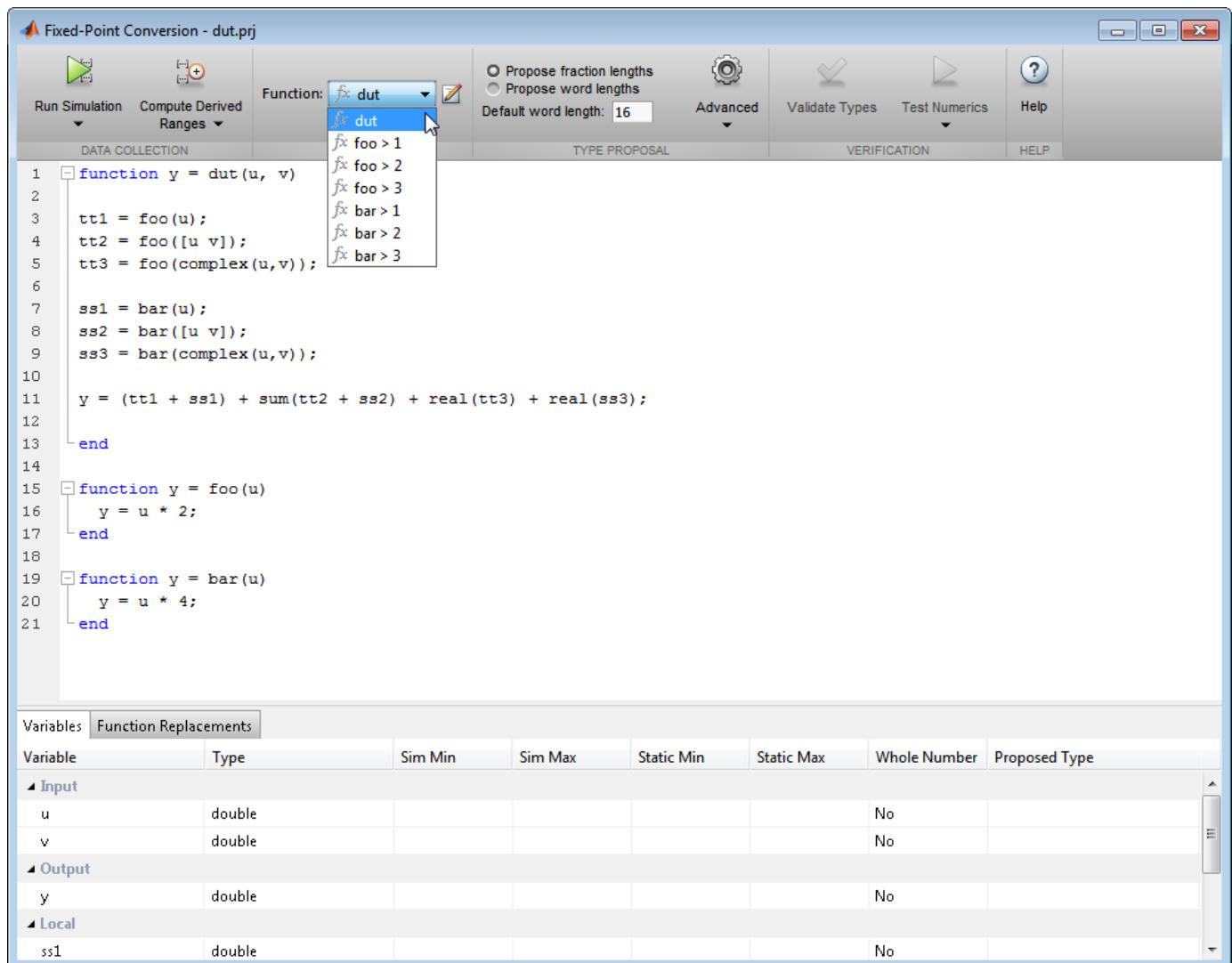
y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);

end

function y = foo(u)
    y = u * 2;
end

function y = bar(u)
    y = u * 4;
end

```



If you select a specialization, the app displays only the variables used by the specialization.

## 4 Fixed-Point Conversion

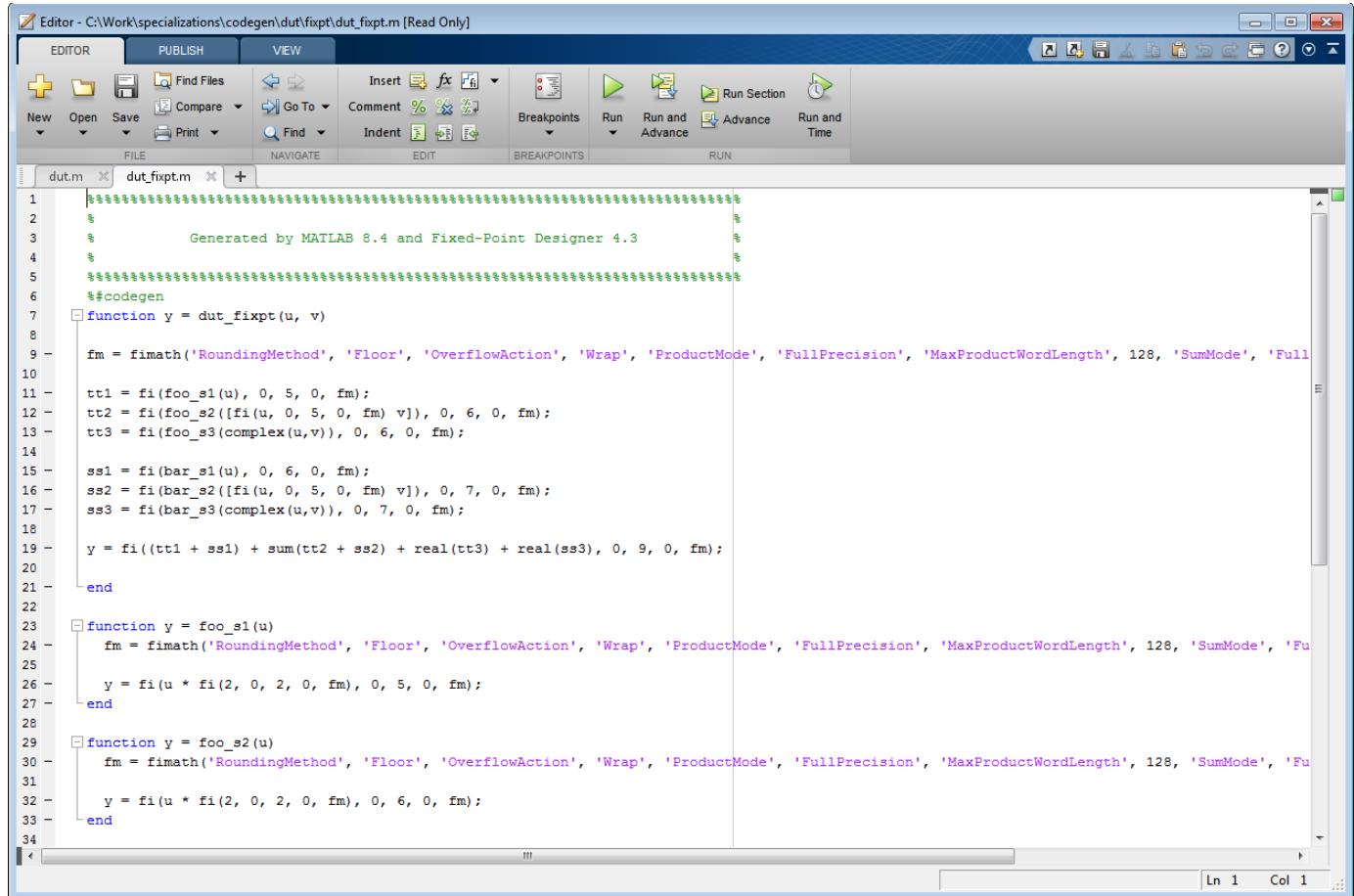
The screenshot shows the MATLAB Fixed-Point Conversion tool window titled "Fixed-Point Conversion - dut.prj". The main area displays the following MATLAB code:

```
1 function y = dut(u, v)
2
3 tt1 = foo(u);
4 tt2 = foo([u v]);
5 tt3 = foo(complex(u,v));
6
7 ss1 = bar(u);
8 ss2 = bar([u v]);
9 ss3 = bar(complex(u,v));
10
11 y = (tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3);
12
13 end
14
15 function y = foo(u)
16 y = u * 2;
17 end
18
19 function y = bar(u)
20 y = u * 4;
21 end
```

Below the code, there are two tabs: "Variables" and "Function Replacements". The "Variables" tab is selected, showing the following table:

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
u	double					No	
y	double					No	

In the generated fixed-point code, the number of each fixed-point specialization matches the number in the Source Code list which makes it easy to trace between the floating-point and fixed-point versions of your code. For example, the generated fixed-point function for `foo > 1` is named `foo_s1`.



The screenshot shows the MATLAB Editor window with the title "Editor - C:\Work\specializations\codegen\dut\fixpt\dut\_fixpt.m [Read Only]". The menu bar includes EDITOR, PUBLISH, and VIEW. The toolbar includes New, Open, Save, Find Files, Compare, Go To, Comment, Breakpoints, Run, Run and Advance, Run Section, and Run and Time. The code editor displays the generated C code for the function `dut_fixpt`. The code includes comments indicating it was generated by MATLAB 8.4 and Fixed-Point Designer 4.3, and it uses the `#codegen` directive. The code defines several functions: `dut_fixpt`, `foo_s1`, `foo_s2`, and `bar_s1`, `bar_s2`, `bar_s3`. It performs operations like multiplication, addition, and summation using fixed-point arithmetic with specified rounding and overflow actions.

```

1 % Generated by MATLAB 8.4 and Fixed-Point Designer 4.3
2 %
3 %#codegen
4
5 function y = dut_fixpt(u, v)
6
7 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full');
8
9 tt1 = fi(foo_s1(u), 0, 5, 0, fm);
10 tt2 = fi(foo_s2([fi(u, 0, 5, 0, fm) v]), 0, 6, 0, fm);
11 tt3 = fi(foo_s3(complex(u,v)), 0, 6, 0, fm);
12
13 ss1 = fi(bar_s1(u), 0, 6, 0, fm);
14 ss2 = fi(bar_s2([fi(u, 0, 5, 0, fm) v]), 0, 7, 0, fm);
15 ss3 = fi(bar_s3(complex(u,v)), 0, 7, 0, fm);
16
17 y = fi((tt1 + ss1) + sum(tt2 + ss2) + real(tt3) + real(ss3), 0, 9, 0, fm);
18
19 end
20
21
22 function y = foo_s1(u)
23 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full');
24
25 y = fi(u * fi(2, 0, 2, 0, fm), 0, 5, 0, fm);
26
27 end
28
29 function y = foo_s2(u)
30 fm = fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'MaxProductWordLength', 128, 'SumMode', 'Full');
31
32 y = fi(u * fi(2, 0, 2, 0, fm), 0, 6, 0, fm);
33
34 end

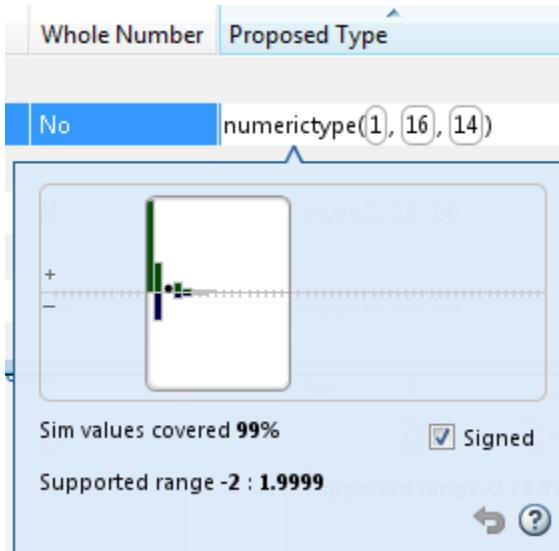
```

## Histogram

To log data for histograms, in the Fixed-Point Conversion window, click **Run Simulation** and select **Log data for histogram**, and then click the Run Simulation button.

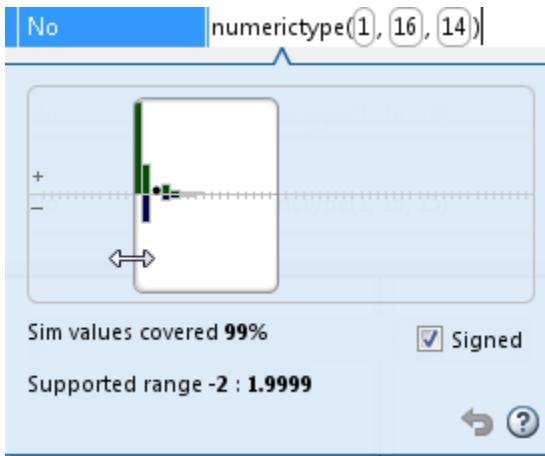
After simulation, to view the histogram for a variable, on the **Variables** tab, click the **Proposed Type** field for that variable.

The histogram provides the range of the proposed data type and the percentage of simulation values that the proposed data type covers. The bit weights are displayed along the X-axis, and the percentage of occurrences along the Y-axis. Each bin in the histogram corresponds to a bit in the binary word. For example, this histogram displays the range for a variable of type `numerictype(1,16,14)`.



You can view the effect of changing the proposed data types by:

- Dragging the edges of the bounding box in the histogram window to change the proposed data type.

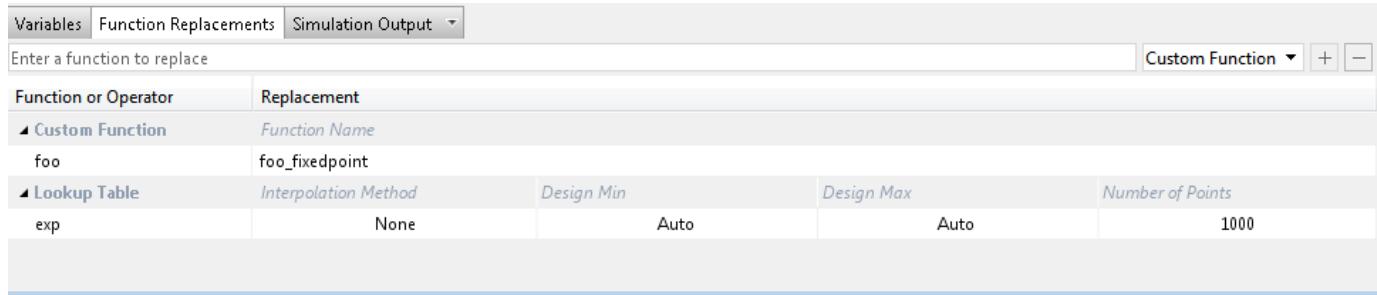


- Selecting or clearing **Signed**.

To revert to the types proposed by the automatic conversion, in the histogram window, click .

## Function Replacements

If your MATLAB code uses functions that do not have fixed-point support, the tool lists these functions on the **Function Replacements** tab. You can choose to replace unsupported functions with a custom function replacement or with a lookup table.



You can add and remove function replacements from this list. If you enter a function replacements for a function, the replacement function is used when you build the project. If you do not enter a replacement, the tool uses the type specified in the original MATLAB code for the function.

---

**Note** Using this table, you can replace the names of the functions but you cannot replace argument patterns.

---

## Validating Types

Selecting **Validate Types** validates the build using the proposed fixed-point data types. If the validation is successful, you are ready to test the numerical behavior of the fixed-point MATLAB algorithm.

If the errors or warnings occur during validation, they are displayed on the **Type Validation Output** tab. If errors or warning occur:

- On the **Variables** tab, inspect the proposed types and manually modified types to verify that they are valid.
- On the **Function Replacements** tab, verify that you have provided function replacements for unsupported functions.

## Testing Numerics

After validating the proposed fixed-point data types, select **Test Numerics** to verify the behavior of the fixed-point MATLAB algorithm. By default, if you added a test bench to define inputs or run a simulation, the tool uses this test bench to test numerics. The tool compares the numerical behavior of the generated fixed-point MATLAB code with the original floating-point MATLAB code. If you select to log inputs and outputs for comparison plots, the tool generates an additional plot for each scalar output. This plot shows the floating-point and fixed-point results and the difference between them. For non-scalar outputs, only the error information is shown.

If the numerical results do not meet your desired accuracy after fixed-point simulation, modify fixed-point data type settings and repeat the type validation and numerical testing steps. You might have to iterate through these steps multiple times to achieve the desired results.

## Detecting Overflows

When testing numerics, selecting **Use scaled doubles to detect overflows** enables overflow detection. When this option is selected, the conversion tool runs the simulation using scaled double versions of the proposed fixed-point types. Because scaled doubles store their data in double-

precision floating-point, they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type. . .

If the tool detects overflows, on its Overflow tab, it provides:

- A list of variables and expressions that overflowed
- Information on how much each variable overflowed
- A link to the variables or expressions in the code window

Variables	Function Replacements	Overflows ▾	
	Function	Line	Description
⚠	overflow_fixpt	7	Overflow error in expression 'x'.
⚠	overflow_fixpt	7	Overflow error in expression 'y'.
⚠	overflow_fixpt	10	Overflow error in expression 'z'.
⚠	overflow_fixpt	10	Overflow error in expression 'z = fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'fi(x*y, 0, 8, 0, fm)'.
⚠	overflow_fixpt	10	Overflow error in expression 'x'.
⚠	overflow_fixpt	10	Overflow error in expression 'x*y'.
⚠	overflow_fixpt	10	Overflow error in expression 'y'.
⚠	overflow_fixpt	11	Overflow error in expression 'z'.

If your original algorithm uses scaled doubles, the tool also provides overflow information for these expressions.

## See Also

["Detect Overflows"](#)

## Custom Plot Functions

The Fixed-Point Conversion tool provides a default time series based plotting function. The conversion process uses this function at the test numerics step to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. For example, plots that show eye diagrams and bit error differences are more suitable in the communications domain and histogram difference plots are more suitable in image processing designs.

You can choose to use a custom plot function at the test numerics step. The Fixed-Point Conversion tool facilitates custom plotting by providing access to the raw logged input and output data before and after fixed-point conversion. You supply a custom plotting function to visualize the differences between the floating-point and fixed-point results. If you specify a custom plot function, the fixed-point conversion process calls the function for each input and output variable, passes in the name of the variable and the function that uses it, and the results of the floating-point and fixed-point simulations.

Your function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.

Use this information to:

- Customize plot headings and axes.
- Choose which variables to plot.
- Generate different error metrics for different output variables.
- A cell array to hold the logged floating-point values for the variable.

This cell array contains values observed during floating-point simulation of the algorithm during the test numerics phase. You might need to reformat this raw data.

- A cell array to hold the logged values for the variable after fixed-point conversion.

This cell array contains values observed during fixed-point simulation of the converted design.

For example, `function customComparisonPlot(varInfo, floatVarVals, fixedPtVarVals)`.

To use a custom plot function, in the Fixed-Point Conversion tool, select **Advanced**, and then set **Custom plot function** to the name of your plot function.

In the programmatic workflow, set the `coder.FixptConfig` configuration object `PlotFunction` property to the name of your plot function. See “Visualize Differences Between Floating-Point and Fixed-Point Results” on page 4-24.

## Visualize Differences Between Floating-Point and Fixed-Point Results

This example shows how to configure the `codegen` function to use a custom plot function to compare the behavior of the generated fixed-point code against the behavior of the original floating-point MATLAB code.

By default, when the `LogIOForComparisonPlotting` option is enabled, the conversion process uses a time series based plotting function to show the floating-point and fixed-point results and the difference between them. However, during fixed-point conversion you might want to visualize the numerical differences in a view that is more suitable for your application domain. This example shows how to customize plotting and produce scatter plots at the test numerics step of the fixed-point conversion.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

See [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/).

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

### Create a New Folder and Copy Relevant Files

- 1 Create a local working folder, for example, `c:\custom_plot`.
- 2 Change to the `docroot\toolbox\fixpoint\examples` folder. At the MATLAB command line, enter:  
`cd(fullfile(docroot, 'toolbox', 'fixpoint', 'examples'))`
- 3 Copy the `myFilter.m`, `myFilterTest.m`, `plotDiff.m`, and `filterData.mat` files to your local working folder.

Type	Name	Description
Function code	<code>myFilter.m</code>	Entry-point MATLAB function
Test file	<code>myFilterTest.m</code>	MATLAB script that tests <code>myFilter.m</code>
Plotting function	<code>plotDiff.m</code>	Custom plot function
MAT-file	<code>filterData.mat</code>	Data to filter.

### The `myFilter` Function

```
function [y, ho] = myFilter(in)

persistent b h;
if isempty(b)
    b = complex(zeros(1,16));
    h = complex(zeros(1,16));
```

```

    h(8) = 1;
end

b = [in, b(1:end-1)];
y = b*h.';

errf = 1-sqrt(real(y)*real(y) + imag(y)*imag(y));
update = 0.001*conj(b)*y*errf;

h = h + update;
h(8) = 1;
ho = h;

end

```

### The myFilterTest File

```

% load data
data = load('filterData.mat');
d = data.symbols;

for idx = 1:4000
    y = myFilter(d(idx));
end

```

### The plotDiff Function

```

% varInfo - structure with information about the variable. It has the following fields
%     i) name
%     ii) functionName
% floatVals - cell array of logged original values for the 'varInfo.name' variable
% fixedVals - cell array of logged values for the 'varInfo.name' variable after Fixed-Point conversion
function plotDiff(varInfo, floatVals, fixedVals)
    varName = varInfo.name;
    fcnName = varInfo.functionName;

    % escape the '_'s because plot titles treat these as subscripts
    escapedVarName = regexp替換(varName, '_', '\_\_');
    escapedFcnName = regexp替換(fcnName, '_', '\_\_');

    % flatten the values
    flatFloatVals = floatVals(1:end);
    flatFixedVals = fixedVals(1:end);

    % build Titles
    floatTitle = [ escapedFcnName ' > ' 'float : ' escapedVarName ];
    fixedTitle = [ escapedFcnName ' > ' 'fixed : ' escapedVarName ];

    data = load('filterData.mat');

    switch varName
        case 'y'
            x_vec = data.symbols;

            figure('Name', 'Comparison plot', 'NumberTitle', 'off');

            % plot floating point values
            y_vec = flatFloatVals;

```

```
    subplot(1, 2, 1);
    plotScatter(x_vec, y_vec, 100, floatTitle);

    % plot fixed point values
    y_vec = flatFixedVals;
    subplot(1, 2, 2);
    plotScatter(x_vec, y_vec, 100, fixedTitle);

otherwise
    % Plot only output 'y' for this example, skip the rest
end

end

function plotScatter(x_vec, y_vec, n, figTitle)
% plot the last n samples
x_plot = x_vec(end-n+1:end);
y_plot = y_vec(end-n+1:end);

hold on
scatter(real(x_plot),imag(x_plot), 'bo');

hold on
scatter(real(y_plot),imag(y_plot), 'rx');

title(figTitle);
end
```

## Set Up Configuration Object

- 1 Create a `coder.FixptConfig` object.

```
fxptcfg = coder.config('fixpt');
```

- 2 Specify the test file name and custom plot function name. Enable logging and numerics testing.

```
fxptcfg.TestBenchName = 'myFilterTest';
fxptcfg.PlotFunction = 'plotDiff';
fxptcfg.TestNumerics = true;
fxptcfg.LogIOForComparisonPlotting = true;
fxptcfg.DefaultWordLength = 16;
```

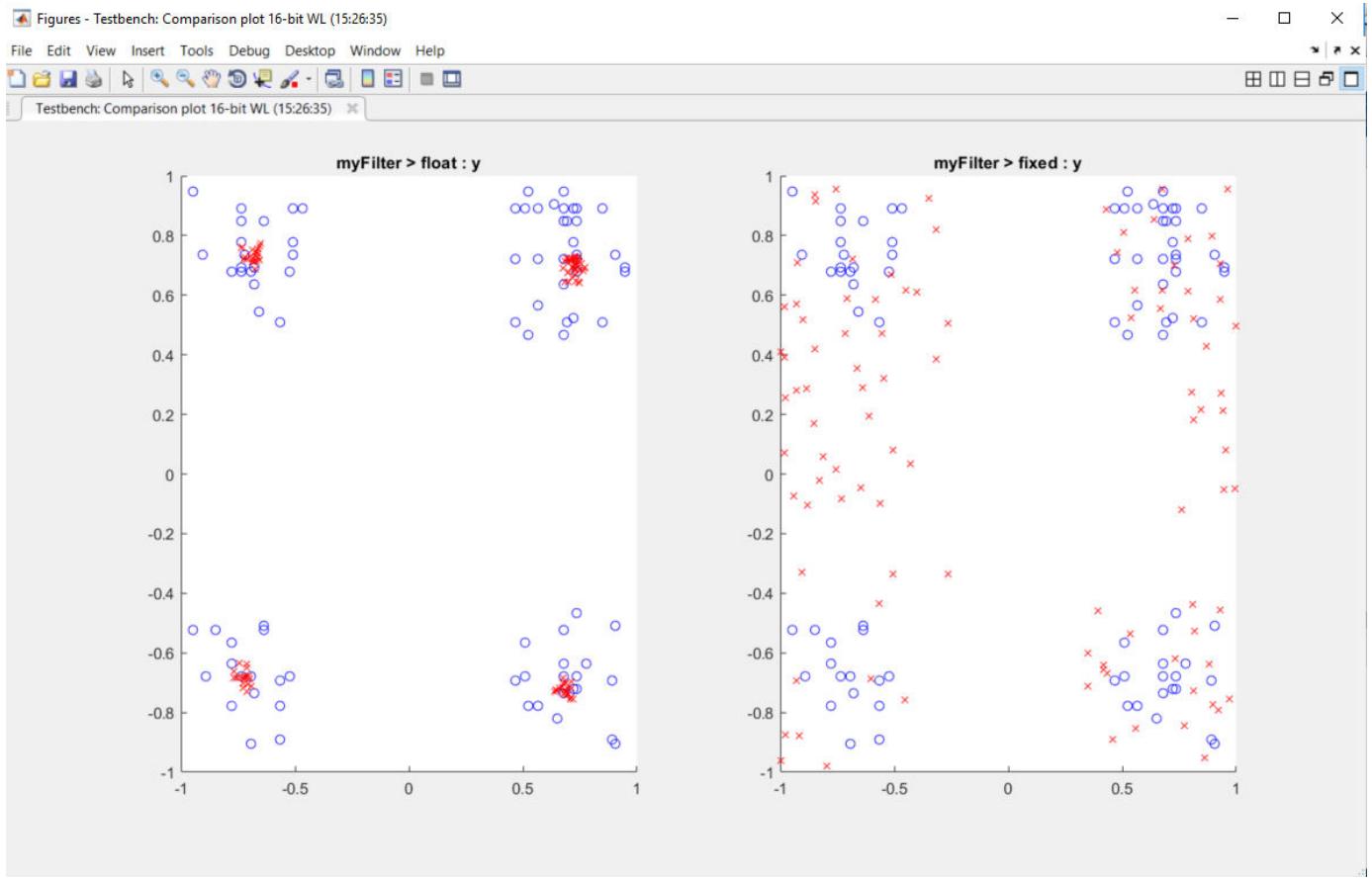
## Convert to Fixed Point

Convert the floating-point MATLAB function, `myFilter`, to fixed-point MATLAB code. You do not need to specify input types for the `codegen` command because it infers the types from the test file.

```
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The conversion process generates fixed-point code using a default word length of 16 and then runs a fixed-point simulation by running the `myFilterTest.m` function and calling the fixed-point version of `myFilter.m`.

Because you selected to log inputs and outputs for comparison plots and to use the custom plotting function, `plotDiff.m`, for these plots, the conversion process uses this function to generate the comparison plot.



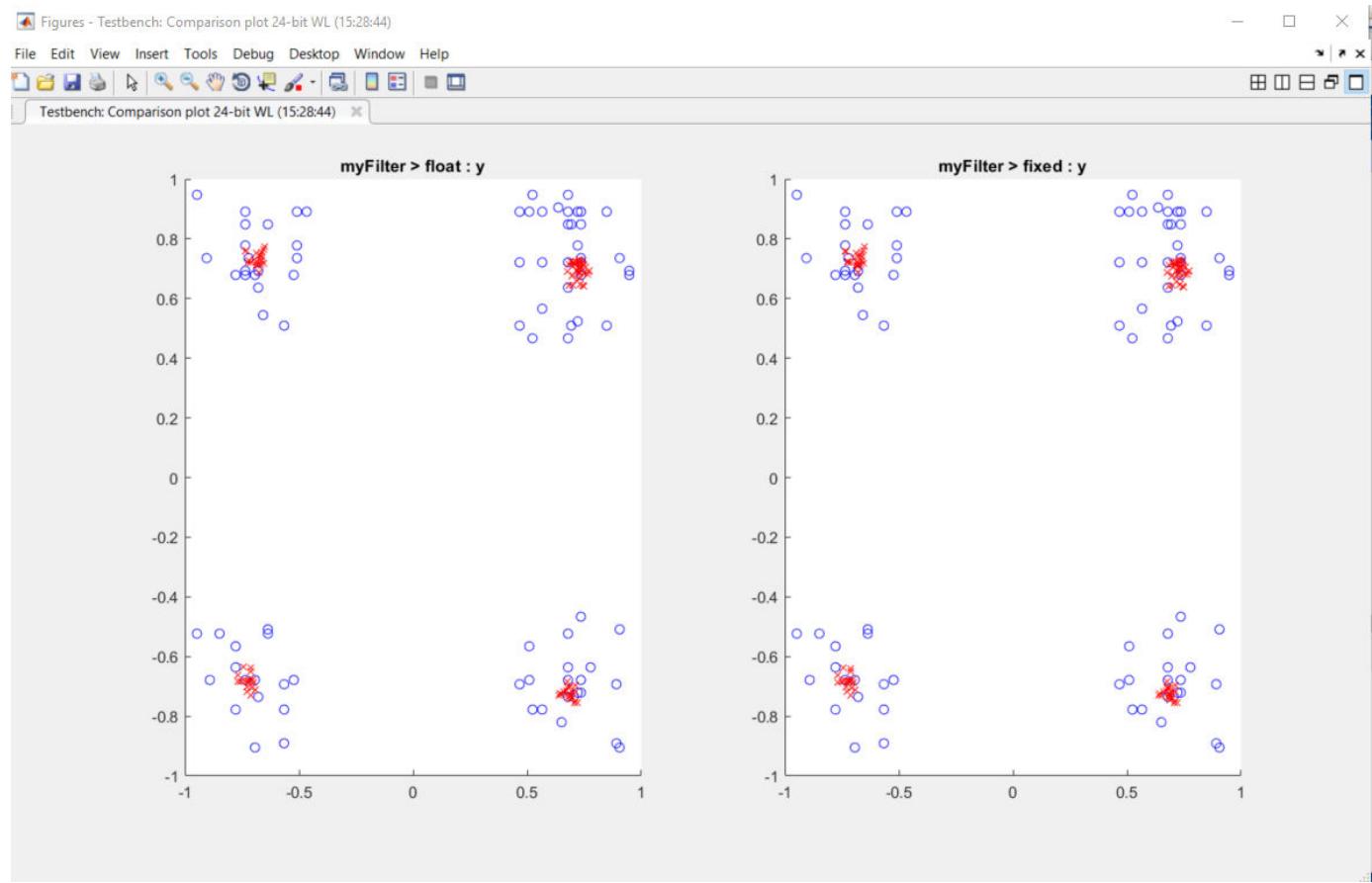
The plot shows that the fixed-point results do not closely match the floating-point results.

Increase the word length to 24 and then convert to fixed point again.

```
fxptcfg.DefaultWordLength = 24;
codegen -args {complex(0, 0)} -float2fixed fxptcfg myFilter
```

The increased word length improved the results. This time, the plot shows that the fixed-point results match the floating-point results.

## 4 Fixed-Point Conversion



# Inspecting Data Using the Simulation Data Inspector

## In this section...

- ["What Is the Simulation Data Inspector?" on page 4-29](#)
- ["Import Logged Data" on page 4-29](#)
- ["Export Logged Data" on page 4-29](#)
- ["Group Signals" on page 4-29](#)
- ["Run Options" on page 4-29](#)
- ["Create Report" on page 4-30](#)
- ["Comparison Options" on page 4-30](#)
- ["Enabling Plotting Using the Simulation Data Inspector" on page 4-30](#)
- ["Save and Load Simulation Data Inspector Sessions" on page 4-30](#)

## What Is the Simulation Data Inspector?

The Simulation Data Inspector allows you to view data logged during the fixed-point conversion process. You can use it to inspect and compare the inputs and outputs to the floating-point and fixed-point versions of your algorithm.

For fixed-point conversion, there is no programmatic interface for the Simulation Data Inspector.

## Import Logged Data

Before importing data into the Simulation Data Inspector, you must have previously logged data to the base workspace or to a MAT-file.

## Export Logged Data

The Simulation Data Inspector provides the capability to save data collected by the fixed-point conversion process to a MAT-file that you can later reload. The format of the MAT-file is different from the format of a MAT-file created from the base workspace.

## Group Signals

You can customize the organization of your logged data in the Simulation Data Inspector **Runs** pane. By default, data is first organized by run. You can then organize your data by logged variable or no hierarchy.

## Run Options

You can configure the Simulation Data Inspector to:

- Append New Runs

In the Run Options dialog box, the default is set to add new runs to the bottom of the run list. To append new runs to the top of the list, select **Add new runs at top**.

- Specify a Run Naming Rule

To specify run naming rules, in the Simulation Data Inspector toolbar, click **Run Options**.

## Create Report

You can create a report of the runs or comparison plots. Specify the name and location of the report file. By default, the Simulation Data Inspector overwrites existing files. To preserve existing reports, select **If report exists, increment file name to prevent overwriting**.

## Comparison Options

To change how signals are matched when runs are compared, specify the **Align by** and **Then by** parameters and then click **OK**.

## Enabling Plotting Using the Simulation Data Inspector

To enable the Simulation Data Inspector, see “Enable Plotting Using the Simulation Data Inspector” on page 4-31.

## Save and Load Simulation Data Inspector Sessions

If you have data in the Simulation Data Inspector and you want to archive or share the data to view in the Simulation Data Inspector later, save the Simulation Data Inspector session. When you save a Simulation Data Inspector session, the MAT-file contains:

- All runs, data, and properties from the **Runs** and **Comparisons** panes.
- Check box selection state for data in the **Runs** pane.

### Save a Session to a MAT-File

- 1 On the **Visualize** tab, click **Save**.
- 2 Browse to where you want to save the MAT-file to, name the file, and click **Save**.

### Load a Saved Simulation Data Inspector Simulation

- 1 On the **Visualize** tab, click **Open**.
- 2 Browse, select the MAT-file saved from the Simulation Data Inspector, and click **Open**.
- 3 If data in the session is plotted on multiple subplots, on the **Format** tab, click **Subplots** and select the subplot layout.

# Enable Plotting Using the Simulation Data Inspector

## In this section...

["From the UI" on page 4-31](#)

["From the Command Line" on page 4-31](#)

## From the UI

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point logged input and output data. In the Fixed-Point Conversion tool:

- 1 Click **Advanced**.
- 2 In the Advanced Settings dialog box, set **Plot with Simulation Data Inspector** to Yes.
- 3 At the Test Numerics stage in the conversion process, click **Test Numerics**, select **Log inputs and outputs for comparison plots**, and then click .

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.

## From the Command Line

You can use the Simulation Data Inspector to inspect and compare floating-point and fixed-point input and output data logged using the function. At the MATLAB command line:

- 1 Create a fixed-point configuration object and configure the test file name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'dti_test';
```

- 2 Select to run the test file to verify the generated fixed-point MATLAB code. Log inputs and outputs for comparison plotting and select to use the Simulation Data Inspector to plot the results.

```
fixptcfg.TestNumerics = true;
fixptcfg.LogIOForComparisonPlotting = true;
fixptcfg.PlotWithSimulationDataInspector = true;
```

- 3 Generate fixed-point MATLAB code using `codegen`.

```
codegen -float2fixed fixptcfg -config cfg dti
```

For an example, see “Propose Fixed-Point Data Types Based on Derived Ranges”.

## Replacing Functions Using Lookup Table Approximations

The software provides an option to generate lookup table approximations for continuous and stateless single-input, single-output functions in your original MATLAB code. These functions must be on the MATLAB path.

You can use this capability to handle functions that are not supported for fixed point and to replace your own custom functions. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. You can control the interpolation method and number of points in the lookup table. By adjusting these settings, you can tune the behavior of replacement function to match the behavior of the original function as closely as possible.

The fixed-point conversion process generates one lookup table approximation per call site of the function that needs replacement.

To use lookup table approximations, see:

- `coder.approximation`
- “Replace the `exp` Function with a Lookup Table” on page 4-39
- “Replace a Custom Function with a Lookup Table” on page 4-33

# Replace a Custom Function with a Lookup Table

## In this section...

"Using the HDL Coder App" on page 4-33

"From the Command Line" on page 4-36

With HDL Coder, you can generate lookup table approximations for functions that do not support fixed-point types, and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder app, or the `fiaccel codegen` function.

## Using the HDL Coder App

This example shows how to replace a custom function with a Lookup Table using the **HDL Coder** app.

### Create Algorithm and Test Files

In a local, writable folder:

- 1 Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

- 2 Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

- 3 Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

### Create and Set up a HDL Coder Project

- 1 Navigate to the work folder that contains the file for this example.
- 2 To open the **HDL Coder** app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.
- 3 In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

### Define Input Types

- 1 To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodetect types**.
- 2 Add `custom_test` as a test file, and then click **Run**.

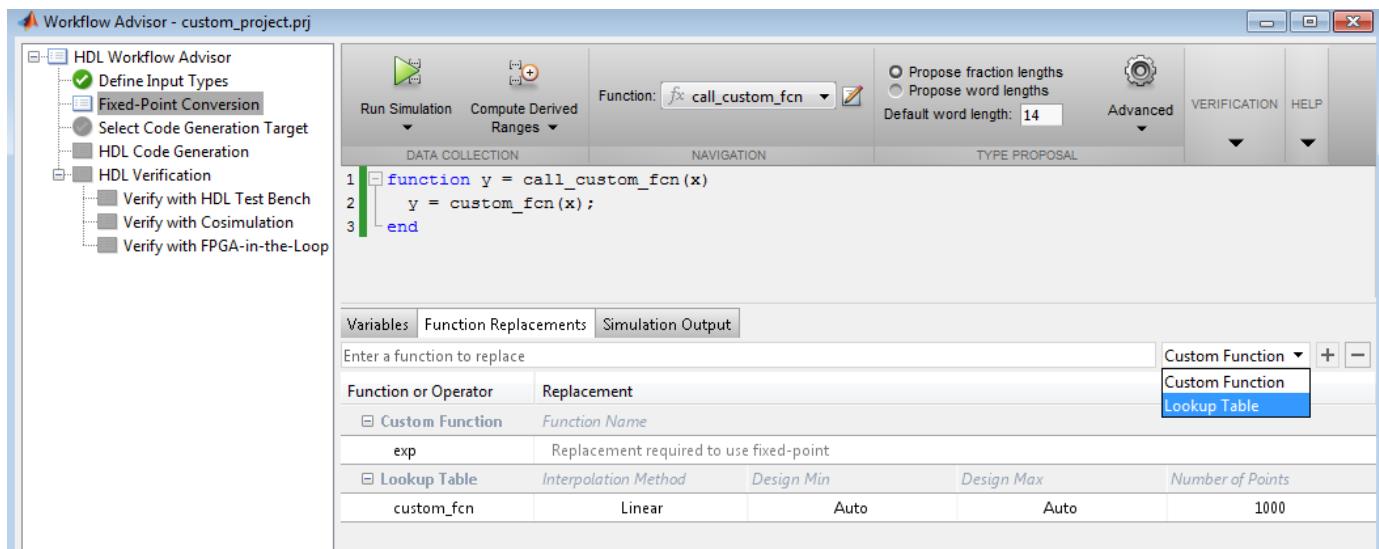
From the test file, HDL Coder determines that `x` is a scalar double.

**3 Click Use These Types.**

### Replace custom\_fcn with Lookup Table

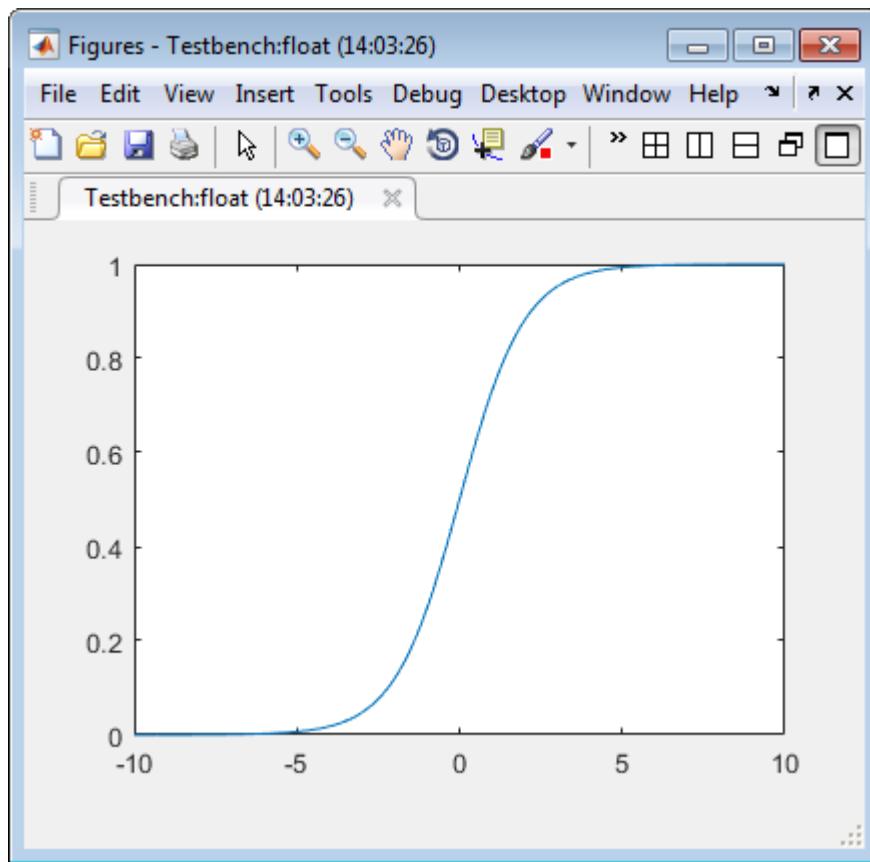
- 1 To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.
- 2 To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select **Lookup Table**, and then click **+**.

By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



- 3 Under **Run Simulation**, select **Log data for histogram**, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

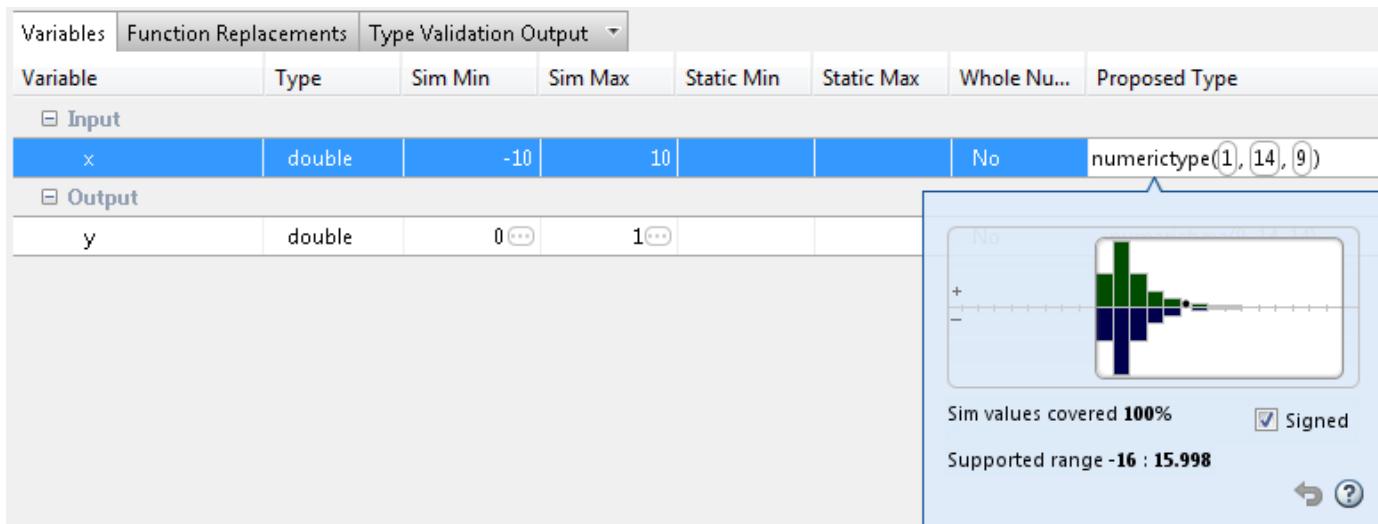
The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.



### Validate Fixed-Point Types

- In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

The histogram provides range information and the percentage of simulation range that the proposed data type covers.



- 2 To validate the build by using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

- 3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor',...
        'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision',...
        'MaxProductWordLength', 128,...
        'SumMode', 'FullPrecision',...
        'MaxSumWordLength', 128);
end
```

## From the Command Line

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

See [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/).

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

Create a MATLAB function, `custom_fcn.m`. This is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

Create a wrapper function that calls `custom_fcn.m`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

Create a test file, `custom_test.m`, that uses `call_custom_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
```

```

y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );

```

Create a function replacement configuration object to approximate `custom_fcn`. Specify the function handle of the custom function and set the number of points to use in the lookup table to 50.

```
q = coder.approximation('Function','custom_fcn',...
    'CandidateFunction',@custom_fcn, 'NumberOfPoints',50);
```

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'custom_test';
fixptcfg.TestNumerics = true;
fixptcfg.addApproximation(q);
```

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg call_custom_fcn
```

`codegen` generates fixed-point MATLAB code in `call_custom_fcn_fixpt.m`.

To view the generated fixed-point code, click the link to `call_custom_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_custom_fcn`, for the `custom_fcn` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. The lookup table uses 50 points as specified. By default, it uses linear interpolation and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `call_custom_fcn_fixpt`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();
    y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

## See Also

`coder.approximation`

## Related Examples

- “Replace the `exp` Function with a Lookup Table” on page 4-39

## More About

- “Replacing Functions Using Lookup Table Approximations” on page 4-32

# Replace the exp Function with a Lookup Table

With HDL Coder, you can handle functions that are not supported for fixed point and replace your own functions. To replace a custom function with a Lookup Table, use the HDL Coder App, or the `fiaccel codegen` function.

## In this section...

["From the UI" on page 4-39](#)

["From the Command Line" on page 4-42](#)

## From the UI

This example shows how to replace a custom function with a Lookup Table using the **HDL Coder** app.

### Create Algorithm and Test Files

In a local, writable folder:

- 1 Create a MATLAB function, `custom_fcn`, which is the function that you want to replace.

```
function y = custom_fcn(x)
    y = 1./(1+exp(-x));
end
```

- 2 Create a wrapper function that calls `custom_fcn`.

```
function y = call_custom_fcn(x)
    y = custom_fcn(x);
end
```

- 3 Create a test file, `custom_test`, which uses `call_custom_fcn`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = call_custom_fcn( x(itr) );
end
plot( x, y );
```

### Create and Set up a HDL Coder Project

- 1 Navigate to the work folder that contains the file for this example.
- 2 To open the **HDL Coder** app, in the MATLAB command prompt, enter `hdlcoder`. Set **Name** to `custom_project.prj` and click **OK**. The project opens in the MATLAB workspace.
- 3 In the project window, on the **MATLAB Function** tab, click the **Add MATLAB function** link. Browse to the file `call_custom_fcn.m`, and then click **OK** to add the file to the project.

### Define Input Types

- 1 To define input types for `call_custom_fcn.m`, on the **MATLAB Function** tab, click **Autodefine types**.
- 2 Add `custom_test` as a test file, and then click **Run**.

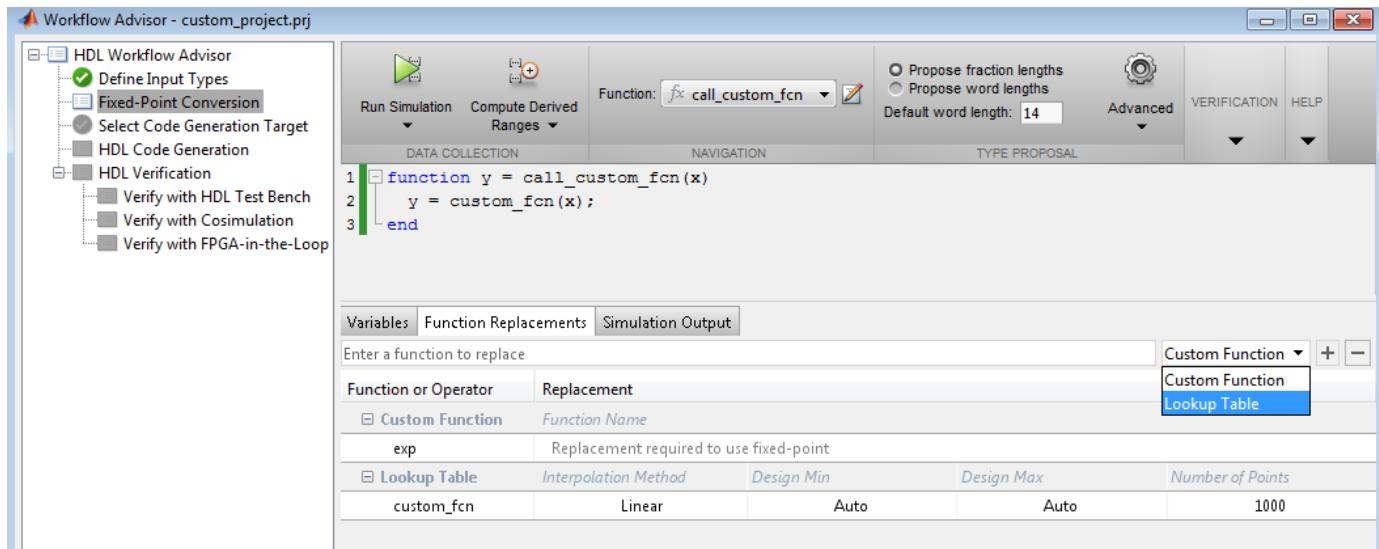
From the test file, HDL Coder determines that `x` is a scalar double.

**3 Click Use These Types.**

### Replace custom\_fcn with Lookup Table

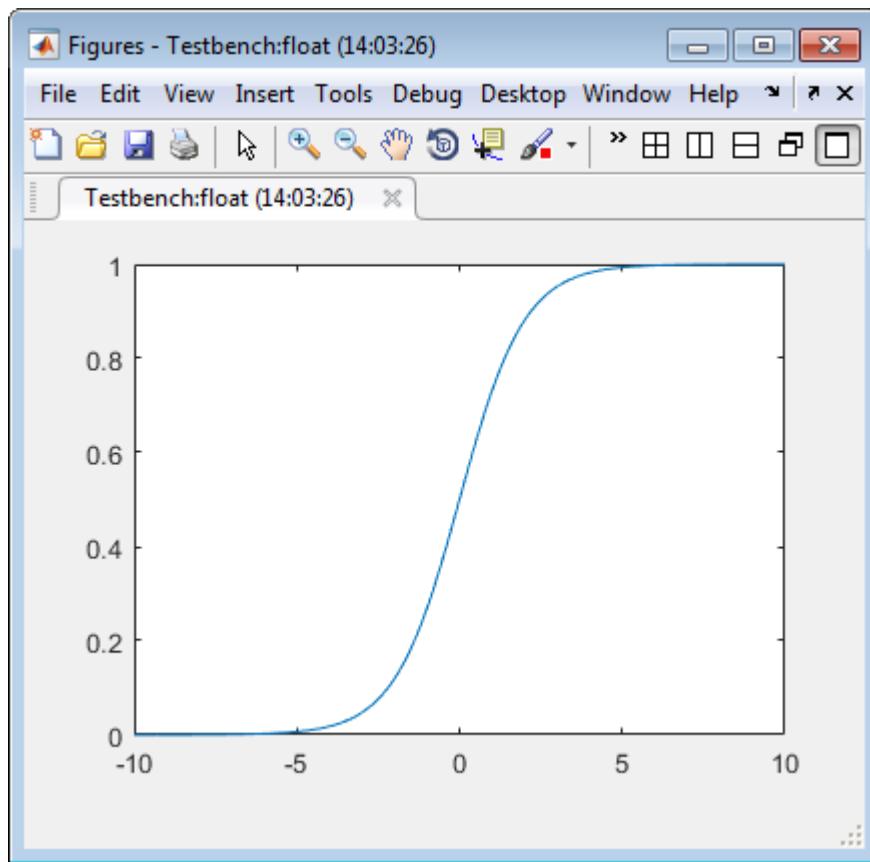
- 1 To open the HDL Workflow Advisor, click **Workflow Advisor**, and in the Workflow Advisor window, click **Fixed-Point Conversion**.
- 2 To replace `custom_fcn` with a Lookup Table, on the **Function Replacements** tab, enter `custom_fcn`, select **Lookup Table**, and then click **+**.

By default, the lookup table uses linear interpolation, 1000 points, and design minimum and maximum values that the app detects by running a simulation or computing derived ranges.



- 3 Under **Run Simulation**, select **Log data for histogram**, and then click **Run Simulation**. Verify that `custom_test` file is selected as the test file.

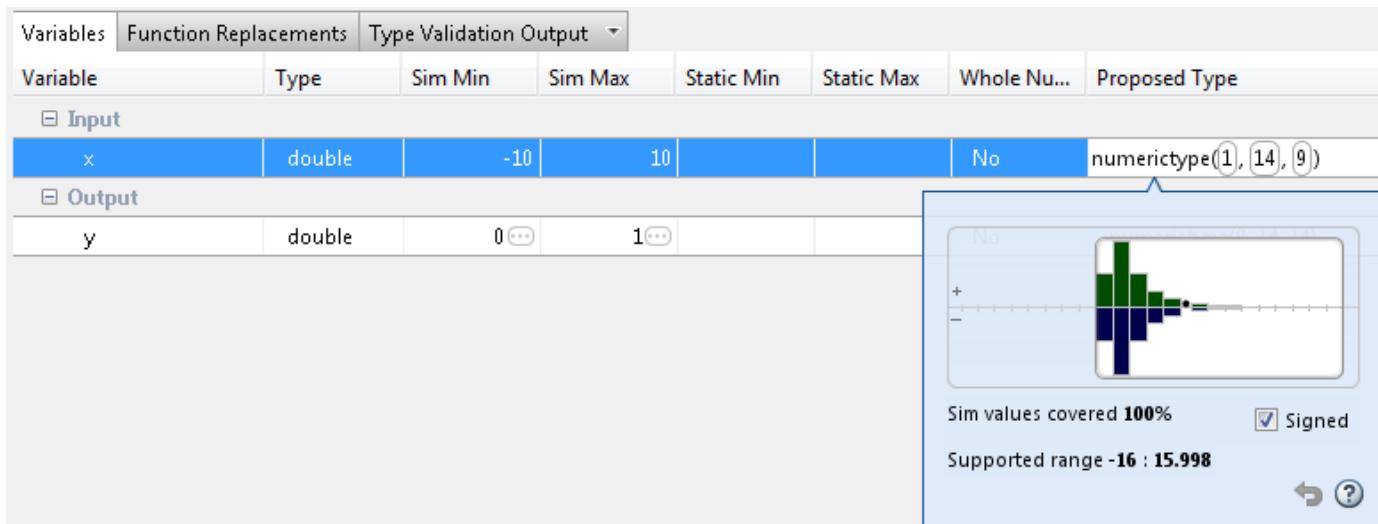
The simulation runs and the tool displays simulation minimum and maximum ranges on the **Variables** tab. HDL Coder plots the simulation results in the MATLAB Editor.



### Validate Fixed-Point Types

- 1 In the **Proposed Type** column, verify that the fixed-point types proposed by software cover the full simulation range. To view logged histogram data for a variable, click its **Proposed Type** field.

The histogram provides range information and the percentage of simulation range that the proposed data type covers.



- 2 To validate the build by using the proposed types, click **Validate Types**.

The software validates the proposed types and generates a fixed-point code, `call_custom_fcn_fixpt`.

- 3 To view the generated fixed-point code, click the `call_custom_fcn_fixpt` link.

The generated fixed-point function, `call_custom_fcn_fixpt.m`, calls this approximation instead of calling `custom_fcn`.

```
function y = call_custom_fcn_fixpt(x)
    fm = get_fimath();

    y = fi(replacement_custom_fcn(x), 0, 14, 14, fm);
end

function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Floor',...
        'OverflowAction', 'Wrap',...
        'ProductMode', 'FullPrecision',...
        'MaxProductWordLength', 128,...
        'SumMode', 'FullPrecision',...
        'MaxSumWordLength', 128);
end
```

## From the Command Line

This example shows how to replace the `exp` function with a lookup table approximation in the generated fixed-point code using the function.

### Prerequisites

To complete this example, you must install the following products:

- MATLAB
- Fixed-Point Designer
- C compiler

See [https://www.mathworks.com/support/compilers/current\\_release/](https://www.mathworks.com/support/compilers/current_release/).

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”.

### Create Algorithm and Test Files

- 1 Create a MATLAB function, `my_fcn.m`, that calls the `exp` function.

```
function y = my_fcn(x)
    y = exp(x);
end
```

- 2 Create a test file, `my_fcn_test.m`, that uses `my_fcn.m`.

```
close all

x = linspace(-10,10,1e3);
for itr = 1e3:-1:1
    y(itr) = my_fcn( x(itr) );
```

```
end
plot( x, y );
```

## Configure Approximation

Create a function replacement configuration object to approximate the `exp` function, using the default settings of linear interpolation and 1000 points in the lookup table.

```
q = coder.approximation('exp');
```

## Set Up Configuration Object

Create a `coder.FixptConfig` object, `fixptcfg`. Specify the test file name and enable numerics testing. Associate the function replacement configuration object with the fixed-point configuration object.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'my_fcn_test';
fixptcfg.TestNumerics = true;
fixptcfg.DefaultWordLength = 16;
fixptcfg.addApproximation(q);
```

## Convert to Fixed Point

Generate fixed-point MATLAB code.

```
codegen -float2fixed fixptcfg my_fcn
```

## View Generated Fixed-Point Code

To view the generated fixed-point code, click the link to `my_fcn_fixpt`.

The generated code contains a lookup table approximation, `replacement_exp`, for the `exp` function. The fixed-point conversion process infers the ranges for the function and then uses an interpolated lookup table to replace the function. By default, the lookup table uses linear interpolation, 1000 points, and the minimum and maximum values detected by running the test file.

The generated fixed-point function, `my_fcn_fixpt`, calls this approximation instead of calling `exp`.

```
function y = my_fcn_fixpt(x)
    fm = get_fimath();
    y = fi(replacement_exp(x), 0, 16, 1, fm);
end
```

You can now test the generated fixed-point code and compare the results against the original MATLAB function. If the behavior of the generated fixed-point code does not match the behavior of the original code closely enough, modify the interpolation method or number of points used in the lookup table and then regenerate code.

## See Also

`coder.approximation`

## Related Examples

- “Replace a Custom Function with a Lookup Table” on page 4-33

## More About

- “Replacing Functions Using Lookup Table Approximations” on page 4-32

# Data Type Issues in Generated Code

Within the fixed-point conversion report, you have the option to highlight MATLAB code that results in double, single, or expensive fixed-point operations. Consider enabling these checks when trying to achieve a strict single, or fixed-point design.

These checks are disabled by default.

## Enable the Highlight Option in a Project

- 1 Open the **Settings** menu.
- 2 Under **Plotting and Reporting**, set **Highlight potential data type issues** to Yes.

## Enable the Highlight Option at the Command Line

- 1 Create a fixed-point code configuration object:

```
cfg = coder.config('fixpt');
2 Set the HighlightPotentialDataTypeIssues property of the configuration object to true.
cfg.HighlightPotentialDataTypeIssues = true;
```

## Stowaway Doubles

When trying to achieve a strict-single or fixed-point design, manual inspection of code can be time-consuming and error prone. This check highlights all expressions that result in a double operation.

## Stowaway Singles

This check highlights all expressions that result in a single operation.

## Expensive Fixed-Point Operations

The expensive fixed-point operations check identifies optimization opportunities for fixed-point code. It highlights expressions in the MATLAB code that require cumbersome multiplication or division, expensive rounding, expensive comparison, or multiword operations. For more information on optimizing generated fixed-point code, see “[Tips for Making Generated Code More Efficient](#)”.

### Cumbersome Operations

Cumbersome operations most often occur due to insufficient range of output. Avoid inputs to a multiply or divide operation that has word lengths larger than the base integer type of your processor. Operations with larger word lengths can be handled in software, but this approach requires much more code and is much slower.

### Expensive Rounding

Traditional handwritten code, especially for control applications, almost always uses “no effort” rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the `floor` rounding method. This check identifies expensive rounding operations in multiplication and division.

### Expensive Comparison Operations

Comparison operations generate extra code when a casting operation is required to do the comparison. For example, when comparing an unsigned integer to a signed integer, one of the inputs must first be cast to the signedness of the other before the comparison operation can be performed. Consider optimizing the data types of the input arguments so that a cast is not required in the generated code.

### Multiword Operations

Multiword operations can be inefficient on hardware. When an operation has an input or output data type larger than the largest word size of your processor, the generated code contains multiword operations. You can avoid multiword operations in the generated code by specifying local `fimath` properties for variables. You can also manually specify input and output word lengths of operations that generate multiword code.

# Working with Fixed-Point Code

This example shows HDL code generation from a fixed-point MATLAB® design that is ready for code generation.

## Introduction

The MATLAB code used in the example is an implementation of viterbi decoder modeled using fixed-point constructs.

```
design_name = 'mlhdlc_viterbi';
testbench_name = 'mlhdlc_viterbi_tb';
```

- 1** MATLAB Design: mlhdlc\_viterbi
- 2** MATLAB testbench: mlhdlc\_viterbi\_tb

Open the design function mlhdlc\_viterbi by clicking on the above link to notice the use of Fixed-Point Designer functions:

- 1** use of 'fi', 'numerictype', and 'fimath' for modeling fixed-point data types
- 2** use of 'bitget', 'bitsliceget', 'bitconcat' for modeling bit-wise operations

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fixpt_design'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Create a New HDL Coder™ Project

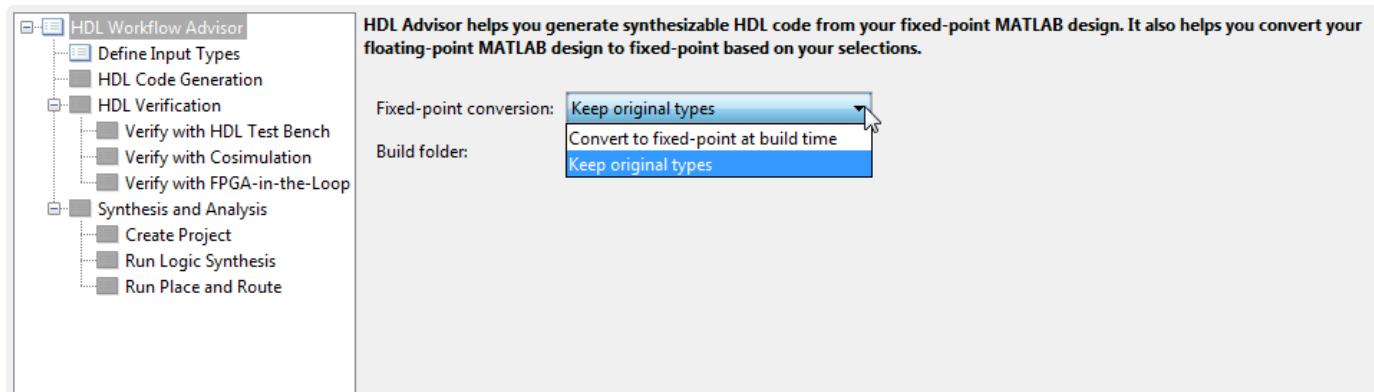
```
coder -hdlcoder -new fixpt_codegen
```

Next, add the file 'mlhdlc\_viterbi.m' to the project as the MATLAB Function and 'mlhdlc\_viterbi\_tb.m' as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

## Skip Fixed-Point Conversion

Launch the HDL Advisor and choose 'Keep original types' on the option 'Fixed-point conversion:'.



The Floating-point to fixed-point conversion related step is removed from the workflow tree when we skip the conversion.

If your design is in floating-point, follow the instructions in “Floating-Point to Fixed-Point Conversion” on page 4-49 and convert your design to fixed-point before moving onto the HDL code generation steps.

### Run HDL Code Generation

Right click on the 'Code Generation' step and choose the option 'Run this task' to run all code generation step directly.

Examine the generated HDL code by clicking on the hyperlinks in the Code Generation Log window.

### Try More Code Generation Options

As this is a large design with considerable number of functions you can try the option 'Generate instantiable code for functions' in the Advanced tab.

Re-examine the generated HDL code and compare it with the previous step.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fixpt_design'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Floating-Point to Fixed-Point Conversion

This example shows how to start with a floating-point design in MATLAB, iteratively converge on an efficient fixed-point design in MATLAB, and verify the numerical accuracy of the generated fixed-point design.

Signal processing applications for reconfigurable platforms require algorithms that are typically specified using floating-point operations. However, for power, cost, and performance reasons, they are usually implemented with fixed-point operations either in software for DSP cores or as special-purpose hardware in FPGAs. Fixed-point conversion can be very challenging and time-consuming, typically demanding 25 to 50 percent of the total design and implementation time. Automated tools can simplify and accelerate the conversion process.

For software implementations, the aim is to define an optimized fixed-point specification which minimizes the code size and the execution time for a given computation accuracy constraint. This optimization is achieved through the modification of the binary point location (for scaling) and the selection of the data word length according to the different data types supported by the target processor.

For hardware implementations, the complete architecture can be optimized. An efficient implementation will minimize both the area used and the power consumption. Thus, the conversion process goal typically is focused around minimizing the operator word length.

The floating-point to fixed-point workflow is currently integrated in the HDL Workflow Advisor as described in "Getting Started with MATLAB to HDL Workflow".

## Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

- 1 Verify that the floating-point design is compatible with code generation.
- 2 Compute fixed-point types based on the simulation of the testbench.
- 3 Generate readable and traceable fixed-point MATLAB code by applying proposed types.
- 4 Verify the generated fixed-point design.
- 5 Compare the numerical accuracy of the generated fixed-point code with the original floating point code.

## MATLAB Design

The MATLAB code used in this example is a simple second-order direct-form 2 transposed filter. This example also contains a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_df2t_filter';
testbench_name = 'mlhdlc_df2t_filter_tb';
```

Examine the MATLAB design.

```
type(design_name);

%%codegen
function y = mlhdlc_df2t_filter(x)

% Copyright 2011-2015 The MathWorks, Inc.
```

```

persistent z;
if isempty(z)
    % Filter states as a column vector
    z = zeros(2,1);
end

% Filter coefficients as constants
b = [0.29290771484375  0.585784912109375  0.292907714843750];
a = [1.0                 0.0                  0.171600341796875];

y      = b(1)*x + z(1);
z(1) = (b(2)*x + z(2)) - a(2) * y;
z(2) = b(3)*x - a(3) * y;

end

```

For the floating-point to fixed-point workflow, it is desirable to have a complete testbench. The quality of the proposed fixed-point data types depends on how well the testbench covers the dynamic range of the design with the desired accuracy.

For details on requirements for floating-point design and the testbench, see **Floating-Point Design Structure** structure section of “Working with Generated Fixed-Point Files” on page 4-66.

```

type(testbench_name);

%
% Copyright 2011-2015 The MathWorks, Inc.

Fs = 256;                      % Sampling frequency
Ts = 1/Fs;                      % Sample time
t = 0:Ts:1-Ts;                  % Time vector from 0 to 1 second
f1 = Fs/2;                      % Target frequency of chirp set to Nyquist
in = sin(pi*f1*t.^2);          % Linear chirp from 0 to Fs/2 Hz in 1 second
out = zeros(size(in));           % Output the same size as the input

for ii=1:length(in)
    out(ii) = mlhdlc_df2t_filter(in(ii));
end

% Plot
figure('Name', [mfilename, '_plot']);
subplot(2,1,1);
plot(in);
xlabel('Time')
ylabel('Amplitude')
title('Input Signal (with Noise)')

subplot(2,1,2);
plot(out);
xlabel('Time')
ylabel('Amplitude')
title('Output Signal (filtered)')

```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix_prj'];

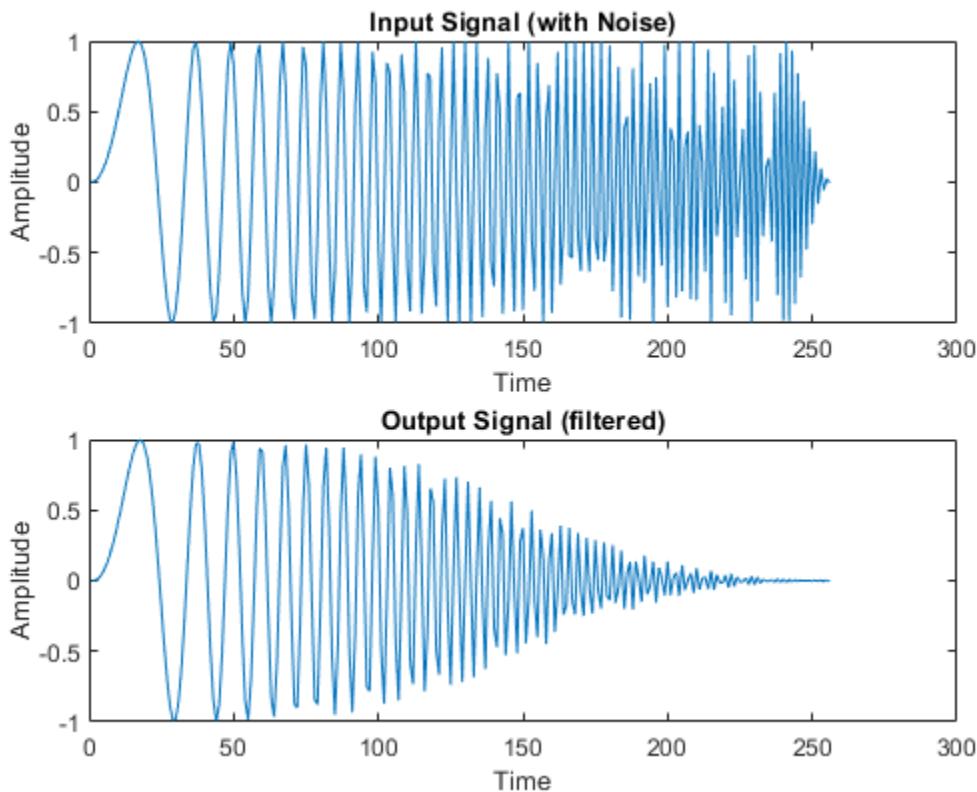
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_df2t_filter_tb
```



## Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc\_filter.m' to the project as the MATLAB Function and 'mlhdlc\_filter\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

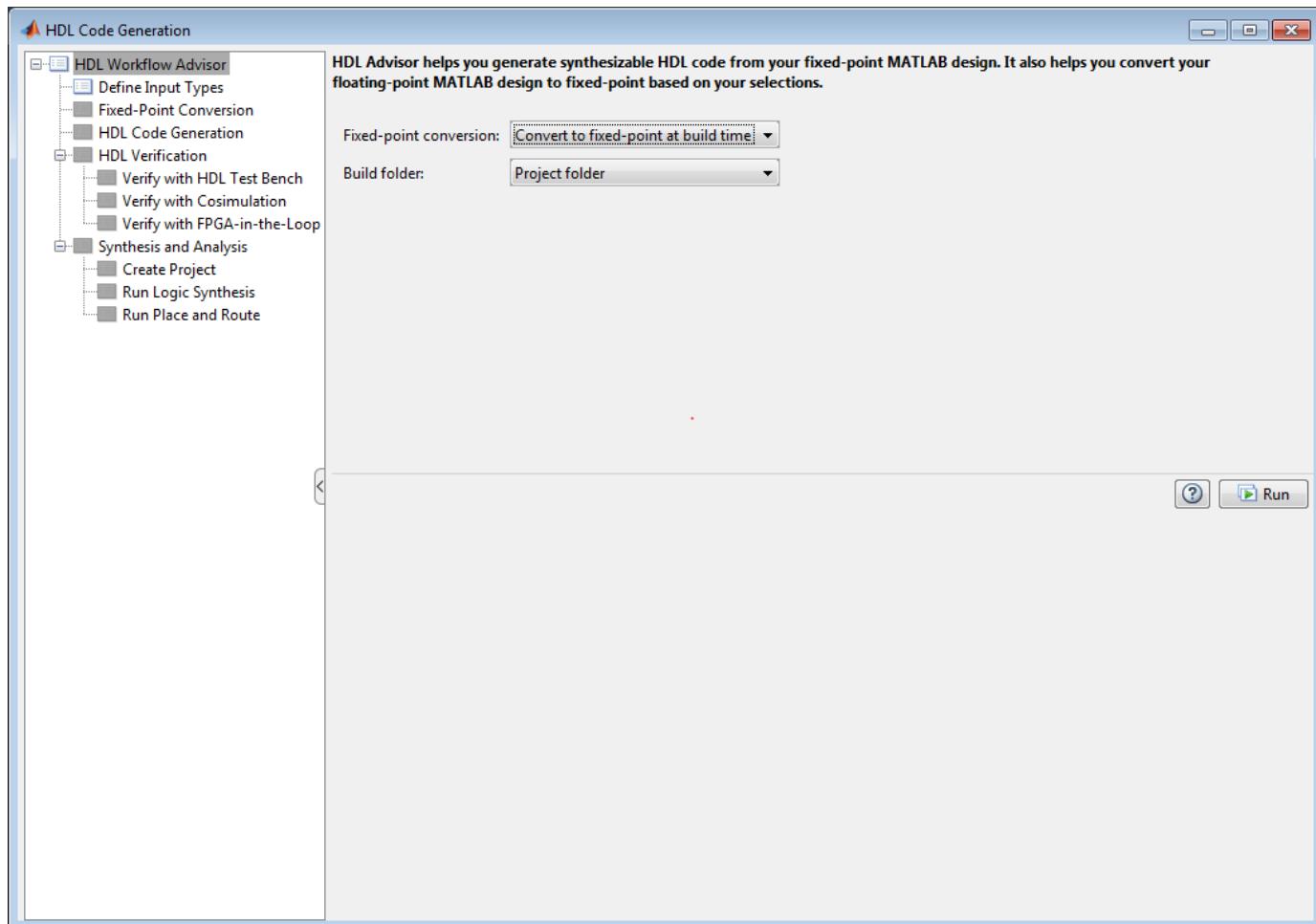
### Fixed-Point Code Generation Workflow

The floating-point to fixed-point conversion workflow allows you to:

- Verify that the floating-point design is code generation compliant
- Propose fixed-point types based on simulation data and word length settings
- Allow the user to manually adjust the proposed fixed-point types
- Validate the proposed fixed-point types
- Verify that the generated fixed-point MATLAB code has the desired numeric accuracy

### Step 1: Launch Workflow Advisor

- 1 Click on the Workflow Advisor button to launch the HDL Workflow Advisor.
- 2 Choose 'Convert to fixed-point at build time' for the option 'Fixed-point conversion'.



## Step 2: Define Input Types

In this step you can define input types manually or by specifying and running the testbench.

- Click 'Run' to execute this step.

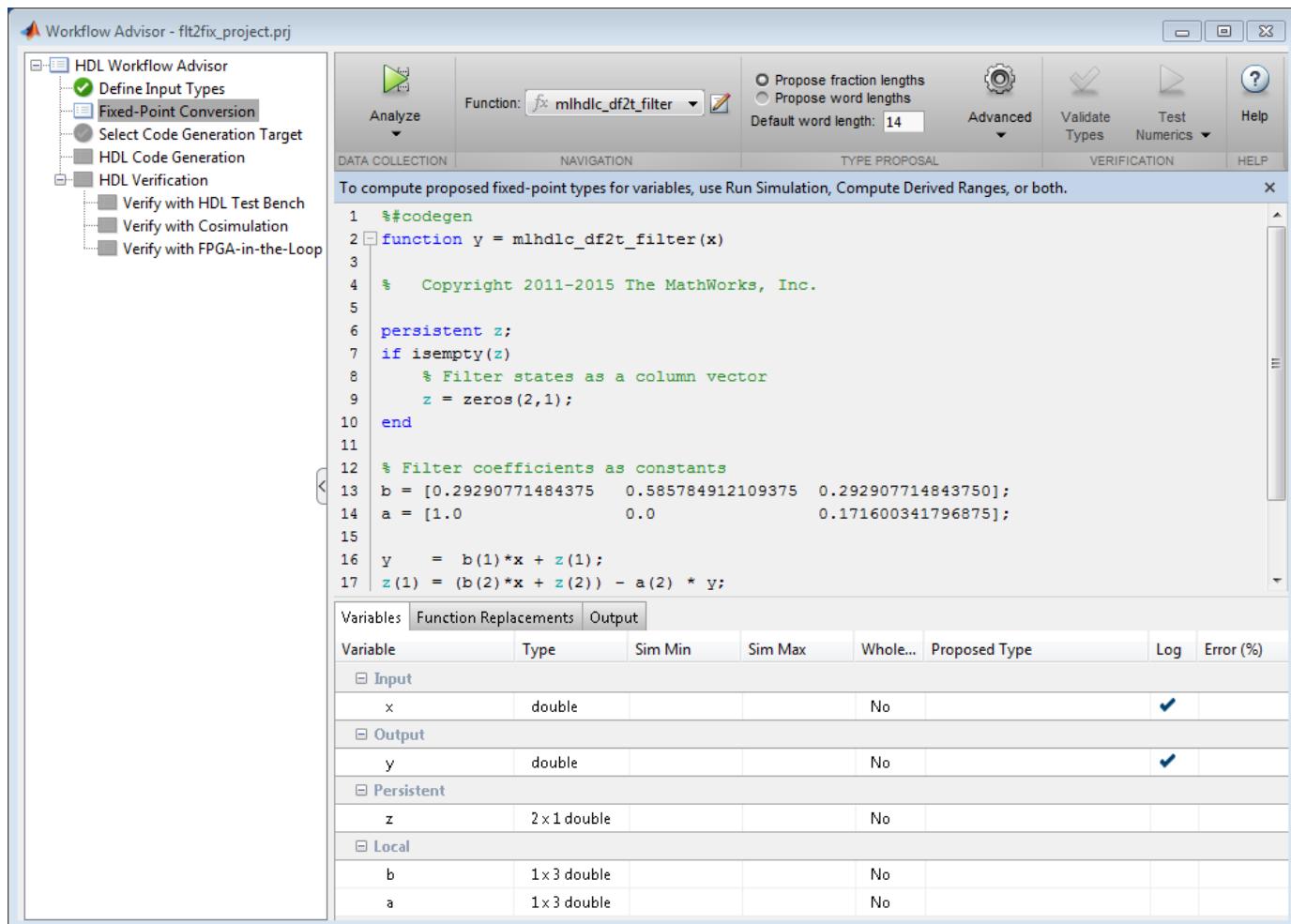
After simulation notice that the input variable 'x' is defined as scalar double 'double(1x1)'

## Step 3: Run Simulation

- Click on the 'Fixed-Point Conversion' step.

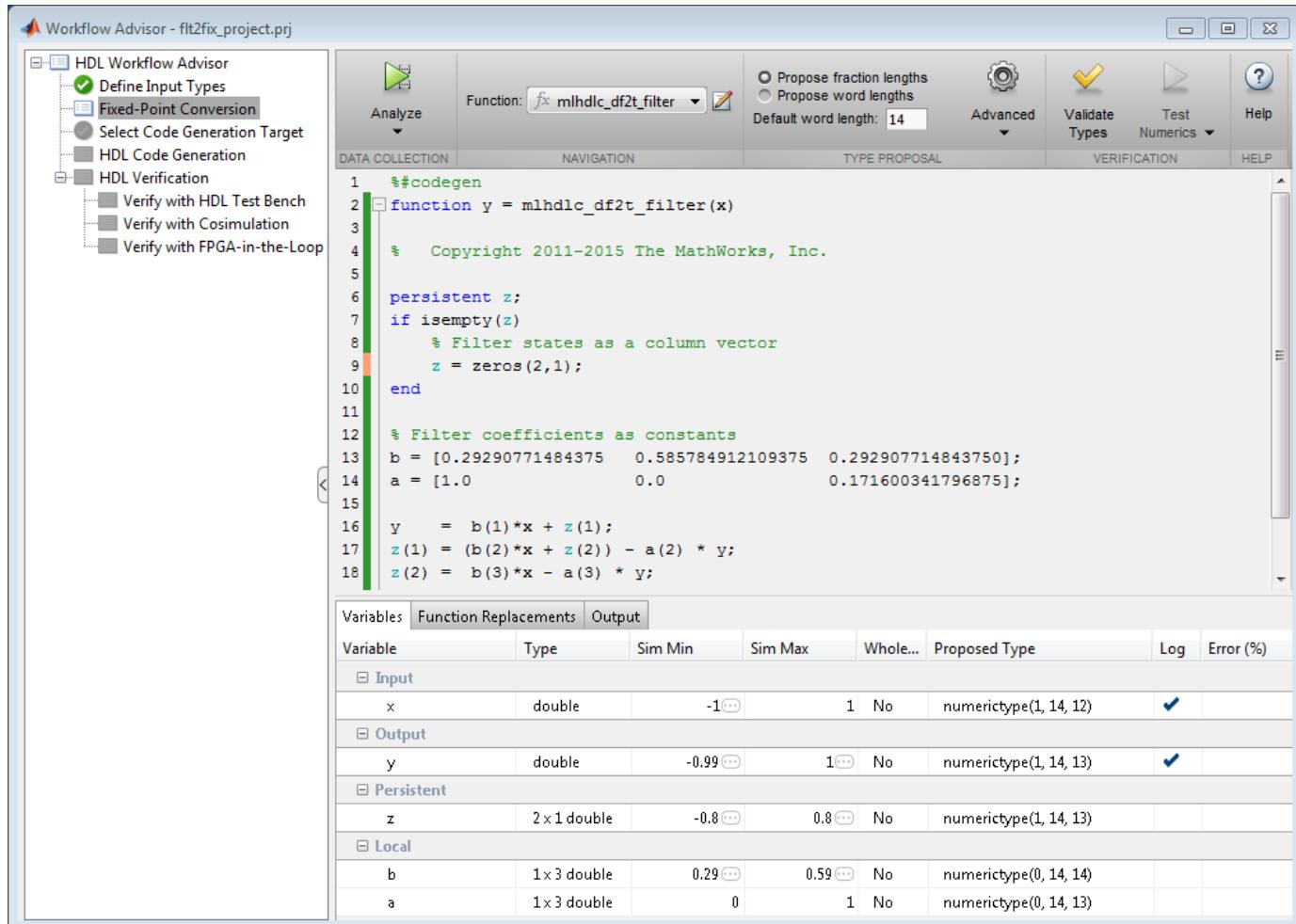
The design is compiled with the input types defined in the previous step and after the compilation is successful the variable table shows inferred types for all the functions in the design.

In this step, the original design is instrumented so that the minimum and maximum values for all variables in the design are collected during simulation.



- Click on the 'Analyze' button.

Notice that the 'Sim Min' and 'Sim Max' table is now populated with simulation ranges. Fixed-point types are proposed based on the default word length settings.



At this stage, based on computed simulation ranges for all variables, you can compute:

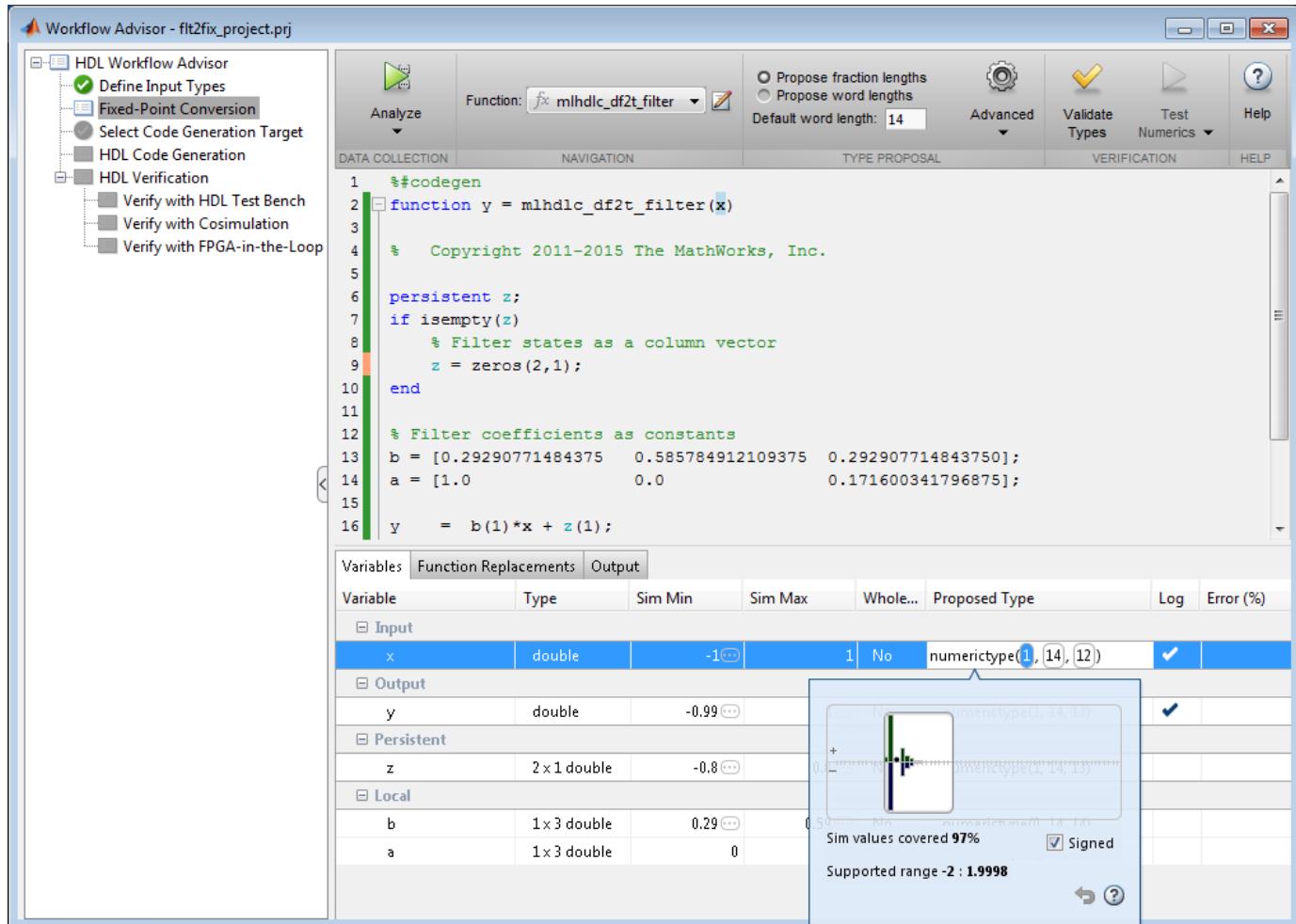
- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

The type table contains the following information for each variable existing in the floating-point MATLAB design, organized by function:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integers.

The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also enable the 'Log histogram data' option in the 'Analyze' button's menu to enable logging of histogram data.

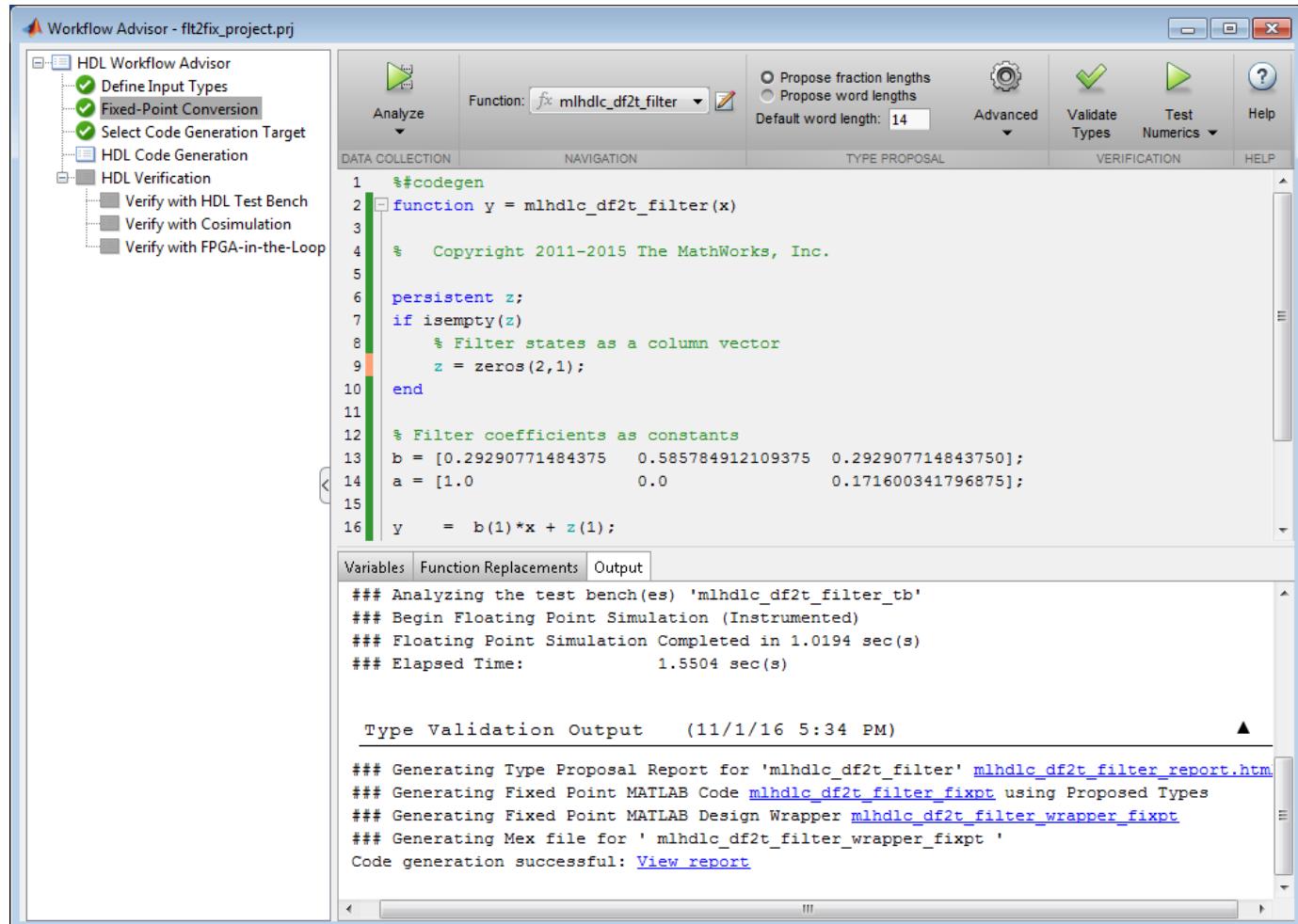


The histogram view concisely gives information about dynamic range of the simulation data for a variable. The x-axis correspond to bit weights and y-axis represents number of occurrences. The proposed numeric type information is overlaid on top of this graph and is editable. Moving the bounding white box left or right changes the position of binary point. Moving the right or left edges correspondingly change fraction length or wordlength. All the changes made to the proposed type are saved in the project.

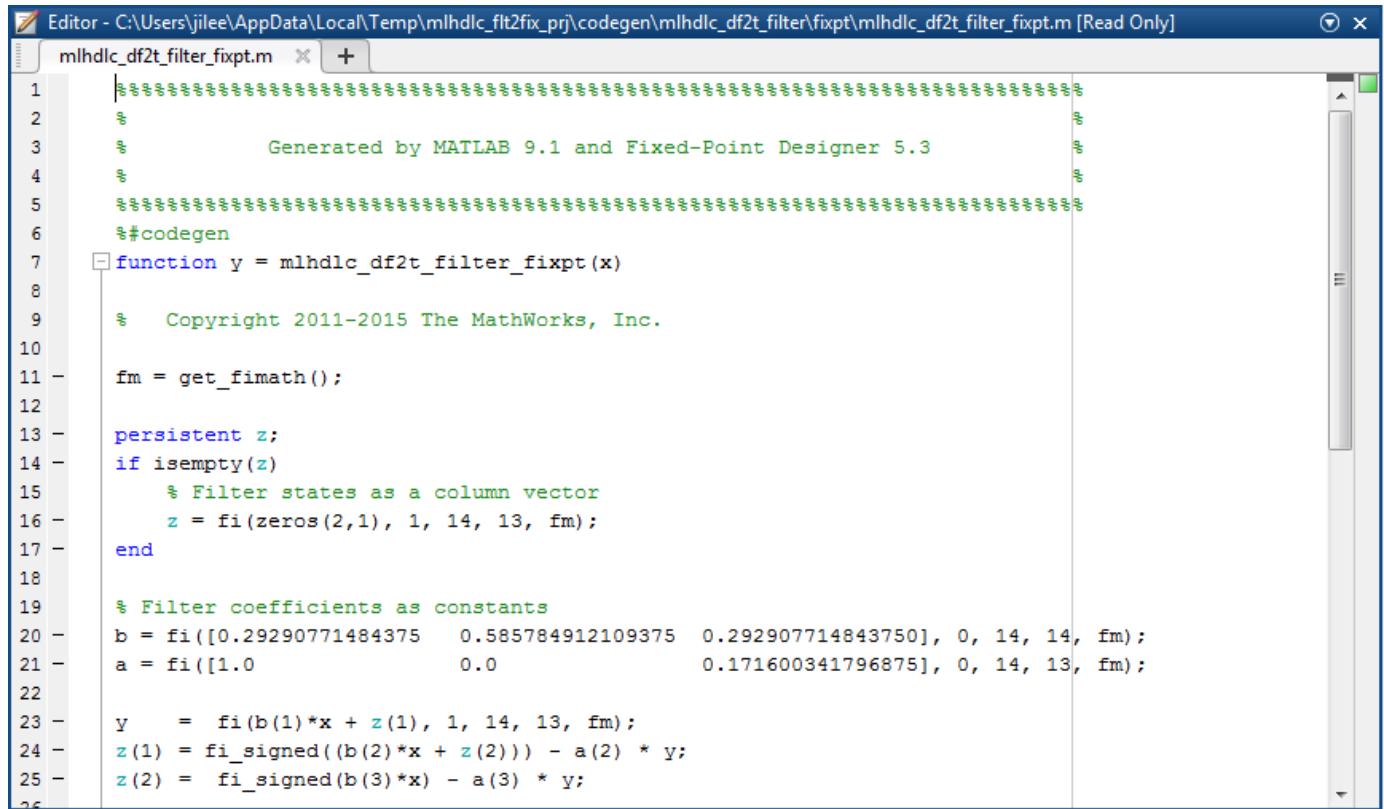
#### Step 4: Validate types

In this step, the fixed-point types from the previous step are used to generate a fixed-point MATLAB design from the original floating-point implementation.

- 1 Click on the 'Validate Types' button.



The generated code and other conversion artifacts are available via hyperlinks in the output window. The fixed-point types are explicitly shown in the generated MATLAB code.



```

Editor - C:\Users\jilee\AppData\Local\Temp\mlhdlc_flt2fix_prj\codegen\mlhdlc_df2t_filter\fixpt\mlhdlc_df2t_filter_fixpt.m [Read Only]
mlhdlc_df2t_filter_fixpt.m + 1
1 % Generated by MATLAB 9.1 and Fixed-Point Designer 5.3
2 %
3 %#codegen
4 function y = mlhdlc_df2t_filter_fixpt(x)
5 % Copyright 2011-2015 The MathWorks, Inc.
6
7 fm = get_fimath();
8
9 persistent z;
10 if isempty(z)
11     % Filter states as a column vector
12     z = fi(zeros(2,1), 1, 14, 13, fm);
13 end
14
15 % Filter coefficients as constants
16 b = fi([0.29290771484375  0.585784912109375  0.292907714843750], 0, 14, 14, fm);
17 a = fi([1.0 0.0 0.171600341796875], 0, 14, 13, fm);
18
19 y = fi(b(1)*x + z(1), 1, 14, 13, fm);
20 z(1) = fi_signed((b(2)*x + z(2))) - a(2) * y;
21 z(2) = fi_signed(b(3)*x) - a(3) * y;
22
23
24
25
26

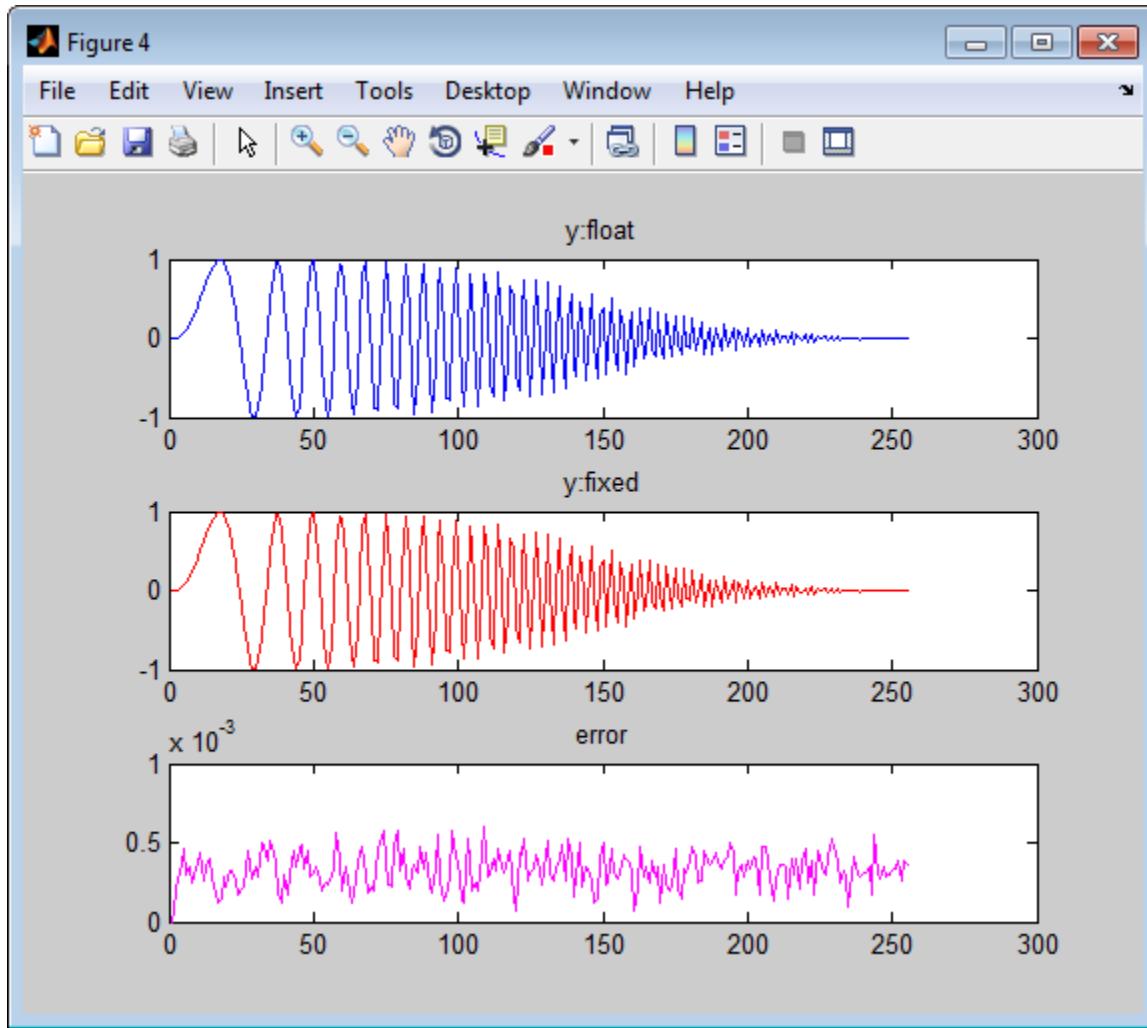
```

## Step 5: Test Numerics

- Click on the 'Test Numerics' button.

In this step, the generated fixed-point code is executed using MATLAB Coder.

If you enable the 'Log all inputs and outputs for comparison plots' option on the 'Test Numerics' pane, an additional plot is generated for each scalar output that shows the floating point and fixed point results, as well as the difference between the two. For non-scalar outputs, only the error information is shown.



### Step 6: Iterate on the Results

If the numerical results do not meet your desired accuracy after fixed-point simulation, you can return to the 'Propose Fixed-Point Types' step in the Workflow Advisor. Adjust the word length settings or individually modify types as desired, and repeat the rest of the steps in the workflow until you achieve your desired results.

Refer to “Fixed-Point Type Conversion and Refinement” on page 4-59 for more details on how to iterate and refine the numerics of the algorithm in the generated fixed-point code.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix_prj'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Fixed-Point Type Conversion and Refinement

This example shows how to achieve your desired numerical accuracy when converting fixed-point MATLAB® code to floating-point code using the HDL Workflow Advisor.

## Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

- 1** Verify the floating-point design is compatible for code generation.
- 2** Compute fixed-point types based on the simulation of the testbench.
- 3** Generate readable and traceable fixed-point MATLAB® code.
- 4** Verify the generated fixed-point design.

This tutorial uses Kalman filter suitable for HDL code generation to illustrate some key aspects of fixed-point conversion workflow, specifically steps 2 and 3 in the above list.

## MATLAB Design

The MATLAB code used in this example implements a simple Kalman filter. This example also contains a MATLAB testbench that exercises the filter.

### Kalman filter implementation suitable for HDL code generation

```
design_name = 'mlhdlc_kalman_hdl';
testbench_name = 'mlhdlc_kalman_hdl_tb';

edit('mlhdlc_kalman_hdl')
edit('mlhdlc_kalman_hdl_tb')
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

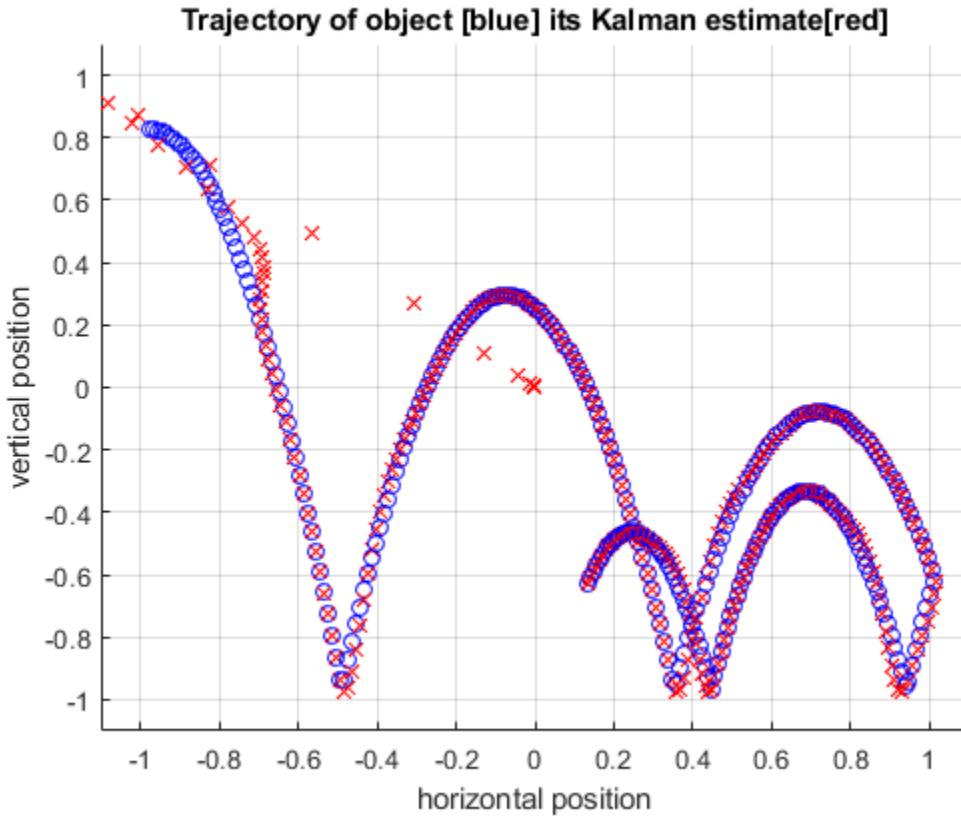
## Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_kalman_hdl_tb

Running -----> mlhdlc_kalman_hdl_tb

Current plot held
Current plot released
```



### Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project
```

Next, add the file 'mlhdlc\_kalman\_hdl.m' to the project as the MATLAB Function and 'mlhdlc\_kalman\_hdl\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Fixed-Point Code Generation Workflow

Perform the following tasks before moving on to the fixed-point type proposal step:

- 1 Click the 'Workflow Advisor' button to launch the HDL Workflow Advisor.
- 2 Choose 'Convert to fixed-point at build time' for the 'Fixed-point conversion' option.
- 3 Click 'Run' button to define input types for the design from the testbench.
- 4 Select the 'Fixed-Point Conversion' workflow step.
- 5 Click 'Analyze' to execute the instrumented floating-point simulation.

Refer to "Floating-Point to Fixed-Point Conversion" on page 4-49 for a more complete tutorial on these steps.

## Determine the Initial Fixed Point Types

After instrumented floating-point simulation completes, you will see 'Fixed-Point Types are proposed' based on the simulation results.

At this stage of the conversion proposes fixed-point types for each variable in the design based on the recorded min/max values of the floating point variables and user input.

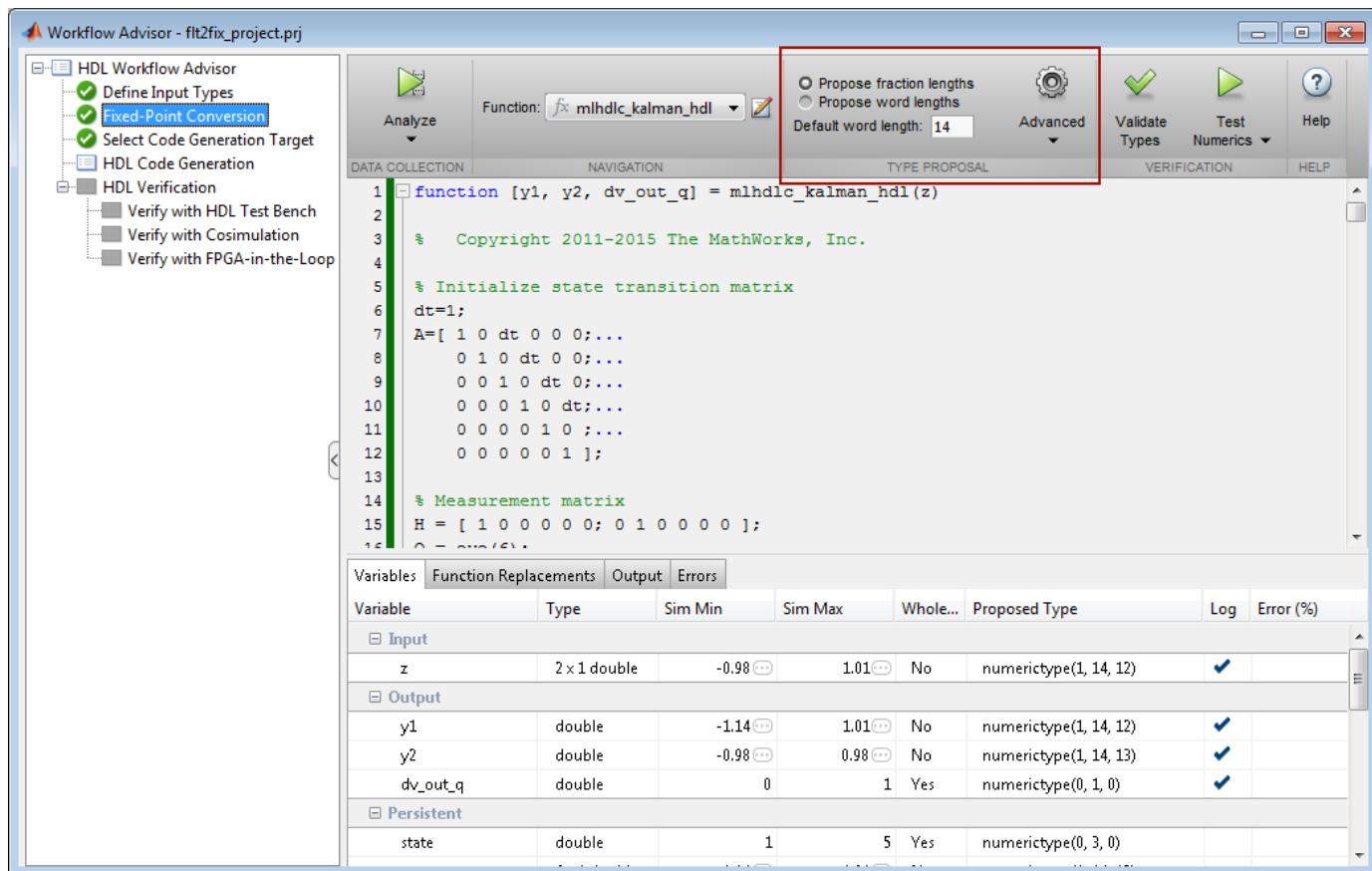
At this point, for all variables, you can (re)compute and propose:

- Fraction lengths for a given fixed word length setting, or
- Word lengths for a given fixed fraction length setting.

## Choose the Word Length Setting

When you are starting with a floating-point design and going through the floating-point to fixed-point conversion for the first time, it is a good practice to start by specifying a 'Default Word Length' setting based on the largest dynamic range of all the variables in the design.

In this example, we start with a default word length of 22 and run the 'Propose Fixed-Point Types' step.



## Explore the Proposed Fixed-Point Type Table

The type table contains the following information for each variable, organized by function, existing in the floating-point MATLAB design:

- Sim Min: The minimum value assigned to the variable during simulation.
- Sim Max: The maximum value assigned to the variable during simulation.
- Whole Number: Whether all values assigned during simulation are integer.

The type proposal step uses the above information and combines it with the user-specified word length settings to propose a fixed-point type for each variable.

You can also use 'Compute Derived Range Analysis' to compute derived ranges and that is covered in detail in this tutorial "Fixed-Point Type Conversion and Derived Ranges" on page 4-72.

### Interpret the Proposed Numeric Types for Variables

Based on the simulation range (min & max) values and the default word length setting, a numeric type is proposed for each variable.

The following table shows numeric type proposals for a 'Default word length' of 22 bits.

Variables	Function Replacements	Output	Errors				
Variable	Type	Sim Min	Sim Max	Whole...	Proposed Type	Log	Error (%)
<b>Input</b>							
z	2x1 double	-0.98	1.01	No	numerictype(1, 14, 12)		
<b>Output</b>							
y1	double	-1.14	1.01	No	numerictype(1, 14, 12)		
y2	double	-0.98	0.98	No	numerictype(1, 14, 13)		
dv_out_q	double	0	1	Yes	numerictype(0, 1, 0)		
<b>Persistent</b>							
state	double	1	5	Yes	numerictype(0, 3, 0)		
x_est	6x1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
p_est	6x6 double	0	472.78	No	numerictype(0, 14, 5)		
y	2x1 double	-1.14	1.01	No	numerictype(1, 14, 12)		
x_prd	6x1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
p_prd	6x6 double	0	896.74	No	numerictype(0, 14, 4)		
z_prd	2x1 double	-1.35	1.17	No	numerictype(1, 14, 12)		
S	2x2 double	0	1896.74	No	numerictype(0, 14, 3)		
B	2x6 double	0	896.74	No	numerictype(0, 14, 4)		
klm_gain	6x2 double	0	0.47	No	numerictype(0, 14, 15)		
dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
backslash_dv_out	double	0	1	Yes	numerictype(0, 1, 0)		
<b>Local</b>							
dt	double	1	1	Yes	numerictype(0, 1, 0)		
A	6x6 double	0	1	Yes	numerictype(0, 1, 0)		
H	2x6 double	0	1	Yes	numerictype(0, 1, 0)		
Q	6x6 double	0	1	Yes	numerictype(0, 1, 0)		
R	2x2 double	0	1000	Yes	numerictype(0, 10, 0)		

Examine the types proposed in the above table for variables instrumented in the top-level design.

Floating-Point Range for variable 'B':

- Simulation Info: SimMin: 0, SimMax: 896.74.., Whole Number: No
- Type Proposed: numerictype(0,22,12) (Signedness: Unsigned, WordLength: 22, FractionLength: 12)

The floating-point range:

- Has the same number of bits as the 'Default word length'.
- Uses the minimum number of bits to completely represent the range.
- Uses the rest of the bits to represent the precision.

Integer Range for variable 'A':

- Simulation Info: SimMin: 0, SimMax: 1, Whole Number: Yes
- Type Proposed: numerictype(0,1,0) (Signedness: Unsigned, WordLength: 1, FractionLength: 0)

The integer range:

- Has the minimum number of bits to represent the whole integer range.
- Has no fractional bits.

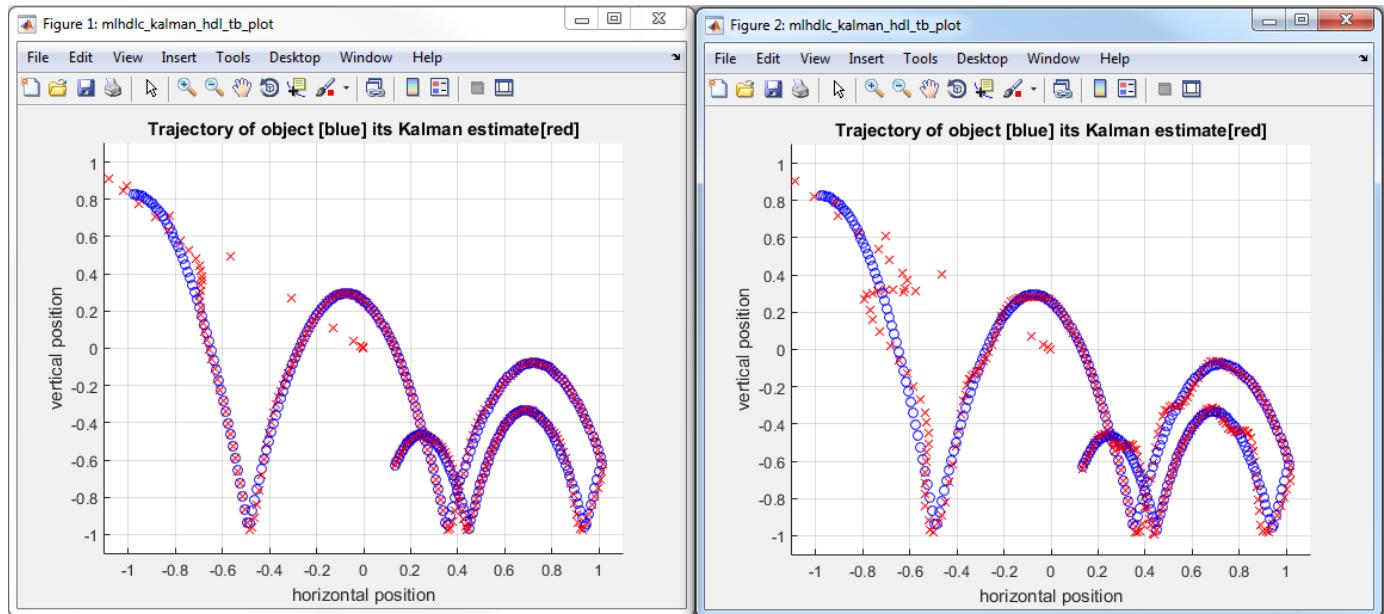
All the information in the table is editable, persists across iterations, and is saved with your code generation project.

### **Generate Fixed-Point Code and Verify the Generated Code**

Based on the numeric types proposed for a default word length of 22, continue with fixed-point code generation and verification steps and observe the plots.

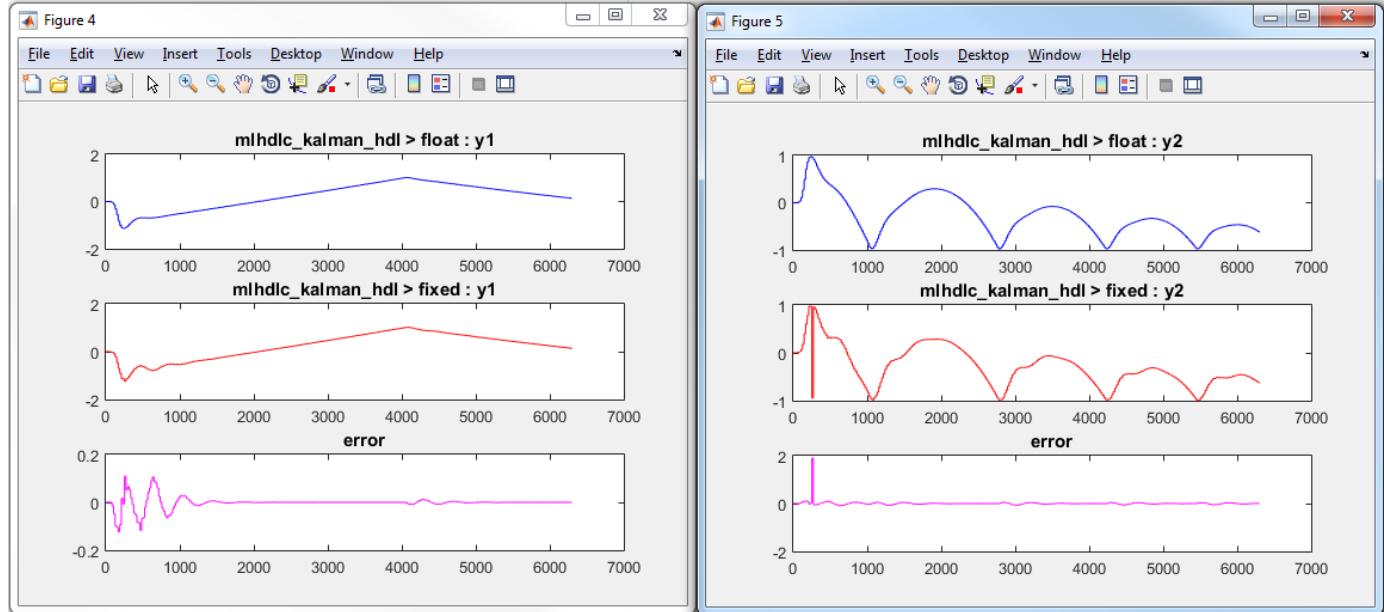
- 1 Click on 'Validate Types' to apply computed fixed-point types.
- 2 Next choose the option 'Log inputs and outputs for comparison plots' and then click on the 'Test Numerics' to rerun the testbench on the fixed-point code.

The plot on the left is generated from testbench during the simulation of floating-point code, the one on the right is generated from the simulation of the generated fixed-point code. Notice, the plots do not match.



Having chosen comparison plots option you will see additional plots that compare the floating and fixed point simulation results for each output variable.

Examine the error graph for each output variable. It is very high for this particular design.



### Iterate on the Results

One way to reduce the error is to increase 'Default word length' and repeat the fixed-point conversion.

In this example design, when a word length of 22 bits is chosen there is a lot of truncation error when representing the precision. More bits are required to the right of the binary point to reduce the truncation errors.

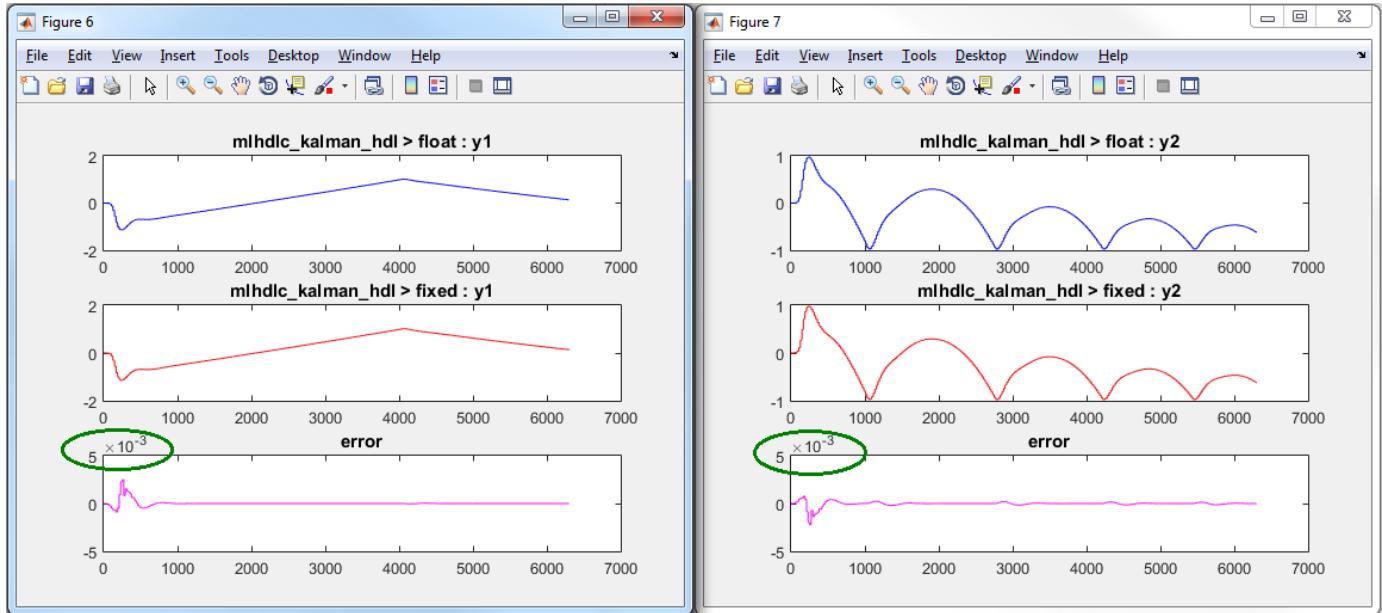
Let us now increase the default word length to 28 bits and repeat the type proposal and validation steps.

- 1 Select a 'Default word length' of 28.

Changing default word length automatically triggers the type proposal step and new fixed-point types are proposed based on the new word length setting. Also notice that type validation needs to be rerun and numerics need to be verified again.

- 1 Click on 'Validate Types'.
- 2 Click on 'Test Numerics' to rerun the testbench on the fixed-point code.

Once these steps are complete, re-examine the comparison plots and notice that the error is now roughly three orders of magnitude smaller.



### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Working with Generated Fixed-Point Files

This example shows how to work with the files generated during floating-point to fixed-point conversion.

### Introduction

This tutorial uses a simple filter implemented in floating-point and an associated testbench to illustrate the file structure of the generated fixed-point code.

```
design_name = 'mlhdlc_filter';
testbench_name = 'mlhdlc_filter_tb';
```

### MATLAB® Code

- 1 MATLAB Design: mlhdlc\_filter
- 2 MATLAB testbench: mlhdlc\_filter\_tb

### Create a New Folder and Copy Relevant Files

Executing the following lines of code copies the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];

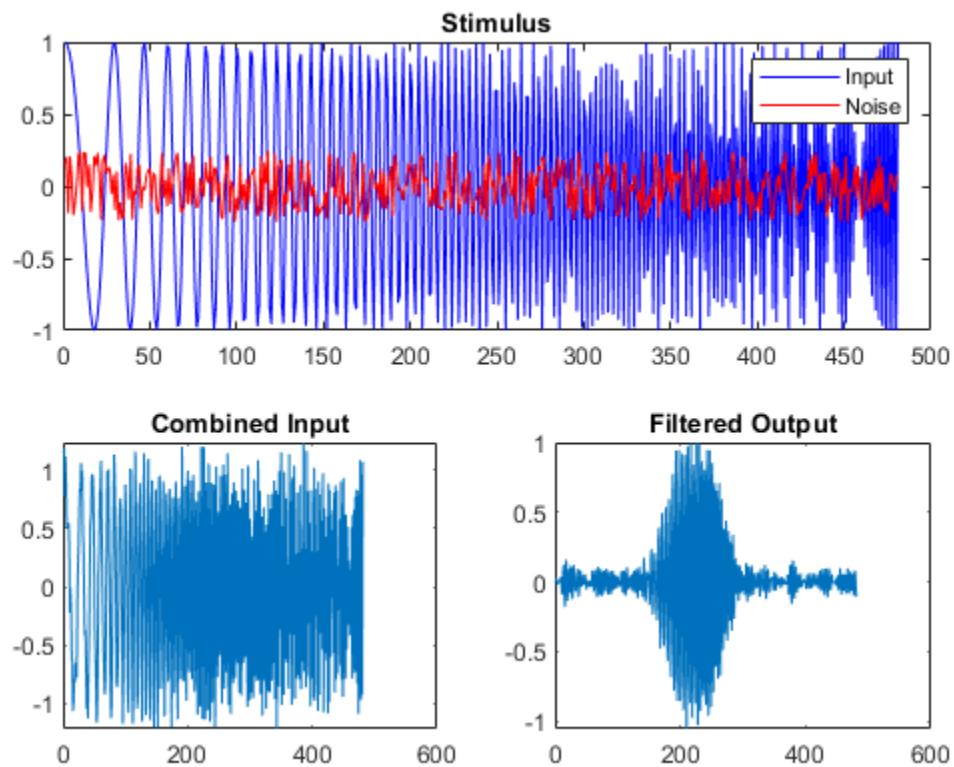
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

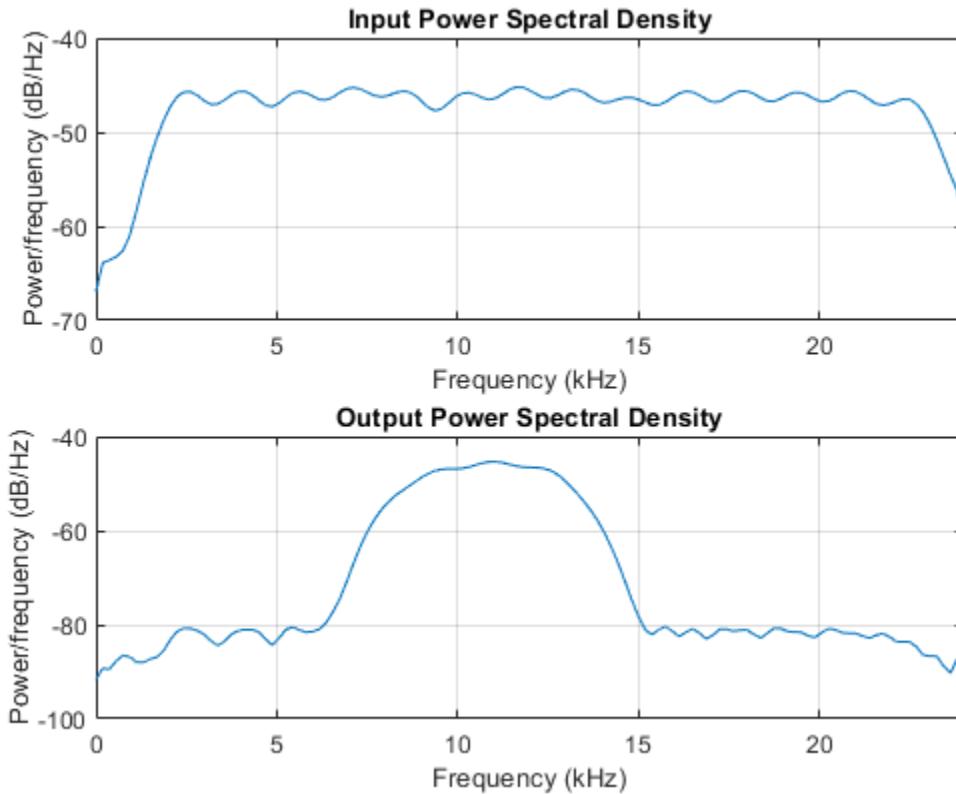
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_filter_tb
```





### Create a New HDL Coder™ Project

To create a new project, enter the following command:

```
coder -hdlcoder -new filt2fix_project
```

Next, add the file 'mlhdlc\_filter' to the project as the MATLAB Function and 'mlhdlc\_filter\_tb' as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Fixed-Point Code Generation Workflow

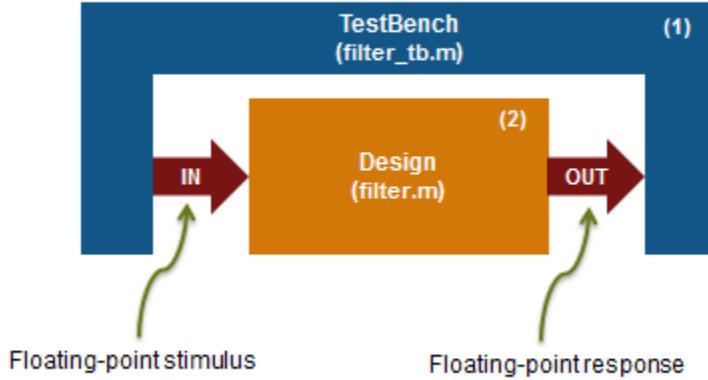
Perform the following tasks in preparation for the fixed-point code generation step:

- 1 Click the **Workflow Advisor** button to launch the Workflow Advisor.
- 2 Choose Convert to fixed-point at build time for the option **Fixed-point conversion**.
- 3 Right-click the **Fixed-Point Conversion** step and select **Run to Selected Task** to execute the instrumented floating-point simulation.

Refer to “Floating-Point to Fixed-Point Conversion” on page 4-49 for a more complete tutorial on these steps.

### Floating-Point Design Structure

The original floating-point design and testbench have the following relationship.



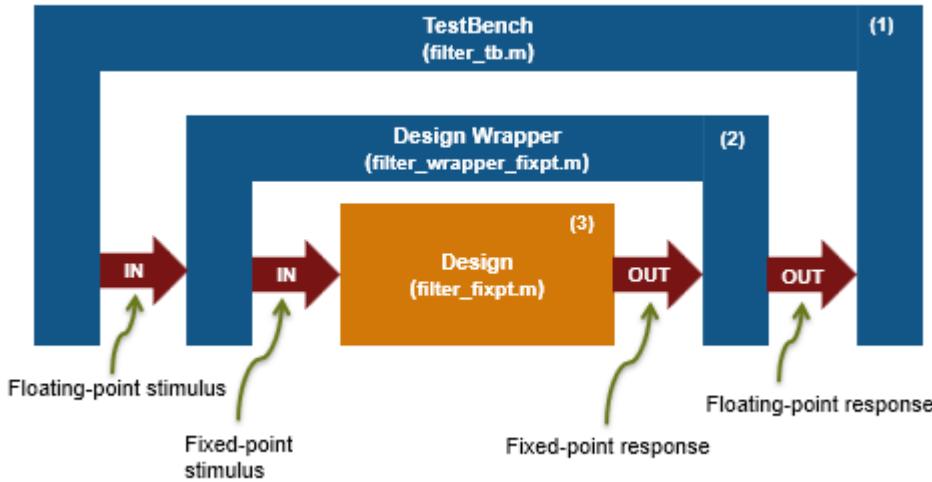
For floating-point to fixed-point conversion, the following requirements apply to the original design and the testbench:

- The testbench 'mlhdlc\_filter\_tb.m' (1) must be a script or a function with no inputs.
- The design 'mlhdlc\_filter.m' (2) must be a function.
- There must be at least one call to the design from the testbench. All call sites contribute when determining the proposed fixed-point types.
- Both the design and testbench can call other sub-functions within the file or other functions on the MATLAB path. Functions that exist within matlab/toolbox are not converted to fixed-point.

In the current example, the MATLAB testbench 'mlhdlc\_filter\_tb' has a single call to the design function 'mlhdlc\_filter'. The testbench calls the design with floating-point inputs and accumulates the floating-point results for plotting.

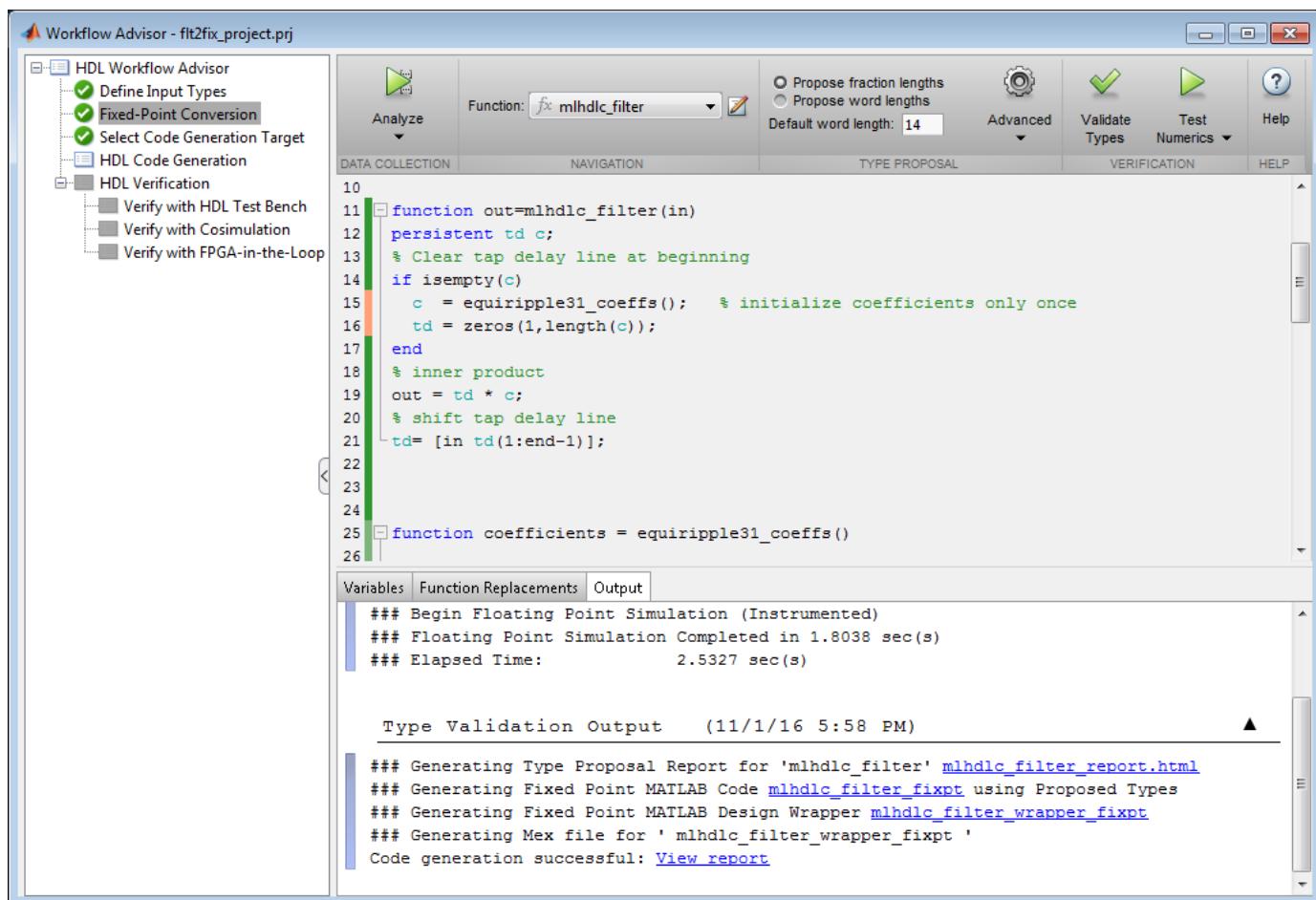
### **Validate Types**

During the type validation step, fixed-point code is generated for this design and compiled to verify that there are no errors when applying the types. The output files will have the following structure.



The following steps are performed during fixed-point type validation process:

- 1 The design file '`mlhdlc_filter.m`' is converted to fixed-point to generate fixed-point MATLAB code, '`mlhdlc_filter_fixpt.m`' (3).
- 2 All user-written functions called in the floating-point design are converted to fixed point and included in the generated design file.
- 3 A new design wrapper file is created, called '`mlhdlc_filter_wrapper_fixpt.m`' (2). This file converts the floating-point data values supplied by the testbench to the fixed-point types determined for the design inputs during the conversion step. These fixed point values are fed into the converted fixed-point design, '`mlhdlc_filter_fixpt.m`'.
- 4 '`mlhdlc_filter_fixpt.m`' will be used for HDL code generation.
- 5 All the generated fixed-point files are stored in the output directory '`codegen/mlhdlc_filter/fixpt`'.



Click the links to the generated code in the Workflow Advisor log Window to examine the generated fixed-point design and wrapper.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde');
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');

```

## Fixed-Point Type Conversion and Derived Ranges

This example shows how to achieve your desired numerical accuracy when converting fixed-point MATLAB® code to floating-point code using static range analysis which helps to compute derived ranges of the variables from design ranges.

### Introduction

The floating-point to fixed-point conversion workflow in HDL Coder™ includes the following steps:

- 1 Verify the floating-point design is compatible for code generation.
- 2 Compute fixed-point types based on the simulation of the testbench.
- 3 Generate readable and traceable fixed-point MATLAB® code.
- 4 Verify the generated fixed-point design.

However, the fixed-point types proposed from the simulation depends on the quality of the testbench. Sometimes it is hard to write testbenches which completely cover paths of the design representing full design ranges of all the variables. Static analysis based workflow can be used in such cases to compute derived ranges from design ranges.

This tutorial uses a symmetric FIR filter whose output signal is integrated over time.

### MATLAB Design

The MATLAB code used in this example implements a simple Kalman filter. This example also contains a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_dti';
testbench_name = 'mlhdlc_dti_tb';
```

- 1 MATLAB Design: mlhdlc\_dti
- 2 MATLAB testbench: mlhdlc\_dti\_tb

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix_dmm'];

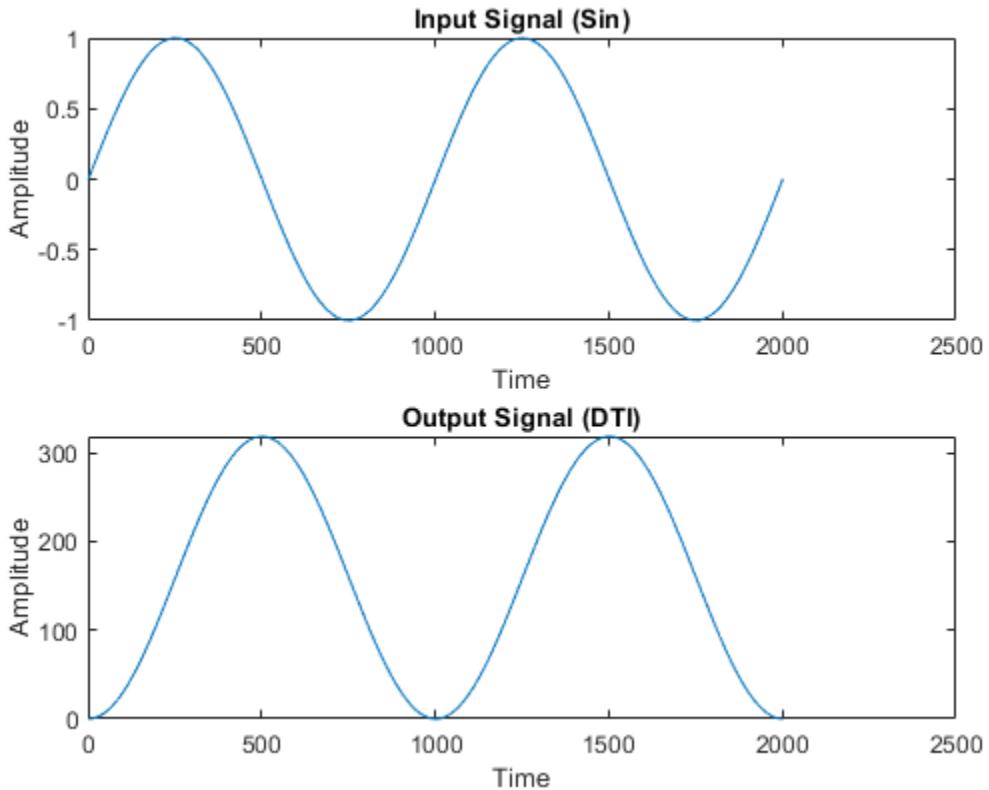
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_dti_tb
```



### Create a New HDL Coder Project

To create a new project, enter the following command:

```
coder -hdlcoder -new flt2fix_project_dmm
```

Next, add the file 'mlhdlc\_dti.m' to the project as the MATLAB Function and 'mlhdlc\_dti\_tb.m' as the MATLAB Test Bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Fixed-Point Code Generation Workflow

Perform the following tasks before moving on to the fixed-point type proposal step:

- 1** Click the 'Workflow Advisor' button to launch the HDL Workflow Advisor.
- 2** Choose 'Convert to fixed-point at build time' for the 'Fixed-point conversion' option.
- 3** Click 'Run' button to define input types for the design from the testbench.
- 4** Select the 'Fixed-Point Conversion' workflow step.
- 5** Click 'Analyze' to execute the instrumented floating-point simulation.

Refer to “Floating-Point to Fixed-Point Conversion” on page 4-49 for a more complete tutorial on these steps.

## Determine the Initial Fixed Point Types

After instrumented floating-point simulation completes, you will see 'Fixed-Point Types are proposed' based on the simulation results.

At this step fixed-point types for each variable in the design based on the recorded min/max values of the floating point variables and user input.

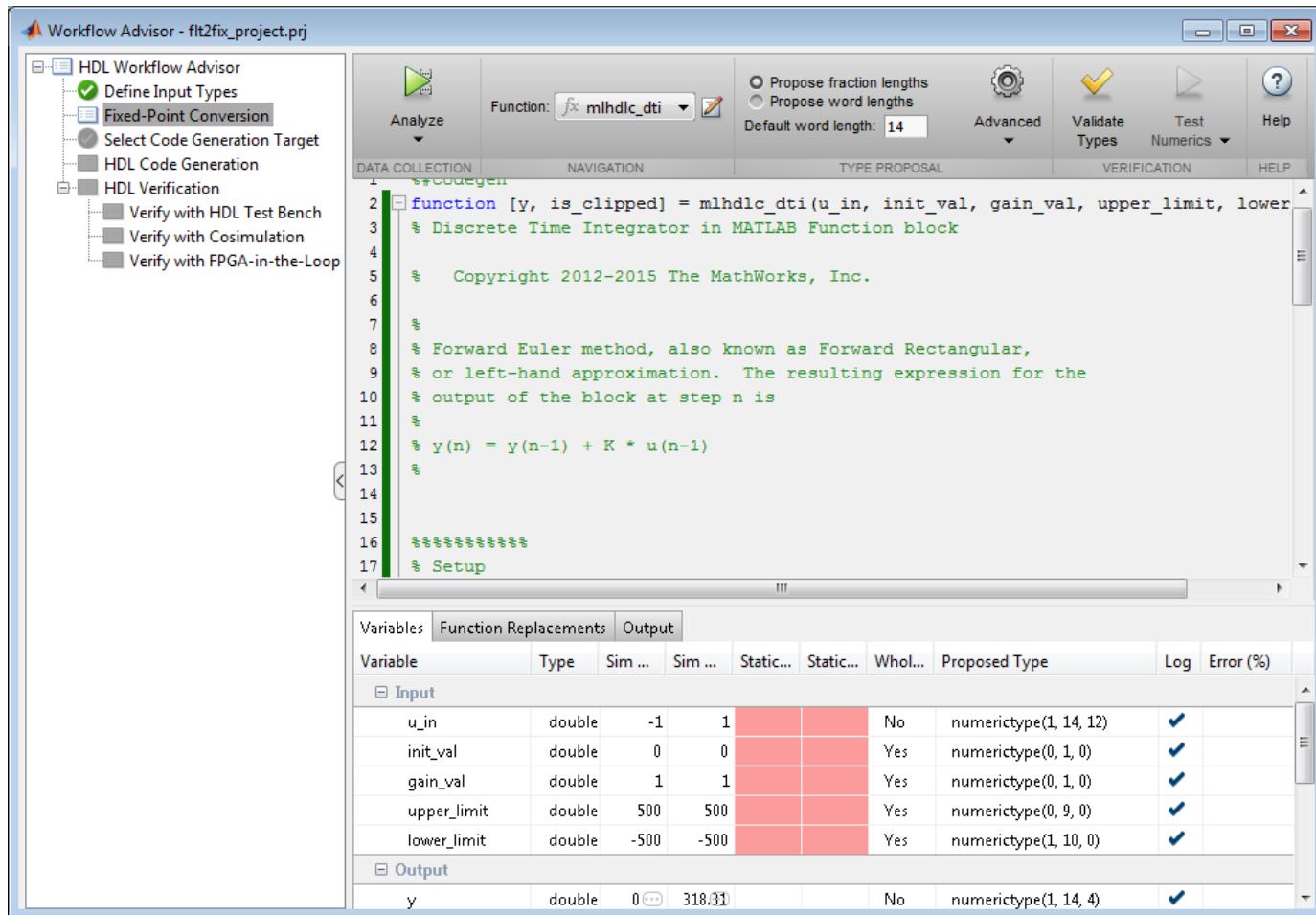
Observe the simulation range of the variable 'is\_clipped' in the function 'mlhdlc\_dti'. You will notice that the simulation range of this variable is a constant value 0. However, if you can observe the code to see that the variable can take values from -1 to 1.

The ranges for the variable can be fixed by updating the testbench. However, it may be desirable to compute program ranges through static analysis.

## Entering Design Ranges and Computing Derived Ranges

In this step you can specify design ranges and compute derived ranges through static analysis.

Enable derived range analysis by clicking the 'analyze ranges using derived range analysis' checkbox in the 'Analyze' button's menu. The tool will then prompt you to specify design ranges for the inputs variables in the Static Min and Static Max columns.



There are multiple ways you can enter design ranges.

- 1 You can manually edit the 'Static Min' and 'Static Max' entries in the table and specify design ranges.
  - 2 You can copy the Sim Min and Sim Max for a variable via right-clicking on the table cell (or)
  - 3 You can Lock or Specify the Output type to be used as the design range

Variable	Type	Sim ...	Sim ...	Static...	Static...	Whol...	Proposed Type	Log	Error (%)
<b>Input</b>									
u_in	double	-1	1				Copy sim range	<input checked="" type="checkbox"/>	
init_val	double	0	0				Copy sim ranges for all top-level inputs	<input checked="" type="checkbox"/>	
gain_val	double	1	1				Copy sim ranges for all persistent variables	<input checked="" type="checkbox"/>	
upper_limit	double	500	500				Copy sim ranges for all global variables	<input checked="" type="checkbox"/>	
lower_limit	double	-500	-500				Clear this static range	<input checked="" type="checkbox"/>	
<b>Output</b>									
y	double	0	318.31				Clear all manually entered static ranges	<input checked="" type="checkbox"/>	
is_clipped	double	0	0				Reset entire table	<input checked="" type="checkbox"/>	

Once all the necessary design ranges are specified you can click on the 'Analyze' button to use derived range analysis.

Notice that the derived range of the variable now includes values taken in all paths of the control flow.

### Insufficient design ranges

Sometimes specifying ranges for input variables alone may not be sufficient for certain designs. For example in a MATLAB design implementing a counter using a persistent variable, the range of the variable depends on number of times the design is called. In such situations you will see computed derived static ranges for the variable reported as -Inf or +Inf. When these imprecise ranges appear please consider specifying ranges for such persistent variables.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde  
mlhdlc_temp_dir = [tempdir 'mlhdlc_flt2fix'];  
clear mex;  
cd (mlhdlc_demo_dir);  
rmdir(mlhdlc_temp_dir, 's');
```

# Generate HDL-compatible lookup table function replacements using 'coder.approximate'

This example shows MATLAB code generation from a floating-point MATLAB® design that is not ready for code generation. We use 'coder.approximate' function to generate a lookup table based MATLAB function. This newly generated function is ready for HDL code generation (not shown in this demo).

## Introduction

The MATLAB code used in the example is sigmoid function, which is used for threshold detection and decision making problems. For example neural networks use sigmoid functions with appropriate thresholds to 'train' systems for learning patterns.

## MATLAB Design

```
design_name = 'mlhdlc_approximate_sigmoid';
testbench_name = 'mlhdlc_approximate_sigmoid_tb';
```

Lets look at the Sqrt Design

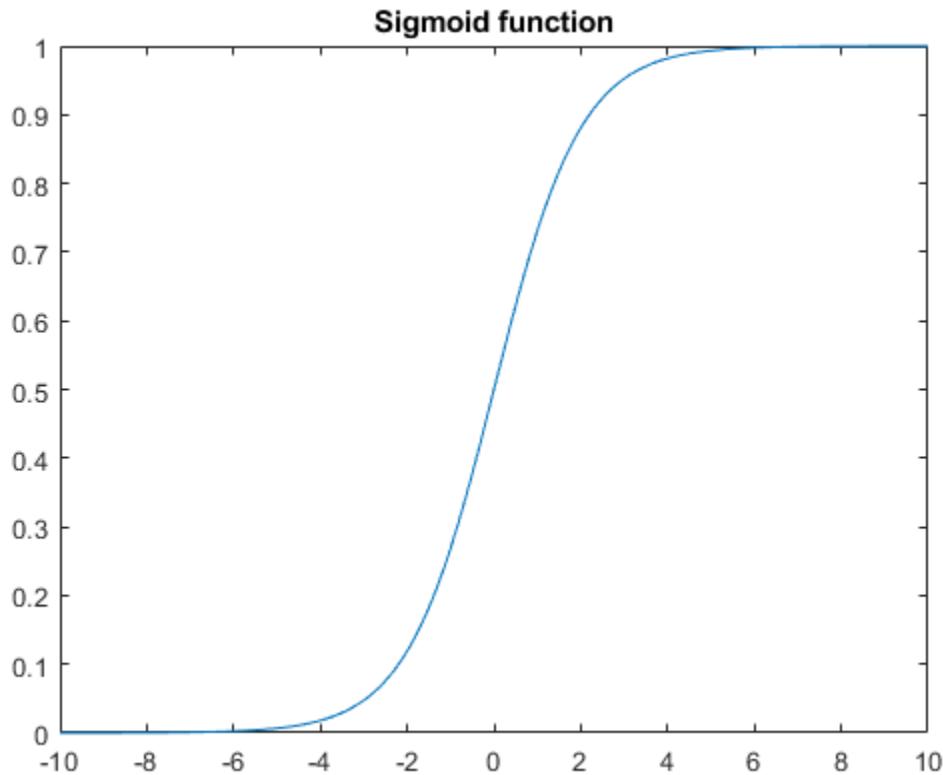
```
dbtype(design_name)

1      function y = mlhdlc_approximate_sigmoid( x )
2      %
3
4      % Copyright 2014-2015 The MathWorks, Inc.
5
6      y = 1./(1+exp(-x));
7      end
```

## Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_approximate_sigmoid_tb
```



- 1 MATLAB Design: mlhdlc\_approximate\_sigmoid
- 2 MATLAB testbench: mlhdlc\_approximate\_sigmoid\_tb

We can use coder.approximate to generate a lookup-table based replacement function for 'mlhdlc\_approximate\_sigmoid'

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fixpt_approximate'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '_design.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Generate fixed-point lookup-table replacements

```
repCfg = coder.approximation('Function', 'mlhdlc_approximate_sigmoid', 'CandidateFunction', @mlhdlc_
    'NumberOfPoints', 50, 'InputRange', [-10, 10], 'FunctionNamePrefix', 'repsig'
coder.approximate(repCfg);
```

First the fixed-point conversion completes with appropriate function replacements, and following console message,

```
### Generating approximation for 'sigmoid' : repsiglookuptable.m
### Generating testbench for 'sigmoid' : repsiglookuptable_tb.m
### LookupTable replacement for function 'sigmoid' used 50 data points
```

This should generate the MATLAB files 'repsig\_lookuputable\_tb', and 'repsig\_lookuputable' containing the testbench and design respectively.

### Test the replacement functions

To visually see the degree of match between lookup-table based replacement function and the original function use the testbench,

```
repsig_lookuputable_tb();
```

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_fixpt_approximate'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```



# Code Generation

---

- “Create and Set Up Your Project” on page 5-2
- “Specify Properties of Entry-Point Function Inputs” on page 5-4
- “Code Generation Reports” on page 5-7
- “Generate Instantiable Code for Functions” on page 5-11
- “Integrate Custom HDL Code Into MATLAB Design” on page 5-12
- “Enable MATLAB Function Block Generation” on page 5-17
- “System Design with HDL Code Generation from MATLAB and Simulink” on page 5-18
- “Specify the Clock Enable Rate” on page 5-21
- “Specify Test Bench Clock Enable Toggle Rate” on page 5-23
- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Lint Tool Script” on page 5-28
- “Generate HDL code from MATLAB functions using automated lookup table generation” on page 5-30
- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-33
- “Minimize Clock Enables” on page 5-37

# Create and Set Up Your Project

## In this section...

- “Create a New Project” on page 5-2
- “Open an Existing Project” on page 5-3
- “Add Files to the Project” on page 5-3

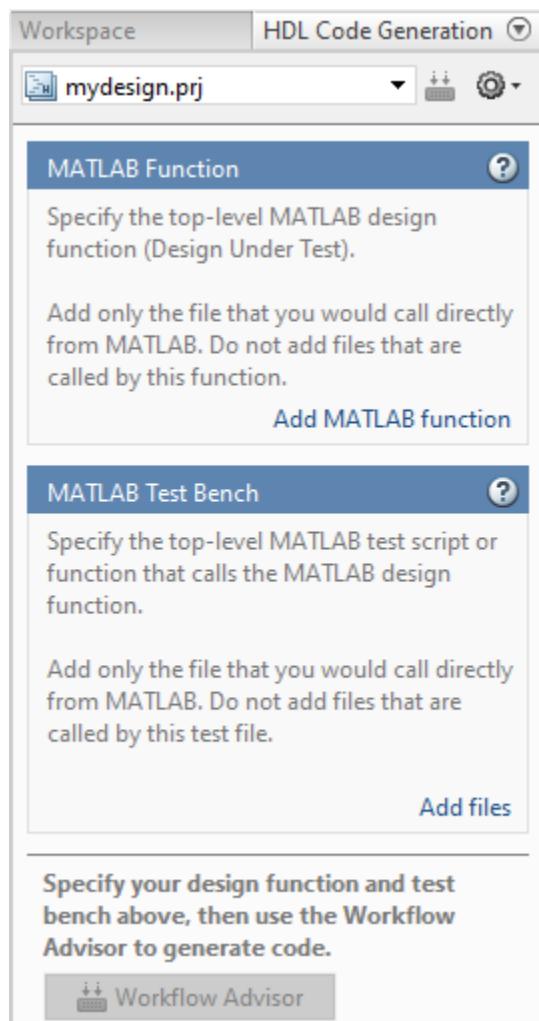
## Create a New Project

- 1 At the MATLAB command line, enter:

```
hdlcoder
```

- 2 Enter a project name in the project dialog box and click **OK**.

HDL Coder creates the project in the local working folder, and, by default, opens the project in the right side of the MATLAB workspace.



Alternatively, you can create a new HDL Coder project from the apps gallery:

- 1 On the **Apps** tab, on the far right of the **Apps** section, click the arrow ▾.
- 2 Under **Code Generation**, click **HDL Coder**.
- 3 Enter a project name in the project dialog box and click **OK**.

## Open an Existing Project

At the MATLAB command line, enter:

```
open project_name
```

where *project\_name* specifies the full path to the project file.

Alternatively, navigate to the folder that contains your project and double-click the *.prj* file.

## Add Files to the Project

### Add the MATLAB Function (Design Under Test)

First, you must add the MATLAB file from which you want to generate code to the project. Add only the top-level function that you call from MATLAB (the Design Under Test). Do not add files that are called by this file. Do not add files that have spaces in their names. The path must not contain spaces, as spaces can lead to code generation failures in certain operating system configurations.

To add a file, do one of the following:

- In the project pane, under **MATLAB Function**, click the **Add MATLAB function** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Function**.

If the functions that you added have inputs, and you do not specify a test bench, you must define these inputs. See “Specify Properties of Entry-Point Function Inputs” on page 5-4.

### Add a MATLAB Test Bench

You must add a MATLAB test bench unless your design does not need fixed-point conversion and you do not want to generate an RTL test bench. If you do not add a test bench, you must define the inputs to your top-level MATLAB function. For more information, see “Specify Properties of Entry-Point Function Inputs” on page 5-4.

To add a test bench, do one of the following:

- In the project panel, under **MATLAB Test Bench**, click the **Add MATLAB test bench** link and browse to the file.
- Drag a file from the current folder and drop it in the project pane under **MATLAB Test Bench**.

# Specify Properties of Entry-Point Function Inputs

## In this section...

- “When to Specify Input Properties” on page 5-4
- “Why You Must Specify Input Properties” on page 5-4
- “Properties to Specify” on page 5-4
- “Rules for Specifying Properties of Primary Inputs” on page 5-5
- “Methods for Defining Properties of Primary Inputs” on page 5-5

## When to Specify Input Properties

If you supply a test bench for your MATLAB algorithm, you do not need to specify the primary function inputs manually. The HDL Coder software uses the test bench to infer the data types.

## Why You Must Specify Input Properties

HDL Coder must determine the properties of all variables in the MATLAB files at compile time. To infer variable properties in MATLAB files, HDL Coder must be able to identify the properties of the inputs to the *primary* function, also known as the *top-level* or *entry-point* function. Therefore, if your primary function has inputs, you must specify the properties of these inputs, to HDL Coder. If your primary function has no input parameters, HDL Coder can compile your MATLAB file without modification. You do not need to specify properties of inputs to local functions or external functions called by the primary function.

If you use the tilde (~) character to specify unused function inputs in an HDL Coder project, and you want a different type to appear in the generated code, specify the type. Otherwise, the inputs default to real, scalar doubles.

## Properties to Specify

If your primary function has inputs, you must specify the following properties for each input.

For	Specify properties				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓
Other inputs	✓	✓	✓		

The following data types are not supported for primary function inputs, although you can use them within the primary function:

- structure
- matrix

Variable-size data is not supported in the test bench or the primary function.

## Default Property Values

HDL Coder assigns the following default values for properties of primary function inputs.

Property	Default
class	double
size	scalar
complexity	real
numerictype	No default
fimath	hdlfimath

### Supported Classes

The following table presents the class names supported by HDL Coder.

Class Name	Description
logical	Logical array of true and false values
char	Character array
int8	8-bit signed integer array
uint8	8-bit unsigned integer array
int16	16-bit signed integer array
uint16	16-bit unsigned integer array
int32	32-bit signed integer array
uint32	32-bit unsigned integer array
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
embedded.fi	Fixed-point number array

### Rules for Specifying Properties of Primary Inputs

When specifying the properties of primary inputs, follow these rules:

- You must specify the class of all primary inputs. If you do not specify the size or complexity of primary inputs, they default to real scalars.
- For each primary function input whose class is fixed point (*fi*), you must specify the input *numerictype* and *fimath* properties.

### Methods for Defining Properties of Primary Inputs

Method	Advantages	Disadvantages

Method	Advantages	Disadvantages
"Define Input Properties by Example at the Command Line"  <b>Note</b> If you define input properties programmatically in the MATLAB file, you cannot use this method	<ul style="list-style-type: none"> <li>Easy to use</li> <li>Does not alter original MATLAB code</li> <li>Designed for prototyping a function that has a few primary inputs</li> </ul>	<ul style="list-style-type: none"> <li>Must be specified at the command line every time you invoke (unless you use a script)</li> <li>Not efficient for specifying memory-intensive inputs such as large structures and arrays</li> </ul>
"Define Input Properties Programmatically in the MATLAB File"	<ul style="list-style-type: none"> <li>Integrated with MATLAB code; no need to redefine properties each time you invoke HDL Coder</li> <li>Provides documentation of property specifications in the MATLAB code</li> <li>Efficient for specifying memory-intensive inputs such as large structures</li> </ul>	<ul style="list-style-type: none"> <li>Uses complex syntax</li> <li>HDL Coder project files do not currently recognize properties defined programmatically. If you are using a project, you must reenter the input types in the project.</li> </ul>

## See Also

# Code Generation Reports

## In this section...

- “Report Generation” on page 5-7
- “Report Location” on page 5-7
- “Errors and Warnings” on page 5-8
- “Files and Functions” on page 5-8
- “MATLAB Source” on page 5-8
- “MATLAB Variables” on page 5-9
- “Additional Reports” on page 5-10
- “Report Limitations” on page 5-10

HDL Coder produces a code generation report that helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated HDL code.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.
- Access additional reports.

## Report Generation

When you enable report generation, the code generator produces a code generation report. To control generation and opening of a code generation report, use app settings, `codegen` options, or configuration object properties.

In the HDL Coder app:

- 1 Open the HDL Coder Workflow Advisor.
- 2 In the HDL Code Generation step options, on the **Coding Style** tab, under **Generated Code Comments**, select the **Generate report** check box.

At the command line, use `codegen` options:

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use configuration object properties:

- To generate a report, set `GenerateReport` to `true`.
- If you want `codegen` to open the report for you, set `LaunchReport` to `true`.

## Report Location

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

## Errors and Warnings

View code generation error, warning, and information messages on the **All Messages** tab. To highlight the source code for an error or warning, click the message. It is a best practice to address the first message because subsequent errors and warnings can be related to the first message.

## Files and Functions

The report lists MATLAB source functions and generated files. In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function in the list. Clicking a function opens the file that contains the function. To edit the selected file in the MATLAB Editor, click **Edit in MATLAB** or click a line number in the code pane.

### Specialized Functions or Classes

When a function is called with different types of inputs or a class uses different types for its properties, the code generator produces specializations. In the **MATLAB Source** pane, numbered functions (or classes) indicate specializations. For example:

```
fx fcn > 1
fx fcn > 2
```

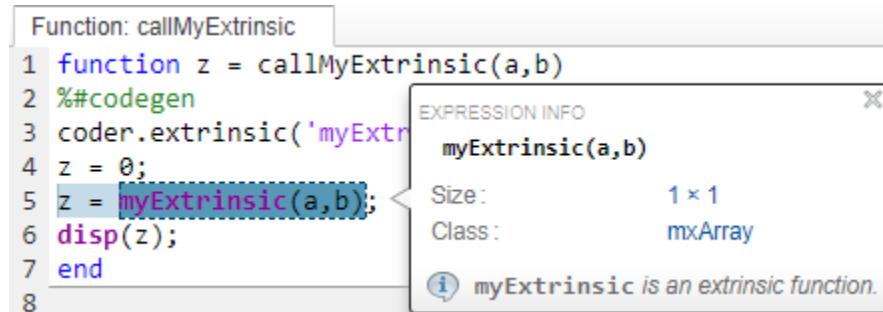
## MATLAB Source

To view a MATLAB function in the code pane, click the function in the **MATLAB Source** pane. To see information about the type of a variable or expression, pause over the variable or expression.

In the code pane, syntax highlighting of MATLAB source code helps you to identify MATLAB syntax elements. Syntax highlighting also helps you to identify certain code generation attributes such as whether a function is extrinsic or whether an argument is constant.

### Extrinsic Functions

In the MATLAB code, the report identifies an extrinsic function with purple text. The information window indicates that the function is extrinsic.



The screenshot shows a MATLAB code editor with the following code:

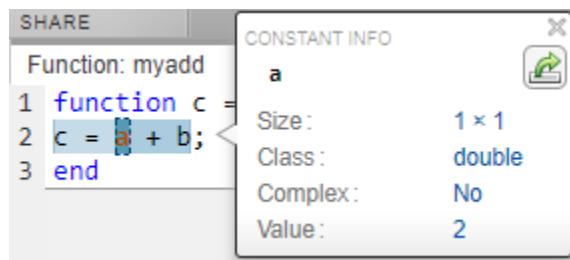
```
1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtr
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8
```

A tooltip window titled "EXPRESSION INFO" is open over the line `z = myExtrinsic(a,b);`. The tooltip displays the following information:

- myExtrinsic(a,b)**
- Size:** 1 × 1
- Class:** mxArray
- Info:** myExtrinsic is an extrinsic function.

## Constant Arguments

In the MATLAB code, orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The information window includes the constant value.



Knowing the value of the constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for different constant argument values.

To export the value to a variable in the workspace, click .

## MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types

This information helps you to debug errors, such as type mismatch errors, and to understand type propagation.

### Visual Indicators on the Variables Tab

This table describes symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	{ : }	Heterogeneous cell array (all elements have the same properties)
Name	{ n }	nth element of a heterogeneous cell array

Column in the Variables Table	Indicator	Description
Class	v > n	v is reused with a different class, size, and complexity. The number n identifies each unique reuse (a reuse with a unique set of properties). When you pause over a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity.
Class	complex prefix	Complex number
Class	 fi	Fixed-point type To see the fixed-point properties, click the badge.

## Additional Reports

The **Summary** tab can have links to these additional reports:

- Conformance report
- Resource report
- “HDL Coding Standard Report” on page 26-2

## Report Limitations

- The entry-point summary shows individual elements of `varargin` and `vargout`, but the variables table does not show them.
- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

## See Also

### More About

- “Basic HDL Code Generation and FPGA Synthesis from MATLAB”
- “Generate HDL Code from MATLAB Code Using the Command Line Interface”

# Generate Instantiable Code for Functions

## In this section...

["How to Generate Instantiable Code for Functions" on page 5-11](#)

["Generate Code Inline for Specific Functions" on page 5-11](#)

["Limitations for Instantiable Code Generation for Functions" on page 5-11](#)

You can use the **Generate instantiable code for functions** option to generate a VHDL entity or Verilog module for each function. The software generates code for each entity or module in a separate file.

## How to Generate Instantiable Code for Functions

To enable instantiable code generation for functions in the UI:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task.
- 2 In the **Advanced** tab, select **Generate instantiable code for functions**.

To enable instantiable code generation for functions programmatically, in your `coder.HdlConfig` object, set the `InstantiateFunctions` property to true. For example, to create a `coder.HdlConfig` object and enable instantiable code generation for functions:

```
hdlcfg = coder.config('hdl');
hdlcfg.InstantiateFunctions = true;
```

## Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

## Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or for loops.
- Any function is called with a nonconstant struct input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

## See Also

`coder.FixptConfig` | `coder.HdlConfig`

## Related Examples

- ["Generating Modular HDL Code for Functions"](#)

## Integrate Custom HDL Code Into MATLAB Design

`hdl.BlackBox` provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

### In this section...

- “Define the `hdl.BlackBox` System object” on page 5-12
- “Use System object In MATLAB Design Function” on page 5-13
- “Generate HDL Code” on page 5-14
- “Limitations for `hdl.BlackBox`” on page 5-16

## Define the `hdl.BlackBox` System object

- 1 Create a user-defined System object that inherits from `hdl.BlackBox`.
- 2 Configure the black box interface to match the port interface for your custom HDL code by setting `hdl.BlackBox` properties in the System object.
- 3 Define the `step` method such that its simulation behavior matches the custom HDL code.

Alternatively, the System object you define can inherit from both `hdl.BlackBox` and the `matlab.system.mixin.Nondirect` class, and you can define `output` and `update` methods to match the custom HDL code simulation behavior.

### Example Code

For example, the following code defines a System object, `CounterBbox`, that inherits from `hdl.BlackBox` and represents custom HDL code for a counter that increments until it reaches a threshold. The `CounterBbox` `reset` and `step` methods model the custom HDL code behavior.

```
classdef CounterBbox < hdl.BlackBox % derive from hdl.BlackBox class
    %Counter: Count up to a threshold.
    %
    % This is an example of a discrete-time System object with state
    % variables.
    %
    properties (Nontunable)
        Threshold = 1
    end

    properties (DiscreteState)
        % Define discrete-time states.
        Count
    end
```

```

methods
    function obj = CounterBbox(varargin)
        % Support name-value pair arguments
        setProperties(obj,nargin varargin{:});
        obj.NumInputs = 1; % define number of inputs
        obj.NumOutputs = 1; % define number of outputs
    end
end

methods (Access=protected)
    % Define simulation behavior.
    % For code generation, the coder uses your custom HDL code instead.
    function resetImpl(obj)
        % Specify initial values for DiscreteState properties
        obj.Count = 0;
    end

    function myout = stepImpl(obj, myin)
        % Implement algorithm. Calculate y as a function of
        % input u and state.
        if (myin > obj.Threshold)
            obj.Count = obj.Count + 1;
        end
        myout = obj.Count;
    end
end
end

```

## Use System object In MATLAB Design Function

After you define your System object, use it in the MATLAB design function by creating an instance and calling its `step` method.

To generate code, you also need to create a test bench function that exercises the top-level design function.

### Example Code

The following example code shows a top-level design function that creates an instance of the `CounterBbox` and calls its `step` method.

```

function [y1, y2] = topLevelDesign(u)

persistent mybboxObj myramObj
if isempty(mybboxObj)
    mybboxObj = CounterBbox; % instantiate the black box
    myramObj = hdl.RAM('RAMType', 'Dual port');
end

y1 = step(mybboxObj, u); % call the system object step method
[~, y2] = step(myramObj, uint8(10), uint8(0), true, uint8(20));

```

The following example code shows a test bench function for the `topLevelDesign` function.

```

clear topLevelDesign
y1 = zeros(1,200);

```

```
y2 = zeros(1,200);
for ii=1:200
    [y1(ii), y2(ii)] = topLevelDesign(ii);
end
plot([1:200], y2)
```

## Generate HDL Code

Generate HDL code using the design function and test bench code.

When you use the generated HDL code, include your custom HDL code with the generated HDL files.

### Example Code

In the following generated VHDL code for the CounterBbox example, you can see that the CounterBbox instance in the MATLAB code maps to an HDL component definition and instantiation, but HDL code is not generated for the step method.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY foo IS
    PORT( clk           : IN  std_logic;
          reset         : IN  std_logic;
          clk_enable    : IN  std_logic;
          u             : IN  std_logic_vector(7 DOWNTO 0);  -- uint8
          ce_out        : OUT std_logic;
          y1            : OUT real;   -- double
          y2            : OUT std_logic_vector(7 DOWNTO 0)  -- uint8
        );
END foo;

ARCHITECTURE rtl OF foo IS

    -- Component Declarations
COMPONENT CounterBbox
    PORT( clk           : IN  std_logic;
          clk_enable    : IN  std_logic;
          reset         : IN  std_logic;
          myin          : IN  std_logic_vector(7 DOWNTO 0);  -- uint8
          myout         : OUT real   -- double
        );
END COMPONENT;

COMPONENT DualPortRAM_Inst0
    PORT( clk           : IN  std_logic;
          enb          : IN  std_logic;
          wr_din       : IN  std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_addr      : IN  std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_en        : IN  std_logic;
          rd_addr      : IN  std_logic_vector(7 DOWNTO 0);  -- uint8
          wr_dout      : OUT std_logic_vector(7 DOWNTO 0);  -- uint8
          rd_dout      : OUT std_logic_vector(7 DOWNTO 0)  -- uint8
        );
END COMPONENT;
```

```

-- Component Configuration Statements
FOR ALL : CounterBbox
    USE ENTITY work.CounterBbox(rtl);

FOR ALL : DualPortRAM_Inst0
    USE ENTITY work.DualPortRAM_Inst0(rtl);

-- Signals
SIGNAL enb          : std_logic;
SIGNAL varargout_1   : real := 0.0; -- double
SIGNAL tmp           : unsigned(7 DOWNTO 0); -- uint8
SIGNAL tmp_1          : unsigned(7 DOWNTO 0); -- uint8
SIGNAL tmp_2          : std_logic;
SIGNAL tmp_3          : unsigned(7 DOWNTO 0); -- uint8
SIGNAL varargout_1_1  : std_logic_vector(7 DOWNTO 0); -- ufix8
SIGNAL varargout_2    : std_logic_vector(7 DOWNTO 0); -- ufix8

BEGIN
    u_CounterBbox : CounterBbox
        PORT MAP( clk => clk,
                   clk_enable => enb,
                   reset => reset,
                   myin => u, -- uint8
                   myout => varargout_1 -- double
                 );

    u_DualPortRAM_Inst0 : DualPortRAM_Inst0
        PORT MAP( clk => clk,
                   enb => enb,
                   wr_din => std_logic_vector(tmp), -- uint8
                   wr_addr => std_logic_vector(tmp_1), -- uint8
                   wr_en => tmp_2,
                   rd_addr => std_logic_vector(tmp_3), -- uint8
                   wr_dout => varargout_1_1, -- uint8
                   rd_dout => varargout_2 -- uint8
                 );

    enb <= clk_enable;
    y1 <= varargout_1;
    --y2 = u;
    tmp <= to_unsigned(2#00001010#, 8);
    tmp_1 <= to_unsigned(2#00000000#, 8);
    tmp_2 <= '1';
    tmp_3 <= to_unsigned(2#00010100#, 8);
    ce_out <= clk_enable;
    y2 <= varargout_2;

END rtl;

```

## Limitations for `hdl.BlackBox`

You cannot use `hdl.BlackBox` to assign values to a VHDL generic or Verilog parameter in your custom HDL code.

### See Also

`hdl.BlackBox`

### Related Examples

- “Generate Board-Independent IP Core from MATLAB Algorithm” on page 5-35

# Enable MATLAB Function Block Generation

## In this section...

- “Requirements for MATLAB Function Block Generation” on page 5-17
- “Enable MATLAB Function Block Generation” on page 5-17
- “Restrictions for MATLAB Function Block Generation” on page 5-17
- “Results of MATLAB Function Block Generation” on page 5-17

## Requirements for MATLAB Function Block Generation

During HDL code generation, your MATLAB algorithm must go through the floating-point to fixed-point conversion process, even if it is already a fixed-point algorithm.

## Enable MATLAB Function Block Generation

### Using the GUI

To enable MATLAB Function block generation using the HDL Workflow Advisor:

- 1 In the HDL Workflow Advisor, on the left, click **Code Generation**.
- 2 In the **Advanced** tab, select the **Generate MATLAB Function Black Box** option.

### Using the Command Line

To enable MATLAB Function block generation, at the command line, enter:

```
hdlcfg = coder.config('hdl');
hdlcfg.GenerateMLFcnBlock = true;
```

## Restrictions for MATLAB Function Block Generation

The top-level MATLAB design function cannot have input or output arguments with the **struct** data type.

## Results of MATLAB Function Block Generation

After you generate HDL code, an untitled model opens containing a MATLAB Function block.

You can use the MATLAB Function block as part of a larger model in Simulink for simulation and further HDL code generation.

To learn more about generating a MATLAB Function block from a MATLAB algorithm, see “System Design with HDL Code Generation from MATLAB and Simulink” on page 5-18.

# System Design with HDL Code Generation from MATLAB and Simulink

This example shows how to generate a MATLAB Function block from a MATLAB® design for system simulation, code generation, and FPGA programming in Simulink®.

## Introduction

HDL Coder can generate HDL code from both MATLAB® and Simulink®. The coder can also generate a Simulink® component, the MATLAB Function block, from your MATLAB code.

This capability enables you to:

- 1 Design an algorithm in MATLAB;
- 2 Generate a MATLAB Function block from your MATLAB design;
- 3 Use the MATLAB component in a Simulink model of the system;
- 4 Simulate and optimize the system model;
- 5 Generate HDL code; and
- 6 Program an FPGA with the entire system design.

In this example, you will generate a MATLAB Function block from MATLAB code that implements a FIR filter.

## MATLAB Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';

1 Design: mlhdlc_fir
2 Test Bench: mlhdlc_fir_tb
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

% Create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

To simulate the design with the test bench prior to code generation to make sure there are no runtime errors, enter the following command:

mlhdlc\_fir\_tb

## Create a New Project

To create a new HDL Coder project, enter the following command:

```
coder -hdlcoder -new fir_project
```

Next, add the file 'mlhdlc\_fir.m' to the project as the MATLAB Function and 'mlhdlc\_fir\_tb.m' as the MATLAB Test Bench.

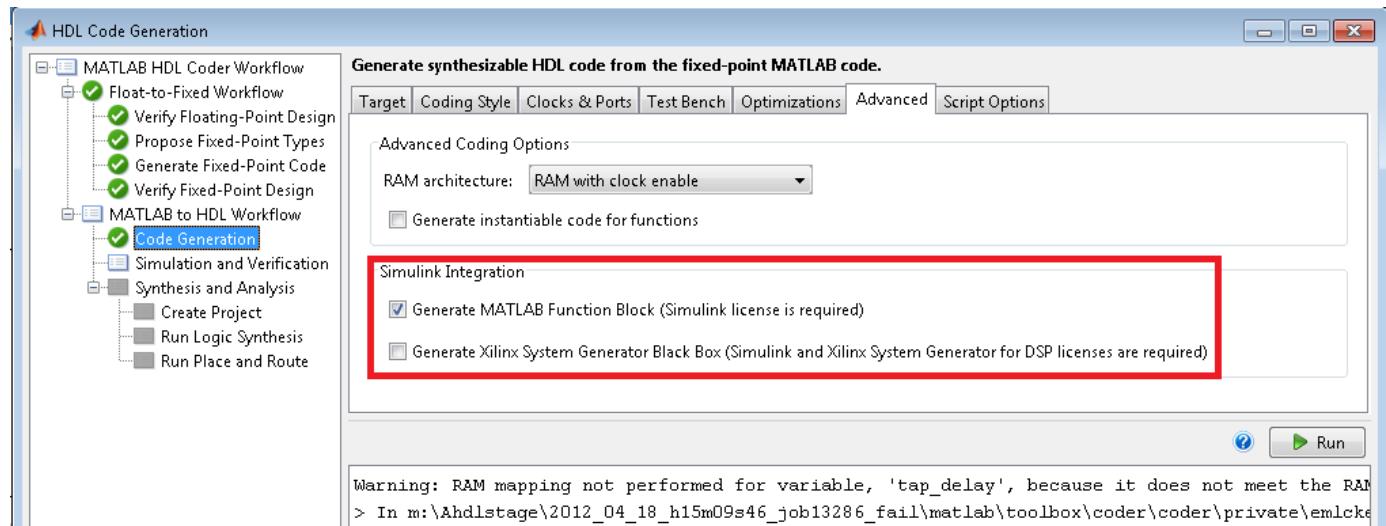
Click the Workflow Advisor button to launch the HDL Workflow Advisor.

## Enable the MATLAB Function Block Option

To generate a MATLAB Function block from a MATLAB HDL design, you must have a Simulink license. If the following command returns '1', Simulink is available:

```
license('test', 'Simulink')
```

In the HDL Workflow Advisor Advanced tab, enable the Generate MATLAB Function Block option.



## Run Floating-Point to Fixed-Point Conversion and Generate Code

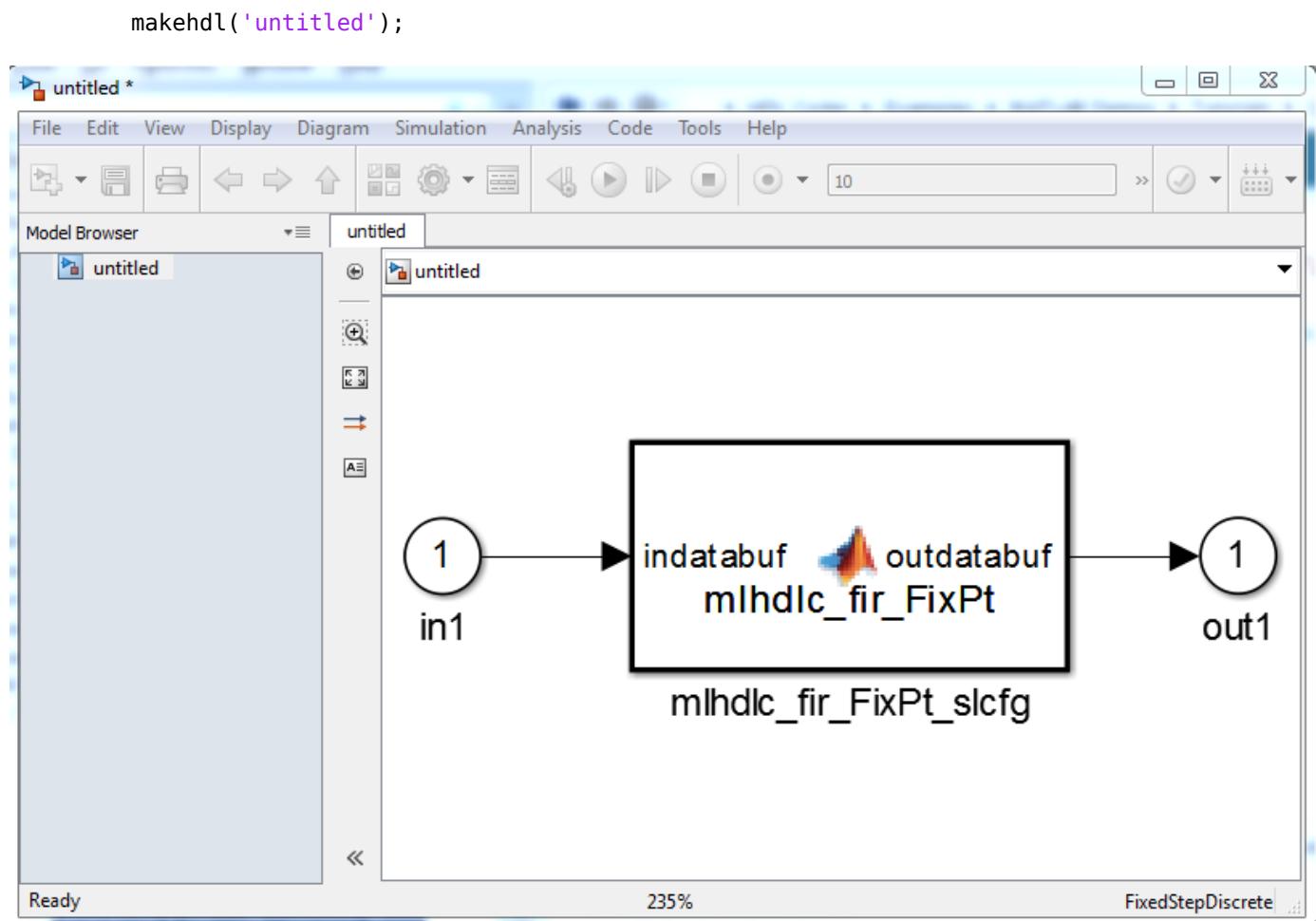
To generate a MATLAB Function block, you must also convert your design from floating-point to fixed-point.

Right-click the 'Code Generation' step and choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

## Examine the Generated MATLAB Function Block

An untitled model opens after HDL code generation. It has a MATLAB Function block containing the fixed-point MATLAB code from your MATLAB HDL design. HDL Coder automatically applies settings to the model and MATLAB Function block so that they can simulate in Simulink and generate HDL code.

To generate HDL code from the MATLAB Function block, enter the following command:



You can rename and save the new block to use in a larger Simulink design.

### Clean Up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Specify the Clock Enable Rate

## In this section...

["Why Specify the Clock Enable Rate?" on page 5-21](#)

["How to Specify the Clock Enable Rate" on page 5-21](#)

## Why Specify the Clock Enable Rate?

When HDL Coder performs area optimizations, it might upsample parts of your design (DUT), and thereby introduce an increase in your required DUT clock frequency.

If the coder upsamples your design, it generates a message indicating the ratio between the new clock frequency and your original clock frequency. For example, the following message indicates that your design's new required clock frequency is 4 times higher than the original frequency:

The design requires 4 times faster clock with respect to the base rate = 1

This frequency increase introduces a rate mismatch between your input clock enable and output clock enable, because the output clock enable runs at the slower original clock frequency.

With the **Drive clock enable at** option, you can choose whether to drive the input clock enable at the faster rate (**DUT base rate**) or at a rate that is less than or equal to the original clock enable rate (**Input data rate**).

## How to Specify the Clock Enable Rate

- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation**. Click the **Clocks & Ports** tab.
- 2 For the **Drive clock enable at** option, select **Input data rate** or **DUT base rate**.

Drive clock enable at Option	Clock Enable Behavior
<b>Input data rate</b> (default)	<p>Each assertion of the input clock enable produces an output clock enable assertion.</p> <p>You can assert the input clock enable at a maximum rate of once every N clocks. N = the upsampled clock rate / original clock rate.</p> <p>For example, if you see the message, "The design requires 4 times faster clock with respect to the base rate = 1", your maximum input clock enable rate is once every 4 clocks.</p>

Drive clock enable at Option	Clock Enable Behavior
<b>DUT base rate</b>	<p>Input clock enable rate does not match the output clock enable rate. You must assert the input clock enable with your input data N times to get 1 output clock enable assertion. N = the upsampled clock rate / original clock rate.</p> <p>For example, if you see the message, “The design requires 4 times faster clock with respect to the base rate = 1”, you must assert the input clock enable 4 times to get 1 output clock enable assertion.</p>

# Specify Test Bench Clock Enable Toggle Rate

## In this section...

["When to Specify Test Bench Clock Enable Toggle Rate" on page 5-23](#)

["How to Specify Test Bench Clock Enable Toggle Rate" on page 5-23](#)

## When to Specify Test Bench Clock Enable Toggle Rate

When you want the test bench to drive your input data at a slower rate than the maximum input clock enable rate, specify the test bench clock enable toggle rate.

This specification can help you to achieve better test coverage, and to simulate the real world input data rate.

---

**Note** The maximum input clock enable rate is once every N clock cycles. N = the upsampled clock rate / original clock rate. Refer to the clock enable behavior for **Input data rate**, in "Specify the Clock Enable Rate" on page 5-21.

---

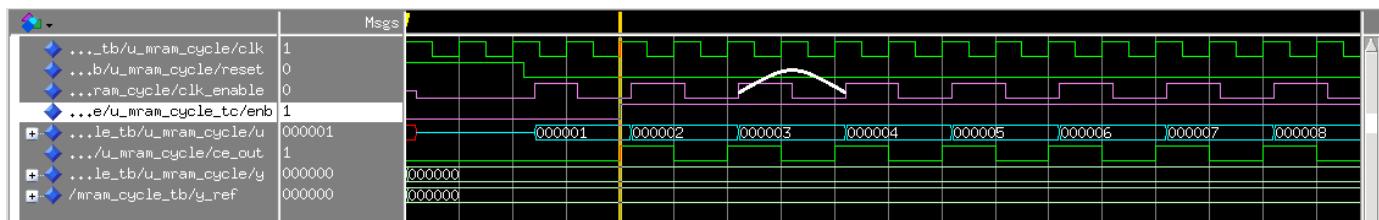
## How to Specify Test Bench Clock Enable Toggle Rate

To set your test bench clock enable toggle rate:

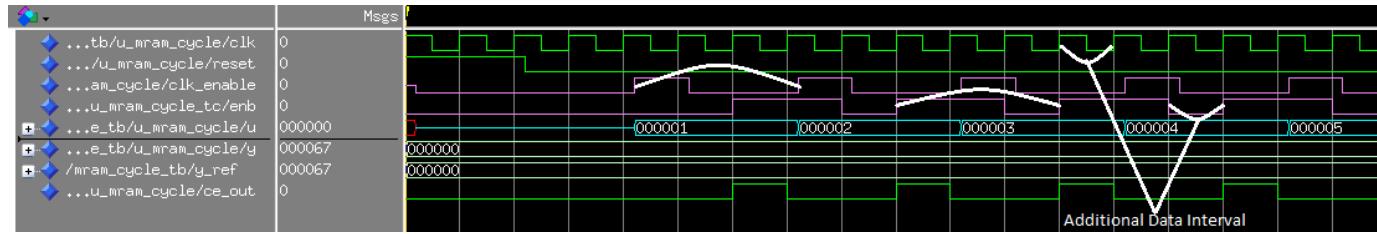
- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation**.
- 2 In the **Clocks & Ports** tab, for the **Drive clock enable at** option, select **Input data rate**.
- 3 In the **Test Bench** tab, for **Input data interval**, enter 0 or an integer greater than the maximum input clock enable interval.

Input data interval, I	Test Bench Clock Enable Behavior
I = 0 (default)	Asserts at the maximum input clock enable rate, or once every N cycles. N = the upsampled clock rate / original clock rate.
I < N	Not valid; generates an error.
I = N	Same as I = 0.
I > N	Asserts every I clock cycles.

For example, this timing diagram shows clock enable behavior with **Input data interval** = 0. Here, the maximum input clock enable rate is once every 2 cycles.



The following timing diagram shows the same test bench and DUT with **Input data interval** = 3.



# Generate an HDL Coding Standard Report from MATLAB

## In this section...

["Using the HDL Workflow Advisor" on page 5-25](#)

["Using the Command Line" on page 5-27](#)

You can generate an HDL coding standard report that shows how well your generated code follows industry standards. You can optionally customize the coding standard report and the coding standard rules.

## Using the HDL Workflow Advisor

To generate an HDL coding standard report using the HDL Workflow Advisor:

- 1 In the **HDL Code Generation** task, select the **Coding Standards** tab.
- 2 For **HDL coding standard**, select **Industry**.

Target Coding Style Coding Standards Clocks & Ports Optimizations Advanced Script Options

Choose coding standard  
HDL coding standard: Industry ▾

Report options  
 Do not show passing rules in coding standard report

Basic coding rules

Check for duplicate names

Check for HDL keywords in design names

Check module, instance, entity name length

Minimum:  2

Maximum:  32

Check signal, port, parameter name length

Minimum:  2

Maximum:  40

RTL description rules

Check for clock enable signals

Check for reset signals

Check for asynchronous reset signals

Minimize use of variables

Check for initial statements that set RAM initial values

Check number of conditional regions

Length:  1

Check if-else statement chain length

Length:  7

Check if-else statement nesting depth

Depth:  3

Check multiplier width

Maximum:  16

RTL design rules

Check for non-integer constants

Check line wrap length

- 3 Optionally, using the other options in the **Coding Standards** tab, customize the coding standard rules.
- 4 Click **Run** to generate code.

After you generate code, the message window shows a link to the HTML compliance report.

## Using the Command Line

To generate an HDL coding standard report using the command line interface, set the `HDLCodingStandard` property to `Industry` in the `coder.HdLCfg` object.

For example, to generate HDL code and an HDL coding standard report for a design, `mlhdlc_sfir`, with a testbench, `mlhdlc_sfir_tb`, enter the following commands:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
codegen -config hdlcfg mlhdlc_sfir

### Generating Resource Utilization Report resource_report.html
### Generating default Industry script file mlhdlc_sfir_mlhdlc_sfir_default.prj
### Industry Compliance report with 0 errors, 8 warnings, 4 messages.
### Generating Industry Compliance Report mlhdlc_sfir_Industry_report.html
```

To open the report, click the report link.

You can customize the coding standard report and coding standard rule checks by specifying an HDL coding standard customization object. For example, suppose you have a design, `mlhdlc_sfir`, and testbench, `mlhdlc_sfir_tb`. You can create an HDL coding standard customization object, `cso`, set the maximum if-else statement chain length to 5 by using the `IfElseChain` property, and generate code:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_sfir_tb';
hdlcfg.HDLCodingStandard='Industry';
cso = hdlcoder.CodingStandard('Industry');
cso.IfElseChain.length = 5;
hdlcfg.HDLCodingStandardCustomizations = cso;
codegen -config hdlcfg mlhdlc_sfir
```

## See Also

### Properties

HDL Coding Standard Customization

## More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-9
- “RTL Description Techniques” on page 26-18
- “RTL Design Methodology Guidelines” on page 26-41

## Generate an HDL Lint Tool Script

You can generate a lint tool script to use with a third-party lint tool to check your generated HDL code.

HDL Coder can generate Tcl scripts for the following lint tools:

- Ascent Lint
- HDL Designer
- Leda
- SpyGlass
- Custom

If you specify one of the supported third-party lint tools, you can either generate a default tool-specific script, or customize the script by specifying the initialization, command, and termination names as a character vector. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination names.

HDL Coder writes the initialization, command, and termination names to a Tcl script that you can use to run the third-party tool.

## How To Generate an HDL Lint Tool Script

### Using the HDL Workflow Advisor

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task.
- 2 In the **Script Options** tab, select **Lint**.
- 3 For **Choose lint tool**, select **Ascent Lint**, **HDL Designer**, **Leda**, **SpyGlass**, or **Custom**.
- 4 Optionally, enter text to customize the **Lint script initialization**, **Lint script command**, and **Lint script termination** fields. For a custom tool, you must specify these fields.

After you generate code, the command window shows a link to the lint tool script.

### Using the Command Line

To generate an HDL lint tool script from the command line, set the **HDLLintTool** property to **AscentLint**, **HDLDesigner**, **Leda**, **SpyGlass** or **Custom** in your **coder.HdlConfig** object.

To disable HDL lint tool script generation, set the **HDLLintTool** property to **None**.

For example, to generate a default SpyGlass lint script using a **coder.HdlConfig** object, *hdlcfg*, enter:

```
hdlcfg.HDLLintTool = 'SpyGlass';
```

After you generate code, the command window shows a link to the lint tool script.

To generate an HDL lint tool script with custom initialization, command, and termination strings, use the **HDLLintTool**, **HDLLintInit**, **HDLLintCmd**, and **HDLLintTerm** properties.

For example, you can use the following command to generate a custom Leda lint script for a DUT subsystem, `sfir_fixed\symmetric_fir`, with custom initialization, termination, and command strings:

```
hdlcfg.HDLLintTool = 'Leda';
hdlcfg.HDLLintInit = 'myInitialization';
hdlcfg.HDLLintCmd = 'myCommand %s';
hdlcfg.HDLLintTerm = 'myTermination';
```

After you generate code, the command window shows a link to the lint tool script.

### Custom Lint Tool Command Specification

If you want to generate a lint tool script for a custom lint tool, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script.

For **Lint script command** or `HDLLintCmd`, specify the lint command in the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

For example, to set the `HDLLintCmd` for a `coder.HdlConfig` object, `hdlcfg`, where the lint command is `custom_lint_tool_command -option1 -option2`, enter:

```
hdlcfg.HDLLintCmd = 'custom_lint_tool_command -option1 -option2 %s';
```

## Generate HDL code from MATLAB functions using automated lookup table generation

This example shows HDL code generation from a floating-point MATLAB® design that is not ready for code generation in two steps. First we use float2fixed conversion process to generate a lookup table based MATLAB function replacements. Next this new MATLAB replacement function is used to generate the HDL code. However these two steps are opaque to the user due to the way float2fixed and MATLAB HDL Coder are invoked.

### Introduction

The MATLAB code used in the example is an implementation of a variable exponent function.

### MATLAB Design

```
design_name = 'mlhdlc_replacement_exp';
testbench_name = 'mlhdlc_replacement_exp_tb';
```

Lets look at the gamma correction Design

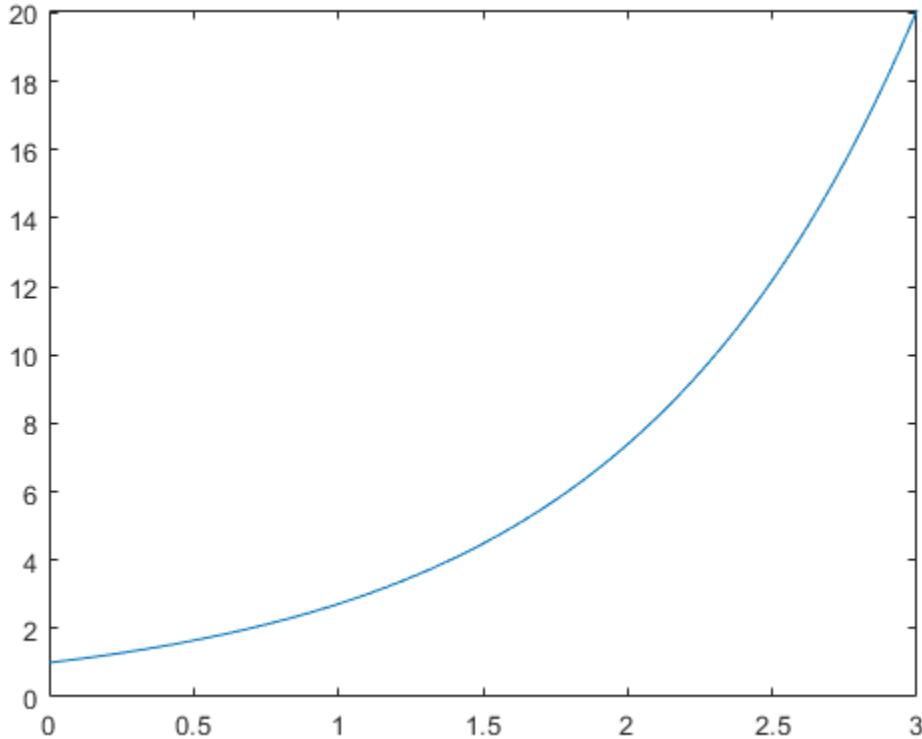
```
dbtype(design_name)

1      function y = mlhdlc_replacement_exp(u)
2      %
3
4      % Copyright 2014-2015 The MathWorks, Inc.
5
6      y = exp(u);
7
8      end
```

### Simulate the Design

It is always a good practice to simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_replacement_exp_tb
```



- 1 MATLAB Design: mlhdlc\_replacement\_exp
- 2 MATLAB testbench: mlhdlc\_replacement\_exp\_tb

Open the design function `mlhdlc_european_call` by clicking on the above link to notice the use of unsupported fixed-point functions like '`log`', and '`exp`'.

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = fullfile(tempdir(), 'mlhdlc_replacement_exp');

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Notes of design and testbench

The design is in the file '`mlhdlc_replacement_exp.m`'. The MATLAB Test Bench is in the file '`mlhdlc_replacement_exp_tb.m`' which can be run separately. We have illustrated a replacement for `exp` function as the purpose of this demo. You may alter the testbench for your desired responses.

### Generate HDL Code using implicit fixed-point conversion

Your design is in the file 'mlhdlc\_replacement\_exp.m' where the exponent function is calculated. The MATLAB Test Bench is in the file 'mlhdlc\_replacement\_exp\_tb.m' which can be run separately.

You will note that currently we do not have out-of-the-box fixed-point support for 'exp' functions, at this moment, in the design; this is where we use the "coder.approximation" object to enable a dynamic lookup-table replacement for these 'unsupported functions'. Run the following code as 'runme.m' file to execute the codegeneration steps.

```
clear design_name testbench_name fxpCfg hdlcfg interp_degree
hsetupdatoolsenv();

design_name = 'mlhdlc_replacement_exp';
testbench_name = 'mlhdlc_replacement_exp_tb';

interp_degree = 0;

%% fixed point converter config
fxpCfg = coder.config('fixpt');
fxpCfg.TestBenchName = 'mlhdlc_replacement_exp_tb';
fxpCfg.TestNumerics = true;

% specify this - for optimized HDL
fxpCfg.DefaultWordLength = 10;

%% exp - replacement config
mathFcnGenCfg = coder.approximation('exp');
% generally use to increase accuracy; specify this as power of 2 for optimized HDL
mathFcnGenCfg.NumberOfPoints = 1024;
mathFcnGenCfg.InterpolationDegree = interp_degree; % can be 0,1,2, or 3
fxpCfg.addApproximation(mathFcnGenCfg);

%% HDL config object
hdlcfg = coder.config('hdl');

hdlcfg.TargetLanguage = 'Verilog';

hdlcfg.DesignFunctionName = design_name;
hdlcfg.TestBenchName = testbench_name;
hdlcfg.GenerateHDLTestBench=true;

hdlcfg.SimulateGeneratedCode=true;

%If you choose VHDL set the ModelSim compile options as well
% hdlcfg.TargetLanguage = 'Verilog';
% hdlcfg.HDLCompileVHDLCmd = 'vcom %s %s -noindexcheck \n';

hdlcfg.ConstantMultiplierOptimization = 'auto'; %optimize out any multipliers from interpolation
hdlcfg.PipelineVariables = 'y u idx_bot x x_idx';%

hdlcfg.InputPipeline = 2;
hdlcfg.OutputPipeline = 2;
hdlcfg.RegisterInputs = true;
hdlcfg.RegisterOutputs = true;

hdlcfg.SynthesizeGeneratedCode = true;
hdlcfg.SynthesisTool = 'Xilinx ISE';
hdlcfg.SynthesisToolChipFamily = 'Virtex7';
hdlcfg.SynthesisToolDeviceName = 'xc7vh580t';
```

```

hdlcfg.SynthesisToolPackageName = 'hcg1155';
hdlcfg.SynthesisToolSpeedValue = '-2G';

%codegen('-config',hdlcfg)

codegen('-float2fixed',fxpCfg,'-config',hdlcfg,'mlhdlc_replacement_exp')

%If you only want to do fixed point conversion and stop/examine the
%intermediate results you can use,

%only F2F conversion
codegen('-float2fixed',fxpCfg,'mlhdlc_replacement_exp')

```

Once you run the 'runme' script, you will see the following output from fixpt converter and HDL Coder. But before we jump ahead, let us see a few notable topics in the config objects to get the fixpt conversion, HDL generation to yield high performance hardware.

**Generating high-clockrate circuits** Using the number of point for replacement functions as power of 2 is recommended for HDL targets. We also supply the ConstantMultiplierOptimization = 'AUTO' to use best of FCSD or CSD approaches in generated HDL code. Note our use of pipelined variables in HDL Code generation to minimize the clock delays, and improve circuit frequency.

### Output and iterative improvements

First the fixed-point conversion completes with appropriate function replacements as,

```

===== Step1: Analyze floating-point code =====

Input types not specified, inferring types by simulating the test bench.

===== Step1a: Verify Floating Point =====

### Analyzing the design 'mlhdlc_replacement_exp'
### Analyzing the test bench(es) 'mlhdlc_replacement_exp_tb'
### Begin Floating Point Simulation (Instrumented)
### Floating Point Simulation Completed in 1.8946 sec(s)
### Elapsed Time: 2.8361 sec(s)

===== Step2: Propose Types based on Range Information =====

===== Step3: Generate Fixed Point Code =====

### Generating Fixed Point MATLAB Code <a href="matlab:edit('codegen/mlhdlc_replacement_exp/fixed_point')">
### Generating Fixed Point MATLAB Design Wrapper <a href="matlab:edit('codegen/mlhdlc_replacement_exp_wrapper')">
### Generating Mex file for ' mlhdlc_replacement_exp_wrapper_fixpt '
Code generation successful: To view the report, open('codegen/mlhdlc_replacement_exp/fixed_point_report.html')
### Generating Type Proposal Report for 'mlhdlc_replacement_exp' <a href="matlab:web('codegen/mlhdlc_replacement_exp_type_proposal_report.html')">

===== Step4: Verify Fixed Point Code =====

### Begin Fixed Point Simulation : mlhdlc_replacement_exp_tb
### Fixed Point Simulation Completed in 1.9497 sec(s)
### Generating Type Proposal Report for 'mlhdlc_replacement_exp_fixpt' <a href="matlab:web('codegen/mlhdlc_replacement_exp_fixpt_type_proposal_report.html')">
### Elapsed Time: 2.6488 sec(s)

As this is a small design with only one replacement functions you can
try different number of points in approximation function generation.
Re-examine the generated HDL code and compare it with the previous step.

### Begin VHDL Code Generation
### Generating HDL Conformance Report <a href="matlab:web('codegen/mlhdlc_replacement_exp_hdls_report.html')">
```

```
### HDL Conformance check complete with 0 errors, 2 warnings, and 0 messages.
### Working on mlhdlc_replacement_exp_fixpt as <a href="matlab:edit('codegen/mlhdlc_replacement...
### Generating package file <a href="matlab:edit('codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc...
### The DUT requires an initial pipeline setup latency. Each output port experiences these addi...
### Output port 0: 12 cycles.
### Output port 1: 12 cycles.

### Generating Resource Utilization Report '<a href="matlab:web('codegen/mlhdlc_replacement_exp...

### Begin TestBench generation.
### Accounting for output port latency: 12 cycles.'
### Collecting data...
### Begin HDL test bench file generation with logged samples
### Generating test bench: codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp_fixpt...
### Creating stimulus vectors ...

### Simulating the design 'mlhdlc_replacement_exp_fixpt' using 'ModelSim'.
### Generating Compilation Report codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp...
### Generating Simulation Report codegen/mlhdlc_replacement_exp/hdlsrc/mlhdlc_replacement_exp...
### Simulation successful.

### Creating Synthesis Project for 'mlhdlc_replacement_exp_fixpt'.
### Synthesis project creation successful.

### Synthesizing the design 'mlhdlc_replacement_exp_fixpt'.
### Generating synthesis report codegen/mlhdlc_replacement_exp/hdlsrc/ise_prj/mlhdlc_replacement...
### Synthesis successful.
```

**Examine the Synthesis Results** Run the logic synthesis step with the following default options if you have ISE installed on your machine. In the synthesis report, note the clock frequency reported by the synthesis tool without any optimization options enabled. Typically timing performance of this design using Xilinx ISE synthesis tool for the **Virtex7** chip family, device **xc7vh580t**, package **hcg1155**, speed grade -2G, with a high clock speed in order of 300 MHz.

### Clean up the Generated Files

You can run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_replacement_exp'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Generate Board-Independent IP Core from MATLAB Algorithm

## In this section...

["Requirements and Limitations for IP Core Generation" on page 5-35](#)

["Generate Board-Independent IP Core" on page 5-35](#)

When you open the HDL Workflow Advisor and run the **IP Core Generation** workflow for your Simulink model, you can specify a generic Xilinx® platform or a generic Intel® platform. The workflow then generates a generic IP core that you can integrate into any target platform of your choice. For IP core integration, define and register a custom reference design for your target board.

## Requirements and Limitations for IP Core Generation

You cannot generate an HDL IP core without any AXI4 slave interface. At least one DUT port must map to an AXI4 or AXI4-Lite interface. To generate an HDL IP core without any AXI4 slave interfaces, use the Simulink IP core generation workflow. For more information, see “[Generate Board-Independent HDL IP Core from Simulink Model](#)” on page 40-19.

In the same IP core, you cannot map to both an AXI4 interface and AXI4-Lite interface.

### AXI4-Lite Interface Restrictions

- The inputs and outputs must have a bit width less than or equal to 32 bits.
- The input and outputs must be scalar.

### AXI4-Stream Video Interface Restrictions

- Ports must have a 32-bit width.
- Ports must be scalar.
- You can have a maximum of one input video port and one output video port.
- The AXI4-Stream Video interface is not supported in **Coprocessing - blocking Processor/FPGA synchronization** must be set to **Free running** mode. Coprocessing – blocking mode is not supported.

## Generate Board-Independent IP Core

To generate a board-independent IP core to use in an embedded system integration environment, such as Intel Qsys, Xilinx EDK, or Xilinx IP Integrator:

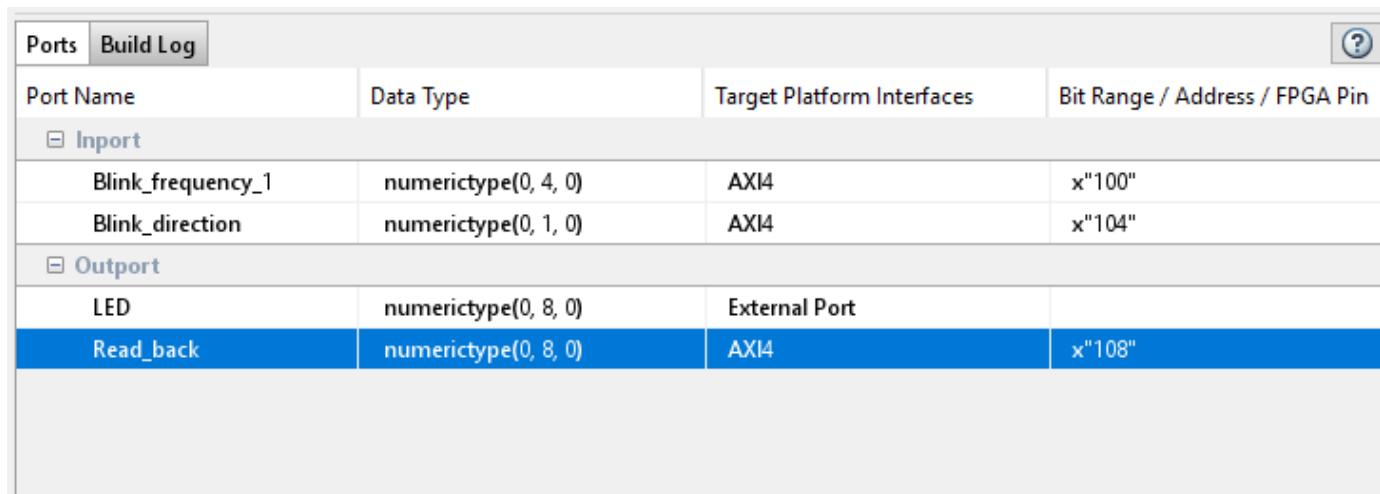
- 1 Create an HDL Coder project containing your MATLAB design and test bench, or open an existing project.
- 2 In the HDL Workflow Advisor, define input types and perform fixed-point conversion.

To learn how to convert your design to fixed-point, see “[Basic HDL Code Generation and FPGA Synthesis from MATLAB](#)”.

- 3 In the HDL Workflow Advisor, in the **Select Code Generation Target** task:
  - **Workflow:** Select **IP Core Generation**.
  - **Platform:** Select **Generic Xilinx Platform** or **Generic Altera Platform**.

Depending on your selection, the code generator automatically sets the **Synthesis tool**. For example, if you select **Generic Xilinx Platform**, **Synthesis tool** automatically changes to **Xilinx Vivado**.

- **Additional source files:** If you are using an `hdl.BlackBox` System object to include existing Verilog or VHDL code, enter the file names. Enter each file name manually, separated with a semicolon (;), or by using the ... button. The source file language must match your target language.
- 4 In the **Set Target Interface** step, for each port, select an option from the **Target Platform Interfaces** drop-down list.



The screenshot shows the 'Ports' tab of the HDL Workflow Advisor. The table has four columns: Port Name, Data Type, Target Platform Interfaces, and Bit Range / Address / FPGA Pin. There are two sections: 'Import' and 'Outport'. Under 'Import', there are two rows: 'Blink\_frequency\_1' (numerictype(0, 4, 0)) mapped to AXI4 at bit range x"100", and 'Blink\_direction' (numerictype(0, 1, 0)) mapped to AXI4 at bit range x"104". Under 'Outport', there are two rows: 'LED' (numerictype(0, 8, 0)) mapped to External Port, and 'Read\_back' (numerictype(0, 8, 0)) mapped to AXI4 at bit range x"108". A blue bar highlights the 'Read\_back' row.

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
<b>Import</b>			
Blink_frequency_1	numerictype(0, 4, 0)	AXI4	x"100"
Blink_direction	numerictype(0, 1, 0)	AXI4	x"104"
<b>Outport</b>			
LED	numerictype(0, 8, 0)	External Port	
Read_back	numerictype(0, 8, 0)	AXI4	x"108"

- 5 In the **HDL Code Generation** step, optionally specify code generation options, then click **Run**.

In the HDL Workflow Advisor message pane, click the IP core report link to view detailed documentation for your generated IP core.

## See Also

### Classes

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

## Related Examples

- “Using IP Core Generation Workflow from MATLAB: LED Blinking” on page 40-143

## More About

- “Custom IP Core Generation” on page 40-10
- “Board and Reference Design Registration System” on page 41-39

# Minimize Clock Enables

## In this section...

- “Using the GUI” on page 5-37
- “Using the Command Line” on page 5-37
- “Limitations” on page 5-38

By default, HDL Coder generates code in a style that is intended to map to registers with clock enables, and the DUT has a top-level clock enable port.

If you do not want to generate registers with clock enables, you can minimize the clock enable logic. For example, if your target hardware contains registers without clock enables, you can save hardware resources by minimizing the clock enable logic.

The following VHDL code shows the default style of generated code, which uses clock enables. The enb signal is the clock enable:

```
Unit_Delay_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay_out1 <= to_signed(0, 32);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay_out1 <= In1_signed;
    END IF;
  END IF;
END PROCESS Unit_Delay_process;
```

The following VHDL code shows the style of code you generate if you minimize clock enables:

```
Unit_Delay_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay_out1 <= to_signed(0, 32);
  ELSIF clk'EVENT AND clk = '1' THEN
    Unit_Delay_out1 <= In1_signed;
  END IF;
END PROCESS Unit_Delay_process;
```

## Using the GUI

To minimize clock enables, in the HDL Workflow Advisor, on the **HDL Code Generation > Set Code Generation Options > Set Optimization Options > General** tab, select **Minimize clock enables**.

## Using the Command Line

To minimize clock enables, in the `coder.HdlConfig` configuration object, set the `MinimizeClockEnables` property to `true`. For example:

```
hdlCfg = coder.config('hdl')
hdlCfg.MinimizeClockEnables = true;
```

## Limitations

If you specify area optimizations that the coder implements by increasing the clock rate in certain regions of the design, you cannot minimize clock enables. The following optimizations prevent clock enable minimization:

- Resource sharing
- RAM mapping
- Loop streaming

# Verification

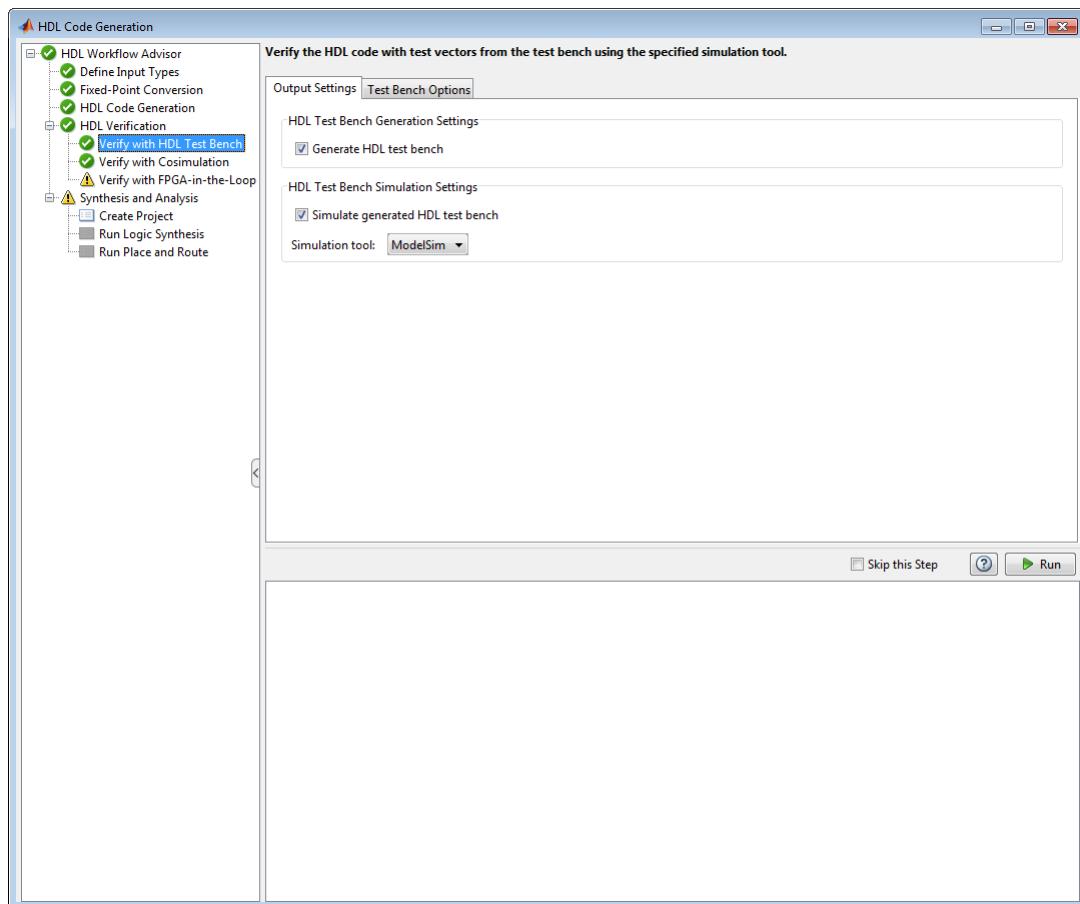
---

- “Verify Code with HDL Test Bench” on page 6-2
- “Test Bench Generation” on page 6-5

## Verify Code with HDL Test Bench

Simulate the generated HDL design under test (DUT) with test vectors from the test bench using the specified simulation tool.

- 1 Start the MATLAB to HDL Workflow Advisor.



- 2 At step **HDL Verification**, click **Verify with HDL Test Bench**.

- 3 Select **Generate HDL test bench**.

This option enables HDL Coder to generate HDL test bench code from your MATLAB test script.

- 4 Optionally, select **Simulate generated HDL test bench**. This option enables MATLAB to simulate the HDL test bench with the HDL DUT.

If you select this option, you must also select the **Simulation tool**.

- 5 For **Test Bench Options**, select and set the optional parameters according to the descriptions in the following table.

HDL Test Bench Parameter	Description
<b>Test bench name postfix</b>	Specify the postfix for the test bench name.

HDL Test Bench Parameter	Description
<b>Force clock</b>	Enable for test bench to force clock input signals.
<b>Clock high time (ns)</b>	Specify the number of nanoseconds the clock is high.
<b>Clock low time (ns)</b>	Specify the number of nanoseconds the clock is low.
<b>Hold time (ns)</b>	Specify the hold time for input signals and forced reset signals.
<b>Force clock enable</b>	Enable to force clock enable.
<b>Clock enable delay (in clock cycles)</b>	Specify time (in clock cycles) between deassertion of reset and assertion of clock enable.
<b>Force reset</b>	Enable for test bench to force reset input signals.
<b>Reset length (in clock cycles)</b>	Specify time (in clock cycles) between assertion and deassertion of reset.
<b>Hold input data between samples</b>	Enable to hold substrate signals between clock samples.
<b>Input data interval</b>	Specifies the number of clock cycles between assertions of clock enable. For more information, see “Specify Test Bench Clock Enable Toggle Rate” on page 5-23.
<b>Initialize test bench inputs</b>	Enable to initialize values on inputs to test bench before test bench drives data to DUT.
<b>Multi file test bench</b>	Enable to divide generated test bench into helper functions, data, and HDL test bench code.
<b>Test bench data file name postfix</b>	Specify the character vector to append to name of test bench data file when generating multi-file test bench.
<b>Test bench reference postfix</b>	Specify the character vector to append to names of reference signals in test bench code.
<b>Ignore data checking (number of samples)</b>	Specify the number of samples at the beginning of simulation during which output data checking is suppressed.
<b>Simulation iteration limit</b>	Specify the maximum number of test samples to use during simulation of generated HDL code.

- 6 Optionally, select **Skip this step** if you don't want to use the HDL test bench to verify the HDL DUT.
- 7 Click **Run**.

If the test bench and simulation is successful, you should see messages similar to these in the message pane:

```
### Begin TestBench generation.  
### Collecting data...  
### Begin HDL test bench file generation with logged samples  
### Generating test bench: mlhdlc_sfir_fixpt_tb.vhd  
### Creating stimulus vectors...  
### Simulating the design 'mlhdlc_sfir_fixpt' using 'ModelSim'.  
### Generating Compilation Report mlhdlc_sfir_fixpt_vsim_log_compile.txt  
### Generating Simulation Report mlhdlc_sfir_fixpt_vsim_log_sim.txt  
### Simulation successful.  
### Elapsed Time: 113.0315 sec(s)
```

If there are errors, those messages appear in the message pane. Fix errors and click **Run**.

# Test Bench Generation

## In this section...

- “How Test Bench Generation Works” on page 6-5
- “Test Bench Data Files” on page 6-5
- “Test Bench Data Type Limitations” on page 6-5
- “Use Constants Instead of File I/O” on page 6-5

## How Test Bench Generation Works

HDL Coder writes the DUT stimulus and reference data from your MATLAB or Simulink simulation to data files (.dat).

During HDL simulation, the HDL test bench reads the saved stimulus from the .dat files. The test bench compares the actual DUT output with the expected output, which is also saved in .dat files. After you generate code, the message window displays links to the test bench data files.

Reference data is delayed by one clock cycle in the waveform viewer compared to default test bench generation due to the delay in reading data from files.

## Test Bench Data Files

The coder saves stimulus and reference data for each DUT input and output in a separate test bench data file (.dat), with the following exceptions:

- Two files are generated for the real and imaginary parts of complex data.
- Constant DUT input data is written to the test bench as constants.

Vector input or output data is saved as a single file.

## Test Bench Data Type Limitations

If you have double, single, or enumeration data types at the DUT inputs and outputs, the simulation data is generated as constants in the test bench code, instead of writing the simulation data to files.

## Use Constants Instead of File I/O

You can generate test bench stimulus and reference data as constants in the test bench code instead of using file I/O. However, simulating a long running test bench that uses constants requires more memory than a test bench that uses file I/O.

Test bench generation automatically generates data as constants if your DUT inputs or outputs use data types that are not supported for file I/O. For details, see “Test Bench Data Type Limitations” on page 6-5.

To generate a test bench that uses constants instead of file I/O:

- 1 In the HDL Workflow Advisor, select the **HDL Verification > Verify with HDL Test Bench** task.
- 2 In the **Test bench Options** tab, disable the **Use file I/O for test bench** option.



# Deployment

---

## Generate Synthesis Scripts

You can generate customized synthesis scripts for the following tools:

- Xilinx Vivado®
- Xilinx ISE
- Microsemi Libero
- Mentor Graphics® Precision
- Altera® Quartus II
- Synopsys® Synplify Pro®

You can also generate a synthesis script for a custom tool by specifying the fields manually.

To generate a synthesis script:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task.
- 2 In the **Script Options** tab, select **Synthesis**.
- 3 For **Choose synthesis tool**, select a tool option.
- 4 If you want to customize your script, use the **Synthesis file postfix**, **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** text fields to do so.

After you generate code, your synthesis Tcl script (`.tcl`) is in the same folder as your generated HDL code.

# Optimization

---

- “RAM Mapping for MATLAB Code” on page 8-2
- “Map Matrices to Block RAMs to Reduce Area” on page 8-3
- “Map Persistent Arrays and `dsp.Delay` to RAM” on page 8-8
- “RAM Mapping Comparison for MATLAB Code” on page 8-11
- “Pipelining MATLAB Code” on page 8-12
- “Pipeline MATLAB Expressions” on page 8-13
- “Distributed Pipelining” on page 8-15
- “Distributed Pipelining for Clock Speed Optimization” on page 8-16
- “Optimize MATLAB Loops” on page 8-20
- “Constant Multiplier Optimization” on page 8-22
- “Resource Sharing of Multipliers to Reduce Area” on page 8-24
- “Loop Streaming to Reduce Area” on page 8-31
- “Constant Multiplier Optimization to Reduce Area” on page 8-36

## RAM Mapping for MATLAB Code

RAM mapping is an area optimization that maps storage and delay elements in your MATLAB code to RAM. Without this optimization, storage and delay elements are mapped to registers. RAM mapping can therefore reduce the area of your design in the target hardware.

You can map the following MATLAB code elements to RAM:

- persistent array variable
- `dsp.Delay` System object
- `hdl.RAM` System object

# Map Matrices to Block RAMs to Reduce Area

This example shows how to use the RAM mapping optimization in HDL Coder™ to map persistent matrix variables to block RAMs in hardware.

## Introduction

One of the attractive features of writing MATLAB code is the ease of creating, accessing, modifying and manipulating matrices in MATLAB.

When processing such MATLAB code, HDL Coder maps these matrices to wires or registers in HDL. For example, local temporary matrix variables are mapped to wires, whereas persistent matrix variables are mapped to registers.

The latter tends to be an inefficient mapping when the matrix size is large, since the number of register resources available is limited. It also complicates synthesis, placement and routing.

Modern FPGAs feature block RAMs that are designed to have large matrices. HDL Coder takes advantage of this feature and automatically maps matrices to block RAMs to improve area efficiency. For certain designs, mapping these persistent matrices to RAMs is mandatory if the design is to be realized. State-of-the-art synthesis tools may not be able to synthesize designs when large matrices are mapped to registers, whereas the problem size is more manageable when the same matrices are mapped to RAMs.

## MATLAB Design

```
design_name = 'mlhdlc_sobel';
testbench_name = 'mlhdlc_sobel_tb';
```

- MATLAB Design: mlhdlc\_sobel
- MATLAB Testbench: mlhdlc\_sobel\_tb
- Input Image: stop\_sign

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderden'
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];

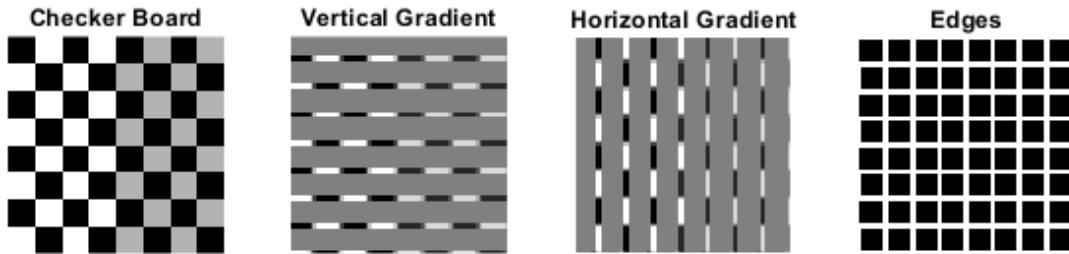
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_sobel_tb
```



### Create a New HDL Coder™ Project

Run the following command to create a new project.

```
coder -hdlcoder -new mlhdlc_ram
```

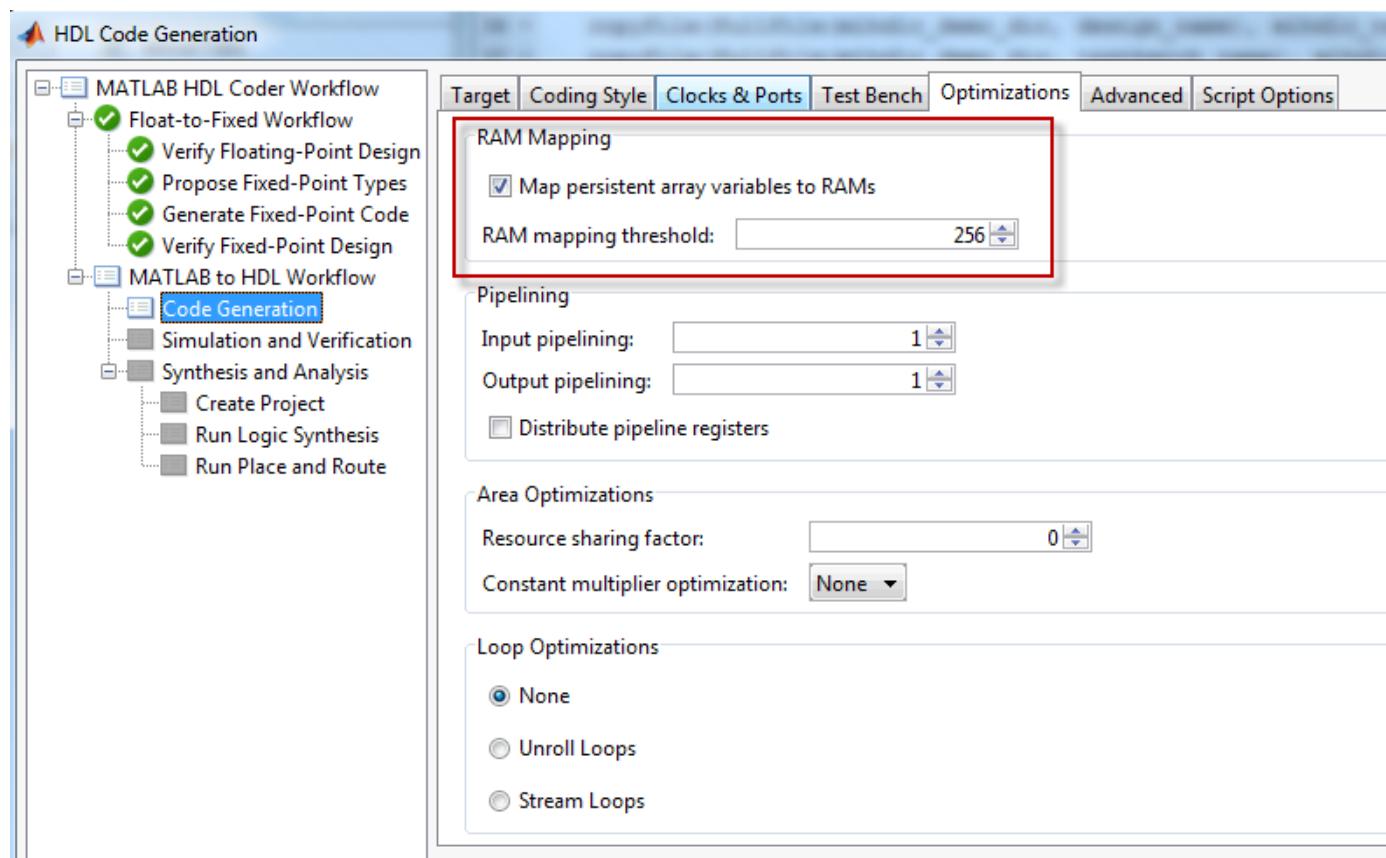
Next, add the file 'mlhdlc\_sobel.m' to the project as the MATLAB function, and 'mlhdlc\_sobel\_tb.m' as the MATLAB test bench.

Refer to “Getting Started with MATLAB to HDL Workflow” for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Turn On the RAM Mapping Optimization

Launch the Workflow Advisor.

The checkbox 'Map persistent array variables to RAMs' needs to be turned on to map persistent variables to block RAMs in the generated code.

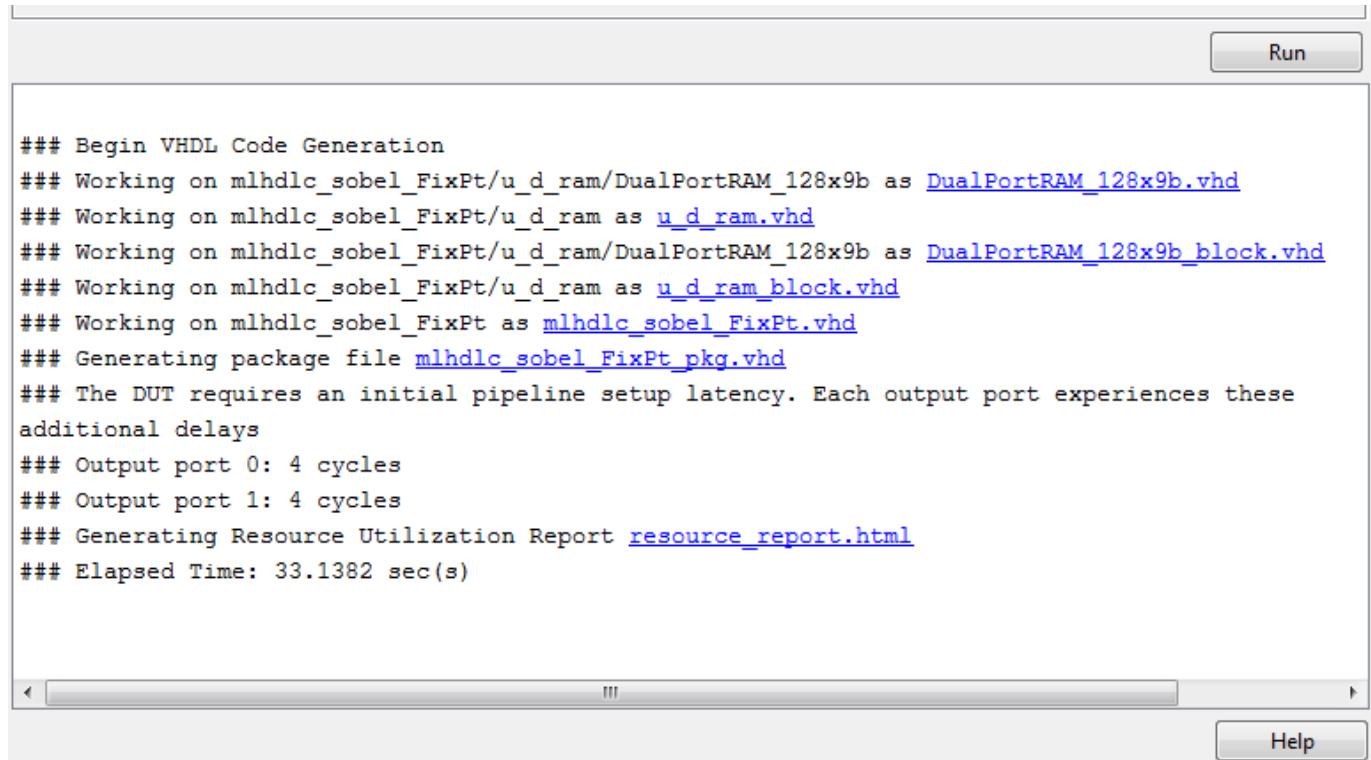


### Run Fixed-Point Conversion and HDL Code Generation

In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

### Examine the Generated Code

Examine the messages in the log window to see the RAM files generated along with the design.



```

### Begin VHDL Code Generation
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM\_128x9b.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u\_d\_ram.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram/DualPortRAM_128x9b as DualPortRAM\_128x9b\_block.vhd
### Working on mlhdlc_sobel_FixPt/u_d_ram as u\_d\_ram\_block.vhd
### Working on mlhdlc_sobel_FixPt as mlhdlc\_sobel\_FixPt.vhd
### Generating package file mlhdlc\_sobel\_FixPt\_pkg.vhd
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays
### Output port 0: 4 cycles
### Output port 1: 4 cycles
### Generating Resource Utilization Report resource\_report.html
### Elapsed Time: 33.1382 sec(s)

```

A warning message appears for each persistent matrix variable not mapped to RAM.

### Examine the Resource Report

Take a look at the generated resource report, which shows the number of RAMs inferred, by following the 'Resource Utilization report...' link in the generated code window.

Multiplicators	0
Adders/Subtractors	19
Registers	29
RAMs	2
Multiplexers	5

### Additional Notes on RAM Mapping

- Persistent matrix variable accesses must be in unconditional regions, i.e., outside any if-else, switch case, or for-loop code.
- MATLAB functions can have any number of RAM matrices.
- All matrix variables in MATLAB that are declared persistent and meet the threshold criteria get mapped to RAMs.
- A warning is shown when a persistent matrix does not get mapped to RAM.
- Read-dependent write data cycles are not allowed: you cannot compute the write data as a function of the data read from the matrix.

- Persistent matrices cannot be copied as a whole or accessed as a sub matrix: matrix access (read/write) is allowed only on single elements of the matrix.
- Mapping persistent matrices with non-zero initial values to RAMs is not supported.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sobel'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Map Persistent Arrays and `dsp.Delay` to RAM

### In this section...

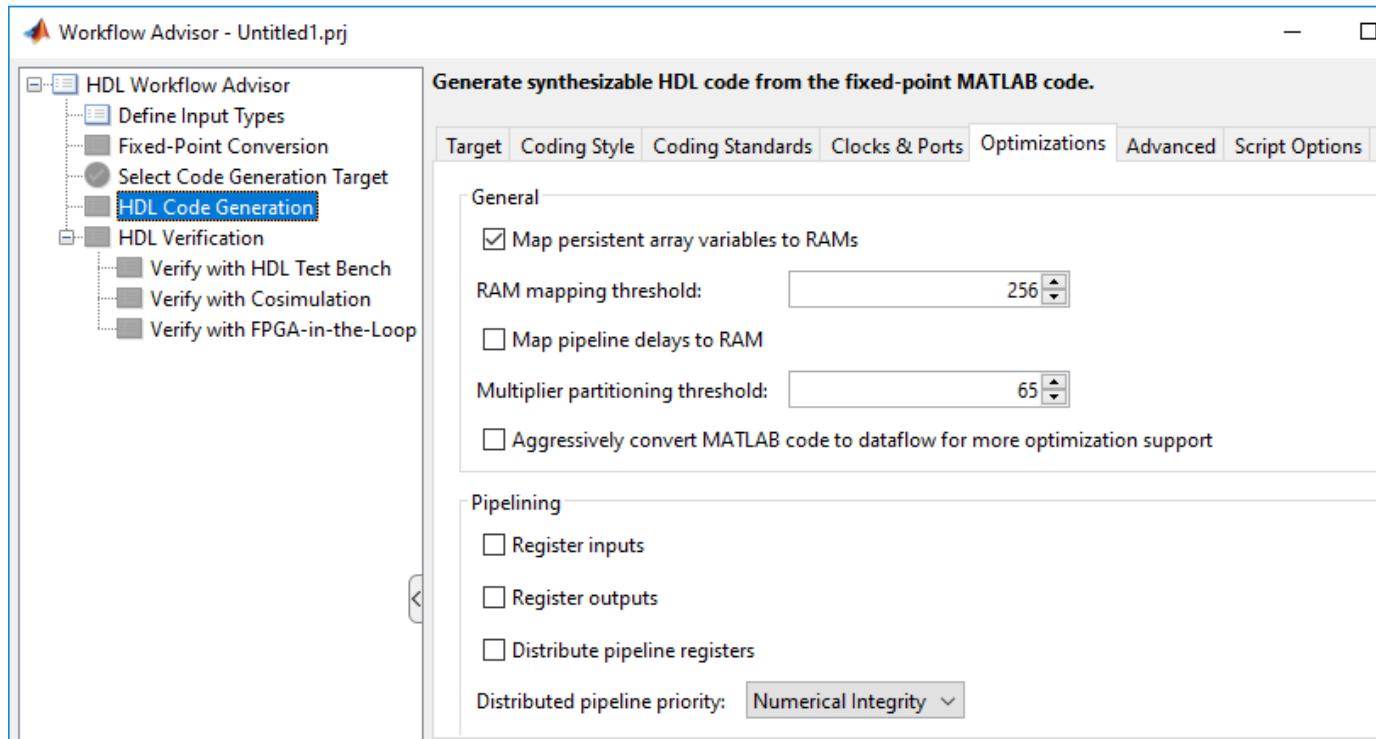
[“How To Enable RAM Mapping” on page 8-8](#)

[“RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-8](#)

[“RAM Mapping Requirements for `dsp.Delay` System Objects” on page 8-10](#)

### How To Enable RAM Mapping

- 1 In the HDL Workflow Advisor, select **MATLAB to HDL Workflow > Code Generation > Optimizations** tab.
- 2 Select the **Map persistent array variables to RAMs** option.
- 3 Set the **RAM mapping threshold** to the size (in bits) of the smallest persistent array, user-defined System object private property, or `dsp.Delay` that you want to map to RAM.



### RAM Mapping Requirements for Persistent Arrays and System object Properties

The following table shows a summary of the RAM mapping behavior for persistent arrays and private properties of a user-defined System object.

Map Persistent Array Variables to RAMs Setting	Mapping Behavior
on	Map to RAM. For restrictions, see "RAM Mapping Restrictions" on page 8-9.
off	Map to registers in the generated HDL code.

### RAM Mapping Restrictions

When you enable RAM mapping, a persistent array or user-defined System object private property maps to a block RAM when all of the following conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not allowed.
- Address computation logic is not read-dependent. For example, computation of a read or write address using the data read from the array is not allowed.
- Persistent variables or user-defined System object private properties are initialized to 0 if they have a cyclic dependency. For example, if you have two persistent variables, A and B, you have a cyclic dependency if A depends on B, and B depends on A.
- If an access is within a conditional statement, the conditional statement uses only simple logic expressions (`&&`, `||`, `~`) or relational operators. For example, in the following code, `r1` does not map to RAM:

```
if (mod(i,2) > 0)
    a = r1(u);
else
    r1(i) = u;
end
```

Rewrite complex conditions, such as conditions that call functions, by assigning them to temporary variables, and using the temporary variables in the conditional statement. For example, to map `r1` to RAM, rewrite the previous code as follows:

```
temp = mod(i,2);
if (temp > 0)
    a = r1(u);
else
    r1(i) = u;
end
```

- The persistent array or user-defined System object private property value depends on external inputs.

For example, in the following code, `bigarray` does not map to RAM because it does not depend on `u`:

```
function z = foo(u)

persistent cnt bigarray
if isempty(cnt)
    cnt = fi(0,1,16,10,hdlfimath);
    bigarray = uint8(zeros(1024,1));
end
z = u + cnt;
idx = uint8(cnt);
```

```

temp = bigarray(idx+1);
cnt(:) = cnt + fi(1,1,16,0,hdlfimath) + temp;
bigarray(idx+1) = idx;

```

- RAMSize is greater than or equal to the RAMMappingThreshold value. RAMSize is the product NumElements \* WordLength \* Complexity.
  - NumElements is the number of elements in the array.
  - WordLength is the number of bits that represent the data type of the array.
  - Complexity is 2 for arrays with a complex base type; 1 otherwise.

If any of the above conditions is false, the persistent array or user-defined System object private property maps to a register in the HDL code.

## RAM Mapping Requirements for dsp.Delay System Objects

A summary of the mapping behavior for a `dsp.Delay` System object is in the following table.

Map Persistent Array Variables to RAMs Option	Mapping Behavior
on	<p>A <code>dsp.Delay</code> System object maps to a block RAM when all of the following conditions are true:</p> <ul style="list-style-type: none"> <li>• Length property is greater than 4.</li> <li>• InitialConditions property is 0.</li> <li>• Delay input data type is one of the following:           <ul style="list-style-type: none"> <li>• Real scalar with a non-floating-point data type.</li> <li>• Complex scalar with real and imaginary parts that are non-floating-point.</li> <li>• Vector where each element is either a non-floating-point real scalar or complex scalar.</li> </ul> </li> <li>• <b>RAMSize</b> is greater than or equal to the <b>RAM Mapping Threshold</b> value.           <ul style="list-style-type: none"> <li>• <b>RAMSize</b> is the product <i>Length</i> * <i>InputWordLength</i>.</li> <li>• <i>InputWordLength</i> is the number of bits that represent the input data type.</li> </ul> </li> </ul> <p>If any of the conditions are false, the <code>dsp.Delay</code> System object maps to registers in the HDL code.</p>
off	A <code>dsp.Delay</code> System object maps to registers in the generated HDL code.

## RAM Mapping Comparison for MATLAB Code

`hdl.RAM`, `dsp.Delay`, persistent array variables, and user-definedSystem object private properties can map to RAM, but have different attributes. The following table summarizes the differences.

Attribute	<code>hdl.RAM</code>	<code>dsp.Delay</code>	<b>Persistent Arrays and User-Defined System object Properties</b>
RAM mapping criteria	Unconditionally maps to RAM	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for <code>dsp.Delay</code> System Objects” on page 8-10.	Maps to RAM in HDL code under specific conditions. See “RAM Mapping Requirements for Persistent Arrays and System object Properties” on page 8-8.
Address generation and port mapping	User specified	Automatic	Automatic
Access scheduling	User specified	Automatically inferred	Automatically inferred
Overclocking	None	None	Local multirate if access schedule requires it.
Latency with respect to simulation in MATLAB.	0	0	2 cycles if local multirate; 1 cycle otherwise.
RAM type	User specified	Dual port	Dual port

## Pipelining MATLAB Code

Pipelining helps achieve a higher maximum clock rate by inserting registers at strategic points in the hardware to break the critical path. However, the higher clock rate comes at the expense of increased chip area and increased initial latency.

### Port Registers

Input and output port registers for modules help partition a larger design so the critical path does not extend across module boundaries. Having a port register at each input and output port is a good design practice for synchronous interfaces. Distributed pipelining does not affect port registers. To insert input or output port registers:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
- 2 Enable **Register inputs**, **Register outputs**, or both.

### Input and Output Pipeline Registers

You can insert multiple input and output pipeline stages. Distributed pipelining can move these input and output pipeline registers to help reduce your critical path within the module. If you insert input and output pipeline stages without applying distributed pipelining, the registers stay at the DUT inputs and outputs.

To insert input or output pipeline register stages:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.
- 2 For **Input pipelining**, **Output pipelining**, or both, enter the number of pipeline register stages.

### Operation Pipelining

Operation pipelining inserts one or more registers at the output of a specific expression in your MATLAB code. If you know a specific expression is part of the critical path, you can add a pipeline register at its output to reduce your critical path.

To learn how to insert a pipeline register at the output of a MATLAB expression, see “Pipeline MATLAB Expressions” on page 8-13.

# Pipeline MATLAB Expressions

## In this section...

["How To Pipeline a MATLAB Expression" on page 8-13](#)

["Limitations of Pipelining for MATLAB Expressions" on page 8-13](#)

With the `coder.hdl.pipeline` pragma, you can specify the placement and number of pipeline registers in the HDL code generated for a MATLAB expression.

If you insert pipeline registers and enable distributed pipelining, HDL Coder automatically moves the pipeline registers to break the critical path.

## How To Pipeline a MATLAB Expression

To insert pipeline registers at the output of an expression in MATLAB code, place the expression in the `coder.hdl.pipeline` pragma. Specify the number of registers.

You can insert pipeline registers in the generated HDL code:

- At the output of the entire right side of an assignment statement.

The following code inserts three pipeline registers at the output of a MATLAB expression,  $a + b * c$ :

```
y = coder.hdl.pipeline(a + b * c, 3);
```

- At an intermediate stage within a longer MATLAB expression.

The following code inserts five pipeline registers after the computation of  $b * c$  within a longer expression,  $a + b * c$ :

```
y = a + coder.hdl.pipeline(b * c, 5);
```

- By nesting multiple instances of the pragma.

The following code inserts five pipeline registers after the computation of  $b * c$ , and two pipeline registers at the output of the whole expression,  $a + b * c$ :

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
```

Alternatively, to insert one pipeline register instead of multiple pipeline registers, you can omit the second argument in the pragma:

```
y = coder.hdl.pipeline(a + b * c);
y = a + coder.hdl.pipeline(b * c);
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c));
```

## Limitations of Pipelining for MATLAB Expressions

---

**Note** When you use the MATLAB code inside a MATLAB Function block and select the MATLAB Datapath architecture, these limitations do not apply.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see “Port Registers” on page 8-12.

## See Also

`coder.hdl.pipeline`

## More About

- “Pipelining MATLAB Code” on page 8-12

# Distributed Pipelining

## In this section...

["What is Distributed Pipelining?" on page 8-15](#)

["Benefits and Costs of Distributed Pipelining" on page 8-15](#)

["Selected Bibliography" on page 8-15](#)

## What is Distributed Pipelining?

Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior.

The HDL Coder software uses an adaptation of the Leiserson-Saxe retiming algorithm.

## Benefits and Costs of Distributed Pipelining

Distributed pipelining can reduce your design's critical path, enabling you to use a higher clock rate and increase throughput.

However, distributed pipelining requires your design to contain a number of delays. If you need to insert additional delays in your design to enable distributed pipelining, this increases the area and the initial latency of your design.

## Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

## Distributed Pipelining for Clock Speed Optimization

This example shows how to use the distributed pipelining and loop unrolling optimizations in HDL Coder to optimize clock speed.

### Introduction

Distributed pipelining is a design-wide optimization supported by HDL Coder for improving clock frequency. When you turn on the 'Distribute Pipeline Registers' option in HDL Coder, the coder redistributes the input and output pipeline registers of the top level function along with other registers in the design in order to minimize the combinatorial logic between registers and thus maximize the clock speed of the chip synthesized from the generated HDL code.

Consider the following example design of a FIR filter. The combinatorial logic from an input or a register to an output or another register contains a sum of products. Loop unrolling and distributed pipelining moves the output registers at the design level to reduce the amount of combinatorial logic, thus increasing clock speed.

### MATLAB® Design

The MATLAB code used in the example is a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';

1 Design: mlhdlc_fir
2 Test Bench: mlhdlc_fir_tb
```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

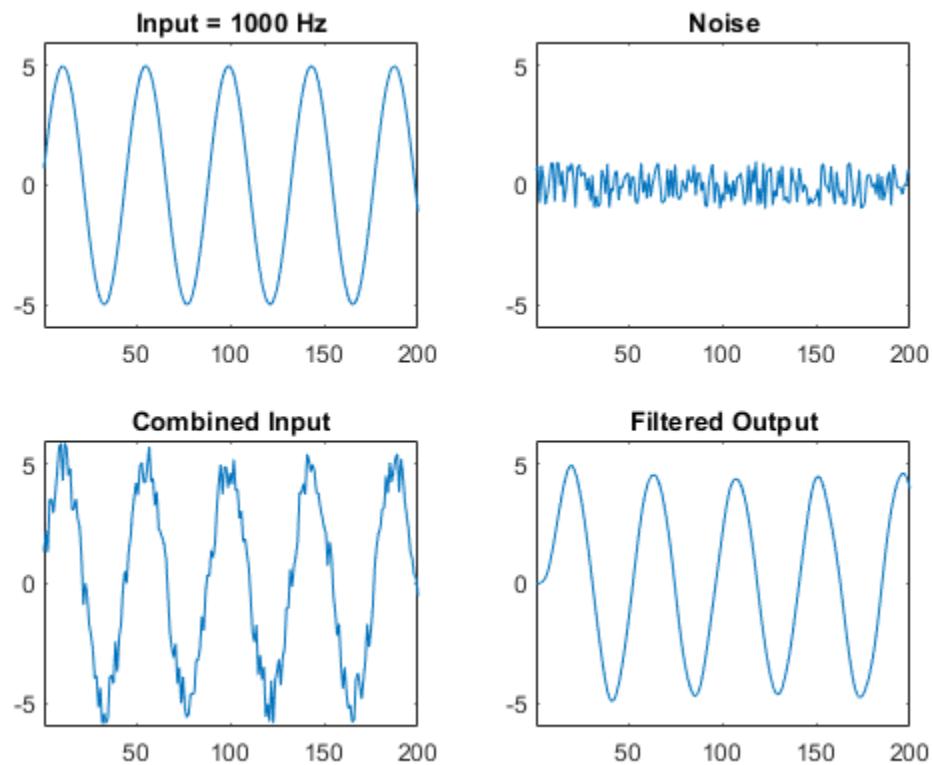
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

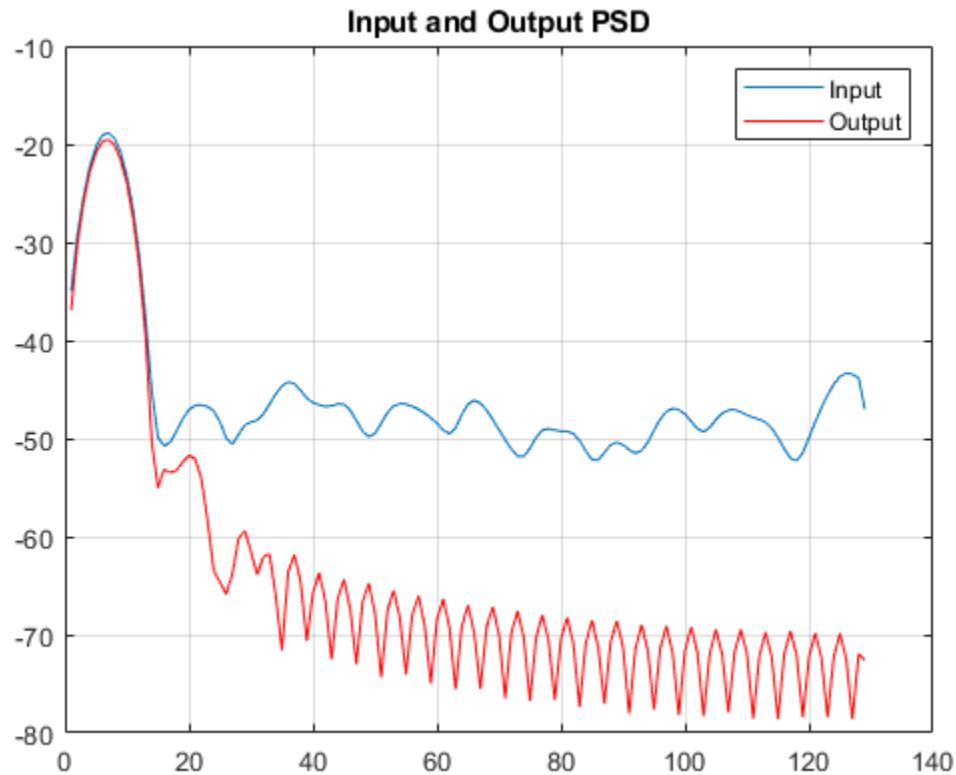
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no run-time errors.

```
mlhdlc_fir_tb
```





### Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_fir_tb';
```

### Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_fir_tb';
```

### Distributed Pipelining

To increase the clock speed, the user can set a number of input and output pipeline stages for any design. In this particular example Input pipelining option is set to '1' and Output pipelining option is set to '20'. Without any additional options turned on these settings will add one input pipeline register at all input ports of the top level design and 20 output pipeline registers at each of the output ports.

If the option 'Distribute pipeline registers' is enabled, HDL Coder tries to reposition the registers to achieve the best clock frequency.

In addition to moving the input and output pipeline registers, HDL Coder also tries to move the registers modeled internally in the design using persistent variables or with system objects like `dsp.Delay`.

Additional opportunities for improvements become available if you unroll loops. The 'Unroll Loops' option unrolls explicit for-loops in MATLAB code in addition to implicit for-loops that are inferred for vector and matrix operations. 'Unroll Loops' is necessary for this example to do distributed pipelining.

```
hdlcfg.InputPipeline = 1;
hdlcfg.OutputPipeline = 20;
hdlcfg.DistributedPipelining = true;
hdlcfg.LoopOptimization = 'UnrollLoops';
```

### **Examine the Synthesis Results**

If you have ISE installed on your machine, run the logic synthesis step

```
hdlcfg.SynthesizeGeneratedCode = true;
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_fir
```

View the result report

```
edit codegen/mlhdlc_fir/hdlsrc/ise_prj/mlhdlc_fir_fixpt_syn_results.txt
```

In the synthesis report, note the clock frequency reported by the synthesis tool. When you synthesize the design with the loop unrolling and distributed pipelining options enabled, you see a significant clock frequency increase with pipelining options turned on.

### **Clean Up the Generated Files**

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Optimize MATLAB Loops

## In this section...

- "Loop Streaming" on page 8-20
- "Loop Unrolling" on page 8-20
- "How to Optimize MATLAB Loops" on page 8-20
- "Limitations for MATLAB Loop Optimization" on page 8-21

With loop optimization, you can stream or unroll loops in generated code. Loop streaming is an area optimization, and loop unrolling is a speed optimization. To optimize loops for MATLAB code that is inside a MATLAB Function block, use the **MATLAB Function** architecture. When you use the **MATLAB Datapath** architecture, the code generator unrolls loops irrespective of the loop optimization setting.

## Loop Streaming

HDL Coder streams a loop by instantiating the loop body once and using that instance for each loop iteration. The code generator oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop.

If you stream a loop, the advantage is decreased hardware resource usage because the loop body is instantiated fewer times. The disadvantage is the hardware implementation runs at a lower speed.

You can partially stream a loop. A partially streamed loop instantiates the loop body more than once, so it uses more area than a fully streamed loop. However, a partially streamed loop also uses less oversampling than a fully streamed loop.

## Loop Unrolling

HDL Coder unrolls a loop by instantiating multiple instances of the loop body in the generated code. You can also partially unroll a loop. The generated code uses a loop statement that contains multiple instances of the original loop body and fewer iterations than the original loop.

The distributed pipelining and resource sharing can optimize the unrolled code. Distributed pipelining can increase speed. Resource sharing can decrease area.

When loop unrolling creates multiple instances, these instances are likely to increase area. Loop unrolling also makes the code harder to read.

## How to Optimize MATLAB Loops

You can specify a global loop optimization by using the HDL Workflow Advisor, or at the command line.

You can also specify a local loop optimization for a specific loop by using the `coder.hdl.loopspec` pragma in the MATLAB code. If you specify both a global and local loop optimization, the local loop optimization overrides the global setting.

### Global Loop Optimization

To specify a loop optimization in the Workflow Advisor:

- 1 In the HDL Workflow Advisor left pane, select **HDL Workflow Advisor > HDL Code Generation**.
- 2 In the **Optimizations** tab, for **Loop Optimizations**, select **None**, **Unroll Loops**, or **Stream Loops**.

To specify a loop optimization at the command line in the MATLAB to HDL workflow, specify the `LoopOptimization` property of the `coder.HdlConfig` object. For example, for a `coder.HdlConfig` object, `hdlcfg`, enter one of the following commands:

```
hdlcfg.LoopOptimization = 'UnrollLoops'; % unroll loops
hdlcfg.LoopOptimization = 'StreamLoops'; % stream loops
hdlcfg.LoopOptimization = 'LoopNone'; % no loop optimization
```

### **Local Loop Optimization**

To learn how to optimize a specific MATLAB loop, see `coder.hdl.loopspec`.

---

**Note** If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

---

## **Limitations for MATLAB Loop Optimization**

HDL Coder cannot stream a loop if:

- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are two or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.
- A persistent variable that is initialized to a nonzero value is updated inside the loop.

HDL Coder can stream a loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

You cannot use the `coder.hdl.loopspec('stream')` pragma:

- In a subfunction. You must specify it in the top-level MATLAB design function.
- For a loop that is nested within another loop.
- For a loop containing a nested loop, unless the streaming factor is equal to the number of iterations.

### **See Also**

`coder.hdl.loopspec`

## Constant Multiplier Optimization

### In this section...

["What is Constant Multiplier Optimization?" on page 8-22](#)

["Specify Constant Multiplier Optimization" on page 8-22](#)

### What is Constant Multiplier Optimization?

The **Constant multiplier optimization** option enables you to specify use of canonical signed digit (CSD) or factored CSD (FCSD) optimizations for processing coefficient multiplier operations.

The following table shows the **Constant multiplier optimization** values.

Constant Multiplier Optimization Value	Description
<b>None</b> (default)	By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
<b>CSD</b>	<p>When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations.</p> <p>CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.</p>
<b>FCSD</b>	<p>This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction.</p> <p>This option lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.</p>
<b>Auto</b>	<p>When you specify this option, HDL Coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required.</p> <p>HDL Coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).</p>

### Specify Constant Multiplier Optimization

To specify constant multiplier optimization:

- 1 In the HDL Workflow Advisor, select the **HDL Code Generation** task and select the **Optimizations** tab.

- 2 For **Constant multiplier optimization**, select **CSD**, **FCSD**, or **Auto**.

## Resource Sharing of Multipliers to Reduce Area

This example shows how to use the resource sharing optimization in HDL Coder™. This optimization identifies functionally equivalent multiplier operations in MATLAB® code and shares them in order to optimize design area. You have control over the number of multipliers to be shared in the design.

### Introduction

Resource sharing is a design-wide optimization supported by HDL Coder™ for implementing area-efficient hardware.

This optimization enables users to share hardware resources by mapping 'N' functionally-equivalent MATLAB operators, in this case multipliers, to a single operator.

The user specifies 'N' using the 'Resource Sharing Factor' option in the optimization panel.

Consider the following example model of a symmetric FIR filter. It contains 4 product blocks that are functionally equivalent and which are mapped to 4 multipliers in hardware. The Resource Utilization Report shows the number of multipliers inferred from the design.

In this example you will run fixed-point conversion on the MATLAB design 'mlhdlc\_sharing' followed by HDL Coder. This prerequisite step normalizes all the multipliers used in the fixed-point code. You will input a 'proposed-type settings' during this fixed-point conversion phase.

### MATLAB Design

The MATLAB code used in the example is a simple symmetric FIR filter written in MATLAB and also has a testbench that exercises the filter.

```
design_name = 'mlhdlc_sharing';
testbench_name = 'mlhdlc_sharing_tb';
```

Let us take a look at the MATLAB design.

```
type(design_name);

%%%%%%%%%%%%%
% MATLAB design: Symmetric FIR Filter
%
% Key Design pattern covered in this example:
% (1) Filter states represented using the persistent variables
% (2) Filter coefficients passed in as parameters
%%%%%%%%%%%%%

% Copyright 2011-2015 The MathWorks, Inc.

%#codegen
function [y_out, x_out] = mlhdlc_sharing(x_in, h)
% Symmetric FIR Filter

persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
    ud1 = 0; ud2 = 0; ud3 = 0; ud4 = 0; ud5 = 0; ud6 = 0; ud7 = 0; ud8 = 0;
end

x_out = ud8;
```

```

a1 = ud1 + ud8;
a2 = ud2 + ud7;
a3 = ud3 + ud6;
a4 = ud4 + ud5;

% filtered output
y_out = (h(1) * a1 + h(2) * a2) + (h(3) * a3 + h(4) * a4);

% update the delay line
ud8 = ud7;
ud7 = ud6;
ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
ud1 = x_in;

end

type(testbench_name);

%%%%%%%%%%%%%
% MATLAB test bench for the FIR filter
%%%%%%%%%%%%%

% Copyright 2011-2015 The MathWorks, Inc.

clear mlhdlc_sharing;

% input signal with noise
x_in = cos(3.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).';

len = length(x_in);
y_out = zeros(1,len);
x_out = zeros(1,len);

% Define a regular MATLAB constant array:
%
% filter coefficients
h = [-0.1339 -0.0838 0.2026 0.4064];

for ii=1:len
    data = x_in(ii);
    % call to the design 'mlhdlc_sfir' that is targeted for hardware
    [y_out(ii), x_out(ii)] = mlhdlc_sharing(data, h);
end

figure('Name', [mfilename, '_plot']);
plot(1:len,y_out);

```

### Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];

% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

### Create a New HDL Coder Project

Run the following command to create a new project:

```
coder -hdlcoder -new mlhdlc_sfir_sharing
```

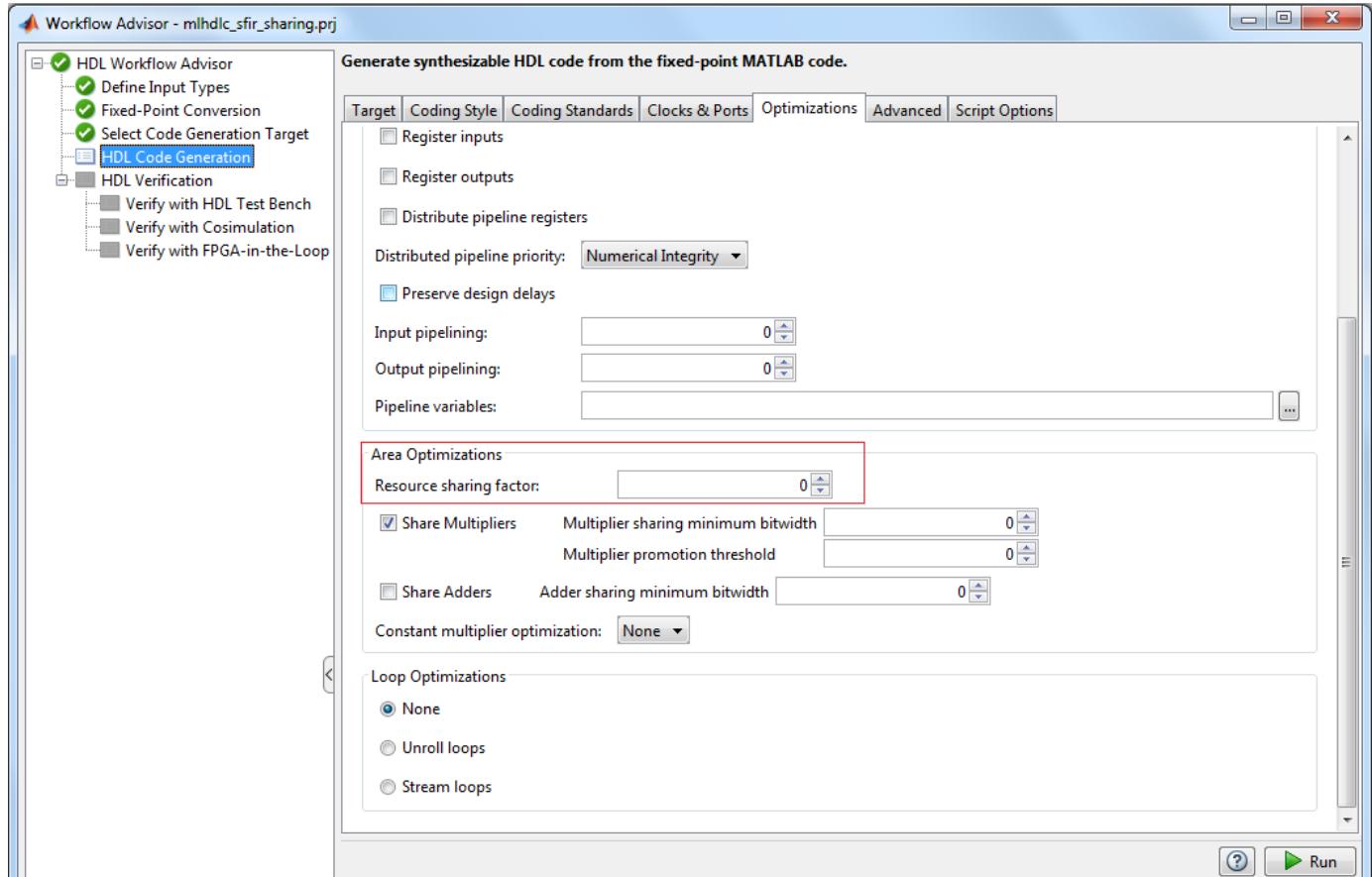
Next, add the file 'mlhdlc\_sharing.m' to the project as the MATLAB Function and 'mlhdlc\_sharing\_tb.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### Realize an N-to-1 Mapping of Multipliers

Turn on the resource sharing optimization by setting the 'Resource Sharing Factor' to a positive integer value.

This parameter specifies 'N' in the N-to-1 hardware mapping. Choose a value of  $N > 1$ .



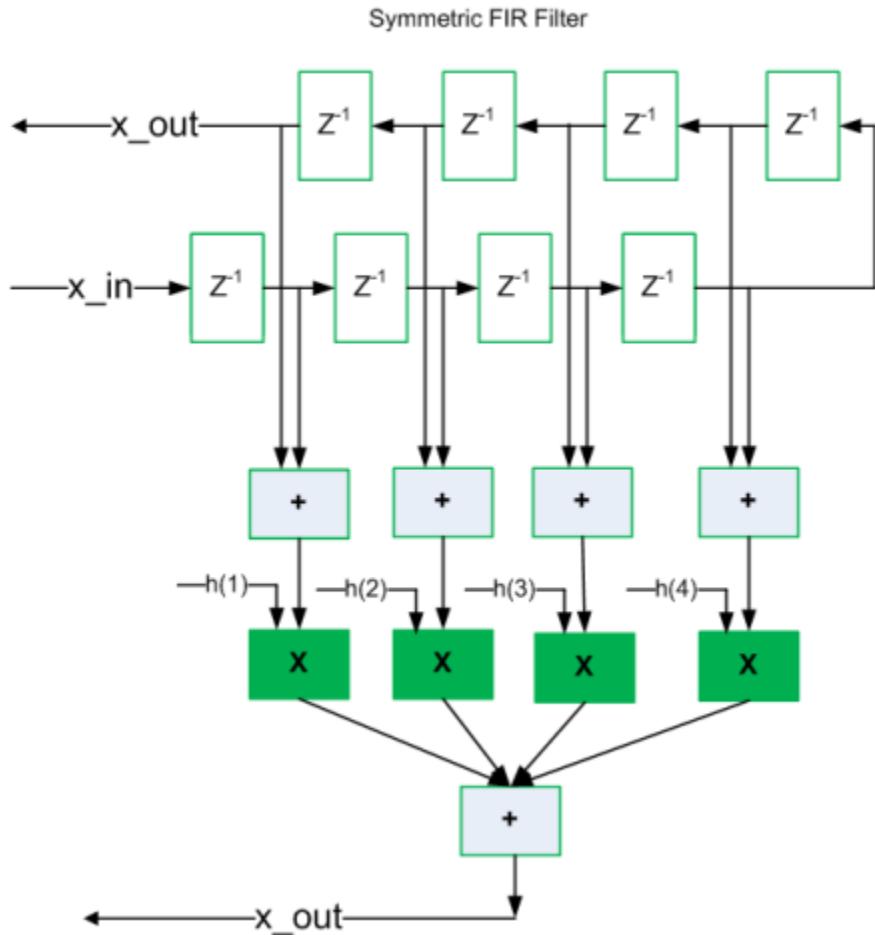
### Examine the Resource Report

There are 4 multiplication operators in this example design. Generating HDL with a 'SharingFactor' of 4 will result in only one multiplier in the generated code.

<b>Multipliers</b>	1
Adders/Subtractors	7
Registers	29
RAMs	0
Multiplexers	12

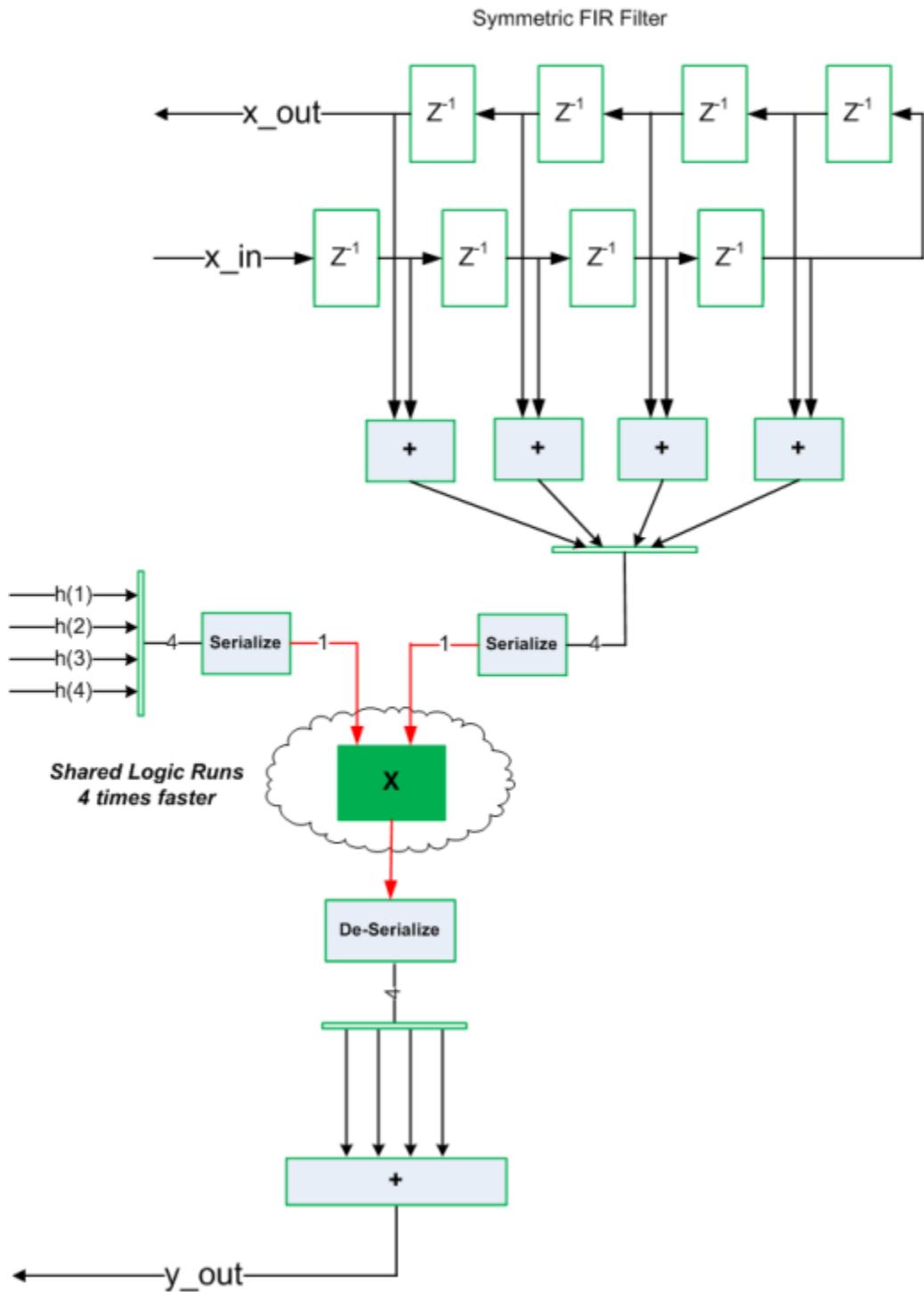
### Sharing Architecture

The following figure shows how the algorithm is implemented in hardware when we synthesize the generated code without turning on the sharing optimization.



The following figure shows the sharing architecture automatically implemented by HDL Coder when the sharing optimization option is turned on.

The inputs to the shared multiplier are time-multiplexed at a faster rate (in this case 4x faster and denoted in red). The outputs are then routed to the respective consumers at a slower rate (in green).



#### Run Fixed-Point Conversion and HDL Code Generation

Launch the Workflow Advisor and right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through the HDL code generation.

The detailed example “Fixed-Point Type Conversion and Derived Ranges” on page 4-72 provides a tutorial for updating the type proposal settings during fixed-point conversion.

Note that to share multipliers of different word-length, in the Optimization -> Resource Sharing tab of HDL Configuration Parameters, specify the 'Multiplier promotion threshold'. For more information, see the Resource Sharing Documentation.

### Run Synthesis and Examine Synthesis Results

Synthesize the generated code from the design with this optimization turned off, then with it turned on, and examine the area numbers in the resource report.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde');
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir_sharing'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

# Loop Streaming to Reduce Area

This example shows how to use the design-level loop streaming optimization in HDL Coder™ to optimize area.

## Introduction

A MATLAB® for loop generates a FOR\_GENERATE loop in VHDL. Such loops are always spatially unrolled for execution in hardware. In other words, the body of the software loop is replicated as many times in hardware as the number of loop iterations. This results in inefficient area usage.

The loop streaming optimization creates an alternative implementation of a software loop, where the body of the loop is shared in hardware. Instead of spatially replicating copies of the loop body, HDL Coder™ creates a single hardware instance of the loop body that is time-multiplexed across loop iterations.

## MATLAB Design

The MATLAB code used in this example implements a simple FIR filter. This example also shows a MATLAB testbench that exercises the filter.

```
design_name = 'mlhdlc_fir';
testbench_name = 'mlhdlc_fir_tb';

1 Design: mlhdlc_fir
2 Test Bench: mlhdlc_fir_tb
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];

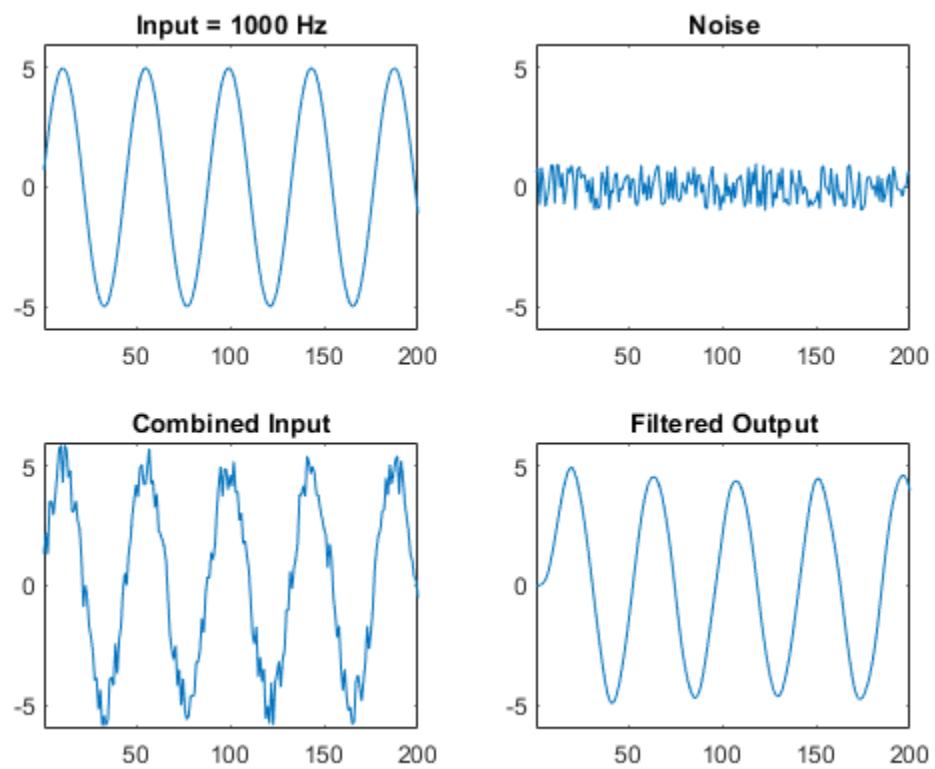
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

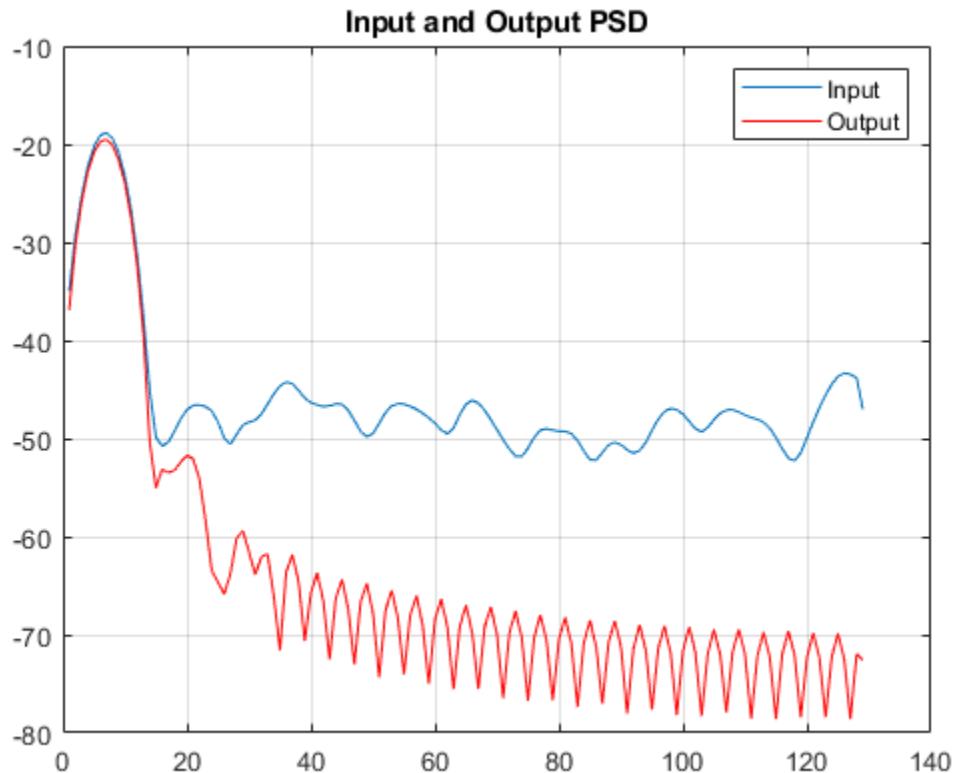
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the testbench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_fir_tb
```





### **Creating a New Project From the Command Line**

To create a new project, enter the following command:

```
coder -hdlcoder -new fir_project
```

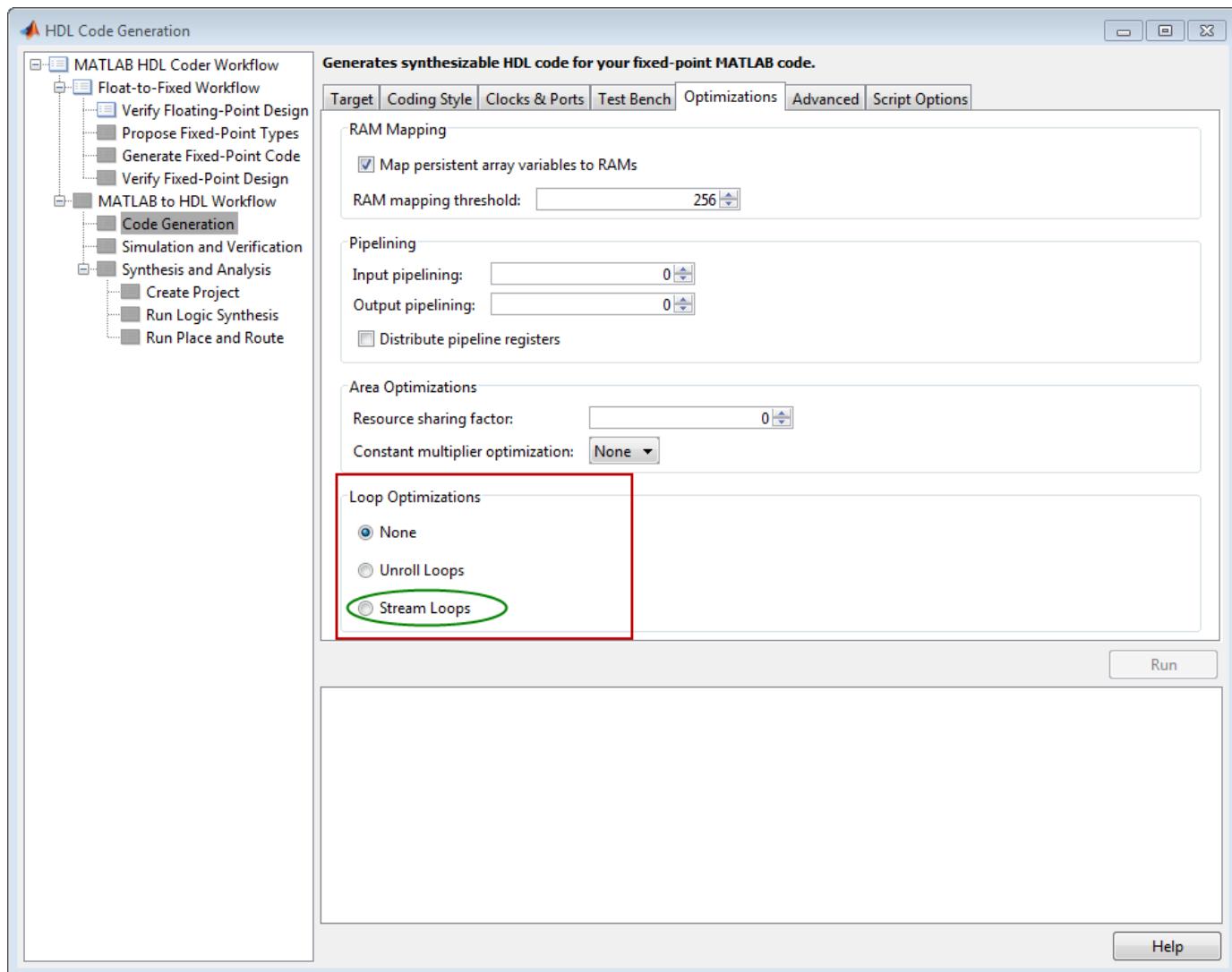
Next, add the file 'mlhdlc\_fir.m' to the project as the MATLAB Function and 'mlhdlc\_fir\_tb.m' as the MATLAB Test Bench.

Launch the Workflow Advisor.

Refer to "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

### **Turn On Loop Streaming**

The loop streaming optimization in HDL Coder converts software loops (either written explicitly using a for-loop statement, or inferred loops from matrix/vector operators) to area-friendly hardware loops.



### Run Fixed-Point Conversion and HDL Code Generation

Right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

### Examine the Generated Code

When you synthesize the design with the loop streaming optimization, you see a reduction in area resources in the resource report. Try generating HDL code with and without the optimization.

The resource report without the loop streaming optimization:

Multiplicators	16
Adders/Subtractors	31
Registers	106
RAMs	0
Multiplexers	0

The resource report with the loop streaming optimization enabled:

Multiplicators	1
Adders/Subtractors	17
Registers	448
RAMs	0
Multiplexers	5

### Known Limitations

Loops will be streamed only if they are regular nested loops. A regular nested loop structure is defined as one where:

- None of the loops in any level of nesting appear in a conditional flow region, i.e. no loop can be embedded within if-else or switch-else regions.
- Loop index variables are monotonically increasing.
- Total number of iterations of the loop structure is non-zero.
- There are no back-to-back loops at the same level of the nesting hierarchy.

### Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_fir'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

## Constant Multiplier Optimization to Reduce Area

This example shows how to perform a design-level area optimization in HDL Coder™ by converting constant multipliers into shifts and adds using canonical signed digit (CSD) techniques. The CSD representation of multiplier constants (for example, in gain coefficients or filter coefficients) significantly reduces the area of the hardware implementation.

### Canonical Signed Digit (CSD) Representation

A signed digit (SD) representation is an augmented binary representation with weights 0, 1 and -1. -1 is represented in HDL Coder generated code as 1'.

$$X_{10} = \sum_{r=0}^{B-1} x_r \cdot 2^r$$

where

$$x_r = 0, 1, -1(\bar{1})$$

For example, here are a couple of signed digit representations for 93:

$$X_{10} = 64 + 16 + 13 = 01011101$$

$$X_{10} = 128 - 32 - 2 - 1 = 10\bar{1}000\bar{1}\bar{1}$$

Note that the signed digit representation is non-unique. A canonical signed digit (CSD) representation is an SD representation with the minimum number of nonzero elements.

Here are some properties of CSD numbers:

- 1 No two consecutive bits in a CSD number are nonzero
- 2 CSD representation uses minimum number of nonzero digits
- 3 CSD representation of a number is unique

### CSD Multiplier

Let us see how a CSD representation can yield an implementation requiring a minimum number of adders.

Let us look at CSD example:

```
y = 231 * x
  = (11100111) * x % 231 in binary form
  = (1001'01001') * x % 231 in signed digit form
  = (256 - 32 + 8 - 1) * x %
  = (x << 8) - (x << 5) + (x << 3) - x % cost of CSD: 3 Adders
```

### HDL Coder CSD Implementation

HDL Coder uses a CSD implementation that differs from the traditional CSD implementation. This implementation preferentially chooses adders over subtractors when using the signed digit representation. In this representation, sometimes two consecutive bits in a CSD number can be nonzero. However, similar to the CSD implementation, the HDL Coder implementation uses the minimum number of nonzero digits. For example:

In the traditional CSD implementation, the number 1373 is represented as:

$$1373 = 0101'01'01'001'01$$

This implementation does not have two consecutive nonzero digits in the representation. The cost of this implementation is 1 adder and 4 subtractors.

In the HDL Coder CSD implementation, the number 1373 is represented as:

$$1373 = 00101011001'01$$

This implementation has two consecutive nonzero digits in the representation but uses the same number of nonzero digits as the previous CSD implementation. The cost of this implementation is 4 adders and 1 subtractor which shows that adders are preferred to subtractors.

### **FCSD Multiplier**

A combination of factorization and CSD representation of a constant multiplier can lead to further reduction in hardware cost (number of adders).

FCSD can further reduce the number of adders in the above constant multiplier:

```
y = 231 * x
y = (7 * 33) * x
y_tmp = (x << 5) + x
y = (y_tmp << 3) - y_tmp           % cost of FCSD: 2 Adders
```

### **CSD/FCSD Costs**

This table shows the costs (C) of all 8-bit multipliers.

$C$	Coefficient
0	1, 2, 4, 8, 16, 32, 64, 128, 256
1	3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255
2	11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 44, 46, 47, 49, 50, 52, 54, 55, 57, 58, 59, 61, 67, 69, 70, 71, 73, 74, 76, 78, 79, 81, 82, 84, 88, 92, 94, 95, 97, 98, 100, 104, 108, 110, 111, 113, 114, 116, 118, 119, 121, 122, 123, 125, 131, 133, 134, 135, 137, 138, 140, 142, 143, 145, 146, 148, 152, 156, 158, 159, 161, 162, 164, 168, 176, 184, 188, 190, 191, 193, 194, 196, 200, 208, 216, 220, 222, 223, 225, 226, 228, 232, 236, 238, 239, 241, 242, 244, 246, 247, 249, 250, 251, 253
3	43, 45, 51, 53, 75, 77, 83, 85, 86, 87, 89, 90, 91, 93, 99, 101, 102, 103, 105, 106, 107, 109, 115, 117, 139, 141, 147, 149, 150, 151, 153, 154, 155, 157, 163, 165, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 187, 189, 195, 197, 198, 199, 201, 202, 204, 206, 207, 209, 210, 212, 214, 215, 217, 218, 219, 221, 227, 229, 230, 231, 233, 234, 235, 237, 243, 245
4	171, 173, 179, 181, 203, 205, 211, 213
Minimum costs through factorization	
2	$45 = 5 \times 9, 51 = 3 \times 17, 75 = 5 \times 15, 85 = 5 \times 17, 90 = 2 \times 9 \times 5, 93 = 3 \times 31, 99 = 3 \times 33, 102 = 2 \times 3 \times 17, 105 = 7 \times 15, 150 = 2 \times 5 \times 15, 153 = 9 \times 17, 155 = 5 \times 31, 165 = 5 \times 33, 170 = 2 \times 5 \times 17, 180 = 4 \times 5 \times 9, 186 = 2 \times 3 \times 31, 189 = 7 \times 9, 195 = 3 \times 65, 198 = 2 \times 3 \times 33, 204 = 4 \times 3 \times 17, 210 = 2 \times 7 \times 15, 217 = 7 \times 31, 231 = 7 \times 33$
3	$171 = 3 \times 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 \times 71, 205 = 5 \times 41, 203 = 7 \times 29$

Reference: *Digital Signal Processing with FPGAs* by Uwe Meyer-Baese

## MATLAB® Design

The MATLAB code used in this example implements a simple FIR filter. The example also shows a MATLAB test bench that exercises the filter.

```
design_name = 'mlhdlc_csd';
testbench_name = 'mlhdlc_csd_tb';

1 Design: mlhdlc_csd
2 Test Bench: mlhdlc_csd_tb
```

## Create a New Folder and Copy Relevant Files

Execute the following lines of code to copy the necessary example files into a temporary folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderder');
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];
```

```
% create a temporary folder and copy the MATLAB files
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
```

## Simulate the Design

Simulate the design with the test bench prior to code generation to make sure there are no runtime errors.

```
mlhdlc_csd_tb
```

## Create a Fixed-Point Conversion Config Object

To perform fixed-point conversion, you need a 'fixpt' config object.

Create a 'fixpt' config object and specify your test bench name:

```
close all;
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_csd_tb';
```

## Create an HDL Code Generation Config Object

To generate code, you must create an 'hdl' config object and set your test bench name:

```
hdlcfg = coder.config('hdl');
hdlcfg.TestBenchName = 'mlhdlc_csd_tb';
```

## Generate Code without Constant Multiplier Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'None';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```
329 -- filtered output
330 --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331 p22y_out_mul_temp <= (-2194) * a1;
332 p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
333 p22y_out_mul_temp_1 <= (-1373) * a2;
334 p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
335 p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
336 p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
337 p22y_out_mul_temp_2 <= 3319 * a3;
338 p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
339 p22y_out_mul_temp_3 <= 6658 * a4;
340 p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
341 p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
342 p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
343 p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
344 y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
345
```

Take a look at the resource report for adder and multiplier usage without the CSD optimization.

Multipliers	4	
Adders/Subtractors	7	
Registers		23
RAMs		0
Multiplexers		0

### Generate Code with CSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'CSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```

329 -- filtered output
330 --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1 )*a1 + h( 2 )*a2) + (h( 3 )*a3 + h( 4 )*a4), 1, 14, 12, fm);
331 -- CSD Encoding (2194) : 0100010010010; Cost (Adders) = 3
332 p22y_out_mul_temp <= - (((resize(a1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a1 & '0' &
333 p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
334 -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
335 p22y_out_mul_temp_1 <= - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' &
336 p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
337 p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
338 p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
339 -- CSD Encoding (3319) : 0110100001'001; Cost (Adders) = 4
340 p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' &
341 p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
342 -- CSD Encoding (6658) : 01101000000010; Cost (Adders) = 3
343 p22y_out_mul_temp_3 <= ((resize(a4 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a4 & '0'
344 p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
345 p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
346 p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
347 p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
348 y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
```

Examine the code with comments that outline the CSD encoding for all the constant multipliers.

Look at the resource report and notice that with the CSD optimization, the number of multipliers is reduced to zero and multipliers are replaced by shifts and adders.

Multipliers	0	
Adders/Subtractors	24	
Registers		23
RAMs		0
Multiplexers		0

## Generate Code with FCSD Optimization

```
hdlcfg.ConstantMultiplierOptimization = 'FCSD';
```

Enable the 'Unroll Loops' option to inline multiplier constants.

```
hdlcfg.LoopOptimization = 'UnrollLoops';
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_csd
```

Examine the generated code.

```
331 -- filtered output
332 --'mlhdlc_csd_FixPt:40' y_out = fi((h( 1)*a1 + h( 2)*a2) + (h( 3)*a3 + h( 4)*a4), 1, 14, 12, fm);
333 -- FCSD for 2194 = 2 X 1097; Total Cost = 3
334 -- CSD Encoding (2) : 10; Cost (Adders) = 0
335 p22y_out_factor <= resize(a1 & '0', 28);
336 -- CSD Encoding (1097) : 010001001001; Cost (Adders) = 3
337 p22y_out_mul_temp <= - (((resize(p22y_out_factor & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor & '0' & '0' & '0' &
338 p22y_out_add_cast <= resize(p22y_out_mul_temp, 29);
339 -- CSD Encoding (1373) : 0101011001'01; Cost (Adders) = 5
340 p22y_out_mul_temp_1 <= - (((((resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a2 & '0' & '0' & '0' & '0' & '0' & '0' &
341 p22y_out_add_cast_1 <= resize(p22y_out_mul_temp_1, 29);
342 p22y_out_add_temp <= p22y_out_add_cast + p22y_out_add_cast_1;
343 p22y_out_add_cast_2 <= resize(p22y_out_add_temp, 30);
344 -- CSD Encoding (319) : 0110100001'001'; Cost (Adders) = 4
345 p22y_out_mul_temp_2 <= (((resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(a3 & '0' & '0' & '0' & '0' & '0' & '0' &
346 p22y_out_add_cast_3 <= resize(p22y_out_mul_temp_2, 29);
347 -- FCSD for 6658 = 2 X 3329; Total Cost = 3
348 -- CSD Encoding (2) : 10; Cost (Adders) = 0
349 p22y_out_factor_1 <= resize(a4 & '0', 28);
350 -- CSD Encoding (3329) : 0110100000001; Cost (Adders) = 3
351 p22y_out_mul_temp_3 <= ((resize(p22y_out_factor_1 & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 28) + resize(p22y_out_factor_1 & '0' &
352 p22y_out_add_cast_4 <= resize(p22y_out_mul_temp_3, 29);
353 p22y_out_add_temp_1 <= p22y_out_add_cast_3 + p22y_out_add_cast_4;
354 p22y_out_add_cast_5 <= resize(p22y_out_add_temp_1, 30);
355 p22y_out_add_temp_2 <= p22y_out_add_cast_2 + p22y_out_add_cast_5;
356 y_out_1 <= p22y_out_add_temp_2(26 DOWNTO 13);
357
358 y_out_2 <= y_out_1;
```

Examine the code with comments that outline the FCSD encoding for all the constant multipliers. In this particular example, the generated code is identical in terms of area resources for the multiplier constants. However, take a look at the factorizations of the constants in the generated code.

If you choose the 'Auto' option, HDL Coder will automatically choose between the CSD and FCSD options for the best result.

## Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderdemo');
mlhdlc_temp_dir = [tempdir 'mlhdlc_csd'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

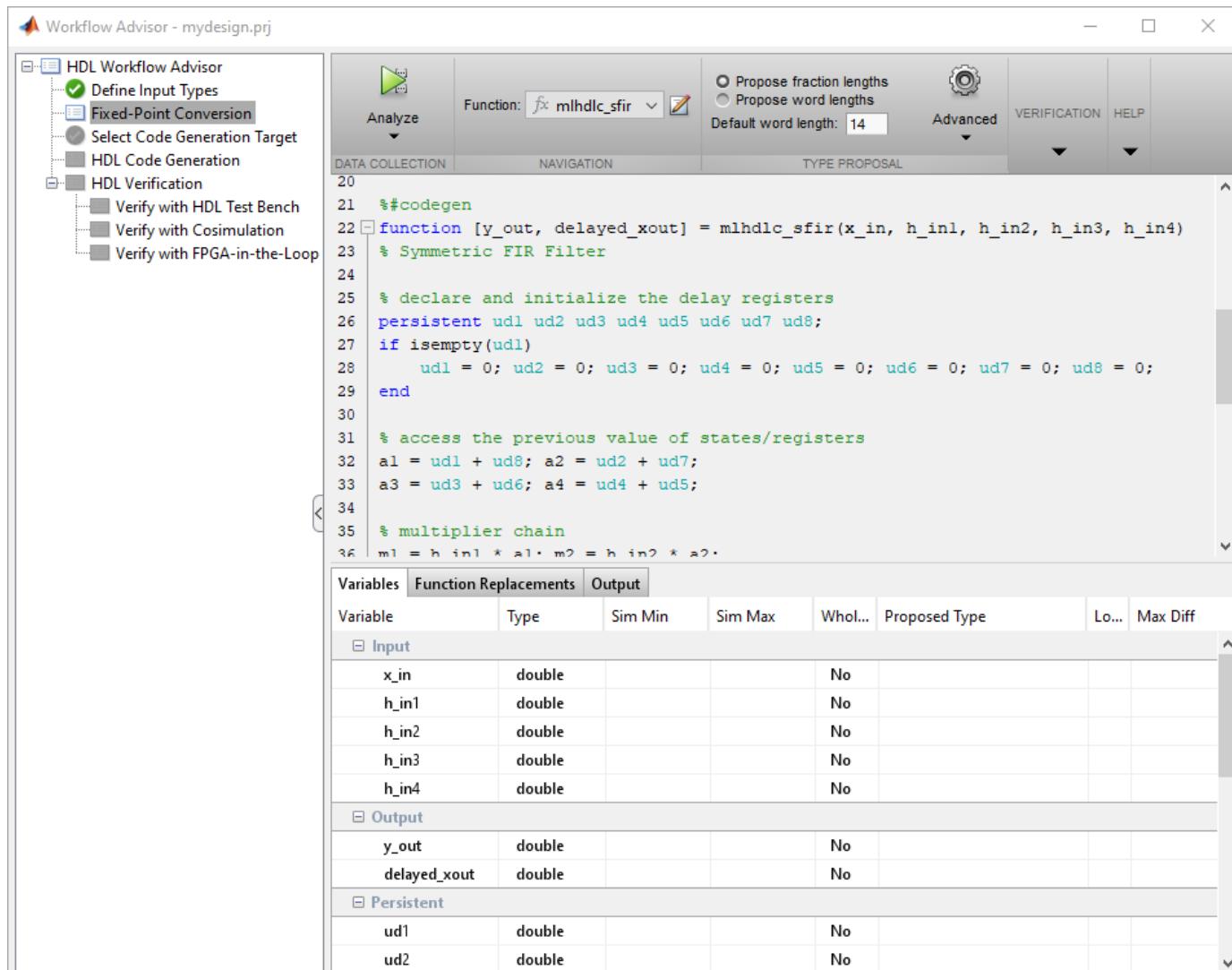


# HDL Workflow Advisor Reference

---

- “HDL Workflow Advisor” on page 9-2
- “MATLAB to HDL Code and Synthesis” on page 9-6

# HDL Workflow Advisor



## Overview

The HDL Workflow Advisor is a tool that supports a suite of tasks covering the stages of the ASIC and FPGA design process, including converting floating-point MATLAB algorithms to fixed-point algorithms. Some tasks perform code validation or checking; others run the HDL code generator or third-party tools. Each folder at the top level of the HDL Workflow Advisor contains a group of related tasks that you can select and run.

Use the HDL Workflow Advisor to:

- Convert floating-point MATLAB algorithms to fixed-point algorithms.

If you already have a fixed-point MATLAB algorithm, set **Design needs conversion to Fixed Point?** to No to skip this step.

- Generate HDL code from fixed-point MATLAB algorithms.
- Simulate the HDL code using a third-party simulation tool.
- Synthesize the HDL code and run a mapping process that maps the synthesized logic design to the target FPGA.
- Run a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.

## Procedures

### Automatically Run Tasks

To automatically run the tasks within a folder:

- 1 Click the **Run** button. The tasks run in order until a task fails.

Alternatively, right-click the folder to open the context menu. From the context menu, select Run to run the tasks within the folder.

- 2 If a task in the folder fails:
  - a Fix the failure using the information in the results pane.
  - b Continue the run by clicking the **Run** button.

### Run Individual Tasks

To run an individual task:

- 1 Click the **Run** button.

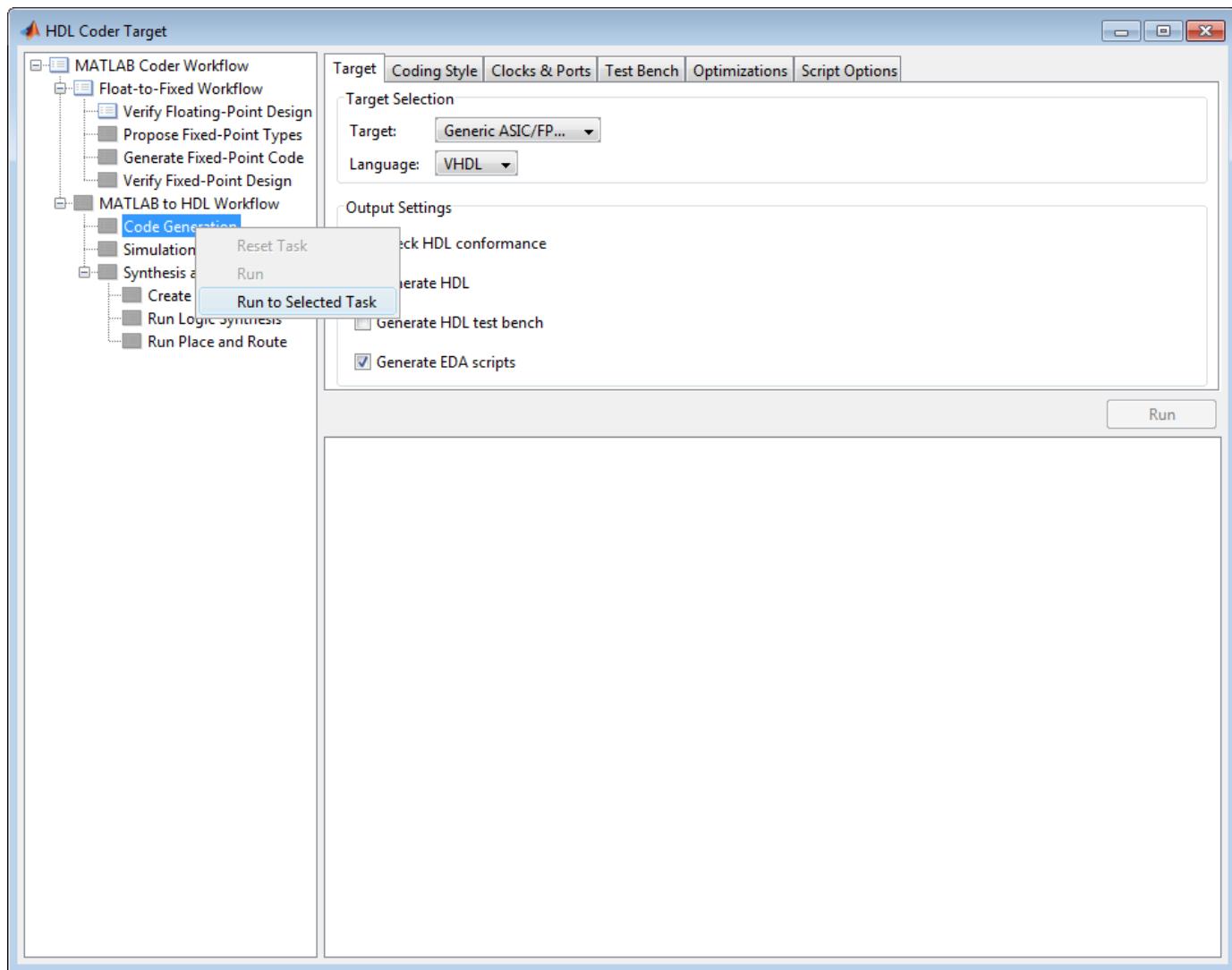
Alternatively, right-click the task to open the context menu. From the context menu, select Run to run the selected task.

- 2 Review Results. The possible results are:  
**Pass:** Move on to the next task.  
**Warning:** Review results, decide whether to move on or fix.  
**Fail:** Review results, do not move on without fixing.
- 3 If required, fix the issue using the information in the results pane.
- 4 Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run**.

### Run to Selected Task

To run the tasks up to and including the currently selected task:

- 1 Select the last task that you want to run.
- 2 Right-click this task to open the context menu.
- 3 From the context menu, select Run to Selected Task.



**Note** If a task before the selected task fails, the Workflow Advisor stops at the failed task.

### Reset a Task

To reset a task:

- 1 Select the task that you want to reset.
- 2 Right-click this task to open the context menu.
- 3 From the context menu, select **Reset Task** to reset this and subsequent tasks.

### Reset All Tasks in a Folder

To reset a task:

- 1 Select the folder that you want to reset.
- 2 Right-click this folder to open the context menu.

- 3** From the context menu, select **Reset Task** to reset the tasks this folder and subsequent folders.

# MATLAB to HDL Code and Synthesis

## In this section...

- “MATLAB to HDL Code Conversion” on page 9-6
- “Code Generation: Target Tab” on page 9-6
- “Code Generation: Coding Style Tab” on page 9-7
- “Code Generation: Clocks and Ports Tab” on page 9-8
- “Code Generation: Test Bench Tab” on page 9-10
- “Code Generation: Optimizations Tab” on page 9-11
- “Simulation and Verification” on page 9-12
- “Synthesis and Analysis” on page 9-13

## MATLAB to HDL Code Conversion

The **MATLAB to HDL Workflow** task in the HDL Workflow Advisor generates HDL code from fixed-point MATLAB code, and simulates and verifies the HDL against the fixed-point algorithm. HDL Coder then runs synthesis, and optionally runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

## Code Generation: Target Tab

Select target hardware and language and required outputs.

### Input Parameters

#### Target

Target hardware. Select from the list:

- Generic ASIC/FPGA
- Xilinx
- Altera
- Simulation

#### Language

Select the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language.

**Default:** VHDL

#### Check HDL Conformance

Enable HDL conformance checking.

**Default:** Off

#### Generate HDL

Enable generation of HDL code for the fixed-point MATLAB algorithm.

**Default:** On

#### Generate HDL Test Bench

Enable generation of HDL code for the fixed-point test bench.

**Default:** Off**Generate EDA Scripts**

Enable generation of script files for third-party electronic design automation (EDA) tools. These scripts let you compile and simulate generated HDL code and synthesize generated HDL code.

**Default:** On**Code Generation: Coding Style Tab**

Parameters that affect the style of the generated code.

**Input Parameters****Preserve MATLAB code comments**

Include MATLAB code comments in generated code.

**Default:** On**Include MATLAB source code as comments**

Include MATLAB source code as comments in the generated code. The comments precede the associated generated code. Includes the function signature in the function banner.

**Default:** On**Generate Report**

Enable a code generation report.

**Default:** Off**VHDL File Extension**

Specify the file name extension for generated VHDL files.

**Default:** .vhd**Verilog File Extension**

Specify the file name extension for generated Verilog files.

**Default:** .v**Comment in header**

Specify comment lines in header of generated HDL and test bench files.

**Default:** None

Text entered in this field as a character vector generates a comment line in the header of the generated code. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included in the text, the code generator emits single-line comments for each newline.

**Package postfix**

HDL Coder applies this option only if a package file is required for the design.

**Default:** \_pkg

**Entity conflict postfix**

Specify the character vector to resolve duplicate VHDL entity or Verilog module names in generated code.

**Default:** \_block

**Reserved word postfix**

Specify a character vector to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

**Default:** \_rsvd

**Clocked process postfix**

Specify a character vector to append to HDL clock process names.

Default: \_process

**Complex real part postfix**

Specify a character vector to append to real part of complex signal names.

**Default:** '\_re'

**Complex imaginary part postfix**

Specify a character vector to append to imaginary part of complex signal names.

**Default:** '\_im'

**Pipeline postfix**

Specify a character vector to append to names of input or output pipeline registers.

**Default:** '\_pipe'

**Enable prefix**

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

**Default:** 'enb'

**Code Generation: Clocks and Ports Tab**

Clock and port settings

**Input Parameters****Reset type**

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers.

**Default:** Asynchronous

**Reset Asserted level**

Specify whether the asserted (active) level of reset input signal is active-high or active-low.

**Default:** Active-high

**Reset input port**

Enter the name for the reset input port in generated HDL code.

**Default:** reset

**Clock input port**

Specify the name for the clock input port in generated HDL code.

**Default:** clk

**Clock enable input port**

Specify the name for the clock enable input port in generated HDL code.

**Default:** clk

**Oversampling factor**

Specify frequency of global oversampling clock as a multiple of the design under test (DUT) base rate (1).

**Default:** 1

**Input data type**

Specify the HDL data type for input ports.

For VHDL, the options are:

- std\_logic\_vector
  - Specifies VHDL type STD\_LOGIC\_VECTOR
- signed/unsigned
  - Specifies VHDL type SIGNED or UNSIGNED

**Default:** std\_logic\_vector

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is 'wire'. Therefore, **Input data type** is disabled when the target language is Verilog.

**Default:** wire

**Output data type**

Specify the HDL data type for output data types.

For VHDL, the options are:

- Same as input data type
  - Specifies that output ports have the same type specified by Input data type.
- std\_logic\_vector
  - Specifies VHDL type STD\_LOGIC\_VECTOR
- signed/unsigned
  - Specifies VHDL type SIGNED or UNSIGNED

**Default:** Same as input data type

For Verilog, the options are:

- In generated Verilog code, the data type for all ports is ‘wire’. Therefore, Output data type is disabled when the target language is Verilog.

**Default:** wire

#### Clock enable output port

Specify the name for the clock enable input port in generated HDL code.

**Default:** clk\_enable

### Code Generation: Test Bench Tab

Test bench settings.

#### Input Parameters

##### Test bench name postfix

Specify a character vector appended to names of reference signals generated in test bench code.

**Default:** '\_tb'

##### Force clock

Specify whether the test bench forces clock enable input signals.

**Default:** On

##### Clock High time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals high (1).

**Default:** 5

##### Clock low time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals low (0).

**Default:** 5

##### Hold time (ns)

Specify a hold time, in nanoseconds, for input signals and forced reset input signals.

**Default:** 2 (given the default clock period of 10 ns)

##### Setup time (ns)

Display setup time for data input signals.

**Default:** 0

##### Force clock enable

Specify whether the test bench forces clock enable input signals.

**Default:** On

**Clock enable delay (in clock cycles)**

Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable.

**Default:** 1

**Force reset**

Specify whether the test bench forces reset input signals.

**Default:** On

**Reset length (in clock cycles)**

Define length of time (in clock cycles) during which reset is asserted.

**Default:** 2

**Hold input data between samples**

Specify how long substrate signal values are held in valid state.

**Default:** On

**Initialize testbench inputs**

Specify initial value driven on test bench inputs before data is asserted to device under test (DUT).

**Default:** Off

**Multi file testbench**

Divide generated test bench into helper functions, data, and HDL test bench code files.

**Default:** Off

**Test bench data file name postfix**

Specify suffix added to test bench data file name when generating multi-file test bench.

**Default:** '\_data'

**Test bench reference post fix**

Specify a character vector to append to names of reference signals generated in test bench code.

**Default:** '\_ref'

**Ignore data checking (number of samples)**

Specify number of samples during which output data checking is suppressed.

**Default:** 0

**Use fiaccel to accelerate test bench logging**

To generate a test bench, HDL Coder simulates the original MATLAB code. Use the Fixed-Point Designer `fiaccel` function to accelerate this simulation and accelerate test bench logging.

**Default:** On

**Code Generation: Optimizations Tab**

Optimization settings

## Input Parameters

### Map persistent array variables to RAMs

Select to map persistent array variables to RAMs instead of mapping to shift registers.

**Default:** Off

Dependencies:

- **RAM Mapping Threshold**
- **Persistent variable names for RAM Mapping**

### RAM Mapping Threshold

Specify the minimum RAM size required for mapping persistent array variables to RAMs.

**Default:** 256

### Persistent variable names for RAM Mapping

Provide the names of the persistent variables to map to RAMs.

**Default:** None

## Input Pipelining

Specify number of pipeline registers to insert at top level input ports. Can improve performance and help to meet timing constraints.

**Default:** 0

## Output Pipelining

Specify number of pipeline registers to insert at top level output ports. Can improve performance and help to meet timing constraints.

**Default:** 0

## Distribute Pipeline Registers

Reduces critical path by changing placement of registers in design. Operates on all registers, including those inserted using the **Input Pipelining** and **Output Pipelining** parameters, and internal design registers.

**Default:** Off

## Sharing Factor

Number of additional sources that can share a single resource, such as a multiplier. To share resources, set **Sharing Factor** to 2 or higher; a value of 0 or 1 turns off sharing.

In a design that performs identical multiplication operations, HDL Coder can reduce the number of multipliers by the sharing factor. This can significantly reduce area.

**Default:** 0

## Simulation and Verification

Simulates the generated HDL code using the selected simulation tool.

## Input Parameters

**Simulation tool**

Lists the available simulation tools.

**Default:** None

### Skip this step

**Default:** Off

## Results and Recommended Actions

Conditions	Recommended Action
No simulation tool available on system path.	Add your simulation tool path to the MATLAB system path, then restart MATLAB. For more information, see “Synthesis Tool Path Setup”.

## Synthesis and Analysis

This folder contains tasks to create a synthesis project for the HDL code. The task then runs the synthesis and, optionally, runs place and route to generate a circuit description suitable for programming an ASIC or FPGA.

## Input Parameters

### Skip this step

**Default:** Off

Skip this step if you are interested only in simulation or you do not have a synthesis tool.

### Create Project

Create synthesis project for supported synthesis tool.

#### Description

This task creates a synthesis project for the selected synthesis tool and loads the project with the HDL code generated for your MATLAB algorithm.

You can select the family, device, package, and speed that you want.

When the project creation is complete, the HDL Workflow Advisor displays a link to the project in the right pane. Click this link to view the project in the synthesis tool's project window.

## Input Parameters

### Synthesis Tool

Select from the list:

- Altera Quartus II

Generate a synthesis project for Altera Quartus II. When you select this option, HDL Coder sets:

- **Chip Family** to Stratix II
- **Device Name** to EP2S60F1020C4

You can manually change these settings.

- Xilinx ISE

Generate a synthesis project for Xilinx ISE. When you select this option, HDL Coder:

- Sets **Chip Family** to Virtex4
- Sets **Device Name** to xc4vsx35
- Sets **Package Name** to ff6...
- Sets **Speed Value** to -...

You can manually change these settings.

#### **Default:** No Synthesis Tool Specified

When you select **No Synthesis Tool Specified**, HDL Coder does not generate a synthesis project. It clears and disables the fields in the **Synthesis Tool Selection** pane.

#### **Chip Family**

Target device family.

**Default:** None

#### **Device Name**

Specific target device, within selected family.

**Default:** None

#### **Package Name**

Available package choices. The family and device determine these choices.

**Default:** None

#### **Speed Value**

Available speed choices. The family, device, and package determine these choices.

**Default:** None

#### **Results and Recommended Actions**

Conditions	Recommended Action
Synthesis tool fails to create project.	Read the error message returned by synthesis tool, then check the synthesis tool version, and check that you have write permission for the project folder.
Synthesis tool does not appear in dropdown list.	Add your synthesis tool path to the MATLAB system path, then restart MATLAB. For more information, see "Synthesis Tool Path Setup".

#### **Run Logic Synthesis**

Launch selected synthesis tool and synthesize the generated HDL code.

## Description

This task:

- Launches the synthesis tool in the background.
- Opens the previously generated synthesis project, compiles HDL code, synthesizes the design, and emits netlists and related files.
- Displays a synthesis log in the **Result** subpane.

## Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.

## Run Place and Route

Launches the synthesis tool in the background and runs a Place and Route process.

## Description

This task:

- Launches the synthesis tool in the background.
- Runs a Place and Route process that takes the circuit description produced by the previous mapping process, and emits a circuit description suitable for programming an FPGA.
- Displays a log in the Result subpane.

## Input Parameters

### Skip this step

If you select **Skip this step**, the HDL Workflow Advisor executes the workflow, but omits the Perform Place and Route, marking it Passed. You might want to select **Skip this step** if you prefer to do place and route work manually.

**Default:** Off

## Results and Recommended Actions

Conditions	Recommended Action
Synthesis tool fails when running place and route.	Read the error message returned by the synthesis tool, modify the MATLAB code, then rerun from the beginning of the HDL Coder workflow.



# HDL Code Generation from Simulink



# Model Design for HDL Code Generation

---

- “Signal and Data Type Support” on page 10-2
- “Use Simulink Templates for HDL Code Generation” on page 10-7
- “Generate DUT Ports for Tunable Parameters” on page 10-17
- “Generate Parameterized Code for Referenced Models” on page 10-20
- “Generating HDL Code for Subsystems with Array of Buses” on page 10-21
- “Implement Control Signals Based Mathematical Functions using HDL Coder” on page 10-24
- “Using ForEach Subsystems in HDL Coder” on page 10-47
- “Generate HDL Code for Blocks Inside For Each Subsystem” on page 10-51
- “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 10-55
- “Model and Debug Test Point Signals with HDL Coder” on page 10-59
- “Allocate Sufficient Delays for Floating-Point Operations” on page 10-67
- “Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials” on page 10-72
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Numeric Considerations with Native Floating-Point” on page 10-84
- “ULP Considerations of Native Floating-Point Operators” on page 10-88
- “Latency Values of Floating Point Operators” on page 10-91
- “Latency Considerations with Native Floating Point” on page 10-96
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109
- “Verify the Generated Code from Native Floating-Point” on page 10-116
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120
- “Supported Data Types and Scope” on page 10-125
- “Import Verilog Code and Generate Simulink Model” on page 10-127
- “Supported Verilog Constructs for HDL Import” on page 10-130
- “Verilog Dataflow Modeling with HDL Import” on page 10-135
- “Simulate and Generate HDL Code for the Float Typecast Block” on page 10-146
- “Generate Simulink® Model From CORDIC Atan2 Verilog® Code” on page 10-148

# Signal and Data Type Support

## In this section...

- “Buses” on page 10-2
- “Enumerations” on page 10-3
- “Matrices” on page 10-3
- “Unsupported Signal and Data Types” on page 10-6

HDL Coder supports code generation for Simulink signal types and data types with a few special cases.

## Buses

If your DUT or other blocks in your model have many input or output signals, you can create bus signals to improve the readability of your model. A bus signal or bus is a composite signal that consists of other signals that are called elements.

You can generate HDL code for designs that use virtual and nonvirtual buses. For example, you can generate code for designs that contain:

- DUT subsystem ports connected to buses.
- Simulink and Stateflow® blocks that support buses and HDL code generation.

## Supported Blocks with Buses

Bus-capable blocks are blocks that can accept bus signals as input and produce bus signals as outputs. For a list of bus-capable blocks that Simulink supports, see “Bus-Capable Blocks”. HDL Coder supports code generation for bus-capable blocks in the **HDL Coder** block library. For more details, see the “HDL Code Generation” section of each block page. The supported blocks include:

- Bus Assignment
- Bus Creator
- Bus Selector
- Constant
- Delay
- Multiport Switch
- Rate Transition
- Signal Conversion
- Signal Specification
- Switch
- Unit Delay
- Vector Concatenate
- Zero-Order Hold

In addition, subsystems, models, and these user-defined functions support buses for simulation and HDL code generation:

- Subsystem
- Model references, see “Model Referencing for HDL Code Generation” on page 27-2.
- Stateflow Chart
- MATLAB Function blocks
- MATLAB System blocks
- Vision HDL Toolbox blocks that accept a `pixelcontrol` bus for control input

### Bus Support Limitations

Buses are not supported in the IP Core Generation workflow. In addition, you cannot generate code for designs that use:

- A Black box model reference connected to a bus.
- A bus input to a Delay block with nonzero **Initial condition**.

### Enumerations

You can generate code for Simulink, MATLAB, or Stateflow enumerations within your design.

### Requirements

- The enumeration values must be monotonically increasing.
- The enumeration strings must have unique names and must not use a reserved keyword in the Verilog or VHDL language.
- If your target language is Verilog, all enumeration member names must be unique within the design.

### Restrictions

Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods:

- IP Core Generation workflow
- FPGA Turnkey workflow
- Simulink Real-Time FPGA I/O workflow
- Customization for the USRP Device workflow
- FPGA-in-the-loop
- HDL Cosimulation

### Matrices

You can use matrix types with these blocks in your design. For more details, see the “HDL Code Generation” section of each block page.

HDL Coder Block Library	Supported blocks
Discontinuities	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Dead Zone</li> <li>• Relay</li> <li>• Saturation</li> </ul>
Discrete	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Delay</li> <li>• Discrete-Time Integrator</li> <li>• Memory</li> <li>• Tapped Delay</li> <li>• Unit Delay</li> <li>• Unit Delay Enabled Synchronous</li> <li>• Unit Delay Enabled Resettable Synchronous</li> <li>• Unit Delay Resettable Synchronous</li> <li>• Zero-Order Hold</li> </ul>
HDL Floating Point Operations	The Rounding Function block is supported.
HDL Operations	All blocks in this library are supported.
HDL RAMs	Blocks in this library are not supported.
HDL Subsystems	Blocks in this library are not supported.
Logic and Bit Operations	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Bit Clear</li> <li>• Bit Concat</li> <li>• Bit Reduce</li> <li>• Bit Rotate</li> <li>• Bit Set</li> <li>• Bit Shift</li> <li>• Bit Slice</li> <li>• Extract Bits</li> <li>• Logical Operator</li> <li>• Relational Operator</li> <li>• Shift Arithmetic</li> </ul>
Lookup Tables	Blocks in this library are not supported.

HDL Coder Block Library	Supported blocks
Math Operations	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Abs</li> <li>• Add</li> <li>• Assignment</li> <li>• Bias</li> <li>• Complex to Real-Imag</li> <li>• Gain</li> <li>• Product, including matrix multiplication.</li> <li>• Matrix Concatenate</li> <li>• Math Function</li> <li>• Real-Imag to Complex</li> <li>• Reshape</li> <li>• Sign</li> <li>• Sum</li> <li>• Unary Minus</li> <li>• Increment Stored Integer</li> <li>• Increment Real World</li> <li>• Decrement Stored Integer</li> <li>• Decrement Real World</li> <li>• Sum of Elements</li> <li>• Product of Elements</li> </ul>
Model Verification	All blocks in this library are supported.
Model-Wide Utilities	The DocBlock is supported. The Model Info block does not support matrix data types.
Ports & Subsystems	The Subsystem block is supported.
Signal Attributes	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Data Type Conversion</li> <li>• Data Type Duplicate</li> <li>• Probe</li> <li>• Rate Transition</li> <li>• Signal Conversion</li> <li>• Signal Specification</li> </ul>
Signal Routing	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>• Mux</li> <li>• Multiport Switch</li> <li>• Selector</li> <li>• Switch</li> </ul>

HDL Coder Block Library	Supported blocks
Sources	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>Constant</li> <li>Enumerated Constant</li> <li>Import</li> </ul>
Sinks	<p>These blocks are supported:</p> <ul style="list-style-type: none"> <li>Display</li> <li>Outport</li> <li>Scope</li> <li>To Workspace</li> <li>To File</li> <li>XY Graph</li> </ul>
User-Defined Functions	The MATLAB Function block is supported.

The code generator does not support matrix types at the interfaces of the Subsystem that you generate HDL code for. Use a Reshape block to convert the matrix input to a 1-D array at the interface. Inside the Subsystem, use another Reshape block that converts the 1-D array back to the matrix type with the dimensionality that you specified.

## Unsupported Signal and Data Types

- Arrays stored in row-major layout are not supported for HDL code generation
- Variable-size signals are not supported for code generation.

## See Also

### Related Examples

- “Generating HDL Code for Subsystems with Array of Buses” on page 10-21

### More About

- “Signal Types”
- “About Data Types in Simulink”
- “Composite Signals”
- “Use Enumerated Data in Simulink Models”
- “Enumerated Data” (Stateflow)

# Use Simulink Templates for HDL Code Generation

## In this section...

["Create Model Using HDL Coder Model Template" on page 10-7](#)

["HDL Coder Model Templates" on page 10-7](#)

HDL Coder model templates in Simulink provide you with design patterns and best practices for models intended for HDL code generation. Models you create from one of the HDL Coder model templates have their configuration parameters and solver settings set up for HDL code generation. To configure an existing model for HDL code generation, use `hdlsetup`.

## Create Model Using HDL Coder Model Template

To model hardware for efficient HDL code generation, create a model using an HDL Coder model template.

- 1 Open the Simulink Start Page. In the MATLAB Home tab, select the **Simulink** button. Alternatively, at the command line, enter:

```
simulink
```

- 2 In the **HDL Coder** section, you see templates that are preconfigured for HDL code generation. Selecting the template opens a blank model in the Simulink Editor. To save the model, select **File > Save As**.
- 3 To open the Simulink Library Browser and then open the **HDL Coder** Block Library, select the **Library Browser** button in the Simulink Editor. Alternatively, at the command line, enter

```
slLibraryBrowser
```

To filter the Simulink Library Browser to show the block libraries that support HDL code generation, use the `hdlllib` function:

```
hdlllib
```

## HDL Coder Model Templates

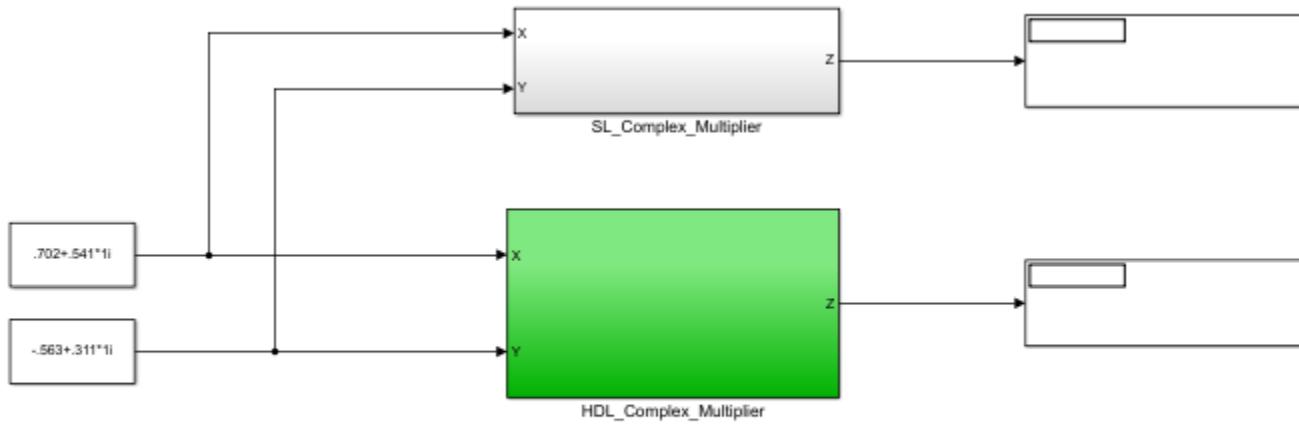
### Complex Multiplier

The Complex Multiplier template shows how to model a complex multiplier-accumulator and manually pipeline the intermediate stages. The hardware implementation of complex multiplication uses four multipliers and two adders.

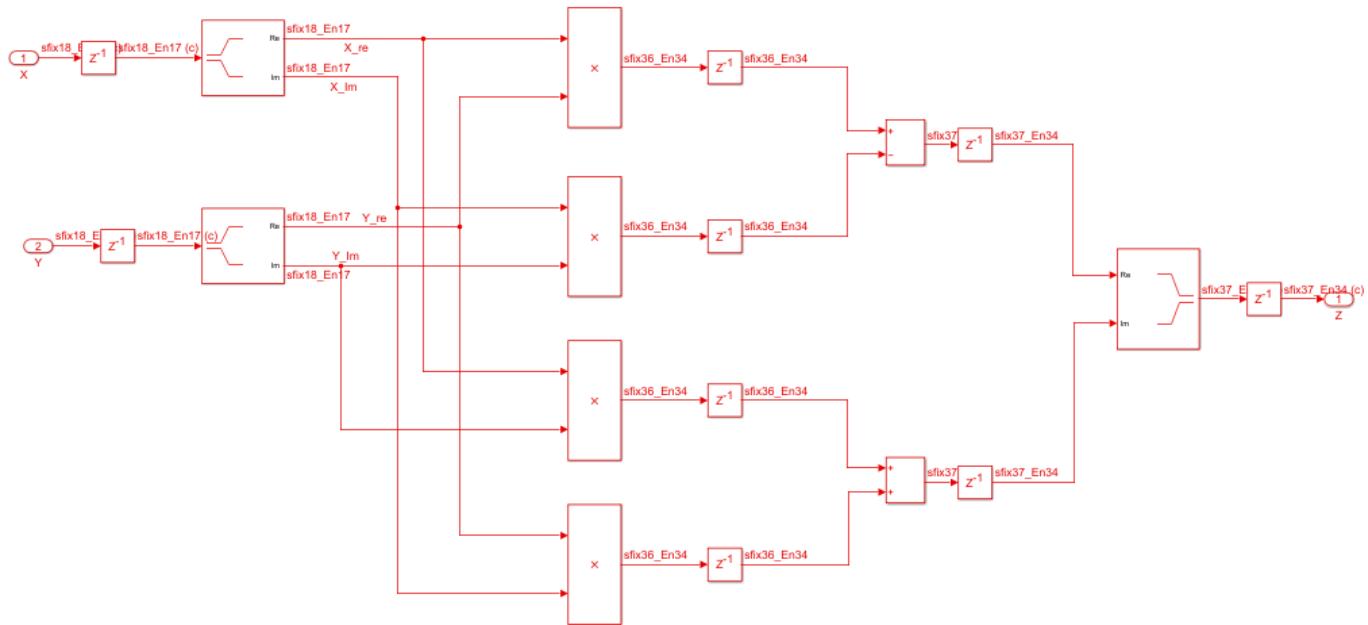
The template applies the following best practices:

- In the Configuration Parameters dialog box, in **HDL Code Generation > Global Settings**, **Reset type** is set to **Synchronous**.
- To improve speed, Delay blocks, which map to registers in hardware, are at the inputs and outputs of the multipliers and adders.
- To support the output data of a full-precision complex multiplier, the output data word length is manually specified to be  $(\text{operand\_word\_length} * 2) + 1$ .

For example, in the template, the operand word length is 18, and the output word length is 37.

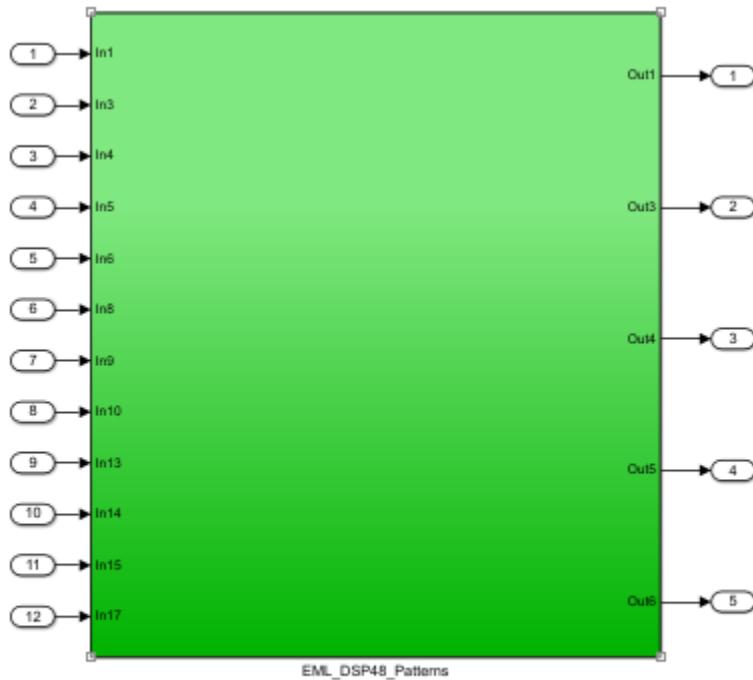


Copyright 2014-2016 The MathWorks, Inc.

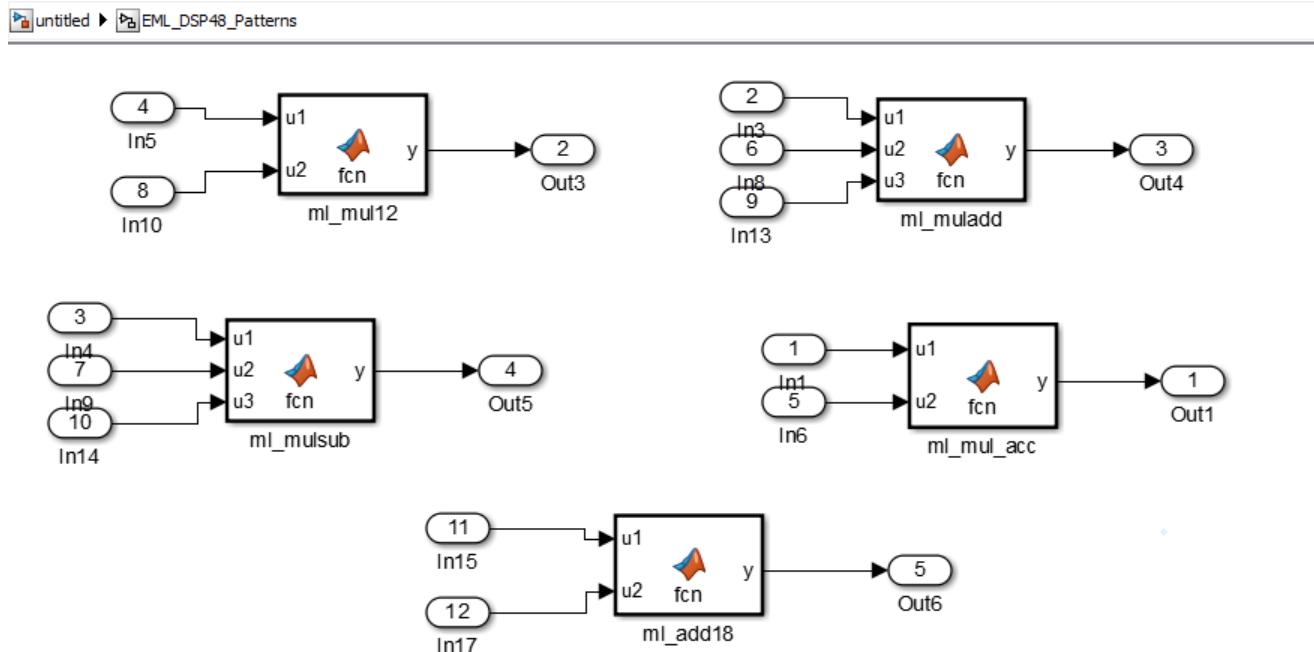


### MATLAB Arithmetic

The MATLAB Arithmetic template contains MATLAB arithmetic operations that infer DSP48s in hardware.



Copyright 2014-2016 The MathWorks, Inc.



For example, the `ml_mul_acc` MATLAB Function block shows how to write a multiply-accumulate operation in MATLAB. [hdlfimath](#) on page 29-29 applies fixed-point math settings for HDL code generation.

```

function y = fcn(u1, u2)

% design of a 6x6 multiplier
% same reset on inputs and outputs
% followed by an adder

nt = numerictype(0,6,0);
nt2 = numerictype(0,12,0);
fm = hdlfimath;

persistent u1_reg u2_reg mul_reg add_reg;
if isempty(u1_reg)
    u1_reg = fi(0, nt, fm);
    u2_reg = fi(0, nt, fm);
    mul_reg = fi(0, nt2, fm);
    add_reg = fi(0, nt2, fm);
end

mul = mul_reg;
mul_reg = u1_reg * u2_reg;
add = add_reg;
add_reg(:) = mul+add;
u1_reg = u1;
u2_reg = u2;

y = add;

```

## ROM

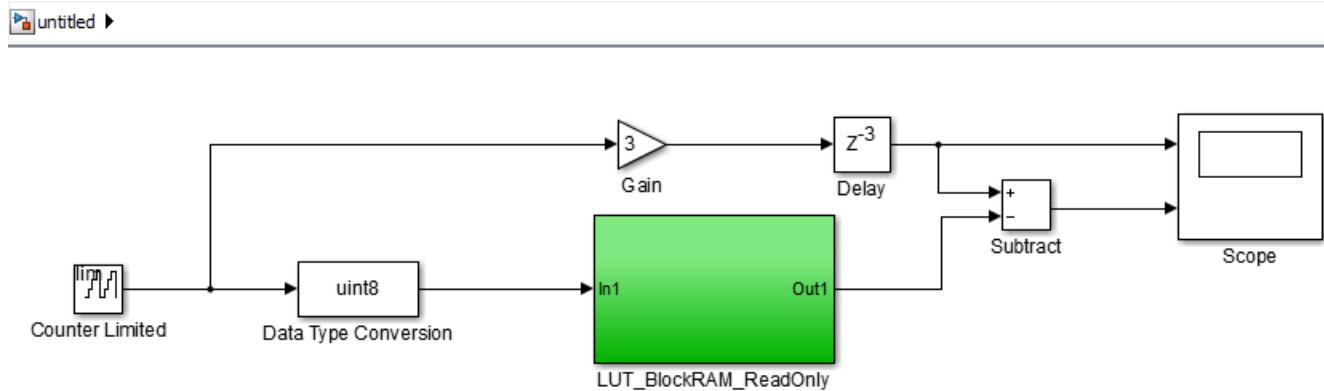
The ROM template is a design pattern that maps to a ROM in hardware.

The template applies the following best practices:

- At the output of the lookup table, there is a Delay block with `ResetType = none`.
- The lookup table is structured such that the spacing between breakpoints is a power of two.

Using table dimensions that are a power of two enables HDL Coder to generate shift operations instead of division operations. If necessary, pad the table with zeros.

- The number of lookup table entries is a power of two. For some synthesis tools, a lookup table that has a power-of-two number of entries maps better to ROM. If necessary, pad the table with zeros.





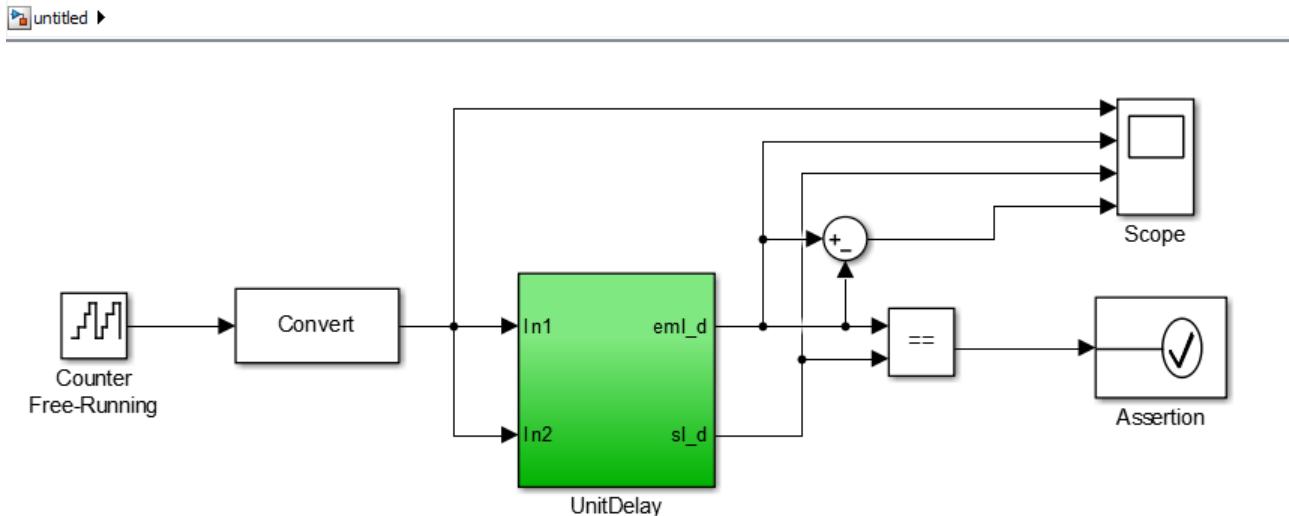
```
x=(0:99)';
Scale_by_3_LUT=3*x;
pad=2^nextpow2(length(Scale_by_3_LUT))-length(Scale_by_3_LUT);
Scale_by_3_LUT_pad=[Scale_by_3_LUT;zeros(pad,1)];
```

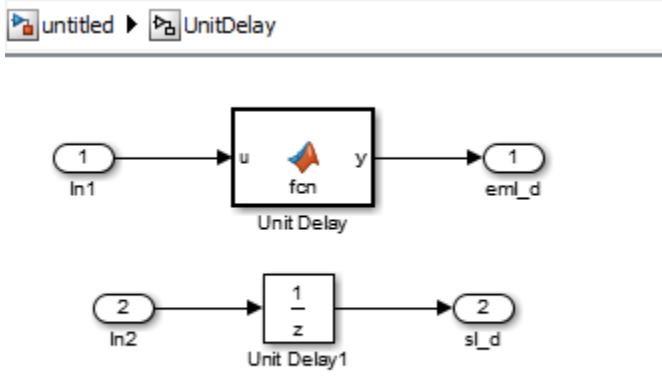
## Register

The Register template shows how to model hardware registers:

- In Simulink, using the Delay block.
- In MATLAB, using persistent variables.

This design pattern also shows how to use `cast` to propagate data types automatically.





The MATLAB code in the MATLAB Function block uses a persistent variable to model the register.

```
function y = fcn(u)
% Unit delay implementation that maps to a register in hardware

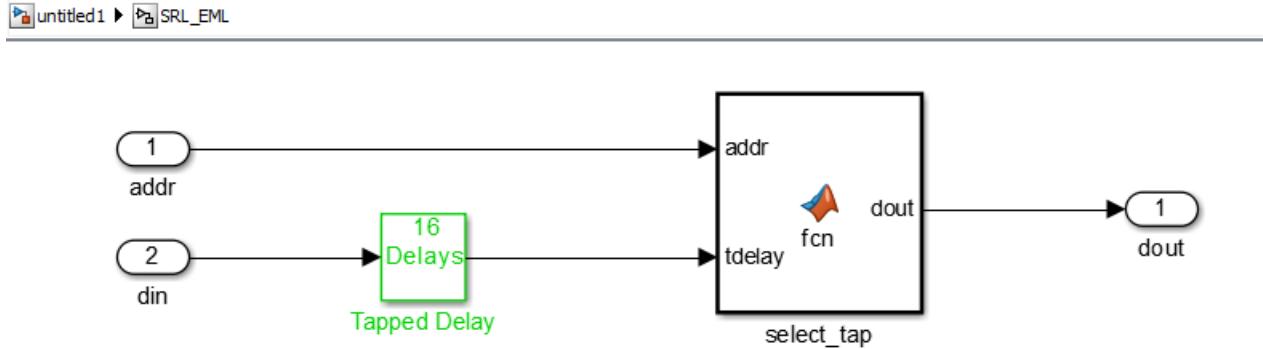
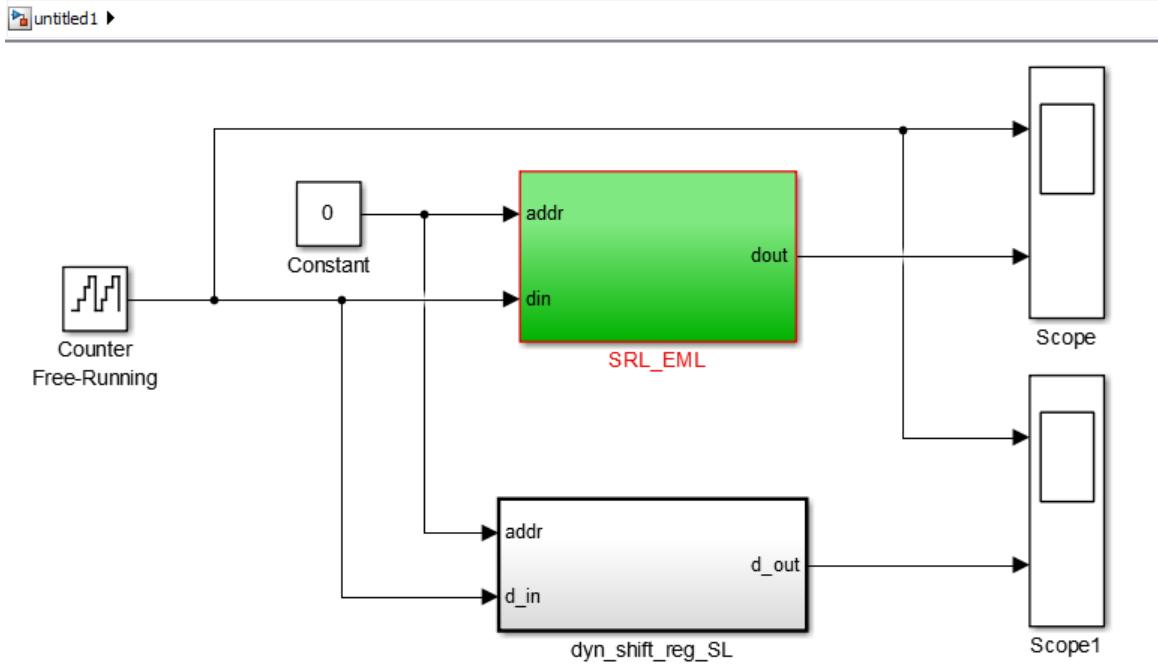
persistent u_d;
if isempty(u_d)
    % defines initial value driven by unit delay at time step 0
    u_d = cast(0, 'like', u);
end

% return delayed input from last sample time hit
y = u_d;

% store the current input
u_d = u;
```

### SRL

The SRL template shows how to implement a shift register that maps to an SRL16 in hardware. You can use a similar pattern to map to an SRL32.



To map to SRL16/32:

- Set **ResetType** = **none** for the tapped delay
- Use **ML fcn** block to create mux logic
- Flatten hierarchy for the subsystem to inline the ML code
- Do not use "include current input in output vector" option for the tapped delay

In the shift register subsystem, the Tapped Delay implements the shift operation, and the MATLAB Function, **select\_tap**, implements the output mux.

In **select\_tap**, the zero-based address, **addr** increments by 1 because MATLAB indices are one-based.

```
function dout = fcn(addr, tdelay)
 %#codegen

addr1 = fi(addr+1,0,5,0);
dout = tdelay(addr1);
```

In the generated code, HDL Coder automatically omits the increment because Verilog and VHDL are zero-based.

The template also applies the following best practices for mapping to an SRL16 in hardware:

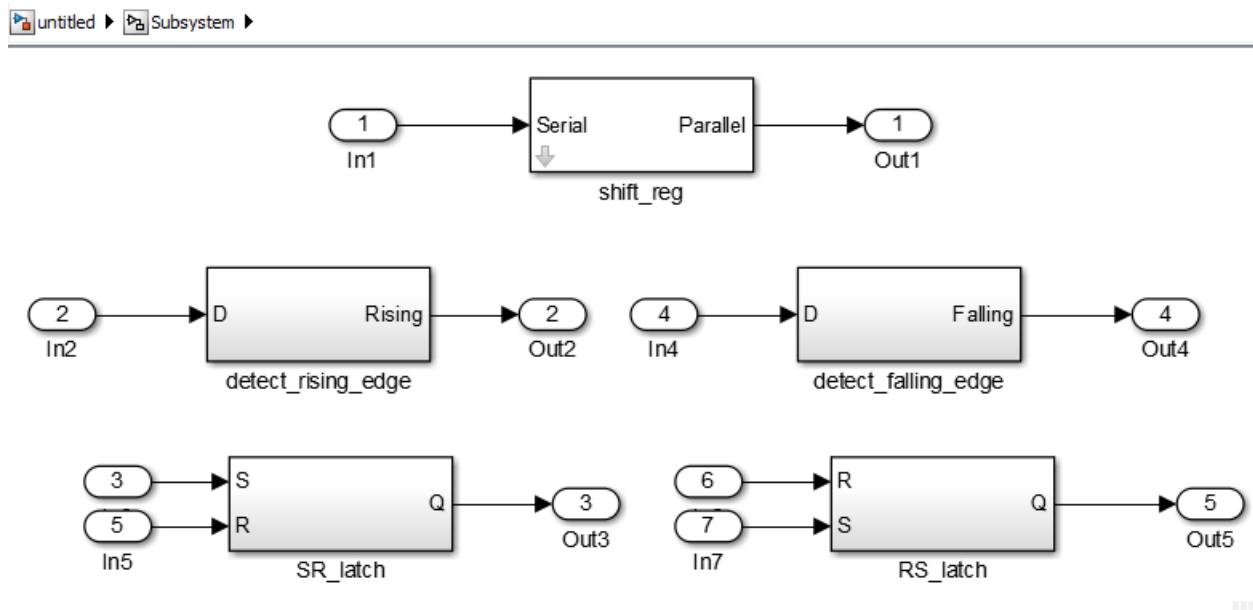
- For the Tapped Delay block:
  - In the Block Parameters dialog box, **Include current input in output vector** is not enabled.
  - In the HDL Block Properties dialog box, **ResetType** is set to none.
- For the Subsystem block, in the HDL Block Properties dialog box, **FlattenHierarchy** is set to on.

### Simulink Hardware Patterns

The Simulink Hardware Patterns template contains design patterns for common hardware operations:

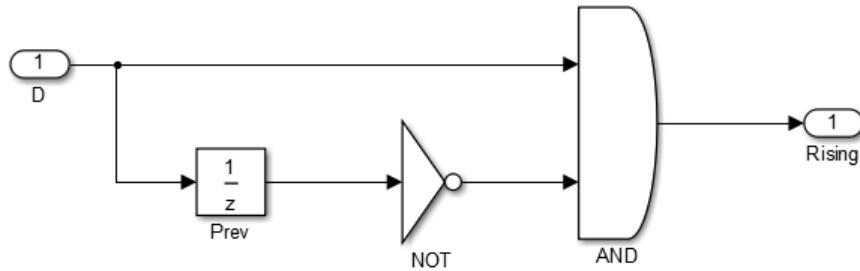
- Serial-to-parallel shift register
- Detect rising edge
- Detect falling edge
- SR latch
- RS latch



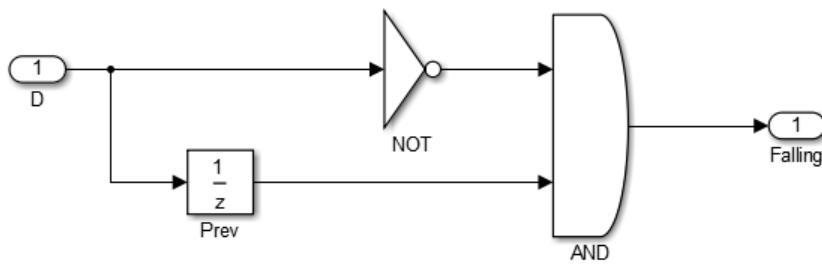


For example, the design patterns for rising edge detection and falling edge detection:

untitled > Subsystem > detect\_rising\_edge

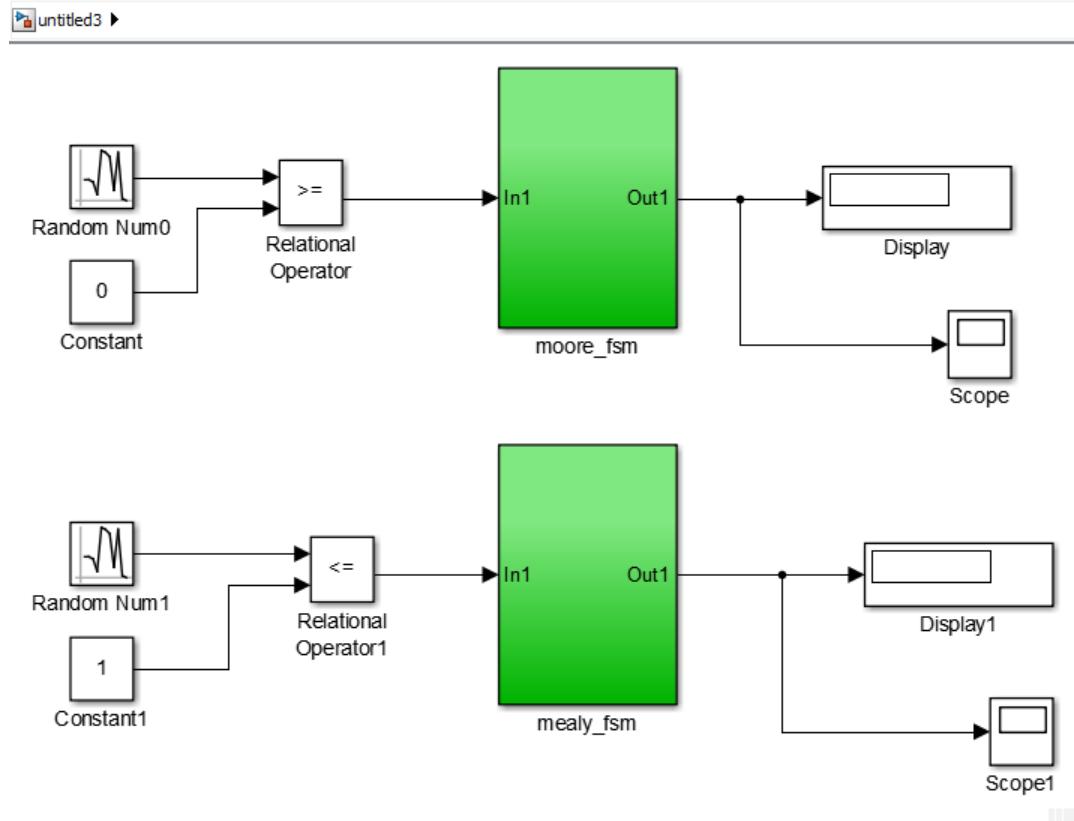


untitled > Subsystem > detect\_falling\_edge



## State Machine in MATLAB

The State Machine in MATLAB template shows how to implement Mealy and Moore state machines using the MATLAB Function block.



To learn more about best practices for modeling state machines, see “Model a State Machine for HDL Code Generation” on page 3-4.

## See Also

`hdlsetup` | `makehdl`

## More About

- “Create HDL-Compatible Simulink Model”
- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “Hardware Modeling with MATLAB Code”

# Generate DUT Ports for Tunable Parameters

## In this section...

- “Prerequisites” on page 10-17
- “Create and Add Tunable Parameter That Maps to DUT Ports” on page 10-17
- “Generated Code” on page 10-18
- “Limitations” on page 10-18
- “Use Tunable Parameter in Other Blocks” on page 10-18

Tunable parameters that you use to adjust your model behavior during simulation can map to top-level DUT ports in your generated HDL code. HDL Coder generates one DUT port per tunable parameter.

You can generate a DUT port for a tunable parameter by using it in one of these blocks:

- Gain
- Constant
- MATLAB Function
- MATLAB System
- Chart
- Truth Table
- State Transition Table

These blocks with the tunable parameter can be at any level of the DUT hierarchy, including within a model reference.

You cannot use HDL cosimulation with a DUT that uses tunable parameters in any of these blocks. If you use a tunable parameter in a block other than these blocks, code is generated inline and does not map to DUT ports. To use the value of a tunable parameter in a Chart or Truth Table block, see “Use Tunable Parameter in Other Blocks” on page 10-18.

You can define and store the tunable parameters in the base workspace or a Simulink data dictionary. However, a Simulink data dictionary provides more capabilities. For details, see “What Is a Data Dictionary?”.

## Prerequisites

- The Simulink compiled data type for all instances of a tunable parameter must be the same.
- Simulink blocks that use tunable parameters with the same name must operate at the same data rate.

To learn more about Simulink compiled data types, see “Control Block Parameter Data Types”.

## Create and Add Tunable Parameter That Maps to DUT Ports

To generate a DUT port for a tunable parameter:

- 1 Create a tunable parameter with `StorageClass` set to `ExportedGlobal`.

For example, to create a tunable parameter, *myParam*, and initialize it to 5, at the command line, enter:

```
myParam = Simulink.Parameter;
myParam.Value = 5;
myParam.CoderInfo.StorageClass = 'ExportedGlobal';
```

Alternatively, using the Model Explorer, create a tunable parameter and:

- In the **Storage Class** column of the **Contents** pane, click the **configure** link.
- In the **Dialog** pane, click **Configure in Coder App**. Set **Storage Class** to `ExportedGlobal`.

See “Create Data Objects from Built-In Data Class Package Simulink”.

- 2 In your Simulink design, use the tunable parameter as the:

- **Constant value** in a Constant block.
- **Gain** parameter in a Gain block.
- MATLAB function argument in a MATLAB Function block.

## Generated Code

The following VHDL code is an example of code that HDL Coder generates for a Gain block with its **Gain** field set to a tunable parameter, *myParam*. You see that the code generator creates a DUT port and adds a comment to indicate that the port corresponds to a tunable parameter.

```
ENTITY s IS
  PORT( In1      : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5
        myParam  : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En5 Tunable port
        Out1     : OUT  std_logic_vector(31 DOWNTO 0) -- sfix32_En10
      );
END s;

ARCHITECTURE rtl OF s IS

  -- Signals
  SIGNAL myParam_signed  : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL In1_signed       : signed(15 DOWNTO 0); -- sfix16_En5
  SIGNAL Gain_out1        : signed(31 DOWNTO 0); -- sfix32_En10

BEGIN
  myParam_signed <= signed(myParam);

  In1_signed <= signed(In1);

  Gain_out1 <= myParam_signed * In1_signed;

  Out1 <= std_logic_vector(Gain_out1);

END rtl;
```

## Limitations

Make sure that the “Use trigger signal as clock” on page 17-41 check box is left cleared by default.

## Use Tunable Parameter in Other Blocks

To use the value of a tunable parameter in a Chart or Truth Table block:

- 1** Create the tunable parameter and use it in a Constant block. on page 10-17
- 2** Add an input port to the block where you want to use the tunable parameter.
- 3** Connect the output of the Constant block to the new input port.

## See Also

### Related Examples

- “Generate Parameterized Code for Referenced Models” on page 10-20

## Generate Parameterized Code for Referenced Models

### In this section...

[“Parameterize Referenced Model for HDL Code Generation” on page 10-20](#)

[“Restrictions” on page 10-20](#)

To generate parameterized code for referenced models, use model arguments. You can use model arguments in a masked or unmasked Model block.

HDL Coder generates a single VHDL entity or Verilog module for the referenced model, even if the DUT has multiple instances of the referenced model. In the generated code, each model argument is a VHDL generic or a Verilog parameter.

### Parameterize Referenced Model for HDL Code Generation

- 1 In the referenced model, create one or more model arguments.

To learn how to create a model argument, see “Specify a Different Value for Each Instance of a Reusable Model”.

- 2 In the referenced model, use each model argument parameter in a Gain or Constant block.
- 3 In the DUT, for each model reference, in the **Model arguments** table, enter values for each model argument.

Alternatively, create a model mask for the referenced model. In the DUT, for each model reference, enter values for each model argument.

- 4 Generate code for the DUT.

### Restrictions

Model argument values:

- Must be scalar.
- Cannot be complex.
- Cannot be enumerated data.

### See Also

#### Related Examples

- [“Generate Reusable Code for Atomic Subsystems” on page 27-17](#)
- [“Generate DUT Ports for Tunable Parameters” on page 10-17](#)

#### More About

- [“Specify a Different Value for Each Instance of a Reusable Model”](#)
- [“Model Referencing for HDL Code Generation” on page 27-2](#)

# Generating HDL Code for Subsystems with Array of Buses

## In this section...

["How HDL Coder Generates Code for Array of Buses" on page 10-21](#)

["Array of Buses Limitations" on page 10-23](#)

An array of buses is an array whose elements are buses. Each element in an array of buses must be nonvirtual and must have the same data type.

The array of buses represents structured data compactly. The array:

- Reduces the model complexity
- Reduces maintenance by organizing and routing signals in your Simulink model for vectorized algorithms

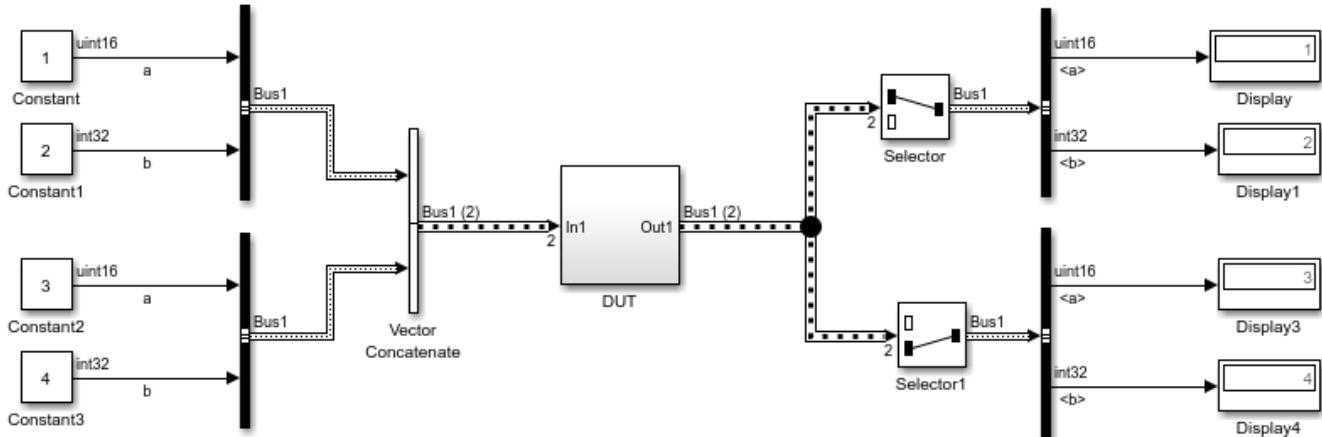
For more information, see "Combine Buses into an Array of Buses".

You can generate HDL code for virtual and nonvirtual blocks that Simulink supports with an array of buses. For more information, see "Use Arrays of Buses in Models".

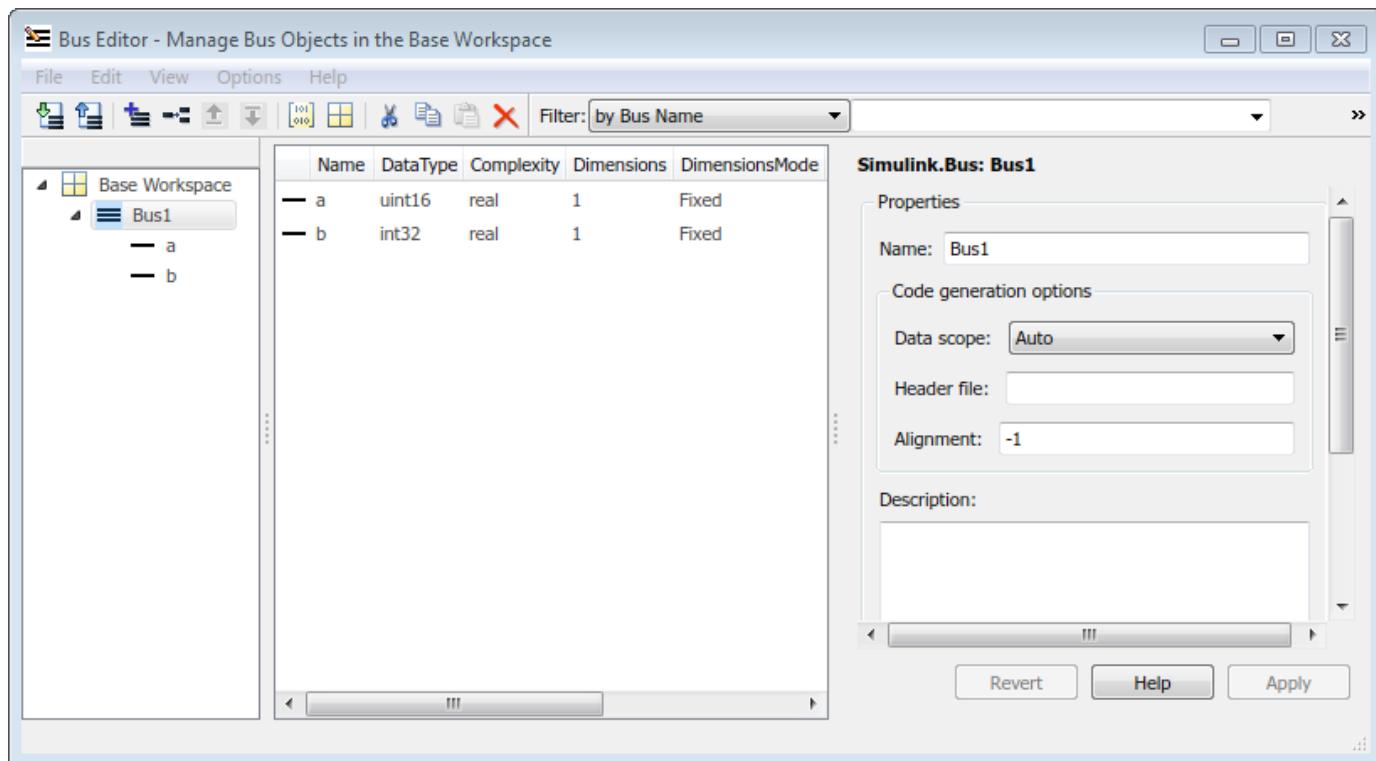
## How HDL Coder Generates Code for Array of Buses

HDL Coder expands the array of buses in your Simulink model into the corresponding scalar signals in the generated code.

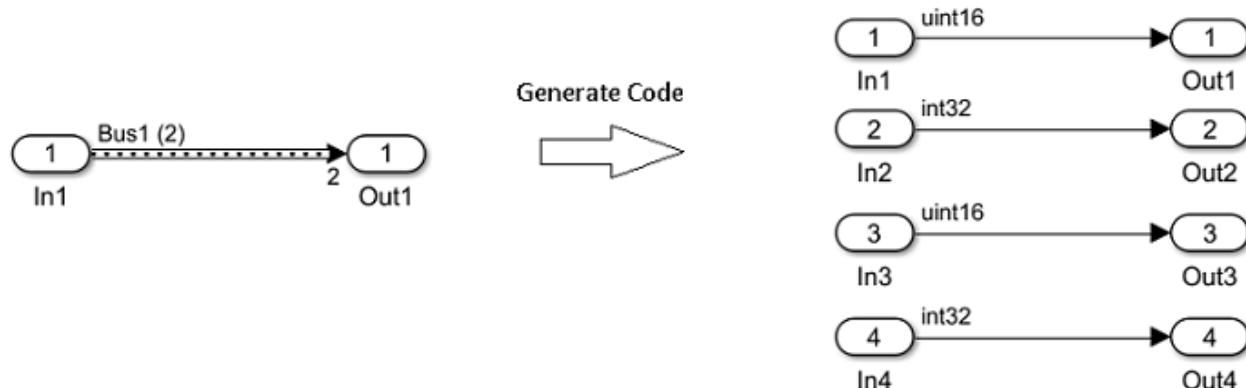
This Simulink model has an array of buses signal at the DUT interface.



The array of buses combines two nonvirtual bus elements, each having scalars **a** and **b** of types **uint16** and **int32** respectively.



The resulting HDL code expands the array of buses into scalars, and contains four scalar input and output ports.



In the generated code, the array of bus expansion results in four scalar signals at the input and output ports. For the first bus object, the input ports are `In_1_a` and `In_1_b`. For the second bus object, they are `In_2_a` and `In_2_b`. At the output, for the first bus object, they are `Out_1_a` and `Out_1_b`. For the second bus object, they are `Out_2_a` and `Out_2_b`.

```
ENTITY DUT IS
PORT( In1_1_a : IN std_logic_vector(15 DOWNTO 0); -- uint16
      In1_1_b : IN std_logic_vector(31 DOWNTO 0); -- int32
      In1_2_a : IN std_logic_vector(15 DOWNTO 0); -- uint16
      In1_2_b : IN std_logic_vector(31 DOWNTO 0); -- int32
```

```

    Out1_1_a : OUT std_logic_vector(15 DOWNTO 0); -- uint16
    Out1_1_b : OUT std_logic_vector(31 DOWNTO 0); -- int32
    Out1_2_a : OUT std_logic_vector(15 DOWNTO 0); -- uint16
    Out1_2_b : OUT std_logic_vector(31 DOWNTO 0) -- int32
  );
END DUT;

```

HDL Coder generates code in accordance with the order in which you specify the bus elements and the array elements in your Simulink model. If you specify the VHDL target language for your Simulink model that contains a bus object with arrays, HDL Coder preserves the arrays in the generated code, and does not expand into scalars.

## Array of Buses Limitations

- Do not use the array of buses inside other data types. You cannot use a bus signal that contains an array of buses.
- MATLAB System and MATLAB Function blocks that contain System Objects are not supported with an array of buses.

## See Also

### More About

- “Signal and Data Type Support” on page 10-2
- “Signal Types”
- “About Data Types in Simulink”
- “Composite Signals”
- “Use Enumerated Data in Simulink Models”
- “Enumerated Data” (Stateflow)

## Implement Control Signals Based Mathematical Functions using HDL Coder

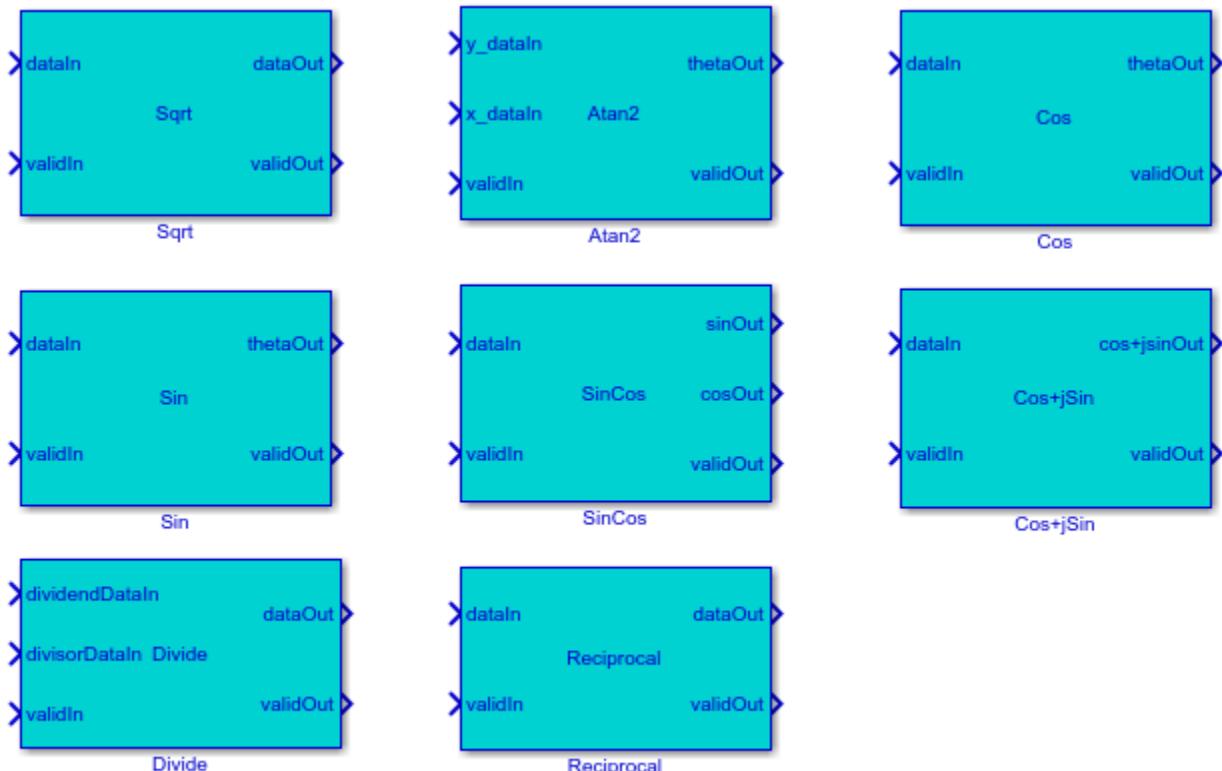
This document gives the overview of the control signal based fixed point mathematical functions in HDLMATHLib and examples associated with all the blocks present in the HDLMATHLib by using HDL Coder™. HDLMATHLib includes following blocks with control ports.

- 1 Sqrt
- 2 Atan2
- 3 Sin
- 4 Cos
- 5 SinCos
- 6 Cos+jSin
- 7 Reciprocal
- 8 Divide

### HDLMathLib Library With Control Ports for Mathematical Functions

To see all the mathematical function blocks in the HDLMATHLib library, open the library using following command.

```
open_system('HDLMathLib')
```



You can see various mathematical function blocks with control ports. Sqrt, Atan2, SinCos, Reciprocal and Divide blocks are described with example in the following sections. You can use sin, cos, Cos+jsin blocks in your model same way by referring below sections.

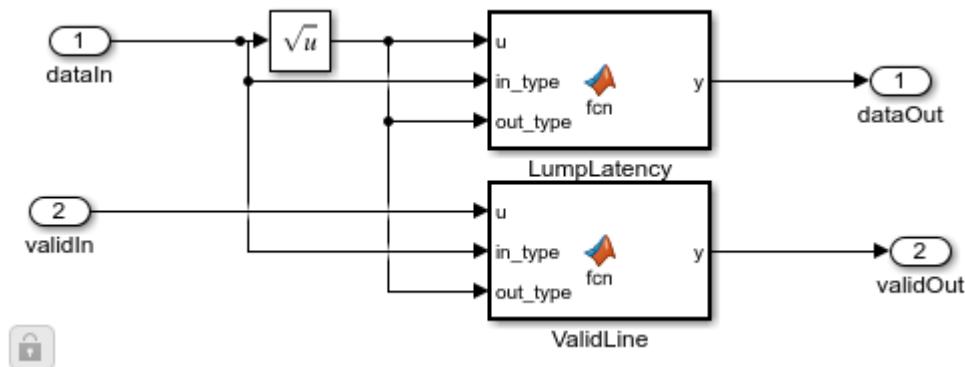
### Sqrt Block with Control Signals

Each port of the Sqrt block is as explained below.

Input Ports		Output Ports	
dataIn	Input data port for the block	dataOut	Square root output data port
validIn	Input data valid control port	validOut	Control port for valid output

When you open the Sqrt block, it uses MATLAB Function blocks for the data and valid lines. This is as shown below.

```
open_system('hdlcoder_sqrt_bitset_control')
open_system('hdlcoder_sqrt_bitset_control/Sqrt')
```



### Example using Control Signals based Sqrt Block

This section shows how to implement an example using control signal based Square root block and generate HDL code by using HDL Coder™.

#### Open and Run Simulink Model

Before opening the model, set the input as follows. You can chose to set different input as per your requirement. This example uses following inputs which is linear.

```
SQRT_input = fi(1/2^17:1/2^17:1,0,18,17)';
```

Specify the wordlength for fixed-point datatypes and the pipeline latency for the model. Go through the documentation for latency calculation.

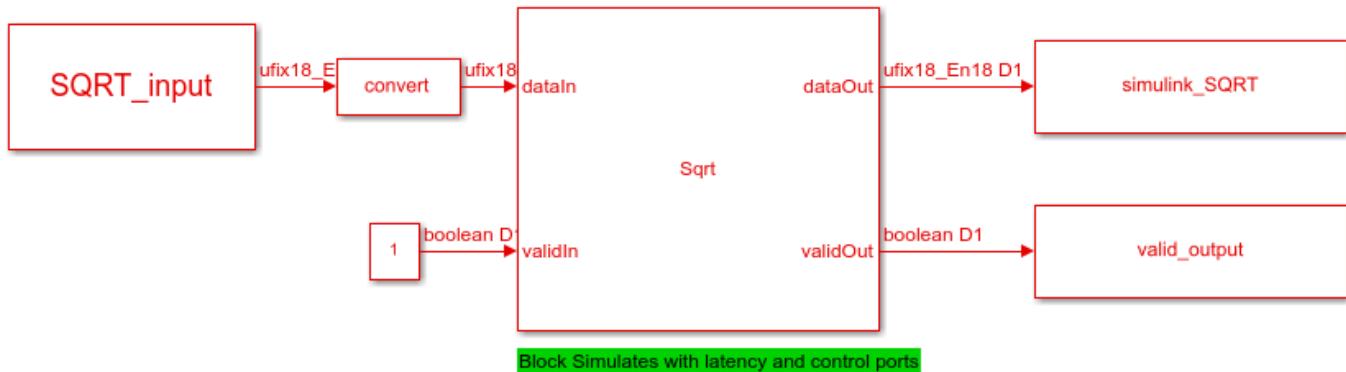
```
WL = 18; latency = 20;
```

Open the model `hdlcoder_sqrt_bitset_control` and specify sufficient stop time that required to process all the input combinations.

```

stopTime = length(SQRT_input)-1+latency;
open_system('hdlcoder_sqrt_bitset_control')
sim('hdlcoder_sqrt_bitset_control')

```



Copyright 2020 The MathWorks, Inc.

You can see the below waveform when you simulate the above model. You can see that dataOut is valid when validOut is high.



### Validate Simulink Output by Using Reference Output

To validate the output obtained by simulating the Simulink model, you can compare the Simulink output with a reference output. To obtain the reference output, use the `sqrt` function.

Compute the reference output by using the `sqrt` function.

```
ref_SQRT = sqrt(double(SQRT_input));
```

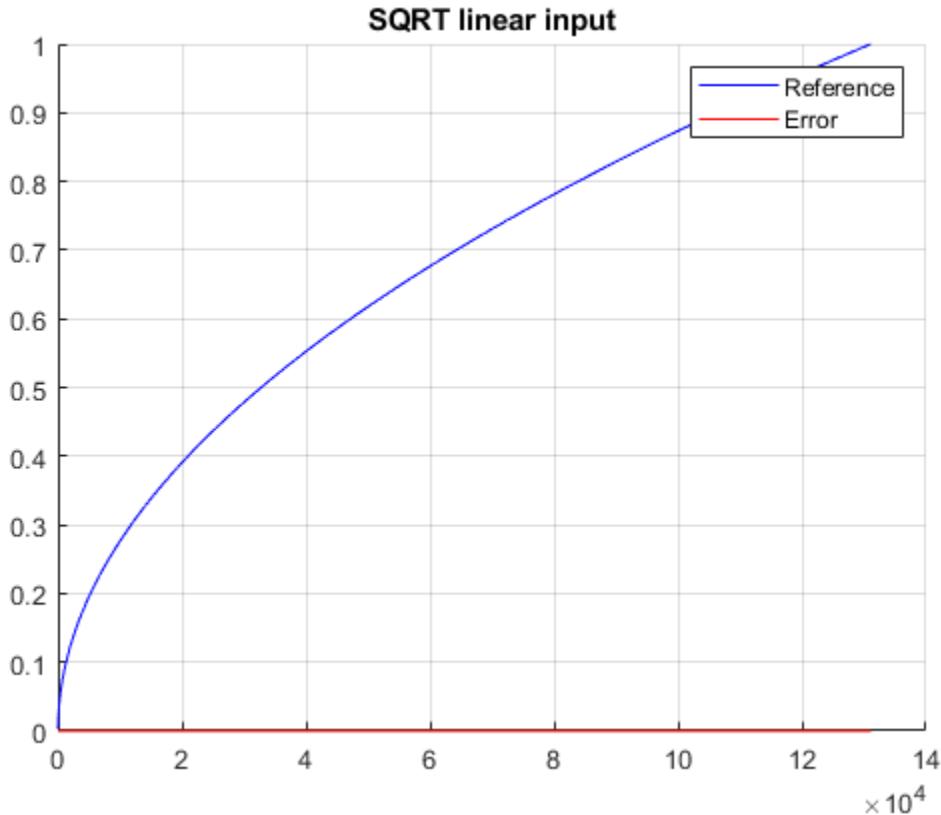
Use logical indexing to extract valid output.

```
implementation_SQRT = simulink_SQRT(valid_output);
```

To validate the output, plot the comparison results by using the `comparison_plot` function in this example. You can see that the maximum error observed from the comparison results is quite small.

```
comparison_plot_sqrt(ref_SQRT,implementation_SQRT,1,'SQRT linear input');
```

Maximum Error SQRT linear input 3.814697e-06  
Maximum PctError SQRT linear input 3.803159e-02



### Generate HDL Code for Square Root Implementation

Before you generate code, you can see the HDL settings saved on the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sqrt_bitset_control')

%% Set Model 'hdlcoder_sqrt_bitset_control' HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_sqrt_bitset_control', 'HDLSubsystem', 'hdlcoder_sqrt_bitset_control/Sqrt');
hdlset_param('hdlcoder_sqrt_bitset_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sqrt_bitset_control', 'ResourceReport', 'on');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_sqrt_bitset_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_sqrt_bitset_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_sqrt_bitset_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_sqrt_bitset_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/LumpLatency', 'Architecture', 'MATLAB Datapath')
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/LumpLatency', 'FlattenHierarchy', 'on');
```

```

hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/ValidLine', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sqrt_bitset_control/Sqrt/ValidLine', 'FlattenHierarchy', 'on');

```

To generate HDL code for the **Sqrt** block in the model, use the **makehdl** function.

```

makehdl('hdlcoder_sqrt_bitset_control/Sqrt')
close_system('hdlcoder_sqrt_bitset_control')
close all;

### Generating HDL for 'hdlcoder_sqrt_bitset_control/Sqrt'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_sqrt_b...
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_sqrt_bitset_control'.
### Working on hdlcoder_sqrt_bitset_control/Sqrt/Sqrt as hdl_prj\hdlsrc\hdlcoder_sqrt_bitset...
### Working on hdlcoder_sqrt_bitset_control/Sqrt as hdl_prj\hdlsrc\hdlcoder_sqrt_bitset_control...
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_152...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_sqrt_bitset_control' complete with 0 errors, 0 warnings, and 0 message...
### HDL code generation complete.

```

### Sqrt Block Synthesis Performance

Following digram shows the Sqrt block synthesis performance on the Xilinx Virtex 7 and intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
234	499	1167	954

Intel Quartus Stratix V (5SEE9F45C2)

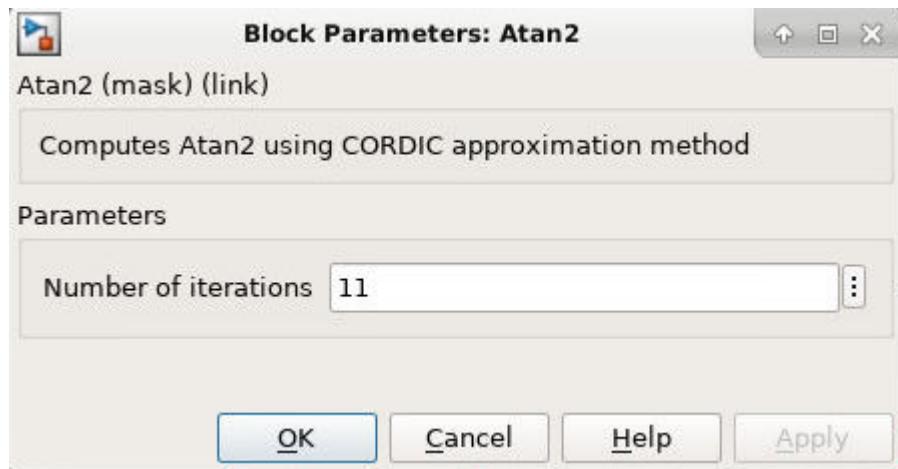
Fmax (MHz)	LABs	ALMs	Registers
185	110	823	848

### Atan2 Block with Control Signals

Each port of the Atan2 block is as explained below.

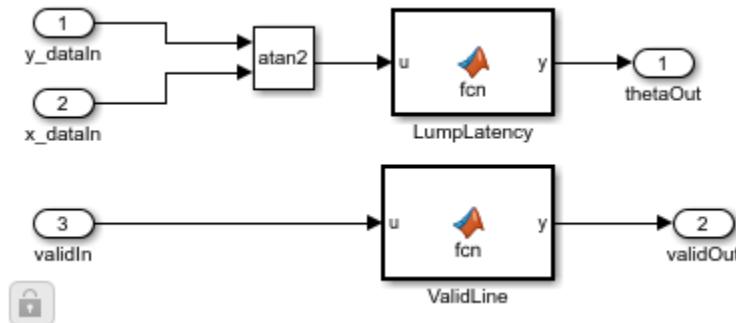
Input Ports		Output Ports	
y_dataln	Input y data port for the block	thetaOut	Angle output port
x_dataln	Input x data port for the block	validOut	Control port for valid output
validIn	Input data valid control port		

The Atan2 block has number of iterations as mask parameter. The default value is 11 and latency depends on this masked parameter.



When you open the Atan2 block, it uses MATLAB Function blocks for the data and valid lines. This is as shown below.

```
open_system('hdlcoder_atan2_control')
open_system('hdlcoder_atan2_control/Atan2')
open_system('hdlcoder_atan2_control/Atan2','force')
```



### Example using Control Signal based Atan2 Block

This section shows how to implement an example using control signal based Atan2 block and generate HDL code by using HDL Coder™.

### Open and Run Simulink Model

Before opening the model, set the input as follows. You can chose to set different input as per your requirement. This example uses following inputs which is linear and sweep through input values  $-\pi$  to  $\pi$ .

```
input_values = (-pi:.01/(2*pi):pi)';
RADIUS = 10.^(-2.5:.25:0);
```

Specify the wordlength for fixed-point datatypes and the pipeline latency for the model. Go through the documentation for latency calculation. The latency depends up on the number of iterations.

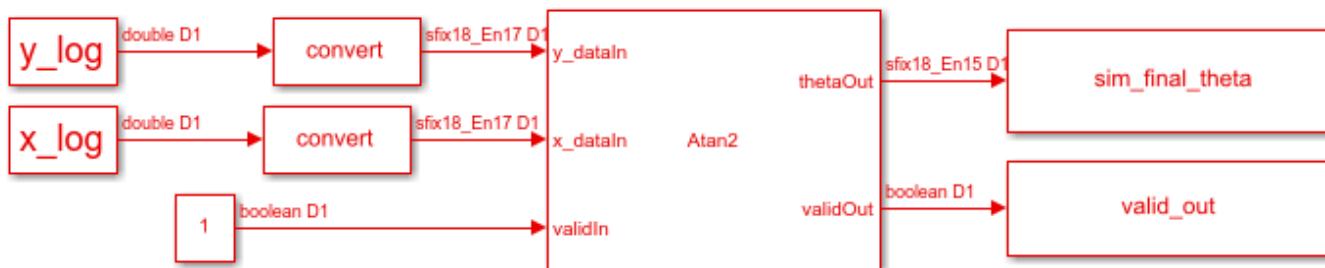
```
WL_atan2 = 18; latency_atan2 = 14;
```

Setup variables for logging input x and y values.

```
x_log = zeros(length(input_values)*length(RADIUS),1);
y_log = zeros(length(input_values)*length(RADIUS),1);
for outerindex = 0:length(RADIUS)-1
    for index = 1:length(input_values)
        input = input_values(index); % access current value
        y = RADIUS(outerindex+1)*sin(input); % compute y
        x = RADIUS(outerindex+1)*cos(input); % compute x
        addr = outerindex*length(input_values)+index;
        y_log(addr) = y;
        x_log(addr) = x;
    end
end
```

Open the model `hdlcoder_atan2_control` and specify sufficient stop time that required to process all the input combinations. The model has Atan2 block that implements the Atan2 using CORDIC algorithm for a `validIn` control signal.

```
stoptime_atan2 = length(x_log)-1+latency_atan2;
close all
open_system('hdlcoder_atan2_control')
sim('hdlcoder_atan2_control')
```



Copyright 2020 The MathWorks, Inc.

You can see the below waveform when you simulate the above model. You can see that `dataOut` is valid when `validOut` is high.

► y_dataIn	22a3f	2019e 3fd2a 3fa28 3e350 2c3 3e6e8 aca0 298 277 14e 100f a09e 11713 3ffa 2386 42c8 367a5 2a13 3f9c6 2ea3
► x_dataIn	cba9	3d754 3f759 3f0e6 279 cf e1e 1e205 3fec1 17c 3#0b 3e81a 3110b 3b931 3fe61 3ac2b 11811 3c932 1d2b 3f10d 11e0
validIn	1	
► thetaOut	11528	0
validOut	1	32cce 2067c 29d22 341d2 a474 3786c

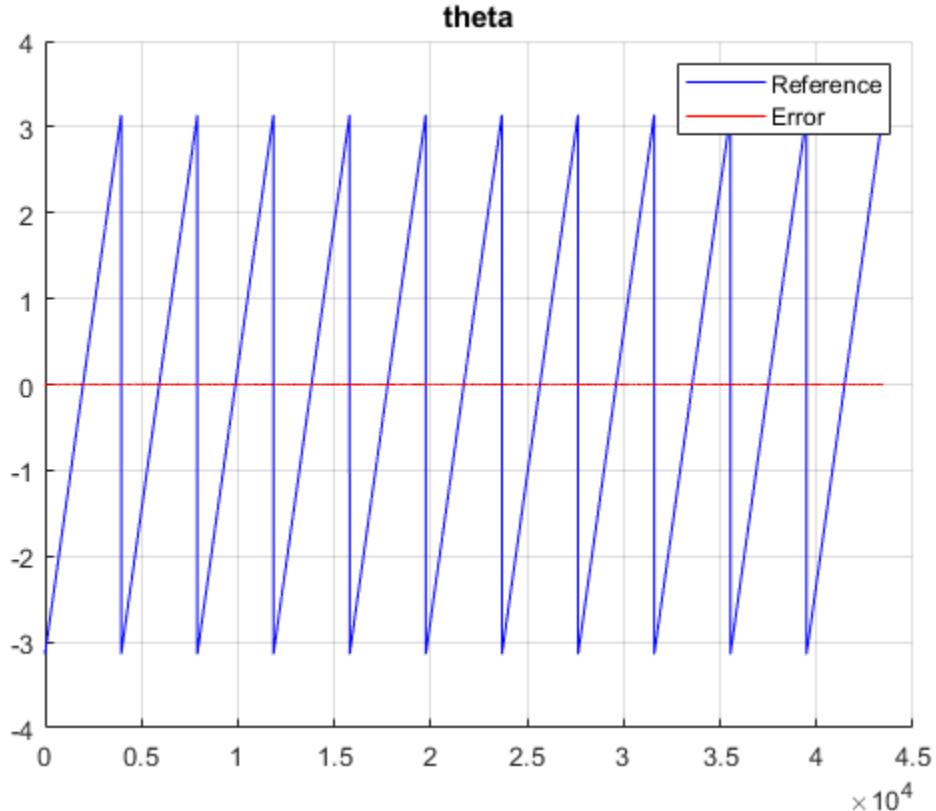
### Validate Simulink Output By Using Reference Output

To validate the output obtained by simulating the Simulink model, you can compare the Simulink output with a reference output. To obtain the reference output, use the atan2 MATLAB function.

Compute the reference output by using the atan2 function. You can see that the maximum error observed from the comparison results is quite small.

```
comparison_plot_atan2(atan2(y_log,x_log),sim_final_theta(valid_out),3,'theta');

Maximum Error theta 7.233221e-03
```



### Generate HDL Code for Atan2 Implementation

Before you generate code, you can see the HDL settings saved on the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_atan2_control')

%% Set Model 'hdlcoder_atan2_control' HDL parameters
hdlset_param('hdlcoder_atan2_control', 'HDLSubsystem', 'hdlcoder_atan2_control/Atan2');
```

```

hdlset_param('hdlcoder_atan2_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_atan2_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_atan2_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_atan2_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_atan2_control', 'TargetFrequency', 500);

hdlset_param('hdlcoder_atan2_control/Atan2/LumpLatency', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_atan2_control/Atan2/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_atan2_control/Atan2/ValidLine', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_atan2_control/Atan2/ValidLine', 'FlattenHierarchy', 'on');

```

To generate HDL code for the Atan2 block in the model, use the `makehdl` function.

```

makehdl('hdlcoder_atan2_control/Atan2')
close_system('hdlcoder_atan2_control')
close all;

### Generating HDL for 'hdlcoder_atan2_control/Atan2'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_atan2_c
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_atan2_control'.
### Working on hdlcoder_atan2_control/Atan2/Atan2 as hdl_prj\hdlsrc\hdlcoder_atan2_control\Atan2.vhd
### Working on hdlcoder_atan2_control/Atan2 as hdl_prj\hdlsrc\hdlcoder_atan2_control\Atan2.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'hdlcoder_atan2_control' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

```

### Atan2 Block Synthesis Performance

Following digrams shows the Atan2 block synthesis performance on the Xilinx Virtex 7 and intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
544	231	725	620

Intel Quartus Stratix V (5SEE9F45C2)

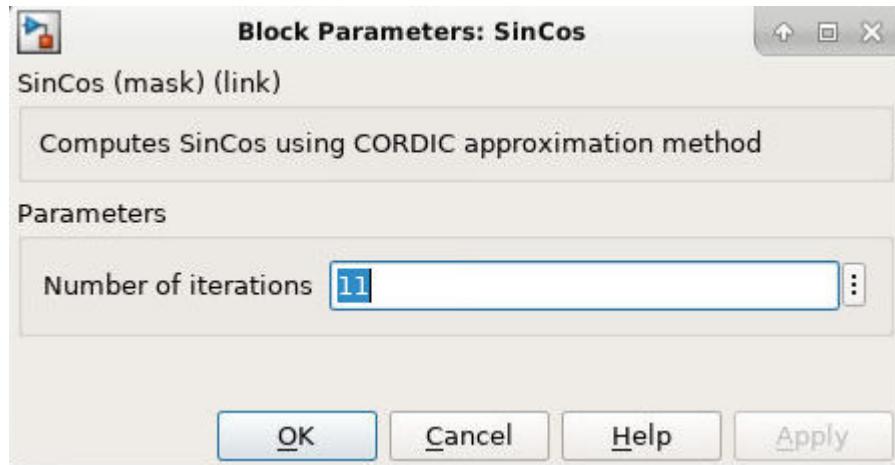
Fmax (MHz)	LABs	ALMs	Registers
565	61	342	674

### SinCos Block with Control Signals

Each port of the SinCos block is as explained below.

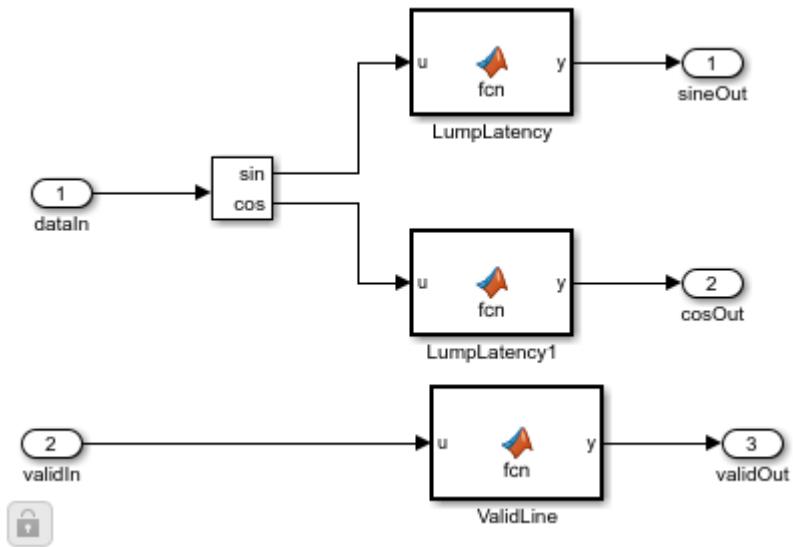
Input Ports		Output Ports	
dataIn	Input data port for the block	sinOut	Sin output port
validIn	Input data valid control port	cosOut	Cos output port
		validOut	Control port for valid output

The SinCos block has number of iterations as the mask parameter. The default value is 11 and latency depends on this masked parameter.



When you open the SinCos block, it uses MATLAB Function blocks for the data and valid lines. This is as shown below.

```
open_system('hdlcoder_sincos_control')
open_system('hdlcoder_sincos_control/SinCos')
open_system('hdlcoder_sincos_control/SinCos','force')
```



### Example using Control Signal based SinCos Block

This section shows how to implement an example using control signal based SinCos block and generate HDL code by using HDL Coder™.

#### Open and Run Simulink Model

Before opening the model, set the input as follows. You can chose to set different input as per your requirement. This example uses following inputs which is linear and sweep through input values -pi to pi.

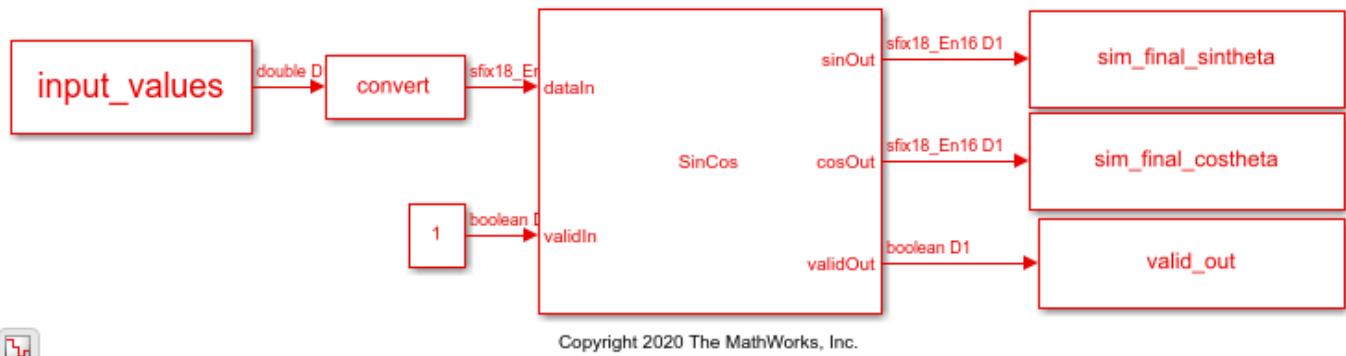
```
input_values = (-pi:.01/(2*pi):pi)';
```

Specify the wordlength for fixed-point datatypes and the pipeline latency for the model. Go through the documentation for latency calculation. The latency depends up on the number of iterations

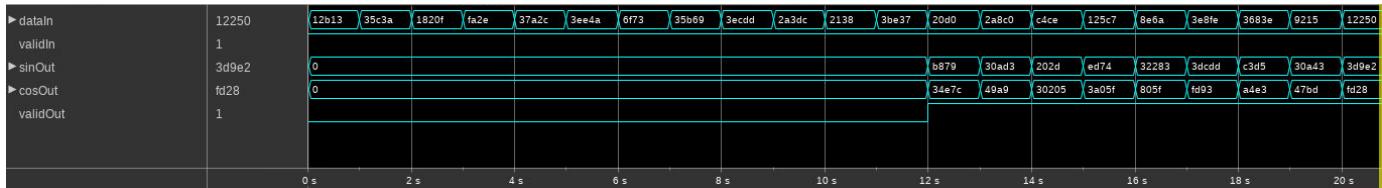
```
WL_SinCos = 18; latency_SinCos = 12;
```

Open the model `hdlcoder_sincos_control` and specify sufficient stop time that required to process all the input combinations. The model has SinCos block that implements the SinCos using CORDIC algorithm for a `validIn` control signal. The remaining trigonometric function blocks(Sin, Cos and Cos + jSin) use the same CORDIC approximation method and the interface ports differs respectively.

```
stoptime_sincos = length(input_values)-1+latency_SinCos;
open_system('hdlcoder_sincos_control')
sim('hdlcoder_sincos_control')
```



You can see the below waveform when you simulate the above model. You can see that dataOut is valid when validOut is high.



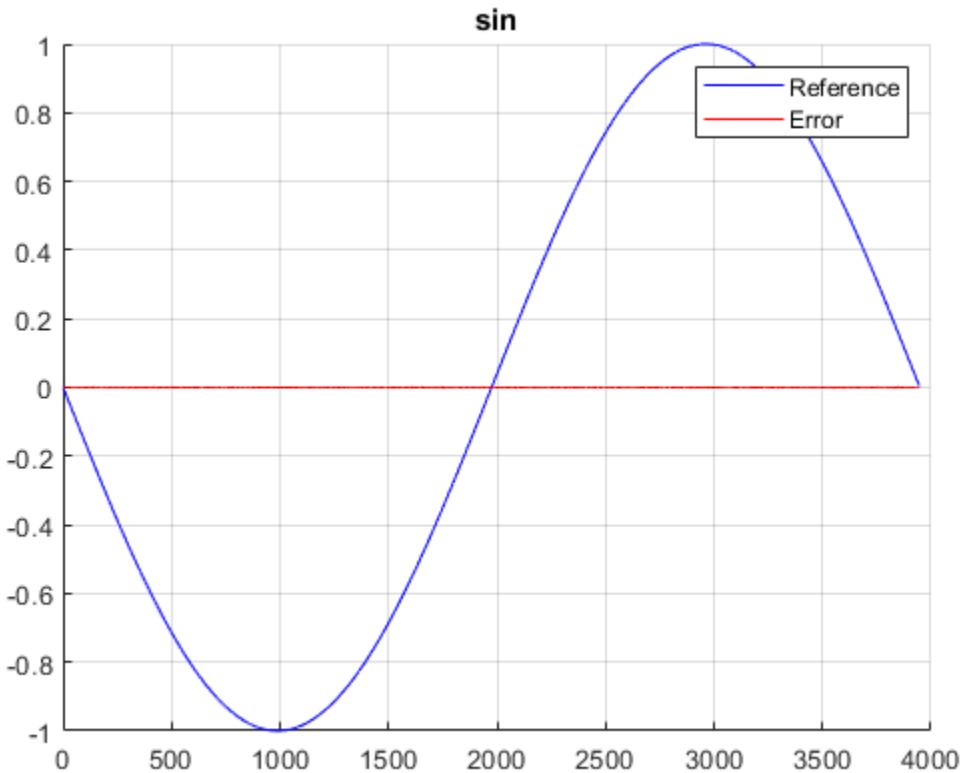
### Validate Simulink Output By Using Reference Output

To validate the output obtained by simulating the Simulink model, you can compare the Simulink output with a reference output. To obtain the reference output, use the `sin` and `cos` MATLAB function.

Compute the reference output by using the `sin` function. You can see that the maximum error observed from the comparison results is quite small.

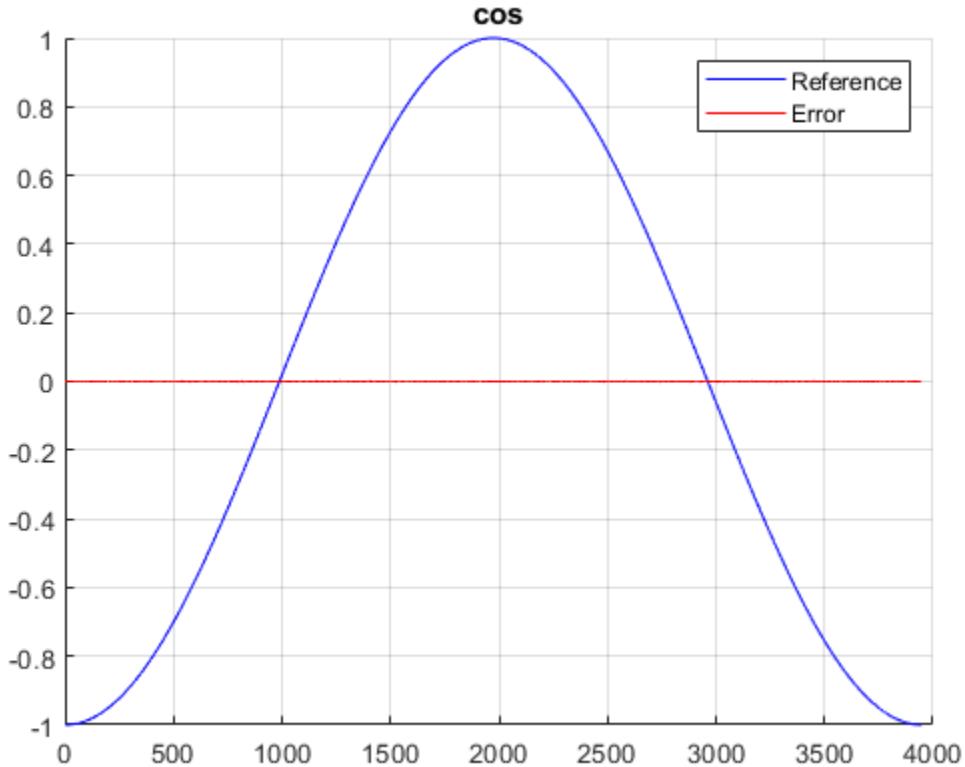
```
comparison_plot_sincos(sin(input_values),sim_final_syntheta(valid_out),5,'sin');

Maximum Error sin 1.005291e-03
```



Compute the reference output by using the Cos function. You can see that the maximum error observed from the comparison results is quite small.

```
comparison_plot_sincos(cos(input_values),sim_final_costheta(valid_out),6,'cos');
Maximum Error cos 1.008159e-03
```



### Generate HDL Code for SinCos Implementation

Before you generate code, you can see the HDL settings saved on the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sincos_control')

%% Set Model 'hdlcoder_sincos_control' HDL parameters
hdlset_param('hdlcoder_sincos_control', 'HDLSubsystem', 'hdlcoder_sincos_control/SinCos');
hdlset_param('hdlcoder_sincos_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sincos_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_sincos_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_sincos_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_sincos_control', 'TargetFrequency', 500);

hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency1', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_sincos_control/SinCos/LumpLatency1', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_sincos_control/SinCos/ValidLine', 'Architecture', 'MATLAB Datapath');
% Set SubSystem HDL parameters
```

```
hdlset_param('hdlcoder_sincos_control/SinCos/ValidLine', 'FlattenHierarchy', 'on');
```

To generate HDL code for the **SinCos** block in the model, use the **makehdl** function.

```
makehdl('hdlcoder_sincos_control/SinCos')
close_system('hdlcoder_sincos_control')
close all;

### Generating HDL for 'hdlcoder_sincos_control/SinCos'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_sincos_
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_sincos_control'.
### Working on hdlcoder_sincos_control/SinCos/SinCos as hdl_prj\hdlsrc\hdlcoder_sincos_control\SinCos\SinCos
### Working on hdlcoder_sincos_control/SinCos as hdl_prj\hdlsrc\hdlcoder_sincos_control\SinCos\SinCos
### Generating package file hdl_prj\hdlsrc\hdlcoder_sincos_control\SinCos_pkg.vhd.
### Creating HDL Code Generation Check Report file:7\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpr
### HDL check for 'hdlcoder_sincos_control' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

### **SinCos Block Synthesis Performance**

Following digrams shows the SinCos block synthesis performance on the Xilinx Virtex 7 and intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
573	226	680	606

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
568	61	352	610

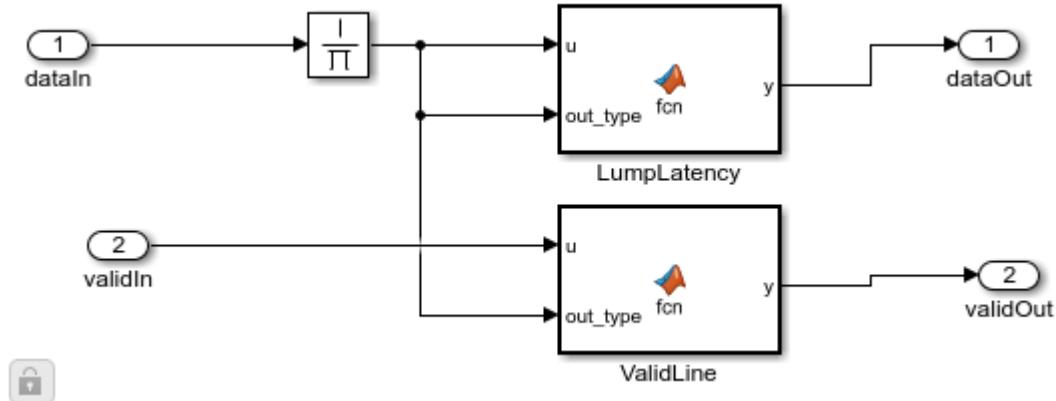
### **Reciprocal Block with Control Signals**

Each port of the Reciprocal block is as explained below.

Input Ports		Output Ports	
dataIn	Input data port for the block	dataOut	data output port
validIn	Input data valid control port	validOut	Control port for valid output

When you open the Reciprocal block, it uses MATLAB Function blocks for the data and valid lines. This is as shown below.

```
open_system('hdlcoder_reciprocal_shiftadd_control')
open_system('hdlcoder_reciprocal_shiftadd_control/Reciprocal')
```



### Example using Control Signal based Reciprocal Block

This section shows how to implement an example using control signal based Reciprocal block and generate HDL code by using HDL Coder™.

#### Open and Run Simulink Model

Before opening the model, set the input as follows. You can chose to set different input as per your requirement. This example uses following inputs which is linear.

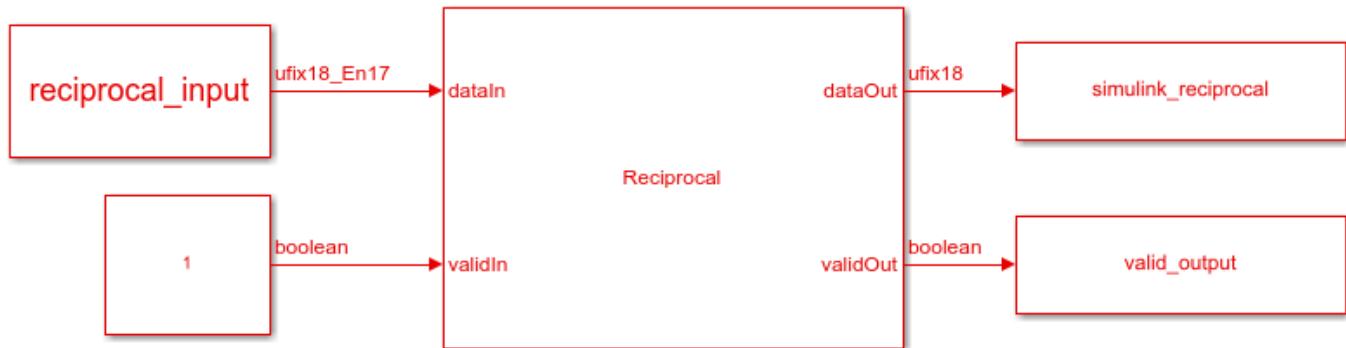
```
reciprocal_input = fi(1/2^17:1/2^17:1,0,18,17)';
```

Specify the wordlength for fixed-point datatypes and the pipeline latency for the model. Go through the documentation for latency calculation.

```
WL_recip = 18; recip_latency = 22;
```

Open the model `hdlcoder_reciprocal_shiftadd_control` and specify sufficient stop time that required to process all the input combinations.

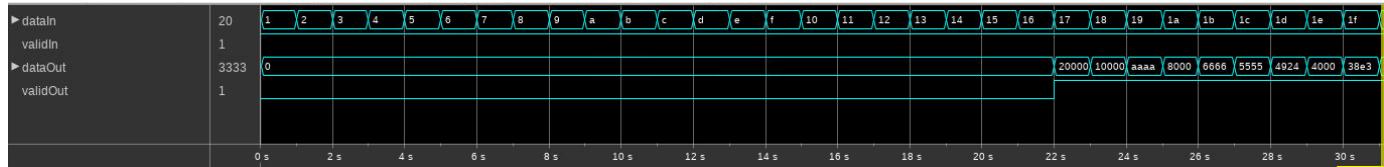
```
stoptime_recip = length(reciprocal_input)-1+recip_latency;
open_system('hdlcoder_reciprocal_shiftadd_control')
sim('hdlcoder_reciprocal_shiftadd_control')
```



Copyright 2020 The MathWorks, Inc.



You can see the below waveform when you simulate the above model. You can see that dataOut is valid when validOut is high.



### Validate Simulink Output By Using Reference Output

To validate the output obtained by simulating the Simulink model, you can compare the Simulink output with a reference output. To obtain the reference output, use the `sqrt` function.

Compute the reference output by using the `reciprocal` operation.

```
ref_reciprocal = 1./double(reciprocal_input);
```

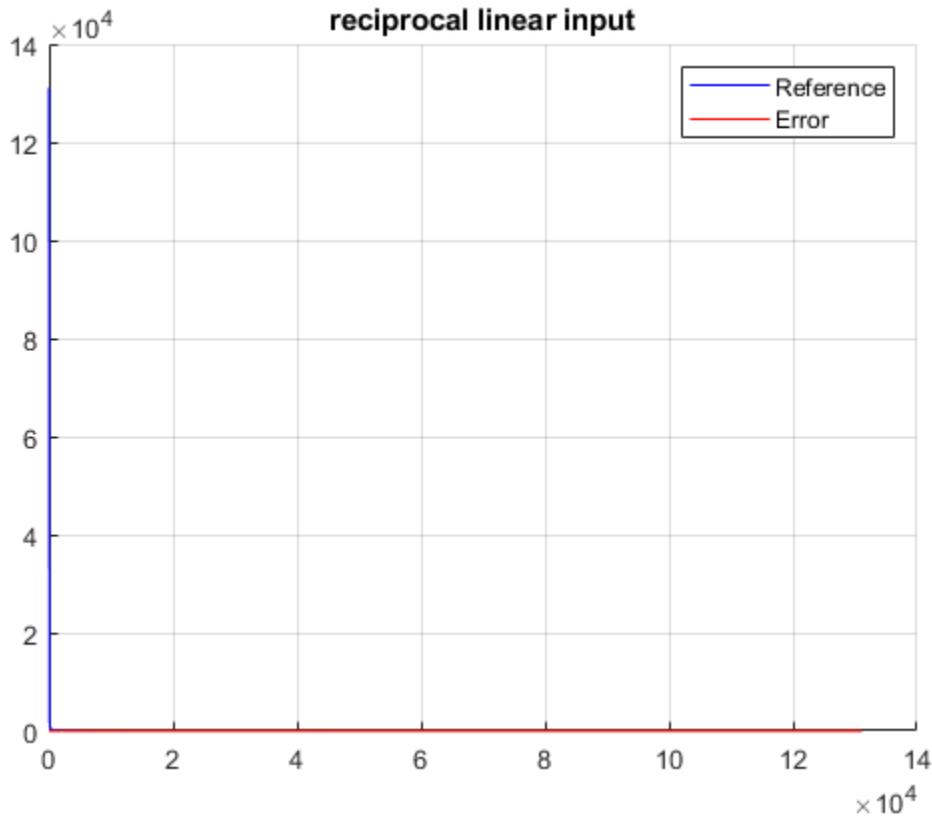
Use logical indexing to extract valid output.

```
implementation_reciprocal = simulink_reciprocal(valid_output);
```

To validate the output, plot the comparison results by using the `comparison_plot_reciprocal` function in this example. You can see that the maximum error observed from the comparison results is quite small.

```
comparison_plot_reciprocal(ref_reciprocal,implementation_reciprocal,9,'reciprocal linear input')
```

Maximum Error reciprocal linear input 9.999771e-01  
 Maximum PctError reciprocal linear input 4.999924e+01



### Generate HDL Code for Reciprocal Implementation

Before you generate code, you can see the HDL settings saved on the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_reciprocal_shiftadd_control')

%% Set Model 'hdlcoder_reciprocal_shiftadd_control' HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'HDLSubsystem', 'hdlcoder_reciprocal_shiftadd');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'ResourceReport', 'on');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_reciprocal_shiftadd_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/LumpLatency', 'Architecture', 'MATLAB');
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/LumpLatency', 'FlattenHierarchy',
```

```

hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/Reciprocal', 'Architecture', 'Shift')
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/ValidLine', 'Architecture', 'MATLAB')
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_reciprocal_shiftadd_control/Reciprocal/ValidLine', 'FlattenHierarchy', 'on')

```

To generate HDL code for the Reciprocal block in the model, use the `makehdl` function.

```

makehdl('hdlcoder_reciprocal_shiftadd_control/Reciprocal')
close_system('hdlcoder_reciprocal_shiftadd_control')
close all;

### Generating HDL for 'hdlcoder_reciprocal_shiftadd_control/Reciprocal'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_reciprocal...
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_reciprocal_shiftadd_control'.
### Working on hdlcoder_reciprocal_shiftadd_control/Reciprocal as hdl_prj\hdlsrc\hdlcod...
### Working on hdlcoder_reciprocal_shiftadd_control/Reciprocal as hdl_prj\hdlsrc\hdlcoder_recip...
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_152...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdlcoder_reciprocal_shiftadd_control' complete with 0 errors, 0 warnings, and ...
### HDL code generation complete.

```

### Reciprocal Block Synthesis Performance

Following digram shows the Reciprocal block synthesis performance on the Xilinx Virtex 7 and intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
464	184	431	766

Intel Quartus Stratix V (5SEE9F45C2)

Fmax (MHz)	LABs	ALMs	Registers
339	68	380	918

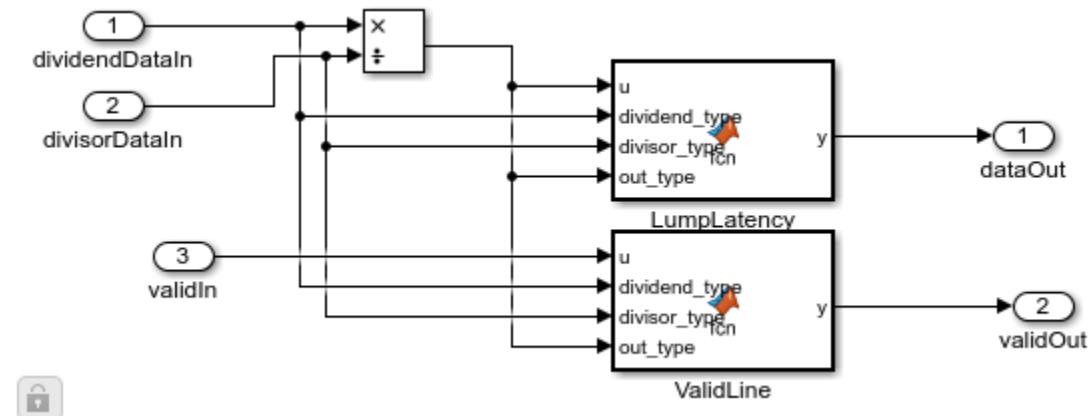
### Divide Block with Control Signals

Each port of the Divide block is as explained below.

Input Ports		Output Ports	
dividendDataIn	Input dividend data port for the block	dataOut	Output data port
divisorDataIn	Input divisor data port for the block	validOut	Control port for valid output
validIn	Input data valid control port		

When you open the Divide block, it uses MATLAB Function blocks for the data and valid lines. This is as shown below.

```
open_system('hdlcoder_divide_shiftadd_control')
open_system('hdlcoder_divide_shiftadd_control/Divide')
```



### Example using Control Signal based Divide Block

This section shows how to implement an example using control signal based Divide block and generate HDL code by using HDL Coder™.

#### Open and Run Simulink Model

Before opening the model, set the input as follows. You can chose to set different input as per your requirement. This example uses following inputs which is linear.

```
dividend_input = fi(1/2^17:1/2^17:1,0,18,17)';
divisor_input = fi(1/2^17:1/2^17:1,0,18,13)';
```

Specify the wordlength for fixed-point datatypes and the pipeline latency for the model. Go through the documentation for latency calculation.

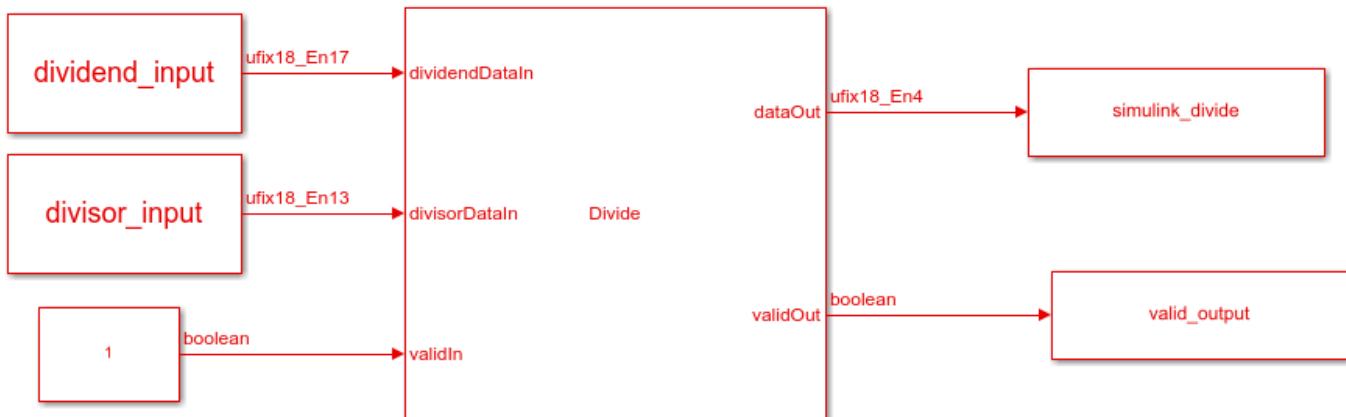
```
WL_divide = 18; divide_latency = 22;
```

Open the model `hdlcoder_divide_shiftadd_control` and specify sufficient stop time that required to process all the input combinations.

```

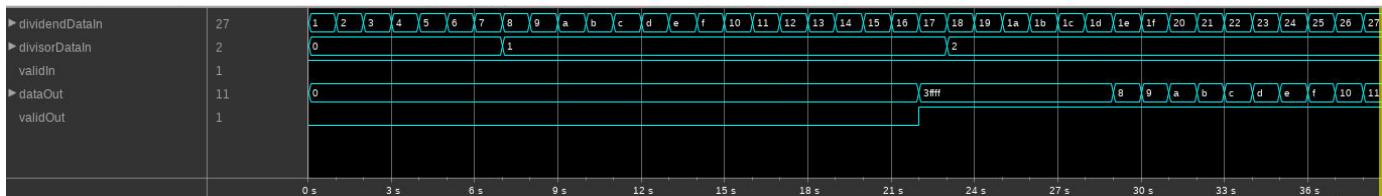
stopTime_divide = length(dividend_input)-1+divide_latency;
open_system('hdlcoder_divide_shiftadd_control')
sim('hdlcoder_divide_shiftadd_control')

```



Copyright 2020 The MathWorks, Inc.

You can see the below waveform when you simulate the above model. You can see that dataOut is valid when validOut is high.



### Validate Simulink Output By Using Reference Output

To validate the output obtained by simulating the Simulink model, you can compare the Simulink output with a reference output. To obtain the reference output, use the `sqrt` function.

Compute the reference output by using the `divide` function.

```
ref_divide = double(dividend_input)./double(divisor_input);
```

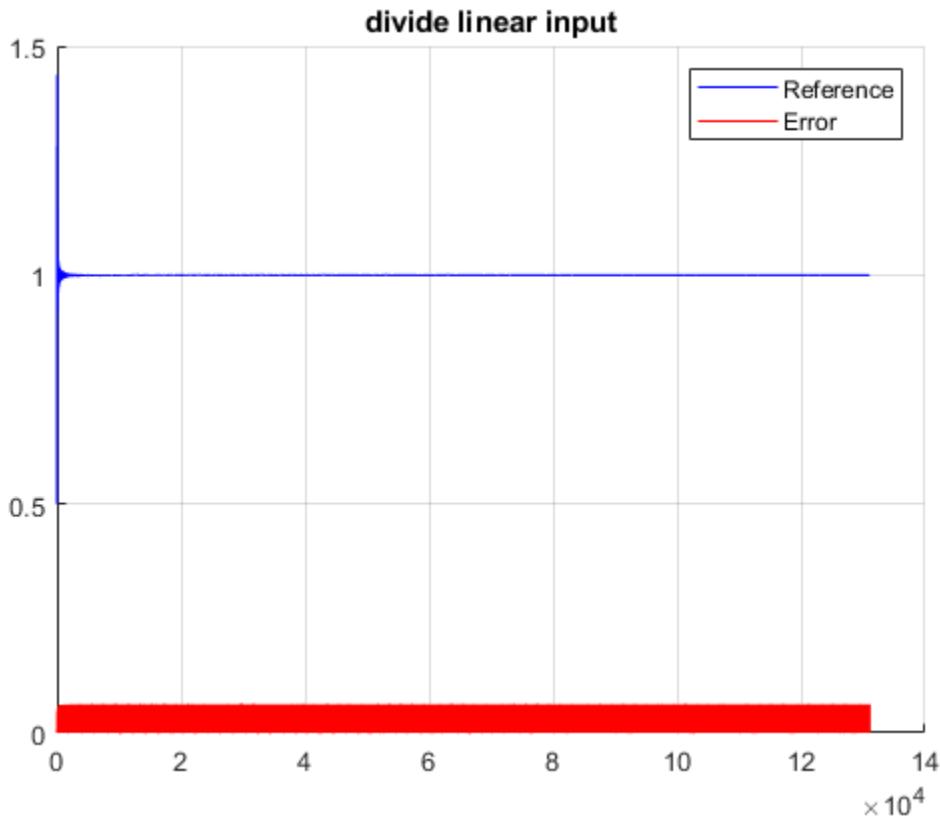
Use logical indexing to extract valid output.

```
implementation_divide = simulink_divide(valid_output);
```

To validate the output, plot the comparison results by using the `comparison_plot_divide` function in this example. You can see that the maximum error observed from the comparison results is quite small.

```
comparison_plot_divide(ref_divide,implementation_divide,11,'divide linear input');
```

```
Maximum Error divide linear input Inf
Maximum PctError divide linear input 6.249285e+00
```



### Generate HDL Code for Divide Implementation

Before you generate code, you can see the HDL settings saved on the model by using the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_divide_shiftadd_control')

%% Set Model 'hdlcoder_divide_shiftadd_control' HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control', 'Backannotation', 'on');
hdlset_param('hdlcoder_divide_shiftadd_control', 'HDLSubsystem', 'hdlcoder_divide_shiftadd_control');
hdlset_param('hdlcoder_divide_shiftadd_control', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_divide_shiftadd_control', 'ResourceReport', 'on');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolDeviceName', 'xc7v2000t');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolPackageName', 'fhg1761');
hdlset_param('hdlcoder_divide_shiftadd_control', 'SynthesisToolSpeedValue', '-2');
hdlset_param('hdlcoder_divide_shiftadd_control', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_divide_shiftadd_control', 'TargetFrequency', 500);
hdlset_param('hdlcoder_divide_shiftadd_control', 'Traceability', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/Divide', 'Architecture', 'ShiftAdd');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/LumpLatency', 'Architecture', 'MATLAB Data');

% Set SubSystem HDL parameters
```

```

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/LumpLatency', 'FlattenHierarchy', 'on');

hdlset_param('hdlcoder_divide_shiftadd_control/Divide/ValidLine', 'Architecture', 'MATLAB Datapa
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_divide_shiftadd_control/Divide/ValidLine', 'FlattenHierarchy', 'on');

```

To generate HDL code for the **Divide** block in the model, use the **makehdl** function.

```

makehdl('hdlcoder_divide_shiftadd_control/Divide')
close_system('hdlcoder_divide_shiftadd_control')
close all;

### Generating HDL for 'hdlcoder_divide_shiftadd_control/Divide'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_divide_
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_divide_shiftadd_control'.
### Working on hdlcoder_divide_shiftadd_control/Divide as hdl_prj\hdlsrc\hdlcoder_divide_
### Working on hdlcoder_divide_shiftadd_control/Divide as hdl_prj\hdlsrc\hdlcoder_divide_shiftadd_
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpo
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpr
### HDL check for 'hdlcoder_divide_shiftadd_control' complete with 0 errors, 0 warnings, and 0 m
### HDL code generation complete.

```

### Divide Block Synthesis Performance

Following digram shows the Divide block synthesis performance on the Xilinx Virtex 7 and intel Stratix V devices.

Xilinx Vivado 7v2000t-fhg1761 (Speed Grade: -2)

Fmax (MHz)	Slices	LUTs	Registers
486	203	462	802

Intel Quartus Stratix V (5SEE9F45C2)

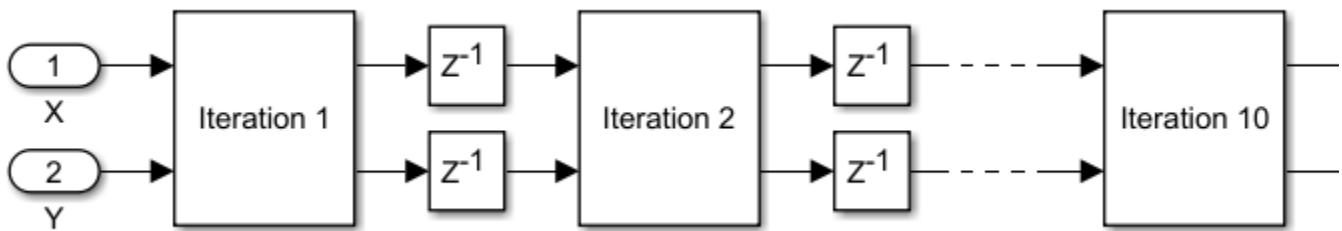
Fmax (MHz)	LABs	ALMs	Registers
460	70	292	1087

# Using ForEach Subsystems in HDL Coder

This example shows how you can use a For Each Subsystem to implement a streaming square root algorithm by cascading identical CORDIC iterations. You can then generate code for the algorithm by using HDL Coder™.

## Using CORDIC Algorithm for Hardware Functions

CORDIC is an iterative algorithm that can be used to approximate fixed-point mathematics such as trigonometric functions, square root, and divide. The iterative core is composed of simple shift and add operations, allowing the algorithm to be implemented efficiently on FPGA or ASIC hardware. For low data rate applications, a single core can be reused to perform all iterations and achieve a very small area footprint. For applications requiring a new data sample to be processed at each clock cycle, a separate core can be used to calculate each iteration in a cascaded chain, as shown in the following diagram.



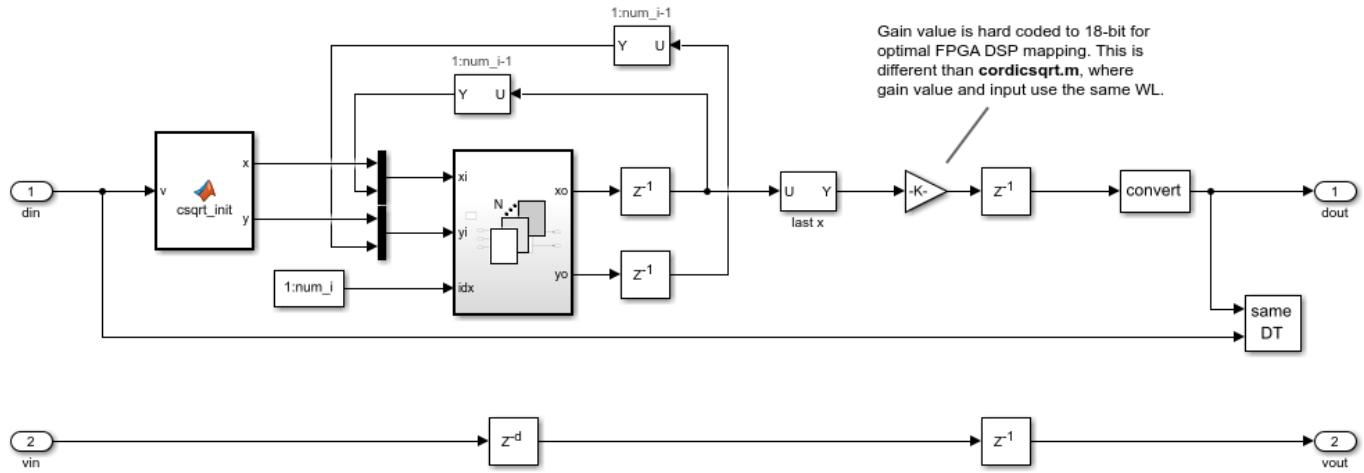
While it is straightforward to manually cascade the cores in Simulink®, the ability to automatically adjust the number of cores based on a parameter value would be highly desirable. You can do exactly that using a For Each Subsystem.

## Cascade CORDIC Iterations Using For Each Subsystem

In this model, the iterative core is placed into a For Each Subsystem to be repeated  $N$  times, where  $N$  is the number of iterations defined in the upper-level block mask. The  $N$  core outputs form a vector at the For Each Subsystem output, where they are pipelined, and then fed back into the For Each Subsystem inputs. Outputs from core  $(1:N-1)$  are connected to inputs of core  $(2:N)$ , exactly the same as in the manually cascaded model.

A valid signal path is included to handle intermittent input data, and tested by inserting random gaps between valid data samples.

```
open_system('hdlcoder_FOREACH_cordic')
open_system('hdlcoder.foreach_cordic/For Each Cordic Sqrt','force')
```



### Compare Output to CORDIC Square Root Reference

Using a 14-bit signed input in the range of  $[0.5, 2)$ , the output of the Simulink model matches the `cordicsqrt` reference function exactly. Input range outside of  $[0.5, 2)$  is not expected to work because the example lacks a normalizer stage.

In addition, the final gain adjustment in the model uses an 18-bit gain parameter for optimal FPGA DSP mapping; while the `cordicsqrt` function matches the gain parameter word length to that of the gain input. This results in slight differences between the Simulink model output and the `cordicsqrt` function when other input data types are used.

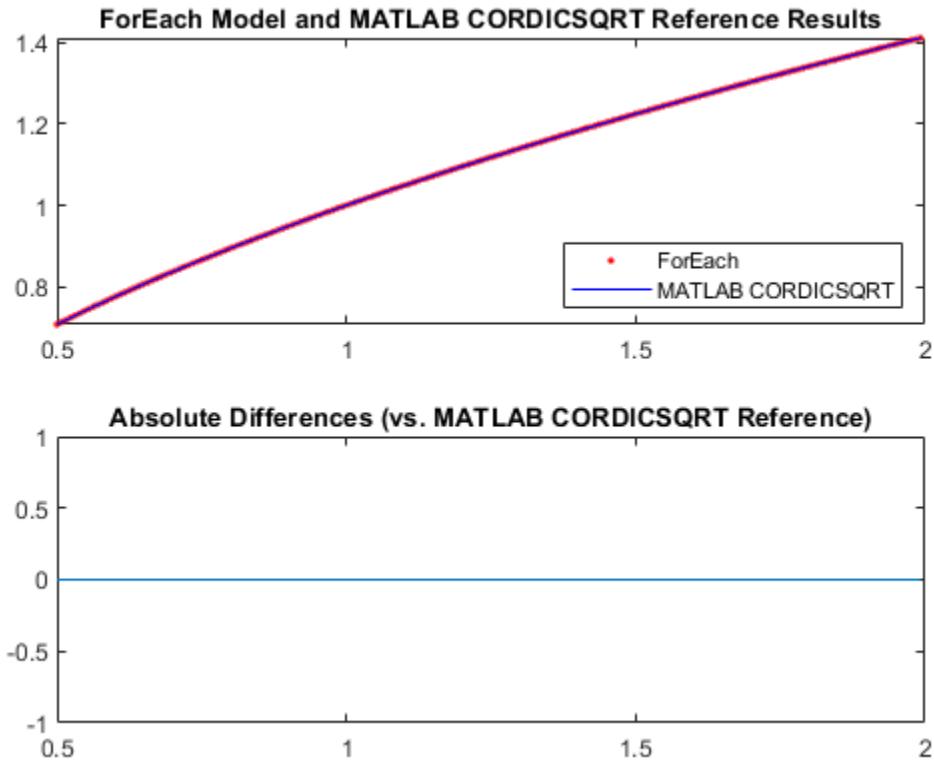
```

slout = sim('hdlcoder.foreach_cordic');
data_out = slout.logsout.getElement('data out').Values.Data;
valid_out = slout.logsout.getElement('valid out').Values.Data;
data_out = data_out(valid_out);

ref_cordic = double(cordicsqrt(v_fix, niter));
data_in = double(v_fix);
data_out = double(data_out');

figure;
subplot(211);
plot(data_in, data_out, 'r.', data_in, ref_cordic, 'b-');
legend('ForEach', 'MATLAB CORDICSQRT', 'Location', 'SouthEast');
title('ForEach Model and MATLAB CORDICSQRT Reference Results');
subplot(212);
absErr = abs(ref_cordic - data_out);
plot(data_in, absErr);
title('Absolute Differences (vs. MATLAB CORDICSQRT Reference)');

```



## Generate HDL Code

```
makehdl('hdlcoder_foreach_cordic/For Each Cordic Sqrt');

### Generating HDL for 'hdlcoder_foreach_cordic/For Each Cordic Sqrt'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_foreach_cordic')">.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_foreach_cordic'.
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/MATLAB Function2 as hdsrc\hdlcoder_foreach_cordic\For Each Cordic Sqrt\Function2.m
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/For Each Subsystem/MATLAB Function1 as hdsrc\hdlcoder_foreach_cordic\For Each Cordic Sqrt\Function1.m
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt/For Each Subsystem as hdsrc\hdlcoder_foreach_cordic\For Each Cordic Sqrt\Subsystems\For Each Subsystem.m
### Working on hdlcoder_foreach_cordic/For Each Cordic Sqrt as hdsrc\hdlcoder_foreach_cordic\For Each Cordic Sqrt\For Each Cordic Sqrt.m
### Generating package file hdsrc\hdlcoder_foreach_cordic\For_Each_Cordic_Sqrt_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdlcoder_foreach_cordic' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

## Additional Modeling Guidelines

Observe the following guidelines when cascading blocks in your algorithm using For Each Subsystem:

- Since For Each Subsystem is atomic, the connection between output of block X and input of block X+1 creates an artificial algebraic loop. To break this loop, place pipeline registers between cascading blocks outside of the For Each Subsystem, as demonstrated in this example.
- A mux block is used to concatenate external input and outputs of block (1:N-1) to form the inputs of the For Each Subsystem. This requires the cascading blocks to use the same input and output data types.

**Related Topics**

- “Generate HDL Code for Blocks Inside For Each Subsystem” on page 10-51
- “Compute Square Root Using CORDIC”

# Generate HDL Code for Blocks Inside For Each Subsystem

This example shows how to use blocks inside a For Each Subsystem in your Simulink™ model, and then generate HDL code.

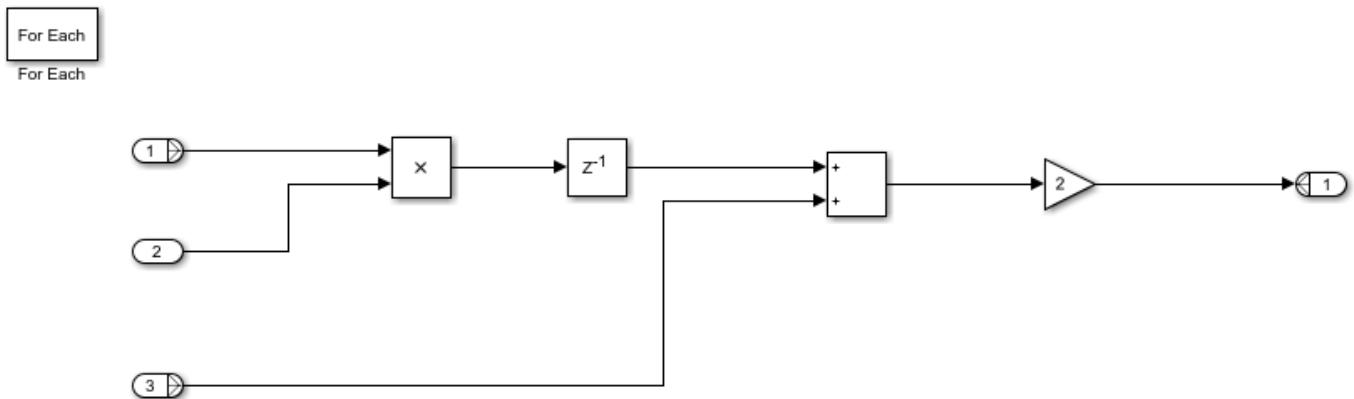
## Why Use a For Each Subsystem?

To repeatedly perform the same algorithm on individual elements or subarrays of the input signals, use the For Each Subsystem block. The set of blocks within the Subsystem replicate the algorithm that is applied to individual elements or equally divided subarrays of the input signals. Using the For Each Subsystem block, you do not have to create and connect replicas of a Subsystem block to model the same algorithm. The For Each Subsystem:

- Supports vector processing, which reduces the simulation time of your model. You can process individual elements or subarrays of an input signal simultaneously.
- Improves code readability by using a for-generate loop in the generated HDL code. The for-generate loop reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.
- Supports HDL code generation for all data types, Simulink™ blocks, and predefined and user-defined system objects.
- Supports optimizations on and inside the block, such as resource sharing and pipelining. The parallel processing capability of the For Each Subsystem block combined with the optimizations that you specify produces high performance on the target FPGA device.

## Modeling With the For Each Subsystem

Open the `foreach_subsystem_example1` model. You see this simple algorithm modeled inside a For Each Subsystem block.

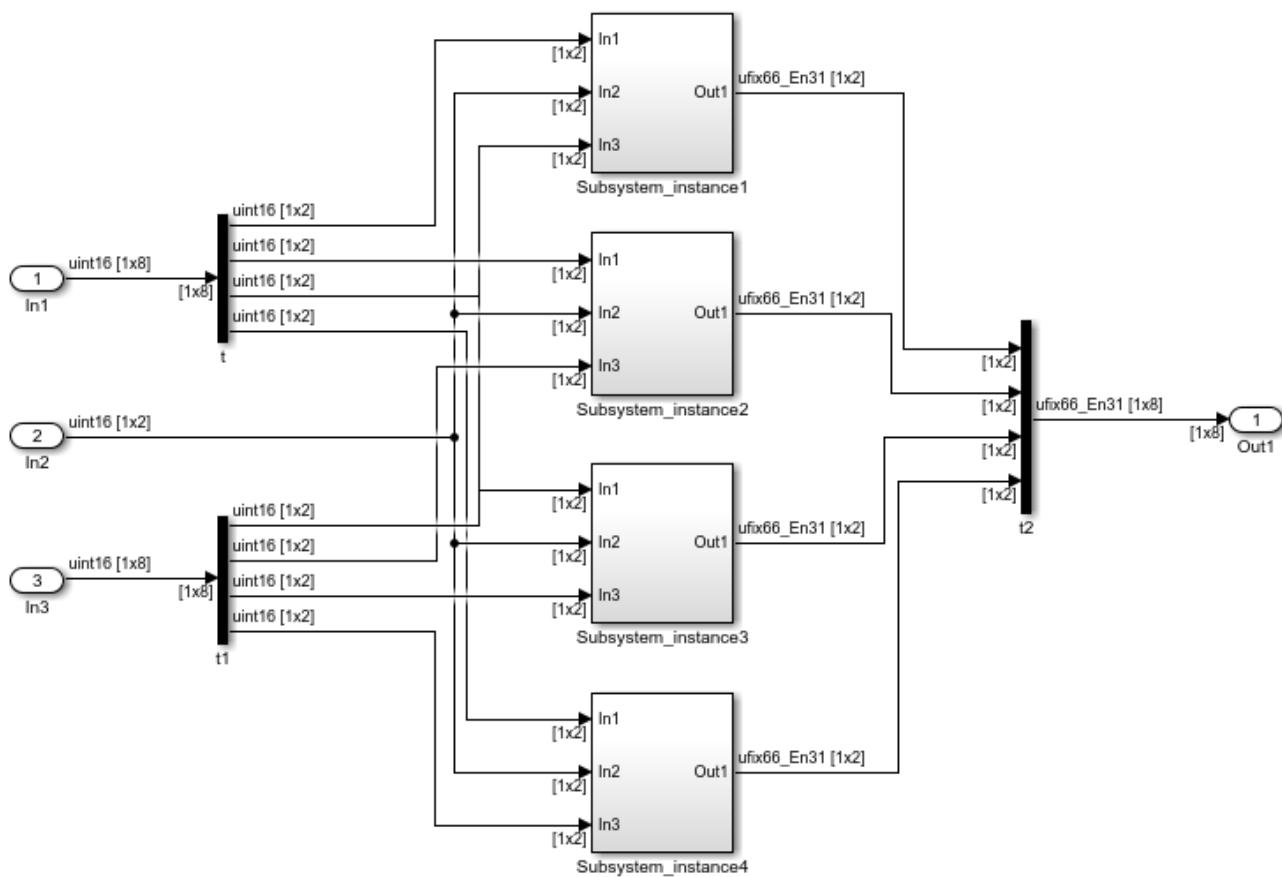


When you simulate the model, you see that the input signals `In1` and `In3` are partitioned into subarrays. To see this partitioning, double-click the For Each block. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which the input signal is partitioned and the width of each partition slice respectively. Based on the input signal sizes and the partitioning that you specify, the For Each Subsystem determines the number of iterations that it requires to compute the algorithm.

In this example, the input signals `In1` and `In3` of size 8 are partitioned into four subarrays, each of size 2. The input signal `In2` of size 2 is not partitioned. To compute the algorithm, the For Each

Subsystem requires four iterations, with each iteration repeating the algorithm on each of the four subarrays of In1 and In3.

The For Each Subsystem simplifies modeling of vectorized algorithms. This figure shows how you can model the same algorithm by creating multiple subsystem instances. This model can become graphically complex and difficult to maintain.



### Using Complex Data Signals

The block does not support complex data types as inputs for HDL code generation. To input a complex signal, you can convert this signal to an array of signals, and then input to the block.

To perform the same algorithm on both real and imaginary parts of the signal:

- 1 Separate the signal into real and imaginary parts by using a Complex to Real-Imag block.
- 2 Create a vector signal that consists of the real and imaginary parts by using a Mux block.

You can then input this vector to the For Each Subsystem block and replicate the same computation on both the real and imaginary parts. At the output of the For Each Subsystem, you can convert the vector output back to a complex signal. Use a Demux block to separate the real and imaginary scalar parts, and then input the scalars to the Real-Imag to Complex block.

## Generate HDL Code

To generate HDL code, in the `foreach_subsystem_example1` model, right-click the `Subsystem_Foreach` block and select **HDL Code > Generate HDL for Subsystem**.

To see the generated HDL code for the `Subsystem_Foreach` block, in the MATLAB™ Command Window, click the `Subsystem_Foreach.vhd` file. In the VHDL code snippet, you see this for-generate loop in the HDL code. This loop creates four subsystem instances, with each instance performing the algorithm on size 2 subarrays of inputs `In1` and `In3`.

```
BEGIN
  -- <S2>/For Each Subsystem
  GEN_LABEL: FOR k IN 0 TO 3 GENERATE
    u_For_Each_Subsystem : For_Each_Subsystem
      PORT MAP( clk => clk,
                 reset => reset,
                 enb => clk_enable,
                 In1 => In1(2*k TO 2*(k+1) - 1), -- uint16 [2]
                 In2 => In2, -- uint16 [2]
                 In3 => In3(2*k TO 2*(k+1) - 1), -- uint16 [2]
                 Out1 => For_Each_Subsystem_out1(2*k TO 2*(k+1) - 1)
      );
  END GENERATE;
```

Certain optimizations that you specify can change the contents of the subsystems that the For Each Subsystem instantiates. In such cases, the code generator does not use for-generate loops in the HDL code. The HDL code does not contain for-generate loops, if you have:

- Bus or complex input signals.
- Certain optimizations enabled on the subsystem, such as resource sharing and streaming.
- Vector inputs that get partitioned into nonscalar signals in the Verilog code. To obtain for-generate loops in the Verilog code, partition the vector signal to scalars.

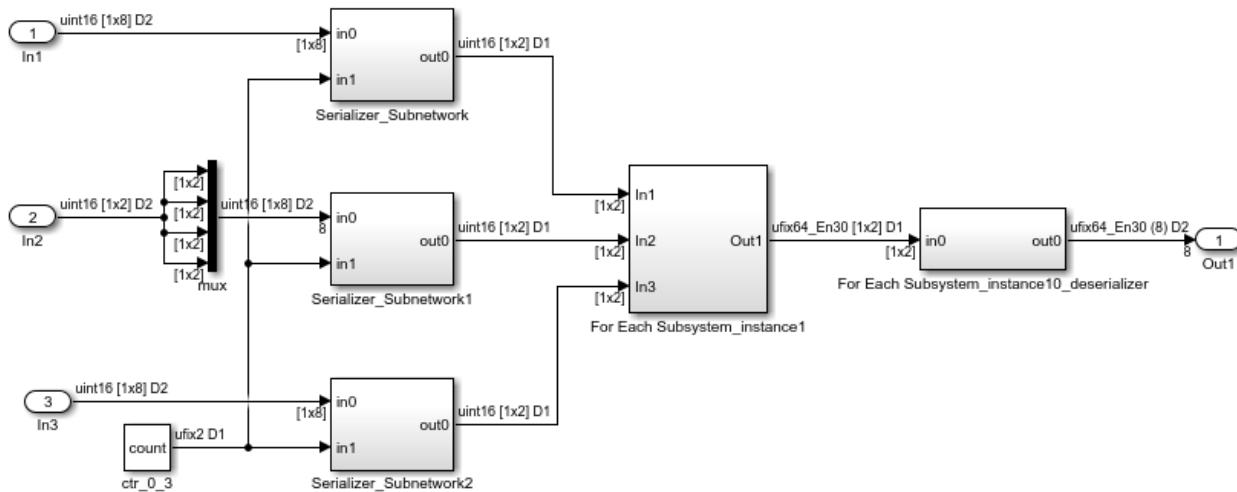
## Optimize the For Each Subsystem Algorithm

To optimize the algorithm contained within the For Each Subsystem, you can enable optimizations such as resource sharing and streaming on the DUT that contains the For Each Subsystem. For example, by using the resource sharing optimization, you can share multiple Subsystem instances that are created by the For Each Subsystem. This optimization reuses the algorithm modeled by the Subsystem across multiple instances and reduces the area usage on the target device.

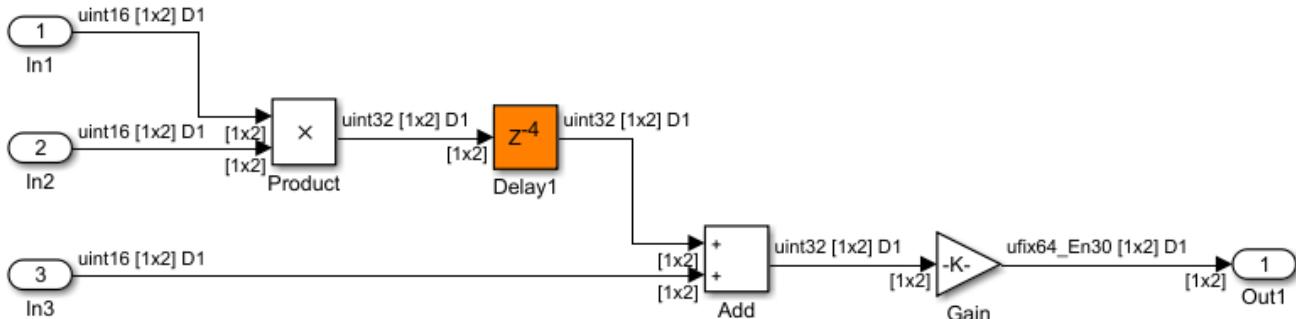
**Note:** When you enable optimizations on the For Each Subsystem, the generated HDL code does not contain for-generate loops.

This example shows how to use the resource sharing optimization on the For Each Subsystem. To share resources, select the Subsystem block that contains the For Each Subsystem and then specify the **Sharing Factor**. In this example, right-click the `Subsystem_Foreach` block and select **HDL Code > HDL Block Properties**. Set the **Sharing Factor** to 4, because the For Each Subsystem generates four Subsystem instances. Then, generate HDL code for the `Subsystem_Foreach` block.

To see the effect of the resource sharing optimization, at the command-line, enter `gm_foreach_subsystem_example1` to open the generated model. In the generated model, you see that the optimization shared the four subsystem instances generated by the For Each Subsystem into one Subsystem For Each Subsystem\_Instance1.



If you double-click the `For Each Subsystem_Instance1` block, you see the algorithm computed for the size 2 subarrays of inputs `In1` and `In3`.



To learn more about the resource sharing optimization, see Resource Sharing.

## See Also

For Each Subsystem

# Field-Oriented Control of a Permanent Magnet Synchronous Machine

In this example you will review a Field-Oriented Control (FOC) algorithm for a Permanent Magnet Synchronous Machine (PMSM). You will test the control algorithm with closed loop system simulation then generate HDL code for the control algorithm. You will also see how tunable parameter data is specified and how corresponding HDL port entities are generated.

## Introduction

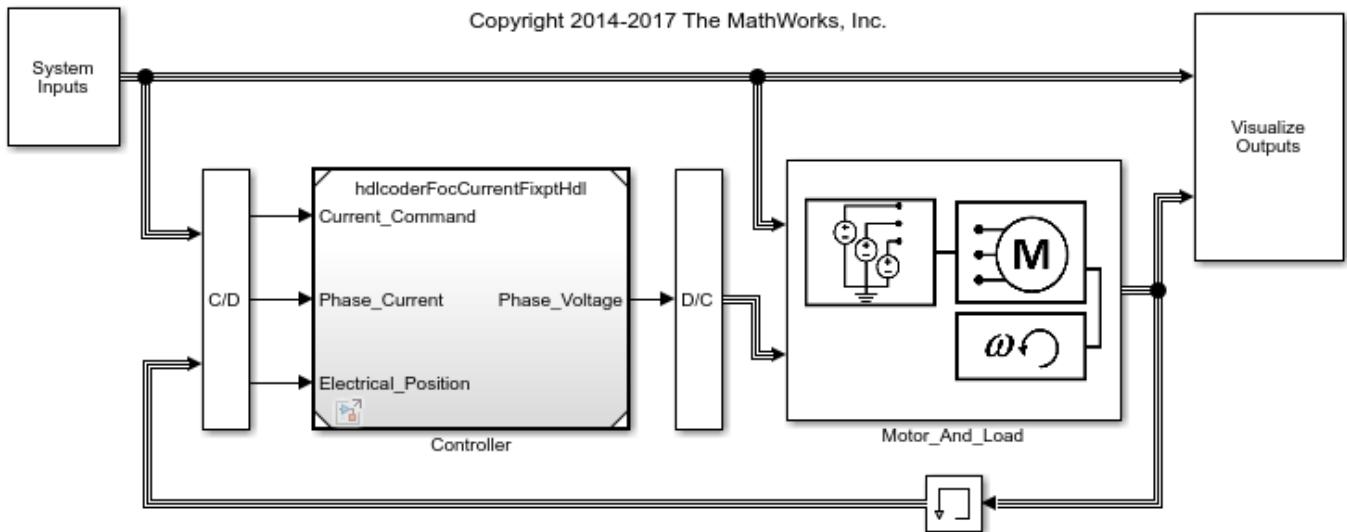
The example is partitioned such that you can generate code for the control algorithm as well as verify the behavior of the control algorithm using a simulation test bench. Simscape (TM) Electrical (TM) is required to run the system simulation test bench model `hdlcoderFocCurrentTestBench.slx` but is not required to generate code from the control algorithm model `hdlcoderFocCurrentFixptHdl.slx`.

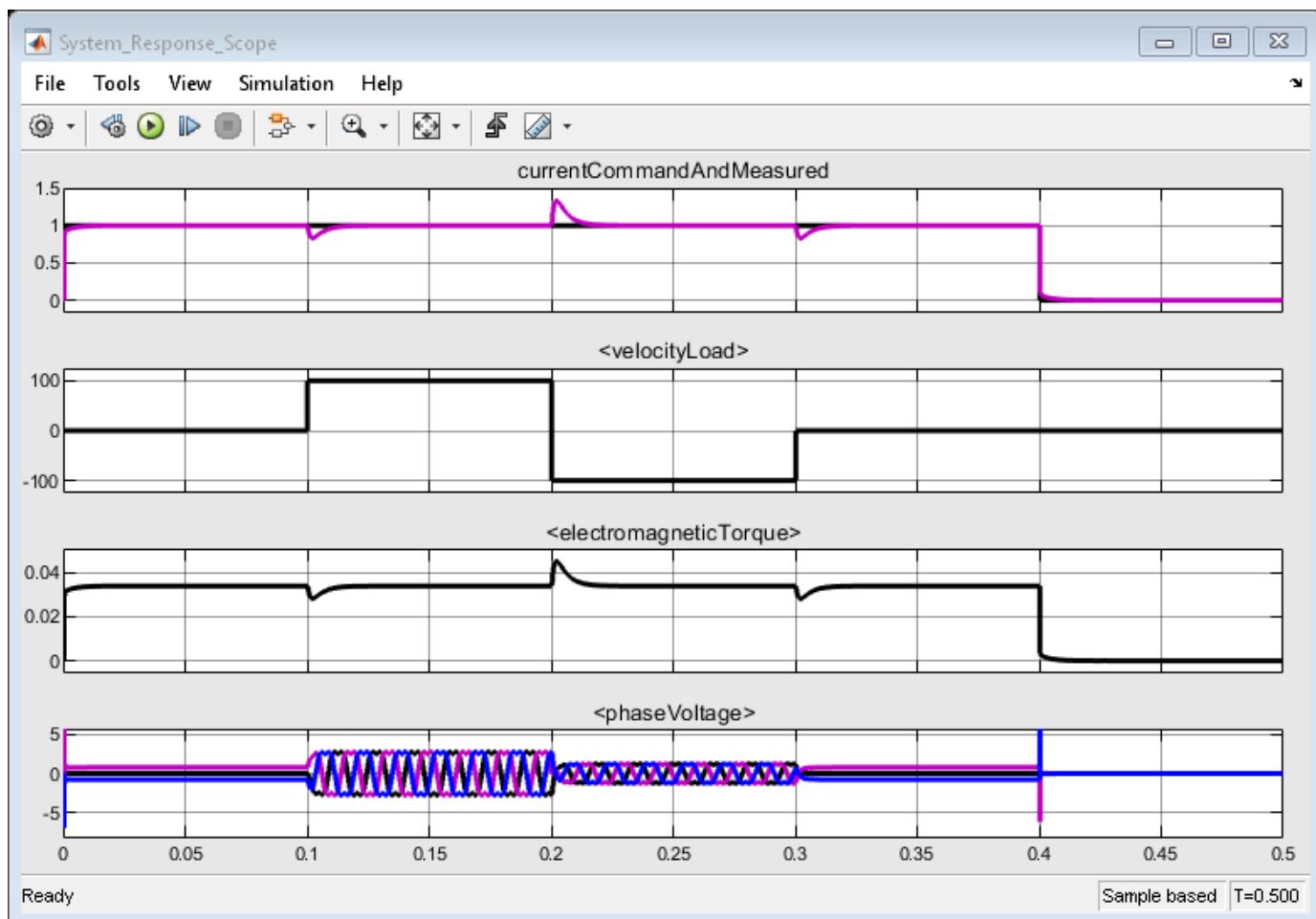
## Verify Behavior through Simulation

In this example FOC is used to regulate phase current to control torque of an electric machine. You can simulate a test bench to explore the behavior of the system. During the simulation, the solver may generate warnings related to zero crossing when the velocity load changes abruptly. You can disable these warnings during the simulation.

```
hasSimPowerSystems = license ('test', 'Power_System_Blocks');
if hasSimPowerSystems
    open_system('hdlcoderFocCurrentTestBench')
    set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','none');
    sim('hdlcoderFocCurrentTestBench')
    set_param('hdlcoderFocCurrentTestBench','IgnoredZcDiagnostic','warn');
end
```

## Field-Oriented Control Current Control Test Bench



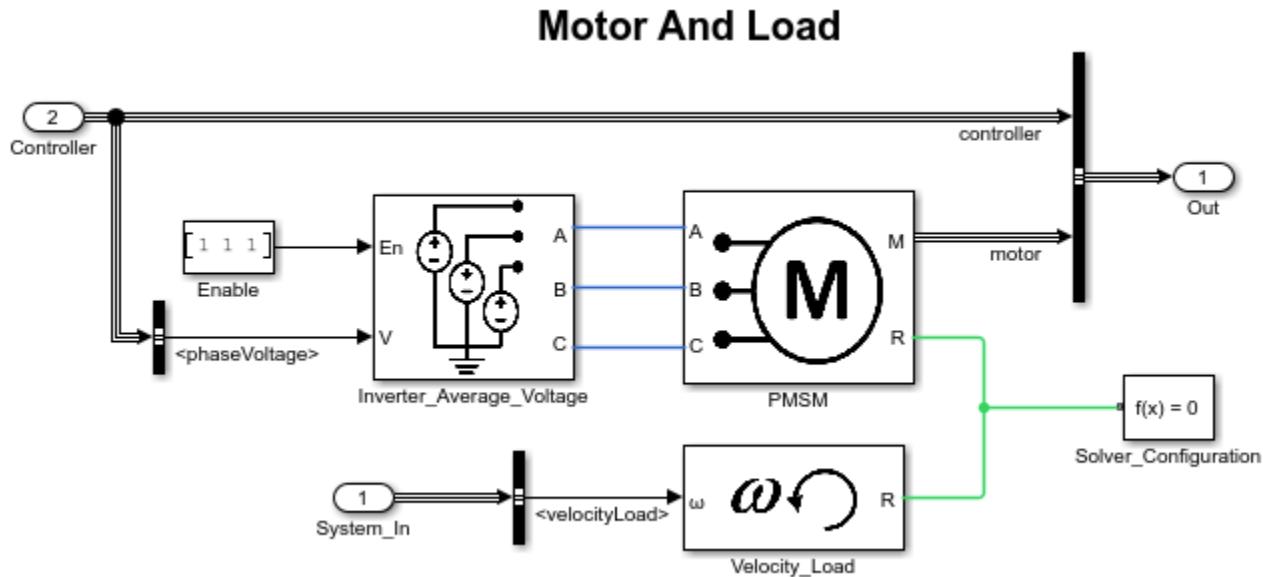


The scope shows that a 1 Amp step current command is requested and the load velocity changes between locked rotor (zero), +100 rad/sec, and -100 rad/sec. The current command represents a quadrature current command to a non-salient PMSM. (The controller regulates the direct current to zero.) Note that for this motor and controller, the electromagnetic torque closely follows the measured quadrature current of the motor.

### Explore Plant Specification

In the Motor\_And\_Load subsystem you will see a mathematical model of the components being controlled. An average model of the inverter is used to drive a constant parameter dq voltage equation model of a PMSM which is connected to a velocity load.

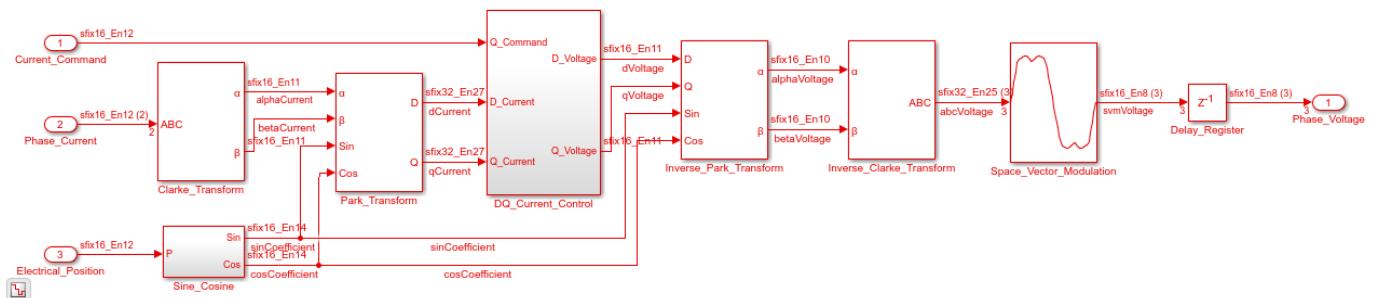
```
if hasSimPowerSystems
    open_system('hdlcoderFocCurrentTestBench/Motor_And_Load')
end
```



### Explore Control Algorithm Specification

The FOC current control algorithm is specified in a separate model. In the control algorithm, the electrical equations of the machine are projected from the three-phase stationary reference frame onto a two phase rotating reference frame using Clarke and Park transforms. This simplifies the control by removing time and position dependencies. Space Vector Modulation enables the controller to achieve greater voltage across the phases than if just the sinusoidal outputs of the inverse Clarke transform were used.

```
load_system('hdlcoderFocCurrentFixptHdl');
open_system('hdlcoderFocCurrentFixptHdl/FOC_Current_Control')
```



### Explore Data Specification

Both the controller and the plant (i.e. motor and load) reference data from the MATLAB ® workspace. A data definition file creates this data and is automatically run within the PreLoadFcn callback of the system test bench model.

```
edit('hdlcoderFocCurrentFixptHdlData.m')
```

When you review this file, notice that the parameters paramCurrentControlP and paramCurrentControlI are specified as Simulink.Parameters whose storage class is set to

'ExportedGlobal'. This tells HDL Coder to generate entity ports for these parameters instead of constant values.

### Generate HDL Code for Control Algorithm

Before generating HDL code, it is important to ensure that the model adheres to certain important settings for HDL code generation. Below are some of the main steps:

- Create a DUT subsystem: For HDL code generation it is always better to create a DUT (Design Under Test) subsystem from which HDL code is generated. This subsystem serves several purposes including being a place-holder for HDL optimization settings.
- Setup for HDL: In order to get ready for HDL code generation, certain solver settings and model settings must be in place. The `hdlsetup` command takes care of all these settings and should be run before HDL code-generation.
- Checking sample times: Applying HDL optimizations requires all block sample times to be inferred as discrete. The main block-type to be cautious of are constants, which derive an 'inf' sample time, by default. We can find these blocks and explicitly set their sample-times to '-1' so they get the correct back-propagated sample times.

```
% You can generate and review the HDL code for the controller.
makehdl('hdlcoderFocCurrentFixptHdl/FOC_Current_Control');

### Generating HDL for 'hdlcoderFocCurrentFixptHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurrentFixptHdl')>.
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocCurrentFixptHdl')>.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFixptHdl'.
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoder...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control as hdlsrc\hdlcoder...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc\...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\h...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderF...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdl...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocC...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Space_Vector_Modulation/Max/Max as ...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Space_Vector_Modulation/Min/Min as ...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control/Space_Vector_Modulation as hdlsrc\h...
### Working on hdlcoderFocCurrentFixptHdl/FOC_Current_Control as hdlsrc\hdlcoderFocCurrentFixptH...
### Generating package file hdlsrc\hdlcoderFocCurrentFixptHdl\FOC_Current_Control_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tph')>.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tph...
### HDL check for 'hdlcoderFocCurrentFixptHdl' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Notice in the generated **hdlcoderFocCurrentFixptHdl.vhd** file that the entity has ports for `paramCurrentControlP` and `paramCurrentControll`.

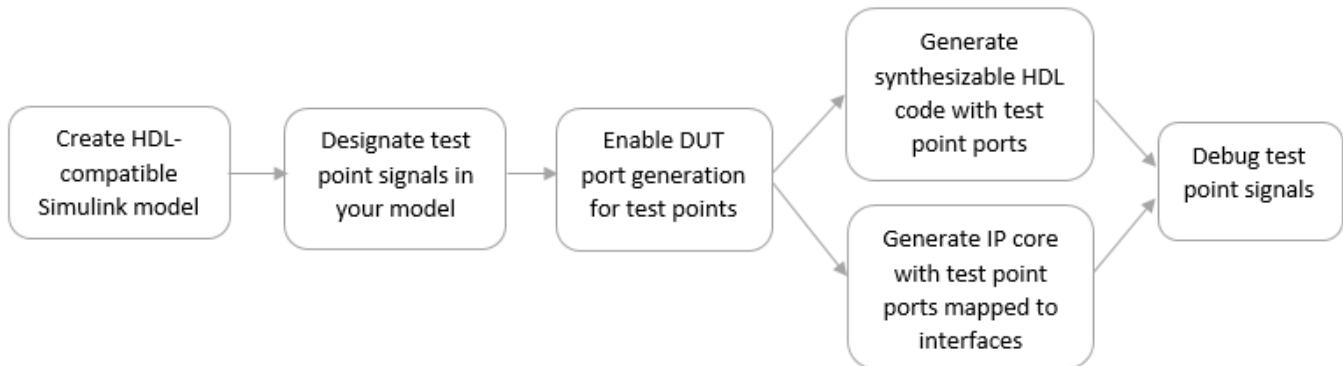
# Model and Debug Test Point Signals with HDL Coder

This example shows how you can mark signals as test points in your Simulink™ model and, after HDL code generation, debug the signals at the top level using the generated model or a test bench.

## Why Use Test Points?

Test points are signals that you can use to easily debug and observe the simulation results at various points in your Simulink™ model. You can observe signals designated as test points with a Floating Scope block in a model. In Simulink™, you can designate any signal in a model as a test point.

After code generation, you can observe test point signals at the DUT output ports and further debug the generated code in downstream workflows. This capability makes debugging your design easier because the code generator can propagate test point signals deep within the subsystem hierarchy to the DUT output ports.



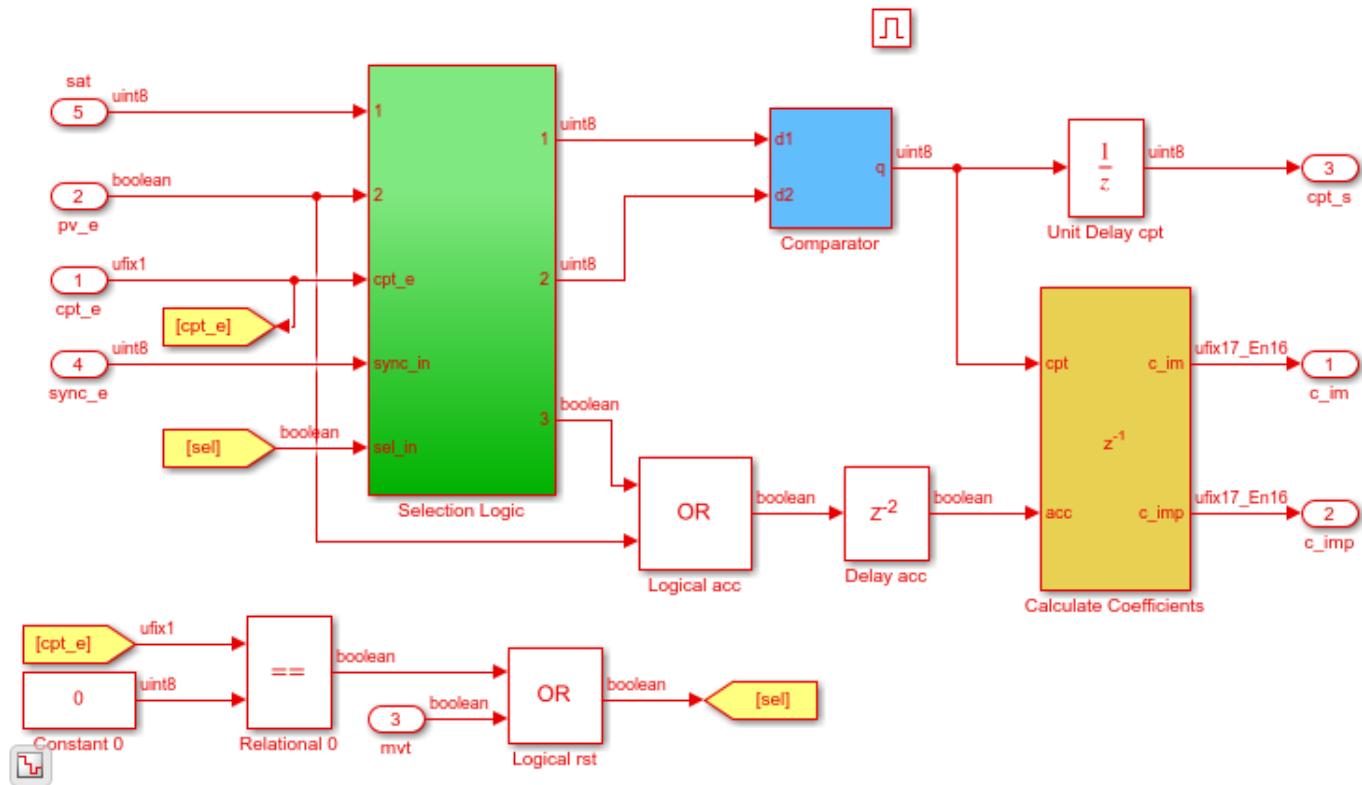
## Create HDL-Compatible Model

Before you designate signals as test points and generate HDL code, make sure that the model you create is compatible for HDL code generation. See “Create HDL-Compatible Simulink Model”.

For this example, open the `hdlcoder_simulink_test_points` model that has been prepared for HDL code generation. The DUT is an Enabled Subsystem that calculates two coefficients based on inputs from a Selection Logic and a Comparator.

```

load_system('hdlcoder_test_points')
open_system('hdlcoder_test_points/DUT/MaJ Counter')
set_param('hdlcoder_test_points', 'SimulationCommand', 'update');
  
```



### Designate Signals as Test Points

To debug internal signals in this model, mark them as test points in either of these ways:

- In the Simulink Editor, to open the Signal Properties dialog box, right-click the signal, and select **Properties**. Then, select **Test Point**.
- At the command line, get the handle to the output port of a block, and then set the port parameter **TestPoint** to 'on'.

For example, enter these commands to designate the output signal from the Logical acc block that performs the OR operation as a test point.

```
portHandles = get_param('hdlcoder_test_points/DUT/MaJ Counter/Logical acc', 'portHandles');
outportHandle = portHandles.Outport;
set_param(outportHandle, 'TestPoint', 'on');
```

If a block has more than one output port, specify the outport handle that you want to designate as testpoint. For example, to designate the second output of a Demux block as a test point, enter this command:

```
set_param(outportHandle(2), 'TestPoint', 'on');
```

Simulink™ displays an indicator on each signal for which you enable the **Test point** setting. If you navigate the model, you see three additional test points. These test points are inside the Selection Logic subsystem, the Comparator Subsystem, and the Calculate Coefficients Subsystem blocks.

To learn more, see “Configure Signals as Test Points”.

## Enable DUT Output Port Generation for Test Points

Before you generate HDL code, to debug signals that are designated as test points, enable HDL DUT port generation for the signals. When you generate code for the model, HDL Coder™ propagates these signals to the DUT as an additional output port.

To enable DUT output port generation for the `hdlcoder_simulink_test_points` model:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Ports** tab, select **Enable HDL DUT port generation for test points**.
- At the command line, use the `EnableTestpoints` property.

```
hdlset_param('hdlcoder_test_points','EnableTestpoints','on')
```

To learn more about this parameter, see “Enable HDL DUT port generation for test points” on page 17-41.

After you enable DUT port generation, you can run either of these workflows:

- Generate HDL code. To deploy the code onto a target FPGA, use the **Generic ASIC/FPGA** workflow in the HDL Workflow Advisor.
- Map test point ports to target platform interfaces, and generate an HDL IP core by using the **IP Core Generation** or **Simulink Real-Time FPGA I/O** workflows that use Xilinx Vivado or Altera Quartus II as the synthesis tools.

## HDL Code Generation and FPGA Targeting

If you want to see the mapping between test point ports in the HDL code and the test point signals in your model, enable generation of the code generation report. The report displays the test point ports with links to the corresponding test point signals in your Simulink™ model.

For example, to enable report generation for the `hdlcoder_simulink_test_points` model:

- In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate resource utilization report**.
- To specify this setting at the command line, use the `ResourceReport` property.

```
hdlset_param('hdlcoder_test_points','ResourceReport','on')
```

To learn more about report generation, see “Create and Use Code Generation Reports” on page 25-2.

To generate HDL code:

- Right-click the DUT Subsystem and select **HDL Code > Generate HDL for Subsystem**.
- At the command line, run `makehdl` on the DUT Subsystem.

To deploy the code onto a target platform, use the **Generic ASIC/FPGA** workflow. In the HDL Workflow Advisor, on the **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **Generic ASIC/FPGA**, specify the **Synthesis tool**, and then run the workflow.

When generating code, HDL Coder opens the Code Generation report. The Code Interface Report section contains links to the test point ports in the **Output Ports** section.

## Output ports

Port Name	Datatype	Bits
ce_out	boolean	1
<a href="#">c_im</a>	ufix17_En16	17
<a href="#">c_imp</a>	ufix17_En16	17
<a href="#">cpt_s</a>	uint8	8
<a href="#">tp_Sum_C_out1</a>	ufix17_En16	17
<a href="#">tp_Relational_out1</a>	boolean	1
<a href="#">tp_Sum_out1</a>	uint8	8
<a href="#">tp_Logical_acc_out1</a>	boolean	1
<a href="#">tp_Relational_0_out1</a>	boolean	1

When you click the links in the test point ports, the code generator highlights the corresponding signals that you designated as test points in your Simulink™ model. Therefore, you can use the report to trace back from the test point port in the generated code to the test point signals in your Simulink™ model.

To see the test point ports in the generated HDL code, open the DUT.v file.

```
output [16:0] c_im; // ufix17_En16
output [7:0] cpt_s; // uint8
output [16:0] tp_Sum_C_out1; // ufix17_En16 Testpoint port
output tp_Relational_out1; // Testpoint port
output [7:0] tp_Sum_out1; // uint8 Testpoint port
output tp_Logical_acc_out1; // Testpoint port
output tp_Relational_0_out1; // Testpoint port
```

You can see the test point ports at the top level module declaration. These ports have the prefix tp\_ and a comment to indicate that they correspond to test point ports. If you specify VHDL as the target language, you can see the test point ports in the entity declaration.

## IP Core Generation and SoC Targeting

To generate an HDL IP core, open the HDL Workflow Advisor. In the Advisor:

- 1 On the **Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**, and specify a **Target platform** that uses Xilinx Vivado or Altera Quartus II as the **Synthesis tool**. If you use the Simulink Real-Time FPGA I/O workflow, specify a **Target platform** that uses Xilinx Vivado as the **Synthesis tool**
- 2 On the **Set Target Reference Design** task, you can specify the HDL Coder default reference designs, or a custom reference design that you want to integrate the HDL IP core into. If you do not specify unique names for test point signals, running this task can fail. To fix this error, in the **Result** subpane, select the link to generate unique names for test point signals. To verify that the task passes, rerun the task.
- 3 On the **Set Target Interface** task, you see the test point ports in the **Target platform interface table**. You can map the ports to AXI4, AXI4-Lite, or External Port interfaces. After you run this

task, the code generator stores this testpoint interface mapping information on the DUT. To see this information, in the HDL Block Properties for the DUT Subsystem, on the **Target Specification** tab, look for the **TestPointMapping** block property. You can reload this information for the DUT across subsequent runs of the workflow.

- 4 On the **Generate RTL Code and IP Core task**, right-click and select **Run to Selected Task** to generate the IP core. The code generator opens an IP Core Generation report that displays the mapping of test point ports to interfaces.

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
<a href="#">cpt_e</a>	Import	ufix1	AXI4-Lite	x"100"
<a href="#">pv_e</a>	Import	boolean	AXI4-Lite	x"104"
<a href="#">mvt</a>	Import	boolean	AXI4-Lite	x"108"
<a href="#">sync_e</a>	Import	uint8	AXI4-Lite	x"110"
<a href="#">sat</a>	Import	uint8	DIP Switches [0:7]	[0:7]
<a href="#">Enable_In</a>	Import	boolean	AXI4-Lite	x"11C"
<a href="#">c_im</a>	Outport	ufix17_En16	External Port	
<a href="#">c_imp</a>	Outport	ufix17_En16	External Port	
<a href="#">cpt_s</a>	Outport	uint8	AXI4-Lite	x"10C"
<a href="#">TestPoint_3</a>	Test point	boolean	AXI4-Lite	x"114"
<a href="#">TestPoint</a>	Test point	ufix17_En16	External Port	
<a href="#">TestPoint_1</a>	Test point	boolean	AXI4-Lite	x"118"
<a href="#">TestPoint_2</a>	Test point	uint8	LEDs General Purpose [0:7]	[0:7]

When you click the links in the test point ports, the code generator highlights the corresponding signals that you designated as test points in your Simulink™ model.

If you open the generated HDL source file, you see the test point signals connected to the IP core wrapper.

```

MaJCounte_ip_axi_lite u_MaJCounte_ip_axi_lite_inst (...  
...  
    .read_TestPoint_3(tp_TestPoint_3_sig), // ufix1  
    .read_TestPoint_1(tp_TestPoint_1_sig), // ufix1  
    ...  
    ...  
);  
  
MaJCounte_ip_dut u_MaJCounte_ip_dut_inst (...  
...  
    .tp_TestPoint(tp_TestPoint_sig), // ufix17_En16  
    .tp_TestPoint_1(tp_TestPoint_1_sig), // ufix1  
    .tp_TestPoint_2(tp_TestPoint_2_sig), // ufix8  
    .tp_TestPoint_3(tp_TestPoint_3_sig) // ufix1  
);
```

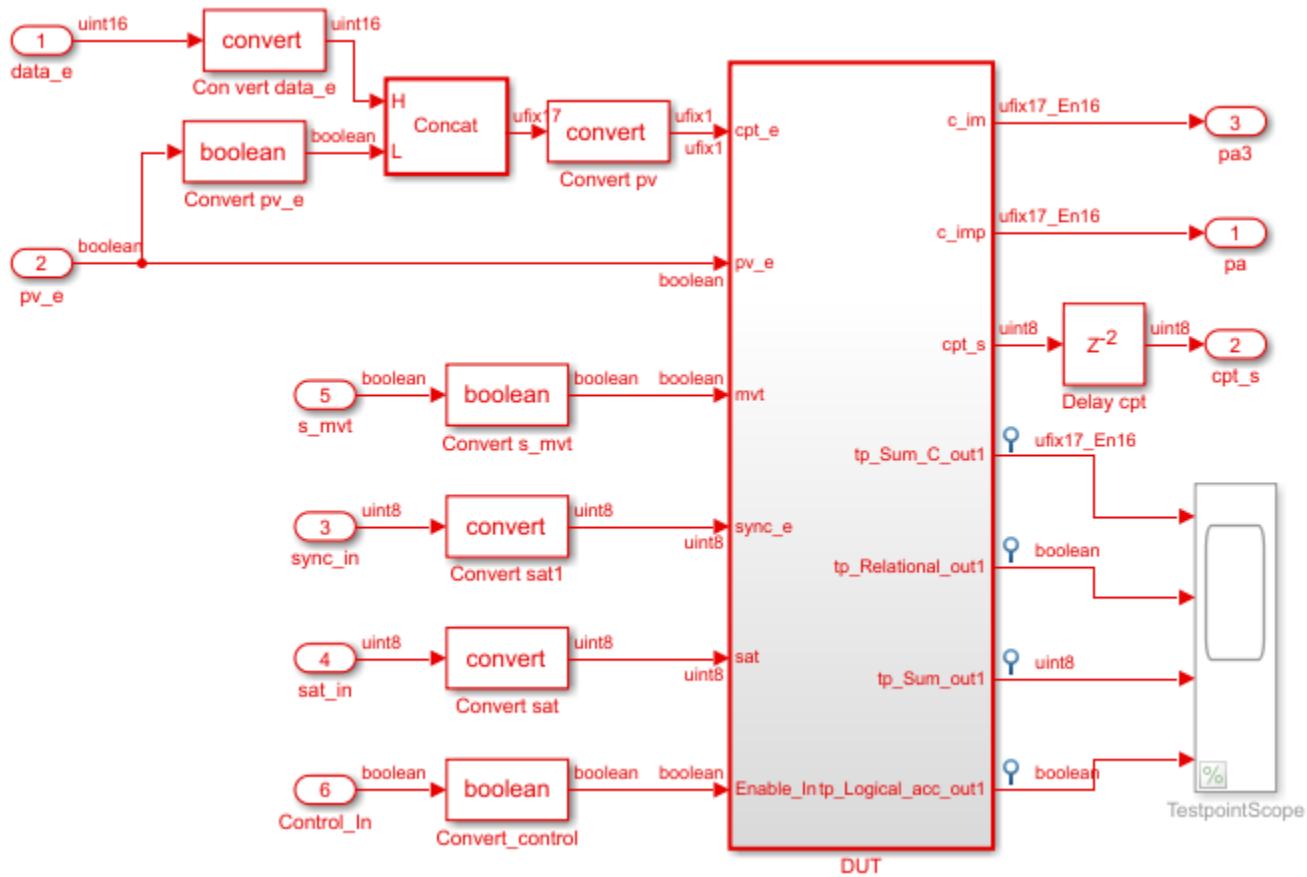
Run the workflow to generate the Software Interface model and integrate the IP core into the target reference design that you specified in the **Set Target Reference Design** task.

To learn more about the IP Core Generation workflow, see “Custom IP Core Generation” on page 40-10 and “Custom IP Core Report” on page 40-13.

### Debug Test Point Signals

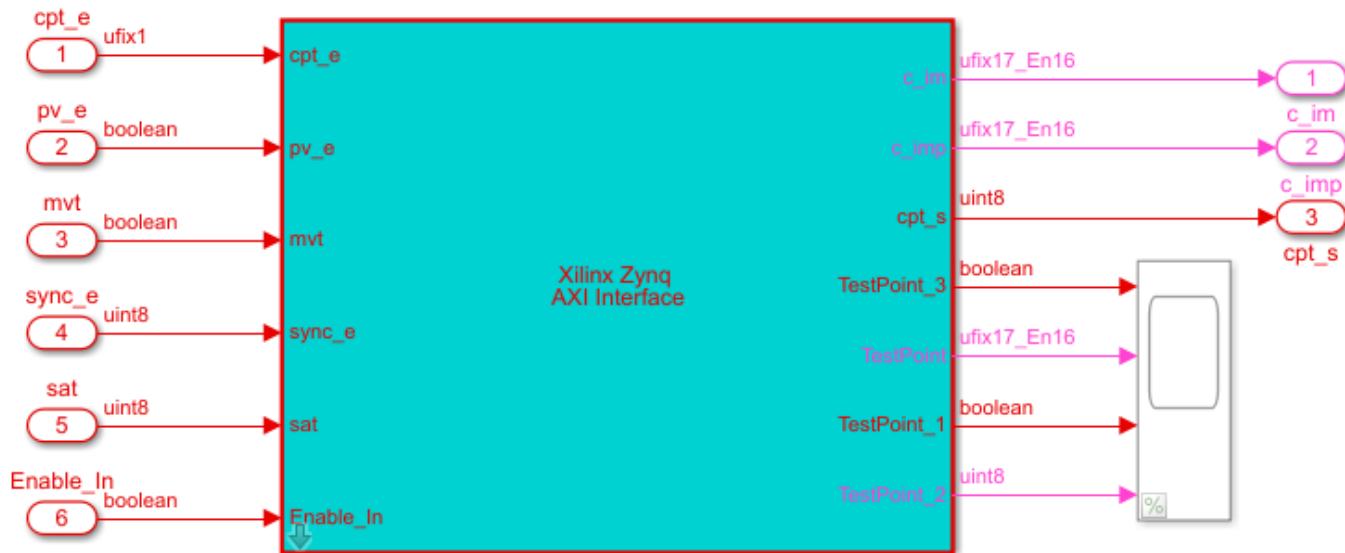
After you generate HDL code or generate an IP core, you can debug the test point signals.

If you generated HDL code for your model or ran the Generic ASIC/FPGA workflow, to debug the test point signals, generate a HDL test bench, or use the generated model. To open the generated model, at the command line, enter `gm_hdlcoder_test_points`.



In the generated model, you see the test points at the DUT output ports connected to a Scope block that is commented out. To observe the simulation results for these signals, uncomment the Scope block, and then run the simulation. If you navigate the generated model, you can see that the code generator creates an output port at the point where you designated the signal as a test point. HDL Coder then propagates these ports to the DUT as additional output ports.

If you run the IP Core Generation workflow to the **Generate Software Interface Model** task, the code generator opens the Software Interface model.



To observe the data on test point signals from the ARM processor, uncomment the Scope block, and then run the Software Interface model.

## Considerations

- Test point ports are considered similar to other output ports in the code generation process. Test point port generation works with all optimizations such as resource sharing, streaming, and distributed pipelining. To learn about various optimizations, see “Speed and Area Optimization”.
- If you generate a validation model, you see that the code generator does not compare the test point signals with the test point ports at the output. You can still observe the test point signals by uncommenting the Scope block and by running the simulation. To learn more about generated model and validation model, see “Generated Model and Validation Model” on page 24-10.
- If you generate a cosimulation model, you see the test point ports connected to a Terminator block. To observe the test points, remove the Terminator blocks, and connect the output ports to a Scope block, and then run cosimulation. You can also observe the waveforms in the HDL simulator that you run cosimulation with. To learn more about cosimulation, see “Generate a Cosimulation Model” on page 27-41.
- If you open the generated model, you see the Scope block commented out for performance considerations.
- You cannot specify the port ordering for the DUT test point ports. HDL Coder™ determines the port ordering when you generate code.
- **Target workflow** must be Generic ASIC/FPGA, IP Core Generation, or Simulink Real-Time FPGA I/O.
- If you use IP Core Generation or Simulink Real-Time FPGA I/O workflows, the **Synthesis tool** must be Xilinx Vivado or Altera Quartus II. Xilinx ISE is not supported.

- If you use IP Core Generation or Simulink Real-Time FPGA I/O workflows, you map the test point ports to AXI4, AXI4-Lite, or External Port interfaces. You cannot map the ports to AXI4-Stream or AXI4-Stream Video interfaces.

## See Also

### More About

- “Configure Signals as Test Points”
- “Enable HDL DUT port generation for test points” on page 17-41
- “Generated Model and Validation Model” on page 24-10

# Allocate Sufficient Delays for Floating-Point Operations

## In this section...

"Problem" on page 10-67

"Cause" on page 10-67

"Solution" on page 10-68

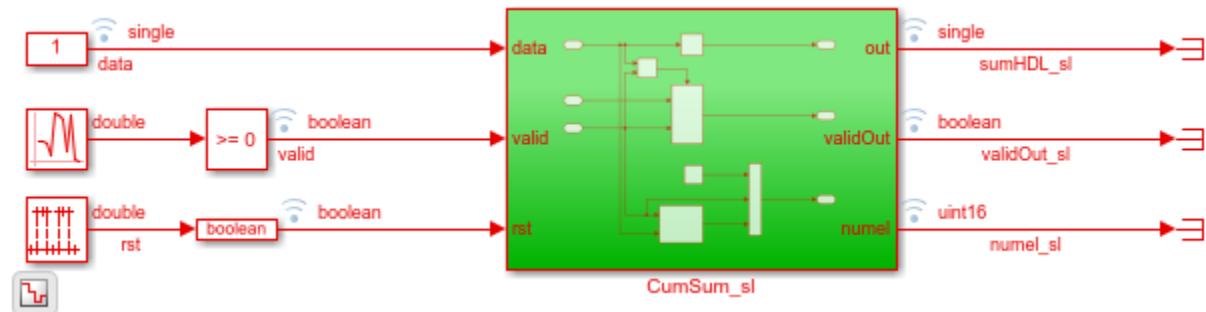
## Problem

Sometimes, when generating code from your floating-point algorithm in Simulink, HDL Coder generates an error that it is unable to allocate sufficient number of delays.

## Cause

This error message generally occurs when you have Simulink™ blocks performing floating-point operations inside a feedback loop. These blocks have a latency. HDL Coder™ is unable to allocate delays to compensate for the latency, because the code generator needs to add delays and balance them to maintain numerical accuracy.

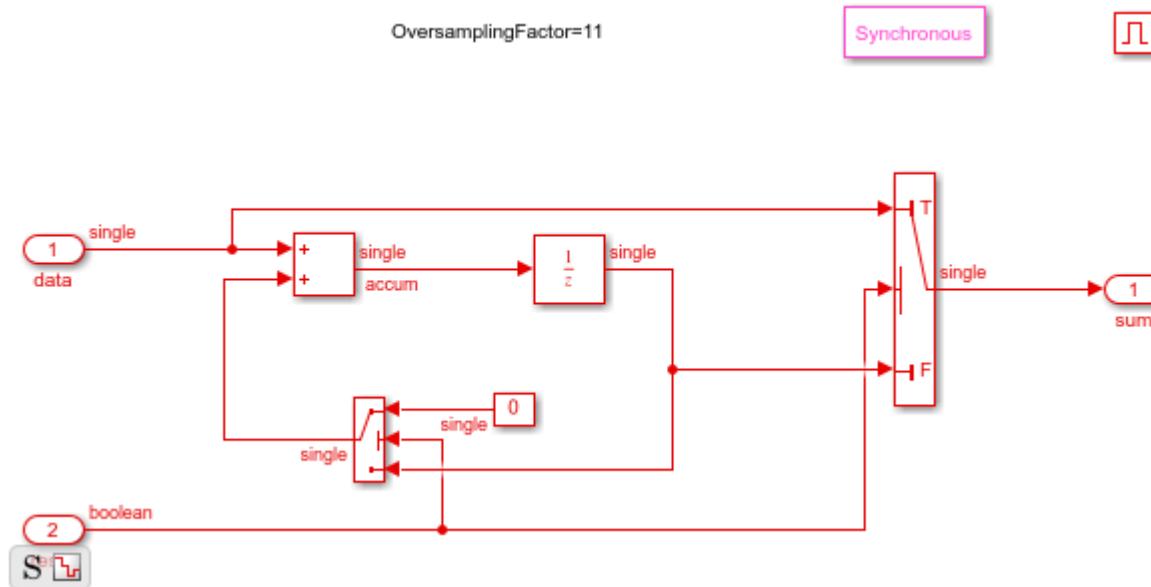
If you open this example model `RunningSum.slx`, you see a Simulink™ model that uses single data types.



To generate HDL code for the `CumSum_Sl` Subsystem, right-click the subsystem and select **HDL Code > Generate HDL for Subsystem**. During code generation, HDL Coder™ generates an error:

Unable to allocate delays to compensate for the 11 delays introduced by Add in native floating-point mode. Consider either increasing the oversampling factor, setting the 'Latency Strategy' to 'Zero', or adding the necessary output pipelines via HDL block properties for other blocks in the model to accommodate for the latency introduced by this block.

By using the path to the block mentioned in the error message, navigate to the Add block in the model. This block is inside a feedback loop.



The Add block has a latency of 11. When generating code, HDL Coder™ cannot allocate 11 delays for the block, because it cannot add matching delays to other paths.

This model serves as an example to illustrate the various strategies to solve this problem.

## Solution

### Strategy 1: Global Oversampling

This modeling paradigm uses the clock-rate pipelining optimization to oversample your design to a clock-rate much faster than the DUT sample rate. To enable this optimization, specify a global oversampling factor for your Simulink model. The floating-point delays then operate at the faster clock-rate and can be allocated successfully. For more information, see “Clock-Rate Pipelining” on page 24-114.

- 1 Specify an oversampling factor that is equal to or greater than the latency of the floating-point operators that are unable to allocate delays. For the `RunningSum` model, specify an oversampling factor at least equal to 12. To learn about the latency values of the floating-point operators, see “Simulink Blocks Supported with Native Floating-Point” on page 10-120.

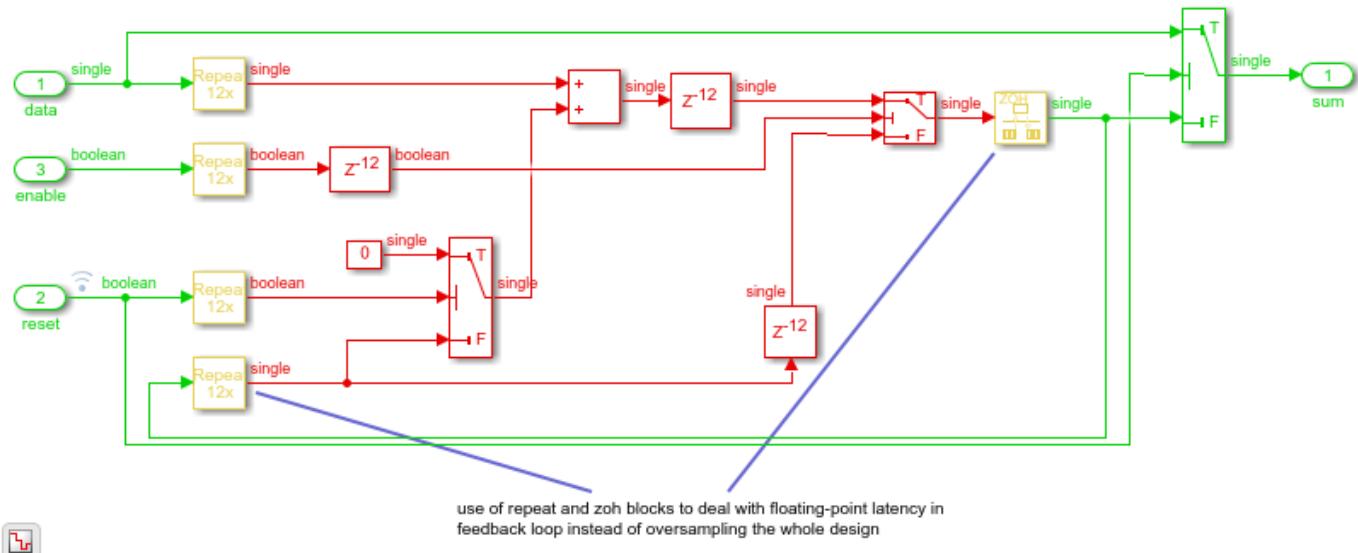
To specify the oversampling factor:

- a In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
  - b Click **Settings**. In the **HDL Code Generation > Global Settings** tab, set **Oversampling factor** to 12.
- 2 Enable hierarchy flattening on the DUT and make sure that subsystems inside the DUT inherit this setting. For the `RunningSum` model, select the `CumSum_sl` subsystem and click **HDL Block Properties** on the **HDL Code** tab, and then set **FlattenHierarchy** to on.

## Strategy 2: Local Oversampling

To model your design at the data rate and selectively increase the sample rate of blocks for which HDL Coder™ is unable to allocate delays, use local oversampling. These blocks then operate at the faster clock rate and can accommodate the required number of delays.

If you open the `RunningSum_OSmanual` model and navigate to the Add block, it shows how you can increase the sample rate of the Add block and allocate delays.



- The blocks that are within the boundary of the Repeat and Zero Order Hold blocks operate at the clock rate that is 12 times faster than the sample rate of the model.
- The subsystem has a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this Delay block, which compensates for the latency of the operator. To balance delays, the subsystem contains Delay blocks of length 12 in other paths.

You can now generate HDL code for the `CumSum_s1` subsystem. To generate HDL code for the `CumSum_S1` Subsystem, right-click the subsystem and select **HDL Code > Generate HDL for Subsystem**.

## Strategy 3: Delay Blocks

Use this modeling paradigm to model your entire design at the Simulink data rate. For blocks that are unable to accommodate the required number of delays, add a Delay block with a sufficient **Delay length** at the output of the blocks. Specify a **Delay length** that is equal to the latency of the floating-point operator. Make sure that you add matching delays in other paths.

For the `RunningSum` model, you can add a Delay block of length 12 at the output of the Add block. When generating code, the Add block absorbs this delay, because the block has a latency of 12.

For more information, see “Latency Considerations with Native Floating Point” on page 10-96.

## Strategy 4: Use Custom Latency

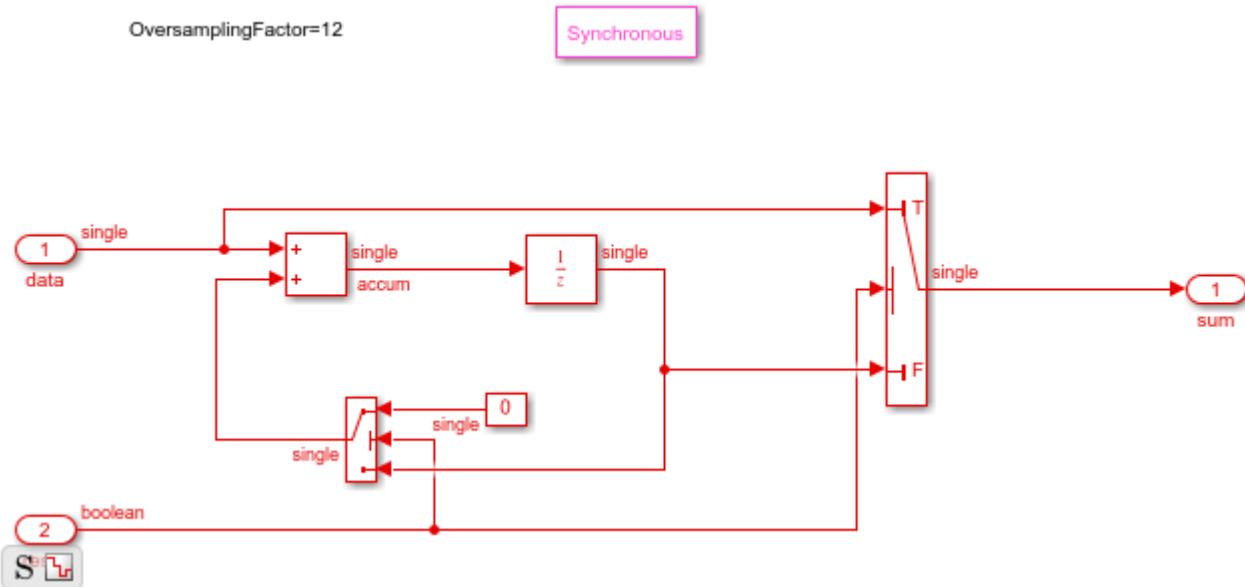
You can use the **Latency Strategy** for various blocks to specify a custom latency value and absorb the additional delays. Using this strategy can optimize your design for trade-offs between:

- Clock frequency and power consumption.
- Oversampling factor and sampling frequency.

To learn more about the trade-offs and blocks for which you can specify a custom latency, see “LatencyStrategy” on page 22-33.

**Note:** When you use the custom latency strategy, make sure that the Subsystem that contains the block for which native floating-point is unable to allocate delays is not a conditional subsystem. That is, the Subsystem must not contain trigger, reset, or enable ports.

To see how using a custom latency can resolve the delay allocation issue, open the model `RunningSum_Custom.slx`.



The model is similar to the original `RunningSum` model but does not have the `Enable` block. Using the `Enable` block can prevent the custom latency strategy from absorbing delays. Specify a custom latency of one for the `Add` block. The `Add` block can then absorb the Unit Delay adjacent to the `Add` block.

You can now generate HDL code for the `CumSum_sl` subsystem.

### Strategy 5: Zero Latency

You can use the zero latency strategy setting for blocks in your design for which native floating point is unable to allocate delays. By default, blocks in your design inherit the native floating-point settings that you specify in the Configuration Parameters dialog box. To specify a zero latency strategy setting for a block, in the HDL Block Properties dialog box for that block, on the **Native Floating Point** tab, set **LatencyStrategy** to **Zero**.

For the `RunningSum` example, set the **LatencyStrategy** of the `Add` block to **Zero**. To choose the native floating point library and specify zero latency strategy, at the command line, enter:

```
fc = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
hdlset_param('RunningSum/CumSum_sl/Subsystem/Add', ...
    'LatencyStrategy', 'Zero');
```

**Note** To obtain good performance on the target FPGA device, it is not recommended to set **Latency Strategy** to Zero from the Configuration Parameters dialog box.

## See Also

[createFloatingPointTargetConfig](#)

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

# Optimize Generated HDL Code for Multirate Designs with Large Rate Differentials

## In this section...

["Issue" on page 10-72](#)

["Description" on page 10-72](#)

["Recommendations" on page 10-74](#)

## Issue

When generating HDL code from your multirate algorithm in Simulink, HDL Coder might generate a large number of pipeline registers that can prevent the HDL design from fitting into an FPGA. This issue occurs due to modeling patterns that might result in large rate differentials. You can address this issue by using modeling techniques to manage sample time ratios.

## Description

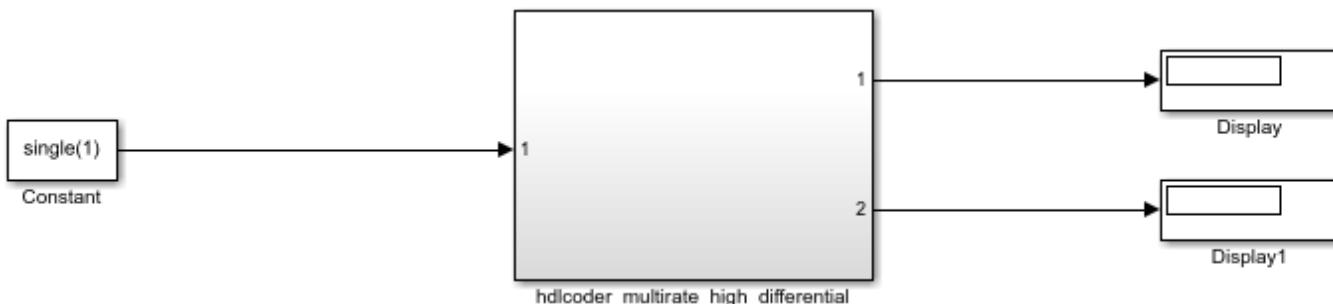
This issue occurs when your Simulink™ model has a significantly large difference in sample rates or uses certain block implementations or optimizations that result in different clock-rate paths, such as:

- Multicycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Fixed-point math functions such as reciprocal, sqrt, or divide
- Resource sharing
- Streaming

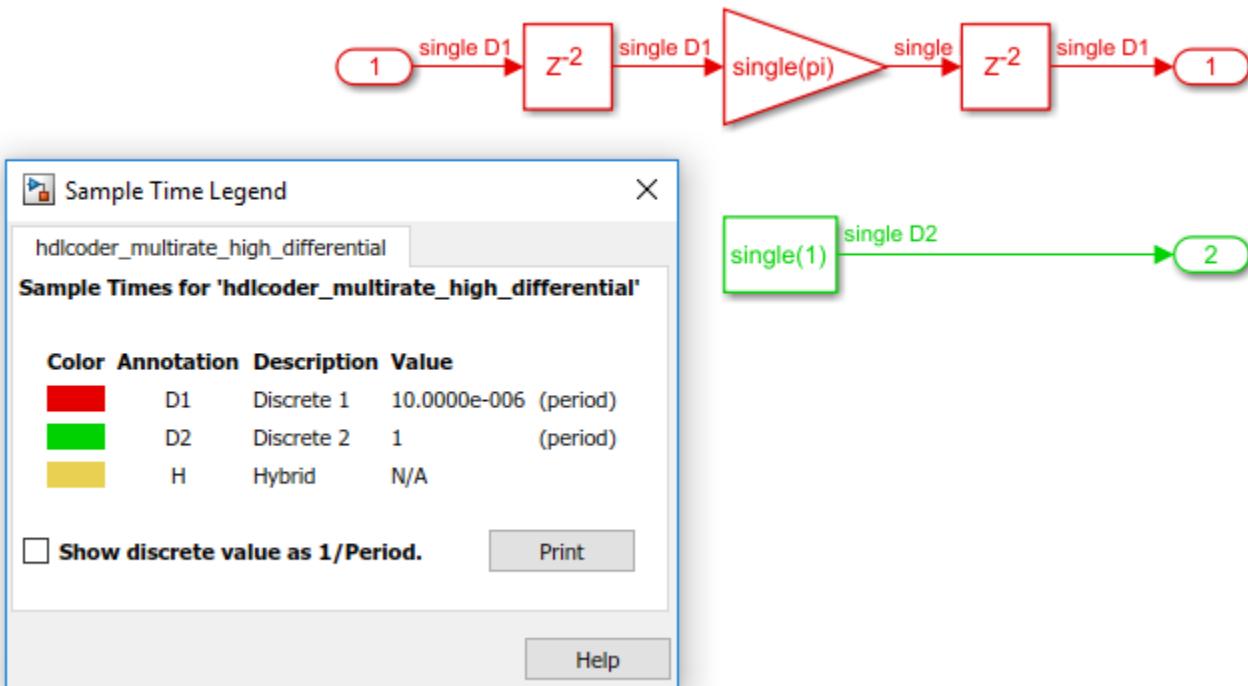
The additional pipelines result in a latency overhead that requires the insertion of matching delays across multiple signal paths operating at different rates. If the ratio of the fastest to the slowest clock rate is quite large, the code generator can potentially introduce a large number of registers in the resulting HDL code. The large number of pipeline registers can increase the size of the generated HDL files, and can prevent the design from fitting into an FPGA.

To see an example of how this issue occurs, open this Simulink™ model.

```
open_system('hdlcoder_multirate_high_differential')
```



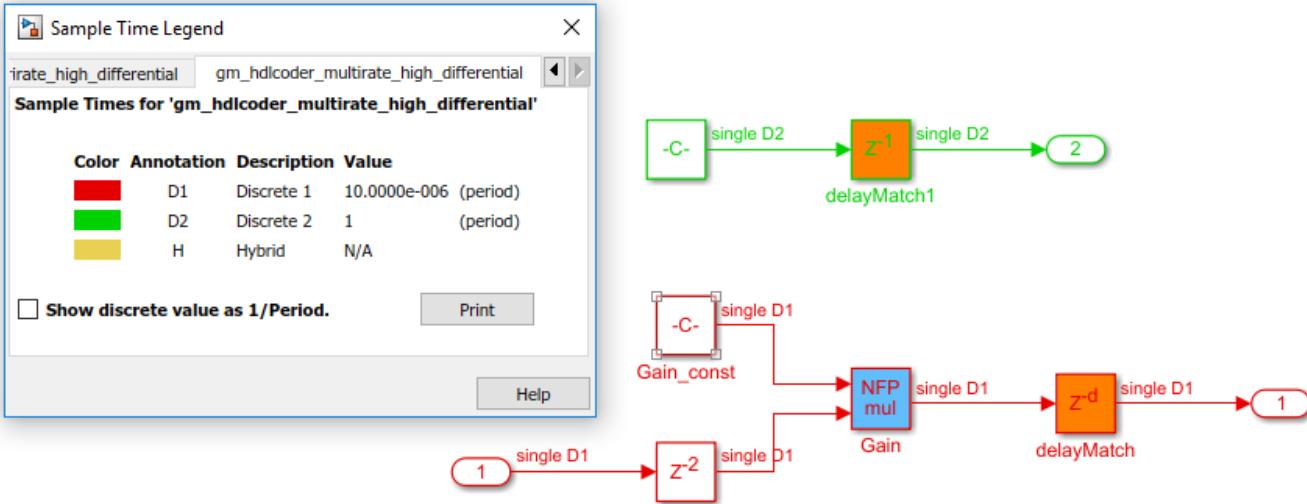
When you compile the model and double-click the `hdlcoder_multirate_high_differential` Subsystem, you can see that the model has a floating-point Gain block, a multicycle operator, in the fast clock-rate region.



Generate HDL code for the `hdlcoder_multirate_high_differential` Subsystem and check the output log.

```
## Generating HDL for 'hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential'.
## Using the config set for model hdlcoder\_multirate\_high\_differential for HDL code generation parameters.
## Starting HDL check.
## The code generation and optimization options you have chosen have introduced additional pipeline delays.
## The delay balancing feature has automatically inserted matching delays for compensation.
## The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
## Output port 0: 100000 cycles.
## Output port 1: 1 cycles.
## Begin VHDL Code Generation for 'hdlcoder_multirate_high_differential'.
## Working on hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential/nfp_mul_comp as hdlsrc\hdlcoder\_multirate\_high\_differential\nfp\_mul\_comp.vhd.
## Working on hdlcoder_multirate_high_differential_to as hdlsrc\hdlcoder\_multirate\_high\_differential\hdlcoder\_multirate\_high\_differential\_to.vhd.
## Working on hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential as hdlsrc\hdlcoder\_multirate\_high\_differential\hdlcoder\_multirate\_high\_differential.vhd.
## Generating package file hdlsrc\hdlcoder\_multirate\_high\_differential\hdlcoder\_multirate\_high\_differential\_pkg.vhd.
## Creating HDL Code Generation Check Report hdlcoder\_multirate\_high\_differential\_report.html
## HDL check for 'hdlcoder_multirate_high_differential' complete with 0 errors, 0 warnings, and 0 messages.
## HDL code generation complete.
```

Open the generated model. At the command line, enter `gm_hdlcoder_multirate_high_differential`. When you compile the model and double-click the `hdlcoder_multirate_high_differential` Subsystem, the model looks as displayed by the sample time legend.



The large output latency on the fast clock rate region of the design is introduced by the code generator to balance delays across multiple output paths of the system. This large latency increases the size of the generated HDL files and reduces the efficiency of the generated code.

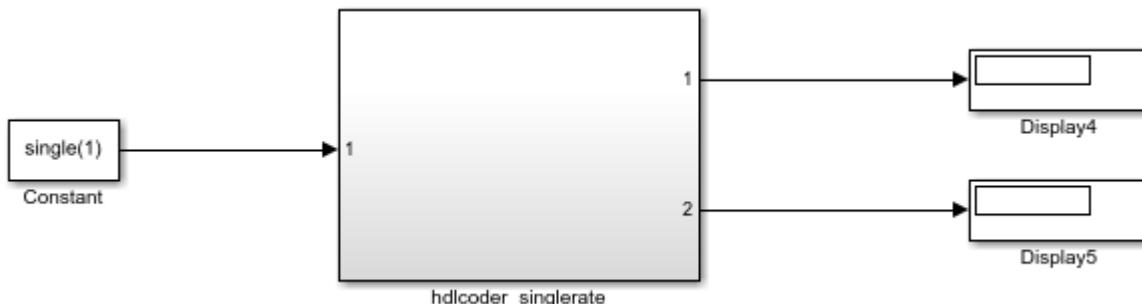
## Recommendations

### Recommendation 1: Use a Single-Rate Model

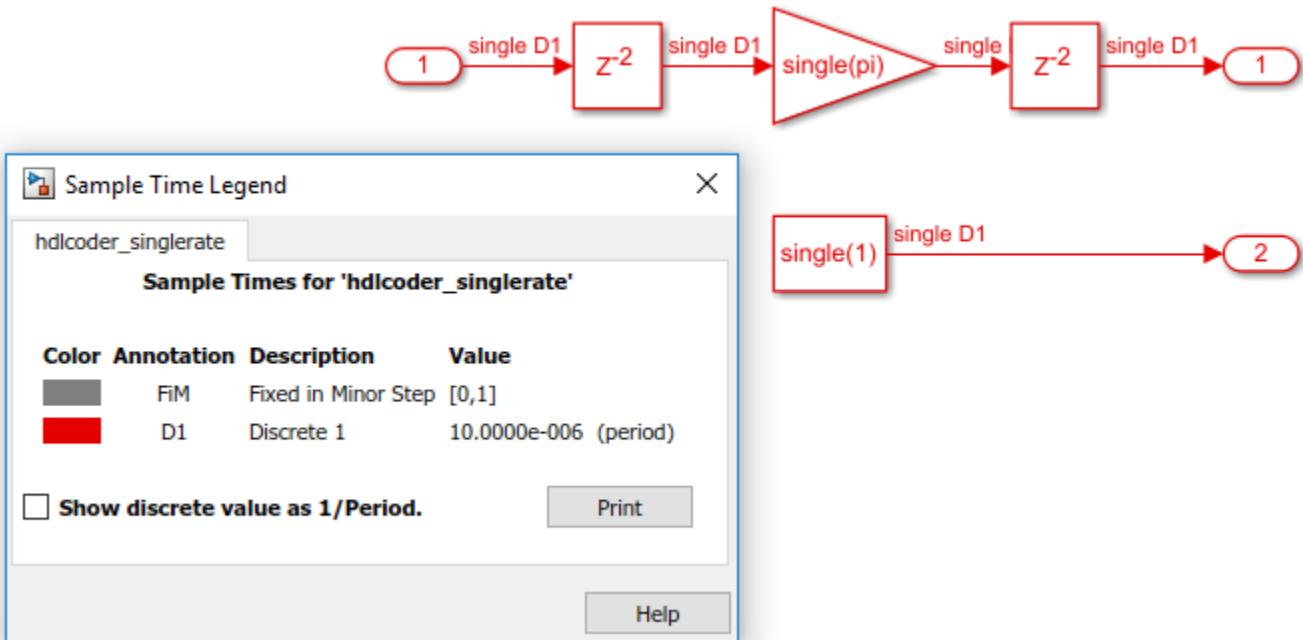
Most applications that you target the HDL code for might not require such a large rate differential. In that case, it is recommended that you use a single-rate model. In this example, you can change the sample rate of the Constant block inside the `hdlcoder_multirate_high_differential` Subsystem to be the same as that of the base model.

Open this model that has the sample time of the Constant block changed to 10E-06, which is the same sample time as the base sample time of the model.

```
open_system('hdlcoder_singlerate')
```



When you compile the model and double-click the `hdlcoder_singlerate` Subsystem, you see that the signal paths in the model operate at the same sample time of 10E-06.



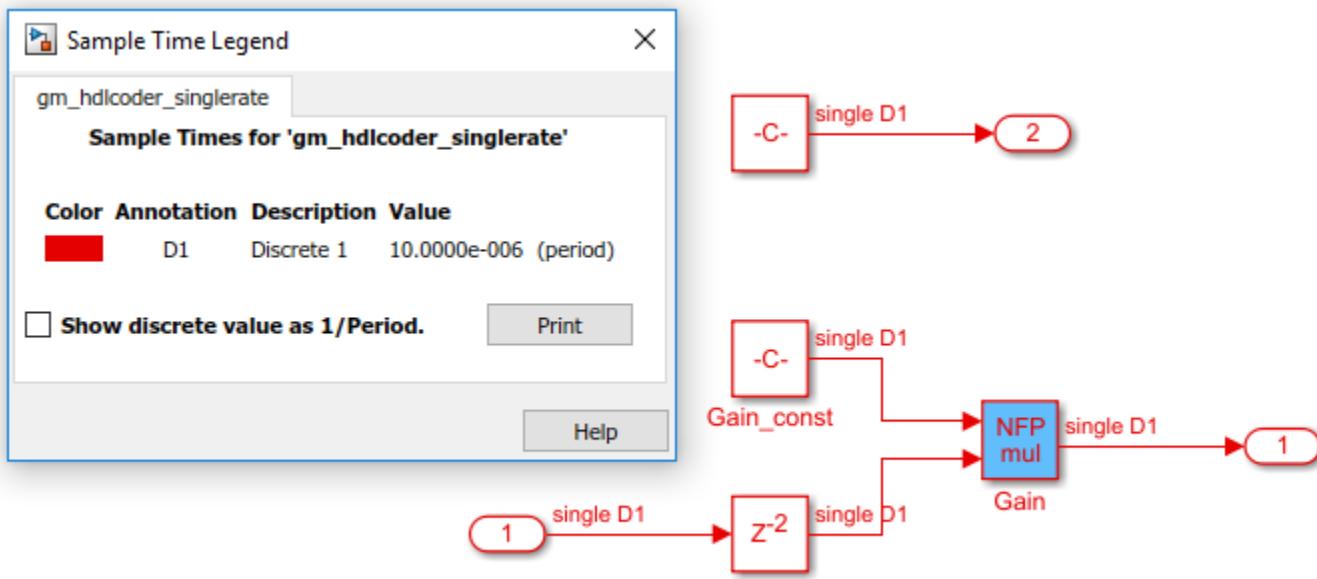
Generate HDL code for the `hdlcoder_singlerate` Subsystem and check the output log.

```

### Generating HDL for 'hdlcoder_singlerate/hdlcoder_singlerate'.
### Using the config set for model hdlcoder\_singlerate for HDL code generation parameters.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
### Output port 0: 6 cycles.
### Output port 1: 6 cycles.
### Begin VHDL Code Generation for 'hdlcoder_singlerate'.
### Working on hdlcoder_singlerate/hdlcoder_singlerate/nfp_mul_comp as hdlsrc\hdlcoder\_singlerate\nfp\_mul\_comp.vhd.
### Working on hdlcoder_singlerate/hdlcoder_singlerate as hdlsrc\hdlcoder\_singlerate\hdlcoder\_singlerate.vhd.
### Generating package file hdlsrc\hdlcoder\_singlerate\hdlcoder\_singlerate\_pkg.vhd.
### Creating HDL Code Generation Check Report hdlcoder\_singlerate\_report.html
### HDL check for 'hdlcoder_singlerate' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

```

You see that the output latency has decreased significantly. Now open the generated model. At the MATLAB™ command line, enter `gm_hdlcoder_singlerate`. When you compile the model and double-click the `hdlcoder_singlerate` Subsystem, the model looks as displayed by the sample time legend.



The generated HDL code is now optimal and uses few registers. Therefore, you can deploy the design to target FPGA platforms.

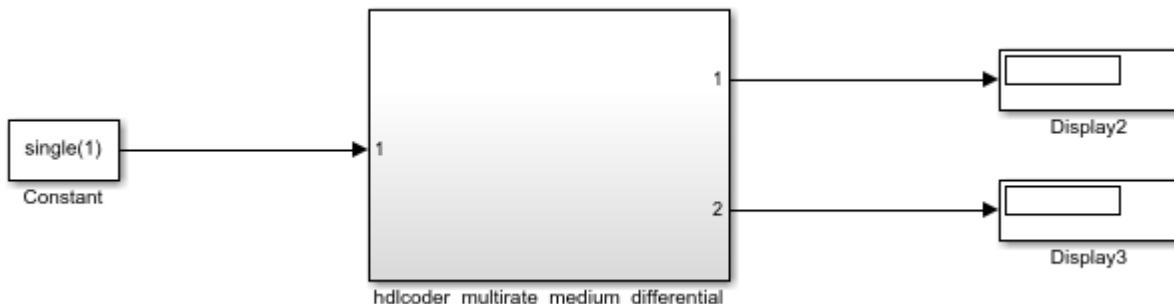
### Recommendation 2: Reduce the Rate Differential

If you want to use a multirate model, it is recommended that you reduce the rate differential. Rate differential corresponds to the ratio of the fastest to the slowest clock rate in your design. If your target application requires two signal paths such that one signal path runs in time units of nanoseconds (ns) and the other signal path runs in time units of microseconds (us), you can choose to retain the multirate paths in your model. Be aware that delay balancing can introduce a significantly large number of registers to balance the signal paths.

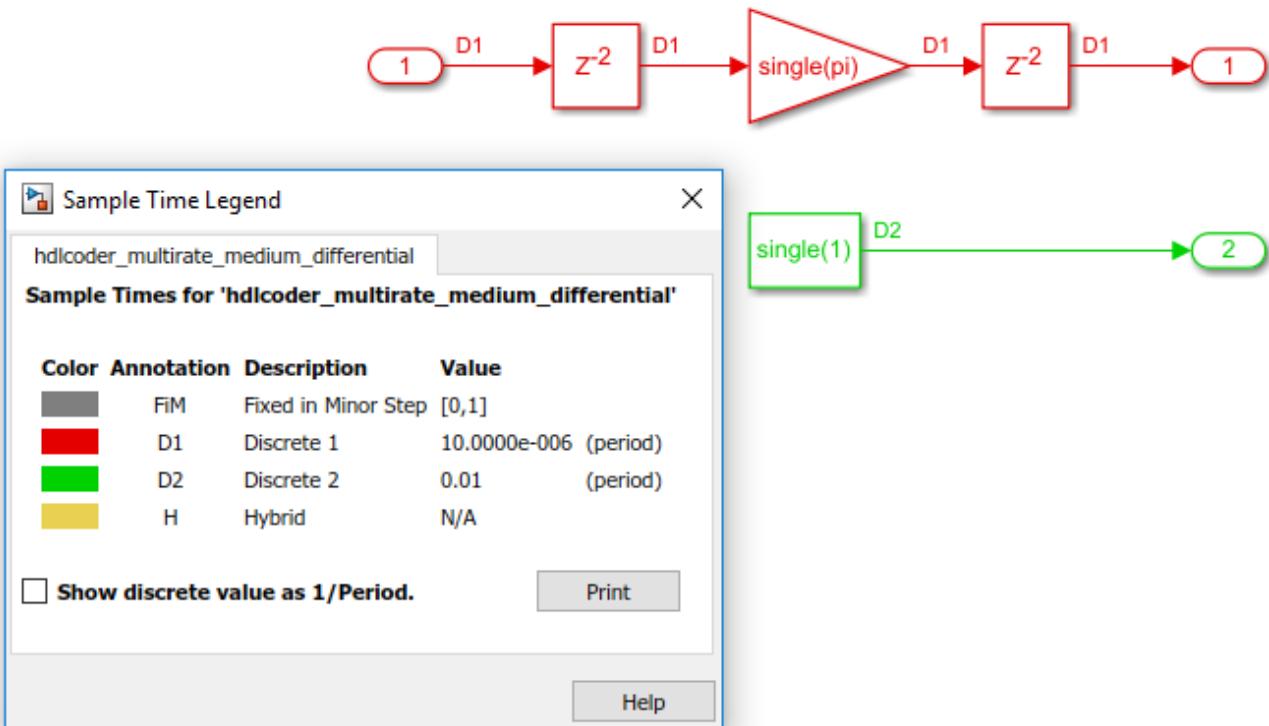
In this example, you can change the sample rate of the Constant block inside the hdlcoder\_multirate\_high\_differential Subsystem to reduce the rate differential.

Open this model that has the sample rate of the Constant block changed to 0.01.

```
open_system ('hdlcoder_multirate_medium_differential')
```



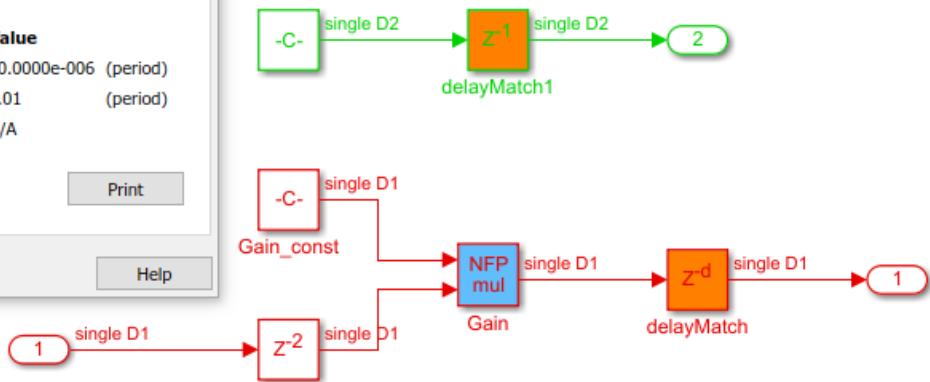
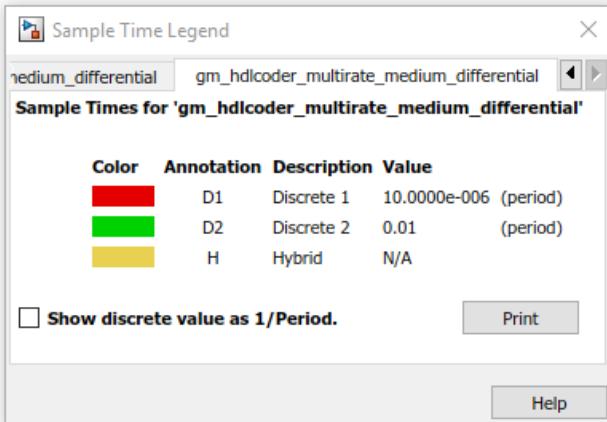
When you compile the model and double-click the hdlcoder\_multirate\_medium\_differential Subsystem, you see that the rate differential between the two signal paths is equal to 1000.



Generate HDL code for the `hdlcoder_multirate_medium_differential` Subsystem and check the output log.

```
## Generating HDL for 'hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential'.
## Using the config set for model hdlcoder\_multirate\_medium\_differential for HDL code generation parameters.
## Starting HDL check.
## The code generation and optimization options you have chosen have introduced additional pipeline delays.
## The delay balancing feature has automatically inserted marching delays for compensation.
## The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
## Output port 0: 1000 cycles.
## Output port 1: 1 cycles.
## Begin VHDL Code Generation for 'hdlcoder_multirate_medium_differential'.
## Working on hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential/nfp_mul_comp as hdlsrc\hdlcoder\_multirate\_medium\_differential\nfp\_mul\_comp.vhd.
## Working on hdlcoder_multirate_medium_differential_tc as hdlsrc\hdlcoder\_multirate\_medium\_differential\hdlcoder\_multirate\_medium\_differential\_tc.vhd.
## Working on hdlcoder_multirate_medium_differential/hdlcoder_multirate_medium_differential as hdlsrc\hdlcoder\_multirate\_medium\_differential\hdlcoder\_multirate\_medium\_differential.vhd.
## Generating package file hdlsrc\hdlcoder\_multirate\_medium\_differential\hdlcoder\_multirate\_medium\_differential\_pkg.vhd.
## Creating HDL Code Generation Check Report hdlcoder\_multirate\_medium\_differential\_report.html
## HDL check for 'hdlcoder_multirate_medium_differential' complete with 0 errors, 0 warnings, and 0 messages.
## HDL code generation complete.
```

Open the generated model. At the MATLAB™ command line, enter `gm_hdlcoder_multirate_medium_differential`. When you compile the generated model and double-click the `hdlcoder_multirate_medium_differential` Subsystem, the model is as displayed by the sample time legend.



The model has a large number of registers, approximately 1000, in the fast clock rate path. The additional cost of registers is expected when you have a control logic that runs at a sample rate that is 1000 times faster than the sample rate of the system. When you deploy the generated code to a target platform, be aware of the constraints in hardware resources on the target platform. This recommendation offers a trade-off between generating optimal HDL code and targeting practical FPGA applications that might require an extremely large rate differential.

### Recommendation 3: Map Pipeline Delays to RAM

To optimize the number of registers that your design uses on the target FPGA device, you can use the **Map Pipeline Delays to RAM** setting. This setting is a trade-off of the pipeline registers that are inserted in the HDL code with RAM resources to save area footprint on the target FPGA device. You can enable this setting in the **HDL Code Generation > Optimizations > General** tab of the Configuration Parameters dialog box.

You can also specify this setting at the command line by using the `MapPipelineDelaysToRAM` property with `hdlset_param` or `makehdl`. You can view the property value by using `hdlget_param`. Use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential', ...
    'MapPipelineDelaysToRAM','on')
```

- When you use `hdlset_param`, you can set the parameter on the model, and then generate HDL code by using `makehdl`.

```
hdlset_param('hdlcoder_multirate_high_differential', ...
    'MapPipelineDelaysToRAM','on')
makehdl('hdlcoder_multirate_high_differential/hdlcoder_multirate_high_differential')
```

Use this setting in combination with the previous recommendations to further improve the efficiency of the generated HDL code and for deploying the code to the target platform.

### See Also

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

# Getting Started with HDL Coder Native Floating-Point Support

## In this section...

- “Key Features” on page 10-80
- “Numeric Considerations and IEEE-754 Standard Compliance” on page 10-80
- “Floating Point Types” on page 10-81
- “Data Type Considerations” on page 10-82

Native floating-point support in HDL Coder enables you to generate code from your floating-point design. If your design has complex math and trigonometric operations or has data with a large dynamic range, use native floating-point.

## Key Features

In your Simulink model:

- You can have half-precision, single-precision, and double-precision floating-point data types and operations.
- You can have a combination of integer, fixed-point, and floating-point operations. By using Data Type Conversion blocks, you can perform conversions between floating-point and fixed-point data types.

The generated code:

- Complies with the IEEE-754 standard of floating-point arithmetic.
- Is target-independent. You can deploy the code on any generic FPGA or an ASIC.
- Does not require floating-point processing units or hard floating-point DSP blocks on the target ASIC or FPGA.

HDL Coder supports:

- Math and trigonometric functions
- Large subset of Simulink blocks
- Denormal numbers
- Customizing the latency of the floating-point operator

## Numeric Considerations and IEEE-754 Standard Compliance

Native floating point technology in HDL Coder adheres to IEEE standard of floating-point arithmetic. For basic arithmetic operations such as addition, subtraction, multiplication, division, and reciprocal, when you generate HDL code in native floating-point mode, the numeric results obtained match the original Simulink model.

Certain advanced math operations such as exponential, logarithm, and trigonometric operators have machine-specific implementation behaviors because these operators use recurring Taylor series and Remez expression based implementations. When you use these operators in native floating-point mode, the generated HDL code can have relatively small numeric differences from the Simulink model. These numeric differences are within a tolerance range and therefore indicate compliance with the IEEE-754 standard.

To generate code that complies with the IEEE-754 standard, HDL Coder supports:

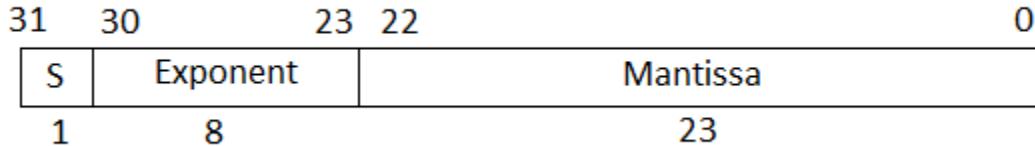
- Round to nearest rounding mode
- Denormal numbers
- Exceptions such as NaN (Not a Number), Inf, and Zero
- Customization of ULP (Units in the Last Place) and relative accuracy

For more information, see “Numeric Considerations with Native Floating-Point” on page 10-84.

## Floating Point Types

### Single Precision

In the IEEE 754-2008 standard, the single-precision floating-point number is 32-bits. The 32-bit number encodes a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa.



This graph is the normalized representation for floating-point numbers. You can compute the actual value of a normal number as:

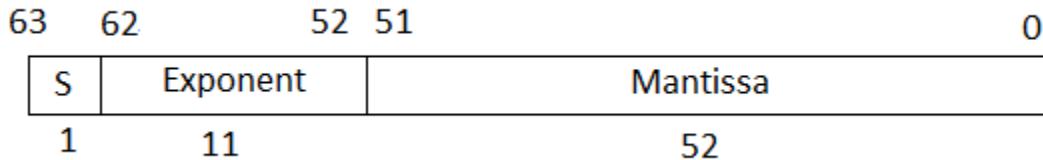
$$\text{value} = (-1)^{\text{sign}} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{(e - 127)}$$

The exponent field represents the exponent plus a bias of 127. The size of the mantissa is 24 bits. The leading bit is a 1, so the representation encodes the lower 23 bits.

Use single-precision types for applications that require larger dynamic range than half-precision types. Single-precision operations consume less memory and has lower latency than double-precision types.

### Double Precision

In the IEEE 754-2008 standard, the single-precision floating-point number is 64-bits. The 64-bit number encodes a 1-bit sign, an 11-bit exponent, and a 52-bit mantissa.

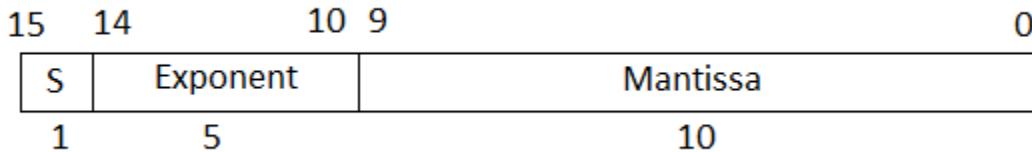


The exponent field represents the exponent plus a bias of 1023. The size of the mantissa is 53 bits. The leading bit is a 1, so the representation encodes the lower 52 bits.

Use double-precision types for applications that require larger dynamic range, accuracy, and precision. These operations consume larger area on the FPGA and lower target frequency.

## Half Precision

In the IEEE 754-2008 standard, the half-precision floating-point number is 16-bits. The 16-bit number encodes a 1-bit sign, a 5-bit exponent, and a 10-bit mantissa.



The exponent field represents the exponent plus a bias of 15. The size of the mantissa is 11 bits. The leading bit is a 1, so the representation encodes the lower 10 bits.

Use half-precision types for applications that require smaller dynamic range, consumes much less memory, has lower latency, and saves FPGA resources.

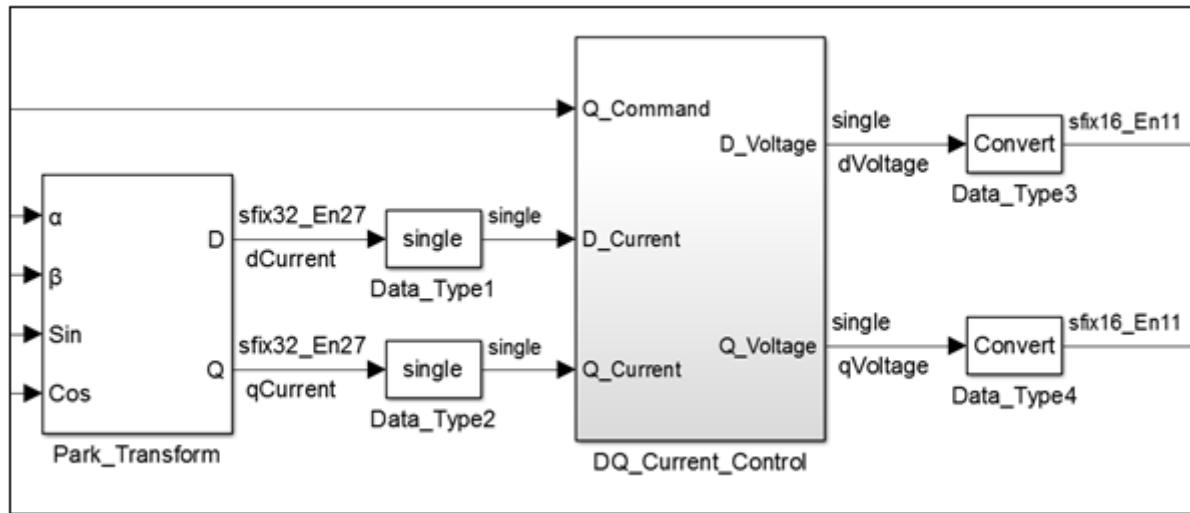
When using half types, you might want to explicitly set the **Output data type** of the blocks to half instead of the default setting Inherit: Inherit via internal rule. To learn how to change the parameters programmatically, see “Set HDL Block Parameters for Multiple Blocks Programmatically” on page 22-52.

## Data Type Considerations

With native floating-point support, HDL Coder supports code generation from Simulink models that contain floating-point signals and fixed-point signals. You might want to model your design with floating-point types to:

- Implement algorithms that have a large or unknown dynamic range that can fall outside the range of representable fixed-point types.
  - Implement complex math and trigonometric operations that are difficult to design in fixed point.
  - Obtain a higher precision and better accuracy.

Floating-point designs can potentially occupy more area on the target hardware. In your Simulink model, it is recommended to use floating-point data types in the algorithm data path and fixed-point data types in the algorithm control logic. This figure shows a section of a Simulink model that uses Single and fixed-point types. By using Data Type Conversion blocks, you can perform conversions between the single and fixed-point types.



## See Also

### Modeling Guidelines

"Modeling with Native Floating Point" on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Floating-Point Tolerance Parameters” on page 19-26
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120

## Numeric Considerations with Native Floating-Point

### In this section...

- “Round to Nearest Rounding Mode” on page 10-84
- “Denormal Numbers” on page 10-84
- “Exception Handling” on page 10-85
- “Relative Accuracy and ULP Considerations” on page 10-85

Native floating-point technology can generate HDL code from your floating-point design. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

HDL Coder generates code that complies with the IEEE-754 standard of floating-point arithmetic. HDL Coder native floating-point supports:

- Round to nearest rounding mode
- Denormal numbers
- Exceptions such as NaN (Not a Number), Inf, and Zero
- Customization of ULP (Units in the Last Place) and relative accuracy

### Round to Nearest Rounding Mode

HDL Coder native floating-point uses the round to nearest even rounding mode. This mode resolves all ties by rounding to the nearest even digit.

This rounding method requires at least three trailing bits after the 23 bits of the mantissa. The MSB is called Guard bit, the middle bit is called the Round bit, and the LSB is called the Sticky bit. The table shows the rounding action that HDL Coder performs based on different values of the three trailing bits. x denotes a *don't care* value and can take either a 0 or a 1.

Rounding bits	Rounding Action
0xx	No action performed.
100	A tie. If the mantissa bit that precedes the Guard bit is a 1, round up, otherwise no action is performed.
101	Round up.
11x	Round up.

### Denormal Numbers

Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. The leading bit of the mantissa is zero.

$$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{-126}$$

Denormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The presence of denormal numbers indicates loss

of significant digits that can accumulate over subsequent operations and eventually result in unexpected values.

The logic to handle denormal numbers involves counting the number of leading zeros and performing a left shift operation to obtain the normalized representation. Addition of this logic increases the area footprint on the target device and can affect the timing of your design.

When using native floating-point support, you can specify whether you want HDL Coder to handle denormal numbers in your design. By default, the code generator does not check for denormal numbers, which saves area on the target platform.

## Exception Handling

If you perform operations such as division by zero or compute the logarithm of a negative number, HDL Coder detects and reports exceptions. The table summarizes the mapping from the encoding of a floating-point number to the value of the number for various kinds of exceptions. x denotes a *don't care* value and can take either a 0 or a 1.

Sign	Exponent	Significand	Value	Description
x	0xFF	0x00000000	$value = (-1)^{S_\infty}$	Infinity
x	0xFF	A nonzero value	$value = NaN$	Not a Number
x	0x00	0x00000000	$value = 0$	Zero
x	0x00	A nonzero value	$value = (-1)^{sign} * (0 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{-126}$	Denormal
x	$0x00 < E < 0xFF$	x	$value = (-1)^{sign} * (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) * 2^{(e - 127)}$	Normal

## Relative Accuracy and ULP Considerations

The representation of infinitely real numbers with a finite number of bits requires an approximation. This approximation can result in rounding errors in floating-point computation. To measure the rounding errors, the floating-point standard uses relative error and ULP (Units in the Last Place) error.

### ULP

If the exponent range is not upper-bounded, Units in Last Place (ULP) of a floating-point number x is the distance between two closest straddling floating-point numbers a and b nearest to x. The IEEE-754 standard requires that the result of an elementary arithmetic operation such as addition, multiplication, and division is correctly round. A correctly rounded result means that the rounded result is within 0.5 ULP of the exact result.

An ULP of one means adding a 1 to the decimal value of the number. The table shows the approximation of pi to nine decimal digits and how the ULP of one changes the approximate value.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP
3.141592741	1078530011	0 10000000 10010010000111111011011	0
3.141592979	1078530012	0 10000000 10010010000111111011100	1

The gap between two consecutively representable floating-point numbers varies according to magnitude.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP
1234567	1234613304	0 10010011 00101101011010000111000	0
1234567.125	1234613305	0 10010011 00101101011010000111001	1

### Relative Error

Relative error measures the difference between a floating-point number and the approximation of the real number. Relative error returns the distance from 1.0 to the next larger number. This table shows how the real value of a number changes with the relative accuracy.

Floating-point number	Value in decimal	IEEE-754 representation for Single Types	ULP	Relative error
8388608	1258291200	0 10010110 00000000000000000000000000000000	0	1
8388607	1258291198	0 10010101 11111111111111111111111111111110	1	2.3841858e-07
1	1065353216	0 01111111 00000000000000000000000000000000	0	1.1920929e-07
2	1073741824	0 10000000 00000000000000000000000000000000	1	2.3841858e-07

The magnitude of the relative error depends on the real value of the floating-point number.

In MATLAB, the `eps` function measures the relative accuracy of the floating-point number. For more information, see `eps`.

## See Also

### Modeling Guidelines

“Modeling with Native Floating Point” on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “ULP Considerations of Native Floating-Point Operators” on page 10-88
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Floating-Point Tolerance Parameters” on page 19-26
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

## ULP Considerations of Native Floating-Point Operators

### In this section...

"Adherence of Native Floating Point Operators to IEEE-754 Standard" on page 10-88

"ULP Values of Floating Point Operators" on page 10-88

"Considerations" on page 10-89

The representation of infinitely real numbers with a finite number of bits requires an approximation. This approximation can result in rounding errors in floating-point computation. To measure the rounding errors, the floating-point standard uses relative error and ULP (Units in the Last Place) error. To learn about relative error, see "Relative Accuracy and ULP Considerations" on page 10-85.

If the exponent range is not upper-bounded, Units in Last Place (ULP) of a floating-point number  $x$  is the distance between two closest straddling floating-point numbers  $a$  and  $b$  nearest to  $x$ . The IEEE-754 standard requires that the result of an elementary arithmetic operation such as addition, multiplication, and division is correctly round. A correctly rounded result means that the rounded result is within 0.5 ULP of the exact result.

### Adherence of Native Floating Point Operators to IEEE-754 Standard

Native floating point technology in HDL Coder follows IEEE standard of floating-point arithmetic. Basic arithmetic operations such as addition, subtraction, multiplication, division, and reciprocal are mandated by IEEE to have zero ULP error. When you perform these operations in native floating-point mode, the numerical results obtained from the generated HDL code match the original Simulink model.

Certain advanced math operations such as exponential, logarithm, and trigonometric operators have machine-specific implementation behaviors because these operators use recurring taylor series and remez expression based implementations. When you use these operators in native floating-point mode, there can be relatively small differences in numerical results between the Simulink model and the generated HDL code.

You can measure the difference in numerical results as a relative error or ULP. A nonzero ULP for these operators does not mean noncompliance with the IEEE standard. A ULP of one is equivalent to a relative error of  $10^{-7}$ . You can ignore such relatively small errors by specifying a custom tolerance value for the ULP when generating a HDL test bench. For example, you can specify a custom floating-point tolerance of one ULP to ignore the error when verifying the generated code. For more information, see "Floating-Point Tolerance Parameters" on page 19-26.

### ULP Values of Floating Point Operators

The table enumerates the ULP of floating-point operators that have a nonzero ULP. In addition to these operators, the HDL Reciprocal block has a ULP of five.

## Math Functions

Simulink Blocks	Units in the Last Place (ULP) error
exp	1
log	1
log10	1
$10^u$	1
pow	1
hypot	1

## Trigonometric Functions

Simulink Blocks	Units in the Last Place (ULP) error
sin	2
cos	2
tan	3
asin	2
acos	2
atan	2
atan2	5
sinh	1
cosh	1
tanh	1
asinh	2
acosh	2
atanh	3
sincos	2

## Considerations

For certain floating-point input values, some blocks can produce simulation results that vary from the MATLAB simulation results. To see the difference in results, before you generate code, enable generation of the validation model. In the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select the **Generate validation model** check box.

- If you perform computations that involve complex numbers and an exception such as `Inf` or `NaN`, the HDL simulation result with native floating point can potentially vary from the Simulink simulation result. For example, if you multiply a complex input with `Inf`, the Simulink simulation result is `Infj` whereas the HDL simulation result is `NaN+Infj`.
- If you compute the square root or logarithm of a negative number, the HDL simulation result with native floating point is `0`. This result matches the simulation result when you verify the design with a SystemVerilog DPI test bench. In Simulink, the result obtained is `NaN`. According to the IEEE-754 standard, if you compute the square root or logarithm of a negative number, the result is that number itself.

- If the input to the Direct Lookup Table (n-D) is of floating-point data type, but the elements of the table use a smaller data type, the generated HDL code can be potentially incorrect. For example, the input is of `single` type and the elements use `uint8` type. To obtain accurate HDL simulation results, use the same data type for the input signal and the elements of the lookup table.
- If you use the Cosine block with the inputs `-7.729179E28` or `7.729179E28`, the generated HDL code has a ULP of 4. For all other inputs, the ULP is 2.
- When you use a Math Function block to compute `mod(a,b)` or `rem(a,b)`, where `a` is the dividend and `b` is the divisor, the simulation result in native floating-point mode varies from the MATLAB simulation result in these cases:
  - If  $b$  is integer and  $\frac{a}{b} > 2^{32}$ , the simulation result in native floating-point mode is zero. For such significant difference in magnitude between the numbers `a` and `b`, this implementation saves area on the target FPGA device.
  - If  $\frac{a}{b}$  is close to  $2^{23}$ , the simulation result in native floating-point mode can potentially vary from the MATLAB simulation results.

## See Also

### Modeling Guidelines

"Modeling with Native Floating Point" on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- "Floating Point Support: Field-Oriented Control Algorithm" on page 10-109

## More About

- "Numeric Considerations with Native Floating-Point" on page 10-84
- "Getting Started with HDL Coder Native Floating-Point Support" on page 10-80
- "Floating-Point Tolerance Parameters" on page 19-26
- "Generate Target-Independent HDL Code with Native Floating-Point" on page 10-103

# Latency Values of Floating Point Operators

## In this section...

- "Math Operations" on page 10-91
- "Trigonometric and Exponential Operations" on page 10-93
- "Comparisons and Conversions" on page 10-94

HDL Coder native floating-point support can generate HDL code from your floating-point design. HDL Coder supports several Simulink blocks and math and trigonometric functions in native floating-point mode. These tables show the default latency values of these floating-point operations. You can customize these latency values. You can also customize the latency settings for most blocks and design for trade-offs between latency and Fmax by specifying custom latency values. To learn more, see "Latency Considerations with Native Floating Point" on page 10-96.

You can see the latency of these floating point operators in MATLAB by entering these commands.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');  
nfpconfig.IPCfg
```

## Math Operations

This table shows the list of basic math operations that are supported with native floating-point in HDL Coder and their latency information. The basic math operations include addition, subtraction, multiplication, and so on. You can use most of these blocks with both `single` and `double` data types. If you do not see an entry of `double` data type corresponding to a block, it means that the block does not support `double` types.

### Basic Math Operators

Simulink Blocks	Data Type	Minimum Output Latency	Maximum Output Latency
Add	Double	6	11
	Single	6	11
	Half	4	8
Subtract	Double	6	11
	Single	6	11
	Half	4	8
Product	Double	6	9
	Single	6	8
	Half	4	6
Divide	Double	31	61
	Single	17	32
	Half	10	19
Math Function	Double	30	60
	Single	16	31
Multiply-Add	Single	8	14
Rounding Function	Double	3	5
	Single	3	5
Unary Minus	Double	-	-
	Single	-	-
	Half	-	-
Sign	Double	-	-
	Single	-	-
Abs	Double	-	-
	Single	-	-

This table shows the math functions that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Math Function block. You can use these blocks with **single** data types. **Double** types are unsupported for the blocks.

## Math Functions

Simulink Blocks	Minimum Output Latency	Maximum Output Latency
HDL Reciprocal	14	21
Rem	15	24
Mod	16	26
Sqrt	16	28
Reciprocal Sqrt	16	30
Hypot	17	33

## Trigonometric and Exponential Operations

This table shows the trigonometric operations that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Trigonometric Function block. You can use these blocks with single data types. Double types are unsupported for the blocks. The Sin block has a minimum latency of 8 and a maximum latency of 14.

### Trigonometric Functions

Simulink Blocks	Minimum Output Latency	Maximum Output Latency
Sin	27	27
Cos	27	27
Tan	33	33
Sincos	27	27
Asin	17	23
Acos	17	23
Atan	36	36
Atan2	42	42
Sinh	18	30
Cosh	17	27
Tanh	25	43
Asinh	94	94
Acosh	93	93
Atanh	67	67

This table shows the exponential operations that are supported with native floating-point in HDL Coder and their latency information. You can select the function using the **Function** setting of the Math Function block. You can use these blocks with single data types. Double types are unsupported for the blocks except Log.

### Exponent/Logarithm/Power

Simulink Blocks	Data Type	Minimum Output Latency	Maximum Output Latency
Exp	Single	16	26
	Half	9	16
Pow	Single	33	54
Pow10	Single	16	26
Log	Double	34	44
	Single	20	27
	Half	9	17
Log10	Single	17	27
	Half	10	18

### Comparisons and Conversions

This table shows operations related to comparing of numbers and data type conversions that are supported with native floating-point in HDL Coder and their latency information. You can use these blocks with both `single` and `double` data types except for the `MinMax` block. This block does not support `double` data types. For the Data Type Conversion block, you can convert between `double`, `half`, and `single` data types, and between floating-point and other fixed-point data types.

#### Comparisons and Conversions

Simulink Blocks	Data Type	Minimum Output Latency	Maximum Output Latency
Data Type Conversion	Double	3	6
	Single	6	6
	Half	3	2
Relational Operator	Double	1	3
	Single	1	3
	Half	1	2
MinMax	Single	3	3

### See Also

#### Modeling Guidelines

“Modeling with Native Floating Point” on page 21-62

#### Functions

`createFloatingPointTargetConfig`

### Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Floating-Point Tolerance Parameters” on page 19-26
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120

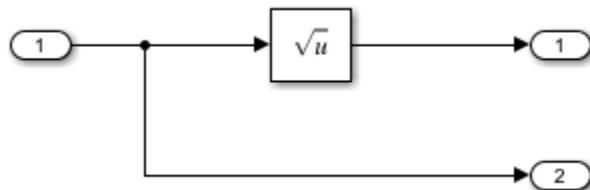
## Latency Considerations with Native Floating Point

HDL Coder™ native floating-point technology can generate HDL code from your floating-point design. Native floating-point operators have a latency. When you generate HDL code, the code generator figures out this latency and adds matching delays to balance parallel paths.

### View Latency of a Floating-Point Operator

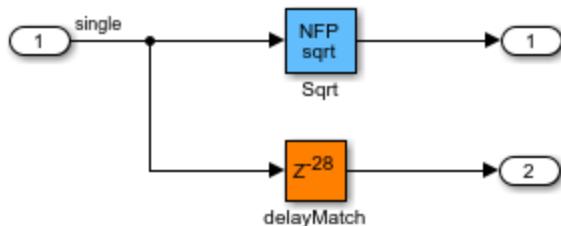
Open the `hdlcoder_nfp_delay_allocation` Simulink™ model. The model uses `single` data types and computes the square root. The model has a parallel path to illustrate how the code generator balances delays.

```
load_system('hdlcoder_nfp_delay_allocation')
open_system('hdlcoder_nfp_delay_allocation/DUT')
```

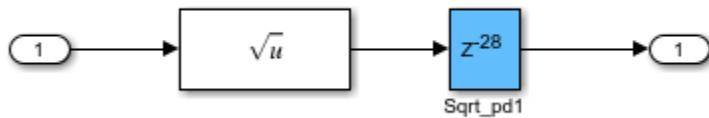


To generate HDL code:

- 1 Right-click the DUT Subsystem and select **HDL Code > Generate HDL for Subsystem**.
- 2 To see the generated model after HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation`.



The NFP\_Sqrt block is the floating-point operator corresponding to the Sqrt block in your model, and has a latency of 28. The code generator determines this latency and adds a matching delay of length 28 in the parallel path. To see the latency of the square root operation, double-click the NFP\_Sqrt block. The **Delay length** of the Sqrt\_pd1 block corresponds to the operator latency.



You can customize the latency of your design. Use custom latency settings to design for trade-offs between latency and throughput. You can then optimize your design implementation on the target FPGA device for area and speed. Customize the latency by using:

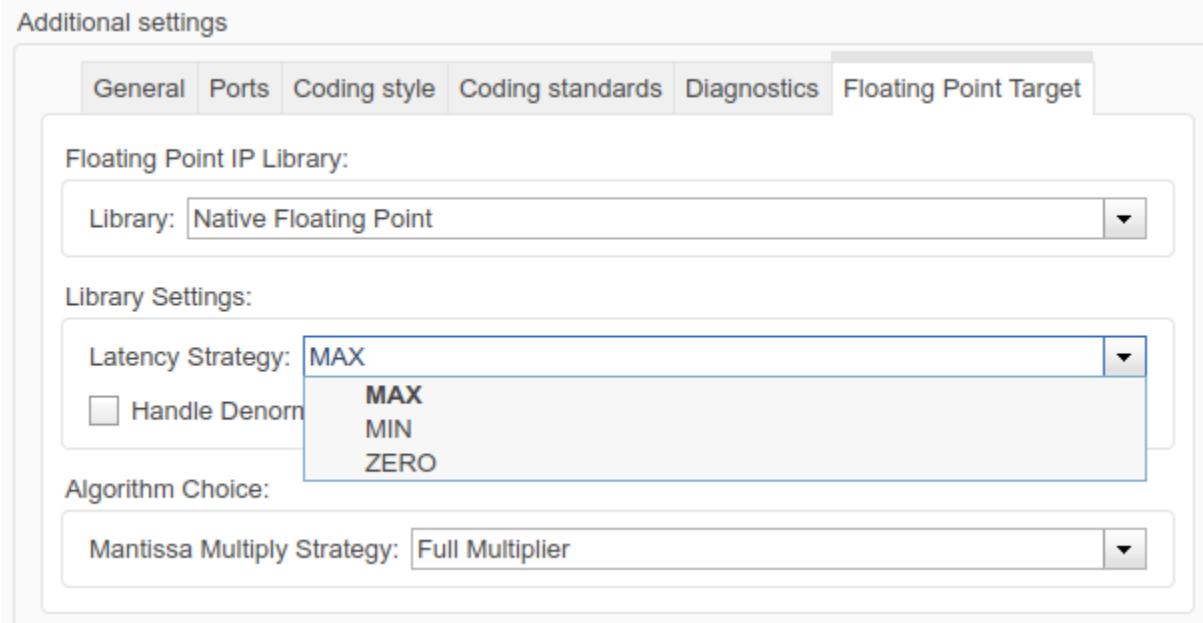
- Latency Strategy setting: Specify whether to map your entire Simulink™ model or individual blocks in your model to maximum, minimum, or zero latency of the floating-point operator.
- Custom Latency: You can specify a custom latency for certain blocks that you use in your Simulink™ model. The custom latency setting can take values from zero to the maximum latency of the floating-point operator.
- Oversampling factor: Increasing the **Oversampling factor** operates the design at a faster clock rate and absorbs the clock-rate pipelines with the latency of the floating-point operator.
- Delay blocks in the model: If your Simulink model has a latency, HDL Coder™ can absorb some or all of the latency with the native floating-point implementation.

### Latency Strategy Setting for Model

You can specify the latency strategy setting for an entire model or for individual blocks in your model.

To specify this setting for a model:

- 1 In the `hdlcoder_nfp_delay_allocation` model, right-click the DUT Subsystem and select **HDL Code > HDL Coder Properties**.
- 2 On the **HDL Code Generation > Global Settings > Floating Point Target** tab, for **Library**, select **Native Floating Point**, and then for **Latency Strategy**, select **MAX**, **MIN**, or **ZERO**.



To specify this setting from the command line:

- Create a `hdlcoder.FloatingPointTargetConfig` object for native floating point by using the `hdlcoder.createFloatingPointTargetConfig` function.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('hdlcoder_nfp_delay_allocation', 'FloatingPointTargetConfiguration', nfpconfig);
```

- Specify the latency strategy by using the `LatencyStrategy` property of the `nfpconfig` object.

```
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX'
```

```
nfpconfig =
  FloatingPointTargetConfig with properties:
```

```
    Library: 'NativeFloatingPoint'
    LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]
    IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPCfg]
```

To see the latency information, generate HDL code and then open the generated model. To open the generated model, enter the command `gm_hdlcoder_nfp_delay_allocation`.

### Custom Latency Strategy for Blocks

For blocks in your Simulink™ model, you can selectively customize the latency strategy. By default, the blocks inherit the latency strategy setting you specify for the model. For certain blocks, you can specify a custom latency value that is between zero and the maximum latency of the floating-point operator.

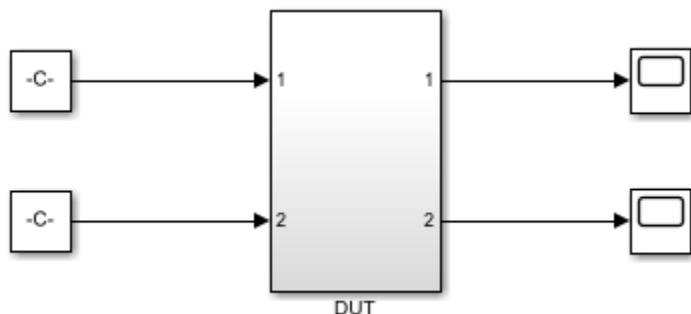
By specifying a custom latency, you can customize your design for trade-offs between:

- Clock frequency and power consumption: A higher latency value increases the maximum clock frequency (Fmax) that you can achieve, which increases the dynamic power consumption.
- Oversampling factor and sampling frequency: A combination of higher latency value and higher oversampling factor increases the Fmax that you can achieve but reduces the sampling frequency.

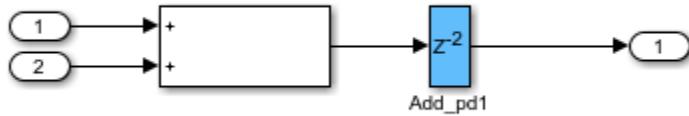
To learn more about this setting and how to specify the latency strategy for a block, see “LatencyStrategy” on page 22-33.

For example, if you have an Add block in the parallel path in your model, you can specify a custom latency value of 2 for the Add block by entering these commands.

```
load_system('hdlcoder_nfp_delay_allocation_custom')
open_system('hdlcoder_nfp_delay_allocation_custom')
hdlset_param('hdlcoder_nfp_delay_allocation_custom/DUT/Add','LatencyStrategy','Custom')
hdlset_param('hdlcoder_nfp_delay_allocation_custom/DUT/Add','NFPCustomLatency',2)
```



To see the latency information, generate HDL code and then open the generated model. To open the generated model, enter the command `gm_hdlcoder_nfp_delay_allocation_custom`. In the generated model, you see that the NFP Add block has a latency of 2.

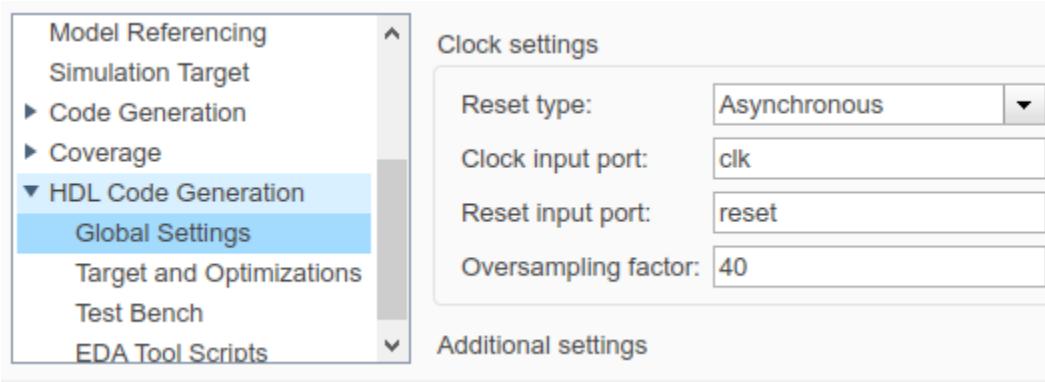


### Oversampling Factor

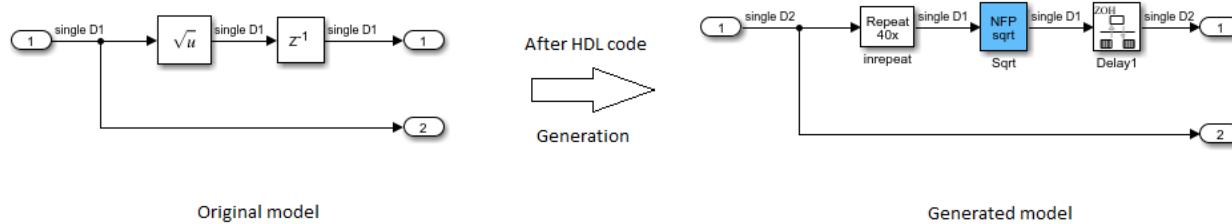
When you design the blocks in your Simulink™ model at the data rate, specify an **Oversampling factor** greater than one. The **Oversampling factor** inserts pipeline registers at a faster clock rate, which improves clock frequency and reduces area usage. To learn more about clock-rate pipelining, see “Clock-Rate Pipelining” on page 24-114.

To see the effect of **Oversampling factor** on the model, in the `hdlcoder_nfp_delay_allocation` model:

- 1 Add a Delay block with **Delay length** 1 at the output of the Sqrt block.
- 2 Right-click the DUT and select **HDL Code > HDL Coder Properties**.
- 3 On the **HDL Code Generation > Global Settings** pane, enter a value of 40 for **Oversampling factor**.



After HDL code generation, the generated model shows the NFP Sqrt block operating at a clock rate that is 40 times faster than the Sqrt block in your model. The NFP Sqrt block absorbed the Delay block in your Simulink™ model. The Delay block now operates at the clock rate. This implementation saves area by absorbing the additional latency, and improves timing by operating at the faster clock rate.



### Delay Absorption in the Model

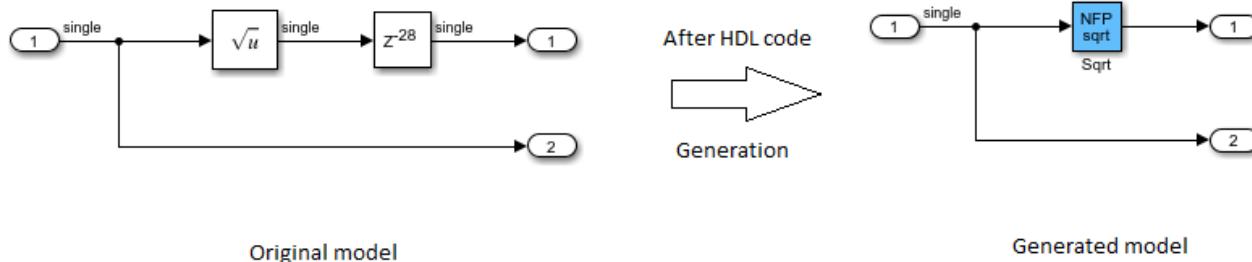
If your Simulink™ model has a Delay block with sufficient **Delay length** adjacent to an operator, HDL Coder™ absorbs the delays as part of the operator latency.

**Note:** To absorb delays, make sure that you group the delays adjacent to the block.

If the **Delay length** is equal to the latency of the floating-point operator, HDL Coder™ absorbs the delays and does not introduce any additional latency.

In the `hdlcoder_nfp_delay_allocation` model:

- 1 Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to 28.
- 2 Generate HDL code for the DUT Subsystem.
- 3 After HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.

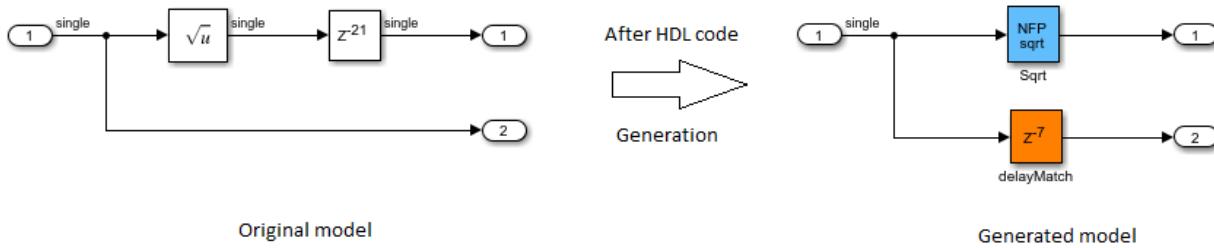


In the generated model, you see that the NFP Sqrt block absorbs the Delay block adjacent to the Sqrt block in your original model. This delay absorption occurs because the operator latency is equal to the **Delay length**. The code generator therefore avoids the additional latency in your model.

If the **Delay length** is less than the operator latency, HDL Coder™ absorbs the available delays and balances parallel paths by adding matching delays.

In the `hdlcoder_nfp_delay_allocation` model:

- 1 Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to 21.
- 2 Generate HDL code for the DUT Subsystem.
- 3 After HDL code generation, at the command line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.

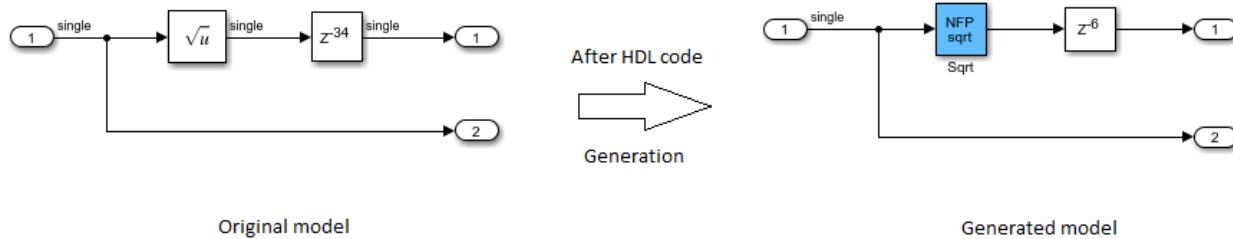


You see that the NFP Sqrt block absorbed a Delay of length 21 and added a matching delay of length 7 in the parallel path because the square root operation requires 28 delays.

If the delay length is greater than the operator latency, the code generator absorbs a certain number of delays equal to the latency and the excess delays appear outside the operator.

In the `hdlcoder_nfp_delay_allocation` model:

- 1 Double-click the Delay block at the output of the Sqrt block and change the **Delay length** to 34.
- 2 Generate HDL code for the DUT Subsystem.
- 3 After HDL code generation, at the command-line, enter `gm_hdlcoder_nfp_delay_allocation` to open the generated model.



The NFP Sqrt block absorbed 28 delays because the square root operation has a latency of 28. The excess latency of 6 is outside the operator.

## See Also

### Modeling Guidelines

“Modeling with Native Floating Point” on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Latency Values of Floating Point Operators” on page 10-91
- “Floating-Point Tolerance Parameters” on page 19-26
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120

# Generate Target-Independent HDL Code with Native Floating-Point

## In this section...

["How HDL Coder Generates Target-Independent HDL Code" on page 10-103](#)

["Enable Native Floating Point and Generate Code" on page 10-104](#)

["View Code Generation Report" on page 10-105](#)

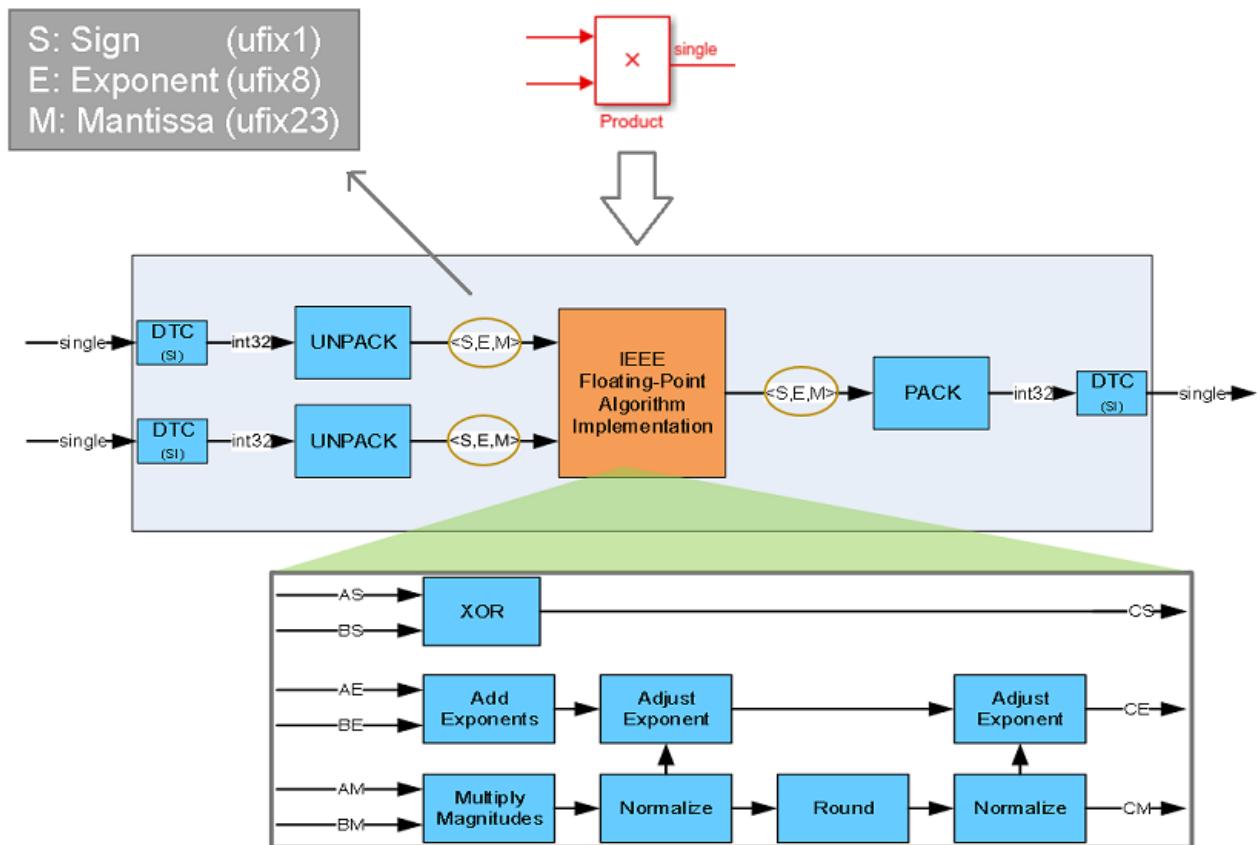
["Analyze Results" on page 10-106](#)

["Limitation" on page 10-108](#)

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

## How HDL Coder Generates Target-Independent HDL Code

This figure shows how HDL Coder generates code with the native floating-point technology.



The Unpack and Pack blocks convert the floating-point types to the sign, exponent, and mantissa. In the figure,  $S$ ,  $E$ , and  $M$  represent the sign, exponent, and mantissa respectively. This interpretation is based on the IEEE-754 standard of floating-point arithmetic.

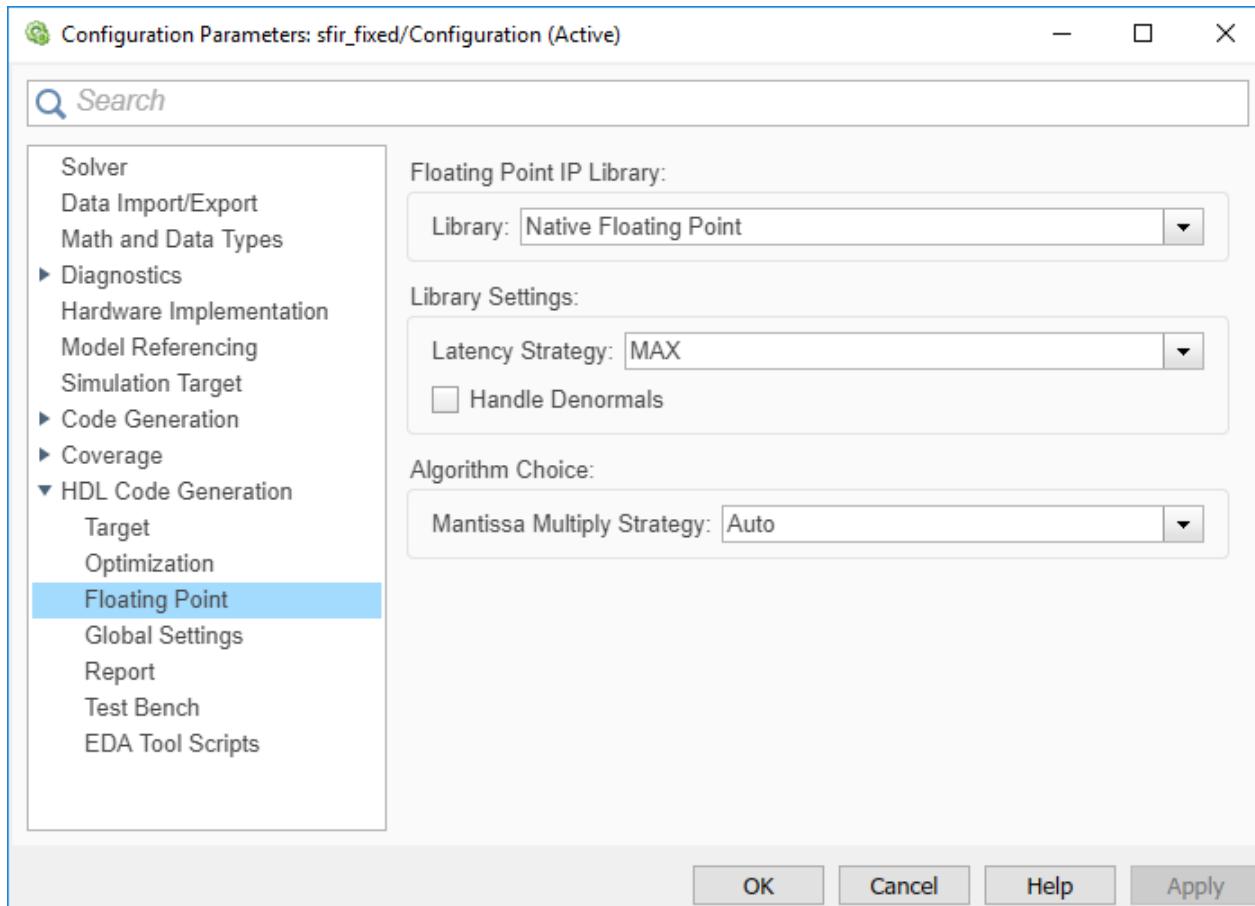
The **Floating-Point Algorithm Implementation** block performs computations on the  $S$ ,  $E$ , and  $M$ . With this conversion, the generated HDL code is target-independent. You can deploy the design on any generic FPGA or an ASIC.

## Enable Native Floating Point and Generate Code

You can generate code in the Configuration Parameters dialog box or at the command line.

To specify the native floating-point settings and generate HDL code in the Configuration Parameters dialog box:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Click **Settings**. In the **HDL Code Generation > Floating Point** pane, for **Library**, select **Native Floating Point**.



- 3 Specify the **Latency Strategy** to map your design to maximum or minimum latency or no latency.
- 4 If you have denormal numbers in your design, select **Handle Denormals**. Denormal numbers are numbers that have an exponent field equal to zero and a nonzero mantissa field. See "Handle Denormals" on page 16-5.

- 5 If your design has multipliers, to specify how you want HDL Coder to implement the multiplication operation, use the **Mantissa Multiplier Strategy**. See “Mantissa Multiplier Strategy” on page 16-6.
- 6 To share floating-point resources, on the **HDL Code Generation > Optimizations > Resource Sharing** tab, make sure that you select **Floating-point IPs**. The number of blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.
- 7 Click **Apply**. In the **HDL Code** tab, click **Generate HDL Code**.

To generate HDL code at the command line, use the `hdlcoder.createFloatingPointTargetConfig` function. You can use this function to create an `hdlcoder.FloatingPointTargetConfig` object for the native floating-point library.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', nfpconfig);
```

Optionally, you can specify the latency strategy and whether you want HDL Coder to handle denormal numbers in your design:

```
nfpconfig.LibrarySettings.HandleDenormals = 'on';
nfpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

To learn how you can verify the generated code, see “Verify the Generated Code from Native Floating-Point” on page 10-116.

## View Code Generation Report

To view the code generation reports of floating-point library mapping, before you begin code generation, enable generation of the Resource Utilization Report and Optimization Report. To enable the reports, on the **HDL Code** tab, click **Settings > Report Options** in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, enable **Generate resource utilization report** and **Generate optimization report**. See also “Create and Use Code Generation Reports” on page 25-2.

To see the list of native floating-point operators that HDL Coder supports and the floating-point operators to which your Simulink blocks mapped to, in the Code Generation Report, select **Native Floating-Point Resource Report**.

### Native Floating-Point Resource Report for sfir\_single

#### Summary of single precision native floating-point operators

adders	7
multipliers	4

A detailed report shows the various resources that the floating-point blocks use on the target device that you specify. See also “Create and Use Code Generation Reports” on page 25-2.

## Detailed Report

### Module nfp\_add\_comp

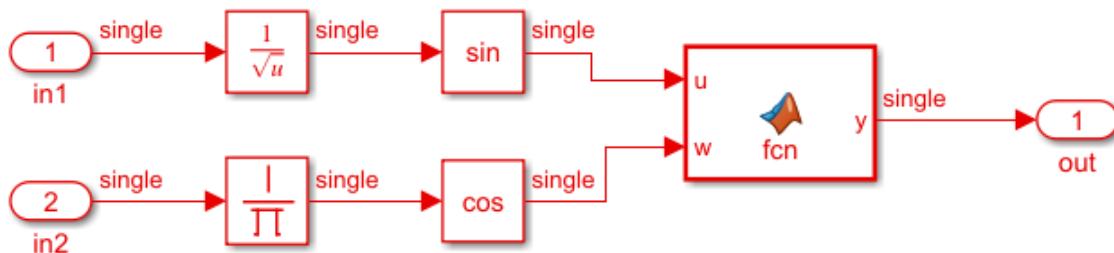
(Latency = "Max")

Multipliers	0
Adders/Subtractors	8
Registers	91
Total 1-Bit Registers	839
RAMs	0
Multiplexers	109
Static Shift operators	0
Dynamic Shift operators	2

To see the native floating-point settings that you applied to the model and whether HDL Coder successfully generated HDL code, in the Code Generation Report, select **Target Code Generation**.

## Analyze Results

Floating point operators have a latency. If your Simulink model does not have delays, when you generate HDL code, the code generator figures out the operator latency and delay balances parallel paths. Consider this Simulink model that has two single inputs and gives a single output.



The MATLAB Function block in the Simulink model contains this code.

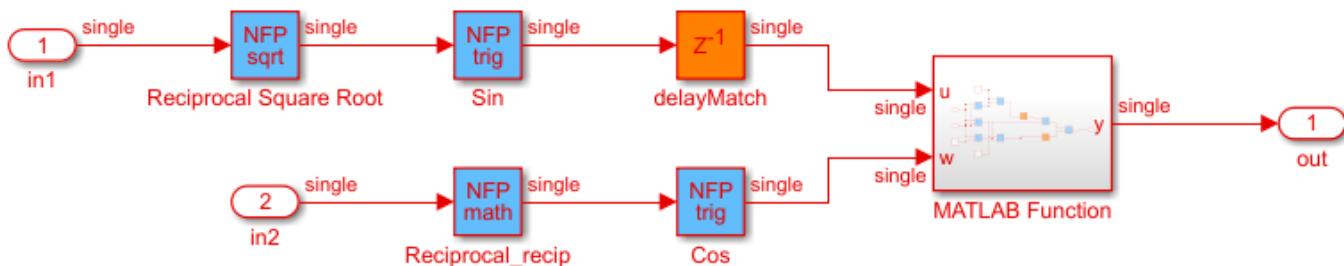
```

function y = fcn(u, w)
 %#codegen

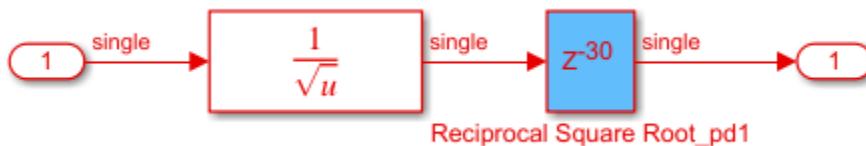
y1 = (u+w) * 20;
y2 = w^16;
y3 = (u-w) / 10;
y = y1 + y2 - y3;

```

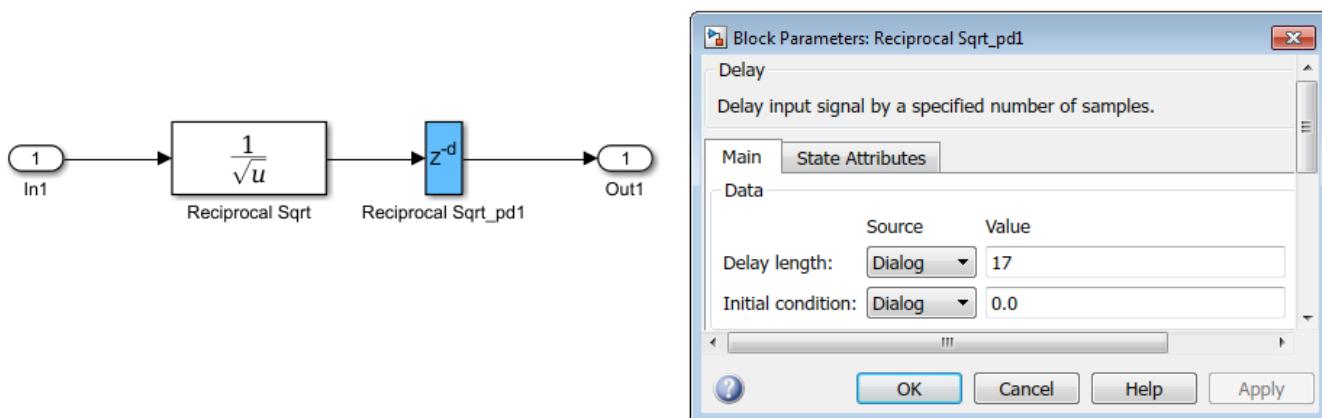
When you generate HDL code, the code generator maps the blocks in your Simulink model to synthesizable native floating-point operators. To see how the code generator implemented the floating-point operations, open the generated model. The blocks **NFP math**, **NFP Sqrt**, and **NFP trig** correspond to the floating-point implementation of the Reciprocal Sqrt, Reciprocal, sin, and cos blocks respectively in your original model.



Every floating-point operator has a latency. The code generator inserted an additional matching delay because the latency of the Reciprocal Sqrt is 30 and latency of Reciprocal is 31. The operator latency is equal to the **Delay length** of the Delay block inside that **NFP** block. For example, if you double-click the **NFP\_sqrt** block, you can get the latency by looking at the **Delay length** of the Delay block. See “Latency Values of Floating Point Operators” on page 10-91.



When you use MATLAB Function blocks with floating-point data types, HDL Coder uses the **MATLAB Datapath** architecture. This architecture treats the MATLAB Function block like a regular Subsystem block. When you generate code, the code generator maps the basic operations such as addition and multiplication to the corresponding native floating-point operators. Open the MATLAB Function subsystem to see how the code generator implemented the MATLAB Function block.



To learn more about the generated model, see “Generated Model and Validation Model” on page 24-10.

## Limitation

To generate HDL code in native floating-point mode, use discrete sample times. Blocks operating at a continuous sample time are not supported.

## See Also

### **Modeling Guidelines**

“Modeling with Native Floating Point” on page 21-62

### **Functions**

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120

# Floating Point Support: Field-Oriented Control Algorithm

In this example you review a Field-Oriented Control (FOC) algorithm for a Permanent Magnet Synchronous Machine (PMSM) implemented using single-precision and half-precision floating-point types.

## Introduction

You have seen the fixed-point version of this design in “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 10-55 that takes a deep dive into how to implement current control algorithm using fixed-point types. The model was converted to fixed-point before generating HDL code.

You can use floating-point single-precision types in your design and generate HDL code natively without converting to fixed-point types. This example shows design considerations when generating code from floating-point single-precision and half-precision variants\* of the fixed point model `hdlcoderFocCurrentFixptHdl`.

The testbench model '`hdlcoderFocCurrentTestBench`' has reference block pointing to the DUT implemented in fixed-point or floating-point.

## Salient features of Native Floating Point support.

- Vendor independent and target agnostic RTL for FPGA and/or ASIC design
- Full range of IEEE-754 features including support for rounding modes, inf and nan data types, and optional support for denormal numbers.
- Extensive math block (add,mul,div,recip,log,exp,sqrt, rsqrt) and trigonometric block (sin, cos, sincos, atan, atan2) support.

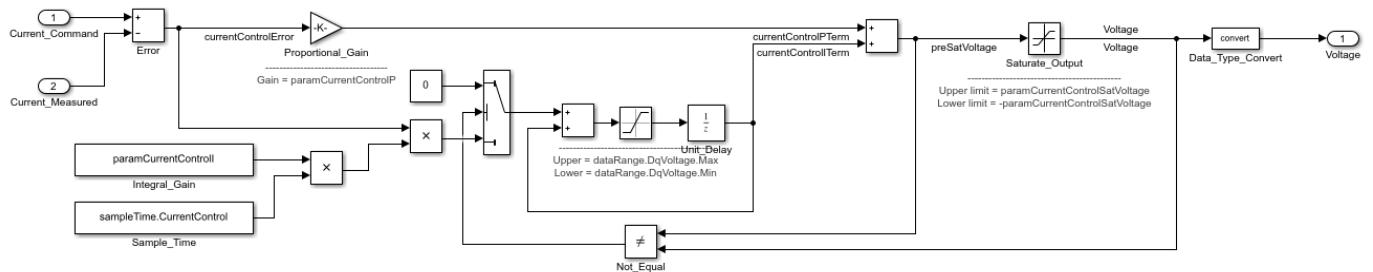
## Why Use Floating-Point Types

Sometimes you may want to start and stay in floating-point to target HDL for the following reasons:

- Your algorithms have large or unknown dynamic ranges (for example integrators in feedback loops)
- Your algorithm uses operations that are difficult to design in fixed-point (ex: atan2)

In the fixed-point version of the example `hdlcoderFocCurrentFloatHDL.slx` you notice that several fixed-point rounding and saturation decisions are made to preserve the numerical behavior of the algorithm. For example the block '`hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Saturate`' which is a saturation block has been placed in the integrator loop so that results do not overflow due to accumulation in the loop.

```
open_system('hdlcoderFocCurrentFixptHdl');
open_system('hdlcoderFocCurrentFixptHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Saturate')
```



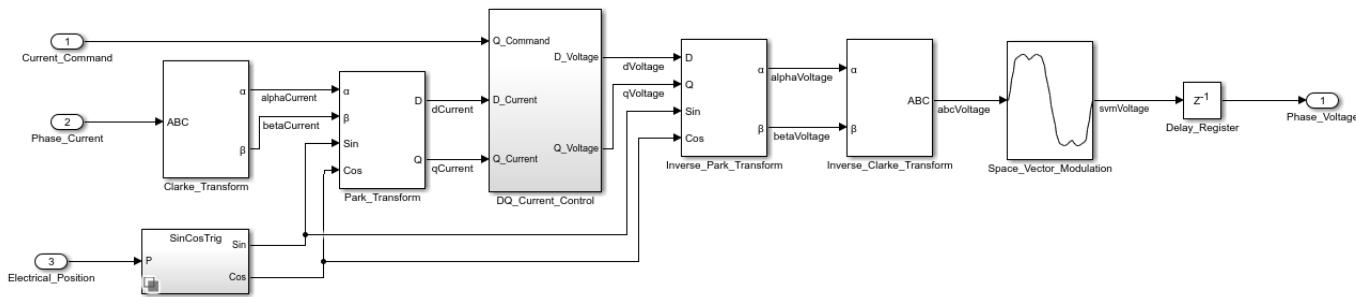
Sometimes, the task of fixed-point conversion could take several weeks to months with multiple levels of algorithm re-validation. It may also lead to undesirable loss of precision which might not be acceptable for some mission critical applications requiring very high accuracy.

In these situations, you can choose to use native floating point synthesis features available in HDL Coder.

### Single-Precision FOC Model

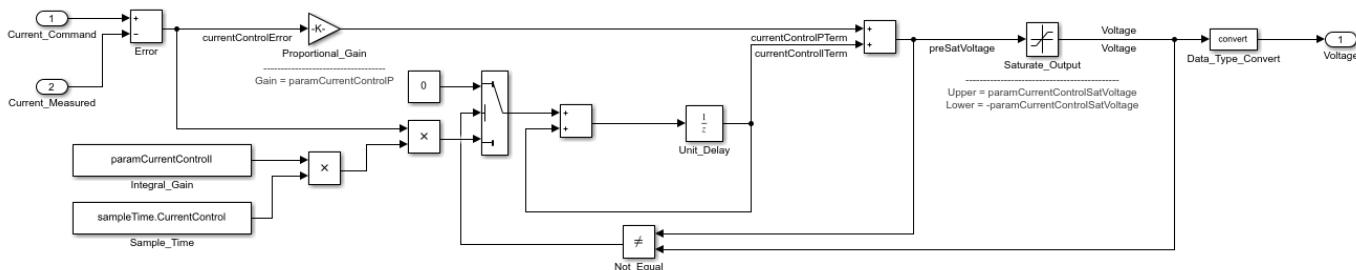
To open the single-precision version of the algorithm, run these commands:

```
load_system('hdlcoderFocCurrentFloatHdl');
open_system('hdlcoderFocCurrentFloatHdl/FOC_Current_Control')
```



In comparison to the fixed-point algorithm, the single-precision model does not require additional rounding and saturation blocks and settings as can be seen in the floating-point version of the model 'hdlcoderFocCurrentFloatHdl/FOC\_Current\_Control/DQ\_Current\_Control/D\_Current\_Control'

```
open_system('hdlcoderFocCurrentFloatHdl/FOC_Current_Control/DQ_Current_Control/D_Current_Control')
```



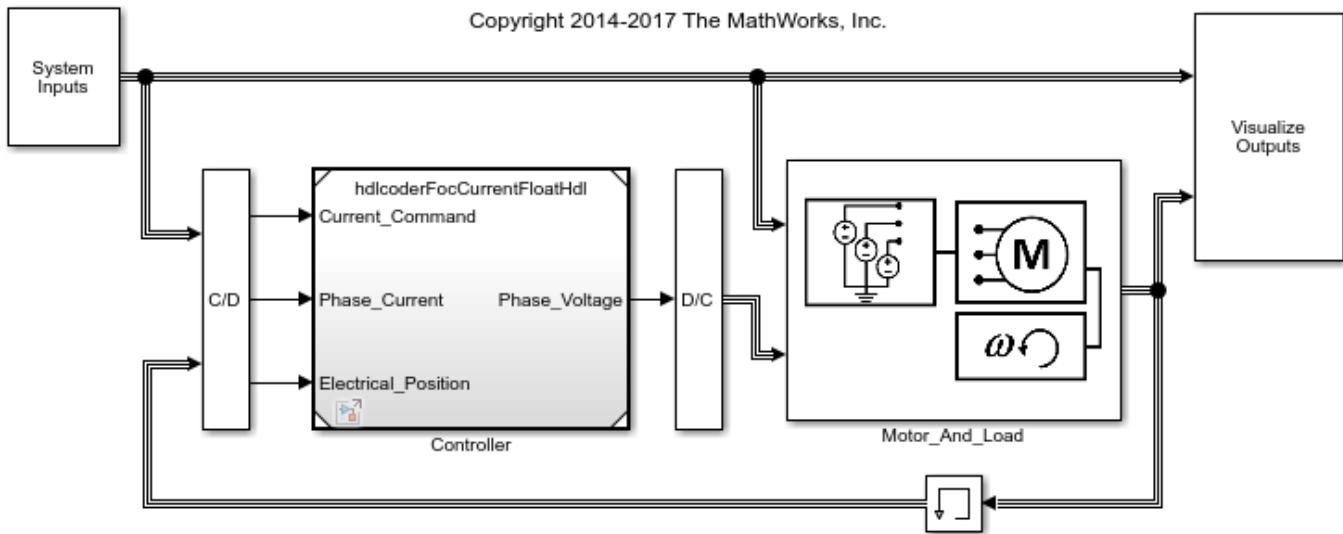
To verify the behavior through simulation, run these commands:

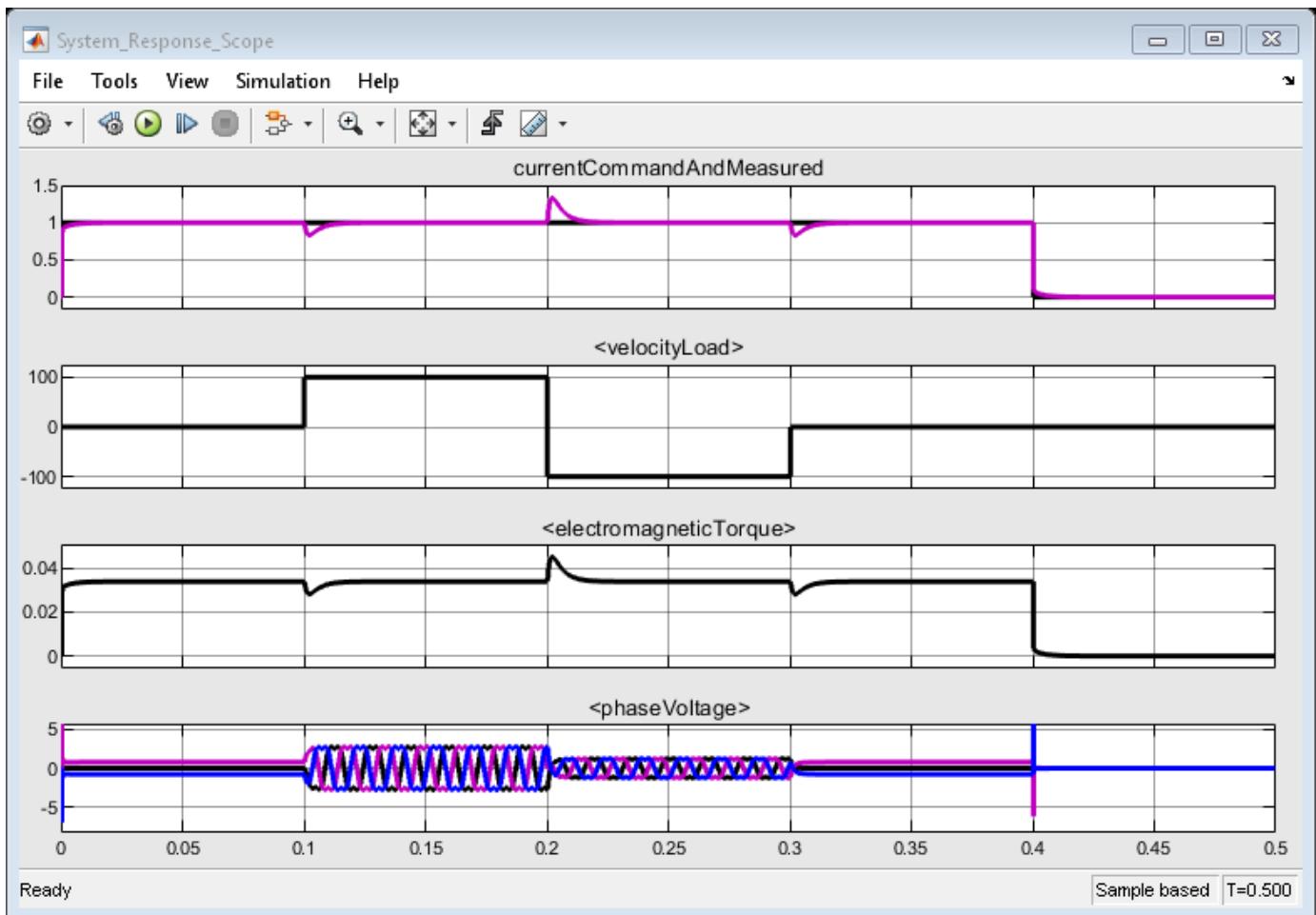
```
hasSimPowerSystems = license ('test', 'Power_System_Blocks');
if hasSimPowerSystems
    open_system('hdlcoderFocCurrentTestBench')

    % set single-precision floating-point model in the testbench
    set_param('hdlcoderFocCurrentTestBench/Controller', 'ModelName', 'hdlcoderFocCurrentFloatHdl');

    set_param('hdlcoderFocCurrentTestBench', 'IgnoredZcDiagnostic','none');
    sim('hdlcoderFocCurrentTestBench')
    set_param('hdlcoderFocCurrentTestBench', 'IgnoredZcDiagnostic','warn');
end
```

## Field-Oriented Control Current Control Test Bench





You can generate HDL code and view the generated code for the controller.

```

hdlset_param('hdlcoderFocCurrentFloatHdl', 'FloatingPointTargetConfiguration', hdlcoder.createFl
makehdl('hdlcoderFocCurrentFloatHdl/FOC_Current_Control');

### Generating HDL for 'hdlcoderFocCurrentFloatHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCur
### Starting HDL check.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script:
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcode
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocCurrentFloatHdl_
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFloatHdl'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_sincos_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdlsrc\hdlcode
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_add_single as hdlsrc\hdlcoderF
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_gain_pow2_single as hdlsrc\hdlco
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_sub_single as hdlsrc\hdlcoderF
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_add2_single as hdlsrc\hdlcoderF
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_mul_single as hdlsrc\hdlcoderF

```

```
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_uminus_single as hdsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control\nfp_uminus_single
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control/nfp_relop_single as hdsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control\nfp_relop_single
### Working on FOC_Current_Control_tc as hdsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control_tc
### Working on hdlcoderFocCurrentFloatHdl/FOC_Current_Control as hdsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control
### Generating package file hdsrc\hdlcoderFocCurrentFloatHdl\FOC_Current_Control_pkg.vhd
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\top_level.html')">
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\top_level.html</a>
### HDL check for 'hdlcoderFocCurrentFloatHdl' complete with 0 errors, 0 warnings, and 2 messages
### HDL code generation complete.
```

## Half-Precision FOC Model

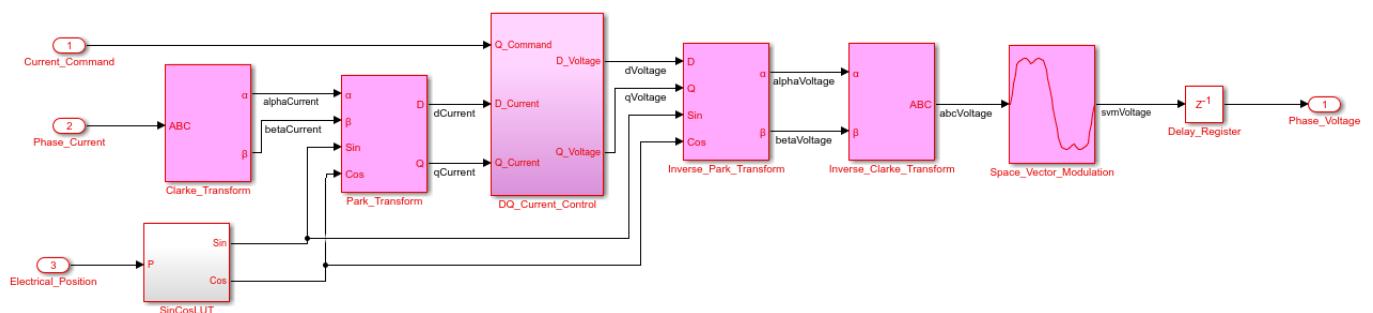
For applications that require smaller dynamic range, you can use **half** types without having to convert your design to use fixed-point types. Using **half** types consumes much less memory, has lower latency, and saves FPGA resources.

Advantages of half-precision floating point types in hardware perspective:

- Low latency
- Low Area
- High speed
- 16-bit floating point behavior which will be useful for optimal storage
- Wider dynamic range compared to integer or fixed point data types of same size

Half-precision types have same features as single- and double-precision floating-point types. Supported operators include basic arithmetic operators(EX: Add/Sub, Mul, Div/Recip) and Gain, Relational operators, Data type conversions.

```
load_system('hdlcoderFocCurrentFloatHalfHdl');
open_system('hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control')
```

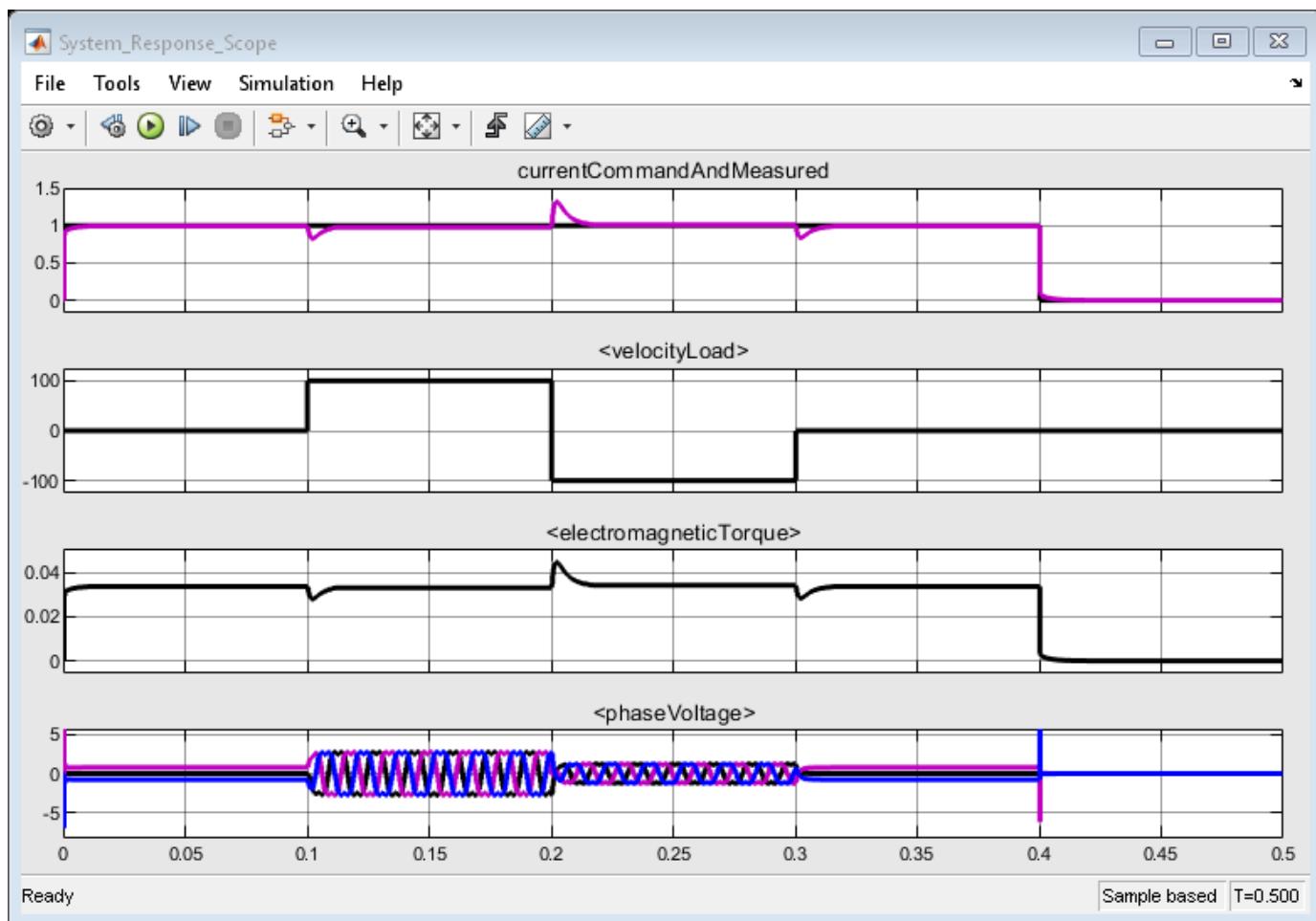


To verify the behavior through simulation, run these commands:

```
hasSimPowerSystems = license('test', 'Power_System_Blocks');
if hasSimPowerSystems
    open_system('hdlcoderFocCurrentTestBench')

    % set half-precision floating-point model in the testbench
    set_param('hdlcoderFocCurrentTestBench/Controller', 'ModelName', 'hdlcoderFocCurrentFloatHalfHdl');

    set_param('hdlcoderFocCurrentTestBench', 'IgnoredZcDiagnostic', 'none');
    sim('hdlcoderFocCurrentTestBench')
    set_param('hdlcoderFocCurrentTestBench', 'IgnoredZcDiagnostic', 'warn');
end
```



You can generate HDL code and view the generated code for the controller.

```

hdlset_param('hdlcoderFocCurrentFloatHalfHdl', 'FloatingPointTargetConfiguration', hdlcoder.createConfigSet);
makehdl('hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control');

### Generating HDL for 'hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control')>.
### Starting HDL check.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoder\highlightDistributedPipelining')>.
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoder\clearHighlighting')>.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocCurrentFloatHalfHdl')>.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocCurrentFloatHalfHdl'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_relop_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_relop_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_add_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_add_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_gain_pow2_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_gain_pow2_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_sub_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_sub_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_add2_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_add2_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_mul_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_mul_half
### Working on hdlcoderFocCurrentFloatHalfHdl/FOC_Current_Control/nfp_uminus_half as hdlsrc\hdlcoder\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_uminus_half

```

```
### Working on hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_relop_half as hdlsrc\hdlcod
### Working on hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_convert_sfix_16_En14_to_ha
### Working on hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control\nfp_convert_half_to_sfix_16_En
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control
### Working on hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control as hdlsrc\hdlcoderFocCurrentFl
### Generating package file hdlsrc\hdlcoderFocCurrentFloatHalfHdl\FOC_Current_Control_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tph
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tph
### HDL check for 'hdlcoderFocCurrentFloatHalfHdl' complete with 0 errors, 0 warnings, and 2 mess
### HDL code generation complete.
```

You can refer to documentation for full native floating-point capabilities available in HDL Coder. See link in the “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80.

## Verify the Generated Code from Native Floating-Point

### In this section...

["Specify the Tolerance Strategy" on page 10-116](#)

["Verify the Generated Code with HDL Test Bench" on page 10-117](#)

["Verify the Generated Code with Cosimulation" on page 10-117](#)

["Limitation" on page 10-119](#)

HDL Coder native floating-point technology can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

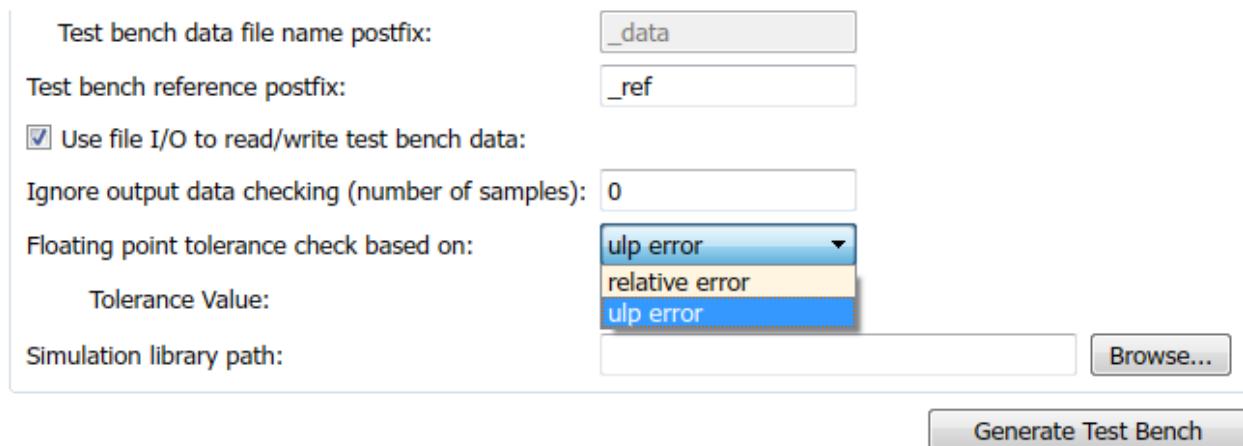
When representing infinitely real numbers with a finite number of bits, there can be rounding errors with the correct rounding range of values that the IEEE-754 standard specifies. To measure the rounding errors, you can specify the floating-point tolerance check based on `relative error` or `ulp error`. For more information about these rounding errors, see ["Relative Accuracy and ULP Considerations" on page 10-85](#).

### Specify the Tolerance Strategy

Before generating the testbench, specify the floating-point tolerance check for verifying the generated code.

To specify the tolerance check in the Configuration Parameters dialog box:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Click **Settings**. In the **HDL Code Generation > Testbench** pane, for **Floating point tolerance check based on**, specify `relative error` or `ulp error`.



- 3 Enter the **Tolerance Value** and click **Apply**. If you choose `relative error`, the default is a tolerance value of `1e-07`. If you choose `ulp error`, the default tolerance value is zero. To learn more, see “Numeric Considerations with Native Floating-Point” on page 10-84.

To specify the tolerance strategy at the command-line, use:

- 1 Specify the floating point tolerance check setting by using `FPToleranceStrategy`.

```
% check for floating-point tolerance based on relative error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'Relative');
```

```
% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');
```

- 2 Based on the `FPToleranceStrategy` setting, enter the tolerance value by using `FPToleranceValue`.

```
% if using relative error, enter custom tolerance value
hdlset_param('FP_test_16a', 'FPToleranceValue', 1e-06);
```

```
% if using ULP error, enter tolerance value greater
% than or equal to 1
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

## Verify the Generated Code with HDL Test Bench

To generate an HDL test bench for verifying the generated code:

- 1 In the Configuration Parameters dialog box, on the **HDL Code Generation > Test Bench** pane, in the Test Bench Generation Output section, select **HDL test bench**.
- 2 In the Configuration section, make sure that **Use file I/O to read/write test bench data** is enabled. To generate a test bench that uses constants instead of file I/O, clear **Use file I/O to read/write test bench data**.
- 3 Click **Apply**, and then click **Generate Test Bench**.

To learn more about how HDL test bench generation works, see “Test Bench Generation” on page 6-5.

## Verify the Generated Code with Cosimulation

To generate a cosimulation model for verifying the generated code:

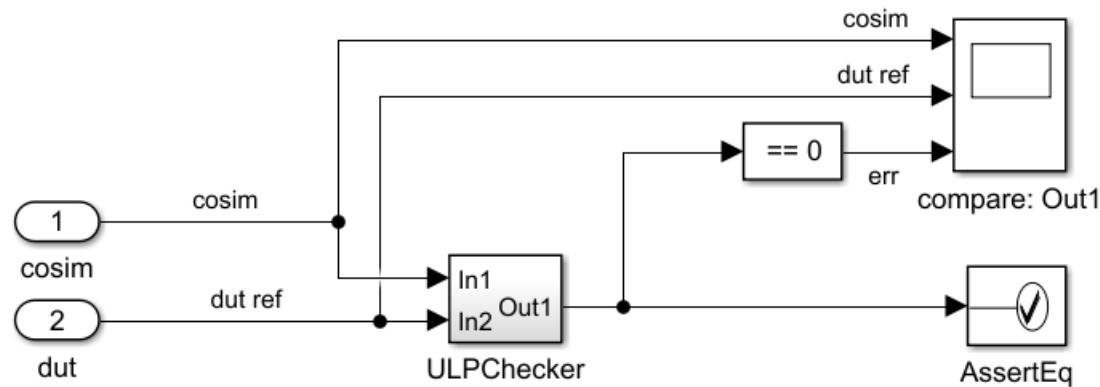
- 1 In the Configuration Parameters dialog box, on the **HDL Code Generation > Test Bench** pane, for **Cosimulation model for use with**, select the cosimulation tool.
- 2 Click **Apply**, and then click **Generate Test Bench**.
- 3 After test bench generation, save the cosimulation model. In the model, double-click the `Compare` subsystem.

Double click to turn 'OFF' all Assertions

Double click to turn 'on/off' all scopes

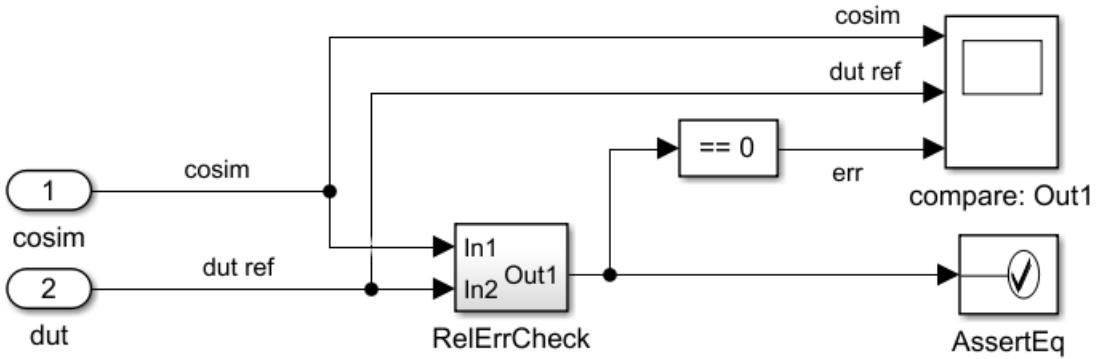


- 4 If you double-click the **Assert\_Out1** block, the block parameters show the **ToleranceValue** that you specify.
- 5 To look inside the **Assert\_Out1** block, click the mask. If you specify the floating-point tolerance check based on `ulp_error`, the model shows a **ULPChecker** block.



The **ULPChecker** has a MATLAB Function block that shows how HDL Coder accounts for the ULP error when checking for numerical accuracy.

If you specify the floating-point tolerance check based on `relative_error`, the model shows a **RelErrCheck** block.



`RelErrCheck` has a MATLAB Function block that shows how HDL Coder accounts for the relative error when checking for numerical accuracy.

- 6 In the Simulink Editor for the model, start simulation. At the end of cosimulation, check the `compare: Out1` scope.

The scope compares the difference between the result signal from the cosimulation block and the reference signal from the DUT.

See also “Generate a Cosimulation Model” on page 27-41.

## Limitation

When verifying the generated code, constructs that use IEEE standards prior to VHDL-2008 are not supported with native floating-point.

## See Also

### Modeling Guidelines

“Modeling with Native Floating Point” on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Numeric Considerations with Native Floating-Point” on page 10-84
- “Floating-Point Tolerance Parameters” on page 19-26
- “Simulink Blocks Supported with Native Floating-Point” on page 10-120

## Simulink Blocks Supported with Native Floating-Point

### In this section...

- “HDL Floating Point Operations Library” on page 10-120
- “Supported Simulink Blocks in Math Operations Library” on page 10-120
- “Supported Simulink Blocks in Other Libraries” on page 10-121
- “Simulink Block Restrictions” on page 10-123

HDL Coder native floating-point can generate target-independent HDL code from your floating-point design. You can synthesize your floating-point design on any generic FPGA or ASIC. Floating-point designs have better precision, higher dynamic range, and a shorter development cycle than fixed-point designs. If your design has complex math and trigonometric operations, use native floating-point technology.

HDL Coder supports several Simulink blocks including math and trigonometric blocks with native floating-point technology.

### HDL Floating Point Operations Library

In the **HDL Floating Point Operations** library, HDL Coder supports all blocks that have `single` and `double` data types in the `Native Floating Point` mode. For certain blocks such as Discrete-Time Integrator and Discrete PID Controller, use zero latency strategy. These blocks contain inherent feedback loops and using a non-value for the latency strategy can result in the code generator being unable to allocate delays. For more information, see “Allocate Sufficient Delays for Floating-Point Operations” on page 10-67.

When you use `half` types, these blocks are supported in `Native Floating Point` mode.

- Add
- Divide
- Float Typecast
- Gain
- Product
- Product of Elements
- Reciprocal
- Relational Operator
- Subtract
- Sum of Elements
- Transpose
- Unary Minus

### Supported Simulink Blocks in Math Operations Library

In the `Math Operations` library, these blocks are supported for HDL code generation:

Block Name	Supported with half data types	Supported with single data types	Supported with double data types	Remarks
Abs	No	Yes	Yes	-
Add	Yes	Yes	Yes	-
Assignment	No	Yes	Yes	-
Bias	No	Yes	Yes	-
Complex to Real-Imag	No	Yes	Yes	-
Divide	Yes	Yes	Yes	-
Dot Product	No	Yes	Yes	-
Gain	Yes	Yes	Yes	-
Math Function	No	Yes	No	-
MinMax	No	Yes	Yes	-
Product	Yes	Yes	Yes	If you configure the block to perform matrix multiplication by setting the <b>Multiplication</b> block parameter to <b>Matrix(*)</b> , the <b>DotProductStrategy</b> must be set to <b>Fully Parallel</b> .
Product of Elements	Yes	Yes	Yes	-
Real-Imag to Complex	No	Yes	Yes	-
Reciprocal Sqrt	No	Yes	Yes	-
Reshape	Yes	Yes	Yes	-
Sqrt	No	Yes	Yes	-
Subtract	Yes	Yes	Yes	-
Sum	Yes	Yes	Yes	-
Sum of Elements	Yes	Yes	Yes	-
Trigonometric Function	No	Yes	No	-
Unary Minus	Yes	Yes	Yes	-
Vector Concatenate	Yes	Yes	Yes	-

## Supported Simulink Blocks in Other Libraries

The table shows the list of supported blocks for HDL code generation in other **HDL Coder** block libraries.

<b>Block Library</b>	<b>Supported with half data types</b>	<b>Supported with single data types</b>	<b>Supported with double data types</b>
Discrete	The supported blocks include the set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.	The supported blocks include: <ul style="list-style-type: none"> <li>• Zero Order Hold</li> <li>• Set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.</li> </ul>	The supported blocks include: <ul style="list-style-type: none"> <li>• Zero Order Hold</li> <li>• Set of delay blocks including the synchronous blocks, and Integer Delay, and Tapped Delay.</li> </ul>
HDL Operations	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
HDL RAMs	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
HDL Subsystems	All blocks are supported.	All blocks are supported.	All blocks are supported.
Logic and Bit Operations	Supported blocks include: <ul style="list-style-type: none"> <li>• Compare to Constant</li> <li>• Relational Operator</li> <li>• Detect Change</li> <li>• Detect Increase</li> <li>• Detect Decrease</li> </ul>	All blocks are supported.	All blocks are supported.
Lookup Tables	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Model Verification	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Model-Wide Utilities	Blocks in this library are not supported.	All blocks are supported.	All blocks are supported.
Ports & Subsystems	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.	Enable, reset, input, and output ports, model references, and subsystem blocks are supported.
Signal Attributes	Supported blocks include: <ul style="list-style-type: none"> <li>• Data Type Conversion</li> <li>• Data Type Duplicate</li> <li>• Rate Transition</li> <li>• Signal Specification</li> </ul>	All blocks are supported.	All blocks are supported.

Block Library	Supported with half data types	Supported with single data types	Supported with double data types
Signal Routing	All blocks are supported.	All blocks are supported.	All blocks are supported.
Sources	The supported blocks include Import, Constant, and Ground blocks.	The supported blocks include Import, Constant, and Ground blocks.	The supported blocks include Import, Constant, and Ground blocks.
Sinks	All blocks are supported.	All blocks are supported.	All blocks are supported.
User-Defined Functions	MATLAB Function blocks are supported.	MATLAB Function blocks are supported.	MATLAB Function blocks are supported.

## Simulink Block Restrictions

In native floating-point mode, the code generator does not support these blocks or block architectures:

- Biquad Filter.
- Switch block with input to the control port as a floating-point type.
- Sum of Elements with complex input types.
- MATLAB System blocks.
- Dot Product in complex mode with **Architecture** as Tree or Linear.
- Discrete FIR Filter with **Architecture** other than Fully Parallel.
- Dead Zone and Dead Zone Dynamic.
- Polar to Cartesian.
- For the Data Type Conversion block:
  - **Stored Integer (SI)** mode for **Input and output to have equal** setting is not supported.
  - The **Saturate on integer overflow** check box must be left cleared.

## See Also

### Modeling Guidelines

“Modeling with Native Floating Point” on page 21-62

### Functions

`createFloatingPointTargetConfig`

## Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

## More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80

- “Numeric Considerations with Native Floating-Point” on page 10-84
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

# Supported Data Types and Scope

In this section...
"Supported Data Types" on page 10-125
"Unsupported Data Types" on page 10-126
"Scope for Variables" on page 10-126

## Supported Data Types

HDL Coder supports the following subset of MATLAB data types.

Types	Supported Data Types	Restrictions
Integer	<ul style="list-style-type: none"> <li>• <code>uint8</code>, <code>uint16</code>, <code>uint32</code>, <code>uint64</code></li> <li>• <code>int8</code>, <code>int16</code>, <code>int32</code>, <code>int64</code></li> </ul>	In Simulink, MATLAB Function block ports must use numeric types <code>sfix64</code> or <code>ufix64</code> for 64-bit data.
Real	<ul style="list-style-type: none"> <li>• <code>double</code></li> <li>• <code>single</code></li> </ul>	<p>HDL code generated with <code>double</code> or <code>single</code> data types can be used for simulation, but is not synthesizable.</p> <p>When you have floating-point data types, to generate synthesizable HDL code, use:</p> <ul style="list-style-type: none"> <li>• HDL Coder native floating-point when you want to deploy the generated code on any generic ASIC or FPGA. To learn more, see "Getting Started with HDL Coder Native Floating-Point Support" on page 10-80.</li> <li>• FPGA floating-point target libraries when you want to map the Simulink model to an Intel or Xilinx FPGA. To learn more, see "Generate HDL Code for FPGA Floating-Point Target Libraries" on page 31-20.</li> </ul>
Character	<code>char</code>	
Logical	<code>logical</code>	
Fixed point	<ul style="list-style-type: none"> <li>• Scaled (binary point only) fixed-point numbers</li> <li>• Custom integers (zero binary point)</li> </ul>	<p>Fixed-point numbers with slope (not equal to 1.0) and bias (not equal to 0.0) are not supported.</p> <p>Maximum word size for fixed-point numbers is 128 bits.</p>
Vectors	<ul style="list-style-type: none"> <li>• <code>unordered {N}</code></li> <li>• <code>row {1, N}</code></li> <li>• <code>column {N, 1}</code></li> </ul>	<p>The maximum number of vector elements allowed is <math>2^{32}</math>.</p> <p>Before a variable is subscripted, it must be fully defined.</p>

Types	Supported Data Types	Restrictions
Matrices	{N, M}	Matrices are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.  Do not use matrices in the testbench.
Structures	<code>struct</code>	Arrays of structures are not supported.  For the FPGA Turnkey and IP Core Generation workflows, structures are supported in the body of the design algorithm, but are not supported as inputs to the top-level design function.
Enumerations	<code>enumeration</code>	Enumeration values must be monotonically increasing.  If your target language is Verilog, all enumeration member names must be unique within the design.  Enumerations at the top-level DUT ports are not supported with the following workflows or verification methods: <ul style="list-style-type: none"> <li>• IP Core Generation workflow</li> <li>• FPGA Turnkey workflow</li> <li>• FPGA-in-the-Loop</li> <li>• HDL Cosimulation</li> </ul>

## Unsupported Data Types

The following data types are not supported:

- Cell array
- Inf

## Scope for Variables

Global variables are not supported for HDL code generation.

# Import Verilog Code and Generate Simulink Model

## In this section...

- ["HDL Import" on page 10-127](#)
- ["HDL Import Requirements" on page 10-127](#)
- ["How to Import HDL Code" on page 10-127](#)
- ["Model Location" on page 10-128](#)
- ["Errors and Warnings" on page 10-128](#)
- ["Limitations of Verilog HDL Import" on page 10-128](#)

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. HDL import parses the input HDL file and generates a Simulink model. The model is a block diagram environment that visually represents the HDL code in terms of functionality and behavior. By importing the HDL code into Simulink, you can verify the functionality of the HDL code by compiling and running simulation on the model in a model-based simulation environment. You can also debug internal signals by logging the signals as test points.

## HDL Import

Round-trip code generation with HDL import is not recommended. Do not use HDL import to import the HDL code that was previously generated from a Simulink model by using the HDL Coder software. The Simulink model that you create is typically at a higher abstraction level. The model generated by HDL import might be at a lower abstraction level. The HDL code you generate from this model might not be usable for production code.

To generate production HDL code, develop your algorithm by using Simulink blocks, MATLAB code, or Stateflow charts. Then, use HDL Coder to generate code.

## HDL Import Requirements

To generate a Simulink model, make sure that the HDL file you import:

- Is free of syntax errors.
- Is synthesizable.
- Uses supported Verilog constructs for the import.

## How to Import HDL Code

To import the HDL code, at the MATLAB Command Window, run the `importhdl` function. For example, to import a Verilog file `example.v`, at the command line, enter:

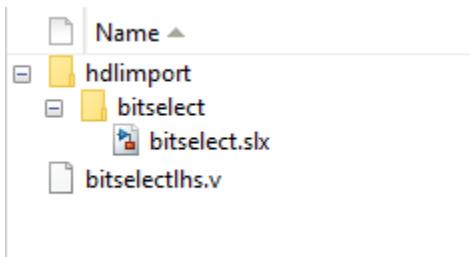
```
importhdl('example.v')
```

The function parses the HDL input file that you specified and generates the corresponding Simulink model, and provides a link to open the model.

The constructs that you use in the HDL code can infer simple Simulink blocks such as Add and Product to RAM blocks such as Dual Rate Dual Port RAM. For examples that illustrate various Simulink models that are inferred, see `importhdl`.

## Model Location

The generated Simulink model is named after the top module in the input HDL file that you specify. The model is saved in the `hdlimport/TopModule` path relative to the current working folder. For example, if you input a file named `bitselectlhs.v` to the `importhdl` function that has `bitselect` as the top module name, the generated Simulink model has the name `bitselect.slx`, and is saved in the `hdlimport/bitselect` path relative to the current folder.



## Errors and Warnings

When you run the `importhdl` function, HDL import verifies the syntax and semantics of the input HDL code. Semantic verification checks for module instantiation constructs, unused ports in the module definition, the sensitivity list of an `always` block, and so on. If HDL import fails, `importhdl` provides an error message and a link to the file name and line number.

For example, consider this Verilog code for a `bitselect` module:

```
module bitselect(a,c);  
  
    input [1:0] a;  
    output [1:0] c;  
  
    c[0] = 0;  
    assign c[1] = a[2];  
  
endmodule
```

When you run the `importhdl` function, HDL import generates an error message:

```
Parser Error: bitselectlhs.v:6:2: error: Syntax Error near '['..
```

The error message indicates that there is a syntax error in line 6. To fix this error, change the syntax to an assignment statement.

```
assign c[0] = 0;
```

## Limitations of Verilog HDL Import

HDL import does not support:

- Importing of VHDL files.
- Importing of Verilog files from a read-only folder.
- Generation of the preprocessing files in a read-only file system that parse the HDL code you input to the `importhdl` function.
- Attribute instances and comments, which are ignored.
- (#)delay values, such as #25, which are ignored.
- Enumeration data types.
- More than one clock signal.
- Modules that are multirate.
- Recursive module instantiation.
- Multiport Switch inference with more than 1024 inputs. If you specify more than 1024 inputs to a Multiport Switch block that gets inferred from the Verilog code, Verilog import generates an error. The error is generated because the Simulink modeling environment does not support more than 1024 inputs for the block.
- ROM detection from the Verilog code.
- Importing of HDL files that use unsupported Verilog constructs. See “Supported Verilog Constructs for HDL Import” on page 10-130.
- Importing of HDL files that use unsupported dataflow modeling patterns. See “Unsupported Verilog Dataflow Patterns” on page 10-137.

## See Also

### Functions

`checkhdl` | `makehdl`

## Related Examples

- “Generate Simulink® Model From CORDIC Atan2 Verilog® Code” on page 10-148

## More About

- “Supported Verilog Constructs for HDL Import” on page 10-130
- “Verilog Dataflow Modeling with HDL Import” on page 10-135

## Supported Verilog Constructs for HDL Import

### In this section...

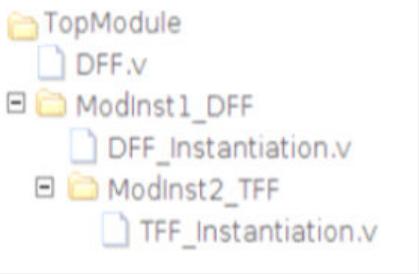
- “Module Definition and Instantiations” on page 10-130
- “Data Types and Vectors” on page 10-131
- “Identifiers and Comments” on page 10-131
- “Assignments” on page 10-132
- “Operators” on page 10-132
- “Conditional and Looping Statements” on page 10-133
- “Procedural Blocks and Events” on page 10-133
- “Other Constructs” on page 10-133

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. To import the HDL code, use the `importhdl` function. Make sure that the constructs used in the HDL code are supported by HDL import.

These tables list the supported Verilog constructs that you can use when you import your HDL code. If you use an unsupported construct, HDL import generates an error when parsing the input HDL file. Verilog HDL import can sometimes ignore the presence of certain constructs in the HDL code. To learn more, see the **Comments** section of the table.

### Module Definition and Instantiations

Verilog Constructs	Supported?	Comments
Library declaration	No	-
Configuration declaration	No	-
Module declaration	Yes	Multiple sample rates and multiple clock inputs are not supported.
Module parameter port list	Yes	-
Port declarations	Yes	INOUT ports are not supported.
Module without ports	No	-
Local parameter declaration	Yes	-
Parameter declaration	Yes	You can use parameters and constants that have a maximum size of 64 bits. By default, the parameter size is 32 bits.

Verilog Constructs	Supported?	Comments
Module instantiation	Yes	<ul style="list-style-type: none"> <li>Unconnected ports in the instantiated modules are removed when importing the Verilog code.</li> <li>Recursive module instantiation is not supported.</li> </ul> <p>Instead, if your top module instantiates modules that are defined in recursive subfolders, <code>importhdl</code> parses all Verilog files. For example, in this figure, <code>importhdl</code> can parse both <code>DFF_Instantiation.v</code> and <code>TFF_Instantiation.v</code> that are instantiated in <code>DFF.v</code>.</p> 

## Data Types and Vectors

Verilog Constructs	Supported?	Comments
Net declaration (Wire, Supply0, Supply1)	Yes	-
Real declaration	No	-
String declaration	No	-
Vector declaration	Yes	-
Array support and array indexing	Yes	-
Reg declaration	Yes	-
Integer declaration	Yes	-

## Identifiers and Comments

Verilog Constructs	Supported?	Comments
Lexical tokens (Whitespace, operator, comment)	Yes	-
Identifiers (Simple, Escaped)	Yes	-
System Functions (\$signed, \$unsigned)	Yes	-

<b>Verilog Constructs</b>	<b>Supported?</b>	<b>Comments</b>
Attribute instances	No	HDL import ignores these constructs.
Comments	No	HDL import ignores these constructs.
Numbers (Decimal, Binary, Hexadecimal, and Octal)	Yes	-
Compiler directives (`define, `undef, `ifndef, `else if)	Yes	-

## Assignments

<b>Verilog Constructs</b>	<b>Supported?</b>	<b>Comments</b>
Continuous assignment	Yes	-
Blocking assignment	Yes	--
Nonblocking assignment	Yes	-
Procedural assignment (Always block)	Yes	-

## Operators

<b>Verilog Constructs</b>	<b>Supported?</b>	<b>Comments</b>
Arithmetic operators (+, -, *, **, /, <<<, >>>)	Yes	-
Logical operators (<<, >>, !, &&,    , ==, !=)	Yes	-
Relational operators (>, <, >=, <=, ==, !=)	Yes	-
Bitwise operators (~, &,  , ^, ~^, ^~)	Yes	-
Unary operators (+, -)	Yes	Supported for restricted data types
Power operators	Yes	Supported for restricted data types
Conditional operators (?:)	Yes	-
Concatenation	Yes	-
Bit Select	Yes	-
Reduction operators (&, ~&,  , ~ , ^, ~^, or ^~)	Yes	-

For an example that illustrates how to use different operators, see “Generate Simulink Model from Verilog Code for Various Operators”.

## Conditional and Looping Statements

Verilog Constructs	Supported?	Comments
If-else statement	Yes	-
Conditional operators (?:)	Yes	-
For loop	Yes	-
Loop Generate construct	Yes	Supports loop generate constructs such as for-generate, case-generate, and if-generate constructs.
Conditional Generate construct	No	-
Generate region	No	-
Genvar declaration	No	-
Case statement	Yes	casex and casez statements are also supported.

## Procedural Blocks and Events

Verilog Constructs	Supported?	Comments
Task declaration	No	-
Initial construct (ROM modeling)	No	-
Sequential blocks	Yes	-
Block declarations	Yes	-
Event control statements	Yes	-
Function calls	Yes	HDL import does not support recursive function calls.
Task enable	No	-
Always construct	Yes	-
Function declaration	Yes	-

## Other Constructs

Verilog Constructs	Supported?	Comments
Gate instantiation	No	-
Specparams	No	-
Specify block	No	-
Semantic verification (unused ports, correct module instantiation)	Yes	-

Verilog Constructs	Supported?	Comments
Clock bundle identification	Yes	Multiple sample rates and multiple clock signals are not supported.
Register inference	Yes	-
Compare to Constant block inference	Yes	-
Gain block inference	Yes	-
RAM inference	Yes	-
ROM inference	No	-
Counter inference	No	-
Drive strength	No	-

## See Also

### Functions

`checkhdl | makehdl`

## Related Examples

- “Generate Simulink® Model From CORDIC Atan2 Verilog® Code” on page 10-148

## More About

- “Import Verilog Code and Generate Simulink Model” on page 10-127
- “Verilog Dataflow Modeling with HDL Import” on page 10-135

# Verilog Dataflow Modeling with HDL Import

## In this section...

["Supported Verilog Dataflow Patterns" on page 10-135](#)

["Unsupported Verilog Dataflow Patterns" on page 10-137](#)

Use HDL import to import synthesizable HDL code into the Simulink modeling environment. To import the HDL code, use the `importhdl` function. Make sure that the constructs used in the HDL code are supported by HDL import.

These tables list the supported Verilog HDL dataflow patterns that you can use when importing the HDL code. If your code uses an unsupported dataflow model such as code that infers a latch, `importhdl` generates an error message with a link to the file name and line number. You can then update the code as illustrated in the preceding examples.

## Supported Verilog Dataflow Patterns

Verilog Dataflow Model	Example Verilog Code
Blocking assignments in sequential always blocks and nonblocking assignments in combinational always blocks.	<p>For example, this Verilog code uses a sequential assignment for the variable <code>temp</code> in a combinational <code>always</code> block.</p> <pre>module dataconv(clk,a,b,c);      input clk; integer i;     input wire [7:0] a, b;     output wire [7:0] c;     reg [1:0] temp [0:7];      always @(*) begin         for (i=0;i&lt;=7;i=i+1) begin             temp[i] &lt;= a[i] + b[i];         end     end      assign c = temp;  endmodule</pre>

Verilog Dataflow Model	Example Verilog Code
<p>Multiple assignments to the same signal. You can use partial and complete assignments to that signal.</p>	<p>This example shows the Verilog code that performs both partial assignment and complete assignment to the variable <code>out1_reg</code>.</p> <pre data-bbox="757 481 1269 1051"> module testPartialAndCompleteAssign   (input [2:0] in1, in2,    input cond, clk,    output [2:0] out1);    reg [2:0] out1_reg;    always@(posedge clk) begin     if(cond) begin       out1_reg[0] = 1'b0;       out1_reg[1] = 1'b0;       out1_reg[2] = 1'b0;     end     else begin       out1_reg = in1 &amp; in2;     end   end   assign out1 = out1_reg;  endmodule </pre>
<p>Multiple assignments to the same variable in the true or false path of a Switch block that gets inferred.</p>	<p>For example, this Verilog code performs multiple assignments to the variable <code>out1</code> inside both the if and else conditions of the always block.</p> <pre data-bbox="757 1294 1165 1759"> module testSwitchMuxing   (input [2:0] in1,    input cond,    output reg [2:0] out1);    always@(*) begin     if (cond) begin       out1 = in1;     end     else begin       out1 = 3'd2;       out1 = 3'd1;     end   end endmodule </pre>

Verilog Dataflow Model	Example Verilog Code
<p>Bit select, part select, and array indexing operations with signals. A Multiport Switch is inferred when array indexing is performed at the RHS. An array indexing operation on the LHS is inferred as an Assign block.</p>	<p>For example, this Verilog code uses multiple assignments to the variable <code>out1</code> in the false branch, which is not supported.</p> <pre data-bbox="757 481 1410 1311"> module ArrayIndexing   (In1, In2, In3, In4,    Sel1, Sel2,    Out1, Out2, Out3);    parameter w = 7;   input [w:0] In1, In2, In3, In4;   input [1:0] Sel1, Sel2;   output [w:0] Out1;   output [1:0] Out2;   output [Out3];    wire [w:0] v[3:0];    assign v[0] = In1;   assign v[1] = In2;   assign v[2] = In3;   assign v[3] = In4;    //Array indexing with signal Sel1   assign Out1 = v[Sel1];    //Part select on array index with signal Sel2   assign Out2 = v[Sel2][7:2];    //Bit select on array index with signal Sel   assign Out3 = v[Sel2][4];  endmodule </pre>

## Unsupported Verilog Dataflow Patterns

These dataflow constructs are unsupported when you import the Verilog code. The *Description and Example Scenarios* column describes each scenario with an example and illustrates how you can avoid this scenario.

Verilog Dataflow Model	Description and Example Scenarios
Latch detection from the Verilog code.	<p>Presence of latches in the HDL code can result in algebraic loops in the generated Simulink model and result in model compilation failures. To avoid latch inference, specify all branches in if-else conditions and in case statements.</p> <p>For example, when you import this Verilog code, the if-condition is inferred as a Switch block. The block has a true path that is specified in the code. The output of the Switch block is fed back directly as input to the false path which results in an algebraic loop.</p> <pre data-bbox="757 726 1442 1360"> module testAlgebraicLoop(input cond,     input [2:0] in1,     output reg [2:0] out1);      // causes latch inference - unsupported     always@(*) begin         if (cond) begin             out1 = in1;         end     end      // Specify else branch to avoid latch inference     // always@(*) begin     //     if (cond) begin     //         out1 = in1;     //     end     //     else begin     //         out1 = in1 + 1;     //     end     // end  endmodule </pre>

Verilog Dataflow Model	Description and Example Scenarios
<p>Performing operations on clock, reset, or clock enable signals in the Verilog code.</p>	<p>When you define signals using names such as <code>clk</code>, <code>rst</code>, and <code>enb</code>, HDL import infers these signals to be the clock, global reset, and clock enable signals.</p> <p>For example, this Verilog code uses an explicit assignment with the clock signal <code>clk</code>. This construct is not supported.</p> <pre data-bbox="755 566 1209 994"> module testOperationOnClkBundle     (input [2:0] in1,      input clk,      output reg [2:0] out1,      output out2);      reg [2:0] out1_reg;      always@(posedge clk) begin         out1_reg &lt;= in1;     end      assign out2 = clk &amp;&amp; 1'b1;  endmodule </pre> <p>Make sure that your Verilog code does not perform operations on any of the signals in the clock bundle. In addition, for signals that you use for performing computations in your code, make sure that the signal names do not match any of the names that are inferred as clock, reset, or enable signals. See the <code>clockBundle</code> name-value pair of the <code>importhdl</code> function for possible signal names that are inferred as signals in the clock bundle.</p>

Verilog Dataflow Model	Description and Example Scenarios
<p>Sensitivity of clock or reset signal to different clock edges or different reset edges inside the same module.</p>	<p>You cannot have one <code>always</code> block with the clock signal sensitive to the positive edge and the other <code>always</code> block with the clock signal sensitive to the negative edge inside the same module.</p> <p>For example, this Verilog code uses the positive and negative edges of the same clock, which is not supported.</p> <pre data-bbox="755 593 1263 1205"> module testMultipleClockEdges     (input [2:0] in1, in2,      input clk,      output [2:0] out1, out2);      reg [2:0] out1_reg, out2_reg;      /* clk sensitivity to posedge */     always@(posedge clk) begin         out1_reg &lt;= in1 &amp;&amp; in2;     end      /* clk sensitivity to negedge */     always@(negedge clk) begin         out2_reg &lt;= in1    in2;     end      assign out1 = out1_reg;     assign out2 = out2_reg;  endmodule </pre> <p>Make sure that your Verilog code does not use both edges of the clock or reset signal in the same module. Use either <code>posedge</code> or <code>negedge</code> of the clock.</p>

Verilog Dataflow Model	Description and Example Scenarios
Realization of synchronous and asynchronous circuits inside the same module.	<p>You cannot use Verilog code realizes both circuits in the same module as shown in the code below. Use different modules for realization of asynchronous and synchronous circuits.</p> <pre data-bbox="755 502 1323 1148"> module testSynchronousAsynchronous     (input [2:0] in1, in2,      input clk, reset,      output [2:0] out1, out2);      reg [2:0] out1_reg, out2_reg;      /* synchronous always block */     always@(posedge clk) begin         out1_reg &lt;= in1 &amp;&amp; in2;     end      /* asynchronous always block */     always@(posedge clk or posedge reset)     begin         out2_reg &lt;= in1    in2;     end      assign out1 = out1_reg;     assign out2 = out2_reg;  endmodule </pre>

Verilog Dataflow Model	Description and Example Scenarios
Initialization of multiple RAMs that are inferred in the same module.	<p>For example, this Verilog code infers <code>sample_store0</code> and <code>sample_store1</code> as RAMs. This construct is not supported. Separate each RAM inference into a single module.</p> <pre data-bbox="757 508 1465 1191"> module testMultipleRAMs     (input [2:0] in1, in2,      input clk, reset,      output reg [2:0] read_data0, read_data1);      reg [2:0] out1_reg, out2_reg;      /* Inference of RAM sample_store0 */     always@(posedge clk) begin         if (write_enable) begin             sample_store0[write_address] &lt;= write_data;         end         read_data0 = sample_store0[read_address0];     end      /* Inference of RAM sample_store1 */     always@(posedge clk) begin         if (write_enable) begin             sample_store1[write_address] &lt;= write_data;         end         read_data1 = sample_store1[read_address1];     end endmodule </pre>

Verilog Dataflow Model	Description and Example Scenarios
<p>Usage of certain constructs when reading initial values from an <code>initial</code> block.</p>	<p>An <code>initial</code> block does not support constructs other than assignment statements or for loops with assignments. The RHS of the assignment statement must use only assignment operators (<code>=</code>, <code>:=</code>, <code>+=</code>, etc.).</p> <p>For example, this Verilog code uses an if-else condition to assign a value to the variable <code>dout_a</code> and assigns a variable <code>write_data</code> to <code>dout_c</code>, which are not supported.</p> <pre data-bbox="757 572 1547 1600"> module testInitial     (input cond,      input [3:0] write_data      output reg [3:0] dout_a, dout_b, doutc);      parameter AddrWidth = 3;      integer i;     reg [3:0] ram [7:0];      initial begin         for (i=0; i&lt;=2**AddrWidth - 1; i=i+1) begin             ram[i] = 0;         end         dout_b = 0;          // if-else condition - not supported         if (cond) begin             dout_a = 0;         end         else begin             dout_a = 4;         end     end      // Variable assignment to RHS - not supported     dout_a = write_data;      // Use assignment statements instead for     // dout_a and dout_c     // dout_a = 4;     // dout_c = 32;     // end     // end  endmodule </pre>

Verilog Dataflow Model	Description and Example Scenarios
All dimensions of signals not referenced at time of usage.	<p>Use all dimensions of a signal in the input Verilog code. If you specify an additional dimension, it creates an indexed bit select.</p> <p>For example, this Verilog code creates an 8-bit variable <code>temp</code>. The assignment inside the <code>always</code> block generates an error because it does not use both dimensions of the variable.</p> <pre data-bbox="755 629 1441 1495"> module dataconv(clk,a,b,c);  input clk; input wire [1:0] a, b; output wire [1:0] c;  reg [7:0] temp_reg [1:0][1:0];  // temp_reg is not indexed // with both dimensions - not supported always @(posedge clk) begin     temp_reg [0] &lt;= a[0] + b[0];     temp_reg [1] &lt;= a[1] + b[1]; end  // Use both dimensions when indexing a variable // always @(posedge clk) begin //     temp_reg [0][0] &lt;= a[0] + b[0]; //     temp_reg [0][1] &lt;= a[0] + b[1]; //     temp_reg [1][0] &lt;= a[1] + b[0]; //     temp_reg [1][1] &lt;= a[1] + b[1];  // You can also perform indexed bit select //     temp_reg [1][0][1] = 1'b0; // end  assign c = temp_reg;  endmodule </pre>

## See Also

### Functions

`checkhdl` | `makehdl`

## Related Examples

- “Generate Simulink® Model From CORDIC Atan2 Verilog® Code” on page 10-148

## More About

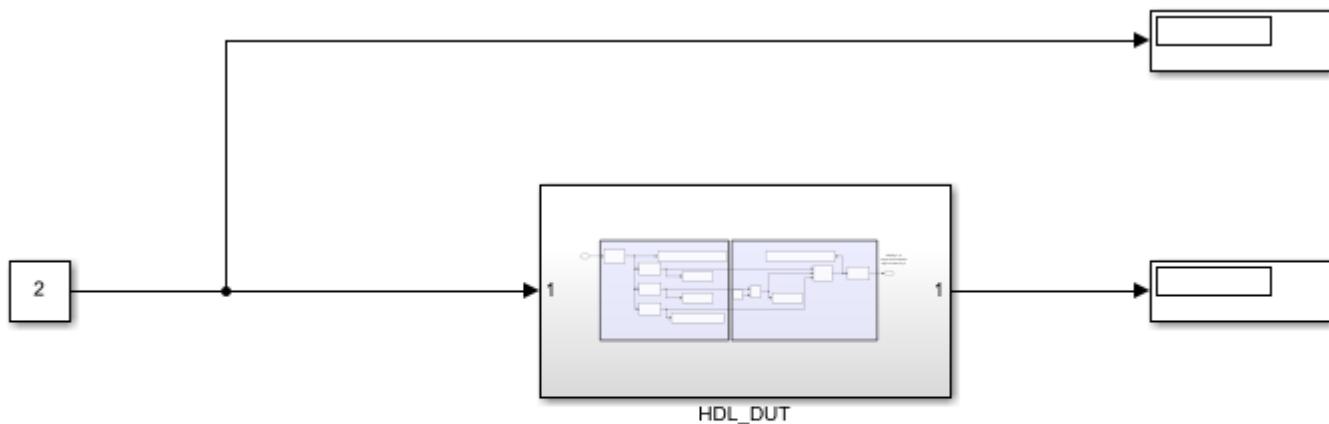
- “Import Verilog Code and Generate Simulink Model” on page 10-127
- “Supported Verilog Constructs for HDL Import” on page 10-130

## Simulate and Generate HDL Code for the Float Typecast Block

This example shows how you can use the Float Typecast block to extract the sign, exponent, and mantissa bits from a floating-point input, and then convert the bits back to a floating-point output after performing any computations.

Open the `hdlcoder_float_typecast_example` model.

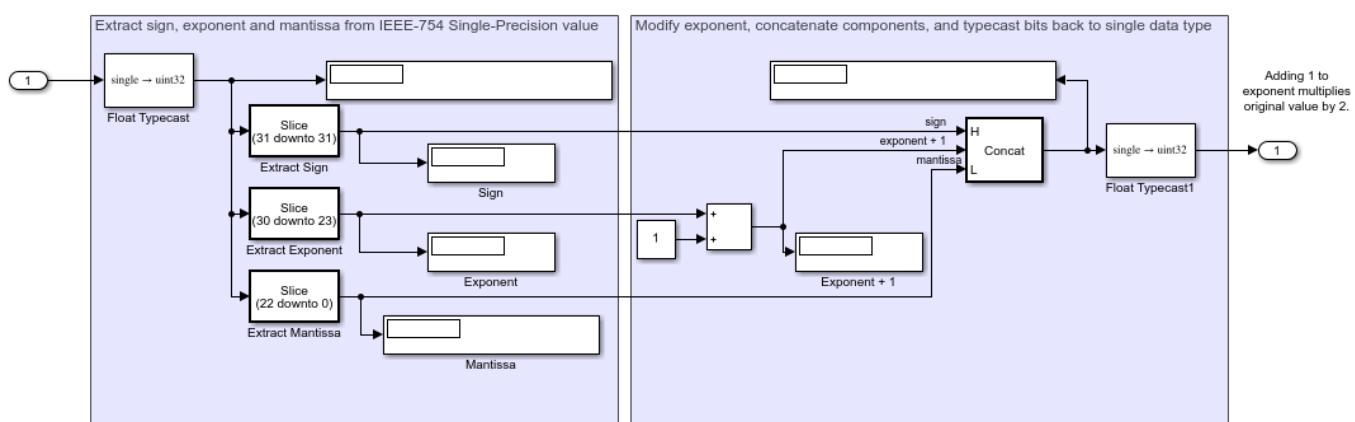
```
open_system('hdlcoder_float_typecast_example')
```



Copyright 2020 The MathWorks, Inc.

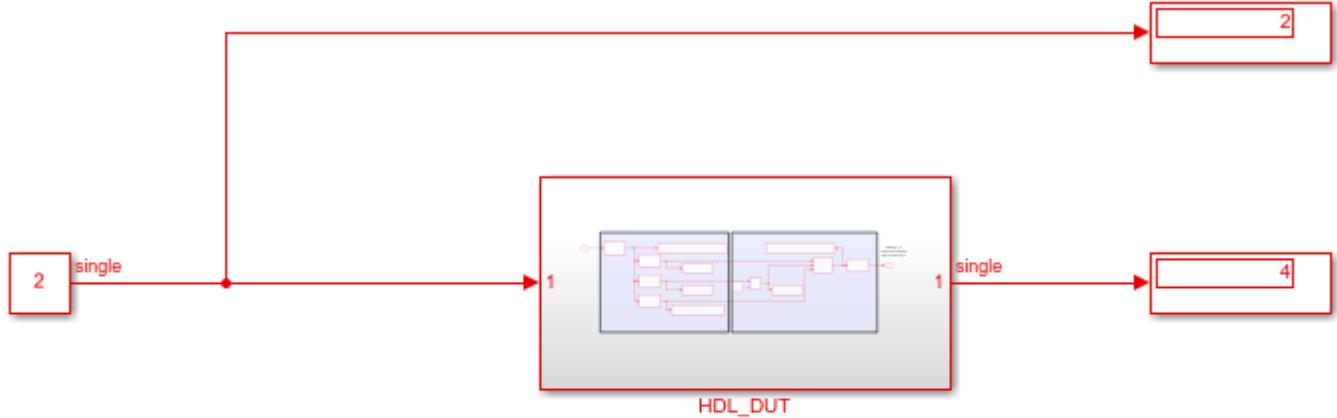
The model multiplies the floating-point input by two to produce the floating-point output. To multiply the input, the algorithm increments the exponent by one. Open the HDL DUT subsystem.

```
open_system('hdlcoder_float_typecast_example/HDL_DUT')
```



The model is already configured for HDL compatibility by using the `hdlsetup` function. Simulate the model.

```
sim('hdlcoder_float_typecast_example')
open_system('hdlcoder_float_typecast_example')
```



Copyright 2020 The MathWorks, Inc.

Before you generate HDL code, enable the Native Floating Point mode.

```
nfpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT');
hdlset_param('hdlcoder_float_typecast_example', ...
    'FloatingPointTargetConfiguration', nfpconfig);
```

Generate HDL code for the `HDL_DUT` subsystem.

```
makehdl('hdlcoder_float_typecast_example')
```

## Generate Simulink® Model From CORDIC Atan2 Verilog® Code

This example shows how you can import a file containing Verilog code and generate the corresponding Simulink model by using the `importhdl` function. `importhdl` imports and parses the specified Verilog files to generate the corresponding Simulink model. The Verilog code in this example contains a CORDIC atan2 algorithm.

### CORDIC 2-Argument Arctangent (atan2) Verilog Design

This Verilog input file implements a CORDIC atan2 algorithm.

```
cordic_atan2_verilog_file = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'cordic');
type(cordic_atan2_verilog_file);

`timescale 1ns / 1ps

module cordic_atan2(
    input clk,                      // clock
    input reset,                     // reset to the system
    input enable,                    // enable
    input signed [15:0] x_in,        // Input x value
    input signed [15:0] y_in,        // Input Y value
    output reg signed [15:0] theta // Input theta value
);
    // Pipeline the input values
    reg signed [17:0]x_in_d; reg signed [17:0]y_in_d;
    always @(posedge clk)
    begin
        if(reset) begin
            x_in_d <={18{1'b0}};
            y_in_d <={18{1'b0}};
        end else if(enable) begin
            // extend the input values for intermediate calculations
            x_in_d <= {{2{x_in[15]}},x_in};
            y_in_d <= {{2{y_in[15]}},y_in};
        end
    end
    // pre quad correction logic
    reg x_qaud_adjust; reg y_qaud_adjust;
    reg y_non_zero; reg signed [17:0]x_pre_quad_out;
    reg signed [17:0]y_pre_quad_out;

    always @(posedge clk)begin
        if(reset)begin
            x_pre_quad_out <= {18{1'b0}};
            y_pre_quad_out <= {18{1'b0}};
            x_qaud_adjust <= 1'b0;
            y_qaud_adjust <= 1'b0;
            y_non_zero <=1'b0;
        end
        else if(enable)begin
            if(x_in_d[17] == 1'b1)begin
                x_pre_quad_out <= -x_in_d;
                x_qaud_adjust <= 1'b1;
            end
        end
    end
endmodule
```

```

        else begin
            x_pre_quad_out <= x_in_d;
            x_qaud_adjust  <= 1'b0;
        end
        if(y_in_d[17] == 1'b1)begin
            y_pre_quad_out <= -y_in_d;
            y_qaud_adjust  <= 1'b1;
            y_non_zero     <=1'b1;
        end
        else begin
            y_pre_quad_out <= y_in_d;
            y_qaud_adjust  <= 1'b0;
            y_non_zero     <= |y_in_d; // reduction or for test non zero or not
        end
    end
end

//LOOKUP TABLE FOR THE ANGLES
wire signed [15:0]LUT[0:14];
wire signed [17:0]x_k[0:15];
wire signed [17:0]y_k[0:15];
wire signed [17:0]z_k[0:15];
wire signed [15:0]theta_temp;
parameter PI = 16'sh6488;

assign LUT[0]  = 16'h1922;
assign LUT[1]  = 16'h0Ed6;
assign LUT[2]  = 16'h07D7;
assign LUT[3]  = 16'h03FB;
assign LUT[4]  = 16'h01FF;
assign LUT[5]  = 16'h0100;
assign LUT[6]  = 16'h0080;
assign LUT[7]  = 16'h0040;
assign LUT[8]  = 16'h0020;
assign LUT[9]  = 16'h0010;
assign LUT[10] = 16'h0008;
assign LUT[11] = 16'h0004;
assign LUT[12] = 16'h0002;
assign LUT[13] = 16'h0001;
assign LUT[14] = 16'h0000;

assign x_k[0]  = x_pre_quad_out;
assign y_k[0]  = y_pre_quad_out;

wire signed [17:0]theta_temp0;
assign theta_temp0 = z_k[15];
assign theta_temp = theta_temp0[15:0];
assign z_k[0] = 18'd0;

//cordic iterator
genvar i;
generate
for (i = 0; i<15; i =i+1)
begin
    Kernel cordic_iterator (.clk (clk),
        .reset(reset),
        .enable (enable ),
        .itr_num(i),

```

```
.x_in(x_k[i]),
.y_in(y_k[i]),
.z_in(z_k[i]),
.lut_constant(LUT[i]),
.x_out(x_k[i+1]),
.y_out(y_k[i+1]),
.z_out(z_k[i+1])
);
end
endgenerate

// matching delays for the control signals of the pre quadrant correction logic
reg x_qaud_adjust_match_delay[0:14];
reg y_qaud_adjust_match_delay[0:14];
reg y_non_zero_match_delay[0:14];
integer k;
always @ (posedge clk) begin
    if (reset) begin
        x_qaud_adjust_match_delay[0] <= 1'b0; y_qaud_adjust_match_delay[0] <= 1'b0;
        y_non_zero_match_delay[0] <= 1'b0; x_qaud_adjust_match_delay[1] <= 1'b0;
        y_qaud_adjust_match_delay[1] <= 1'b0; y_non_zero_match_delay[1] <= 1'b0;
        x_qaud_adjust_match_delay[2] <= 1'b0; y_qaud_adjust_match_delay[2] <= 1'b0;
        y_non_zero_match_delay[2] <= 1'b0; x_qaud_adjust_match_delay[3] <= 1'b0;
        y_qaud_adjust_match_delay[3] <= 1'b0; y_non_zero_match_delay[3] <= 1'b0;
        x_qaud_adjust_match_delay[4] <= 1'b0; y_qaud_adjust_match_delay[4] <= 1'b0;
        y_non_zero_match_delay[4] <= 1'b0; x_qaud_adjust_match_delay[5] <= 1'b0;
        y_qaud_adjust_match_delay[5] <= 1'b0; y_non_zero_match_delay[5] <= 1'b0;
        x_qaud_adjust_match_delay[6] <= 1'b0; y_qaud_adjust_match_delay[6] <= 1'b0;
        y_non_zero_match_delay[6] <= 1'b0; x_qaud_adjust_match_delay[7] <= 1'b0;
        y_qaud_adjust_match_delay[7] <= 1'b0; y_non_zero_match_delay[7] <= 1'b0;
        x_qaud_adjust_match_delay[8] <= 1'b0; y_qaud_adjust_match_delay[8] <= 1'b0;
        y_non_zero_match_delay[8] <= 1'b0; x_qaud_adjust_match_delay[9] <= 1'b0;
        y_qaud_adjust_match_delay[9] <= 1'b0; y_non_zero_match_delay[9] <= 1'b0;
        x_qaud_adjust_match_delay[10] <= 1'b0; y_qaud_adjust_match_delay[10] <= 1'b0;
        y_non_zero_match_delay[10] <= 1'b0; x_qaud_adjust_match_delay[11] <= 1'b0;
        y_qaud_adjust_match_delay[11] <= 1'b0; y_non_zero_match_delay[11] <= 1'b0;
        x_qaud_adjust_match_delay[12] <= 1'b0; y_qaud_adjust_match_delay[12] <= 1'b0;
        y_non_zero_match_delay[12] <= 1'b0; x_qaud_adjust_match_delay[13] <= 1'b0;
        y_qaud_adjust_match_delay[13] <= 1'b0; y_non_zero_match_delay[13] <= 1'b0;
        x_qaud_adjust_match_delay[14] <= 1'b0; y_qaud_adjust_match_delay[14] <= 1'b0;
        y_non_zero_match_delay[14] <= 1'b0;
    end
    else if (enable) begin
        x_qaud_adjust_match_delay[0] <= x_qaud_adjust;
        y_qaud_adjust_match_delay[0] <= y_qaud_adjust;
        y_non_zero_match_delay[0] <= y_non_zero;
        for (k = 0; k < 14; k = k + 1) begin
            x_qaud_adjust_match_delay[k + 1] <= x_qaud_adjust_match_delay[k];
            y_qaud_adjust_match_delay[k + 1] <= y_qaud_adjust_match_delay[k];
            y_non_zero_match_delay[k + 1] <= y_non_zero_match_delay[k];
        end
    end
end
// post quadrant correction logic

always @ (posedge clk) begin
    if (reset)
        theta <= 16'd0;
```

```

else if(enable)begin
    if(y_non_zero_match_delay[14])begin
        if(x_qaud_adjust_match_delay[14])begin
            if(y_qaud_adjust_match_delay[14])
                theta <=theta_temp -PI;
            else
                theta <=PI -theta_temp;
        end
        else begin
            if(y_qaud_adjust_match_delay[14])
                theta <= -theta_temp;
            else
                theta <= theta_temp;
        end
    end
    else if(x_qaud_adjust_match_delay[14])begin
        theta <= PI;
    end
    else begin
        theta <= 16'd0;
    end
end
endmodule

module Kernel(
    input clk,
    input reset,
    input enable,
    input signed [17:0]x_in,
    input signed [17:0]y_in,
    input signed [17:0]z_in,
    input [4:0]itr_num,
    input signed [15:0]lut_constant,
    output reg signed [17:0]x_out,
    output reg signed [17:0]y_out,
    output reg signed [17:0]z_out);

wire signed [17:0] lut_constant_signExtension = {{2{lut_constant[15]}},lut_constant};

always @(posedge clk)begin
    if(reset)begin
        x_out <={18{1'b0}};
        y_out <={18{1'b0}};
        z_out <= {18{1'b0}};
    end
    else if(enable)begin
        if(y_in[17])begin
            x_out <= x_in -(y_in>>>itr_num);
            y_out <= y_in +(x_in>>>itr_num);
            z_out <= z_in - lut_constant_signExtension;
        end
        else begin
            x_out <= x_in +(y_in>>>itr_num);
            y_out <= y_in - (x_in>>>itr_num);
            z_out <= z_in + lut_constant_signExtension;
        end
    end
end

```

```
        end  
    end  
  
endmodule
```

This Verilog design contains various commonly used Verilog constructs such as:

- Continuous assignments
- Always blocks
- Conditional Statements
- Module instantiation
- Generate Constructs
- For Loop

### Import HDL File Containing CORDIC atan2 Algorithm

To import the Verilog file, specify the file name as an argument to the `importhdl` function.

```
importhdl(cordic_atan2_verilog_file);  
  
### Parsing <a href="matlab:edit('B:\matlab\toolbox\hdlcoder\hdlcoderdemos\cordic_atan2.v')">cordic_atan2.v  
### Top Module name: 'cordic_atan2'.  
### Identified ClkName::clk.  
### Identified RstName::reset.  
### Identified ClkEnbName::enable.  
Warning: Unused signals detected in the Demux block created for vector index signal 'x_k'. A Demux block was created for vector index signal 'x_k'.  
Warning: Unused signals detected in the Demux block created for vector index signal 'y_k'. A Demux block was created for vector index signal 'y_k'.  
### Hdl Import parsing done.  
### Removing unconnected components.  
### Unconnected components detected when importing the HDL code. These components are removed from the target model.  
### Creating Target model cordic_atan2  
### Start Layout...  
### Working on hierarchy at ---> 'cordic_atan2'.  
### Laying out components.  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2'.  
### Laying out components.  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator1'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator10'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator11'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator12'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator13'.  
### Laying out components.  
### Drawing block edges...  
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator14'.  
### Laying out components.
```

```

### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator2'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator3'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator4'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator5'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator6'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator7'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator8'.
### Laying out components.
### Drawing block edges...
### Working on hierarchy at ---> 'cordic_atan2/cordic_atan2/cordic_iterator9'.
### Laying out components.
### Drawing block edges...
### Drawing block edges...
### Drawing block edges...
### Setting model parameters.
### Generated model file C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp0e00ac9a\hdlimport\cordic_atan2.slx
### Importhdl completed.

```

**importhdl** parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink model **cordic\_atan2.slx**. The generated model uses the same name as the top module in the input Verilog file.

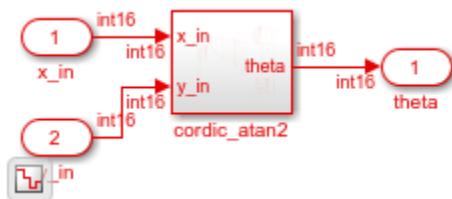
### Examine Generated Simulink Model

To open the generated Simulink model, click the link in the Command Window. The model is saved in the **hdlimport/cordic\_atan2** path relative to the current folder. You can simulate the model and observe the simulation results.

```

open_system('hdlimport/cordic_atan2/cordic_atan2')
Simulink.BlockDiagram.arrangeSystem('cordic_atan2')
set_param('cordic_atan2', 'UnconnectedOutputMsg', 'None');
sim('hdlimport/cordic_atan2/cordic_atan2.slx');

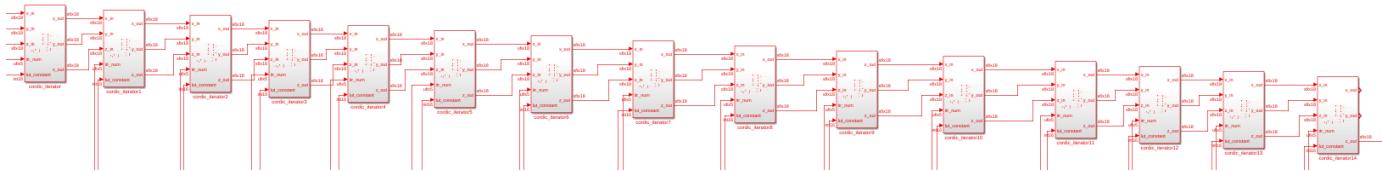
```



### Generated Module instances

The Verilog code instantiates 15 kernel modules by using the generate construct. In the generated Simulink model, 15 kernel modules are seen.

```
generate
  for (i = 0; i<15; i =i+1)
  begin
    Kernel cordic_iterator (.clk(clk),
                           .reset(reset),
                           .enable(enable),
                           .itr_num(i),
                           .x_in(x_k[i]),
                           .y_in(y_k[i]),
                           .z_in(z_k[i]),
                           .lut_constant(LUT[i]),
                           .x_out(x_k[i+1]),
                           .y_out(y_k[i+1]),
                           .z_out(z_k[i+1]));
  end
endgenerate
```



### Simulink model for the Kernel Module

```
module Kernel(
  input clk,
  input reset,
  input enable,
  input signed [17:0]x_in,
  input signed [17:0]y_in,
  input signed [17:0]z_in,
  input [4:0]itr_num,
  input signed [15:0]lut_constant,
  output reg signed [17:0]x_out,
  output reg signed [17:0]y_out,
  output reg signed [17:0]z_out);

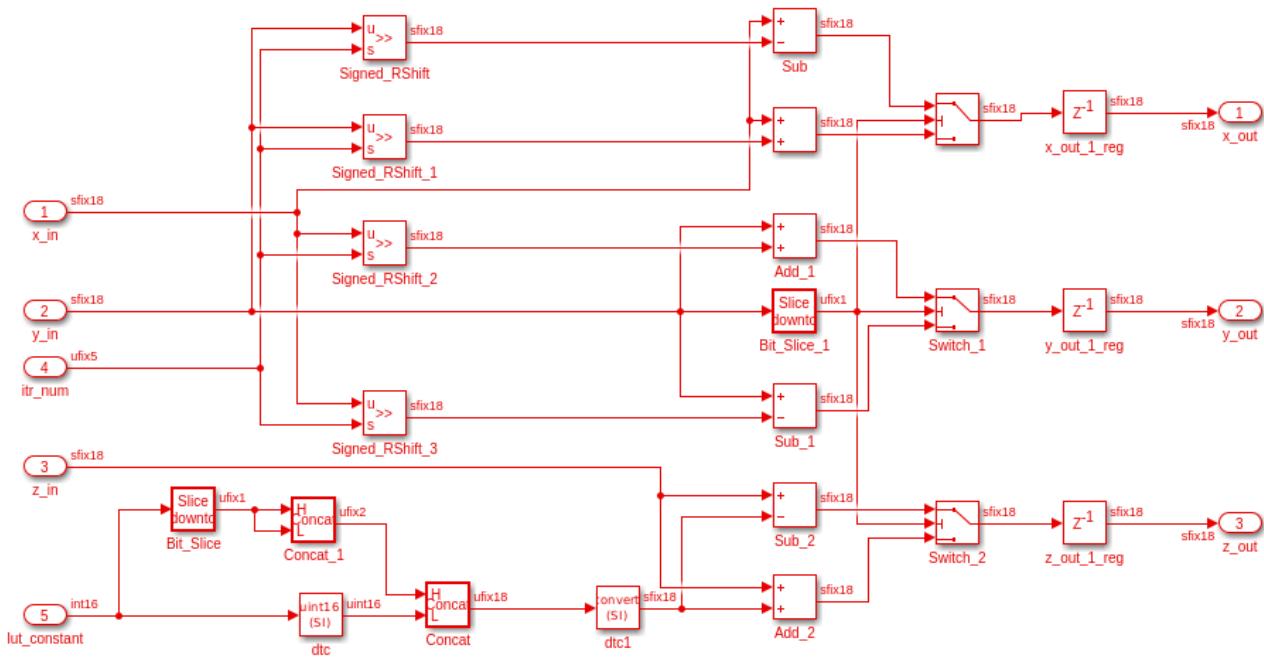
  wire signed [17:0] lut_constant_signExtension = {{2{lut_constant[15]}},lut_constant};

  always @(posedge clk)begin
    if(reset)begin
      x_out <= {18{1'b0}};
      y_out <= {18{1'b0}};
      z_out <= {18{1'b0}};
    end
    else if(enable)begin
      if(y_in[17])begin
        x_out <= x_in - (y_in>>>itr_num);
        y_out <= y_in + (x_in>>>itr_num);
        z_out <= z_in - lut_constant_signExtension;
      end
    end
  end
endmodule
```

```

    else begin
        x_out <= x_in +(y_in>>>itr_num);
        y_out <= y_in - (x_in>>>itr_num);
        z_out <= z_in + lut_constant_signExtension;
    end
end
endmodule

```



## References

- [importhdl](#)
- “Import Verilog Code and Generate Simulink Model” on page 10-127



# Simulink to HDL Examples for Communication and Signal Processing Applications

---

- “Programmable FIR Filter for FPGA” on page 11-2
- “Multichannel FIR Filter for FPGA” on page 11-8
- “Implement FFT for FPGA Using FFT HDL Optimized Block” on page 11-11
- “High Throughput Channelizer for FPGA” on page 11-15
- “HDL Implementation of a Digital Down-Converter for LTE” on page 11-23
- “HDL Optimized QPSK Transmitter” on page 11-42
- “HDL Optimized QPSK Receiver with Captured Data” on page 11-49
- “HDL Optimized QAM Transmitter and Receiver” on page 11-59
- “Airplane Tracking with ADS-B Captured Data” on page 11-78
- “HDL Code Generation for Viterbi Decoder” on page 11-85
- “Design Video Processing Algorithms for HDL in Simulink” on page 11-91
- “Edge Detection and Image Overlay” on page 11-98
- “Lane Detection” on page 11-103

## Programmable FIR Filter for FPGA

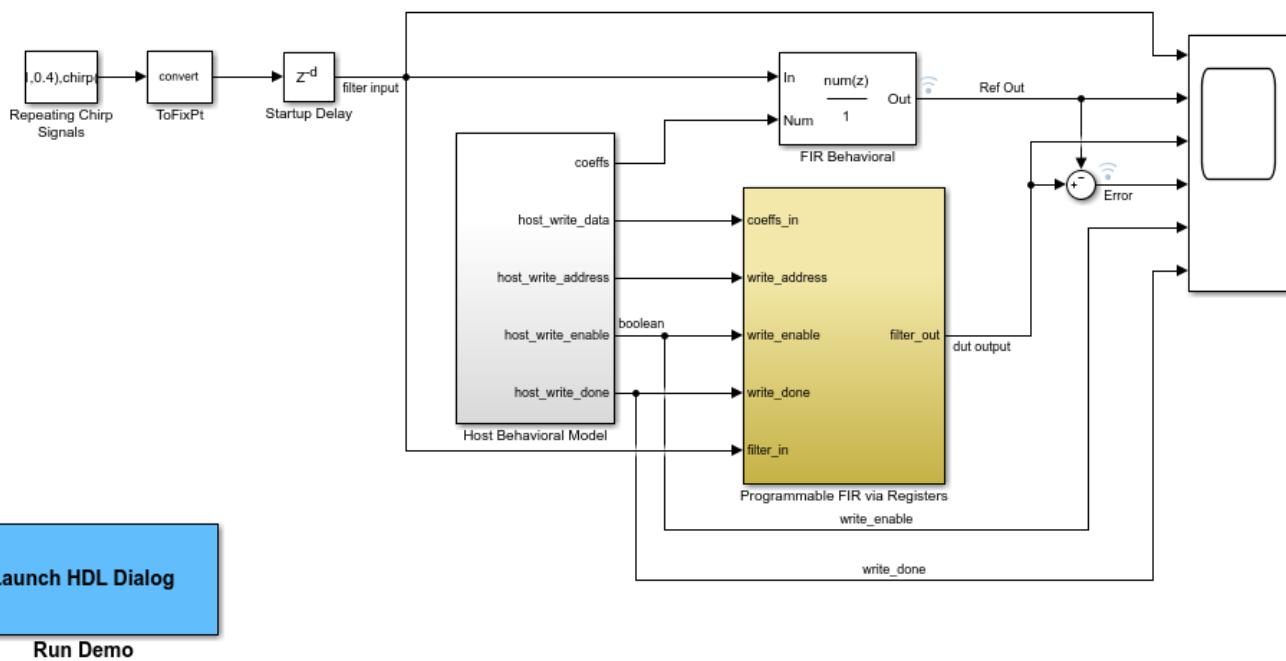
This example shows how to implement a programmable FIR filter for hardware. You can program the filter to a desired response by loading the coefficients into internal registers using the host interface.

In this example, we will implement a bank of filters, each having different responses, on a chip. If all of the filters have a direct-form FIR structure, and the same length, then we can use a host interface to load the coefficients for each response to a register file when needed.

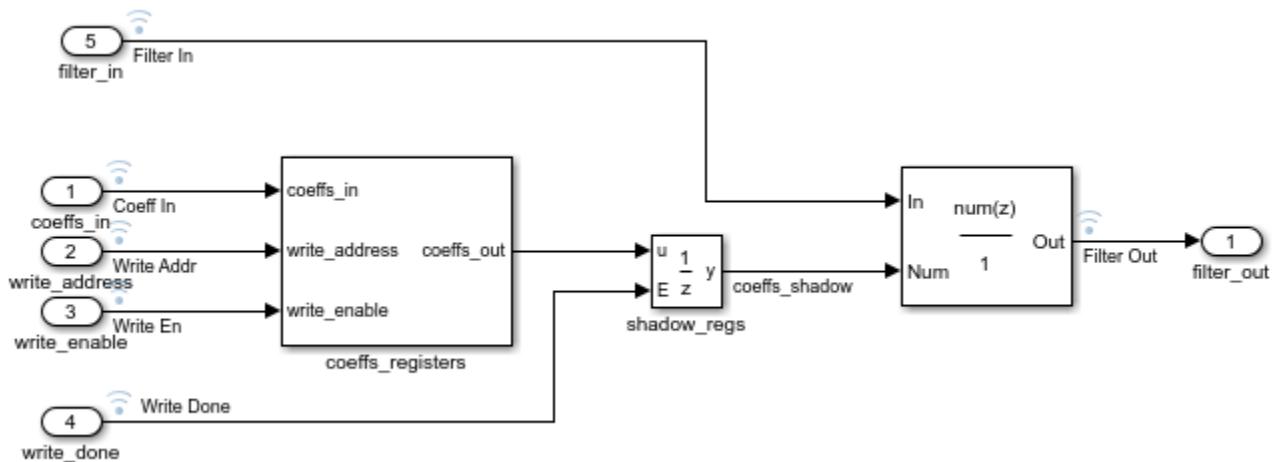
This design adds latency of a few cycles before the input samples can be processed with the loaded coefficients. However, it has the advantage that the same filter hardware can be programmed with new coefficients to obtain a different filter response. This saves chip area, as otherwise each filter would be implemented separately on the chip.

### Model Programmable FIR Filter

Consider two FIR filters, one with a lowpass response and the other with a highpass response. The coefficients are specified by using the Model Properties>Callbacks>InitFcn function.



The *Programmable FIR via Registers* block loads the lowpass coefficients from the *Host Behavioral Model*, and processes the input chirp samples first. Then the block loads the highpass coefficients and processes the same chirp samples again.

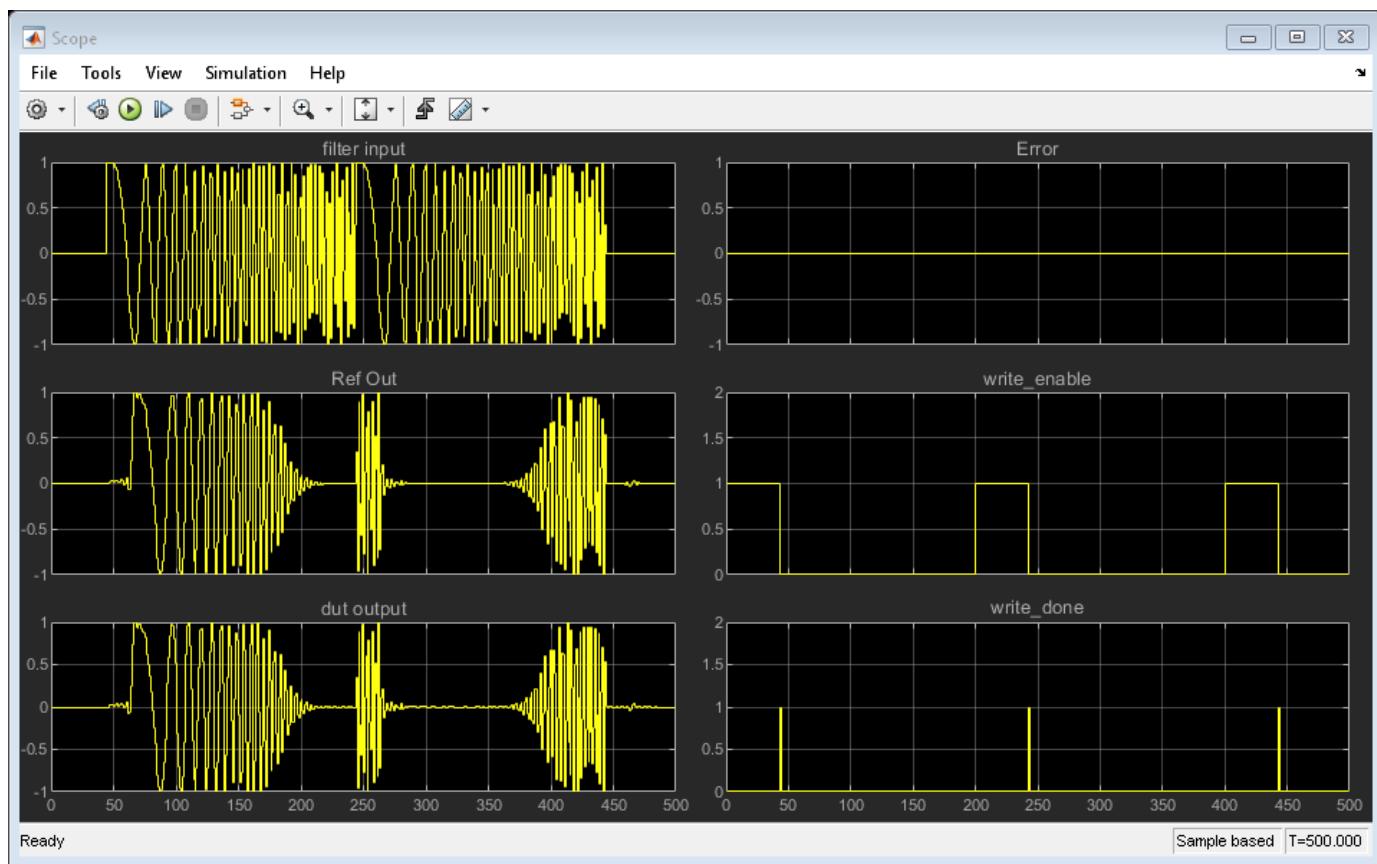


The *coeffs\_registers* block loads the coefficients into internal registers when the `write_enable` signal is high. The shadow registers are updated from the coefficients registers when the `write_done` signal is high. The shadow registers enable simultaneous loading and processing of data by the filter entity. The blocks load the second set of coefficients at the same time as processing the last few input samples.

This model is configured to use a fully parallel architecture for the Discrete FIR Filter block. You can also choose serial architectures from the **HDL Block Properties** menu.

### Simulink® Simulation Results

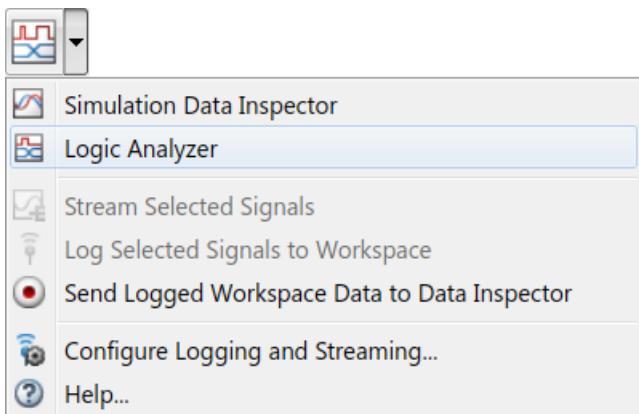
To compare the Design Under Test (DUT) with the reference filter, open the Scope and run the example model.



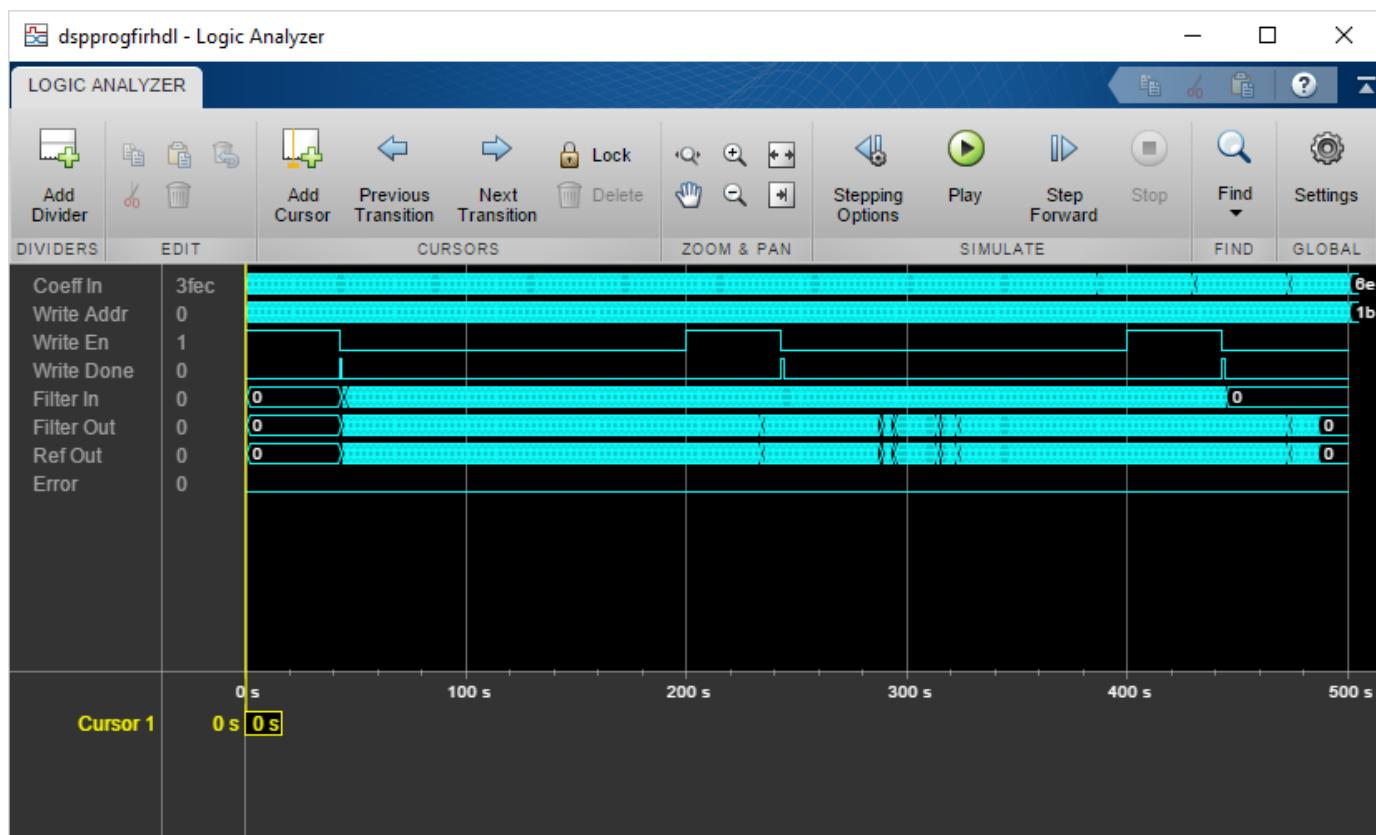
### Using the Logic Analyzer

You can also view the signals in the Logic Analyzer. The Logic Analyzer enables you to view multiple signals in one window. It also makes it easy to spot the transitions in the signals.

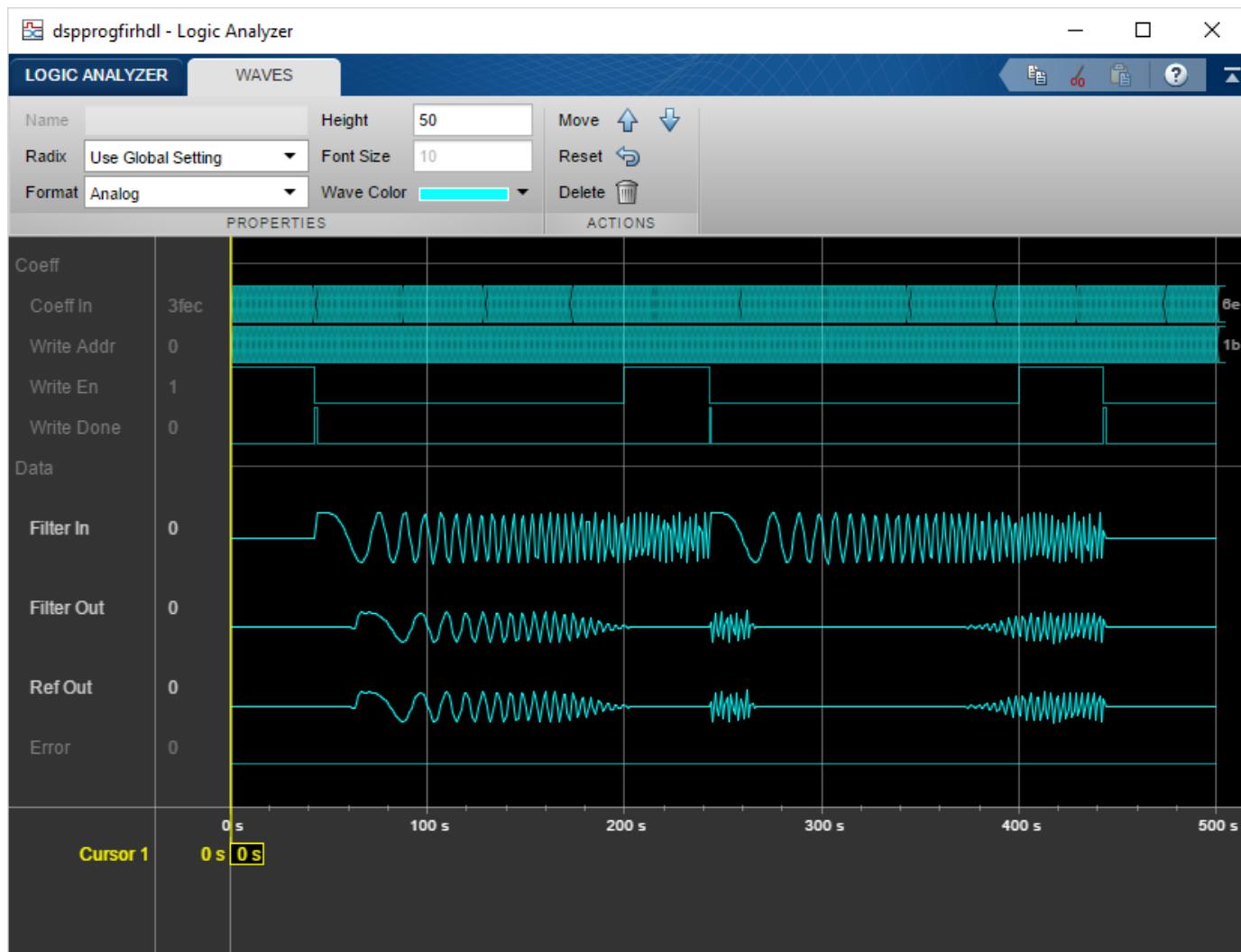
Launch the Logic Analyzer from the model's toolbar.



The signals of interest -- input coefficients, write address, write enable, write done, filter in, filter out, reference out, and error have been added to the Logic Analyzer for observation.



The Logic Analyzer display can also be controlled on a per-wave or per-divider basis. To modify an individual wave or divider, select a wave or divider and then click on the "Wave" tab. A useful mode of visualization in the Logic Analyzer is the Analog format.



For further information on the Logic Analyzer, refer to the Logic Analyzer (DSP System Toolbox) documentation.

### Generate HDL Code and Test Bench

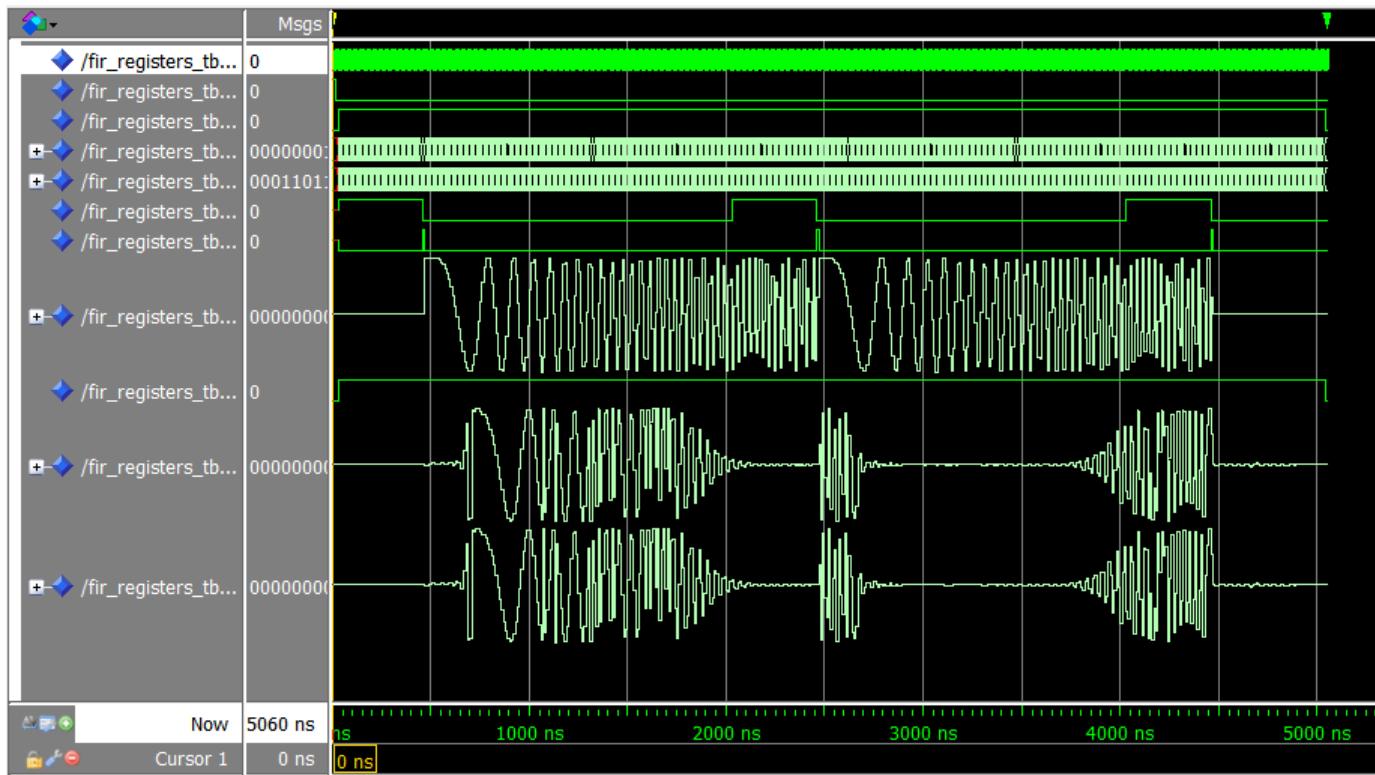
You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code.

```
systemname = [modelname '/Programmable FIR via Registers'];
makehdl(systemname);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior. makehdltb(systemname);

### ModelSim™ Simulation Results

The following figure shows the ModelSim HDL simulator after running the generated .do file scripts for the test bench. Compare the ModelSim result with the Simulink result as plotted before.



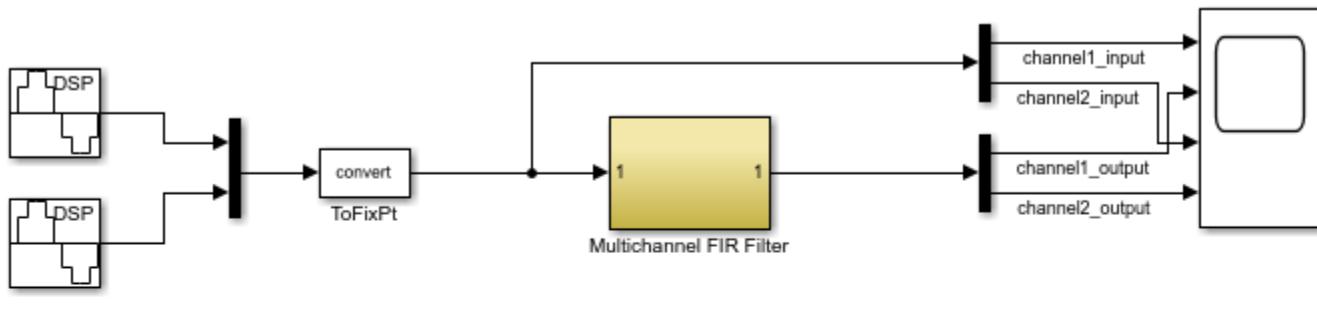
## Multichannel FIR Filter for FPGA

This example shows how to implement a discrete FIR filter with multiple input data streams for hardware.

In many DSP applications, multiple data streams are filtered by the same filter. The straightforward solution is to implement a separate filter for each channel. You can create a more area-efficient structure by sharing one filter implementation across multiple channels. The resulting hardware requires a faster clock rate compared to the clock rate used for a single channel filter.

### Model Multichannel FIR Filter

```
modelname = 'dspmultichannelhdl';
open_system(modelname);
```



**Launch HDL Dialog**

Copyright 2012 The MathWorks, Inc.

The model contains a two-channel FIR filter. The input data vector includes two streams of sinusoidal signal with different frequencies. The input data streams are processed by a lowpass filter whose coefficients are specified by the Model Properties `InitFcn` Callback function.

Select a fully parallel architecture for the Discrete FIR Filter block, and enable resource sharing across multiple channels.

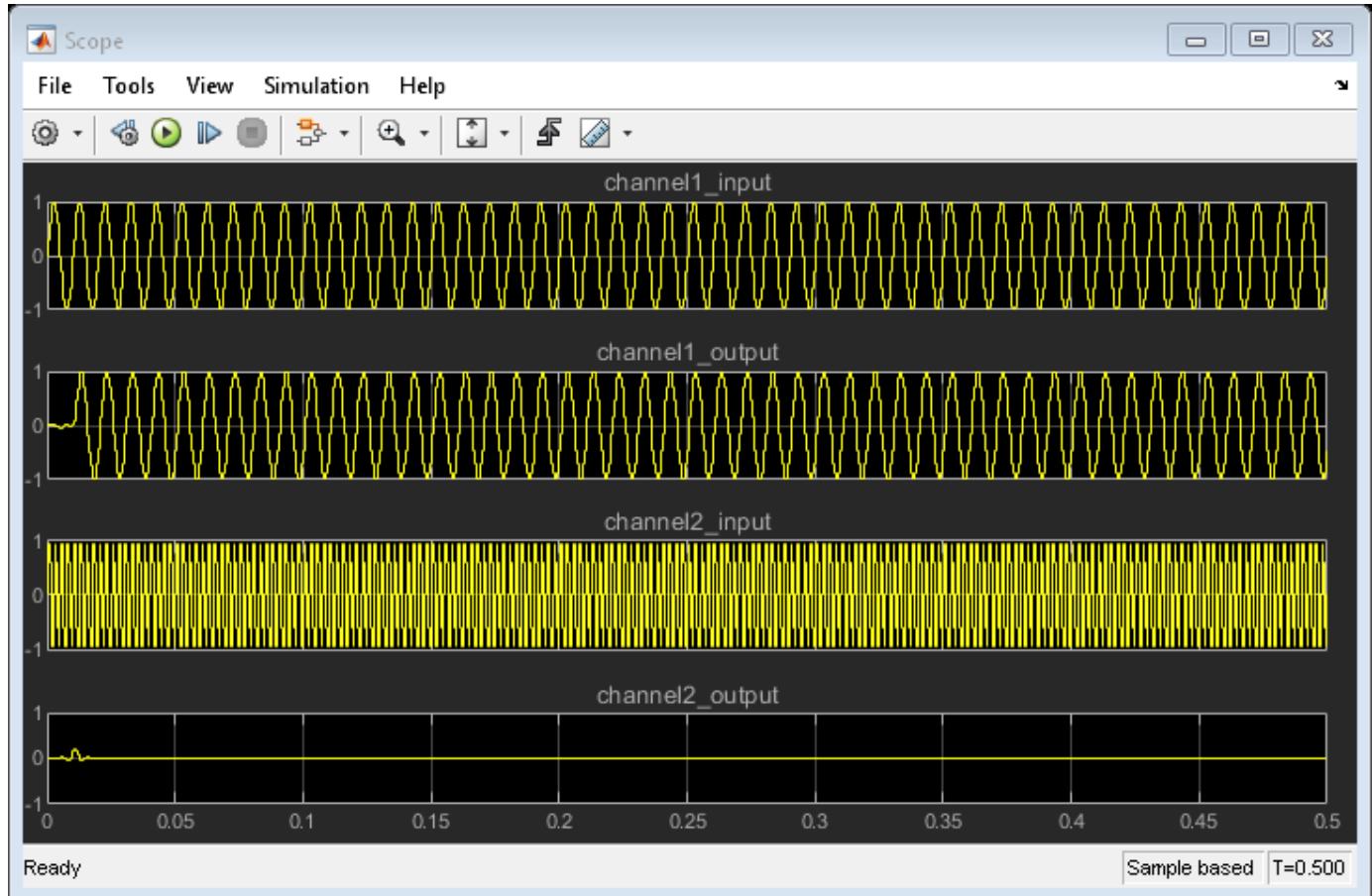
```
systemname = [modelname '/Multichannel FIR Filter'];
blockname = [systemname '/Discrete FIR Filter'];
set_param(blockname,'FilterStructure','Direct form symmetric');
hdlset_param(blockname,'Architecture','Fully Parallel');
hdlset_param(blockname,'ChannelSharing','On');
```

You can alternatively specify these settings on the **HDL Block Properties** menu, which you access by right-clicking a block and selecting **HDL Code > HDL Block Properties**.

## Simulation Results

Run the example model and open the scope to compare the two data streams.

```
sim(modelname);
open_system([modelname '/Scope']);
```



## Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code for the Multichannel FIR Filter subsystem. Enable the resource use report.

```
makehdl(systemname, 'resource', 'on');
```

Use this command to generate a test bench that compares the HDL simulation results with the Simulink model results.

```
makehdltb(systemname);
```

## Compare Resource Utilization

To compare resource use with and without sharing, you can disable sharing resources across channels and generate HDL code again, then compare the resource use reports.

```
hdlset_param(blockname,'ChannelSharing','Off');
makehdl(systemname,'resource','on');
```

**Summary**

Multipliers	44
Adders/Subtractors	86
Registers	86
RAMs	0
Multiplexers	0

Channels are not shared

**Summary**

Multipliers	22
Adders/Subtractors	43
Registers	91
RAMs	0
Multiplexers	1

Channels are shared

# Implement FFT for FPGA Using FFT HDL Optimized Block

This example shows how to use the FFT HDL Optimized block to implement a FFT for hardware.

The FFT and IFFT HDL Optimized blocks and system objects support simulation and HDL code generation for many applications. They provide two architectures optimized for different use cases:

- **Streaming Radix  $2^2$**  - For high throughput applications. Achieves gigasamples per second (GSPS) when you use vector input.
- **Burst Radix 2** - For low area applications. Uses only one complex butterfly.

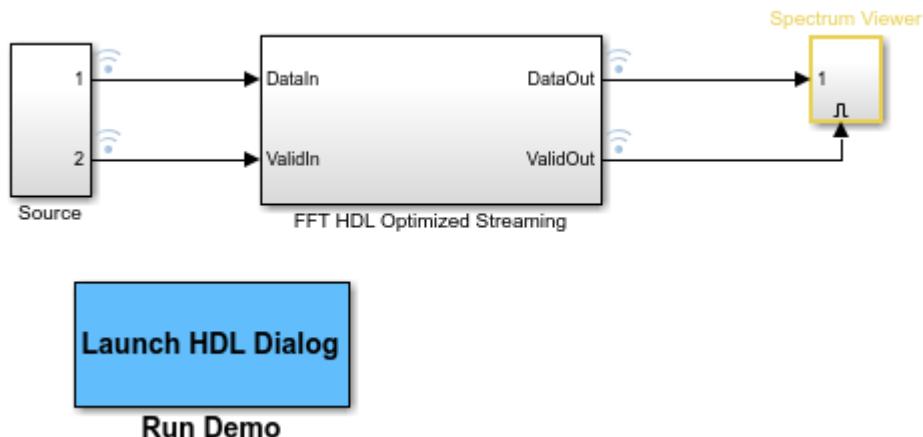
This example includes two models that show how to use the streaming and burst architectures of the FFT HDL Optimized block, respectively.

## Streaming Radix $2^2$ Architecture

Modern ADCs are capable of sampling signals at sample rates up to several gigasamples per second. However, clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. One way to perform GSPS processing on an FPGA is to process multiple samples at the same time at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface that accepts scalar input at GHz clock rate and produces a vector of samples at a lower clock rate. Therefore modern signal processing requires vector processing.

The **Streaming Radix  $2^2$**  architecture is designed to support the high throughput use case. This example model uses an input vector size of 8, and calculates the FFT using the **Streaming Radix  $2^2$**  architecture. For timing diagram, supported features, and FPGA resource usage, see the FFT HDL Optimized (DSP System Toolbox) block reference page.

```
modelname = 'FFTHDL0optimizedExample_Streaming';
open_system(modelname);
```



The InitFcn callback (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

```
FFTLength = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at  $1*2e6$  Hz. The input vector size is 8 samples.

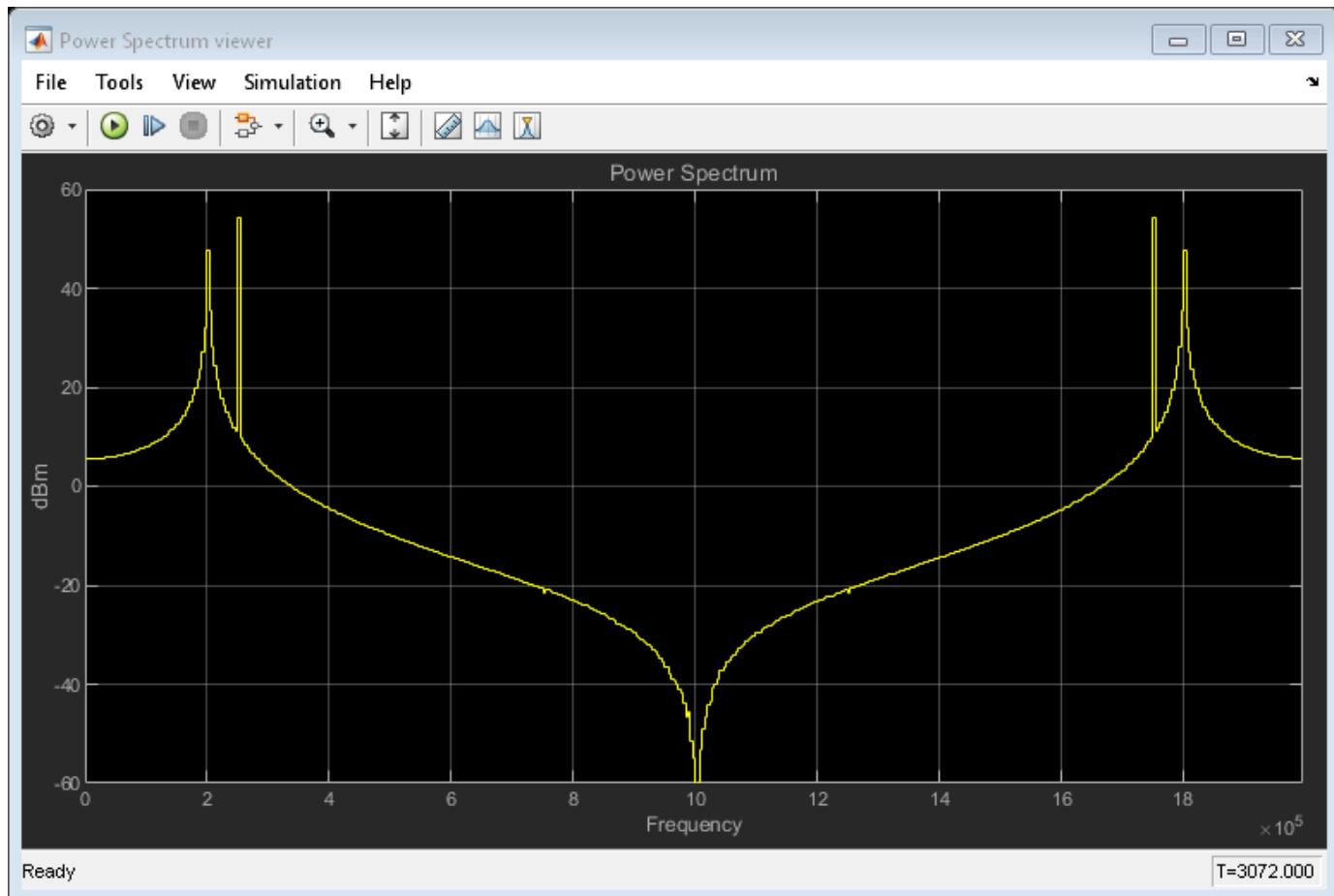
```
FrameSize      = 8;
Fs             = 1*2e6;
```

To demonstrate that data does not need to come continuously, this example applies valid input every other cycle.

```
ValidPattern = [1,0];
```

Open the Spectrum Viewer and run the example model.

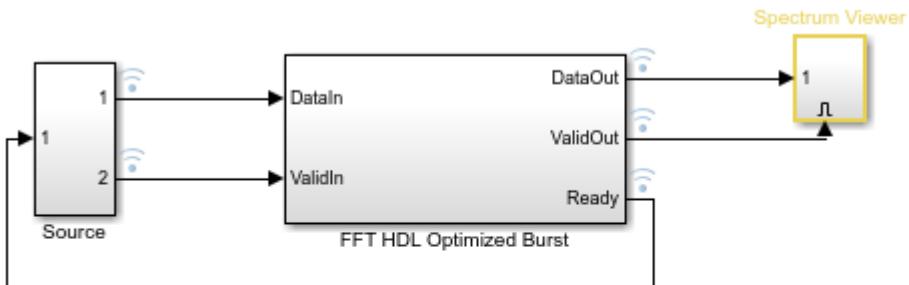
```
open_system('FFTHDL0optimizedExample_Streaming/Spectrum Viewer/Power Spectrum viewer');
sim(modelname);
```



### Burst Radix 2 (Minimum Resource) Architecture

Use the **Burst** Radix 2 architecture for applications with limited FPGA resources, especially when the FFT length is big. This architecture uses only one complex butterfly to calculate the FFT. The design accepts data while **ready** is asserted, and starts processing once the whole FFT frame is saved into the memory. While processing, the design cannot accept data, so **ready** is de-asserted. For timing diagram, supported features, and FPGA resource usage, see the FFT HDL Optimized (DSP System Toolbox) block reference page.

```
modelname = 'FFTHDL0optimizedExample_Burst';
open_system(modelname);
```



**Launch HDL Dialog**  
**Run Demo**

The InitFcn callback (Model Properties > Callbacks > InitFcn) sets parameters for the model. In this example, the parameters control the size of the FFT and the input data characteristics.

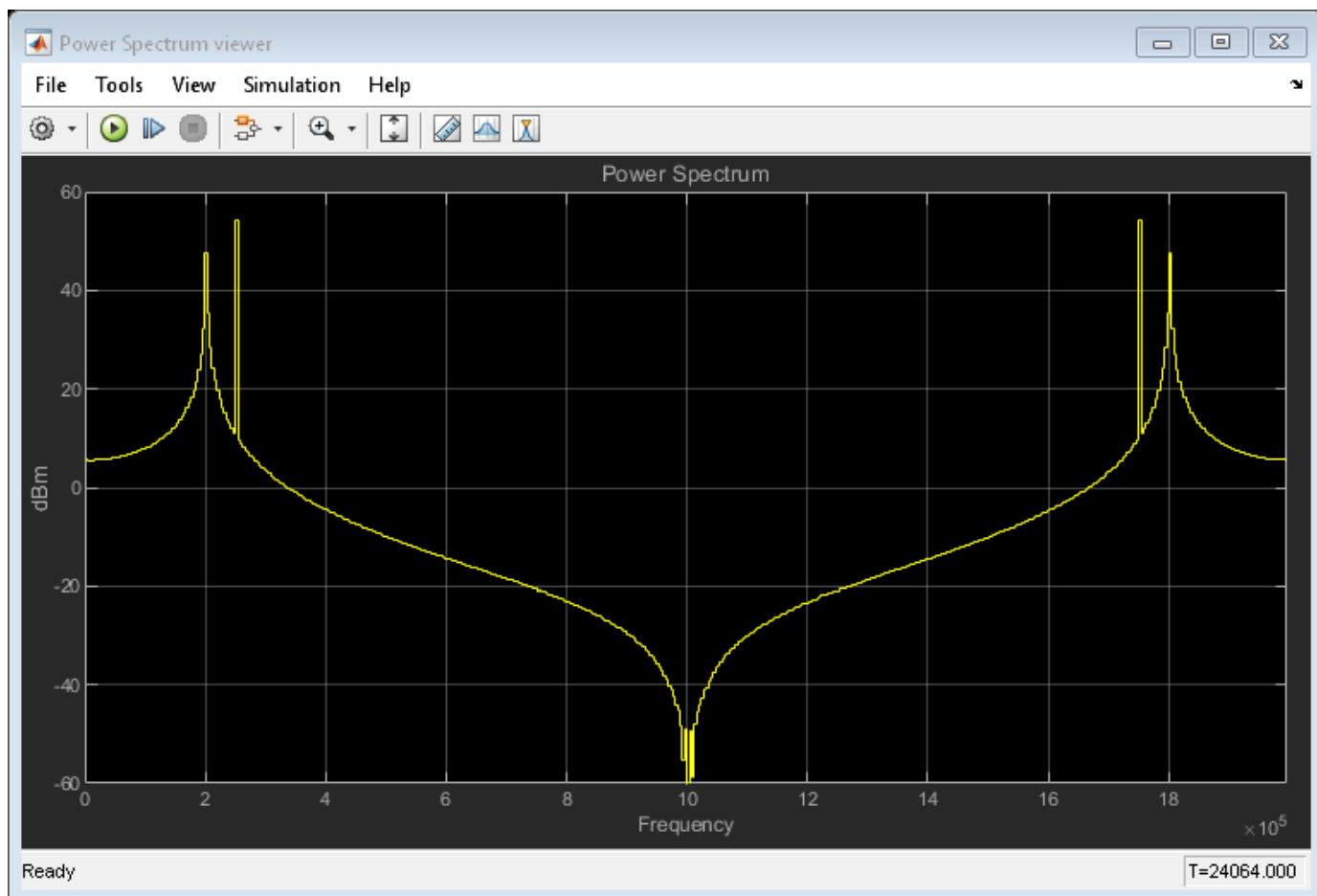
```
FFTLength = 512;
```

The input data is two sine waves, 200 KHz and 250 KHz, each sampled at  $1*2e6$  Hz. Data is valid every cycle.

```
Fs = 1*2e6;
ValidPattern = 1;
```

Open the Spectrum Viewer and run the example model.

```
open_system('FFTHDLOptimizedExample_Burst/Spectrum Viewer/Power Spectrum viewer');
sim(modelname);
```



### Generate HDL Code and Test Bench

An HDL Coder™ license is required to generate HDL for this example.

Choose one of the models to generate HDL code and test bench:

```
systemname = 'FFTHDL0optimizedExample_Burst/FFT HDL Optimized Burst';
```

or

```
systemname = 'FFTHDL0optimizedExample_Streaming/FFT HDL Optimized Streaming';
```

Use this command to generate HDL code for either FFT mode. The generated can be used for any FPGA or ASIC target.

```
makehdl(systemname);
```

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior.

```
makehdltb(systemname);
```

# High Throughput Channelizer for FPGA

This example shows how to implement a high throughput (Gigasamples per second, GSPS) channelizer for hardware by using a polyphase filter bank.

High speed signal processing is a requirement for applications such as radar, broadband wireless and backhaul.

Modern ADCs are capable of sampling signals at sample rates up to several Gigasamples per second. But the clock speeds for the fastest FPGA fall short of this sample rate. FPGAs typically run at hundreds of MHz. An approach to perform GSPS processing on an FPGA is to move from scalar processing to vector processing and process multiple samples at the same time at a much lower clock rate. Many modern FPGAs support the JESD204B standard interface that accepts scalar input at GHz clock rate and produces a vector of samples at a lower clock rate.

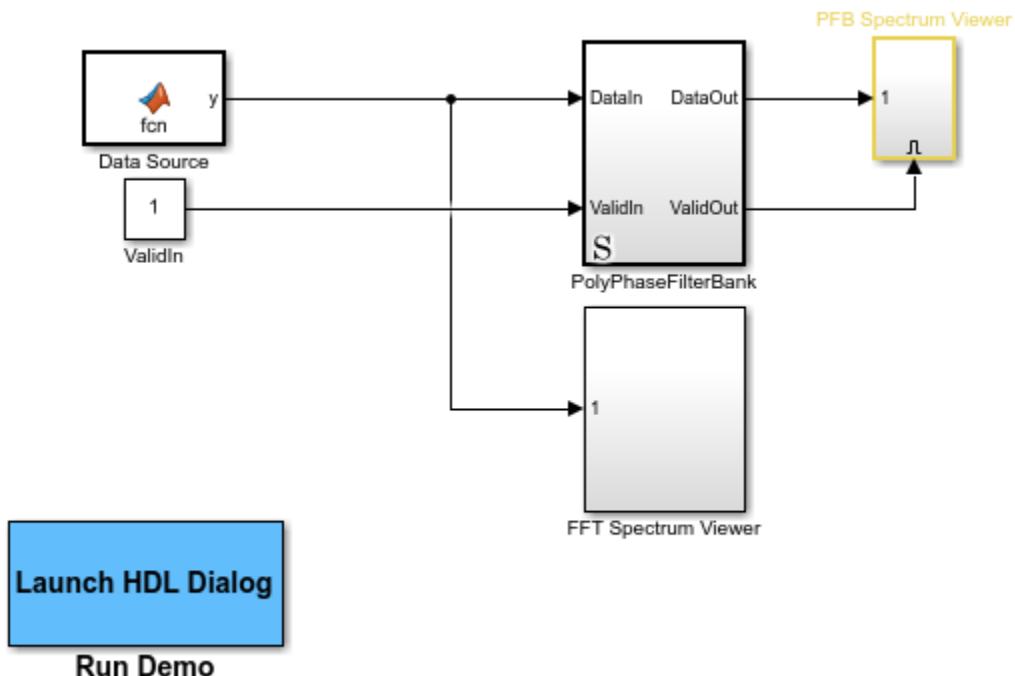
In this example we show how to design a signal processing application for GSPS throughput in Simulink. Input data is vectorized through a JESD204B interface and available at a lower clock rate in the FPGA. The model is a polyphase filter bank which consists of a filter and an FFT that processes 16 samples at a time. The polyphase filter bank technique is used to minimize the FFT's inaccuracy due to leakage and scalloping loss. See "High Resolution Spectral Analysis" (DSP System Toolbox) for more information about the polyphase filter bank.

The first part of the example implements a polyphase filter bank with a 4-tap filter.

The second part of the example uses the Channelizer HDL Optimized block configured for a 12-tap filter. The Channelizer HDL Optimized block uses the polyphase filter bank technique.

## Polyphase Filter Bank

```
modelname = 'PolyphaseFilterBankHDLExample_4tap';
open_system(modelname);
```



Copyright 2016 The MathWorks, Inc.

The `InitFcn` callback (Model Properties > Callbacks > InitFcn) sets up the model. This model uses a 512-point FFT with a four tap filter for each band. Use the `dsp.Channelizer` (DSP System Toolbox) System object™ to generate the coefficients. The polyphase method of the Channelizer object generates a 512-by-4 matrix. Each row represents the coefficients for each band. The coefficients are cast into fixed-point with the same word length as the input signal.

```
FFTLength = 512;
h = dsp.Channelizer;
h.NumTapsPerBand = 4;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef = fi(polyphase(h),1,15,14,'RoundingMethod','Convergent');
```

The algorithm requires 512 filters (one filter for each band). For a vector input of 16 samples we can reuse 16 filters, 32 times.

```
InVect      = 16;
ReuseFactor = FFTLength/InVect;
```

To synthesize the filter to a higher clock rate, we pipeline the multiplier and the coefficient bank. These values are explained in the "Optimized Hardware Considerations" section.

```
Multiplication_PipeLine = 2;
CoefBank_PipeLine      = 1;
```

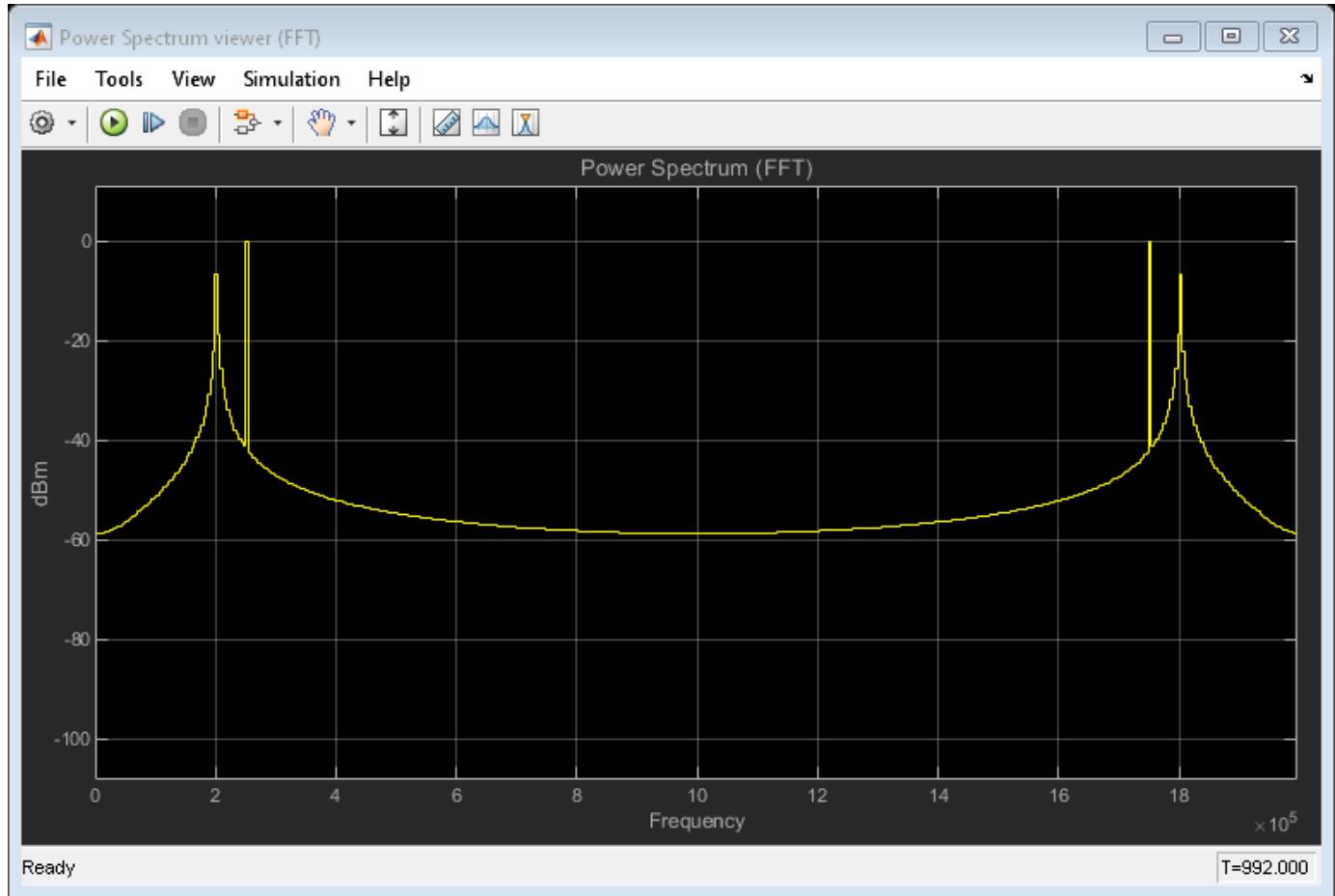
### Data Source

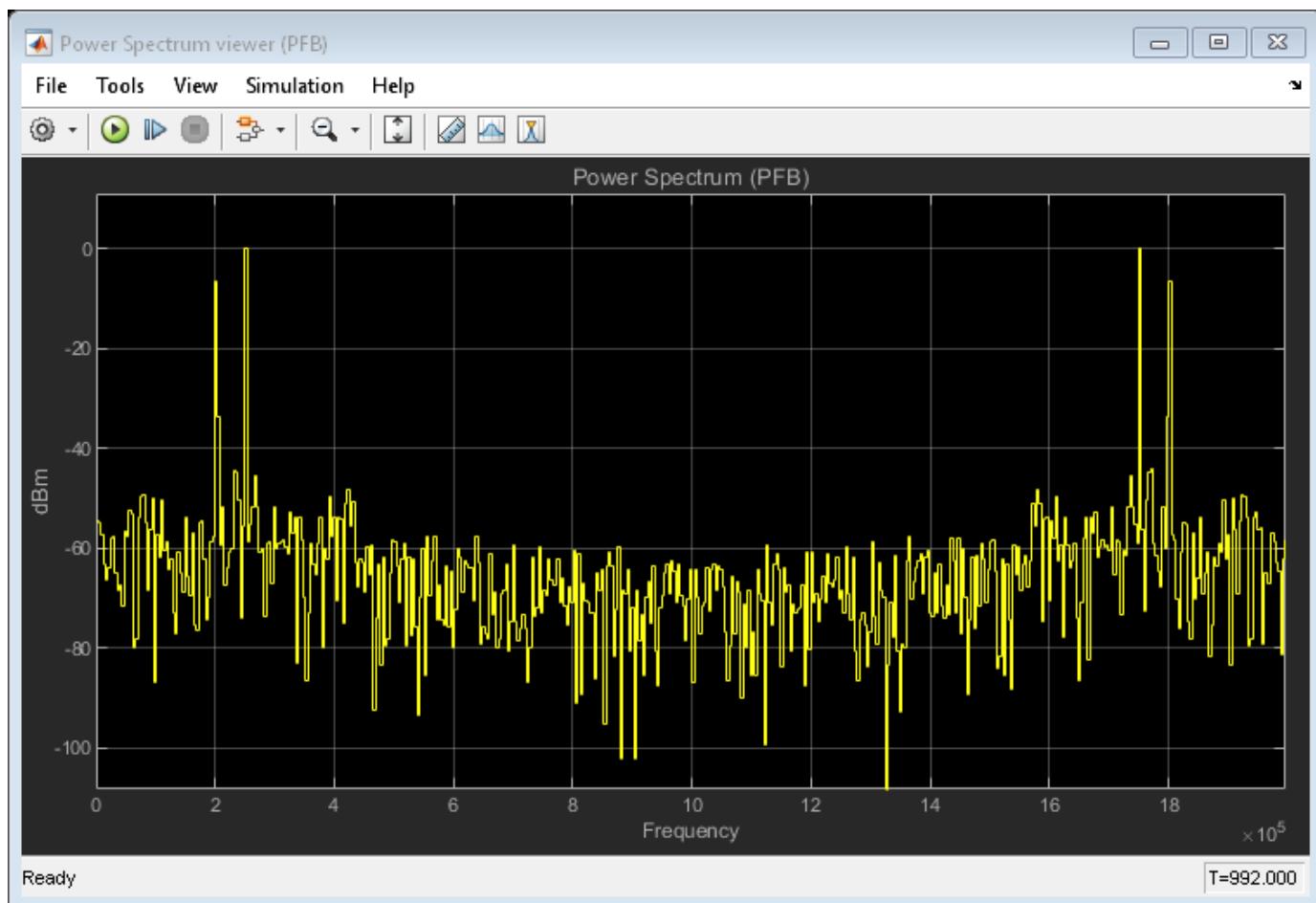
The input data consists of two sine waves, 200 KHz and 250 KHz.

## Simulation Results

To visualize the spectrum result, open the spectrum viewers and run the model.

```
open_system('PolyphaseFilterBankHDLExample_4tap/FFT Spectrum Viewer/Power Spectrum viewer (FFT)');  
open_system('PolyphaseFilterBankHDLExample_4tap/PFB Spectrum Viewer/Power Spectrum viewer (PFB)');  
sim(modelname);
```





The polyphase filter bank Power Spectrum Viewer shows the improvement in the power spectrum and minimization of frequency leakage and scalloping compared with using only an FFT. By comparing the two spectrums, and zooming in between 100 KHz and 300 KHz, observe that the polyphase filter bank has fewer peaks over -40 dB than the classic FFT.

### Optimized Hardware Considerations

- **Data type :** The data word length affects both the accuracy of the result and the resources used in the hardware. For this example we design the filter at full precision. With an input data type of `fixdt(1,15,13)`, the output is `fixdt(1,18,17)`. The absolute values of the filter coefficients are all smaller than 1 so the data doesn't grow after each multiplication, and we need to add one bit for each addition. To keep the accuracy in the FFT, we need to grow one bit for each stage. This makes the twiddle factor multiplication bigger at each stage. For many FPGAs it is desirable to keep multiplication size smaller than 18x18. Since a 512 point FFT has 9 stages, the input of the FFT cannot be more than 11 bits. By exploring the filter coefficients, we observe that the first 8 binary digits of the maximum coefficient are zero, and therefore we can cast the coefficients to `fixdt(1,7,14)` instead of `fixdt(1,15,14)`. Also we observe that the maximum value of the Datatype block output inside the polyphase filter bank has 7 leading zeros after the binary point, and therefore we cast the filter output to `fixdt(1,11,17)` instead. This keeps the multiplier size inside the FFT smaller than 18-by-18 and saves hardware resources.
- **Design for speed:**

- 1** *State control block:* The State Control block is used in Synchronous mode to generate hardware friendly code for the delay block with enable port.
- 2** *Minimize clock enable :* The model is set to generate HDL code with the Minimize Clock Enable option turned on (In Configuration Parameters choose > HDL Code Generation > Global settings > Ports > Minimize clock enables). This option is supported when the model is single rate. Clock enable is a global signal which is not recommended for high speed designs.
- 3** *Usage of the DSP block in FPGA:* In order to map multipliers into a DSP block in the FPGA, the multipliers should be pipelined. In this example we pipeline the multipliers (2 delays before and 2 delays after) by setting Multiplication\_PipeLine = 2; These pipeline registers should not have a reset. Set the reset type to none for each pipeline (right-click the Delay block and select HDL Code > HDL Block Properties > Reset Type = None).
- 4** *Usage of ROM in FPGA:* The Coefficient block inside the Coefficient Bank is a combinatorial block. In order to map this block into a ROM, add a register after the block. The delay length is set by CoefBank\_PipeLine. Set the reset type for these delays to none (right-click the Delay block and select HDL Code > HDL Block Properties > Reset Type = None).

### Generate HDL Code and Test Bench

You must have an HDL Coder™ license to generate HDL code for this example model. Use this command to generate HDL code. systemname = 'PolyphaseFilterBankHDLExample\_4tap/PolyPhaseFilterBank'; makehdl(systemname);

Use this command to generate a test bench that compares the results of an HDL simulation against the Simulink simulation behavior. makehdltb(systemname);

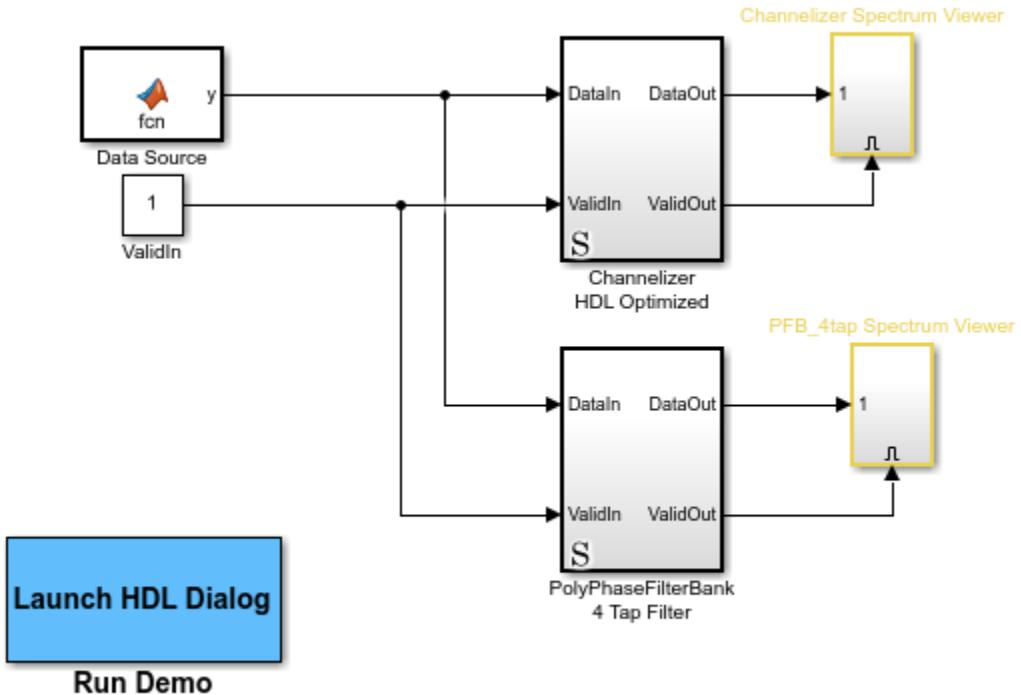
### Synthesis Results

The design was synthesized for Xilinx Virtex 7 (xc7vx550t-ffg1158, speed grade 3) using ISE. The design achieves a clock frequency of 499.525 MHz (before place and route). At 16 samples per clock, this translates to 8 GSPS throughput. Note that this subsystem has a high number of I/O ports and it is not suitable as a standalone design targeted to the FPGA.

### HDL Optimized Channelizer

To improve the frequency response, use a filter with more taps. The following model uses the Channelizer HDL Optimized block, configured with a 12-tap filter to improve the spectrum. Using a built-in Channelizer HDL Optimized block makes it easier to change design parameters.

```
modelName = 'PolyphaseFilterBankHDLExample_HDLChannelizer';
open_system(modelName);
```



The model uses workspace variables to configure the FFT and filter. In this case, the model uses a 512-point FFT and a 12-tap filter for each band. The number of coefficients for the channelizer is 512 frequency bands times 12 tap per frequency band. The `tf(h)` method generates all the coefficients.

```
InVect      = 16;
FFTLength  = 512;
h = dsp.Channelizer;
h.NumTapsPerBand = 12;
h.NumFrequencyBands = FFTLength;
h.StopbandAttenuation = 60;
coef12Tap = tf(h);
```

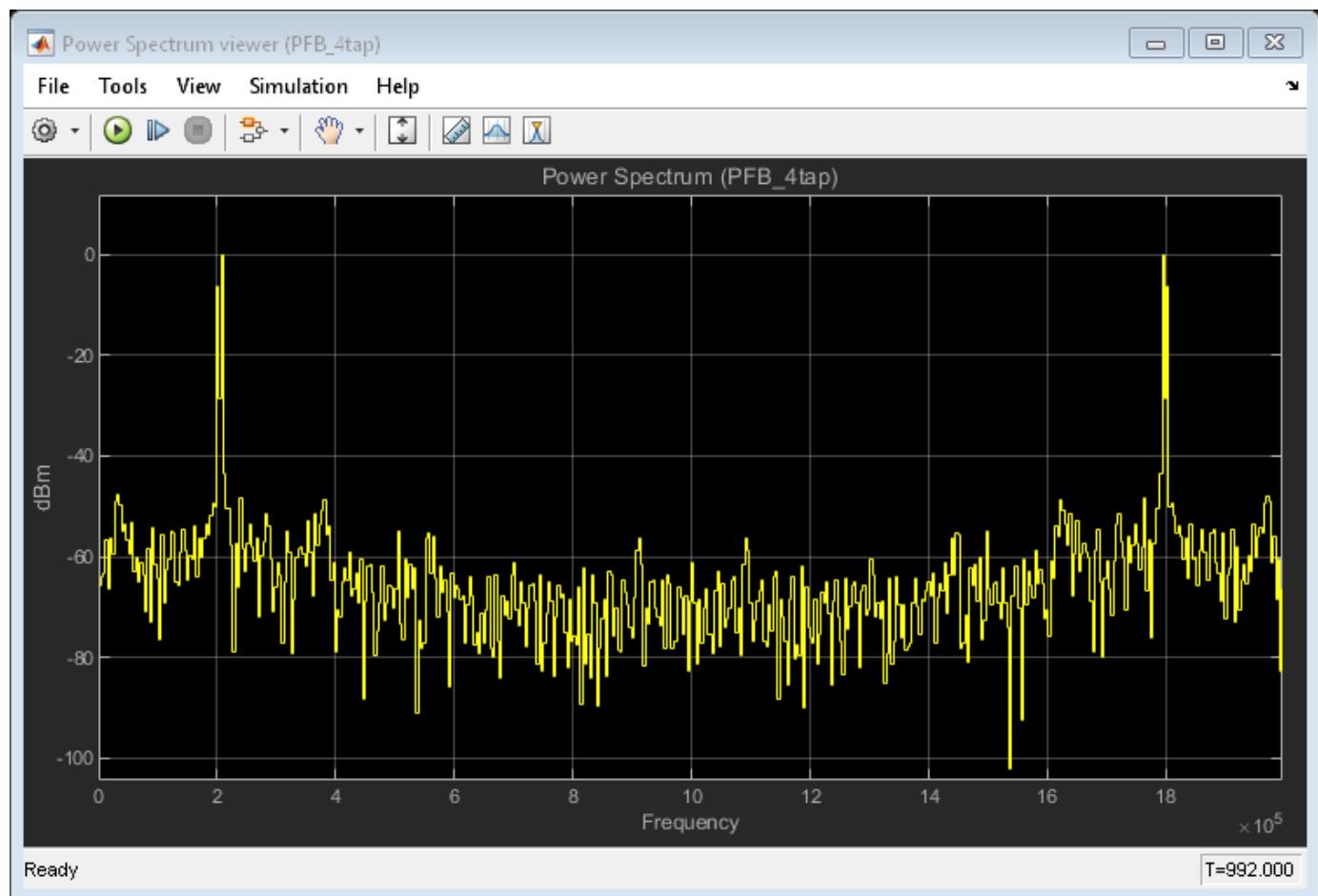
## Data Source

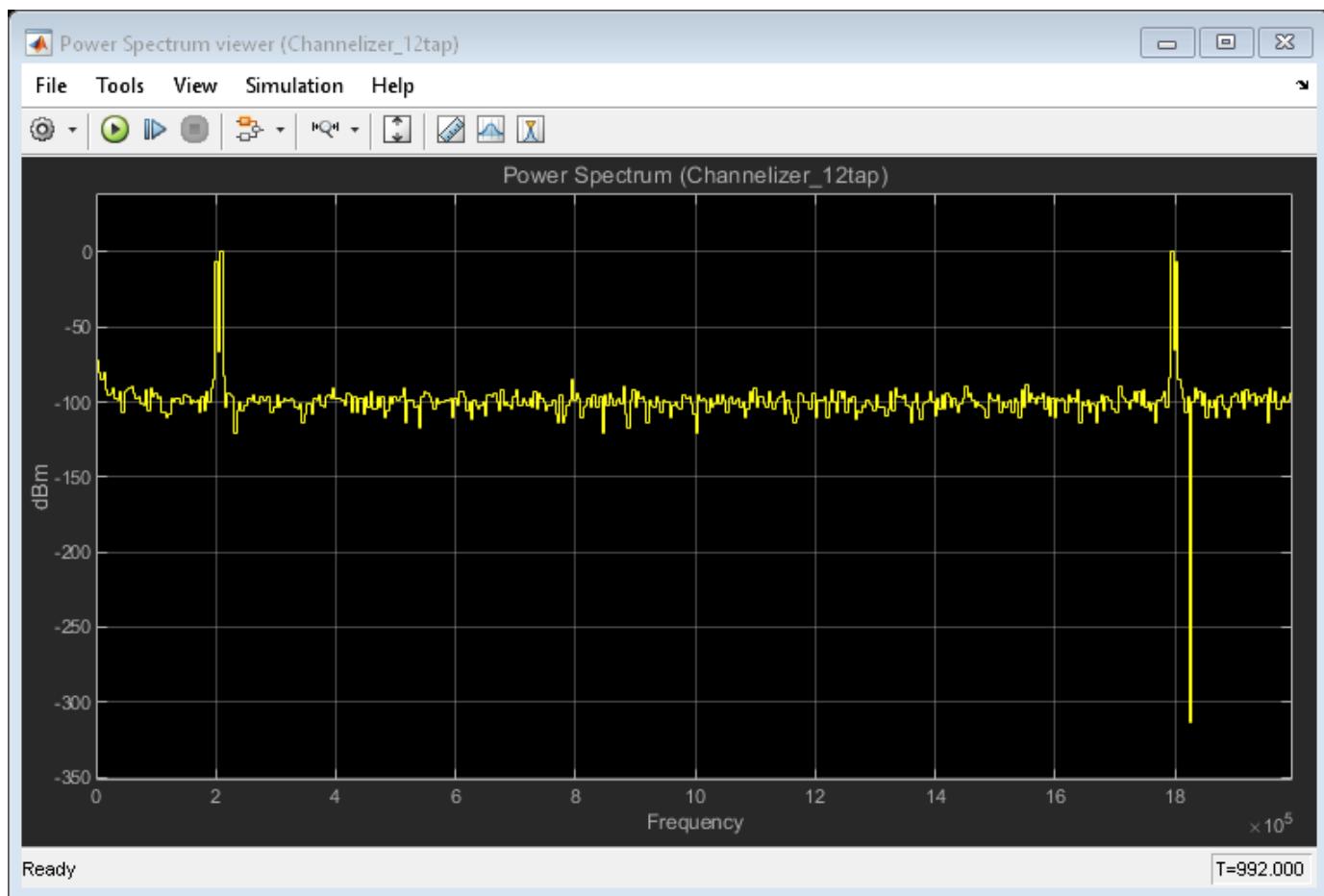
The input data consists of two sine waves, 200 KHz and 206.5 KHz. The frequencies are closer to each other than the first example to illustrate the difference between a channelizer and a 4-tap filter in spectrum resolution.

## Simulation Results

To visualize the spectrum result, open the spectrum viewers and run the model.

```
open_system('PolyphaseFilterBankHDLExample_HDLChannelizer/PFB_4tap Spectrum Viewer/Power Spectrum');
open_system('PolyphaseFilterBankHDLExample_HDLChannelizer/Channelizer Spectrum Viewer/Power Spectrum');
sim(modelname);
```





The Power Spectrum Viewer for the Channelizer\_12tap model shows the improvement in the power spectrum of the polyphase filter bank with 12-tap filter compared to the 4-tap filter in the previous model. Compare the spectrum results for the channelizer and 4-tap polyphase filter banks. Zoom in between 100 KHz and 300 KHz to observe that the channelizer detects only two peaks while the 4-tap polyphase filter bank detects more than 2 peaks. Two peaks is the expected result since the input signal has only two frequency components.

# HDL Implementation of a Digital Down-Converter for LTE

This example shows how to design a digital down-converter (DDC) for radio communication applications such as LTE, and generate HDL code with HDL Coder™.

## Introduction

DDCs are widely used in digital communication receivers to convert Radio Frequency (RF) or Intermediate Frequency (IF) signals to baseband. The DDC operation shifts the signal to a lower frequency and reduces its sampling rate to facilitate subsequent processing stages. The DDC presented here performs complex frequency translation followed by sample rate conversion using a 4-stage filter chain. The example starts by designing the DDC with DSP System Toolbox™ functions in floating point. Each stage is then converted to fixed-point, and then used in a Simulink® model which generates synthesizable HDL code. Two test signals are used to demonstrate and verify the DDC operation:

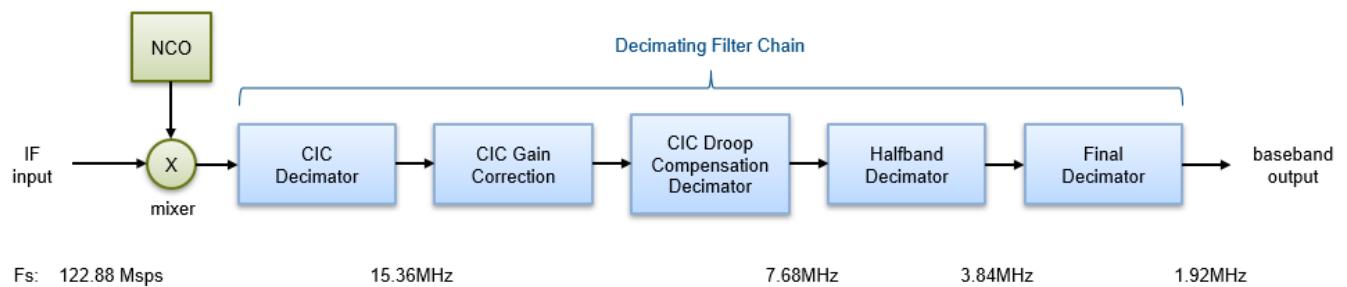
- 1 A sinusoid modulated onto a 32 MHz IF carrier.
- 2 An LTE downlink signal with a bandwidth of 1.4 MHz, modulated onto a 32 MHz IF carrier.

The example measures signal quality at the output of the floating-point and fixed-point DDCs, and compares the two. Finally, FPGA implementation results are presented.

Note: This example uses `DDCTestUtils`, a helper class containing functions for generating stimulus and analyzing the DDC output. See the `DDCTestUtils.m` file for more info.

## DDC Structure

The DDC consists of a Numerically Controlled Oscillator (NCO), a mixer, and a decimating filter chain. The filter chain consists of a CIC decimator, CIC gain correction, a CIC compensation decimator (FIR), a halfband FIR decimator, and a final FIR decimator. The overall response of the filter chain is equivalent to that of a single decimation filter with the same specification, however, splitting the filter into multiple decimation stages results in a more efficient design which uses fewer hardware resources. The CIC decimator provides a large initial decimation factor, which enables subsequent filters to work at lower rates. The CIC compensation decimator improves the spectral response by compensating for the CIC droop while decimating by two. The halfband is an intermediate decimator while the final decimator implements the precise Fpass and Fstop characteristics of the DDC. Due to the lower sampling rates, the filters nearer the end of the chain can optimize resource use by sharing multipliers. A block diagram of the DDC is shown below.



The input to the DDC is sampled at 122.88 Msps while the output sample rate is 1.92 Msps. Therefore the overall decimation factor is 64. 1.92 Msps is the typical sampling rate used by LTE receivers to perform cell search and MIB (Master Information Block) recovery. The DDC filters have therefore been designed to suit this application. The DDC is optimized to run at a clock rate of 122.88 MHz.

### **DDC Design**

This section explains how to design the DDC using floating-point operations and filter-design functions in MATLAB®.

### **DDC Parameters**

The desired DDC response is defined by the input sampling rate, carrier frequency, and filter characteristics. Modifying this desired filter response may require changes to the HDL Block Properties of the filter blocks in the Simulink model. HDL Block Properties are discussed later in the example.

```
FsIn = 122.88e6; % Sampling rate at input to DDC
Fc = 32e6; % Carrier frequency
Fpass = 540e3; % Passband frequency, equivalent to 36x15kHz LTE subcarriers
Fstop = 700e3; % Stopband frequency
Ap = 0.1; % Passband ripple
Ast = 60; % Stopband attenuation
```

The rest of this section shows how to design each filter in turn.

### **Cascade Integrator-Comb (CIC) Decimator**

The first filter stage is implemented as a CIC decimator because of its ability to implement a large decimation factor efficiently. The response of a CIC filter is similar to a cascade of moving average filters, however no multipliers or divides are used. As a result, the CIC filter has a large DC gain.

```
cicParams.DecimationFactor = 8;
cicParams.DifferentialDelay = 1;
cicParams.NumSections = 3;
cicParams.FsOut = FsIn/cicParams.DecimationFactor;

cicFilt = dsp.CICDecimator(cicParams.DecimationFactor, ...
    cicParams.DifferentialDelay, cicParams.NumSections) %#ok<*NOPTS>

cicGain = gain(cicFilt)

cicFilt =
dsp.CICDecimator with properties:
    DecimationFactor: 8
    DifferentialDelay: 1
    NumSections: 3
    FixedPointDataType: 'Full precision'

cicGain =
```

512

The CIC gain is a power of two, therefore it can be easily corrected for in hardware with a shift operation. For analysis purposes, the gain correction is represented in MATLAB by a one-tap `dsp.FIRFilter` System object.

```
cicGainCorr = dsp.FIRFilter('Numerator',1/cicGain)
```

```
cicGainCorr =
```

```
dsp.FIRFilter with properties:
```

```
    Structure: 'Direct form'
    NumeratorSource: 'Property'
        Numerator: 0.0020
    InitialConditions: 0
```

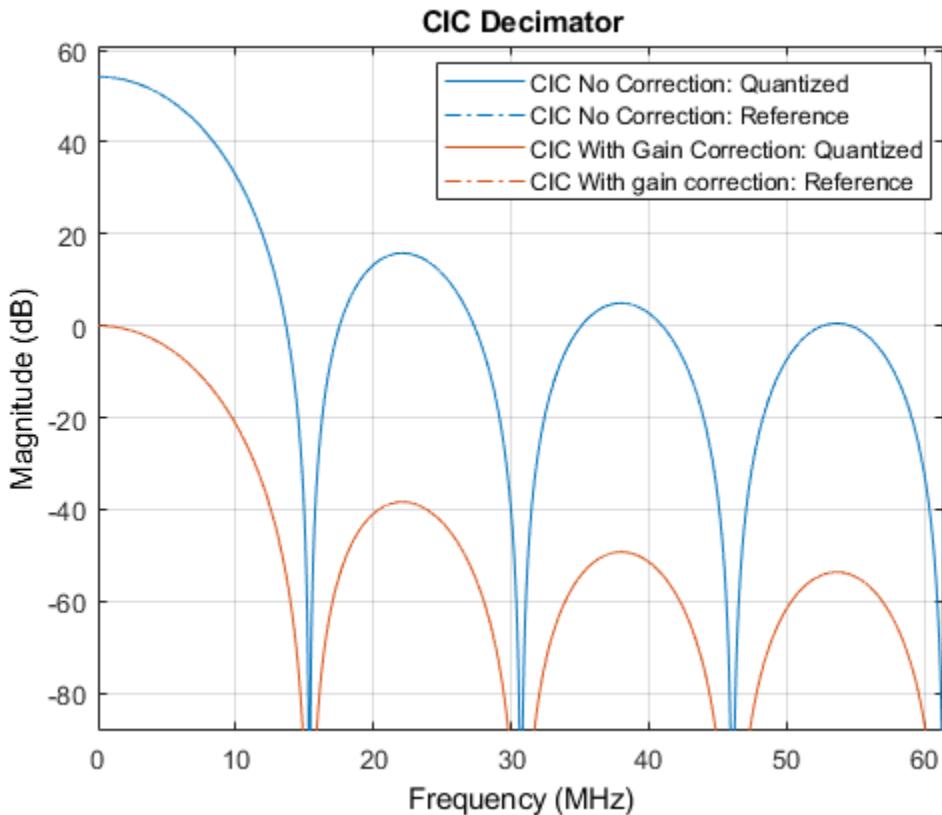
```
Use get to show all properties
```

Use `fvtool` to display the magnitude response of the CIC filter with and without gain correction. For analysis, combine the CIC filter and the gain correction filter into a `dsp.FilterCascade` System object. CIC filters always use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ddcPlots.cicDecim = fvtool(
    cicFilt, ...
    dsp.FilterCascade(cicFilt,cicGainCorr), ...
    'Fs',[FsIn,FsIn]);

DDCTestUtils.setPlotNameAndTitle('CIC Decimator');

legend(...,
    'CIC No Correction: Quantized', ...
    'CIC No Correction: Reference', ...
    'CIC With Gain Correction: Quantized', ...
    'CIC With gain correction: Reference');
```



### CIC Droop Compensation Filter

The magnitude response of the CIC filter has a significant *droop* within the passband region, therefore an FIR-based droop compensation filter is used to flatten the passband response. The droop compensator is configured with the same parameters as the CIC decimator. This filter also implements decimation by a factor of two, therefore its bandlimiting characteristics are specified. Specify the filter requirements and then use the `design` function to return a filter System object with those characteristics.

```

compParams.R      = 2;                      % CIC compensation decimation factor
compParams.Fpass = Fstop;                   % CIC comp passband frequency
compParams.FsOut = cicParams.FsOut/compParams.R; % New sampling rate
compParams.Fstop = compParams.FsOut - Fstop;   % CIC comp stopband frequency
compParams.Ap     = Ap;                      % Same Ap as overall filter
compParams.Ast    = Ast;                     % Same Ast as overall filter

compSpec = fdesign.decimator(compParams.R, 'ciccomp', ...
    cicParams.DifferentialDelay, ...
    cicParams.NumSections, ...
    cicParams.DecimationFactor, ...
    'Fp,Fst,Ap,Ast',...
    compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
    cicParams.FsOut);

compFilt = design(compSpec, 'SystemObject',true)

```

```

compFilt =
dsp.FIRDecimator with properties:

    NumeratorSource: 'Property'
        Numerator: [-0.0398 -0.0126 0.2901 0.5258 0.2901 -0.0126 -0.0398]
    DecimationFactor: 2
        Structure: 'Direct form'

Use get to show all properties

```

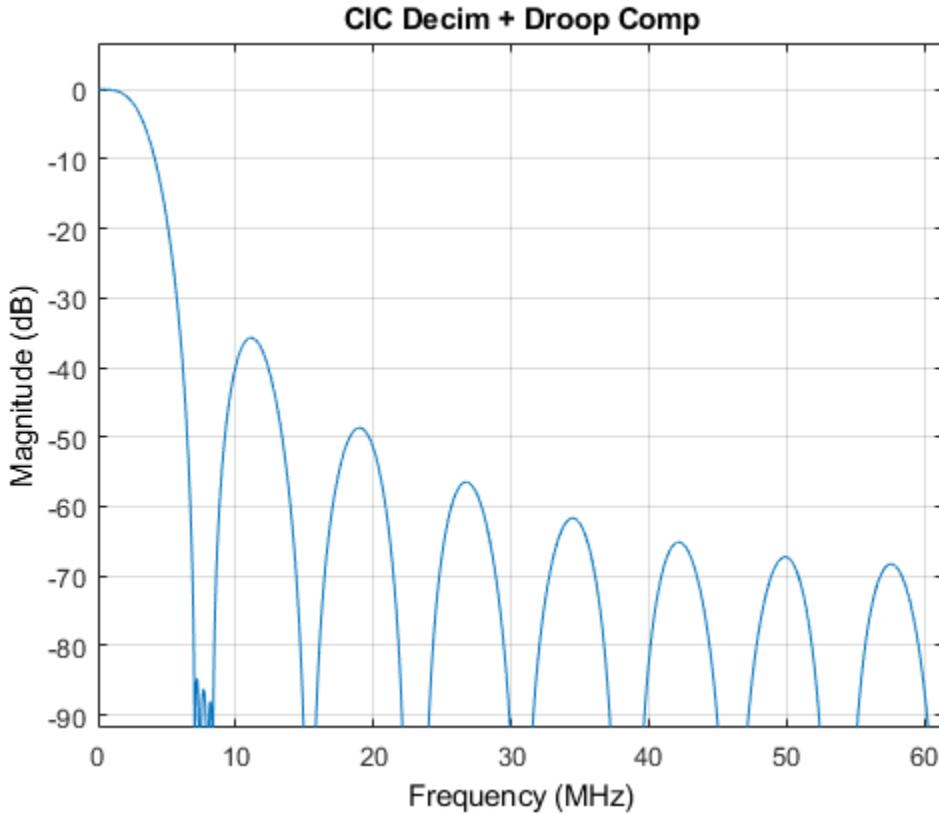
Plot the combined response of the CIC filter (with gain correction) and droop compensation.

```

ddcPlots.cicComp = fvtool(
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt), ...
    'Fs',FsIn,'Legend','off');

DDCTestUtils.setPlotNameAndTitle('CIC Decim + Droop Comp');

```



### Halfband Decimator

The halfband filter provides efficient decimation by two. Halfband filters are efficient because approximately half of their coefficients are equal to zero.

```

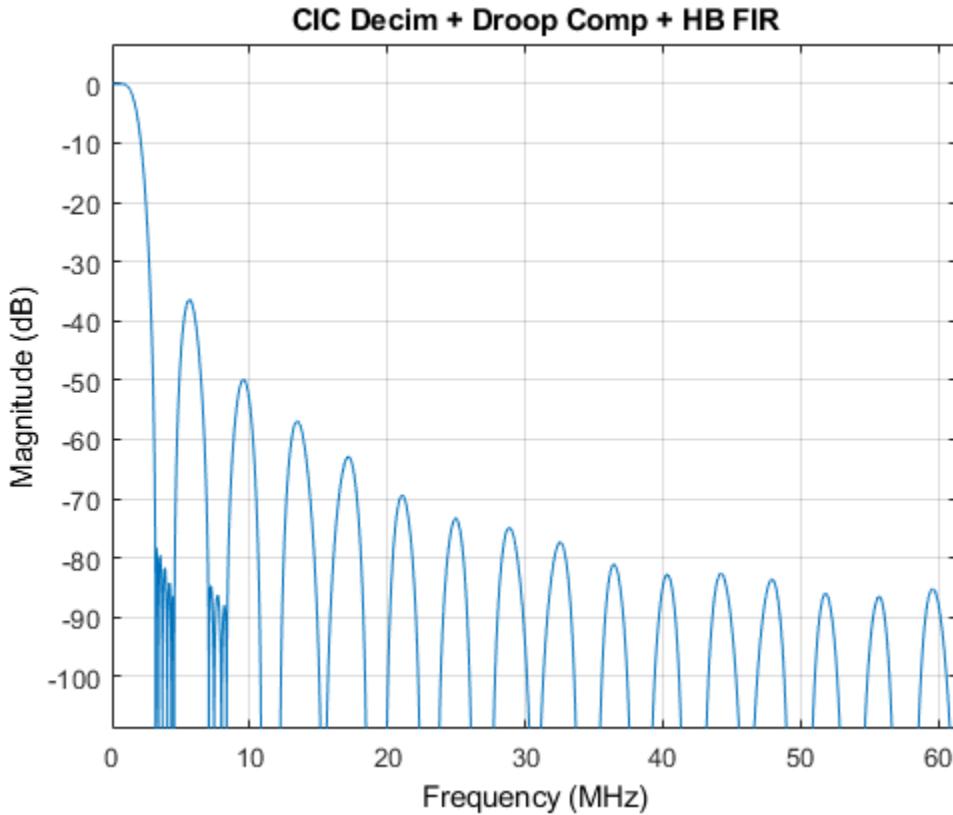
hbParams.FsOut          = compParams.FsOut/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fstop;

```

```
hbParams.StopbandAttenuation = Ast;  
  
hbSpec = fdesign.decimator(2, 'halfband', ...  
    'Tw,Ast', ...  
    hbParams.TransitionWidth, ...  
    hbParams.StopbandAttenuation, ...  
    compParams.FsOut);  
  
hbFilt = design(hbSpec, 'SystemObject', true)  
  
hbFilt =  
  
dsp.FIRDecimator with properties:  
  
    NumeratorSource: 'Property'  
        Numerator: [1x11 double]  
    DecimationFactor: 2  
        Structure: 'Direct form'  
  
Use get to show all properties
```

Plot the response of the DDC up to the halfband filter output.

```
ddcPlots.halfbandFIR = fvtool(...  
    dsp.FilterCascade(cicFilt, cicGainCorr, compFilt, hbFilt), ...  
    'Fs', FsIn, 'Legend', 'off');  
  
DDCTestUtils.setPlotNameAndTitle('CIC Decim + Droop Comp + HB FIR');
```



### Final FIR Decimator

The final FIR implements the detailed passband and stopband characteristics of the DDC. This filter has more coefficients than the preceding FIR filters, however it operates at a lower sampling rate, which enables more resource sharing on hardware.

```
% Add 3dB of headroom to the stopband attenuation so that the DDC still meets the
% spec after fixed-point quantization. This value was determined by trial and error
% with |fvtool|.
finalSpec = fdesign.decimator(2,'lowpass',...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast+3,hbParams.FsOut);

finalFilt = design(finalSpec,'equiripple','SystemObject',true)

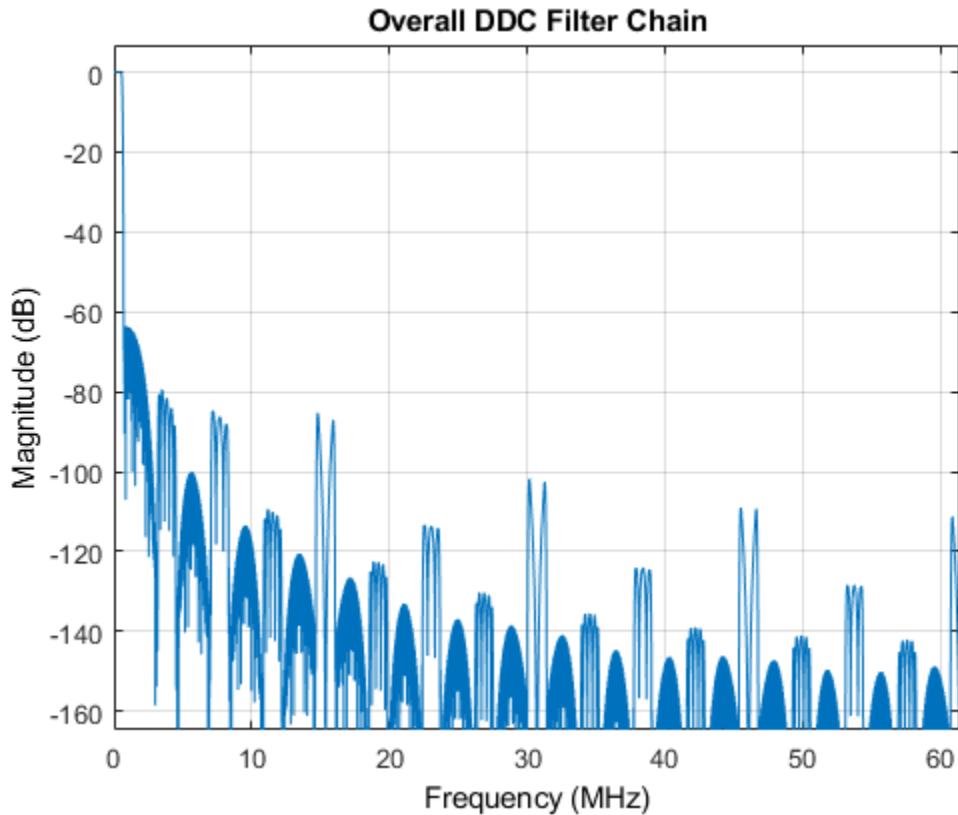
finalFilt =
dsp.FIRDecimator with properties:

    NumeratorSource: 'Property'
        Numerator: [1x70 double]
    DecimationFactor: 2
        Structure: 'Direct form'

Use get to show all properties
```

Visualize the overall magnitude response of the DDC.

```
ddcFilterChain = dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.overallResponse = fvtool(ddcFilterChain,'Fs',FsIn,'Legend','off');
DDCTestUtils.setPlotNameAndTitle('Overall DDC Filter Chain');
```



### Fixed-Point Conversion

The frequency response of the floating-point DDC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

### Filter Quantization

This example uses 16-bit coefficients, which is sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks required for an FPGA implementation. The input to the DDC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provides extra headroom and precision in the intermediate signals.

For the CIC decimator, choosing the `Minimum section word lengths` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

Configure the fixed-point parameters of the gain correction and FIR-based System objects. While not shown explicitly, the object uses the default `RoundingMethod` and `OverflowAction` settings (`Floor` and `Wrap` respectively).

```
% CIC Gain Correction
cicGainCorr.FullPrecisionOverride = false;
cicGainCorr.CoefficientsDataType = 'Custom';
cicGainCorr.CustomCoefficientsDataType = numerictype(fi(cicGainCorr.Numerator,1,16));
cicGainCorr.OutputDataType = 'Custom';
cicGainCorr.CustomOutputDataType = numerictype(1,18,16);

% CIC Droop Compensation
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,15);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,16);

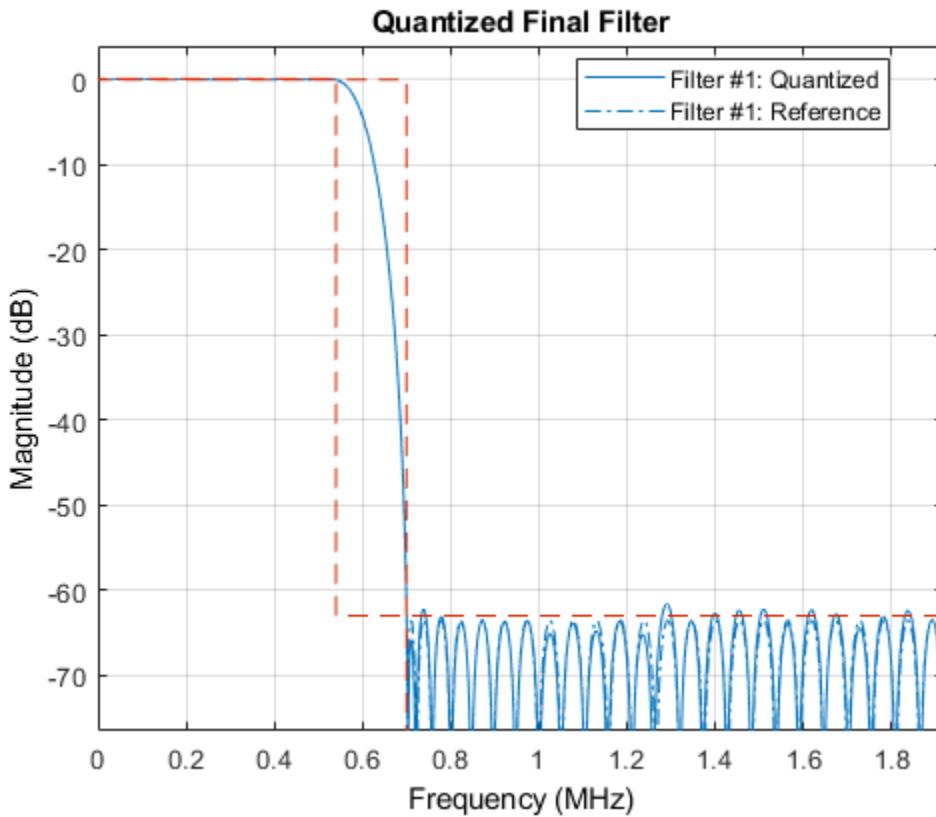
% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,15);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,16);

% FIR
finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],16,15);
finalFilt.ProductDataType = 'Full precision';
finalFilt.AccumulatorDataType = 'Full precision';
finalFilt.OutputDataType = 'Custom';
finalFilt.CustomOutputDataType = numerictype([],18,16);
```

### Fixed-Point Analysis

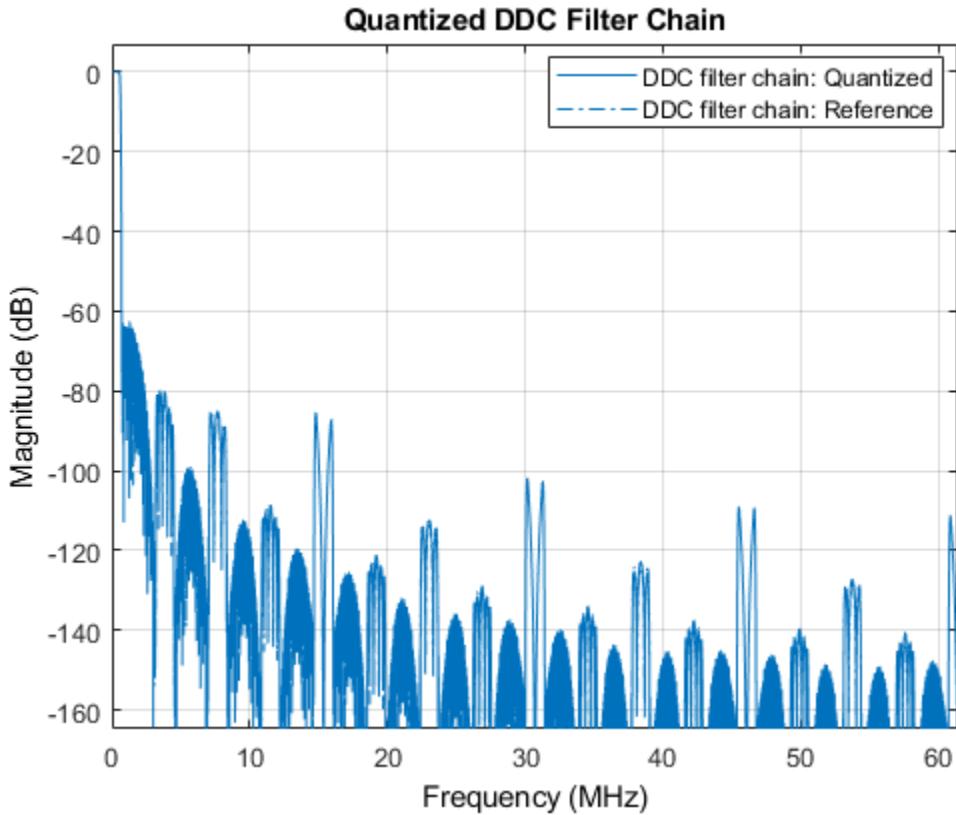
Inspect the quantization effects with `fvtool`. The filters can be analyzed individually, or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, the effect of quantizing the final FIR filter stage is shown.

```
ddcPlots.quantizedFIR = fvtool(finalFilt,'Fs',hbParams.FsOut,'arithmetic','fixed');
DDCTestUtils.setPlotNameAndTitle('Quantized Final Filter');
```



Redefine the `ddcFilterChain` cascade object to include the fixed-point properties of the individual filters. Then use `fvtool` to analyze the entire filter chain and confirm that the quantized DDC still meets the specification.

```
ddcFilterChain = dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.quantizedDDCResponse = fvtool(ddcFilterChain,'Fs',FsIn,'Arithmetic','fixed');
DDCTestUtils.setPlotNameAndTitle('Quantized DDC Filter Chain');
legend(...  
    'DDC filter chain: Quantized', ...  
    'DDC filter chain: Reference');
```



### HDL-Optimized Simulink Model

The next step in the design flow is to implement the DDC in Simulink using HDL Coder compatible blocks.

### Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the filter chain variables already defined. Next, define the Numerically Controlled Oscillator (NCO) parameters, and the input signal. These parameters are used to configure the NCO block.

Specify the desired frequency resolution. Calculate the number of accumulator bits required to achieve the desired resolution, and define the number of quantized accumulator bits. The quantized output of the accumulator is used to address the sine lookup table inside the NCO. Also compute the phase increment needed to generate the specified carrier frequency. Phase dither is applied to those accumulator bits which are removed during quantization.

```
nco.Fd = 1;
nco.AccWL      = nextpow2(FsIn/nco.Fd) + 1;
nco.QuantAccWL = 12;
nco.PhaseInc   = round((-Fc * 2^nco.AccWL)/FsIn);
nco.NumDitherBits = nco.AccWL - nco.QuantAccWL;
```

The input to the DDC comes from `ddcIn`. For now, assign a dummy value for `ddcIn` so that the model can compute its data types. During testing, `ddcIn` provides input data to the model.

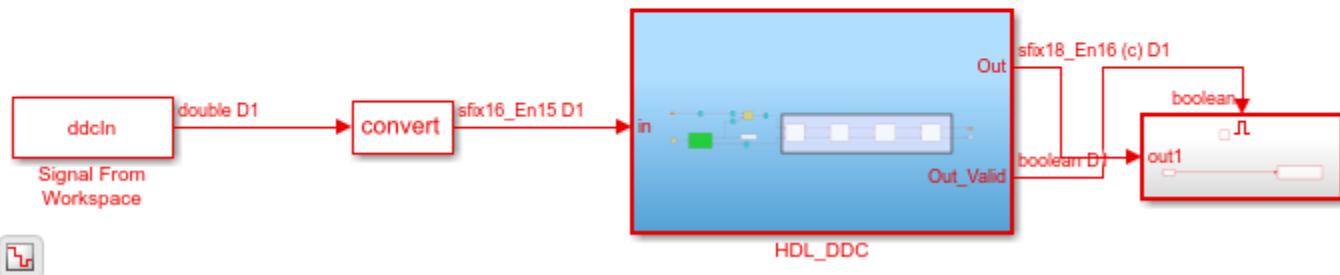
```
ddcIn = 0; %#ok<NASGU>
```

## Model Structure

The top level of the DDC Simulink model is shown. The model imports `ddcIn` from the MATLAB workspace using a **Signal From Workspace** block, converts it to 16-bits and then applies it to the DDC. HDL code can be generated from the **HDL\_DDC** subsystem.

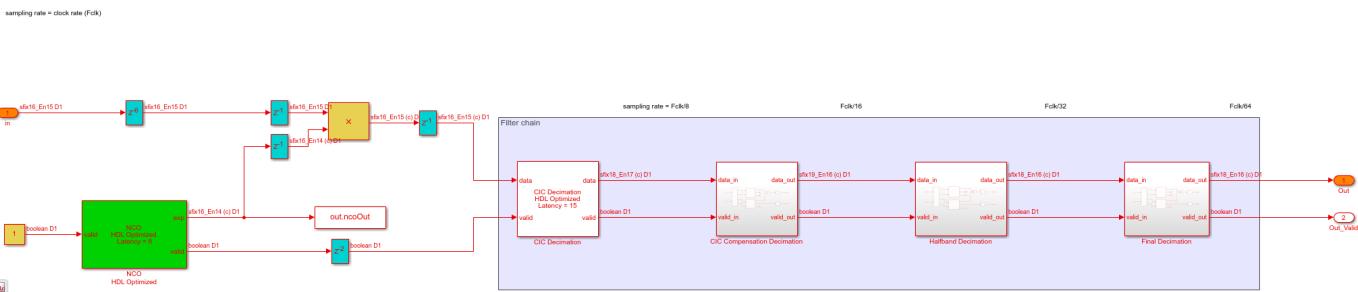
```
modelName = 'DDCHDLMImplementation';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
set_param(modelName, 'Open','on');
```

### Implementation of a Digital Down-Converter for LTE in HDL



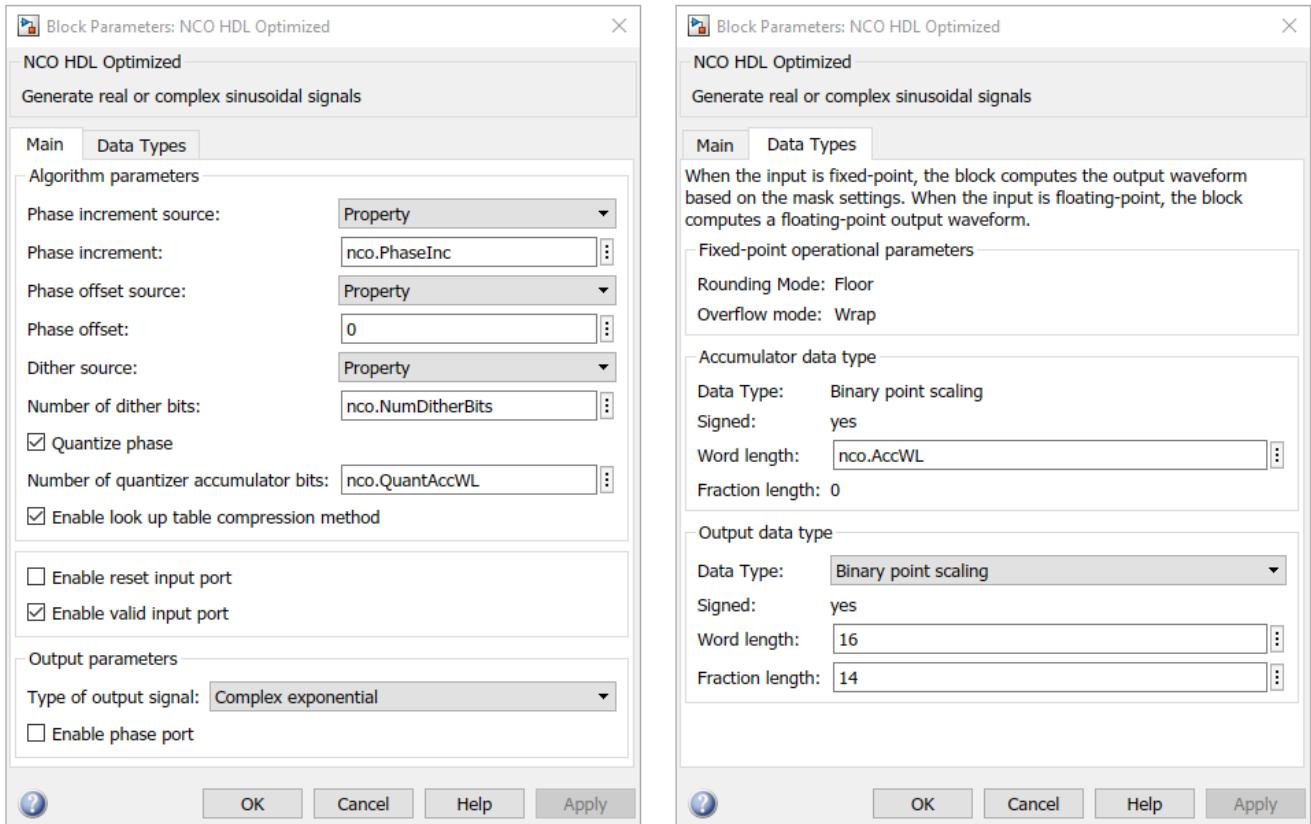
The DDC implementation is inside the **HDL\_DDC** subsystem. The **NCO HDL Optimized** block generates a complex phasor at the carrier frequency. This signal goes to a mixer which multiplies it with the input signal. The output of the mixer is then fed to the filter chain, where it is decimated to 1.92 Msps.

```
set_param([modelName '/HDL_DDC'], 'Open', 'on');
```



## NCO Block Parameters

The NCO block is configured with the parameters defined in the `nco` structure. Both tabs of the block's parameter dialog are shown.



## CIC Decimation and Gain Correction

The first filter stage is a Cascade Integrator-Comb (CIC) Decimator implemented with a CIC Decimation HDL Optimized block. The block parameters are set to the `cicParams` structure values. The gain correction is implemented by selecting the **Gain correction** parameter.

### Filter Block Parameters

The filters are configured by using the properties of the corresponding System objects. The CIC Compensation, Halfband Decimation, and Final Decimation filters operate at effective sample rates that are lower than the clock rate ( $F_{clk}$ ) by factors of 8, 16, and 32, respectively. These sample rates are implemented by using the **valid** signal to indicate which samples are valid at a particular rate. The signals in the filter chain all have the same Simulink sample time.

The CIC Compensation, Halfband Decimation, and Final Decimation filters are each implemented by a MATLAB Function Block and two Discrete FIR Filter HDL Optimized blocks in a polyphase decomposition. Polyphase decomposition implements the transform function

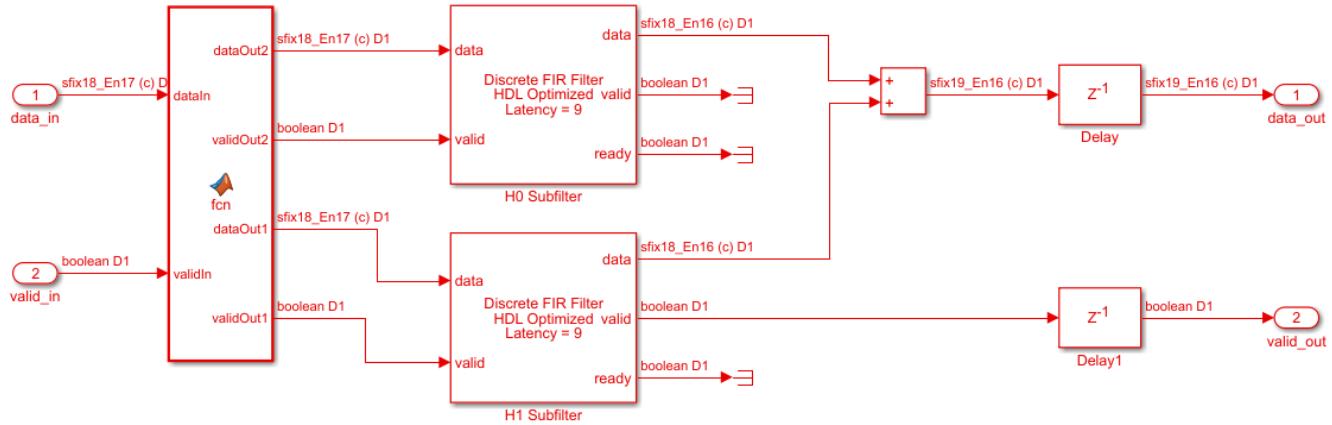
$$H(z) = H_0(z) + z^{-1}H_1(z), \text{ where } H(z) = a_0 + a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + \dots,$$

$H_0(z) = a_0 + a_2z^{-2} + \dots$  and  $H_1(z) = a_1 + a_3z^{-3} + \dots$ . Polyphase decomposition is a resource-efficient way to implement decimation filters. The MATLAB Function Block holds two input samples every two cycles and passes them at the same time to the parallel pair of Discrete FIR Filter HDL

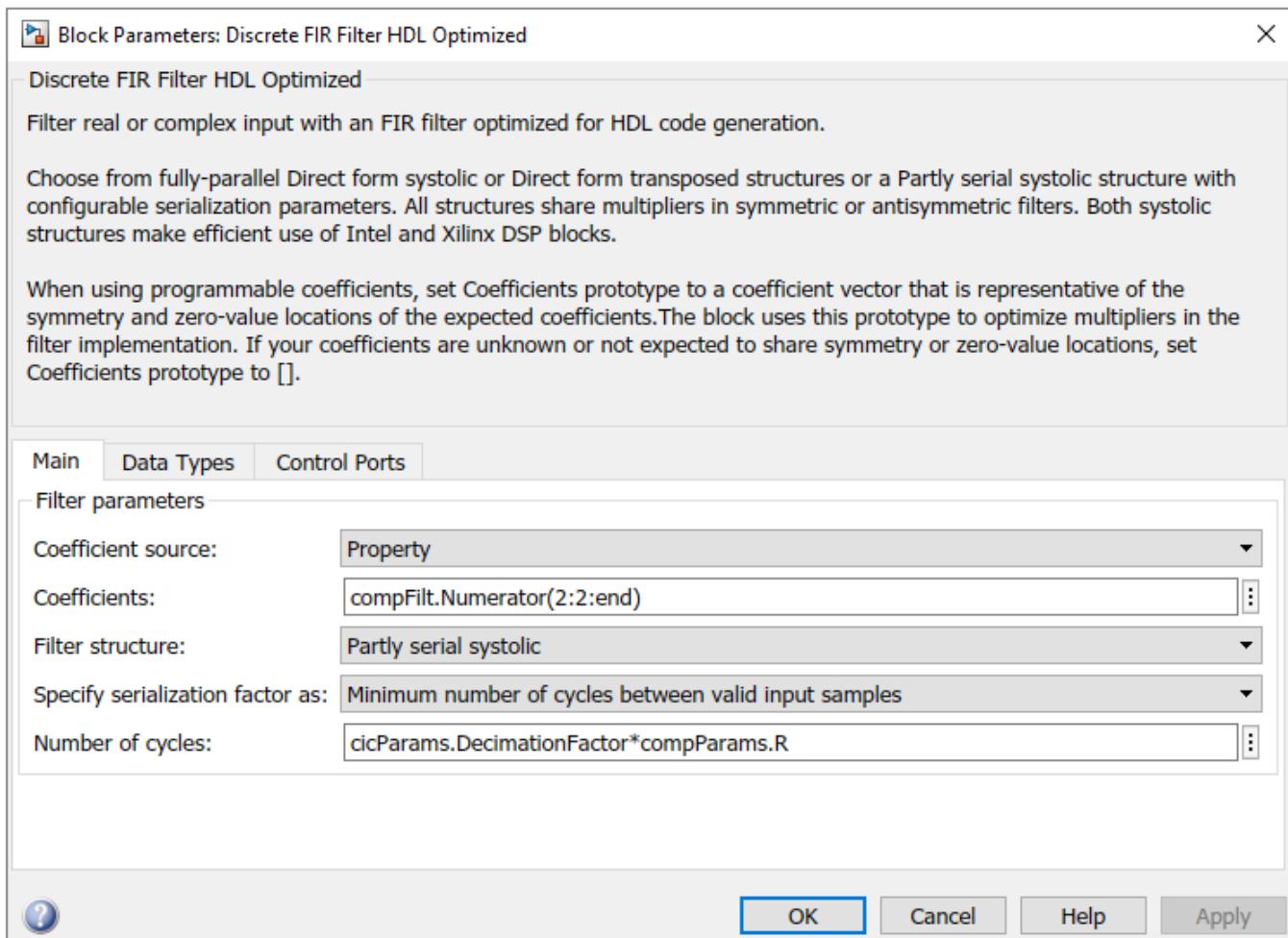
Optimized blocks  $H_0(z)$  and  $H_1(z)$ . The lower subfilter  $H_1(z)$  and the upper subfilter  $H_0(z)$  each contain half of the filter coefficients and process half of the input data.

For example, the CIC Compensation Decimation subsystem implements a 7-coefficient filter. The upper subfilter,  $H_0(z)$ , has 4 coefficients and the lower subfilter,  $H_1(z)$ , has 3 coefficients. Each filter receives a sample and generates an output every 16 cycles. Because each filter processes one sample every 16 cycles, the subfilter blocks can share hardware resources in time. To optimize hardware resources in this way, both of the subfilters have the **Filter structure** parameter set to **Partly serial systolic**.

The diagram shows the CIC Compensation Decimation subsystem. The Halfband Decimation and the Final Decimation subsystems use the same structure.



All the filter blocks are configured with the parameters defined in their corresponding structures. For example, the image shows the block parameters for the CIC Compensation Decimation block. The **Number of cycles** parameter is the minimum number of cycles between input samples. The input to the CIC Compensation Decimation block is sampled at `cicParams.DecimationFactor*compParams.R`, which is 16 cycles for both subfilters.



The serial filter implementation reuses the multipliers in time over the number of clock cycles you specify. Without this optimization, the CIC Compensation Decimation filter with complex input data would use 14 multipliers. After the optimization, each of  $H_0(z)$  and  $H_1(z)$  uses 2 multipliers for a total of 4. Similarly, the Halfband Decimation and Final Decimation subsystems use 4 multipliers each.

### Sinusoid on Carrier Test and Verification

To test the DDC, modulate a 40kHz sinusoid onto the carrier frequency and pass it through the DDC. Then measure the Spurious Free Dynamic Range (SFDR) of the resulting tone and the SFDR of the NCO output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40kHz test tone, modulated onto the carrier.
ddcIn = DDCTestUtils.GenerateTestTone(40e3,Fc);

% Demodulate the test signal with the floating point DDC.
ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
release(ddcFilterChain);
```

```
% Demodulate the test signal by executing the modified Simulink model with the sim function.
out = sim(modelName);

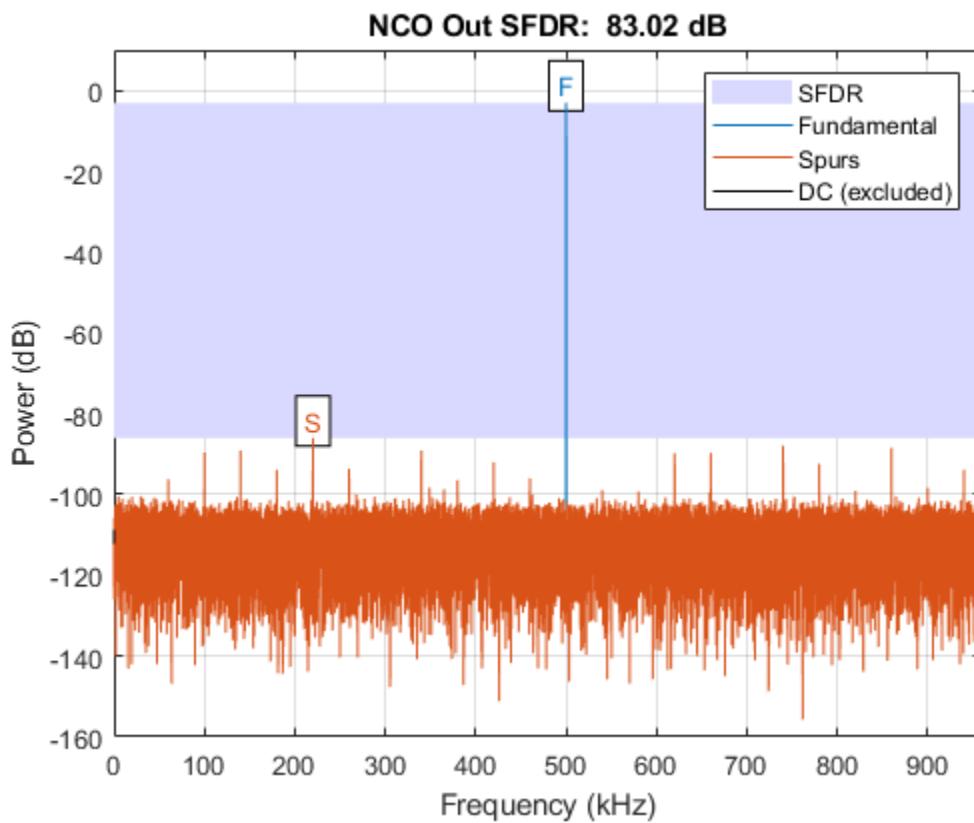
% Measure the SFDR of the NCO, floating point DDC and the fixed-point DDC outputs.
results.sfdrNCO      = sfdr(real(out.ncoOut),FsIn/64);
results.sfdrFloatDDC = sfdr(real(ddcOut),FsIn/64);
results.sfdrFixedDDC = sfdr(real(out.ddcOut),FsIn/64);

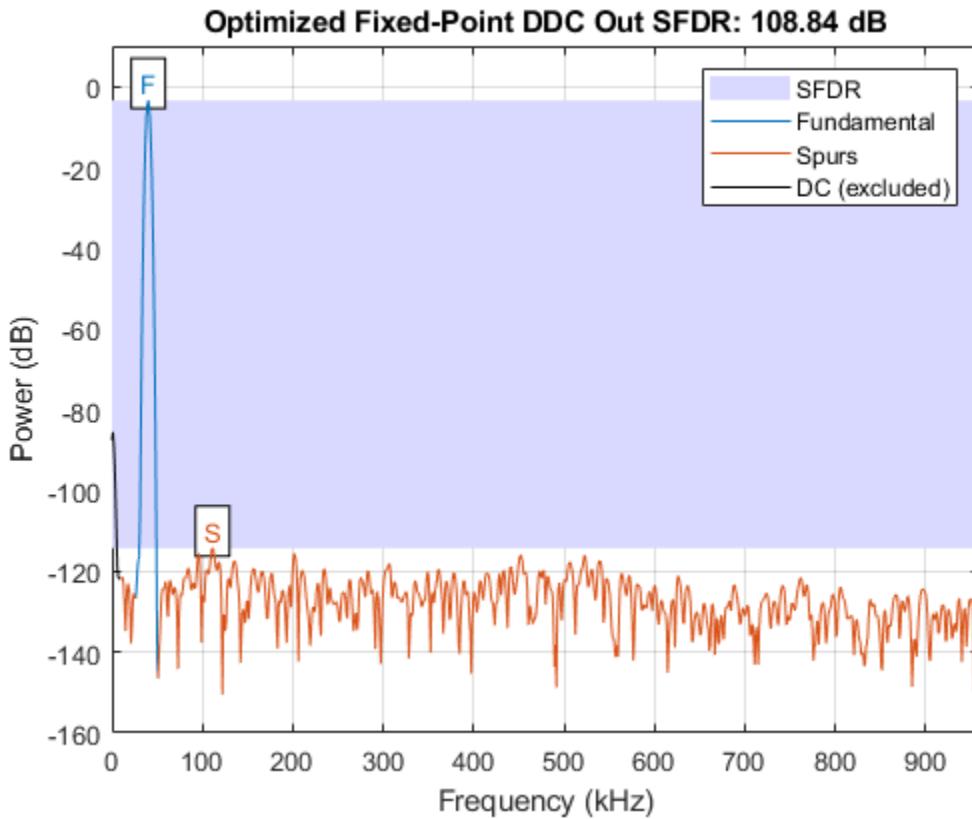
disp('Spurious Free Dynamic Range (SFDR) Measurements');
disp(['    Floating point DDC SFDR: ',num2str(results.sfdrFloatDDC) ' dB']);
disp(['    Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['    Optimized Fixed-point DDC SFDR: ',num2str(results.sfdrFixedDDC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DDC outputs.
ddcPlots.ncoOutSFDR = figure;
sfdr(real(out.ncoOut),FsIn/64);
DDCTestUtils.setPlotNameAndTitle(['NCO Out ' get(gca,'Title').String]);

ddcPlots.OptddcOutSFDR = figure;
sfdr(real(out.ddcOut),FsIn/64);
DDCTestUtils.setPlotNameAndTitle(['Optimized Fixed-Point DDC Out ' get(gca,'Title').String]);

Spurious Free Dynamic Range (SFDR) Measurements
    Floating point DDC SFDR: 291.3483 dB
    Fixed-point NCO SFDR: 83.0249 dB
    Optimized Fixed-point DDC SFDR: 108.8419 dB
```





### LTE signal test

```

rng(0);

if license('test','LTE_Toolbox')

    % Generate a modulated LTE test signal with LTE Toolbox
    [ddcIn, sigInfo] = DDCTestUtils.GenerateLTETestSignal(Fc);

    % Downconvert with a MATLAB Floating Point Model
    ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
    release(ddcFilterChain);

    % Downconvert using Simulink model
    ddcIn=[ddcIn;zeros(320,1)]; % Adding zeros to make up propagation latency to output complete
    out = sim(modelName);

    results.evmFloat = DDCTestUtils.MeasureEVM(sigInfo,ddcOut);
    results.evmFixed = DDCTestUtils.MeasureEVM(sigInfo,out.ddcOut);

    disp('LTE Error Vector Magnitude (EVM) Measurements');
    disp(['    Floating point DDC RMS EVM: ' num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['    Floating point DDC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['    Fixed-point HDL Optimized DDC RMS EVM: ' num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['    Fixed-point HDL Optimized DDC Peak EVM: ' num2str(results.evmFixed.Peak*100,3) '%']);

```

```

fprintf(newline);

end

LTE Error Vector Magnitude (EVM) Measurements
Floating point DDC RMS EVM: 0.633%
Floating point DDC Peak EVM: 2.44%
Fixed-point HDL Optimized DDC RMS EVM: 0.731%
Fixed-point HDL Optimized DDC Peak EVM: 2.69%

```

### HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL testbench for the **HDL\_DDC** subsystem. The DDC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The post place-and-route resource utilization results are shown in the table. The design met timing with a clock frequency of 313 MHz.

```

T = table(...  

    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}), ...  

    categorical({'2660'; '318'; '5951'; '1.0'; '18'}), ...  

    'VariableNames',{'Resource','Usage'})

```

```
T =
```

Resource	Usage
LUT	2660
LUTRAM	318
FF	5951
BRAM	1.0
DSP	18

## HDL Optimized QPSK Transmitter

This example shows how to optimize the QPSK transmitter modeled in the QPSK Transmitter and Receiver example for HDL code generation and hardware implementation.

### Introduction

The HDL Optimized QPSK Transmitter example shows how Simulink blocks that support HDL code generation can be used to implement the baseband processing of a digital communications transmitter.

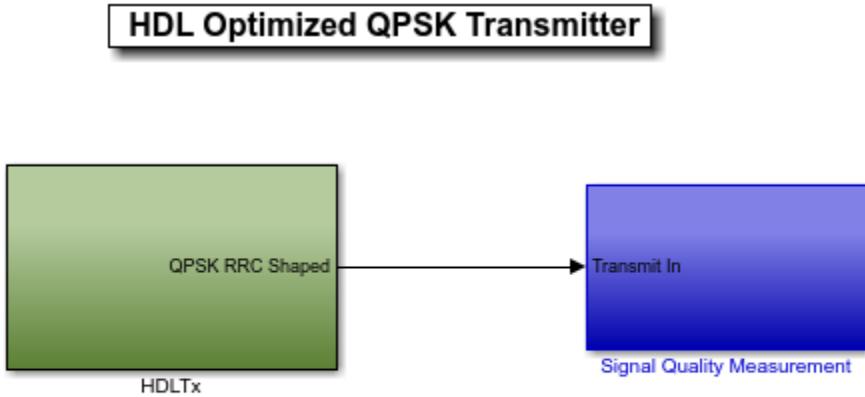
The HDLTx subsystem can be cascaded with the HDLRx subsystem given in the HDL Optimized QPSK Receiver example, in order to form a complete QPSK transmitter-receiver chain from which HDL code can be generated.

At the top level of hierarchy, there are two components: HDLTx and Signal Quality Measurement. The HDLTx subsystem has been optimized for HDL code generation by using hardware friendly blocks and design practices.

The HDLTx subsystem generates the In-phase (I) and Quadrature (Q) channels representing a QPSK (Quaternary-Phase-Shift-Keying) symbol mapping. This is sent in as a complex valued vector to the Root Raised Cosine (RRC) transmit pulse shaping filter. The output of the HDLTx subsystem therefore consists of the pulse shaped and upsampled symbols.

The Signal Quality Measurement subsystem is intended for software based validation of the design and is therefore not supported for HDL code generation.

The Signal Quality Measurement component consists of the matching RRC receive filter and a Constellation Diagram scope. The purpose of the scope is twofold. It both displays the output data, and calculates the Error Vector Magnitude (EVM) and Modulation Error Ratio (MER) values. This allows the designer to investigate the noise introduced into the system from the RRC filter design. This will arise from two sources - (a) the quantization of the coefficient values in the RRC filter and (b) the truncated impulse response of the RRC filter (roll-off factor 0.5). The EVM can therefore be improved by either increasing the number of fractional bits for the coefficient values or by redesigning the RRC filter to give an improved response.



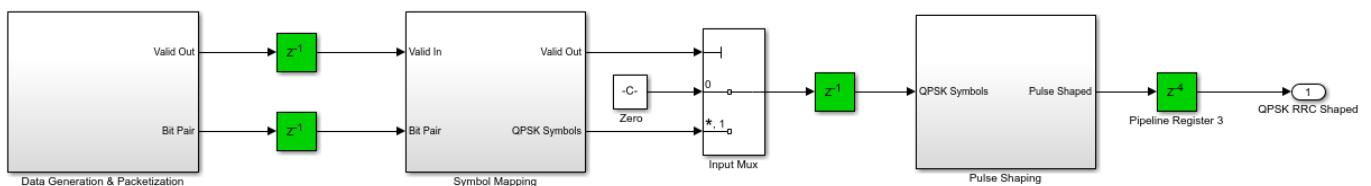
Copyright 2014 The MathWorks, Inc.

## Overview of HDL Optimized QPSK Tx

The HDLTx component is composed of three subsystems -

- Data Generation & Packetization
- Symbol Mapping
- Pulse Shaping

Pipeline registers are inserted between the subsystems in order to minimize the combinatorial path delay between components and therefore maximize the achievable clock frequency (i.e. minimizing the critical path delay). The Data Generation & Packetization subsystem contains several sub-components, which are also pipelined. There is, therefore, a fixed latency before valid data appears at the input to the Pulse Shaping component. To ensure that only valid data is transmitted, valid out and valid in ports are included which are used to control a 2:1 Multiplexer (Input Mux).

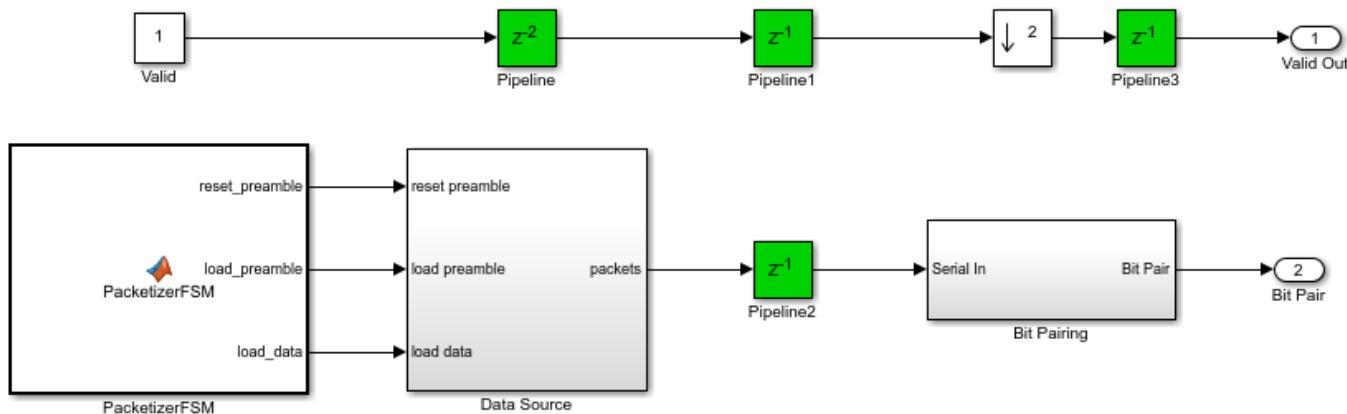


The Data Generation & Packetization, Symbol Mapping, and Pulse Shaping subsystems are further described in the following sections.

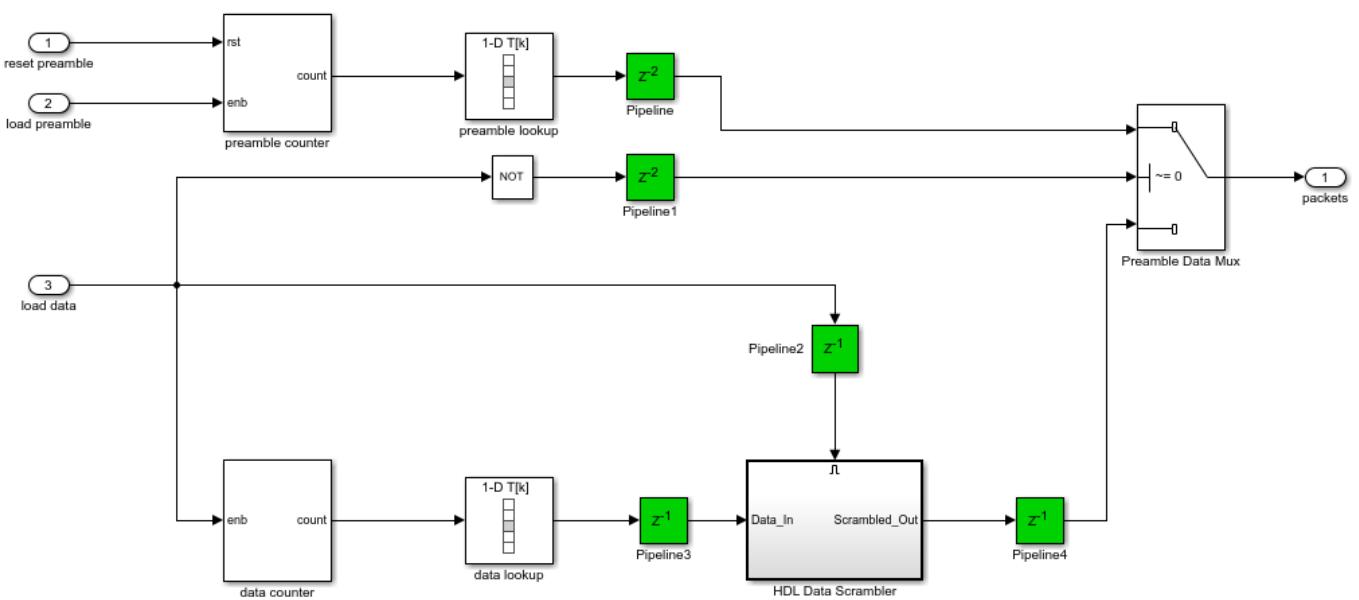
### 1 - Data Generation & Packetization

The following figure illustrates the top level hierarchy of the component. The Packetizer FSM and Data Source generate both the preamble and data bits, and perform scrambling and packetization. The packets consist of a 26 bit Barker code preamble and 174 bits of scrambled data. The Barker code preamble is used in the HDL Optimized QPSK Receiver Example for autocorrelation based packet detection via the use of a matched filter. The Bit Pairing subsystem converts the single bit

serial data input into a two bit output of half the sampling rate, providing the correct data format for the symbol mapper. Internally, the data source block has a pipeline delay of 2 samples. In addition, there is a pipeline delay between the data source and the bit pairing subsystem. The valid signal is therefore delayed to match the pipeline delay of the data path. The Bit Pairing subsystem reduces the sample rate by a factor of 2. Placing a downsample by two in the valid control path ensures that the sample rate of the valid control path matches that of the signal path.

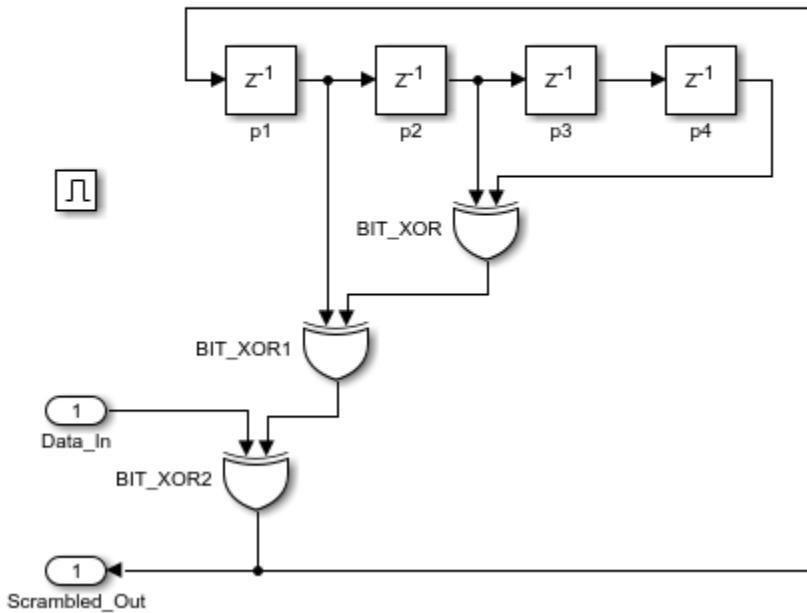


- **Packetizer FSM** - The Packetizer FSM component consists of a Moore machine, created using a MATLAB™ function block. The FSM has two states - Pack\_Preamble, and Append\_Data. The Pack\_Preamble state serves to enable the preamble counter via the load\_preamble output. The preamble counter generates the address signal for the preamble Lookup Table (LUT), incrementing from 0 → 25. Following this, the FSM moves into the Append\_Data state, asserting the load\_data signal, where it shall remain for 174 cycles. During this time, the reset\_preamble output is held high, while the load\_preamble output is held low. The FSM shall sequentially switch between the two states in turn.
- **Data Source** - As shown in the following figure, the Data Source component contains two LUTs, containing the preamble and data bits. The preamble bits are invariant between packets, and so the counter used to address the preamble LUT is reset after the loading of each preamble. The counter used to address the data LUT is allowed to wrap around after reaching the maximum address value, which corresponds to the final character of the packet body string "Hello World 099". Each of the LUTs is zero padded in order to ensure that the number of data entries is of a power of two length. In addition to enabling the counter for the data LUT, the load data input is used to control when the data scrambler component should be enabled, and to control whether the preamble or data bits should be passed to the output via the 2:1 Preamble Data Mux.

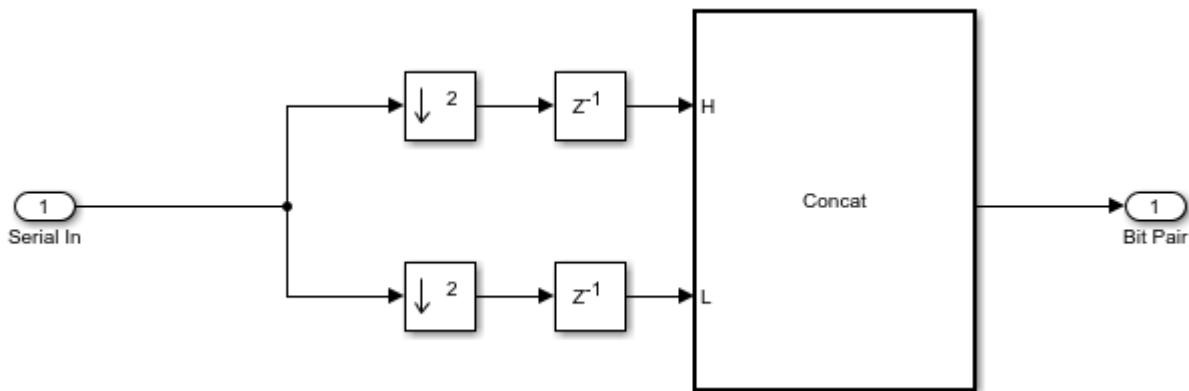


- **Data Scrambler** - The data scrambler is detailed in the following figure. The scrambler is built from first principles using XOR gates (for modulo 2 addition) and registers. By using an enabled subsystem, it is ensured that the scrambler is only enabled when there is new input data to be processed.

HDL optimized scrambler, implementing the scrambler polynomial [1 1 1 0 1], with initial states [0 0 0 0]

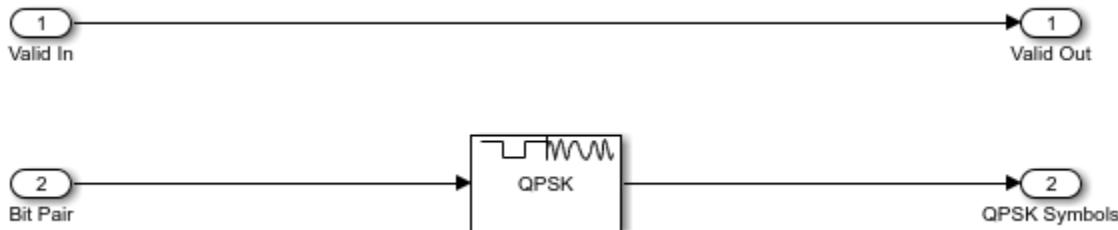


- **Bit Pairing**- The purpose of the Bit Pairing subsystem is to group pairs of bits into an unsigned two bit output - the input format expected by the symbol mapping component. To achieve this, a pair of downsamplers are used to reduce the sample rate by a factor of two, with the first downampler selecting the second phase and the second downampler selecting the first phase. The Bit Concat block concatenates the single bit inputs into an unsigned two bit output.



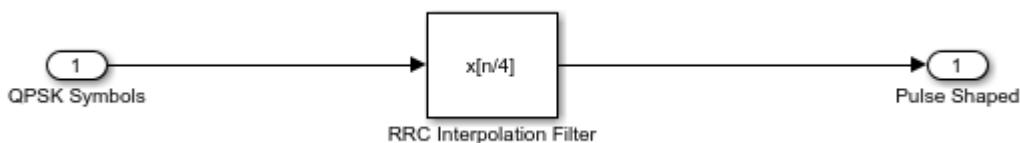
## 2 - Symbol Mapper

The Symbol Mapper uses the QPSK Modulator Baseband block from the Communications Toolbox™ in order to map the integer input value [0,1,2,3] onto the appropriate complex valued symbol [1 + 1i, -1 + 1i, 1 - 1i, -1 - 1i]. Mapping the symbols in this manner allows for a 2 bit value to be used to represent each symbol. This allows the wordlengths in the RRC transmit filter for the product and accumulator datapaths to be optimized. The block uses a Gray Mapping scheme.



## 3 - Pulse Shaping

The Pulse Shaping component uses an FIR Interpolation filter, featuring an upsampling factor of four, and a Root Raised Cosine impulse response. The receive filter in the HDL optimized QPSK Receiver demo forms a matched filter to this transmit filter. The filter is pipelined (see HDL Block Properties) in order to ensure that the combinatorial delay is minimized throughout the design.



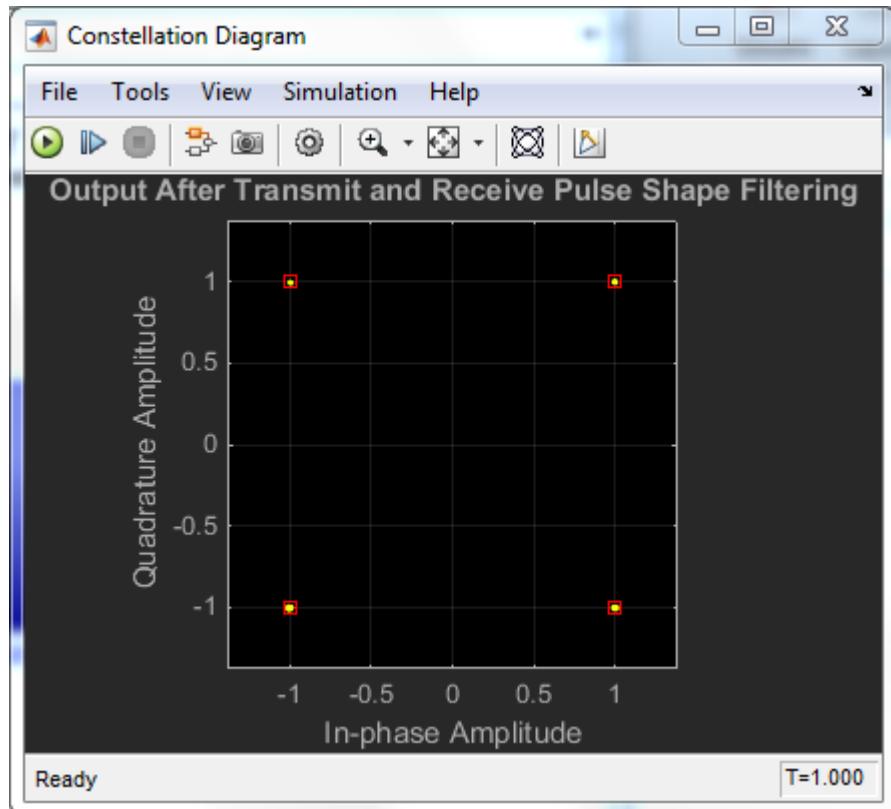
## Results and Displays

During simulation, the constellation obtained after passing the signal through the transmit and receive filter chain is displayed via the Constellation Diagram Scope (contained within the Signal Quality Measurement component). In addition, the scope displays the calculated EVM and MER via the **Tools --> Measurements --> Signal Quality** option.

The EVM is the calculated magnitude of the error vector between the ideal constellation point (given by the immediate symbol mapper output) and the constellation point obtained after transmit and receive pulse shape filtering has been performed. The EVM gives a figure of merit regarding the distortion introduced by the transmit and receive filtering. In this case, the EVM allows the system designer to determine if the distortion introduced by the pulse shaping filters due to the effects of fixed point quantization and non-ideal (truncated) impulse responses is acceptable or otherwise. For further information, please refer to the EVM Measurement block documentation.

The MER is a measure of the signal-to-noise ratio in digital modulation applications. It is the ratio of the average reference signal power to the mean square error. For further information, please refer to the MER Measurement block documentation.

Shown below is a snapshot from the Constellation Diagram scope output.



## HDL Code Generation

The HDL code generated from the **HDLTx** subsystem has been synthesized using Xilinx® ISE on a Virtex6 (XC6VLX240T) FPGA, and ran at around 300 MHz. When operating the RRC transmit filter at maximum rate, the QPSK input symbols are generated at 75M symbols/second. The resulting

theoretical maximum data rate is therefore 150M samples/second, taking into account the data transmitted over both the I and the Q channels.

You can use the commands **makehdl** and **makehdltb** to generate HDL code and testbench for subsystems in **HDLTx**. To generate the HDL code, use the following command:

```
makehdl(subsystemname)
```

To generate a testbench, use the following command:

```
makehdltb(subsystemname)
```

When generating the testbench, a validation model will be created which includes the internal pipeline delay of the RRC transmit filter. The second constellation diagram which is invoked illustrates the output of the validation model.

# HDL Optimized QPSK Receiver with Captured Data

This example shows how to optimize the QPSK receiver modeled in QPSK Transmitter and Receiver example for HDL code generation and hardware implementation. The HDL-optimized model shows a QPSK receiver that addresses real-world communications issues like carrier frequency, phase offset, and timing recovery in a hardware-friendly manner.

## Overview

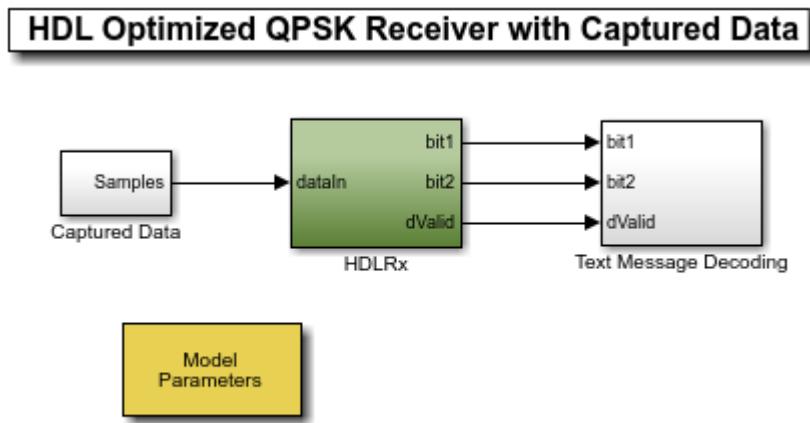
The HDL Optimized QPSK Receiver with Captured Data example provides a hardware-friendly solution that performs baseband processing to handle a time-varying frequency offset and a time-varying symbol delay. Specifically, this example provides an HDL-optimized reference design of a practical digital receiver to mitigate the above-mentioned impairments, and includes coarse frequency compensation, PLL-based fine frequency compensation, timing recovery with fixed-rate resampling, bit stuffing/skipping, frame synchronization, and phase ambiguity resolution.

Compared with the implementation of the receiver in the QPSK Transmitter and Receiver example, three major modifications have been made for efficient HDL code generation:

- **Streaming Input and Output:** The HDL optimized QPSK receiver processes data one sample at a time. The captured real-world signal is streamed into the receiver front-end. The streaming output of the HDL optimized receiver is buffered and passed to the text message decoder.
- **Fixed-point:** The QPSK receiver logic operates in fixed-point mode.
- **HDL optimized architecture:** Several blocks have been redesigned to use hardware efficient algorithms and architectures.

## Structure of the Example

The top-level structure of the QPSK receiver model is shown in the following figure. The **HDLRx** subsystem has been optimized for HDL code generation.

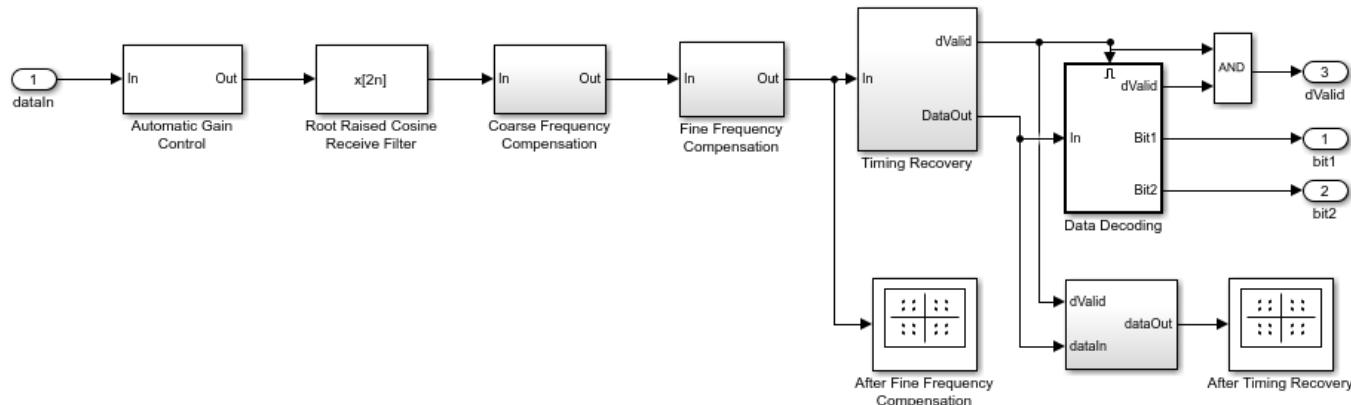


Copyright 2014-15 The MathWorks, Inc.

The input data is captured using two USRP® devices and the Communications Toolbox Support Package for USRP® Radio. Specifically, one USRP® device runs the QPSK Transmitter with USRP® Hardware example model and acts as a transmitter, while the other device is the receiver running the

companion model QPSK Receiver with USRP® Hardware example. The captured data represents the baseband received signal with a sampling rate of 200 KHz. The data is sample-based and has a length of 200001, which corresponds to a period of 1 s.

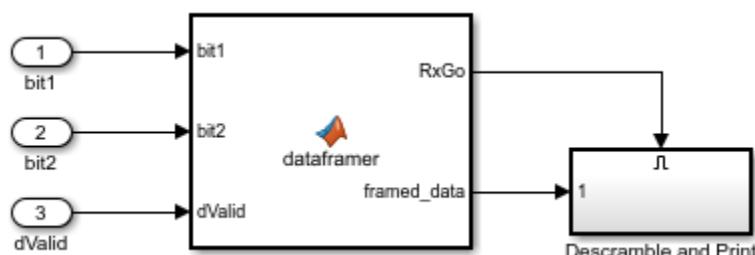
The following diagram shows the detailed structure of the **HDLRx** subsystem.



The subsystems within are further described in the following sections.

1. **Automatic Gain Control (AGC)** - Adjusts the received signal amplitude to a desired level
2. **Root Raised Cosine Receive Filter** - Uses a rolloff factor of 0.5, and downsamples the input signal by two
3. **Coarse Frequency Compensation** - Estimates an approximate frequency offset of the received signal and corrects it
4. **Fine Frequency Compensation** - Compensates for the residual frequency offset and the phase offset
5. **Timing Recovery** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants
6. **Data Decoding** - Aligns the frame boundaries, resolves the phase ambiguity caused by the Fine Frequency Compensation subsystem, and demodulates the signal

The structure of the **Text Message Decoding** subsystem is shown below.



This subsystem is expected to be run in software, therefore, it is preferable to employ frame-based signals to speed up the computation. The **HDLRx** subsystem outputs three sample-based Boolean

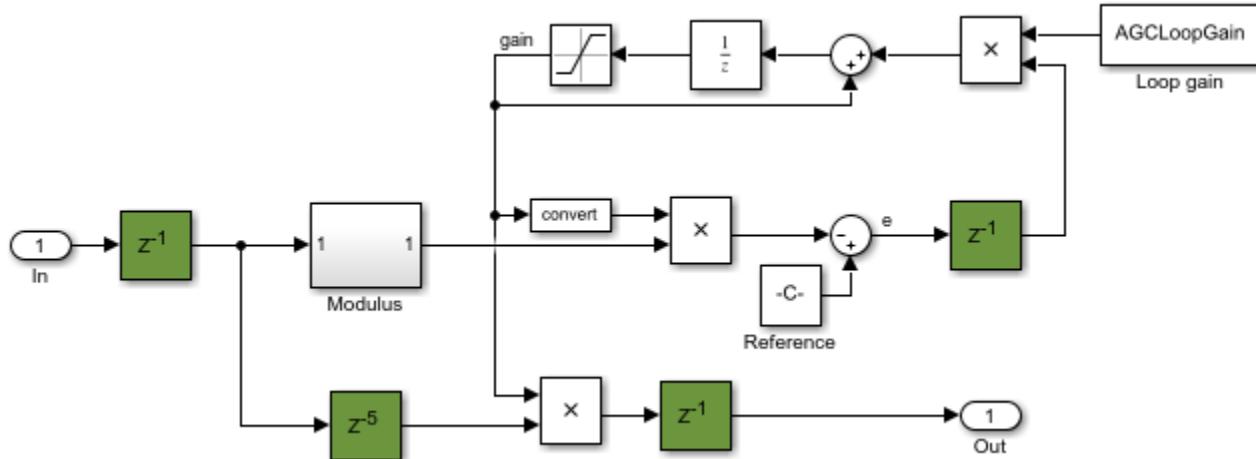
signals: bit1, bit2, and dValid. Given that the downstream processing requires a frame signal, the task of converting sample-based signals to frame-based counterparts is accomplished by the **dataframe** block. The demodulated bit pair, bit1 and bit2, is valid only when dValid is set high. The **dataframe** block uses the dValid signal to properly fill up a delay line with bit1 and bit2. The **Descramble and Print** subsystem processes the received data only when its enable signal goes high. This occurs when both the delay line accumulates exactly 200 valid demodulated bits and the RxGo signal is sent high. While the simulation is running, the **Descramble and Print** subsystem outputs the string "Hello world ##### to the MATLAB® command window, where ##### is a repeating sequence of '000', '001', '002', ..., '099'.

## HDL Optimized QPSK Receiver

### 1. AGC

The phase error detector gain  $K_p$  of the phase and timing error detectors is proportional to the received signal amplitude and the average symbol energy. To ensure an optimum loop design, the signal amplitude at the inputs of the carrier recovery and timing recovery loops must be stable. The **AGC** ensures that the amplitude of the input of the **Coarse Frequency Compensation** subsystem is  $1/\text{Upsampling Factor}$ , so that the equivalent gains of the phase and timing error detectors stay constant over time. The **AGC** is placed before the **Root Raised Cosine Receive Filter** so that the signal amplitude can be measured with an oversampling factor of four, thus improving the accuracy of the estimate. Refer to Chapter 7.2.2 and Chapter 8.4.1 of [ 1 ] for details on how to design the phase detector gain  $K_p$ .

The AGC structure is shown in the following diagram, and pipeline registers are shown in green throughout the model.



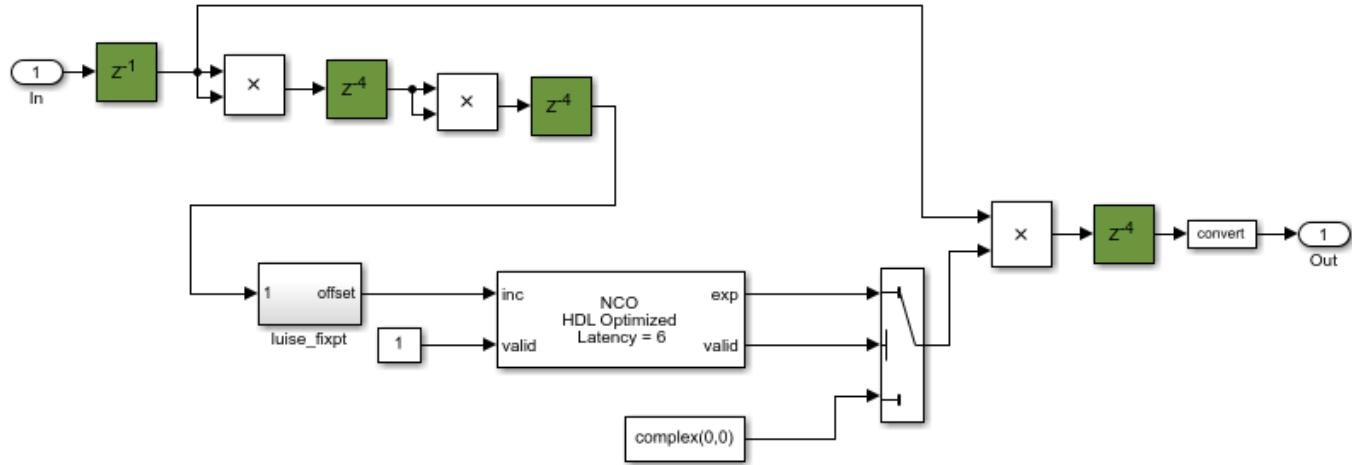
### 2. Root Raised Cosine Receive Filter

The **Root Raised Cosine Receive Filter** downsamples the input signal by a factor of two, with a rolloff factor of 0.5. It provides matched filtering for the transmitted waveform to boost the signal-to-noise ratio and facilitate the downstream signal processing.

The **Root Raised Cosine Receive Filter** is implemented using a fully parallel architecture.

### 3. Coarse Frequency Compensation

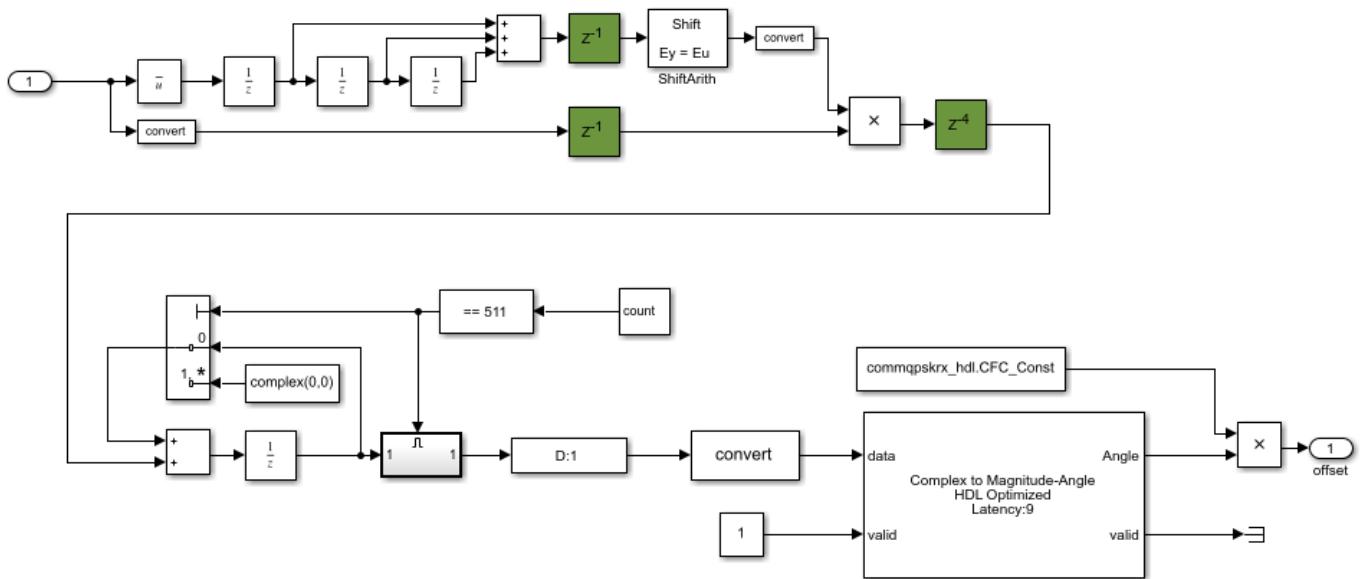
The **Coarse Frequency Compensation** subsystem corrects the input signal with a rough estimate of the frequency offset. The following diagram shows the **Coarse Frequency Compensation** subsystem.



This subsystem uses a baseband QPSK signal with a designated phase index  $n$ , frequency offset  $\Delta f$  and phase offset  $\Delta\phi$  expressed as  $e^{(j(n\pi/2+\Delta ft+\Delta\phi))}$ ,  $n = 0, 1, 2, 3$ . First, the subsystem raises the input signal to the power of four to obtain  $e^{(j(4\Delta ft+4\Delta\phi))}$ , which is not a function of the QPSK modulation. This is implemented by cascading two product blocks. Then, from the modulation-independent signal, it estimates the tone at four times the frequency offset. After dividing the estimate by four, the so-obtained frequency offset is corrected in the original signal. There is usually a residual frequency offset even after the **coarse frequency compensation**, which would cause a slow rotation of the constellation. The **Fine Frequency Compensation** subsystem compensates for this residual frequency.

Comparing the implementation of the **Coarse Frequency Compensation** subsystem here with those in QPSK Transmitter and Receiver examples and QPSK Receiver with USRP® Hardware example, we can see several modifications:

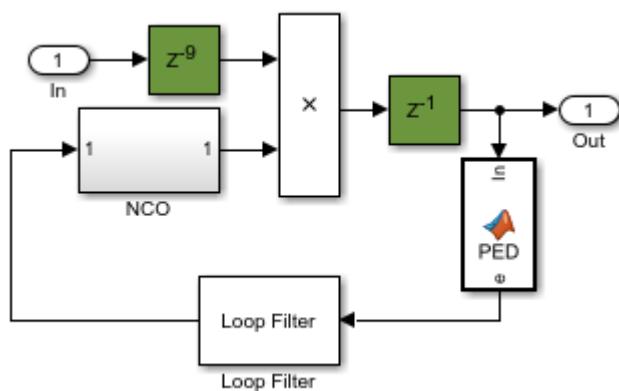
- To save resources, the FFT algorithm has been replaced by the frequency estimation algorithm proposed in [ 2 ], which is referred to as the Luise algorithm. Pipeline registers have been used in the data path of the Luise algorithm to break the critical path in the design. See diagram below.



- The *angle* function, which constitutes a key component in the Luise algorithm, is computed using the Complex to Magnitude-Angle HDL Optimized block. This block computes the phase using the hardware friendly CORDIC algorithm. To learn more about the Complex to Magnitude-Angle HDL Optimized block, refer to the DSP System Toolbox documentation.
- The detected phase offset is sent to an NCO to generate a complex exponential signal that is used to correct the phase offset in the original signal. The NCO HDL Optimized block maps the lookup table into a ROM, and provides a lookup table compression option to significantly reduce the lookup table size. To learn more about the NCO HDL Optimized block, refer to the DSP System Toolbox documentation.

#### 4. Fine Frequency Compensation

The Fine Frequency Compensation subsystem, shown in the following figure, implements a phase-locked loop (PLL), described in Chapter 7 of [ 1 ], to track the residual frequency offset and the phase offset in the input signal.

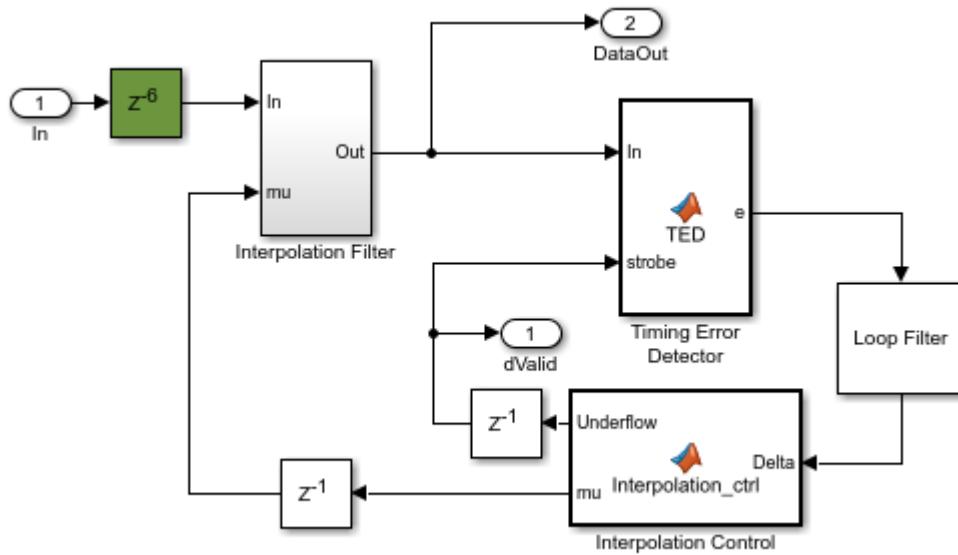


A maximum likelihood **Phase Error Detector (PED)**, described in Chapter 7.2.2 of [ 1 ], generates the phase error. A tunable proportional-plus-integral **Loop Filter**, described in Appendix C.2 of [ 1 ],

filters the error signal and then feeds it into the **NCO** block. The **NCO** block generates a complex exponential signal that is used to correct the residual frequency and phase offsets in the output of the **Coarse Frequency Compensation** subsystem. *Loop Bandwidth* (normalized by the sample rate) and *Loop Damping Factor* are tunable for the **Loop Filter**. The default normalized loop bandwidth is set to 0.06 and the default damping factor is set to 2.5 (over damping), so that the PLL quickly locks to the intended phase while introducing little phase noise.

## 5. Timing Recovery

The **Timing Recovery** subsystem is shown in the following diagram.



The **Timing Recovery** subsystem implements a PLL, described in Chapter 8 of [ 1 ], to correct the timing error in the received signal. On average, the **Timing Recovery** subsystem generates one output sample for every two input samples.

The **Interpolation Control** subsystem implements a decrementing modulo-1 counter, described in Chapter 8.4.3 of [ 1 ], to generate the control signal to facilitate the **Data Decoding** subsystem to properly select the interpolants of the **Interpolation Filter**. This control signal also enables the **Timing Error Detector (TED)**, so that it calculates the timing errors at the correct timing instants. The **Interpolation Control** subsystem updates the timing difference for the **Interpolation Filter**, generating interpolants at optimum sampling instants.

The **Interpolation Filter** is a Farrow parabolic filter with  $\alpha = 0.5$  as described in Chapter 8.4.2 of [ 1 ]. The filter uses an  $\alpha$  of 0.5 so that all the filter coefficients become 1, -1/2 and 3/2, which significantly simplifies the interpolator structure.

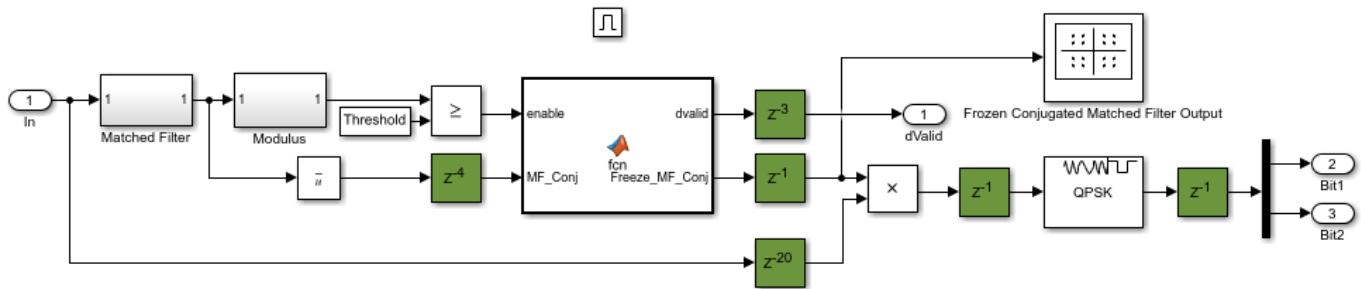
Based on the interpolants, timing errors are generated by a zero-crossing **Timing Error Detector** as described in Chapter 8.4.1 of [ 1 ], filtered by a tunable proportional-plus-integral **Loop Filter** as described in Appendix C.2 of [ 1 ], and fed into the **Interpolation Control** for a timing difference update. *Loop Bandwidth* (normalized by the sample rate) and *Loop Damping Factor* are tunable for the **Loop Filter**. The default normalized loop bandwidth is set to 0.01 and the default damping factor is set to unity (critical damping) so that the PLL quickly locks to the correct timing while introducing little phase noise.

When the timing error (delay) reaches symbol boundaries, there is one extra or missing interpolant in the output. The TED implements bit stuffing or skipping to handle the extra or missing interpolants. You can refer to Chapter 8.4.4 of [ 1 ] for details of bit stuffing/skipping.

The timing recovery loop normally generates one output symbol for every two input samples. It also outputs a timing strobe (dValid signal) that runs at the input sample rate. Under normal circumstances, the strobe value is simply a sequence of alternating ones and zeros. However, this occurs only when the relative delay between transmitter and receiver contains some fractional part of one symbol period and the integer part of the delay (in symbols) remains constant. If the integer part of the relative delay changes, the strobe value can have two consecutive zeros or two consecutive ones.

## 6. Data Decoding

The **Data Decoding** subsystem performs frame synchronization, phase ambiguity resolution, and QPSK demodulation. Its structure is shown in the diagram below:



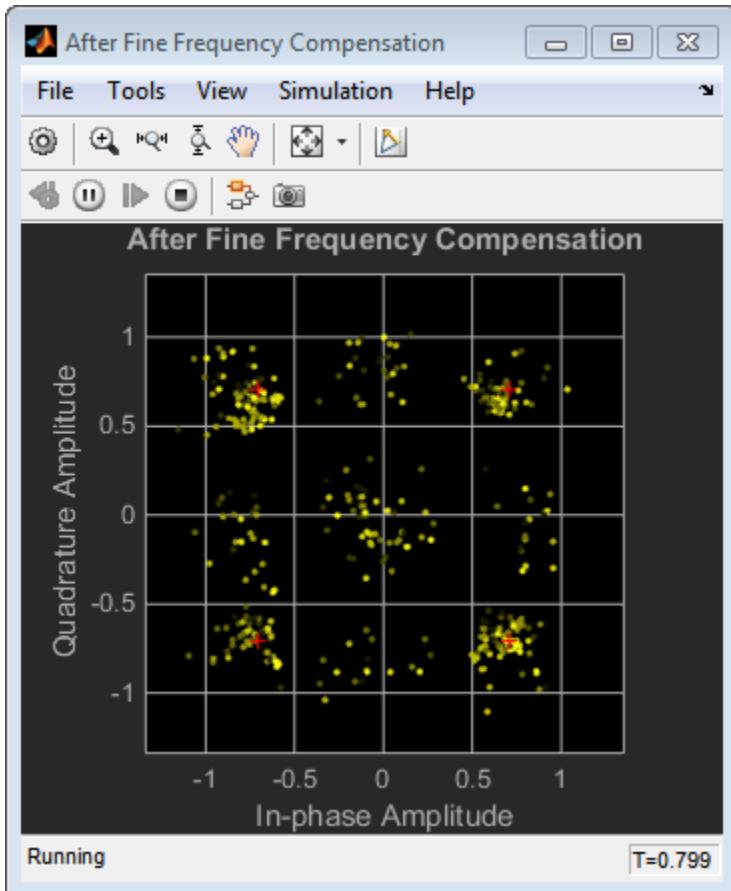
- **Frame synchronization:** The **Matched Filter** subsystem uses a QPSK-modulated Barker code as a reference to correlate against the received symbols. The modulus of the matched filter output is calculated in the **Modulus** subsystem and then compared with a threshold. Frame synchronization is declared if the modulus output exceeds the threshold. The threshold for frame synchronization is tunable: a large value increases the miss probability whereas a small value increases the probability of false alarm. In this example, the threshold value is set to 16.
- **Phase ambiguity resolution:** The carrier phase PLL of the **Fine Frequency Compensation** subsystem may lock to the unmodulated carrier with a phase shift of 0, 90, 180, or 270 degrees, which can cause a phase ambiguity. For details of phase ambiguity and its resolution, refer to Chapter 7.2.2 and 7.7 in [ 1 ]. The angle of the matched filter output determines the extra phase shift. The **Matched Filter** output is fed into the conjugate block to negate the extra phase shift. Once frame synchronization is achieved, the conjugated version of the matched filter output is frozen and multiplied with all the symbols in a frame to effectively resolve the phase ambiguity issue.
- **QPSK demodulation:** Each corrected symbol is demodulated and mapped to a pair of bits based on the symbol mapping of QPSK constellation.

## Results and Displays

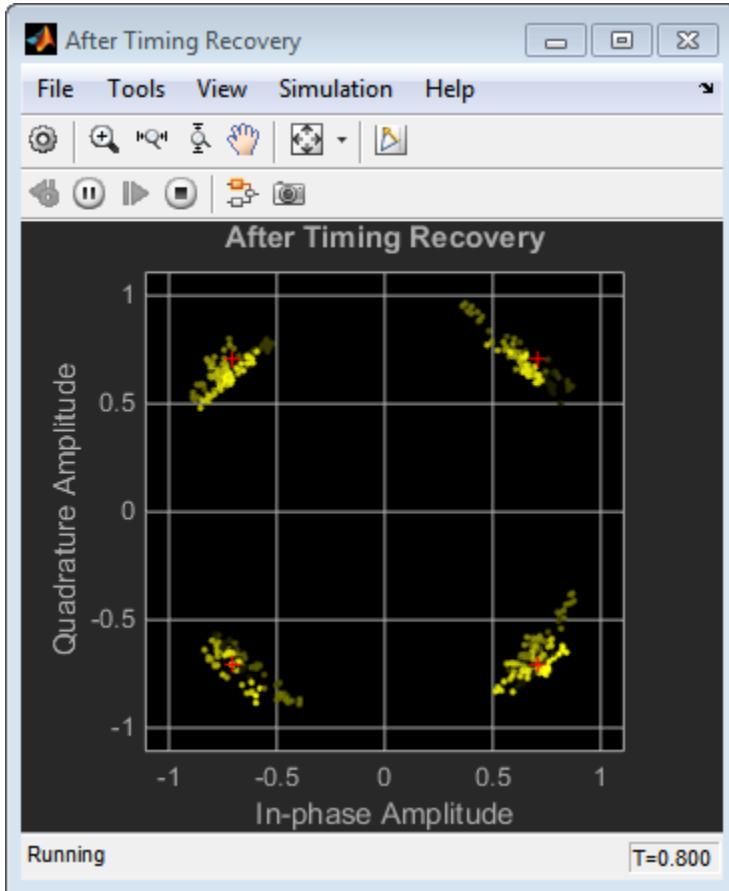
When running the simulation, the model displays three scatter plots to show the constellation of the **Fine Frequency Compensation** output, the **Timing Recovery** output, and the frozen conjugated matched filter output, respectively.

The following diagram shows the constellation plot of the **Fine Frequency Compensation** output. The cluster is scattered around, mainly due to two reasons:

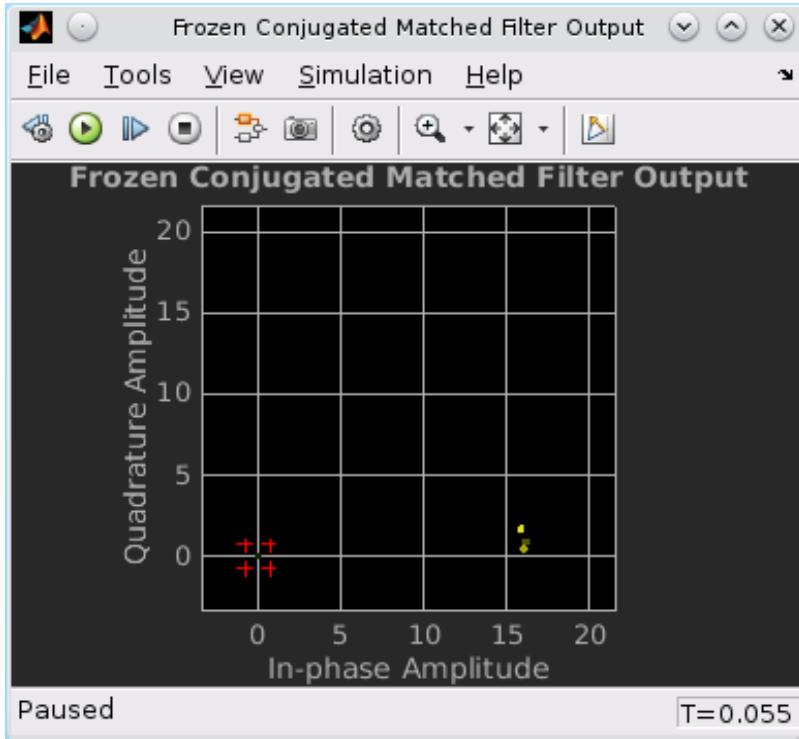
- The timing error between the clocks at the transmitter and receiver
- The signals are oversampled by a factor of two. Therefore, half of the symbols are in the transition state between QPSK symbols.



The following diagram shows the constellation of the **Timing Recovery** output. One observes four concentrated clusters around the true 4-point constellation for QPSK modulation. This verifies the effectiveness of the **Timing Recovery** subsystem. However, as mentioned before, the **Fine Frequency Compensation** subsystem may lock the signal with a phase shift of 0, 90, 180, or 270 degree. Therefore, we need to address the phase ambiguity issue before demodulating the signal.



The following figure shows the constellation plot of the frozen conjugated matched filter output. During the entire course of simulation, the signals group on the positive side of the horizontal axis. This grouping indicates that there are no phase ambiguity issues in this example. If compensation was required for an undesired phase shift in order to resolve the phase ambiguity issue, the constellation plots after **Timing Recovery** would be rotated.



### HDL Code Generation

Pipeline registers (shown in green) have been added throughout the model to make sure the **HDLRx** subsystem does not have a long critical path. The HDL code generated from the **HDLRx** subsystem was synthesized using Xilinx® ISE on a Virtex6 (xc6vlx75t) FPGA, and the circuit ran at about 145 MHz.

You can use the commands **makehdl** and **makehdltb** to generate HDL code and testbench for subsystems in **HDLRx**. To generate the HDL code, use the following command:

```
makehdl(subsysname)
```

To generate testbench, use the following command:

```
makehdltb(subsysname)
```

### References

1. Michael Rice, "Digital Communications - A Discrete-Time Approach", Prentice Hall, April 2008.
2. M. Luise and R. Reggiannini, "Carrier frequency recovery in all-digital modems for burst-mode transmissions," *IEEE Trans. Communications*, pp. 1169-1178, 1995.

### Copyright Notice

USRP® is a trademark of National Instruments Corp.

# HDL Optimized QAM Transmitter and Receiver

This example shows how to implement a 64-QAM transmitter and receiver for HDL code generation and hardware implementation.

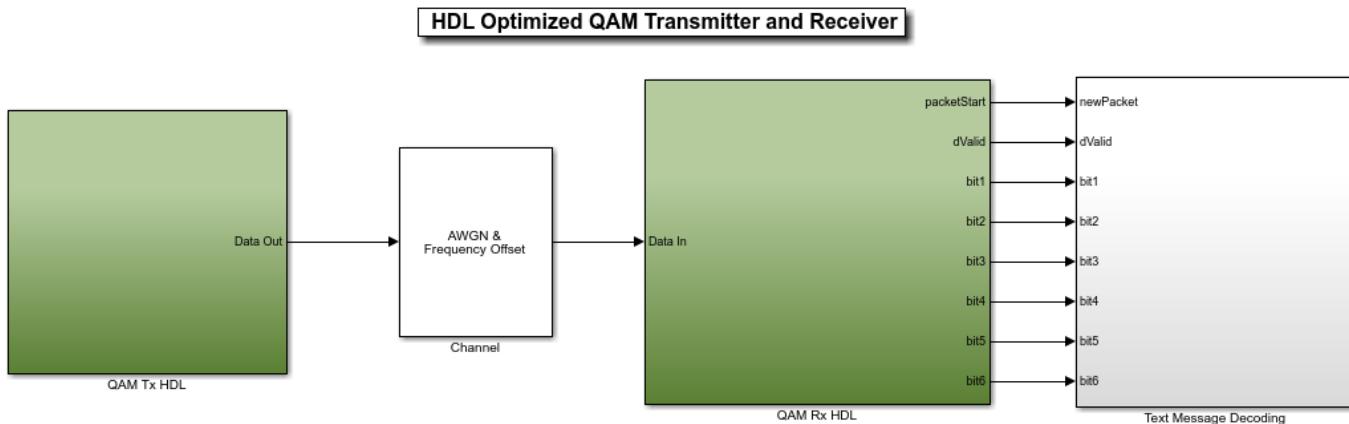
## Overview

The **HDL Optimized QAM Transmitter and Receiver** example shows how to use Simulink® blocks that support HDL code generation to implement the baseband processing of a digital communications transmitter and receiver.

The **HDL QAM Tx** subsystem generates a complex valued, 64-QAM modulated constellation. A floating point channel model, **Channel**, is used to add attenuation, channel noise, carrier frequency offset and fractional delay in order to demonstrate the operation of the receiver subsystem. The **HDL QAM Rx** subsystem implements a practical digital receiver to mitigate the channel impairments using coarse frequency recovery, timing recovery, frame synchronization and magnitude and phase recovery. The received data packets are then decoded and printed to the MATLAB® Command Window by the **Text Message Decoding** subsystem.

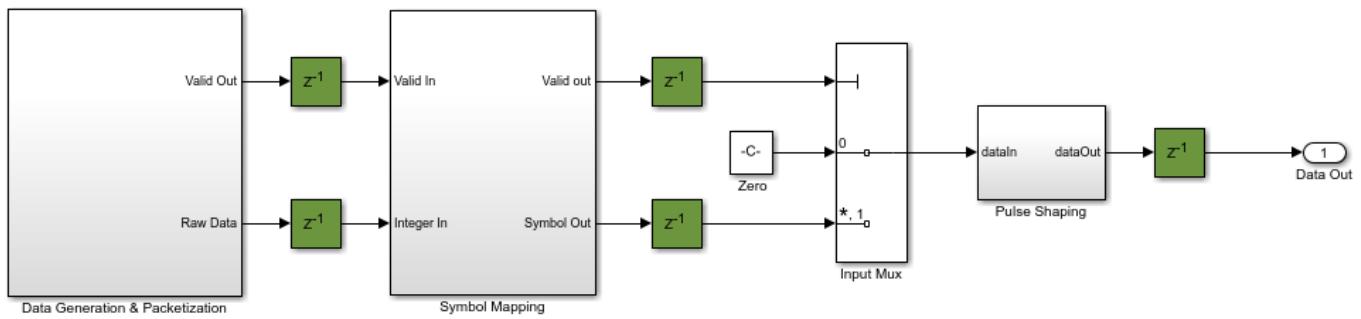
## Structure of the Example

The top-level structure of the QAM receiver model is shown in the following figure. The **QAM Tx HDL** and **QAM Rx HDL** subsystems have been optimized for HDL code generation.



Copyright 2014-2015 The MathWorks, Inc.

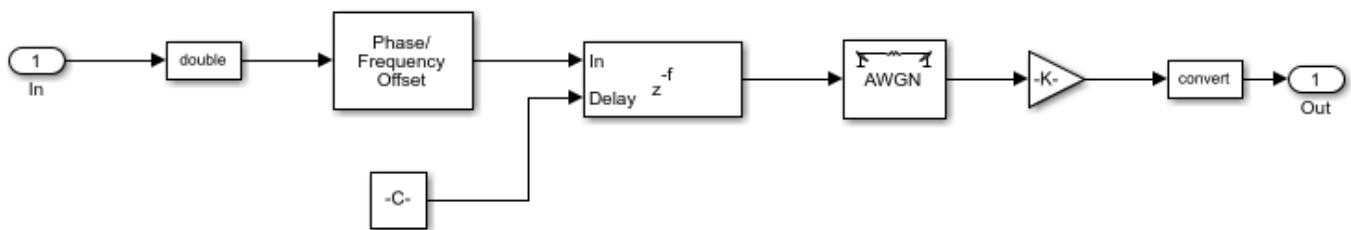
The detailed structure of the **QAM Tx HDL** subsystem can be seen in the figure below.



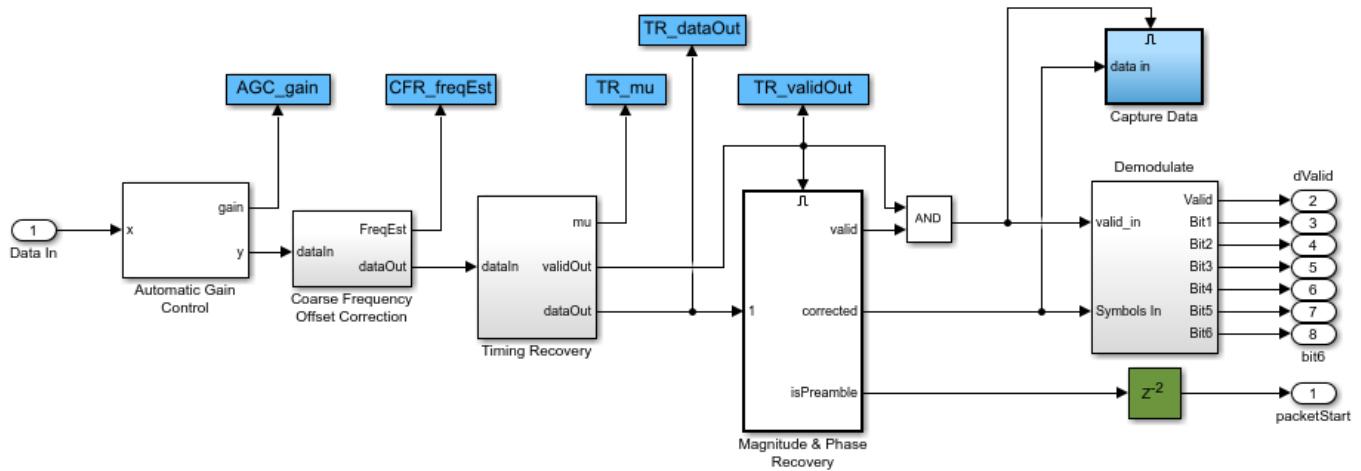
The **QAM Tx HDL** subsystem contains the following components, which are described in more detail in the **HDL Optimized QAM Transmitter** section.

- **Data Generation & Packetization** - Generates the packets to be transmitted, grouping the bits for mapping to symbols
- **Symbol Mapping** - Maps the bits output from the **Data Generation & Packetization** subsystem to QAM symbols
- **Pulse Shaping** - Performs pulse shaping and upsampling of the symbols using an interpolating RRC (Root Raised Cosine) filter prior to transmission

The structure of the **Channel** can be seen below. As the **Channel** subsystem is intended to be a rough approximation of a AWGN channel with attenuation and frequency offset it is intended to be run in software. As a result blocks which are not supported for HDL code generation can be used here, such as the **Phase/Frequency Offset** block. The **Phase/Frequency Offset** block does not support fixed point data types, hence the conversion to double at the input of the **Channel** subsystem. The signal is converted back to fixed point before being output from the **Channel** subsystem. A fractional delay and AWGN are applied to the transmitted signal and the **Gain** block attenuates the signal.



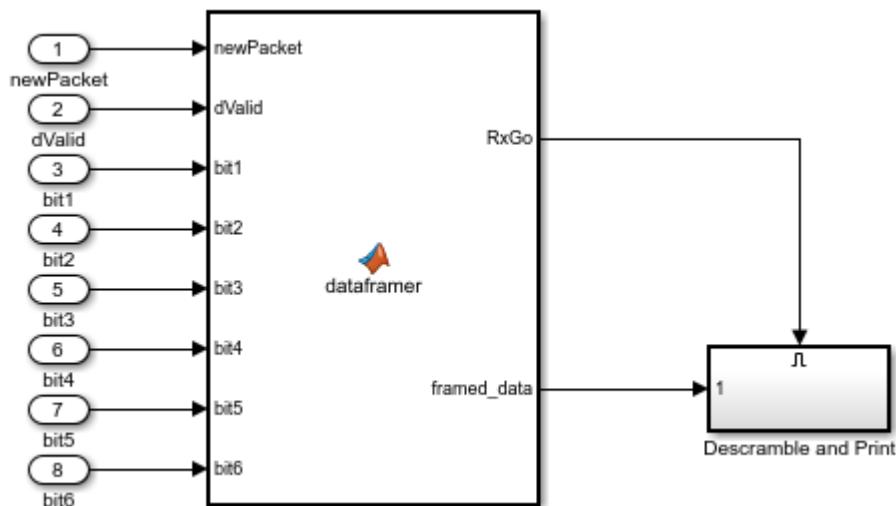
The detailed structure of the **QAM Rx HDL** subsystem can be seen in the figure below.



The **QAM Rx HDL** subsystem contains the following components which are described in more detail in the **HDL Optimized QAM Receiver** section.

- **Automatic Gain Control (AGC)** - Normalizes the received signal power
- **Coarse Frequency Offset Correction** - Estimates the approximate frequency offset and corrects. The subsystem also contains the receive RRC filter which downsamples by 2
- **Timing Recovery** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants
- **Magnitude & Phase Recovery** - Performs packet detection, fine grained phase and amplitude correction
- **Demodulate** - Demodulates the signal, de-mapping symbols to bits

The structure of the **Text Message Decoding** subsystem is shown below.



This subsystem is expected to be run in software, therefore, it is preferable to employ frame-based signals to speed up the computation. The **Text Message Decoding** subsystem has eight sample-based Boolean input signals: dValid, packetStart and signals bit1 to bit6. Conversion from sample-based signals to frame-based counterparts is implemented by the **dataframer** MATLAB function block. The demodulated bits are valid only when dValid is set high. The **dataframer** block uses the dValid signal to fill up a delay line with the received bits and the **newPacket** signal to forward the data stored in the delay line to the output and reset the delay line. The **Descramble and Print** subsystem processes the received data only when its enable signal goes high. This occurs when either the delay line accumulates 336 valid demodulated bits or the newPacket signal is high. This will cause the **dataframer** to set the RxGo signal high. While the simulation is running, the **Descramble and Print** subsystem outputs the string "Hello world! ~64QAM test string~ #####" to the MATLAB command window, where ##### is a repeating sequence of '000', '001', '002', ..., '099'. Every 50 packets, the bit error rate of the data in the last 50 successfully received packets is also displayed in the MATLAB Command Window.

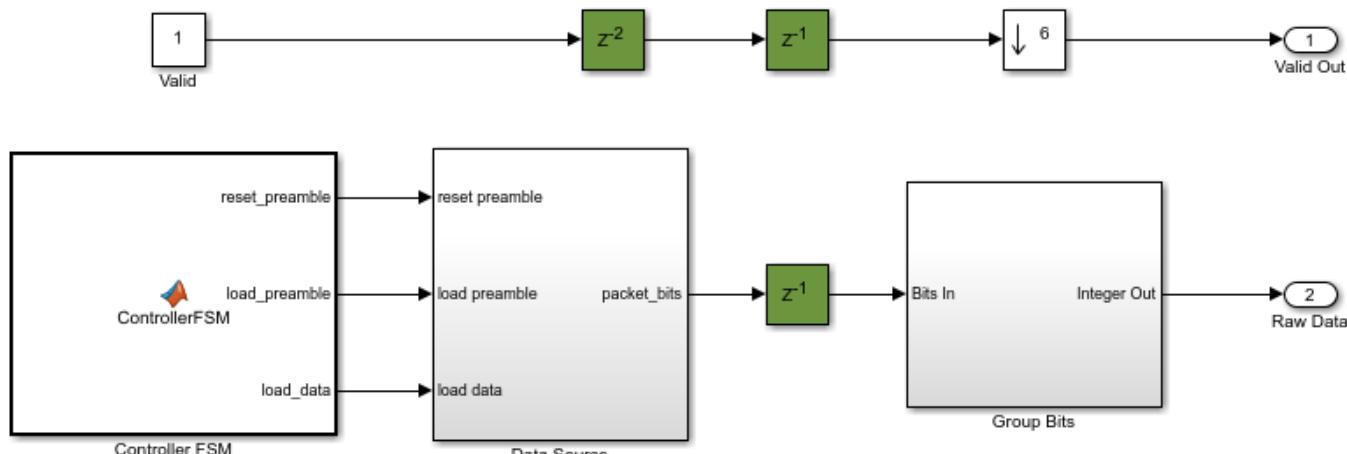
### HDL Optimized QAM Transmitter (HDL QAM Tx)

The **HDL Optimized Transmitter** contains the **Data Generation & Packetization**, **Symbol Mapping**, and **Pulse Shaping** blocks which are described in detail in the following sections.

#### 1 - Data Generation & Packetization

The **Controller FSM** (Finite State Machine) and **Data Source** generates the preamble bits, and the data bits, performs scrambling and builds the packets. Each packet consists of an 84-bit Barker code preamble and 252 bits of scrambled data. The **Group Bits** block converts the input data bit stream into a six bit integer at 1/6th of the input sampling rate, as required by the symbol mapper.

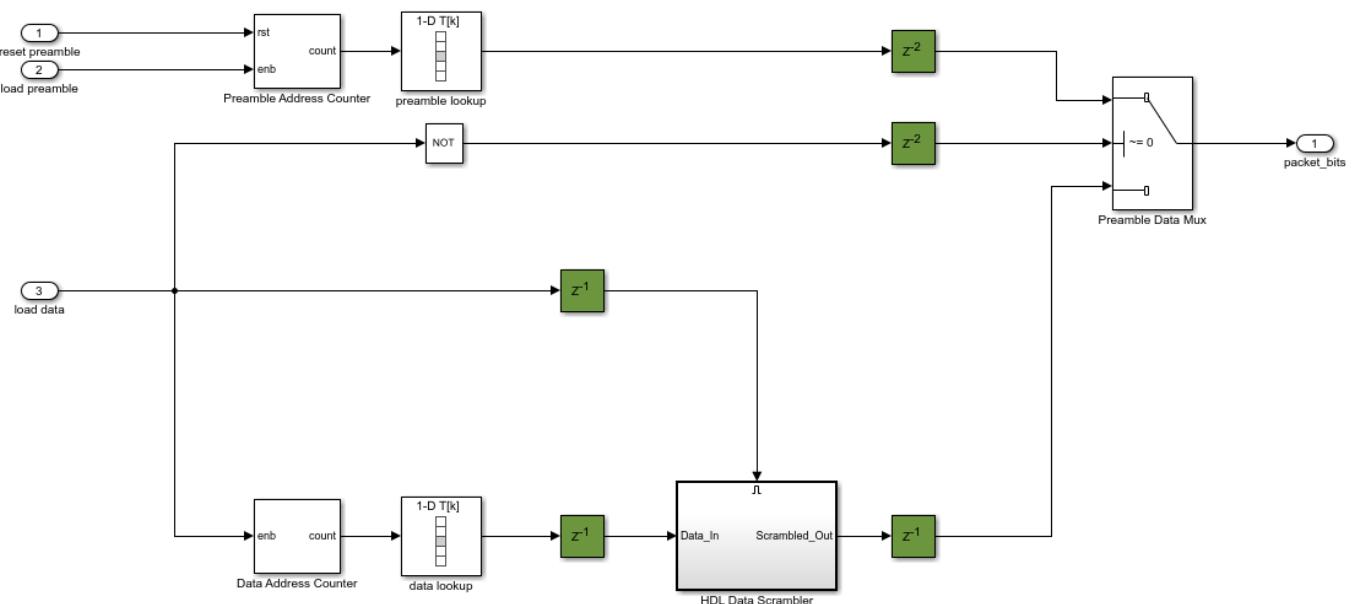
The **Data Source** subsystem has a pipeline delay of 2 samples. In addition there is a pipeline delay between the data source and the bit pairing subsystem. The valid signal is therefore delayed to match the pipeline delay of the data path. The **Group Bits** subsystem reduces the sample rate by a factor of 6. Placing a downsample by 6 in the valid control path ensures that the sample rate matches that of the signal path.

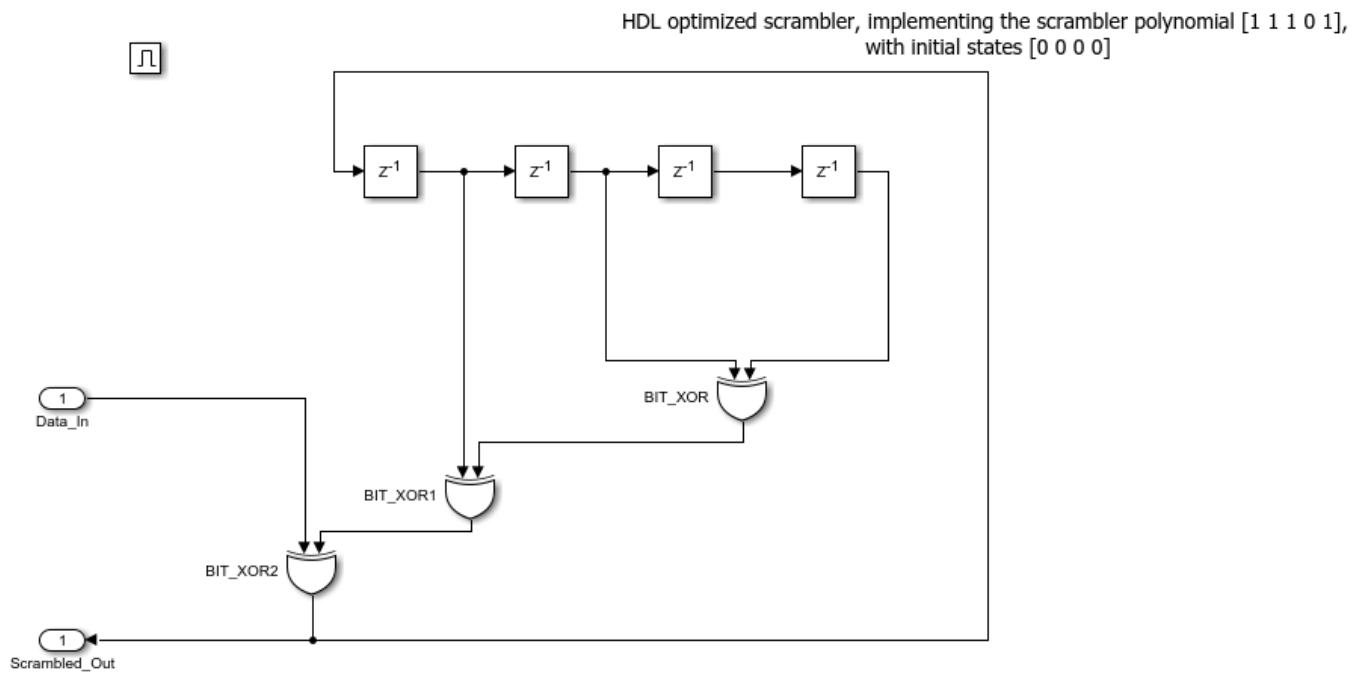


- **Controller FSM** - The **Controller FSM** implements a control state machine using a MATLAB™ function block. The FSM has two states - **Pack\_Preamble** and **Append\_Data**. The **Pack\_Preamble** state asserts the **load\_preamble** signal and de-asserts the **reset\_preamble** and the **load\_data** signals. The FSM will remain in this state for 84 clock cycles. Following this the

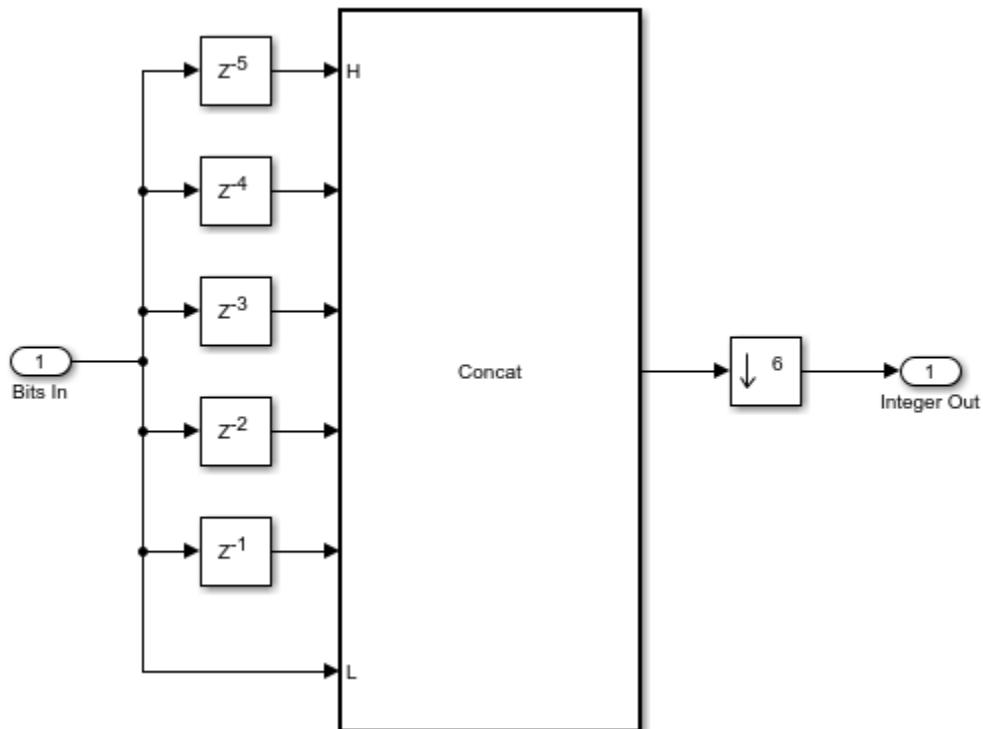
FSM moves into the **Append\_Data** state, asserting the **load\_data** signal and the **reset\_preamble** signal while releasing the **load\_preamble** signal. The FSM will remain in this state for 252 clock cycles. The **load\_preamble** and **reset\_preamble** are boolean and are used to control the **Preamble Address Counter** which manages the load of the preamble at the start of each packet. The **load\_data** signal is boolean and is used to enable the **Data Address Counter** which controls the loading of data into the packet.

- **Data Source** - The **Data Source** Subsystem contains two LUTs, storing the preamble and data bits. The **preamble lookup** LUT is addressed by the **Preamble Address Counter**, which is controlled by the **reset preamble** and **load preamble** signals generated by the **Controller FSM**. The **data lookup** LUT is addressed by the **Data Address Counter**, which is enabled by the **load\_data** signal generated by the **Controller FSM**. The **Preamble Address Counter** has a reset signal, generated by the **Controller FSM**, as the same preamble is inserted at the start of each packet. The **Data Address Counter** does not have a reset signal as the data address sequence is much longer and will vary for each packet as different data bits are placed within each packet. In addition to enabling the counter for the data LUT, the **load data** input is used to control when the **HDL Data Scrambler** component should be enabled, and to control selection of preamble or data bits via the **Preamble Data Mux**.



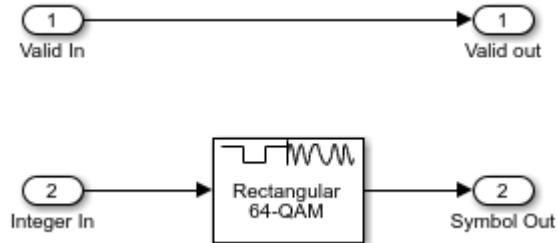


- **Group Bits** - The purpose of the **Group Bits** subsystem is to group six individual bits into a six-bit unsigned integer output - the format expected by the symbol mapping component. A number of delays are used to align 6 bits at the input of the **Bit Concat** block which concatenates into a six-bit unsigned output. This output is then downsampled to select the correct grouping of bits.



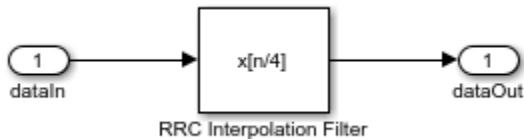
## 2 - Symbol Mapping

The **Symbol Mapping** subsystem uses the **Rectangular QAM Modulator Baseband** block to map the integer input value onto the appropriate 64-QAM complex valued symbol. The block uses a Gray Mapping scheme.



## 3 - Pulse Shaping

The Pulse Shaping subsystem uses an **RRC Interpolation Filter** block with an upsampling factor of 4. A matched filter is implemented in the receiver. The filter is pipelined (see HDL Block Properties).



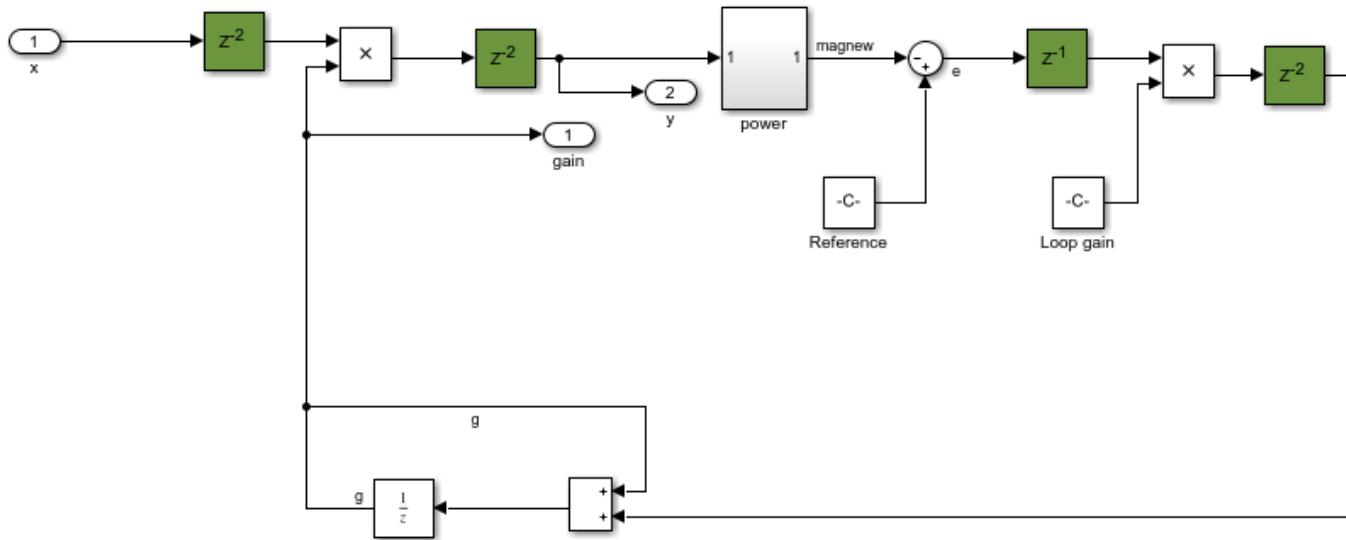
## HDL Optimized QAM Receiver (HDL QAM Rx)

The **HDL Optimized Receiver** contains the **AGC**, **Coarse Frequency Offset Correction**, **Timing Recovery**, **Magnitude & Phase Recovery**, and **Demodulate** blocks, which are described in detail in the following sections.

### 1 - AGC

The **AGC** ensures that the amplitude of the input of the **Coarse Frequency Compensation** is normalized to the range 1 to -1.

The **AGC** structure is shown in the following diagram, with pipeline registers shown in green throughout the model.

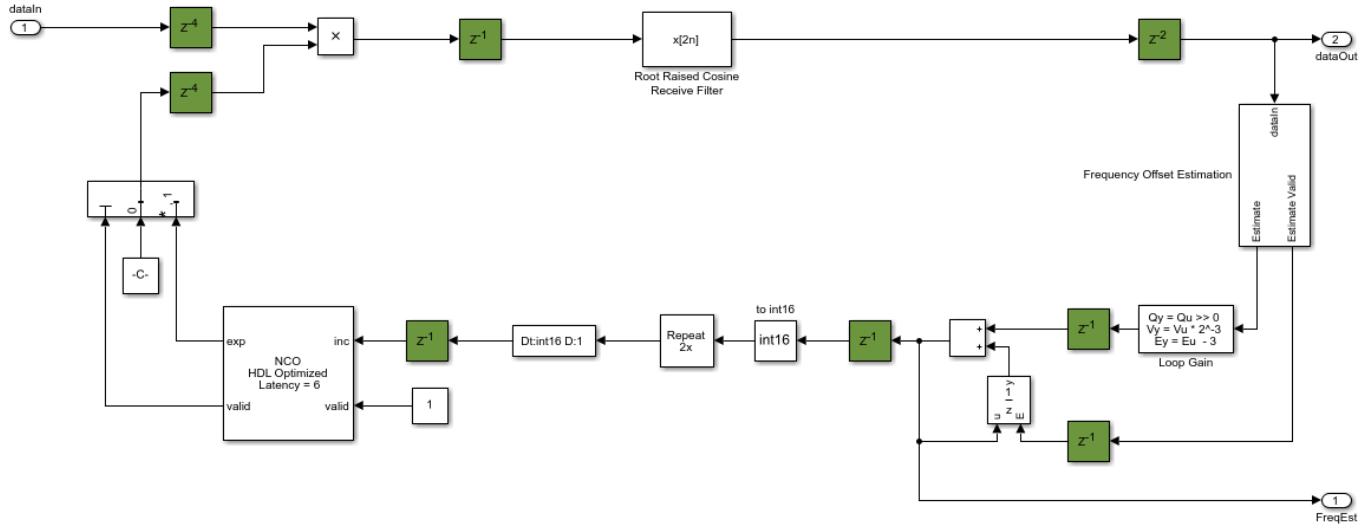


## 2 - Coarse Frequency Offset Correction

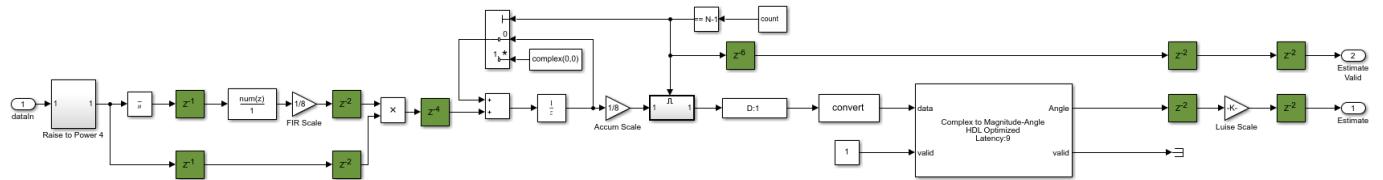
The **Coarse Frequency Offset Correction** subsystem estimates and corrects for the frequency offset using the Luise-Reggiannini algorithm [ 1 ]. The **Frequency Offset Estimation** subsystem makes an estimate based on the output of the **Root Raised Cosine Receive Filter**, then frequency offset correction based on this estimate is applied at the input to the **Root Raised Cosine Receive Filter**. This ensures that the desired portion of the received signal bandwidth is better aligned with the receiver filter frequency response, improving the SNR compared to correcting at the output of the **Root Raised Cosine Receive Filter**.

As the estimation and correction algorithm is operating in a closed loop, making iterative updates to the previous estimates of the frequency offset, the system will gradually converge towards a result. A **Loop Gain** is included to implement averaging of the estimates. This architecture is described in [ 1 ]. The **Root Raised Cosine Receive Filter** implements a downsampling operation so it is necessary to upsample the feedback signal, using the repeat block, to match the rate at the input to the filter.

Note that there is a residual frequency offset at the output of the **Coarse Frequency Offset Correction** subsystem that varies over time, even if the frequency offset at the input to the subsystem remains the same, as new estimates of the offset are made. Fine grained correction of the residual offset is performed later in the receiver by the **Magnitude and Phase Recovery** subsystem.

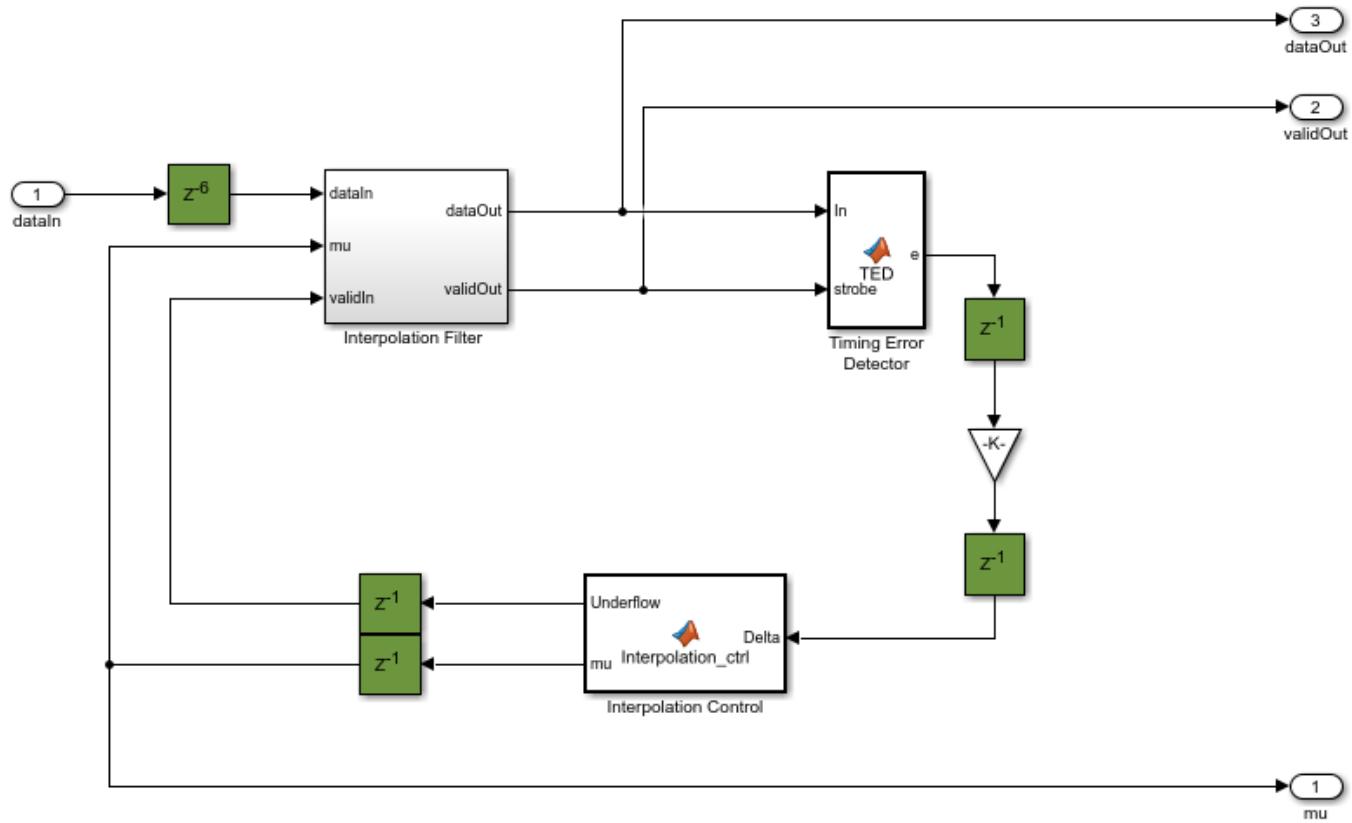


- Frequency Offset Estimation :** The **Frequency Offset Estimation** subsystem implements the Luise-Regiannini algorithm, described in [ 1 ]. The signal is first raised to the power four to implement a 4th power phase estimator as described in [ 2 ]. This is implemented by 2 cascaded product blocks, with pipelining added to improve hardware performance. The **Discrete FIR Filter** implements the filter with rectangular weights, made up of all ones, described in [ 1 ]. The **FIR Scale** scales the FIR output to account for the filter gain. The **Complex To Magnitude-Angle HDL Optimized** block is used to implement the *angle* function, as required by the Luise-Regiannini algorithm. This block computes the phase using the hardware friendly CORDIC algorithm. For more information, see the Complex to Magnitude-Angle HDL Optimized (DSP System Toolbox) block in DSP System Toolbox™ . Before the **Frequency Offset Estimation** subsystem output, the signal is scaled as required by the Luise-Regiannini algorithm and, in addition, is scaled to match the word length of the NCO.



### 3 - Timing Recovery

The **Timing Recovery** subsystem is shown in the following diagram.



The **Timing Recovery** subsystem implements a PLL, described in Chapter 8 of [ 3 ], to correct the timing error in the received signal. On average, the **Timing Recovery** subsystem generates one output sample for every two input samples.

The **Interpolation Control** function block implements a decrementing modulo-1 counter, described in Chapter 8.4.3 of [ 3 ], to generate the control signal to facilitate the selection of the interpolants of the **Interpolation Filter**. This control signal also enables the **Timing Error Detector (TED)**, so that it calculates the timing errors at the correct timing instants. The **Interpolation Control** subsystem updates the timing difference, **mu**, for the **Interpolation Filter**, generating interpolants at optimum sampling instants.

The **Interpolation Filter** is a Farrow parabolic filter with  $\alpha = 0.5$  as described in Chapter 8.4.2 of [ 3 ]. The filter uses an  $\alpha$  of 0.5 so that all the filter coefficients become 1, -1/2 and 3/2, which significantly simplifies the interpolator structure. Based on the interpolants, timing errors are generated by a zero-crossing **Timing Error Detector** as described in Chapter 8.4.1 of [ 3 ].

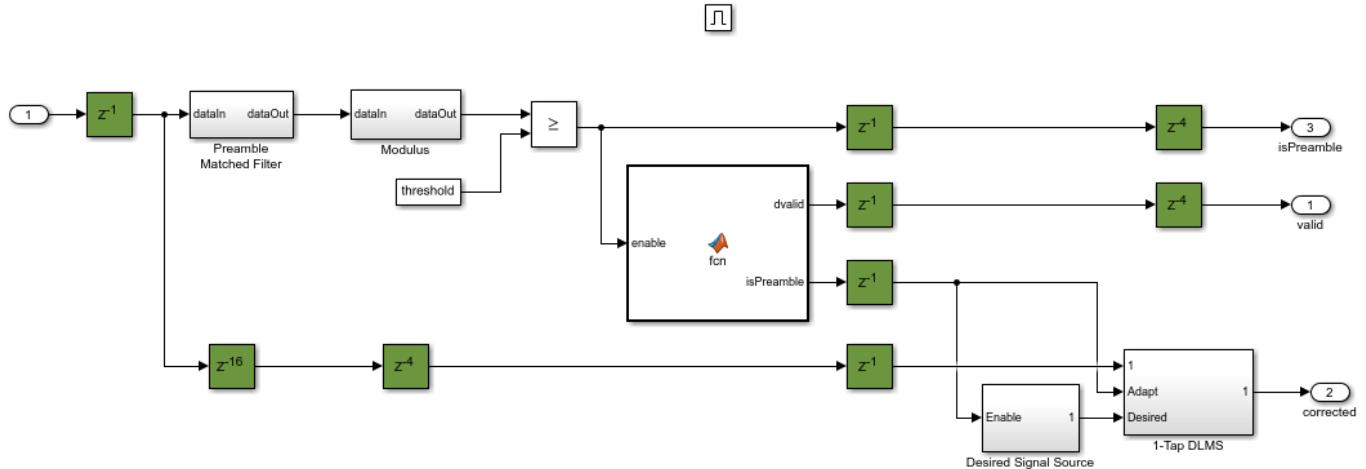
The **Interpolation Filter** introduces a fractional delay to the signal in order to compensate for the timing error. The fractional delay is controlled by the **mu** input signal. When the timing error (delay) reaches symbol boundaries, there is one extra or missing interpolant in the output. The **Timing Error Detector** implements bit stuffing or skipping to handle the extra or missing interpolants.

Refer to Chapter 8.4.4 of [ 3 ] for details of bit stuffing and skipping. The timing recovery loop normally generates one output symbol for every two input samples. It also outputs a timing strobe (**validOut** signal) that runs at the input sample rate. Under normal circumstances, the strobe value is simply a sequence of alternating ones and zeros. However, this occurs only when the relative delay

between transmitter and receiver contains some fractional part of one symbol period and the integer part of the delay (in symbols) remains constant. If the integer part of the relative delay changes, the strobe value can have two consecutive zeros or two consecutive ones.

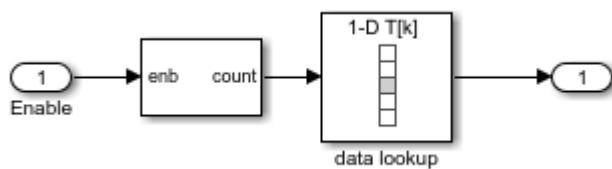
#### 4 - Magnitude & Phase Recovery

The **Magnitude & Phase Recovery** subsystem performs packet synchronization, fine grained frequency recovery and fine grained amplitude recovery.

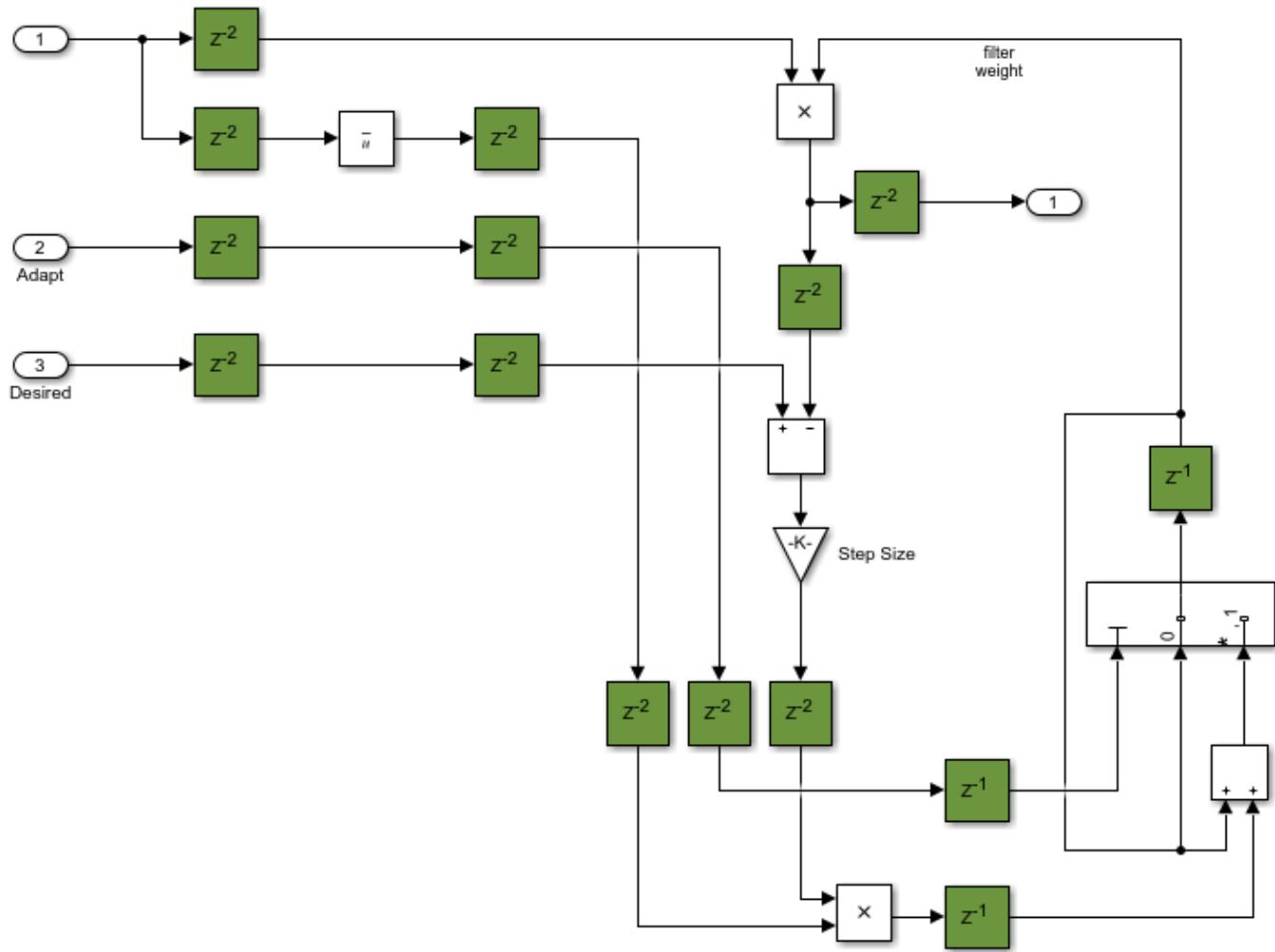


- Packet Synchronization:** The **Preamble Matched Filter** subsystem uses the time-reversed complex conjugate of the preamble as the filter weights. The modulus of the output of the **Preamble Matched Filter** subsystem is calculated using the **Modulus** subsystem. The output of the **Modulus** subsystem is then compared to a threshold to detect the preamble at the start of a packet. The MATLAB function block generates a signal, **isPreamble**, which is held high for the duration of the preamble of each packet. The MATLAB function block also generates the **dvalid** signal which is set high for the duration of the packet when a preamble has been detected.
- Fine Grained Magnitude and Phase Recovery :** The **1-Tap DLMS** (Delayed Least Mean Squares) filter subsystem, adapting over the preamble and using the reference signal generated by **Desired Signal Source**, corrects for both phase and magnitude errors. The **isPreamble** signal, generated by the MATLAB function block and set high for the 14 preamble symbols once a packet has been detected, is used to enable the desired signal source and to enable the **Adapt** input of the **1-Tap DLMS**. When the **isPreamble** signal is low, the weight in the **1-Tap DLMS** is held and the **Desired Signal Source** is reset. The Delayed LMS (DLMS) [ 4 ] algorithm is used here to allow for more pipelining to be introduced and, therefore, reduce the critical path in the filter and increase the maximum clock rate achievable after being implemented in hardware.

The internal structure of the **Desired Signal Source** subsystem is shown below. The **data lookup** LUT contains the preamble symbols.

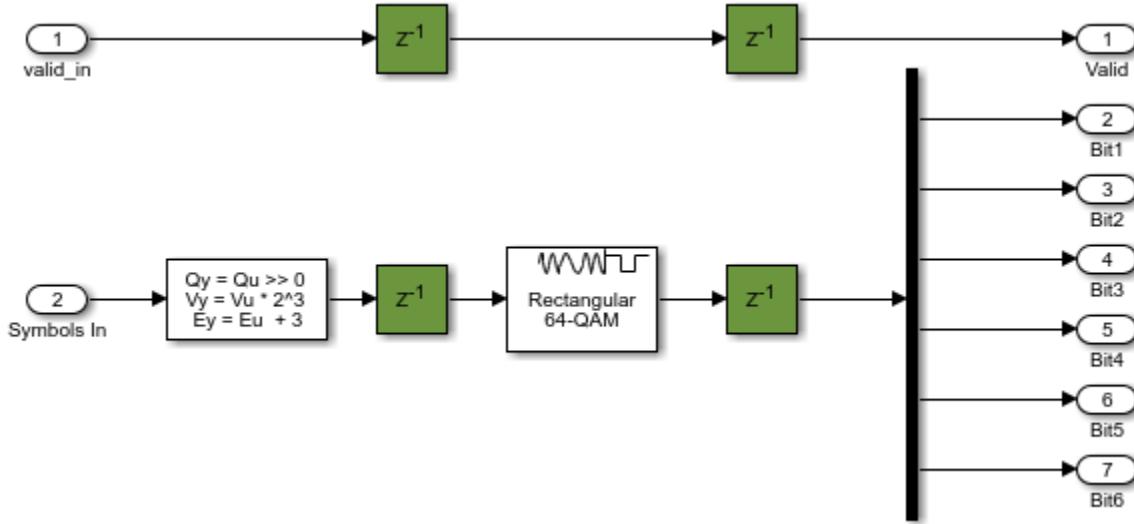


The internal structure of the **1-Tap DLMS** subsystem is shown below.



### 5 - Demodulate

The **Demodulate** subsystem maps each 64-QAM input symbol to bits, outputting 6 bits for each input symbol. To generate HDL for the **Rectangular QAM Demodulator Baseband** block, the minimum distance between symbols must be set to 2. This is 8 times larger than the distance between the symbols generated in the transmitter. As a result, the symbols input to the **Demodulate** subsystem must be scaled up appropriately. This is done using the **Shift Arithmetic** block which shifts the binary point left by 3 bits to achieve the required multiplication by 8.



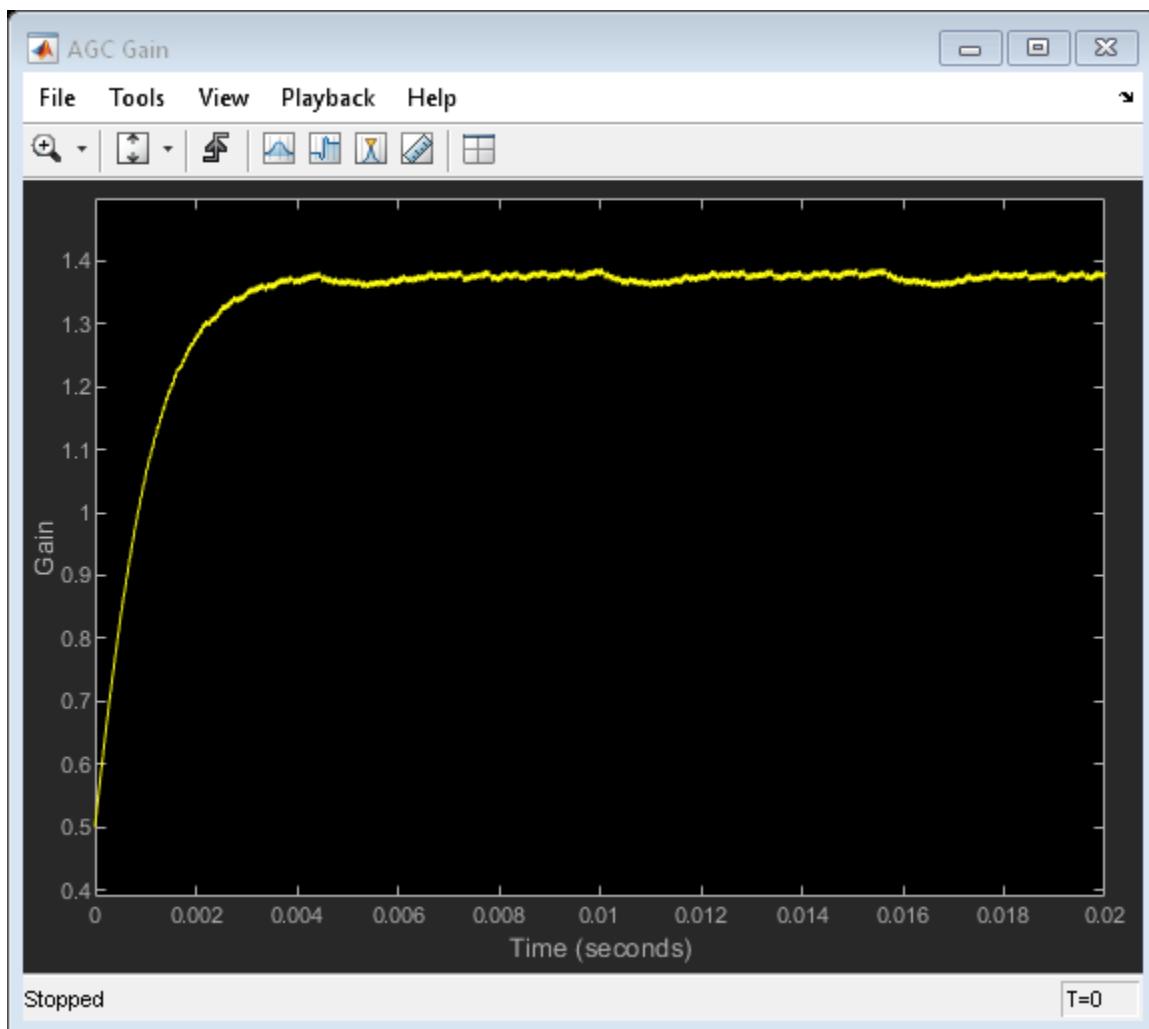
## Results and Displays

During the simulation, the model displays successfully received packets in the MATLAB Command Window. At every 50 packets, the bit error rate of the data in the last 50 successfully received packets is also displayed in the MATLAB Command Window.

After running the simulation, the model displays six different figures illustrating different aspects of the receiver performance. These are shown below, along with an explanation of each plot. The first five plots show the adaption, over the simulation duration, of the **Automatic Gain Control**, the **Frequency Offset Estimation**, the **Timing Recovery** position estimate, the real part of the constellation at the output of the **Timing Recovery** subsystem, and at the output of the **Magnitude & Phase Recover** subsystem. The last plot shows the constellation diagram at the output of **Magnitude & Phase Recovery** subsystem after any adaption has taken place.

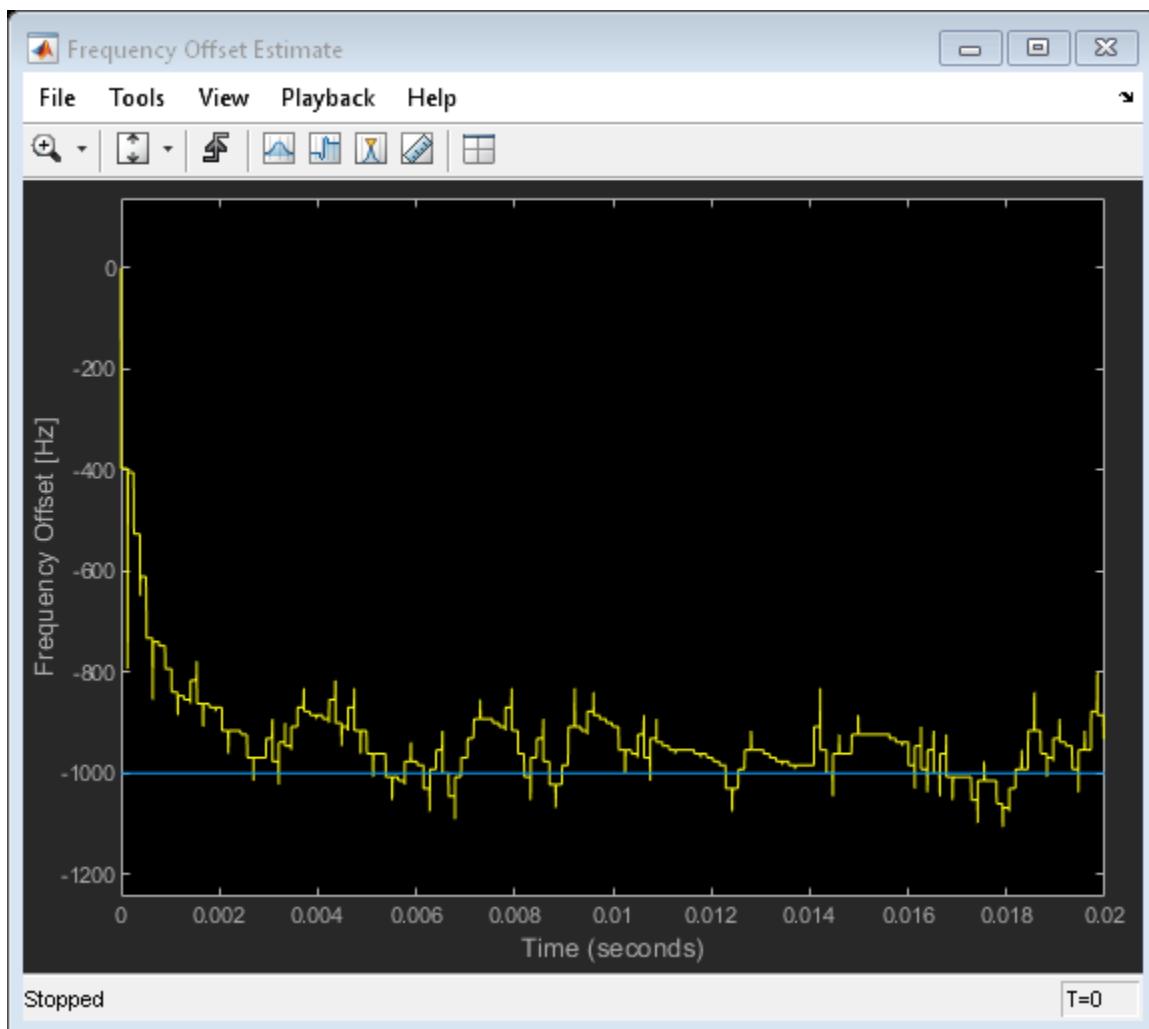
- **AGC Gain Plot**

The following plot illustrates the **Automatic Gain Control** subsystem adapting over time to normalize the output. A balance must be struck between how quickly the AGC adapts and how much ripple there is after the gain has reached a relatively constant level. Using a larger AGC loop gain adapts faster but the amplitude after adaption varies more. Using a smaller loop gain slows the adaption of the AGC, smoothing the level after adaption but taking longer to adapt.



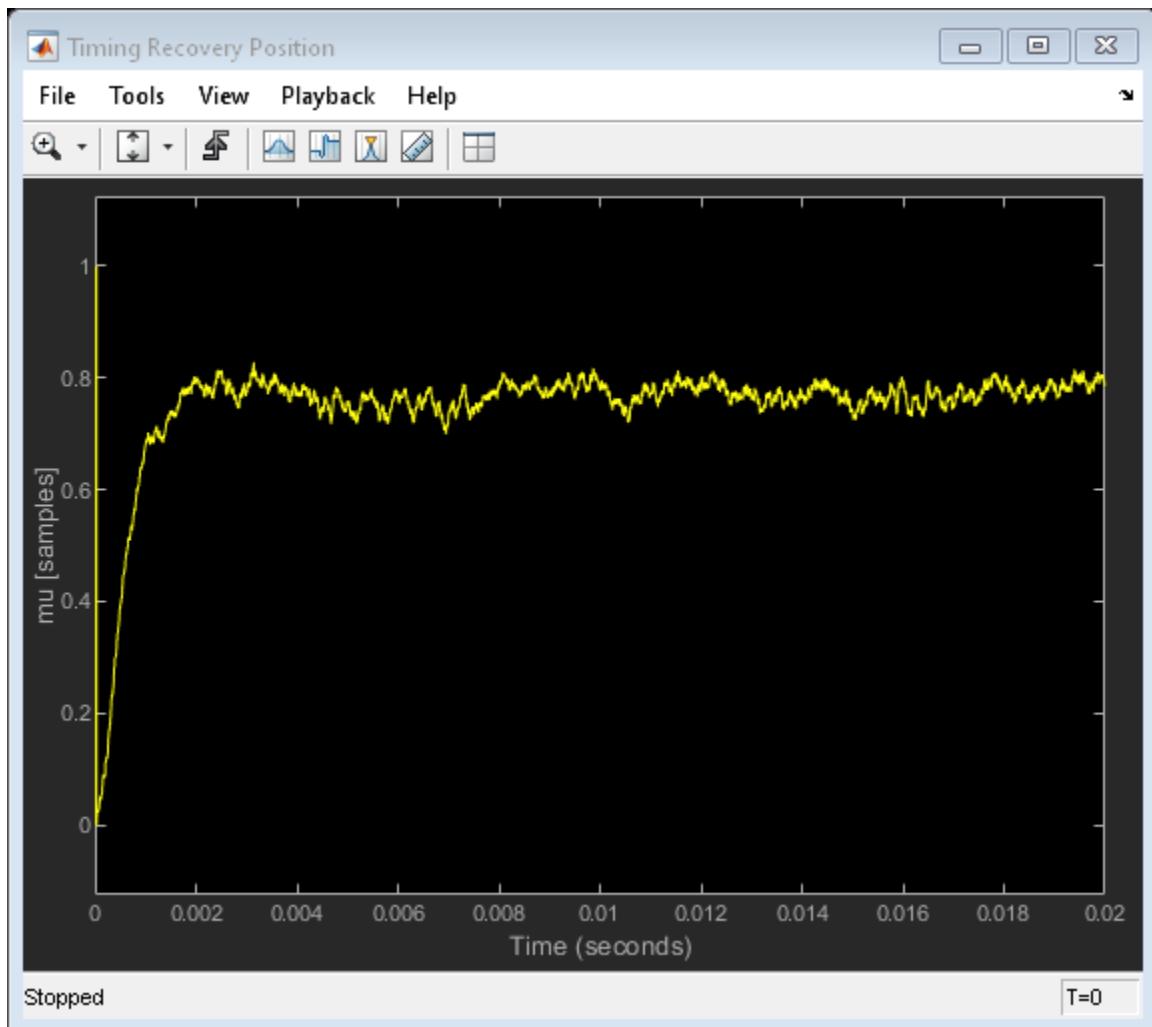
- **Frequency Offset Estimate Plot**

The following plot illustrates how the coarse frequency offset gradually adapts towards the frequency offset introduced by the system (the blue horizontal line). It shows that while the estimate comes close to the actual frequency offset, there is still a residual error that must be addressed later in the system.



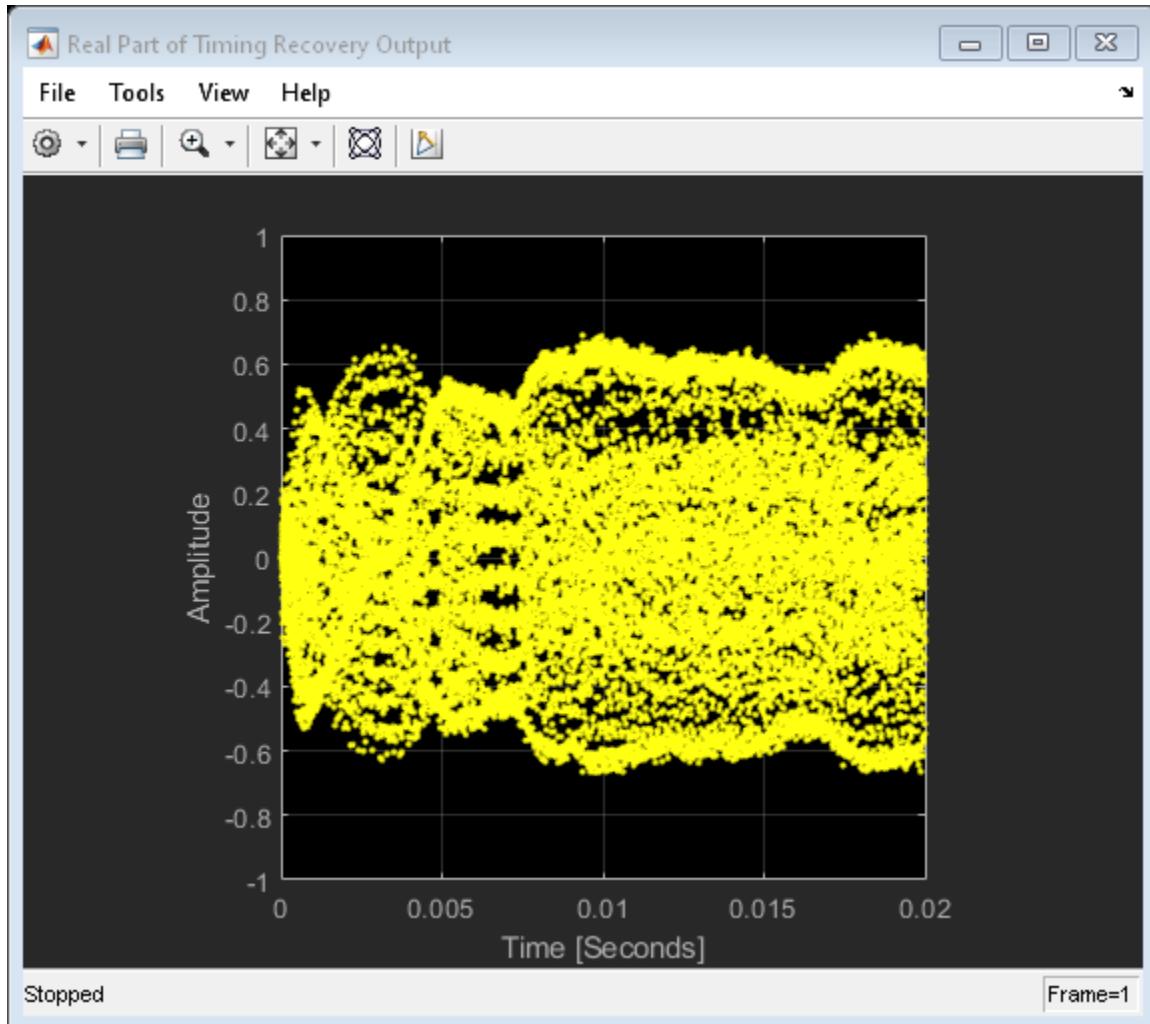
- **Timing Recovery Position Plot**

The following plot shows the **mu** input to the **Interpolation Filter**. Note that **mu** converges to a steady state (with some ripple) over time as the channel delay is not varying during the simulation.



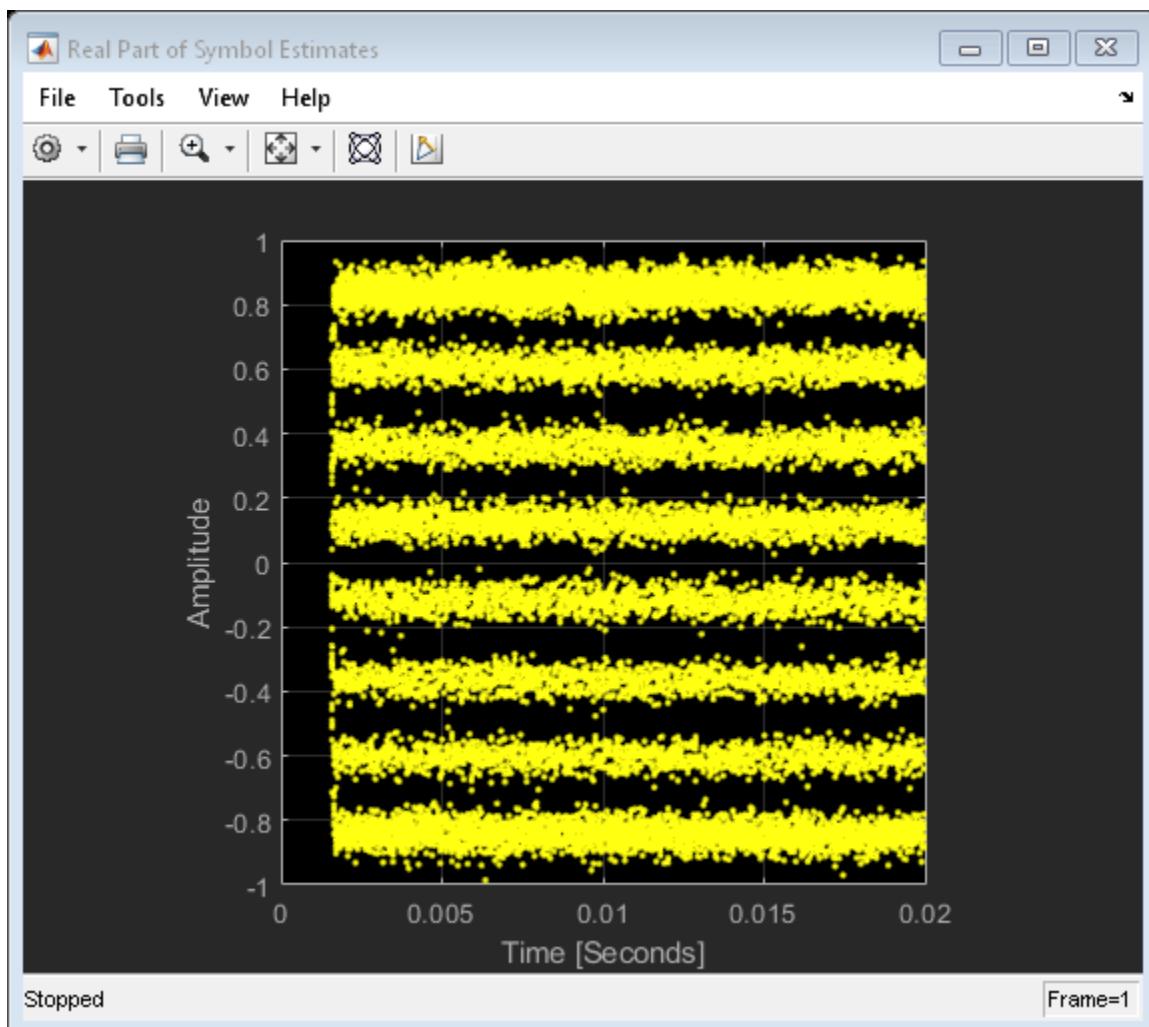
- **Real Part of Timing Recovery Output Plot**

The following plot illustrates how the real part of the **Timing Recovery** subsystem output is beginning to converge towards the eight distinct amplitude levels expected for 64QAM. However, as the residual frequency offset remaining after the coarse frequency recovery has not yet been corrected at this point in the receiver, the quality of the signal varies with the distinct amplitude levels more clearly visible at some points than at others. The constellation still has some rotation at this point in the receiver.



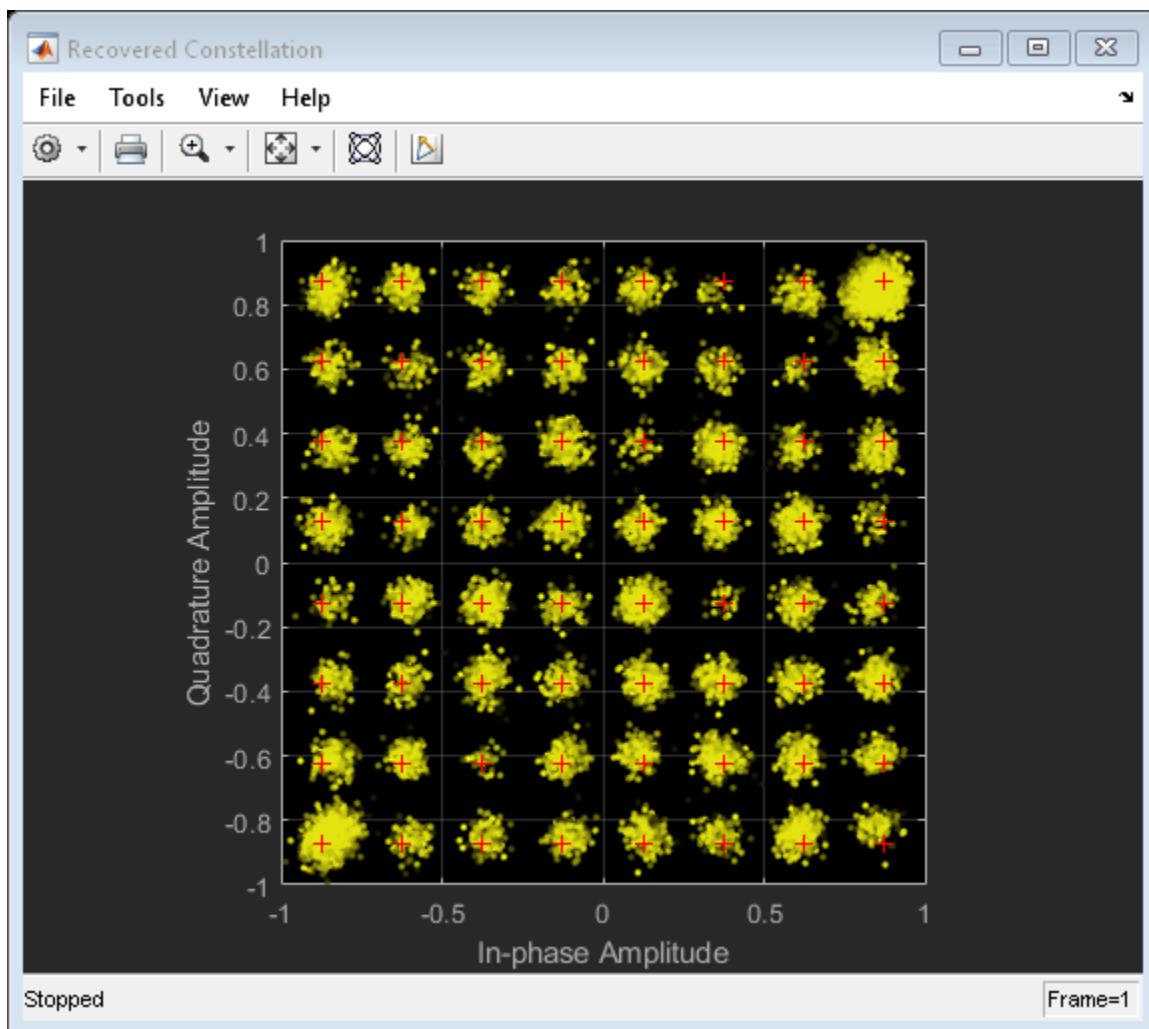
- **Real Part of Symbol Estimates Plot**

The following plot shows how the real part of output of the **Magnitude & Phase Recovery** subsystem adapts over time. Unlike the previous plot, this diagram is generated after the fine frequency recovery, therefore the constellation should not be rotating. There are no samples initially as the output from the block is not valid, and then eight clear amplitude levels should be seen - representing the eight real amplitude levels of the 64-QAM constellation.



- **Recovered Constellation Plot**

The following plot shows the constellation at the output of the **Magnitude & Phase Recovery** subsystem after the system has had time to adapt to the channel. Reducing the channel noise should reduce the size of each of the constellation points; increasing the channel noise begins to merge the distinct constellation points together. If the system has not successfully corrected for the frequency offset, then rotation of the constellation is visible here.



## References

1. M. Luise and R. Reggiannini, "Carrier frequency recovery in all-digital modems for burst-mode transmissions," *IEEE Trans. Communications*, pp. 1169-1178, 1995.
2. Moeneclaey, M. and De Jonghe, G. "ML-oriented NDA carrier synchronization for general rotationally symmetric signal constellations", *IEEE Trans. Communications*, pp.2531-2533, 1994.
3. Michael Rice, "Digital Communications - A Discrete-Time Approach", Prentice Hall, April 2008.
4. G. Long , F. Ling and J. G. Proakis "The LMS algorithm with delayed coefficient adaptation", *IEEE Trans. on Acoustics, Speech and Signal Processing*, pp.1397-1405, 1989.

## Airplane Tracking with ADS-B Captured Data

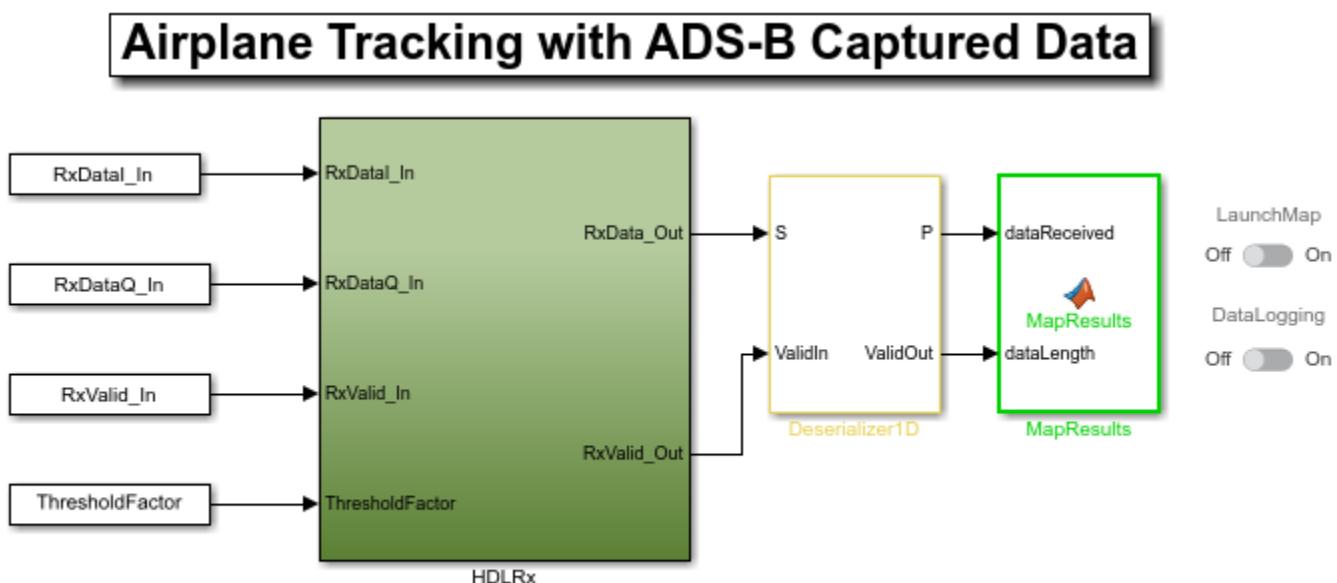
This example shows how to implement the Automatic Dependent Surveillance - Broadcast (ADS-B) receiver for HDL code generation and hardware implementation. This example decodes ADS-B extended squitter messages which can be used to track the airplane. The HDL-optimized model in this example uses Simulink® blocks that support HDL code generation to implement the ADS-B Receiver. This example model is used for real-time processing in “HW/SW Co-Design Implementation of ADS-B Transmitter/Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio), which requires the Communications Toolbox™ Support Package for Xilinx® Zynq®-Based Radio.

### Introduction

ADS-B is an air traffic management and control surveillance system. The broadcast messages (approximately once per second) contain the flight information including position and velocity. For introduction on ADS-B technology and modes of transmission, see [ 1 ]. The **HDLRx** subsystem is optimized for HDL code generation. The captured received signal is streamed into the receiver (**HDLRx** subsystem) front end. The streaming output of the receiver is buffered and passed to the **MapResults** MATLAB® function to view the output.

### Structure of the Example

The model supports both Normal and Accelerator modes. The top-level structure of the ADS-B receiver model is shown in the following figure.

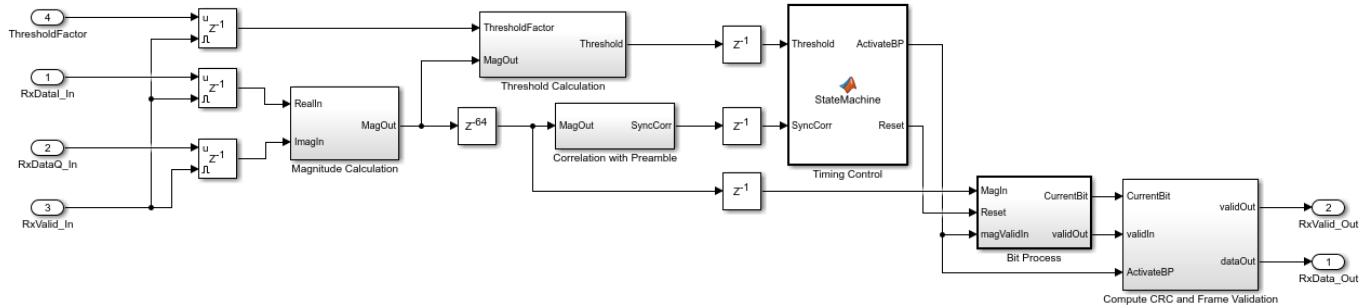


Copyright 2018 The MathWorks, Inc.

The receiver input data is captured using “HW/SW Co-Design Implementation of ADS-B Transmitter/Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) running on the Zynq® platform. The captured data represents the baseband received signal with a sampling rate of 4 MHz. The data contains 8 frames of extended squitter messages. The ADS-B transmitter modulates the 112-bit extended squitter messages using 2-

bit pulse-position modulation, and adds a 16-bit prefix. Then, to generate 4 MHz data, each 240-bit message is zero-padded and upsampled by 2.

This diagram shows the detailed structure of the **HDLRx** subsystem.



The subsystems listed here are described further in the following sections.

1. **Magnitude Calculation** - Finds the complex modulus of the received input signal
2. **Threshold Calculation** - Calculates the threshold value based on received input signal strength
3. **Correlation with Preamble** - Correlates the received signal with reference signal to detect the preamble
4. **Timing Control** - Provides timing synchronization for the receiver
5. **Bit Process** - Decodes symbols using PPM demodulation
6. **Compute CRC and Frame Validation** - Validates the frame by checking for CRC errors

## HDL Optimized ADS-B Receiver

### 1. Magnitude Calculation

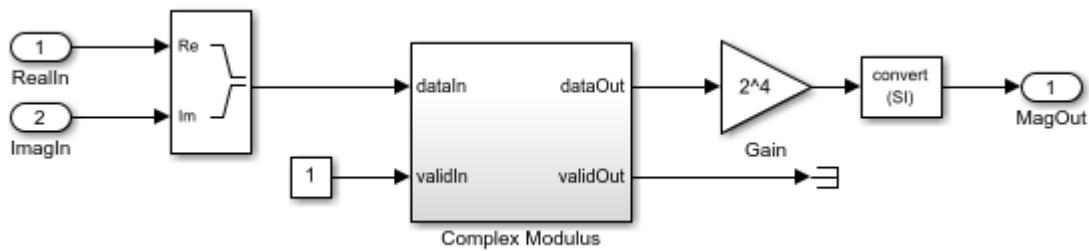
The inputs to the **Magnitude Calculation** subsystem are the in-phase (real) and quadrature (imaginary) phase samples. This subsystem outputs the modulus of the complex number. The  $\sqrt{I^2 + Q^2} = |L| + 0.4 * |S|$  algorithm described on page 238 of [ 2 ].

$$\sqrt{I^2 + Q^2} = |L| + 0.4 * |S|$$

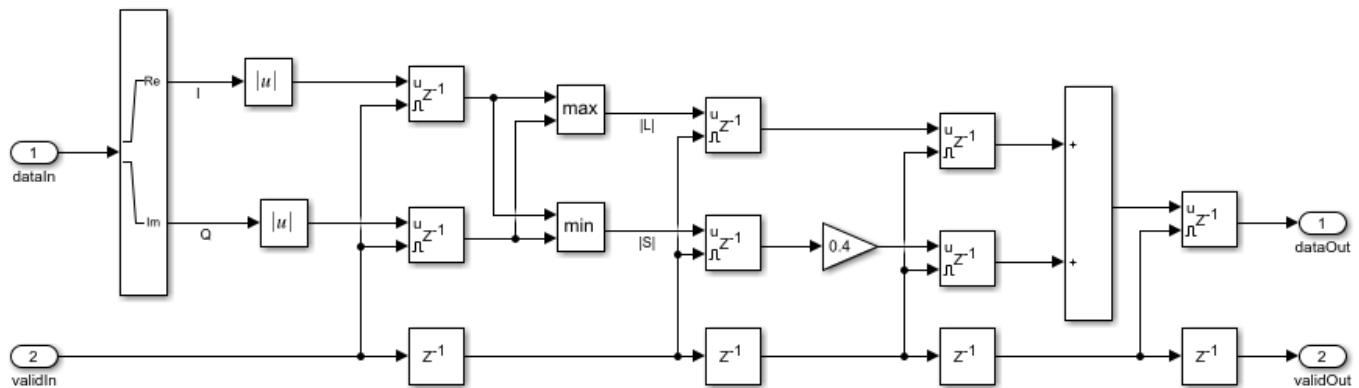
where

- | L | is the larger value of | I | or | Q |
- | S | is the smaller value of | I | or | Q |.

The Gain block converts received input from 12-bit to 16-bit word length.

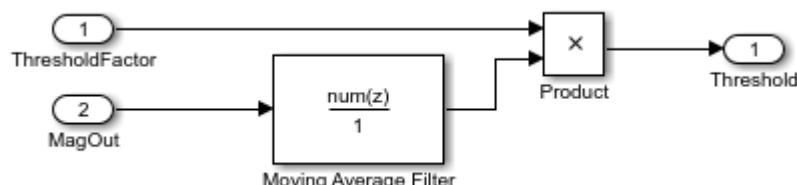


For the implementation of " $|L|+0.4|S|$  algorithm", see the following model.



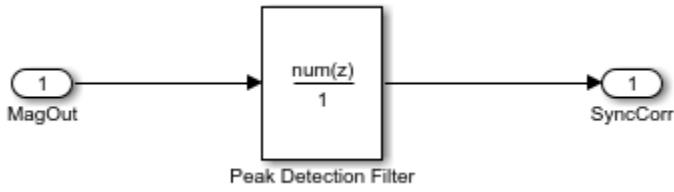
## 2. Threshold Calculation

The **Threshold Calculation** subsystem calculates the signal energy and applies a scaling factor to create a threshold for preamble detection. Moving Average Filter is a serial FIR filter architecture with 32 coefficients that operates on the magnitude values. The coefficients of the FIR filter are selected to find the average energy of the received signal. This example scales the signal energy by 5 to detect valid ADS-B preambles. For details on FIR filter, see Discrete FIR Filter.



## 3. Correlation with Preamble

The **Correlation with Preamble** subsystem correlates the received signal with the ADS-B reference/preamble sequence [1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0] using a peak detection filter. The peak detection filter is a serial FIR Filter architecture, configured with coefficients that match the preamble sequence. Preamble correlation identifies potential ADS-B transmissions and aligns our bit detection algorithm with the first message bit. The preamble is detected if the peak amplitude exceeds the scaled threshold value. Once the preamble is detected, the correlation value is passed on as input(SyncCorr) to the **Timing Control** block.

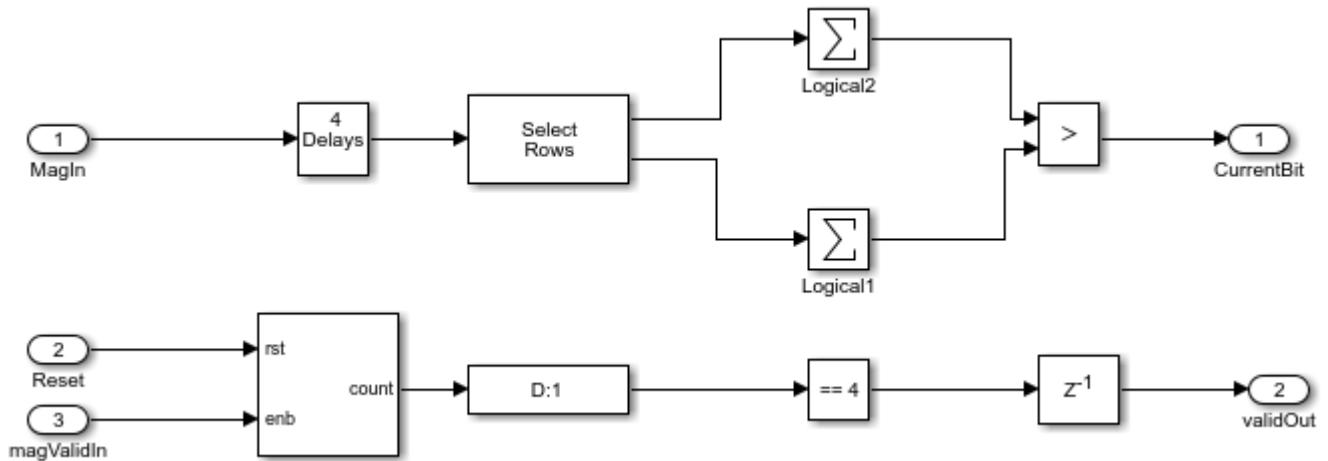


#### 4. Timing Control

The **Timing Control** block is a state machine that detects the preamble and generates the control signals **ActivateBP** and **Reset**, that indicate the start of frame, end of frame and reset status to the **Bit Process** and **Compute CRC and Frame Validation** blocks.

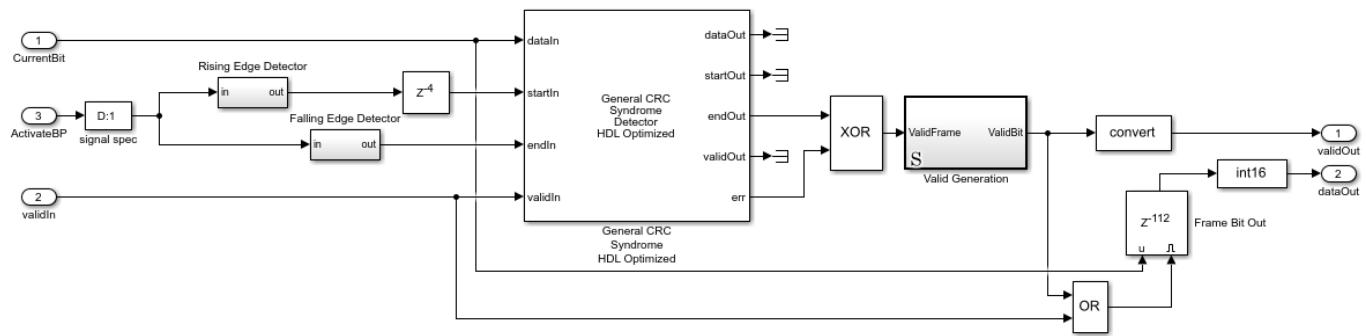
#### 5. Bit Process

The **Bit Process** subsystem demodulates and down converts the 4 MHz received signal to a 1 MHz bit sequence. Each data bit is represented by four PPM bits. To demodulate, the block finds the sum of the first two bits and the last two bits of each quadruplet. Then, it compares the sums to determine the original bit value. The output valid signal is asserted every fourth cycle to align with 1 MHz bit sequence.



#### 6. Compute CRC and Frame Validation

This subsystem checks for mismatches in the 24-bit checksum of each 88-bit message. The CRC block needs an indication of the frame boundaries to determine which bits are the checksum. The rising edge of the **ActivateBP** signal generated from the **Timing Control** block indicates the start of frame, and the falling edge indicates the end of the frame. The start signal is delayed to match the demod latency. When the block output err signal is zero, the frame is a valid ADS-B message. The subsystem buffers the message bits until the message is confirmed to have no CRC error.



### Launch Map and Log Data

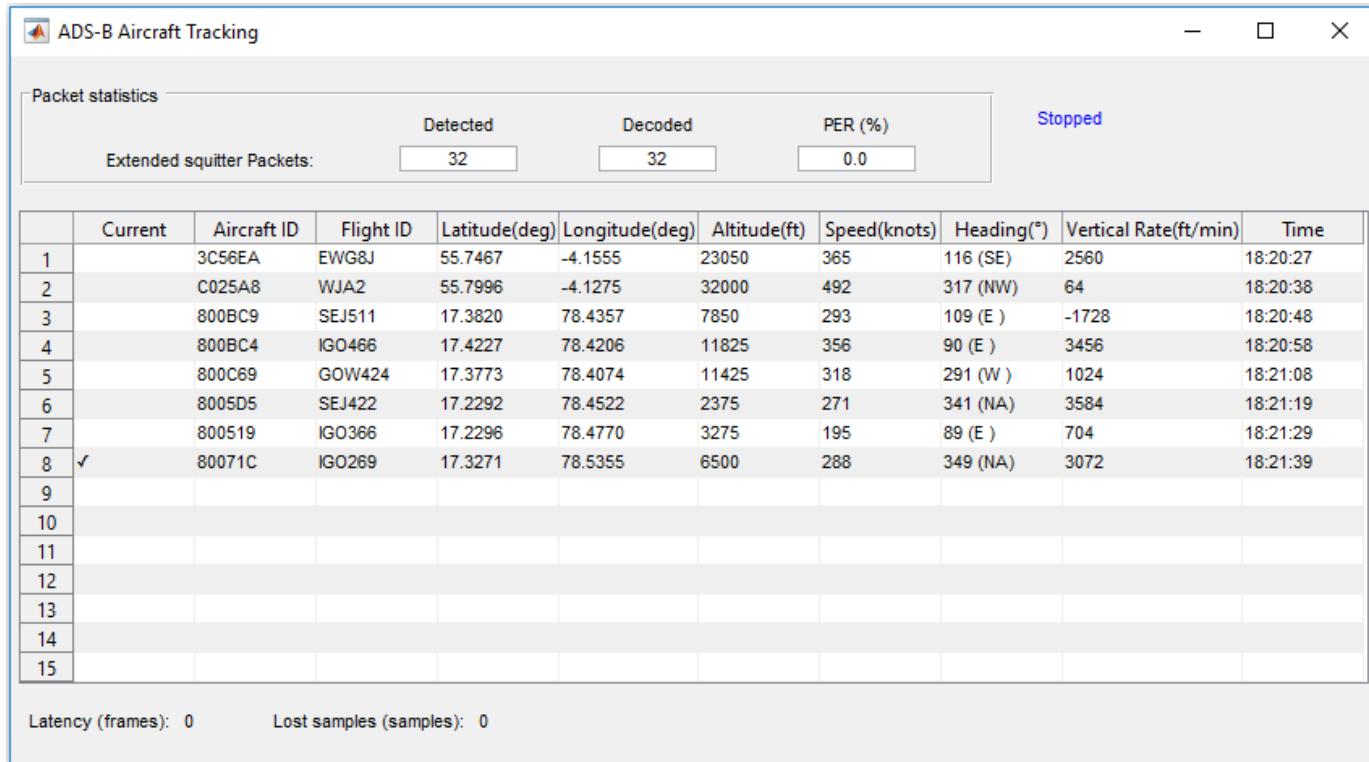
You can launch the map and start text file logging using the two slider switches (Launch Map and Data Logging).

**Launch Map** - Launch the map where the tracked flights can be viewed. **NOTE:** You must have a Mapping Toolbox™ license to use this feature.

**Data Logging** - Save the captured data in a TXT file. You can use the saved data for later for post processing.

### Results and Displays

The **HDLRx** subsystem demodulates and decodes the ADS-B data and the output is streamed through **Deserializer1D** block and **MapResults** MATLAB function, which produces hexadecimal output information about the aircraft. Each extended squitter Mode S packet contains partial information (any of Aircraft ID, Flight ID, Altitude, Speed, and Location) about the aircraft and the table is built up from multiple messages. The output is obtained as shown in the following diagram. The packet statistics include the number of detected packets, the number of correctly decoded packets, and packet error rate (PER). These aircraft details match the transmitted values from the “HW/SW Co-Design Implementation of ADS-B Transmitter/Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.



## HDL Code Generation and Synthesis Results

Pipeline registers have been added to the model to make sure that **HDLRx** subsystem does not have a long critical path. The HDL code generated from the **HDLRx** subsystem was synthesized using Xilinx® Vivado® on a **Zynq** FPGA with the device 7z045ffg900-2, and the design achieves **264.2 MHz** clock frequency, which is sufficient to decode the real-time ADS-B signals. The generated HDL code is tested and verified in the real-time example “HW/SW Co-Design Implementation of ADS-B Transmitter/Receiver Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio). To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license. The following table shows the synthesis results of this example.

Synthesis Frequency	264.2 MHz	
Resources	Used	Utilization in %
Slice LUTs	831	0.38
Slice Registers	1689	0.39
DSP48E1	2	0.22
F7 Muxes	24	0.02
F8 Muxes	8	0.01

You can use the commands `makehdl` and `makehdltb` to generate HDL code and a test bench for the **HDLRx** subsystem. To generate the HDL code, use the following command:

```
makehdl('commadsbrxhdl/HDLRx')
```

To generate a test bench, use the following command:

```
makehdltb('commadsbrxhdl/HDLRx')
```

## References

- 1 International Civil Aviation Organization, Annex 10, Volume 4. Surveillance and Collision Avoidance Systems.
- 2 Marvin E. Frerking, Digital Signal Processing in Communication Systems, Springer Science Business Media, New York, 1994.

# HDL Code Generation for Viterbi Decoder

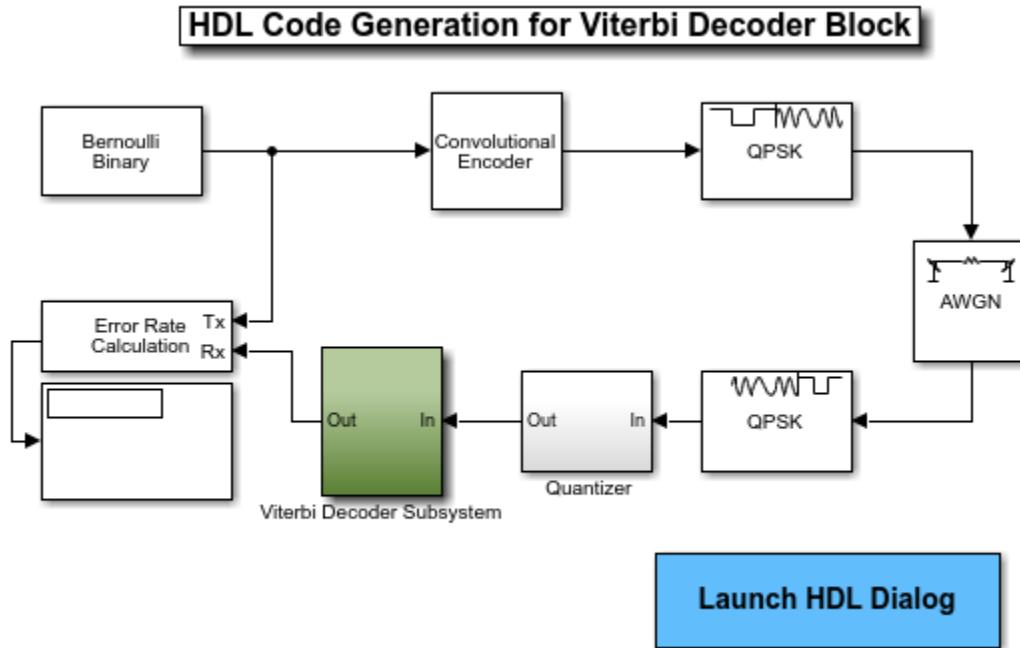
This example shows HDL code generation support for the Viterbi Decoder block. It shows how to check, generate, and verify the HDL code you generate from a fixed-point Viterbi Decoder model. This example also discusses the settings you can use to alter the HDL code you generate.

## Introduction

The model shows HDL code generation for a fixed-point Viterbi Decoder block used in soft decision convolutional decoding. To learn more about HDL support for Viterbi Decoder, refer to the documentation.

To open the model, run the following commands:

```
modelname = 'hdlcoder_commviterbi';
open_system(modelname);
```



Copyright 2014 The MathWorks, Inc.

In this model, the top-level subsystem *Viterbi Decoder Subsystem* contains the Viterbi Decoder block. To open this subsystem, run the following commands:

```
systemname = [modelname '/Viterbi Decoder Subsystem'];
open_system(systemname);
```



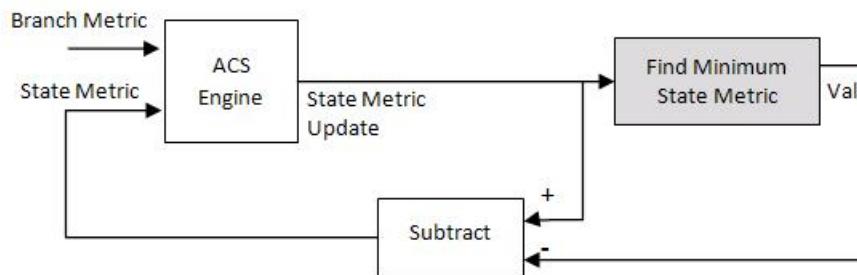
### The Viterbi Decoding Algorithm

There are three main components to the Viterbi decoding algorithm. They are the branch metric computation (BMC), add-compare-select (ACS), and traceback decoding. The following diagram illustrates the three units in the Viterbi decoding algorithm:

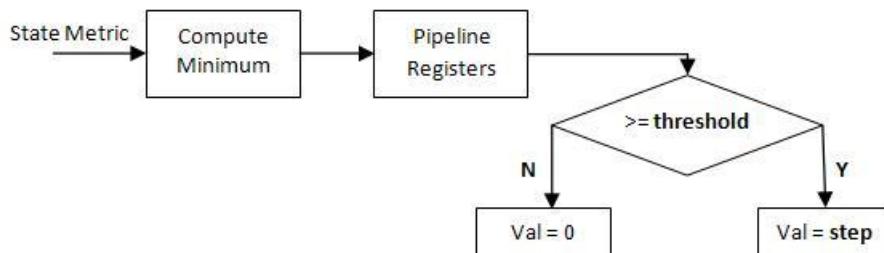


### The Renormalization Method

The Viterbi Decoder prevents the overflow of the state metrics in the ACS component by subtracting the minimum value of the state metrics at each time step, as shown in the following figure:



Obtaining the minimum value of all the state metric elements in one clock cycle results in a poor clock frequency for the circuit. The performance of the circuit may be improved by adding pipeline registers. However, simply subtracting the minimum value delayed by pipeline registers from the state metrics may still lead to overflow. The hardware architecture modifies the renormalization method and avoids the state metric overflow in three steps. First, the architecture calculates values for the threshold and step parameters, based on the trellis structure and the number of soft decision bits. Second, the delayed minimum value is compared to the threshold. Last, if the minimum value is greater than or equal to the threshold value, the implementation subtracts the step value from the state metric; otherwise no adjustment is performed. The following figure illustrates the modified renormalization method:



### Optimal State Metric Word Length Calculation

The hardware implementation calculates the optimal word length of the state metric and compares it with the value you specify for the block. The hardware architecture uses the optimal value if it is smaller than the one you specify. A message is displayed to show the value during HDL code

generation. If the calculated value is larger than the value you specify, an error message is reported and the optimal value is displayed.

Applying the calculated optimal state metric word length in the hardware implementation may significantly reduce the hardware resource if the value you specify is too large. For example, if you set 16 bits as the state metric word length but only 9 bits are required to achieve the same numerical results, applying the calculated optimal state metric word length in the hardware architecture saves approximately 40 percent of the register resources. The calculated optimal state metric word length for some typical trellises is displayed in the following table:

Decoding rate	Free distance	Example Trellis	Number of soft decision bits	Optimal word length
1/2	10	7, [171 133]	3	8
1/2	10	7, [171 133]	1	5
1/3	15	7, [171 133 165]	3	9
1/3	15	7, [171 133 165]	1	6
1/4	24	9,[463 535 733 745]	3	9
1/4	24	9,[463 535 733 745]	1	7

### Check and Generate Code for a Fixed-point Viterbi Model

This model decodes a DVB rate 1/2 , constraint length 7,(171,133) convolutional code with 3 bits soft decision. The decoder runs at continuous mode with the traceback depth of 32. The state metric word length is set to 16 bits. To validate the parameter settings of the Viterbi Decoder block, you can run the following commands:

- `workingdir = tempname;`
- `checkhdl(systemname,'TargetDirectory',workingdir);`

Running `checkhdl` generates messages that report:

- the default value of **TracebackStagesPerPipeline**. More information on this parameter can be found in the section **Pipelining the register-based traceback unit**,
- the state metric word length used in the HDL code compared with the one set on the block mask,
- the total delay introduced by the pipeline registers with respect to the original Viterbi block.

To generate HDL for the subsystem containing the Viterbi Decoder block, run the following commands: `workingdir = tempname; makehdl(systemname,'TargetDirectory',workingdir);`

The top level VHDL file name matches the name of the block in the model. The `Viterbi_Decoder` component generated in the `Viterbi_Decoder.vhd` contains three components: `BranchMetric`, `ACS`, and `Traceback`. The `ACS` and `Traceback` components instantiate components `ACSUnit` and `TracebackUnit` multiple times respectively. Data type definitions are included in the package file `Viterbi_Decoder_Subsystem_pkg.vhd`.

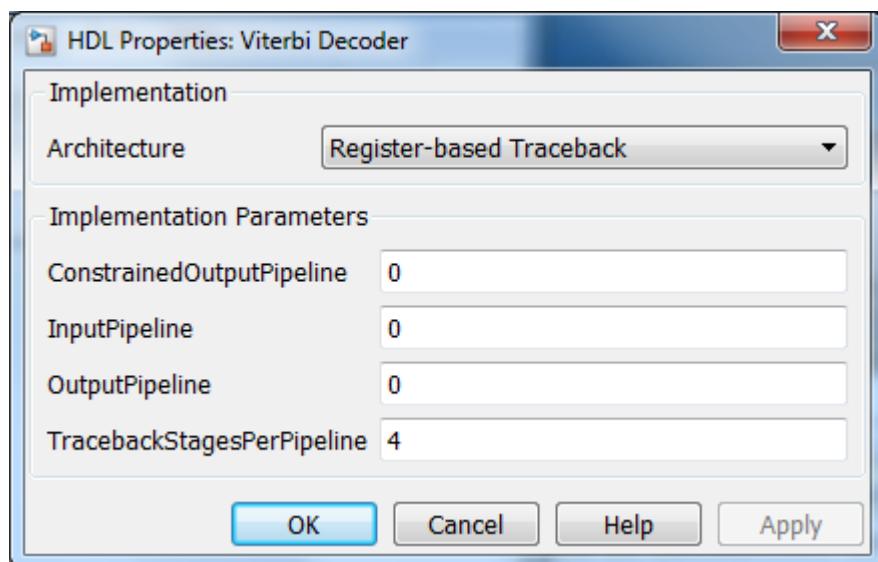
To generate a testbench for the subsystem containing the Viterbi Decoder block, run the following command: `makehdltb(systemname,'TargetDirectory',workingdir);`

### Optimization of The Traceback Unit

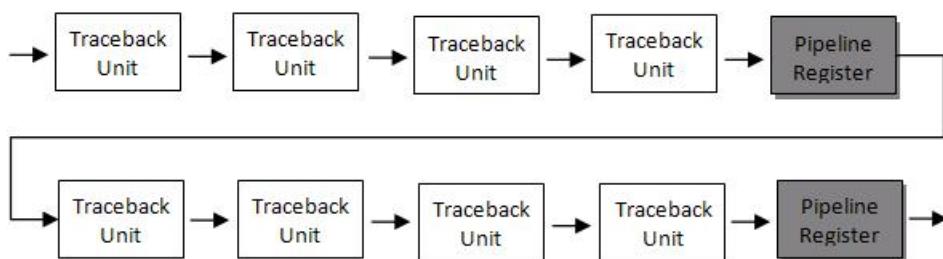
They are two methods to optimize the traceback unit: pipelining the register-based traceback or using the RAM-based traceback architecture.

- **Pipelining the register-based traceback unit**

The Viterbi Decoder block decodes every bit by tracing back through a traceback depth you define for the block. Because the block implements a complete traceback for each decision bit, registers are used to store the minimum state index and branch decision in the Traceback Decoding unit. This unit may be pipelined in order to improve the performance of the generated circuit. Pipeline registers can be added to the traceback unit by specifying the number of traceback stages per pipeline register. This can be done by setting the **TracebackStagesPerPipeline** implementation parameter for the Viterbi Decoder in the HDL block properties dialog. Right click the Viterbi Decoder block to navigate to the **HDL Block Properties** menu.



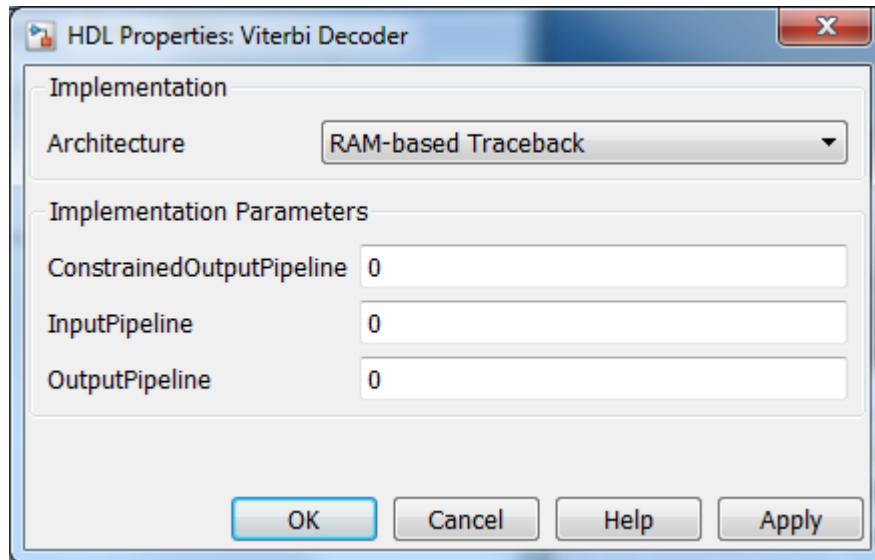
Setting the property value to 4 results in the insertion of a pipeline register for every four traceback units in the model, as illustrated in the following figure:



The **TracebackStagesPerPipeline** implementation parameter provides you a way of balancing the circuit performance based on system requirements. A smaller parameter value indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the number results in a lower usage of registers along with a decrease in the circuit speed. In our experiment with the rate 1/2, constraint length 7,(171,133) convolutional code, adjusting the **TracebackStagesPerPipeline** parameter from 4 to 8 reduces the pipeline register usage in half, with the circuit speed changing from 173MHz to 94 MHz.

- **RAM-based traceback**

Instead of using registers, you can choose to use RAMs to save the survivor branch information. This can be done by setting the HDL Architecture property of the Viterbi Decoder block to RAM-based Traceback.



There are two major differences between the register-based and the RAM-based traceback architectures.

*Firstly*, the register-based implementation combines the traceback and decode operations into one step and uses the best state found from the minimum operation as the decoding initial state. The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data.

*Secondly*, the register-based implementation decodes one bit after a complete traceback; while the RAM-based implementation traces back through  $M$  samples, decodes the previous  $M$  bits in reverse order, and releases one bit in order at each clock cycle.

Due to the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback. A longer traceback depth, for example, 10 times of constraint length, is recommended in the RAM-based traceback to achieve a similar bit error rate (BER) as the register-based implementation.

The size of RAM required for the implementation depends on the trellis and the traceback depth. The following table summarizes the RAM usage for some typical trellis structures.

Constraint length	Example trellis	Traceback depth	Memory size(bits)	Block RAMs
3	3,[5 7]	30	3x30x4	1
4	4,[15 17])	40	3x40x8	1
5	5,[37 27 33 25 35]	50	3x50x16	1
6	6,[ 73 75 55 65 47 57]	60	3x60x32	1
7	7,[171 133]	70	3x70x64	2
8	(8,[225 331 367])	80	3x80x128	4
9	9,[463 535 733 745]	90	3x90x256	8

Our experiment with the rate 1/2, constraint length 7, (171, 133) convolutional code shows that the RAM-based traceback unit uses 90% fewer registers than the register-based traceback unit (with pipelining every 4 stages) using similar clock constraints in synthesis. The two implementations provide a register-RAM tradeoff that can be tailored to the individual design.

### **Selected References**

- 1** Clark, G. C. Jr. and J. Bibb Cain., *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.
- 2** G. Feygin and P. G. Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," *IEEE Transactions on Communications*, vol. 41, no. 3, pp. 425-429, March 1993.

# Design Video Processing Algorithms for HDL in Simulink

This example shows how to design a hardware-targeted image filter using Vision HDL Toolbox™ blocks. It also uses Computer Vision Toolbox™ blocks.

The key features of a model for hardware-targeted video processing in Simulink® are:

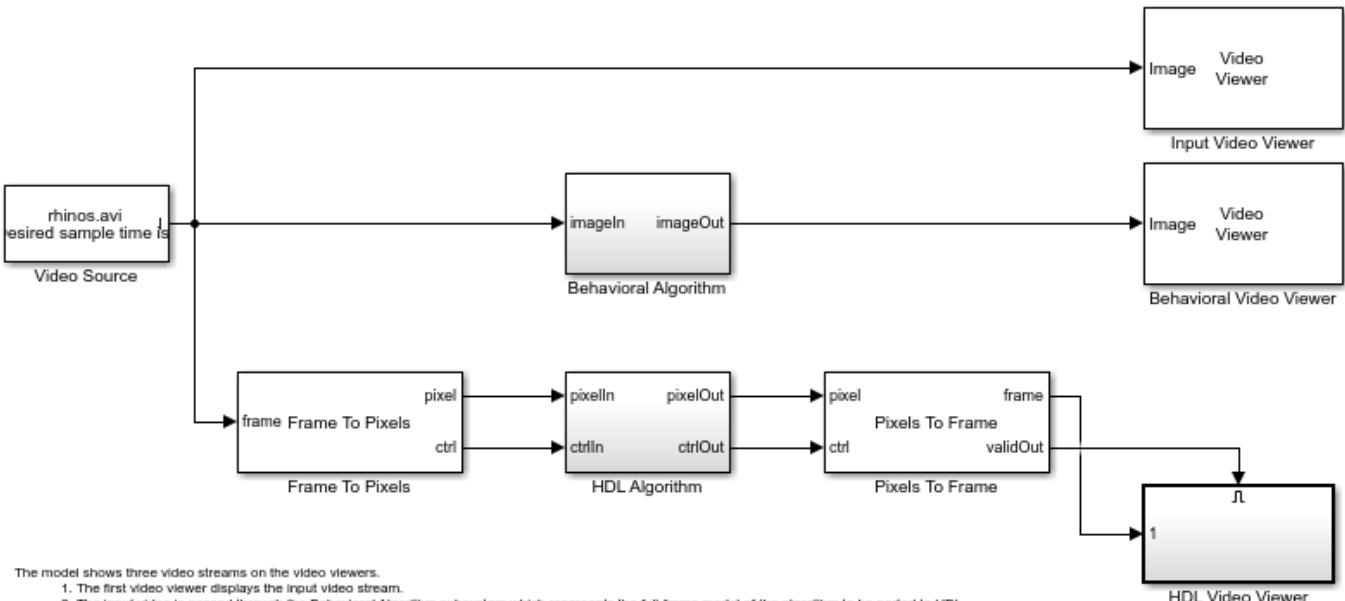
- **Streaming pixel interface:** Blocks in Vision HDL Toolbox use a streaming pixel interface. Serial processing is efficient for hardware designs, because less memory is required to store pixel data for computation. The serial interface allows the block to operate independently of image size and format and makes the design more resilient to video timing errors. For further information, see "Streaming Pixel Interface" (Vision HDL Toolbox).
- **Subsystem targeted for HDL code generation:** Design a hardware-friendly pixel-streaming video processing model by selecting blocks from the Vision HDL Toolbox libraries. The part of the design targeted for HDL code generation must be in a separate subsystem.
- **Conversion to frame-based video:** For verification, you can display frame-based video or compare the result of your hardware-compatible design with the output of a Simulink behavioral model. Vision HDL Toolbox provides a block that allows you to deserialize the output of your design.

## Open Model Template

This tutorial uses a Simulink model template to get started.

Click the Simulink button, or type `simulink` at the MATLAB® command prompt. On the Simulink start page, find the Vision HDL Toolbox section, and click the Basic Model template.

The template creates a new model that you can customize. Save the model with a new name.



The model shows three video streams on the video viewers.

1. The first video viewer displays the input video stream.
2. The input video is passed through the Behavioral Algorithm subsystem which represents the full-frame model of the algorithm to be ported to HDL.
3. On the third stream, the input video is converted to a streaming pixel format using the Frame to Pixels blocks, passed through the HDL Algorithm subsystem and then converted back to a frame using the Pixels to Frame block.
4. The model is configured for HDL code generation using the `hdlsetuptool` function.
5. The video format is defined by Model Simulation Callback Parameters (File -> Model Properties -> Model Properties -> Callbacks -> InitFcn).
6. To run this model, you must have a license for the Computer Vision Toolbox™.

You can

1. Add blocks to the Behavioral Algorithm and HDL algorithm subsystems.
2. Change the video format by changing the settings in the Video source, Frame To Pixels and Pixels To Frame blocks.
3. Generate HDL code for the HDL Algorithm subsystem by right-clicking on the subsystem -> HDL Coder -> Generate HDL for Subsystem.

### Import Data

The template includes a Video Source block that contains a 240p video sample. Each pixel is a scalar `uint8` value representing intensity. A best practice is to design and debug your design using a small frame size for quick debug cycles, before scaling up to larger image sizes. You can use this 240p source to debug a design targeted for 1080p video.

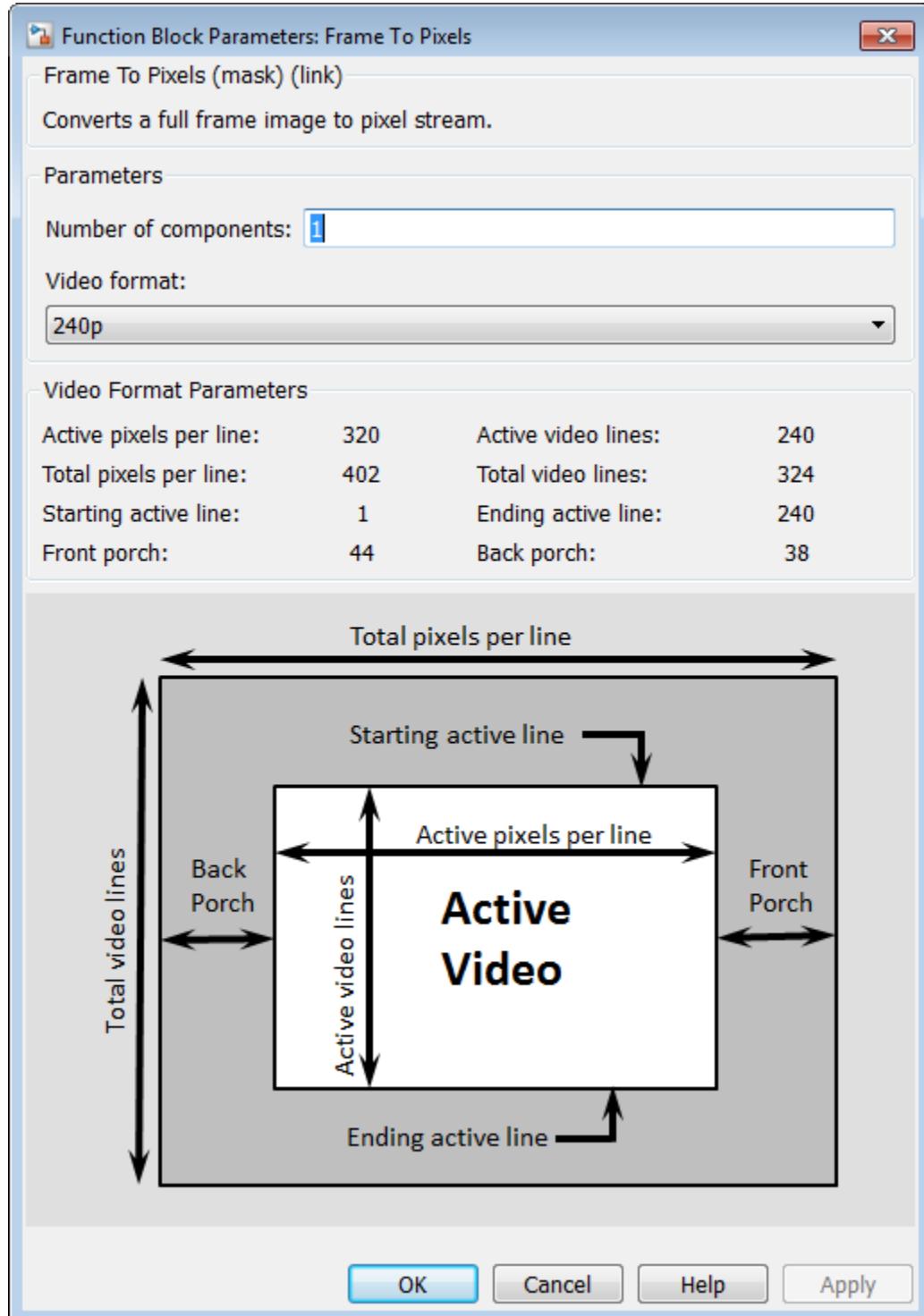
### Serialize Data

The Frame To Pixels block converts framed video to a stream of pixels and control structures. This block provides input for a subsystem targeted for HDL code generation, but it does not itself support HDL code generation.

The template includes an instance of this block. To simulate with a standard video format, choose a predefined video padding format to match your input source. To simulate with a custom-size image, choose the dimensions of the inactive regions that you want to surround the image with. This tutorial uses a standard video format.

Open the Frame To Pixels block dialog box to view the settings. The source video is in 240p grayscale format. A scalar integer represents the intensity value of each pixel. To match the input video, set **Number of components** to 1, and the **Video format** to 240p.

Note : The sample time of the video source must match the total number of pixels in the frame size you select in the Frame To Pixels block. Set the sample time to **Total pixels per line × Total lines**. In the `InitFcn` callback, the template creates a workspace variable, `totalPixels`, for the sample time of a 240p frame.



### Design HDL-Compatible Model

Design a subsystem targeted for HDL code generation, by modifying the HDL Algorithm subsystem. The subsystem input and output ports use the streaming pixel format described in the previous section. Open the HDL Algorithm subsystem to edit it.

In the Simulink Library Browser, click Vision HDL Toolbox. You can also open this library by typing `visionhdllib` at the MATLAB command prompt.

Select an image processing block. This example uses the Image Filter (Vision HDL Toolbox) block from the Filtering sublibrary. You can also access this library by typing `visionhdlfilter` at the MATLAB command prompt. Add the Image Filter block to the HDL Algorithm subsystem and connect the ports.



Open Image Filter block and make the following changes:

- Set **Filter coefficients** to `ones(4,4)/16` to implement a  $4 \times 4$  blur operation.
- Set **Padding method** to **Symmetric**.
- Set **Line buffer size** to a power of 2 that accommodates the active line size of the largest required frame format. This parameter does not affect simulation speed, so it does not need to be reduced when simulating with a small test image. The default, 2048, accommodates 1080p video format.
- On the **Data Types** tab, under **Data Type**, set **Coefficients** to `fixdt(0,1,4)`.

### Design Behavioral Model

You can visually or mathematically compare your HDL-targeted design with a behavioral model to verify the hardware design and monitor quantization error. The template includes a Behavioral Model subsystem with frame-based input and output ports for this purpose. Double-click on the Behavioral Model to edit it.

For this tutorial, add the 2-D FIR Filter (Computer Vision Toolbox) block from Computer Vision System Toolbox. This block filters the entire frame at once.

Open the 2-D FIR Filter block and make the following changes to match the configuration of the Image Filter block from Vision HDL Toolbox:

- Set **Coefficients** to `ones(4,4)/16` to implement a  $4 \times 4$  blur operation.
- Set **Padding options** to **Symmetric**.
- On the **Data Types** tab, under **Data Type**, set **Coefficients** to `fixdt(0,2,4)`.

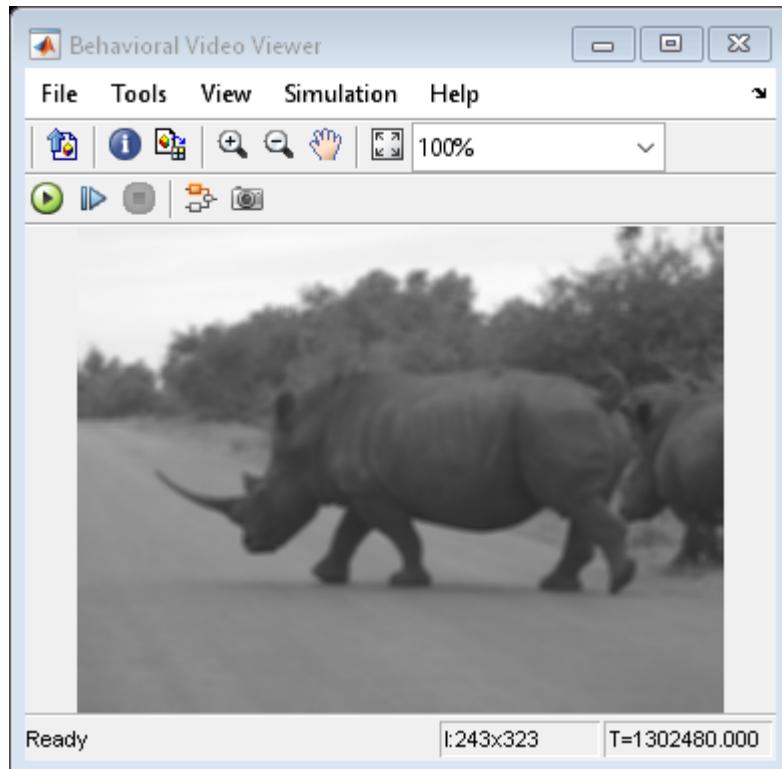
### Deserialize Filtered Pixel Stream

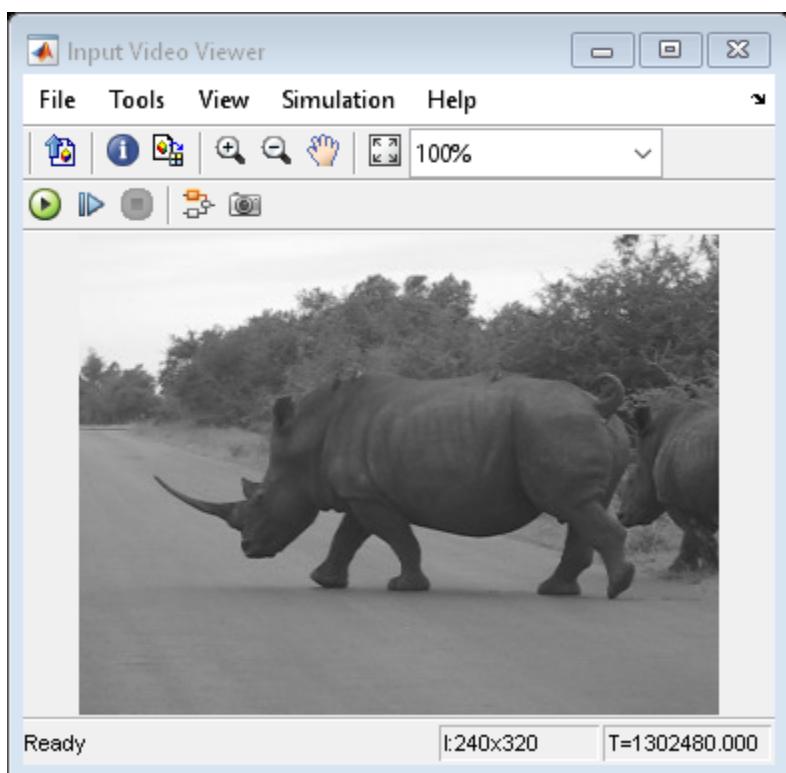
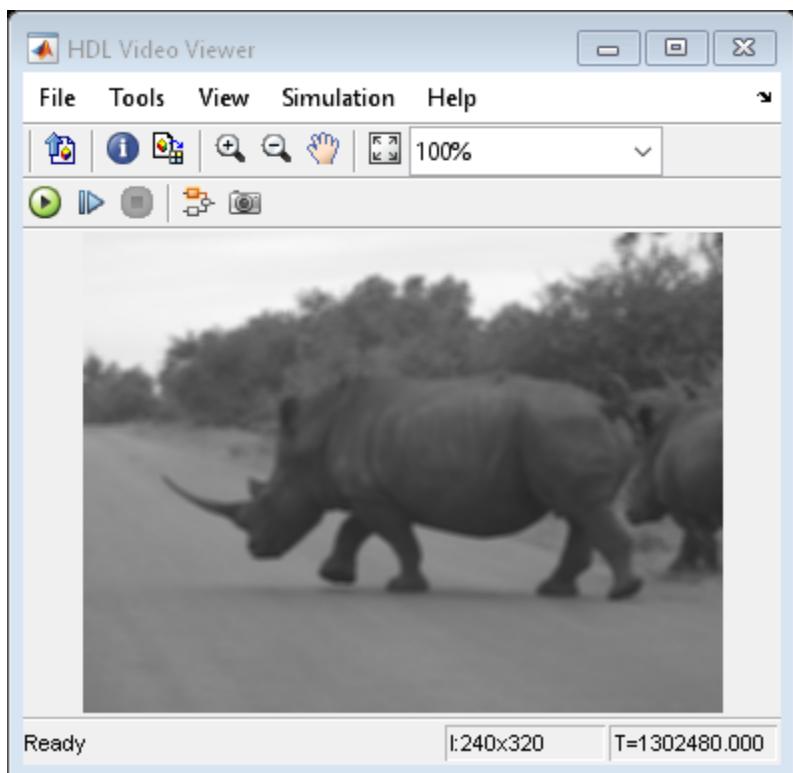
Use the Pixels To Frame block included in the template to deserialize the data for display.

Open the Pixels To Frame block. Set the image dimension properties to match the input video and the settings you specified in the Frame To Pixels block. For this tutorial, the **Number of components** is set to 1 and the **Video format** is set to 240p. The block converts the stream of output pixels and control signals back to a matrix representing a frame.

## Display Results and Compare to Behavioral Model

Use the Video Viewer blocks included in the template to compare the output frames visually. The `validOut` signal of the Pixels To Frame block is connected to the `Enable` port of the viewer. Run the model to display the results.





### Generate HDL Code

Once your design is working in simulation, you can use HDL Coder™ to generate HDL code for the HDL Algorithm subsystem. See “Generate HDL Code From Simulink” (Vision HDL Toolbox).

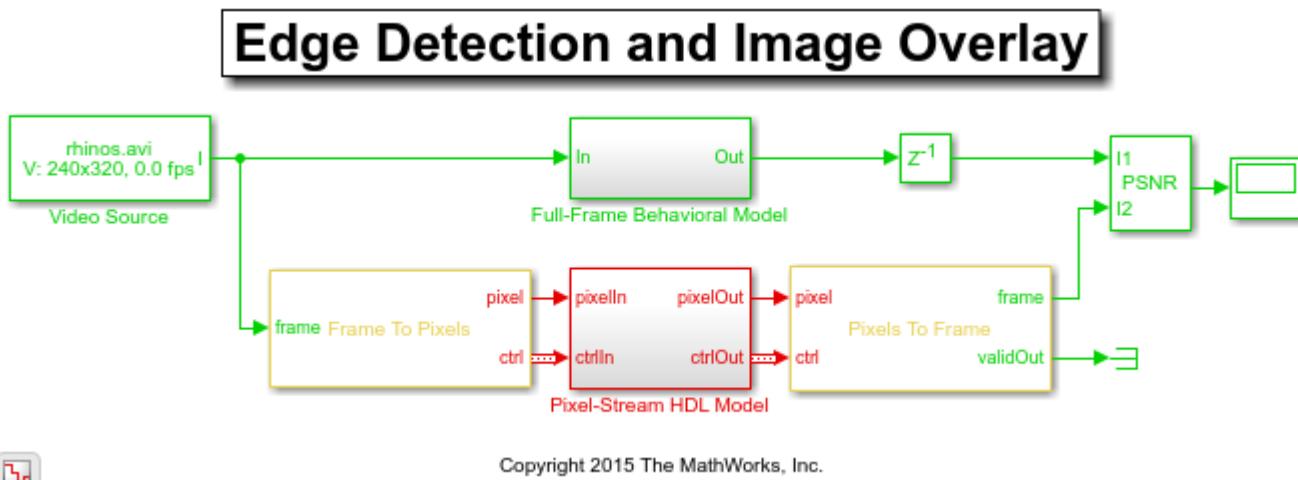
## Edge Detection and Image Overlay

This example shows how to detect and highlight object edges in a video stream. The behavior of the pixel-stream Sobel Edge Detector block, video stream alignment, and overlay, is verified by comparing the results with the same algorithm calculated by the full-frame blocks from the Computer Vision Toolbox™.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx™ Zynq™ reference design. See “Developing Vision Algorithms for Zynq-Based Hardware” (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

### Structure of the Example

The EdgeDetectionAndOverlayHDL.slx system is shown below.

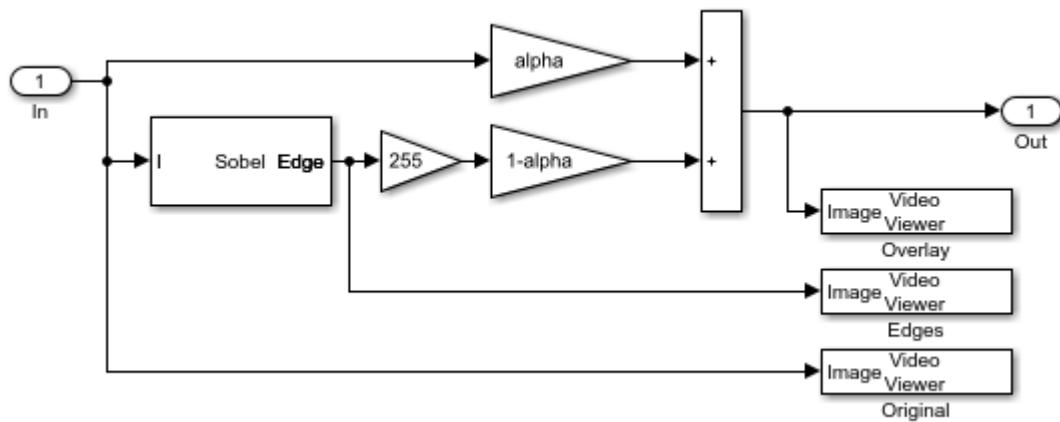


Copyright 2015 The MathWorks, Inc.

The difference in the color of the lines feeding the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model** subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition is because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate.

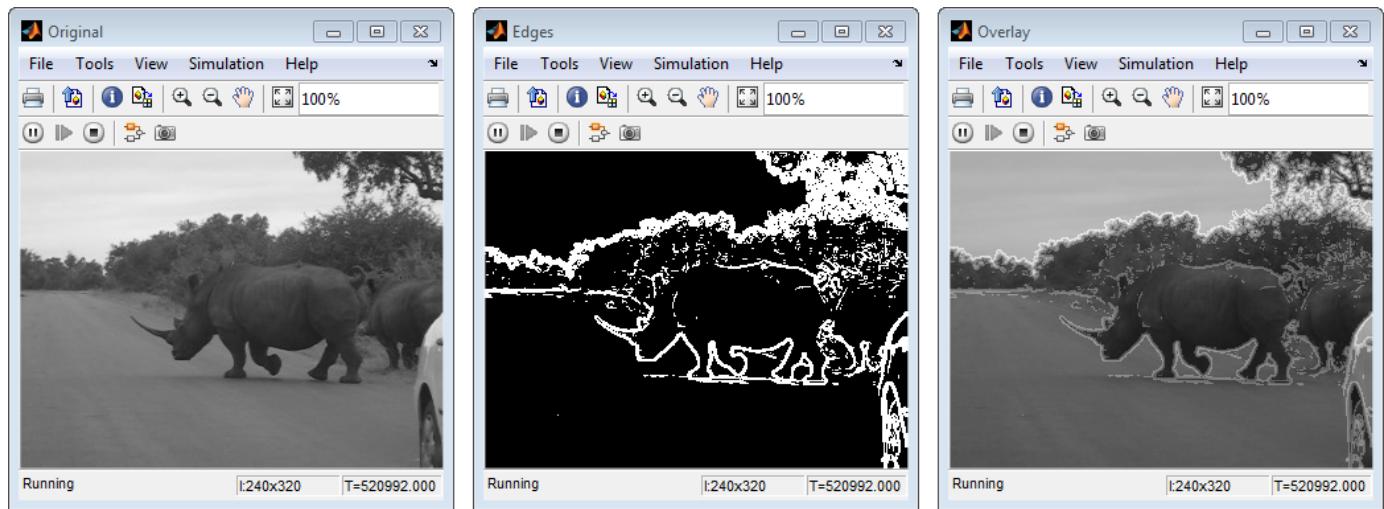
### Full-Frame Behavioral Model

The following diagram shows the structure of the **Full-Frame Behavioral Model** subsystem, which employs the frame-based **Edge Detection** block.



Given that the frame-based **Edge Detection** block does not introduce latency, image overlay is performed by weighting the source image and the **Edge Detection** output image, and adding them together in a straightforward manner.

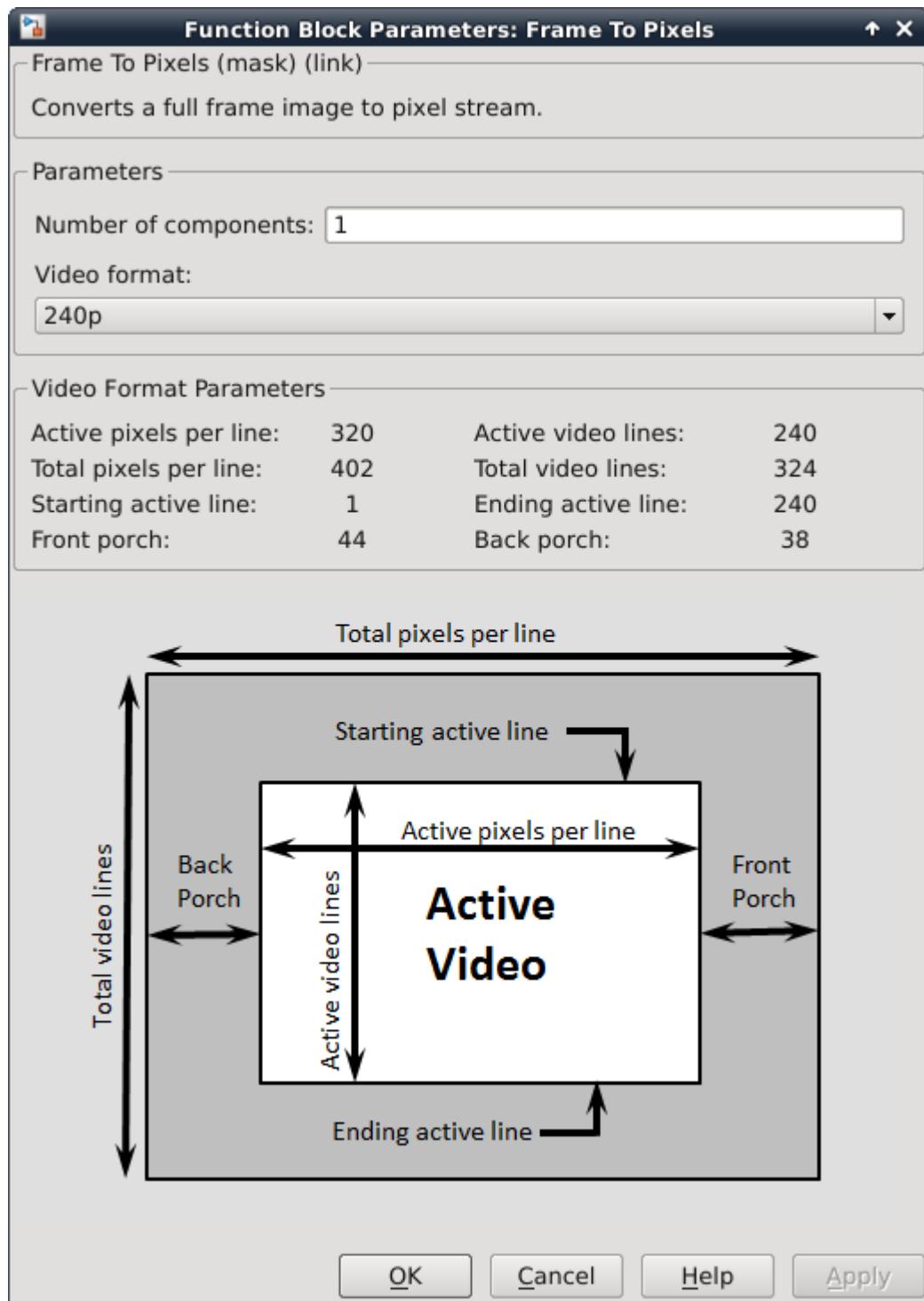
One frame of the source video, the edge detection result, and the overlaid image are shown from left to right in the diagram below.



It is a good practice to develop a behavioral system using blocks that process full image frames, the **Full-Frame Behavioral Model** subsystem in this example, before moving forward to working on an FPGA-targeting design. Such a behavioral model helps verify the video processing design. Later on, it can serve as a reference for verifying the implementation of the algorithm targeted to an FPGA. Specifically, the **PSNR** (peak signal-to-noise ratio) block at the top level of the model compares the results from full-frame processing with those from pixel-stream processing.

### Frame To Pixels: Generating a Pixel Stream

The task of the **Frame To Pixels** is to convert a full frame image to pixel stream. To simulate the effect of horizontal and vertical blanking periods found in real life hardware video systems, the active image is augmented with non-image data. For more information on the streaming pixel protocol, see "Streaming Pixel Interface" (Vision HDL Toolbox). The **Frame To Pixels** block is configured as shown:



The **Number of components** field is set to 1 for grayscale image input, and the **Video format** field is 240p to match that of the video source.

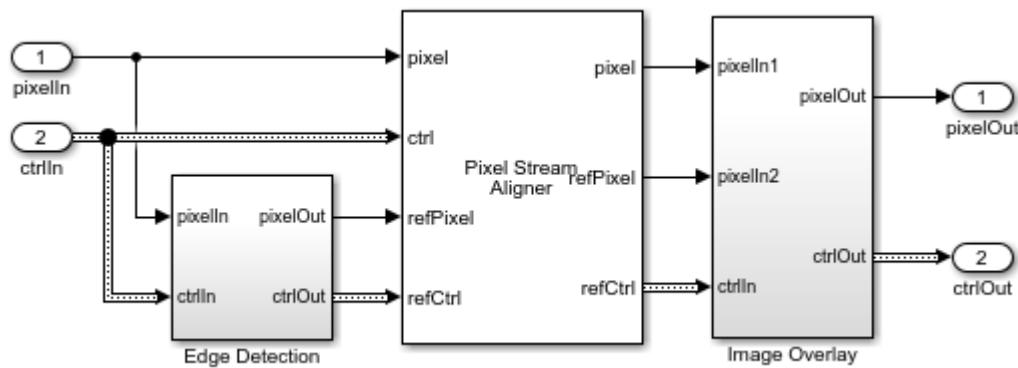
In this example, the Active Video region corresponds to the 240x320 matrix of the dark image from the upstream **Corruption** block. Six other parameters, namely, **Total pixels per line**, **Total video**

**lines**, **Starting active line**, **Ending active line**, **Front porch**, and **Back porch** specify how many non-image data will be augmented on the four sides of the Active Video. For more information, see the Frame To Pixels (Vision HDL Toolbox) block reference page.

Note that the sample time of the **Video Source** is determined by the product of **Total pixels per line** and **Total video lines**.

### Pixel-Stream Edge Detection and Image Overlay

The **Pixel-Stream HDL Model** subsystem is shown in the diagram below. You can generate HDL code from this subsystem.



Due to the nature of pixel-stream processing, unlike the **Edge Detection** block in the **Full-Frame Behavioral Model**, the **Edge Detector** block from the Vision HDL Toolbox™ will introduce latency. The latency prevents us from directly weighting and adding two images to obtain the overlaid image. To address this issue, the **Pixel Stream Aligner** block is used to synchronize the two pixel streams before the sum.

To properly use this block, refPixel and refCtrl must be connected to the pixel and control bus that are associated with a delayed pixel stream. In our example, due to the latency introduced by the **Edge Detector**, the pixel stream coming out of the **Edge Detector** is delayed with respect to that feeding into it. Therefore, the upstream source of refPixel and refCtrl are the Edge and ctrl output of the **Edge Detector**.

### Pixels To Frame: Converting Pixel Stream Back to Full Frame

As a companion to **Frame To Pixels** that converts a full image frame to pixel stream, the **Pixels To Frame** block, reversely, converts the pixel stream back to the full frame by making use of the synchronization signals. Since the output of the **Pixels To Frame** block is a 2-D matrix of a full image, there is no need to further carry on the bus containing five synchronization signals.

The **Number of components** field and the **Video format** fields of both Frame To Pixels and Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

### Verifying the Pixel Stream Processing Design

While building the streaming portion of the design, the **PSNR** block continuously verifies results against the original full-frame design. The **Delay** block on the top level of the model time-aligns the 2-D matrices for a fair comparison. During the course of the simulation, the **PSNR** block should give **inf** output, indicating that the output image from the **Full-Frame Behavioral Model** matches the image generated from the stream processing **Pixel-Stream HDL Model**.

## Exploring the Example

The example allows you to experiment with different threshold and alpha values to examine their effect on the quality of the overlaid images. Specifically, two workspace variables *thresholdValue* and *alpha* with initial values 7 and 0.8, respectively, are created upon opening the model. You can modify their values using the MATLAB command line as follows:

```
thresholdValue=8  
alpha=0.5
```

The updated *thresholdValue* will be propagated to the **Threshold** field of the **Edge Detection** block inside the **Full-Frame Behavioral Model** and the **Edge Detector** block inside **Pixel-Stream HDL Model/Edge Detection**. The *alpha* value will be propagated to the **Gain1** block in the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model/Image Overlay**, and the value of  $1 - \alpha$  goes to **Gain2** blocks. Closing the model clears both variables from your workspace.

In this example, the valid range of *thresholdValue* is between 0 and 256, inclusive. Setting *thresholdValue* equal to or greater than 257 triggers a message **Parameter overflow occurred for 'threshold'**. The higher you set the *thresholdValue*, the smaller the amount of edges the example finds in the video.

The valid range of *alpha* is between 0 and 1, inclusive. It determines the weights for edge detection output image and the original source image before adding them. The overlay operation is a linear interpolation according to the following formula.

$$\text{overlaid image} = \alpha * \text{source image} + (1 - \alpha) * \text{edge image}.$$

Therefore, when *alpha* = 0, the overlaid image is the edge detection output, and when *alpha* = 1 it becomes the source image.

## Generate HDL Code and Verify Its Behavior

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

To generate a test bench, use the following command:

```
makehdltb('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

# Lane Detection

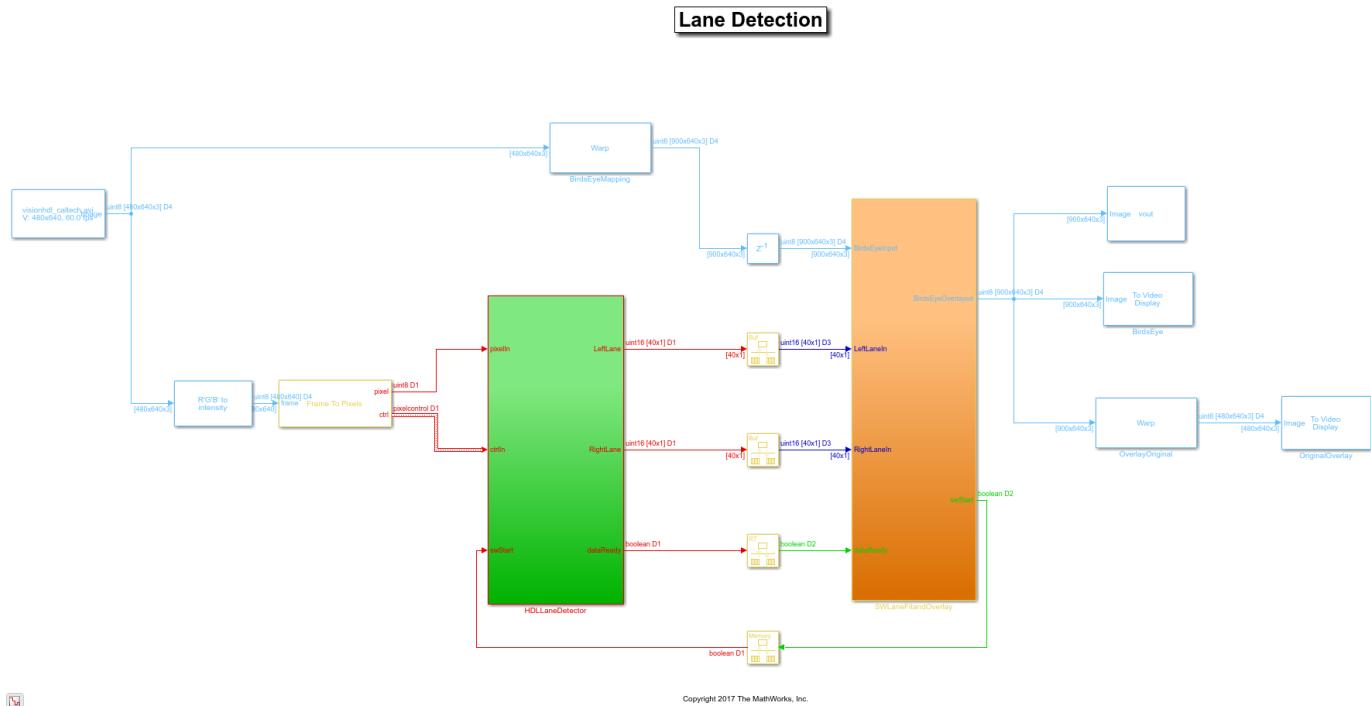
This example shows how to implement a lane-marking detection algorithm for FPGAs.

Lane detection is a critical processing stage in Advanced Driving Assistance Systems (ADAS). Automatically detecting lane boundaries from a video stream is computationally challenging and therefore hardware accelerators such as FPGAs and GPUs are often required to achieve real time performance.

In this example model, an FPGA-based lane candidate generator is coupled with a software-based polynomial fitting engine, to determine lane boundaries.

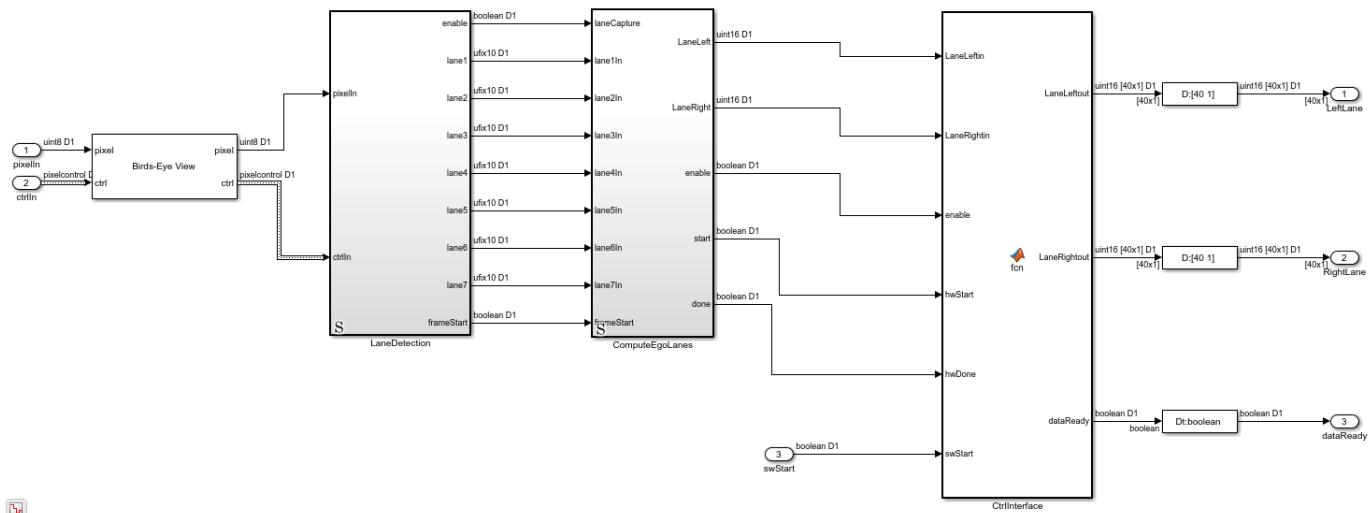
## System Overview

The LaneDetectionHDL.slx system is shown below. The HDLLaneDetector subsystem represents the hardware accelerated part of the design, while the SWLaneFitandOverlay subsystem represent the software based polynomial fitting engine. Prior to the Frame to Pixels block, the RGB input is converted to intensity color space.



## HDL Lane Detector

The HDL Lane Detector represents the hardware-accelerated part of the design. This subsystem receives the input pixel stream from the front-facing camera source, transforms the view to obtain the birds-eye view, locates lane marking candidates from the transformed view and then buffers them up into a vector to send to the software side for curve fitting and overlay.



## Birds-Eye View

The Birds-Eye View block transforms the front-facing camera view to a birds-eye perspective. Working with the images in this view simplifies the processing requirements of the downstream lane detection algorithms. The front-facing view suffers from perspective distortion, causing the lanes to converge at the vanishing point. The first stage of the system corrects the perspective distortion by transforming to the birds-eye view.

The Inverse Perspective Mapping is given by the following expression:

$$(\hat{x}, \hat{y}) = \text{round} \left( \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \right)$$

The homography matrix,  $\mathbf{h}$ , is derived from four intrinsic parameters of the physical camera setup, namely the focal length, pitch, height and principle point (from a pinhole camera model). Please refer to Computer Vision System Toolbox™ documentation for further details.

Direct evaluation of the source (front-facing) to destination (birds-eye) mapping in real time on FPGA/ASIC hardware is challenging. The requirement for division along with the potential for non-sequential memory access from a frame buffer mean that the computational requirements of this part of the design are substantial. Therefore instead of directly evaluating the IPM calculation in real time, an offline analysis of the input to output mapping has been performed and used to pre-compute a mapping scheme. This is possible as the homography matrix is fixed after factory calibration/installation of the camera, due to the camera position, height and pitch being fixed.

In this particular example, the birds-eye output image is a frame of [700x640] dimensions, whereas the front-facing input image is of [480x640] dimensions. There is not sufficient blanking available in order to output the full birds-eye frame before the next front-facing camera input is streamed in. Internally, the Birds-Eye view block will therefore lock up when it begins to process a new input frame, and will not accept new frame data until it has finished outputting the current birds-eye frame.

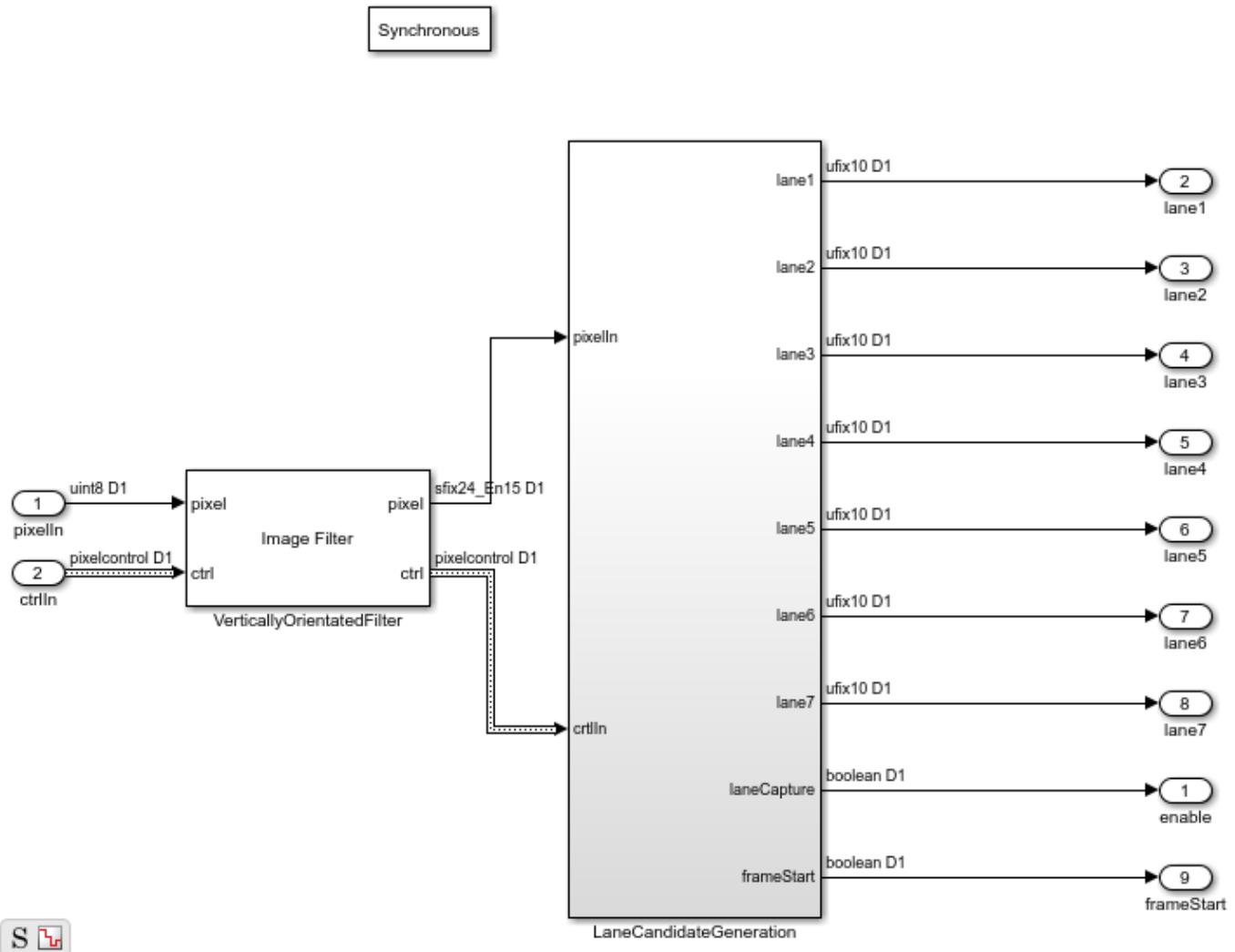
## Line Buffering and Address Computation

A full sized projective transformation from input to output would result in a [900x640] output image. This requires that the full [480x640] input image is stored in memory, while the source pixel location is calculated using the source location and homography matrix. Ideally on-chip memory should be

used for this purpose, removing the requirement for an off-chip frame buffer. Analysis of the mapping of input line to output line reveals that in order to generate the first 700 lines of the top down birds eye output image, around 50 lines of the input image are required. This is an acceptable number of lines to store using on-chip memory.

## Lane Detection

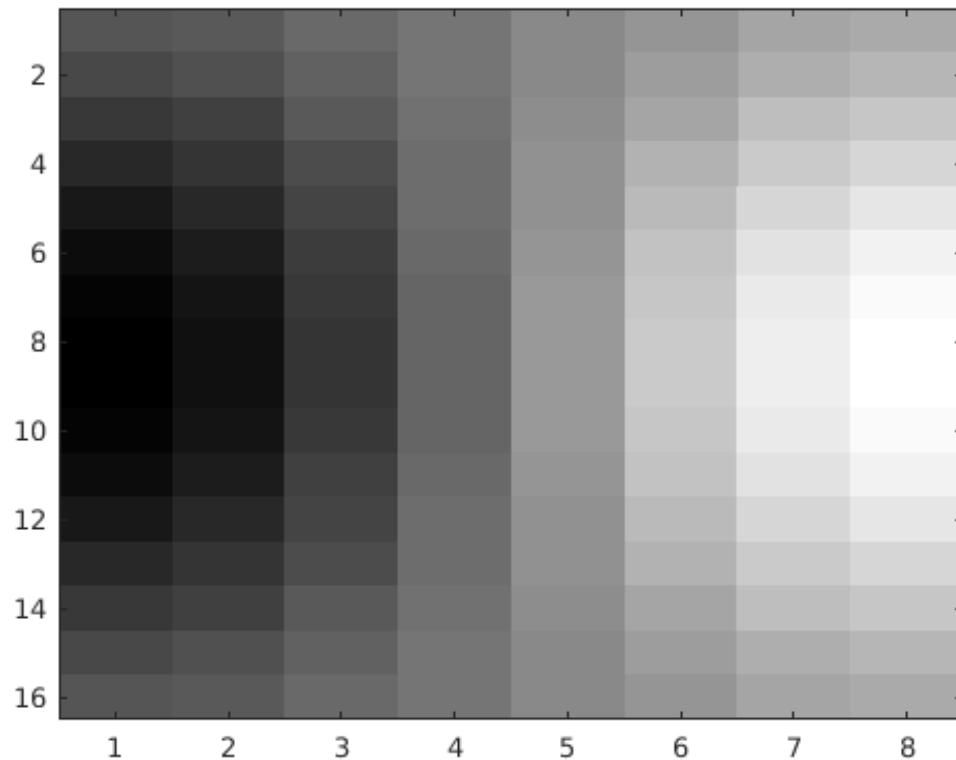
With the birds-eye view image obtained, the actual lane detection can be performed. There are many techniques which can be considered for this purpose. To achieve an implementation which is robust, works well on streaming image data and which can be implemented in FPGA/ASIC hardware at reasonable resource cost, this example uses the approach described in [1]. This algorithm performs a full image convolution with a vertically oriented first order Gaussian derivative filter kernel, followed by sub-region processing.



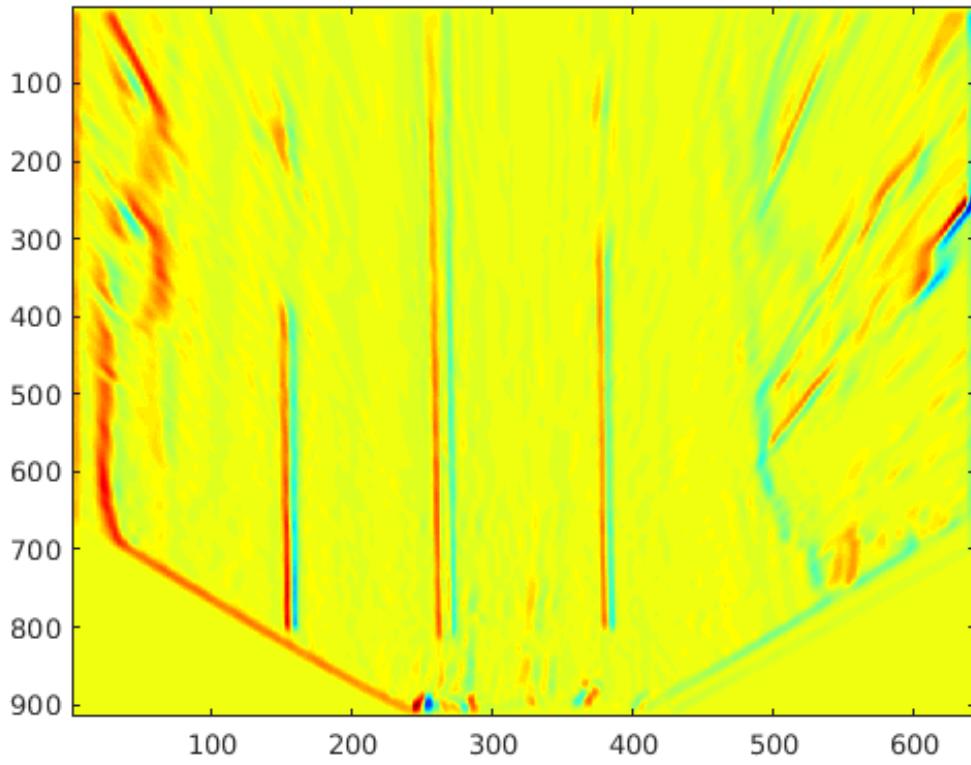
### Vertically Oriented Filter Convolution

Immediately following the birds-eye mapping of the input image, the output is convolved with a filter designed to locate strips of high intensity pixels on a dark background. The width of the kernel is 8

pixels, which relates to the width of the lines that appear in the birds-eye image. The height is set to 16 which relates to the size of the dashed lane markings which appear in the image. As the birds-eye image is physically related to the height, pitch etc. of the camera, the width at which lanes appear in this image is intrinsically related to the physical measurement on the road. The width and height of the kernel may need to be updated when operating the lane detection system in different countries.

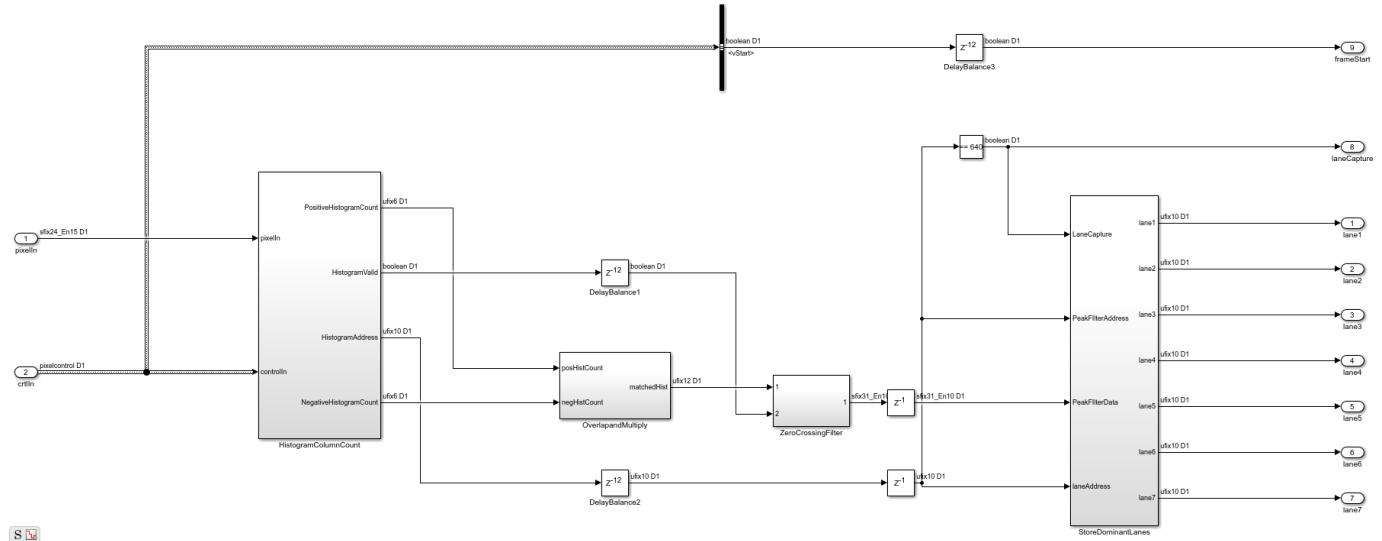


The output of the filter kernel is shown below, using jet colormap to highlight differences in intensity. Because the filter kernel is a general, vertically oriented Gaussian derivative, there is some response from many different regions. However, for the locations where a lane marking is present, there is a strong positive response located next to a strong negative response, which is consistent across columns. This characteristic of the filter output is used in the next stage of the detection algorithm to locate valid lane candidates.



## Lane Candidate Generation

After convolution with the Gaussian derivative kernel, sub-region processing of the output is performed in order to find the coordinates where a lane marking is present. Each region consists of 18 lines, with a ping-pong memory scheme in place to ensure that data can be continuously streamed through the subsystem.



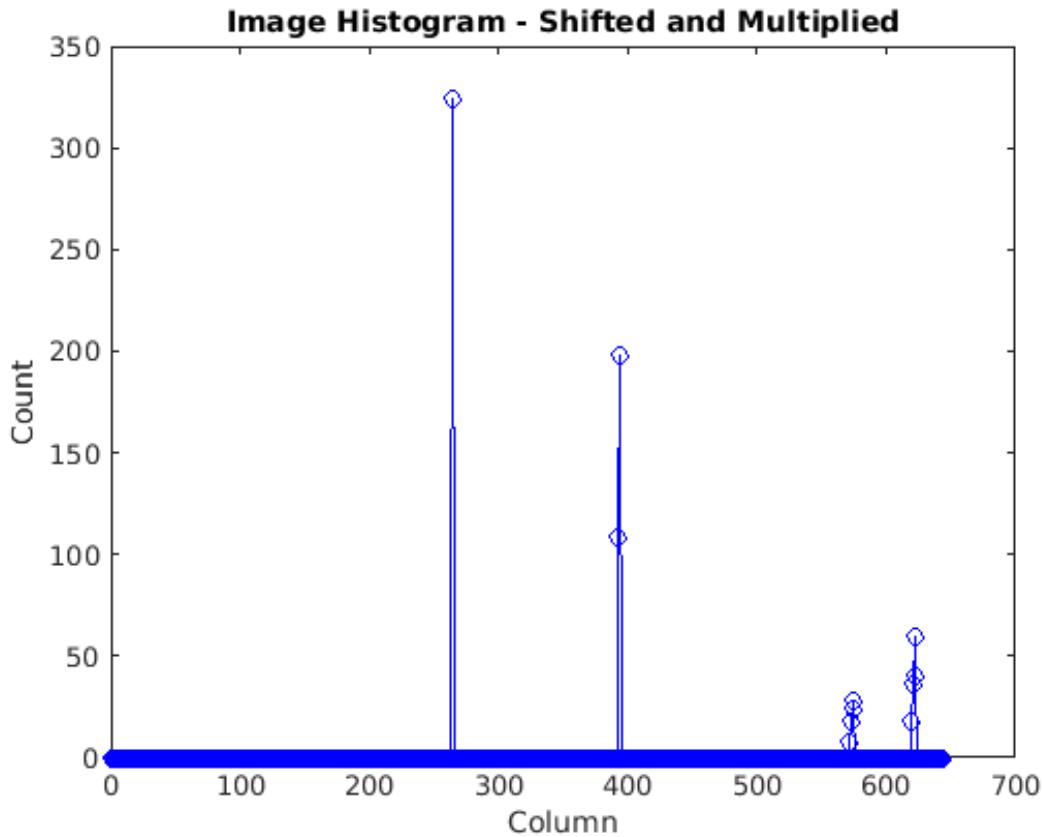
### Histogram Column Count

Firstly, HistogramColumnCount counts the number of thresholded pixels in each column over the 18 line region. A high column count indicates that a lane is likely present in the region. This count is performed for both the positive and the negative thresholded images. The positive histogram counts are offset to account for the kernel width. Lane candidates occur where the positive count and negative counts are both high. This exploits the previously noted property of the convolution output where positive tracks appear next to negative tracks.

Internally, the column counting histogram generates the control signalling that selects an 18 line region, computes the column histogram, and outputs the result when ready. A ping-pong buffering scheme is in place which allows one histogram to be reading while the next is writing.

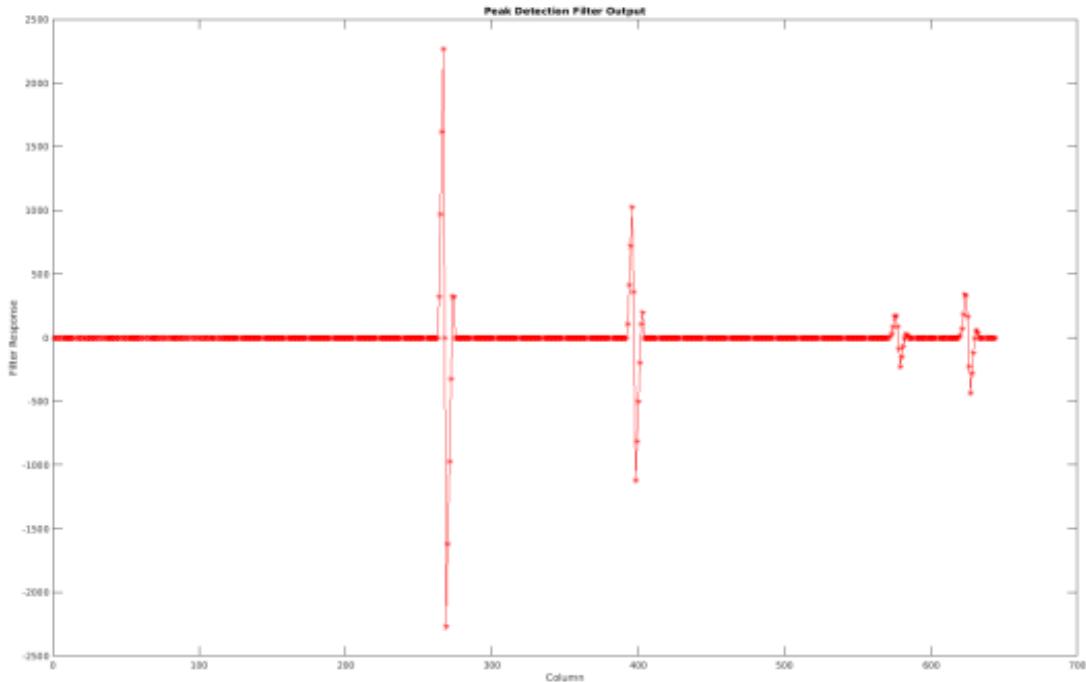
### Overlap and Multiply

As noted, when a lane is present in the birds-eye image, the convolution result will produce strips of high-intensity positive output located next to strips of high-intensity negative output. The positive and negative column count histograms locate such regions. In order to amplify these locations, the positive count output is delayed by 8 clock cycles (an intrinsic parameter related to the kernel width), and the positive and negative counts are multiplied together. This amplifies columns where the positive and negative counts are in agreement, and minimizes regions where there is disagreement between the positive and negative counts. The design is pipelined in order to ensure high throughput operation.



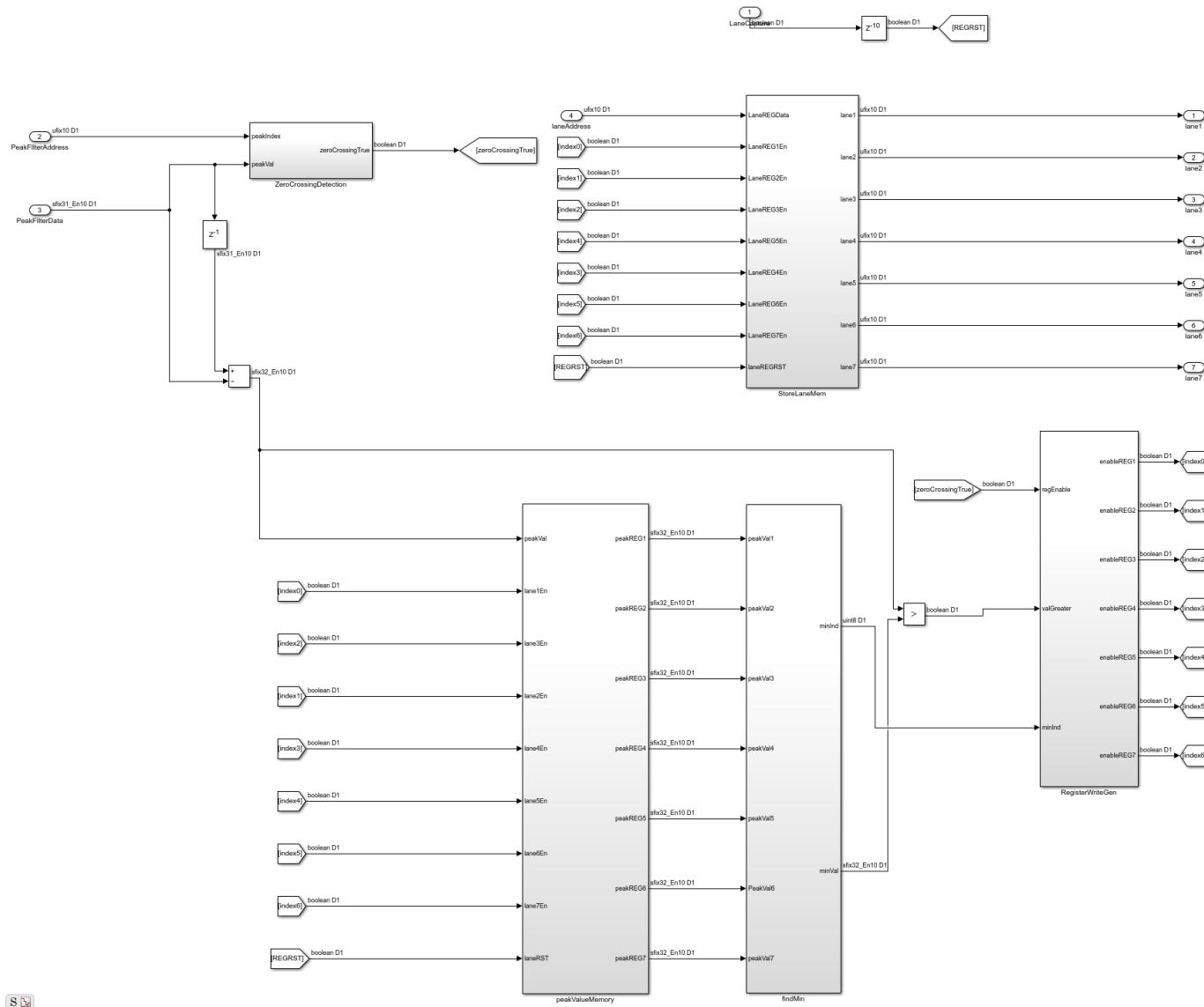
## Zero Crossing Filter

At the output of the Overlap and Multiply subsystem, peaks appear where there are lane markings present. A peak detection algorithm determines the columns where lane markings are present. Because the SNR is relatively high in the data, this example uses a simple FIR filtering operation followed by zero crossing detection. The Zero Crossing Filter is implemented using the Discrete FIR Filter block from DSP System Toolbox™. It is pipelined for high-throughput operation.



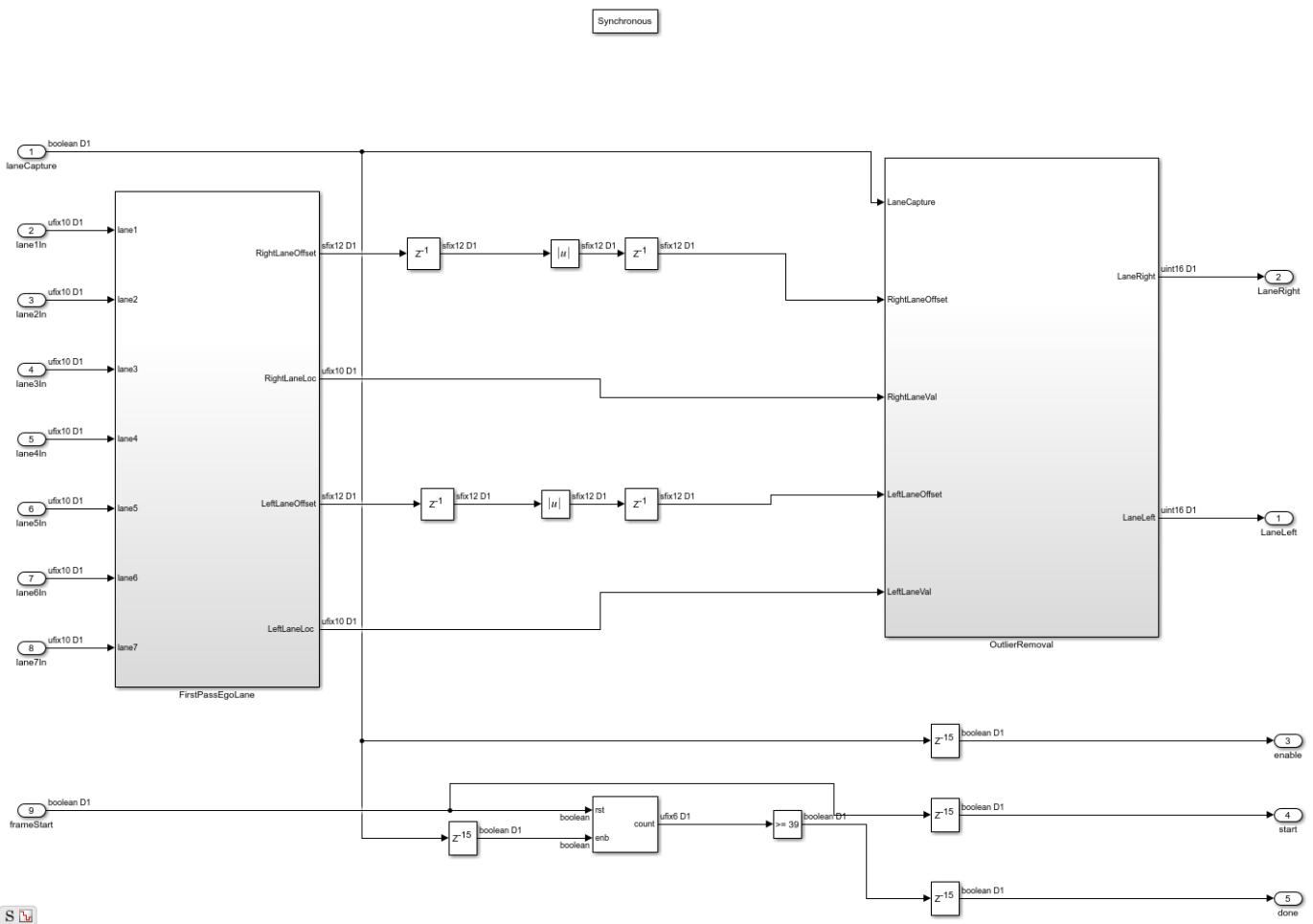
## Store Dominant Lanes

The zero crossing filter output is then passed into the Store Dominant Lanes subsystem. This subsystem has a maximum memory of 7 entries, and is reset every time a new batch of 18 lines is reached. Therefore, for each sub-region 7 potential lane candidates are generated. In this subsystem, the Zero Crossing Filter output is streamed through, and examined for potential zero crossings. If a zero crossing does occur, then the difference between the address immediately prior to zero crossing and the address after zero crossing is taken in order to get a measurement of the size of the peak. The subsystem stores the zero crossing locations with the highest magnitude.



## Compute Ego Lanes

The Lane Detection subsystem outputs the 7 most viable lane markings. In many applications, we are most interested in the lane markings that contain the lane in which the vehicle is driving. By computing the so called "Ego-Lanes" on the hardware side of the design, we can reduce the memory bandwidth between hardware and software, by sending 2 lanes rather than 7 to the processor. The Ego-Lane computation is split into two subsystems. The FirstPassEgoLane subsystem assumes that the centre column of the image corresponds to the middle of the lane, when the vehicle is correctly operating within the lane boundaries. The lane candidates which are closest to the center are therefore assumed as the ego lanes. The Outlier Removal subsystem maintains an average width of the distance from lane markings to centre coordinate. Lane markers which are not within tolerance of the current width are rejected. Performing early rejection of lane markers gives better results when performing curve fitting later on in the design.

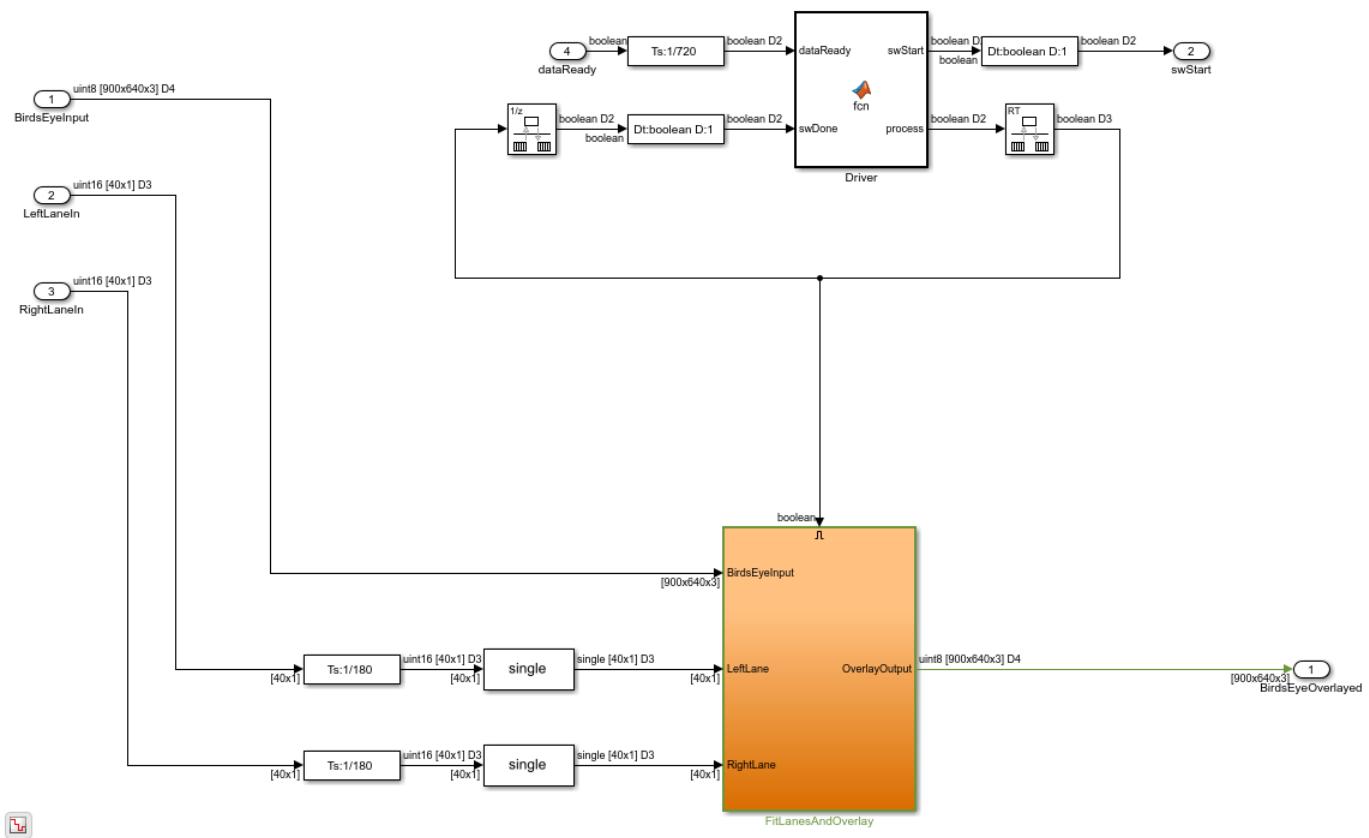


## Control Interface

Finally, the computed ego lanes are sent to the CtrlInterface MATLAB function subsystem. This state machine uses the four control signal inputs - enable, hwStart, hwDone, and swStart to determine when to start buffering, accept new lane coordinate into the 40x1 buffer and finally indicate to the software that all 40 lane coordinates have been buffered and so the lane fitting and overlay can be performed. The dataReady signal ensures that software will not attempt lane fitting until all 40 coordinates have been buffered, while the swStart signal ensures that the current set of 40 coordinates will be held until lane fitting is completed.

## Software Lane Fit and Overlay

The detected ego-lanes are then passed to the SW Lane Fit and Overlay subsystem, where robust curve fitting and overlay is performed. Recall that the birds-eye output is produced once every two frames or so rather than on every consecutive frame. The curve fitting and overlay is therefore placed in an enabled subsystem, which is only enabled when new ego lanes are produced.



## Driver

The Driver MATLAB Function subsystem controls the synchronization between hardware and software. Initially it is in a polling state, where it samples the **dataReady** input at regular intervals per frame to determine when hardware has buffered a full  $[40 \times 1]$  vector of lane coordinates. Once this occurs, it transitions into software processing state where **swStart** and **process** outputs are held high. The Driver remains in the software processing state until **swDone** input is high. Seeing as the process output loops back to **swDone** input with a rate transition block in between, there is effectively a constant time budget specified for the **FitLanesandOverlay** subsystem to perform the fitting and overlay. When **swDone** is high, the Driver will transition into a synchronization state, where **swStart** is held low to indicate to hardware that processing is complete. The synchronization between software and hardware is such that hardware will hold the  $[40 \times 1]$  vector of lane coordinates until the **swStart** signal transitions back to low. When this occurs, **dataReady** output of hardware will then transition back to low.

## Fit Lanes and Overlay

The Fit Lanes and Overlay subsystem is enabled by the Driver. It performs the necessary arithmetic required in order to fit a polynomial onto the lane coordinate data received at input, and then draws the fitted lane and lane coordinates onto the Birds-Eye image.

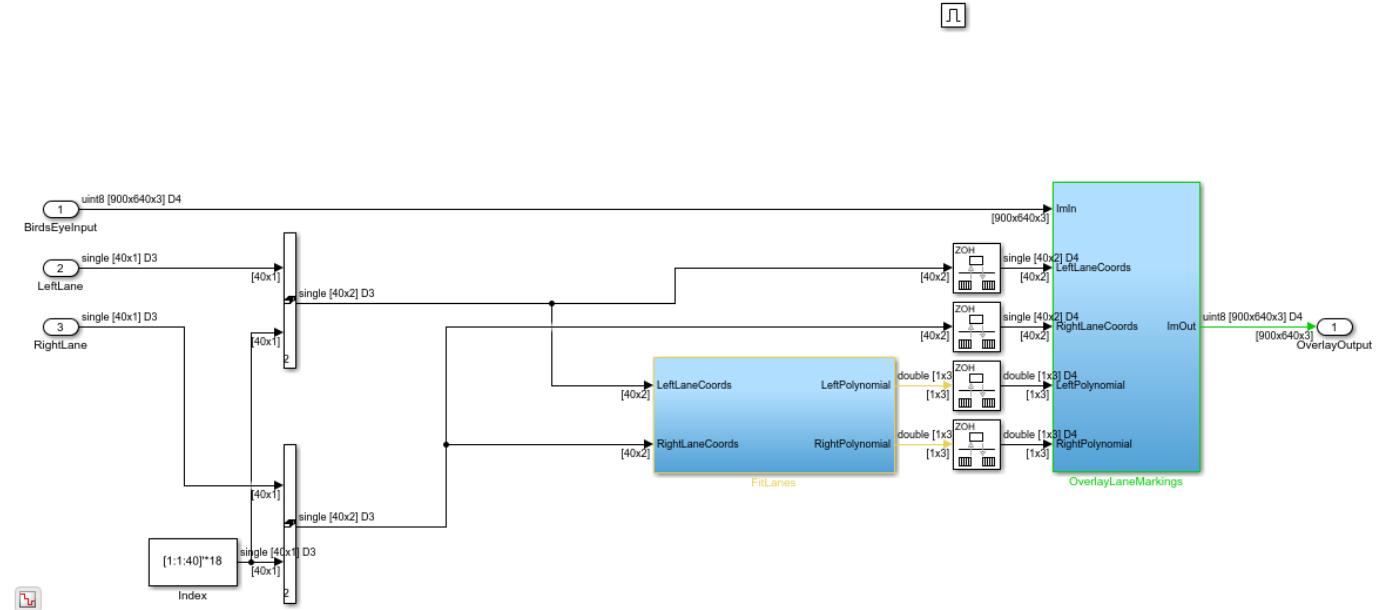
## Fit Lanes

The Fit Lanes subsystem runs a RANSAC based line-fitting routine on the generated lane candidates. RANSAC is an iterative algorithm which builds up a table of inliers based on a distance measure

between the proposed curve, and the input data. At the output of this subsystem, there is a [3x1] vector which specifies the polynomial coefficients found by the RANSAC routine.

### Overlay Lane Markings

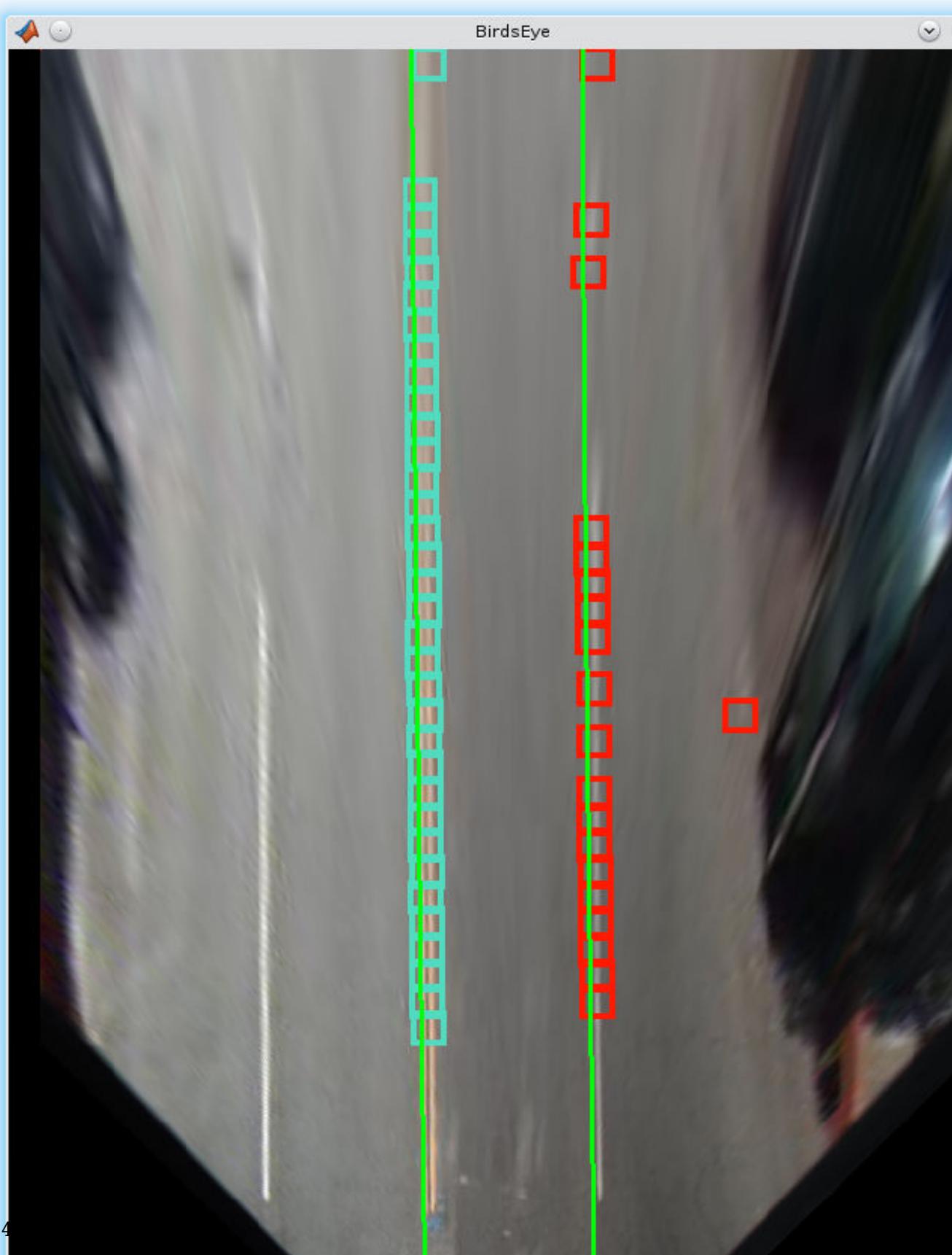
The Overlay Lane Markings subsystem performs image visualization operations. It overlays the ego lanes and curves found by the lane-fitting routine.



### Results of the Simulation

The model includes two video displays shown at the output of the simulation results. The **BirdsEye** display shows the output in the warped perspective after lane candidates have been overlaid, polynomial fitting has been performed and the resulting polynomial overlaid onto the image. The **OriginalOverlay** display shows the **BirdsEye** output warped back into the original perspective.

Due to the large frame sizes used in this model, simulation can take a relatively long time to complete. If you have an HDL Verifier™ license, you can accelerate simulation speed by directly running the HDL Lane Detector subsystem in hardware using FPGA in the Loop (TM).





### HDL Code Generation

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command.

```
makehdl('LaneDetectionHDL/HDLLaneDetector')
```

To generate the test bench, use the following command. Note that test bench generation takes a long time due to the large data size. You may want to reduce the simulation time before generating the test bench.

```
makehdltb('LaneDetectionHDL/HDLLaneDetector')
```

For faster test bench simulation, you can generate a SystemVerilog DPIC test bench using the following command.

```
makehdltb('LaneDetectionHDL/HDLLaneDetector','GenerateSVDPITestBench','ModelSim')
```

### Conclusion

This example has provided insight into the challenges of designing ADAS systems in general, with particular emphasis paid to the acceleration of critical parts of the design in hardware.

### References

[1] R. K. Satzoda and Mohan M. Trivedi, "Vision based Lane Analysis: Exploration of Issues and Approaches for Embedded Realization", 2013 IEEE Conference on Computer Vision and Pattern Recognition.

[2] Video from Caltech Lanes Dataset - Mohamed Aly, "Real time Detection of Lane Markers in Urban Streets", 2008 IEEE Intelligent Vehicles Symposium - used with permission.

# Code Generation Options in the HDL Coder Dialog Boxes

---

- “Set HDL Code Generation Options” on page 12-2
- “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6
- “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 12-11
- “Generate HDL Code from Simulink Model from Command Line” on page 12-15

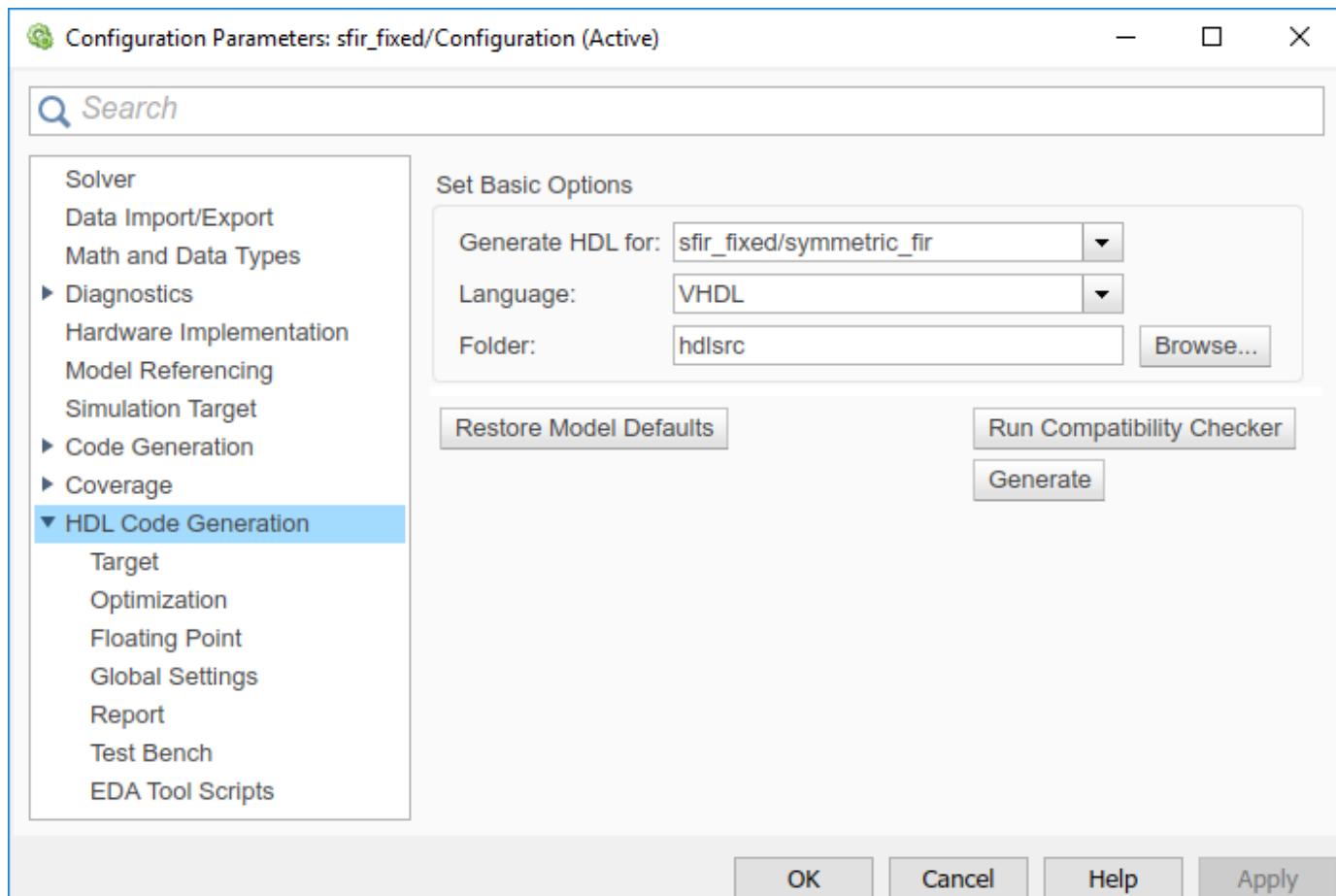
## Set HDL Code Generation Options

### In this section...

- “HDL Code Generation Options in the Configuration Parameters Dialog Box” on page 12-2
- “HDL Code Tab in Simulink Toolstrip” on page 12-3
- “HDL Code Options in the Block Context Menu” on page 12-4
- “The HDL Block Properties Dialog Box” on page 12-5

### HDL Code Generation Options in the Configuration Parameters Dialog Box

The following figure shows the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. To open this dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.

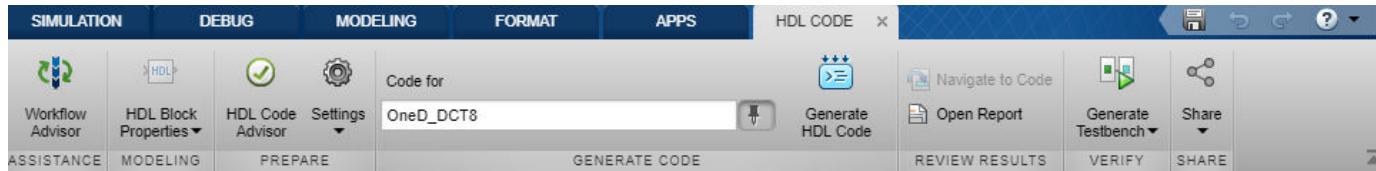


**Note** When the **HDL Code Generation** pane of the Configuration Parameters dialog box appears, clicking the **Help** button displays general help for the Configuration Parameters dialog box.

For more information, see “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6.

## HDL Code Tab in Simulink Toolbar

The Simulink Toolbar contains contextual tabs that appear only when you need to access them. To access the **HDL Code** tab, open the **HDL Coder** app from the **Apps** tab on the Simulink Toolbar.



The **HDL Code** tab provides shortcuts to the HDL code generation options. You can also use this tab to initiate code generation.

Options include:

- **Workflow Advisor**: Open the HDL Workflow Advisor.
- **HDL Block Properties**: Open the HDL-compatible block library in the Simulink Library Browser or open the HDL Block Properties dialog box for a block that you select in your model.

---

**Note** After you open the HDL-compatible block library, to restore the Library Browser to the default view, in the Library Browser, click the button.

---

- **HDL Code Advisor**: Open the HDL Code Advisor for the model or the selected Subsystem.
- **Settings**: Open the **HDL Code Generation** pane in the Configuration Parameters dialog box.
  - **Report Options**: Open the **HDL Code Generation > Report** pane.
  - **Remove HDL Configuration from Model**: The HDL configuration component is internal data that HDL Coder creates and attaches to a model. This component lets you view the **HDL Code Generation** pane in the Configurations Parameters dialog box, and use the **HDL Code Generation** pane to set HDL code generation options. To remove the HDL Code Generation configuration component to or from a model, select this option. For more information, see “Add or Remove the HDL Configuration Component” on page 25-24.
- **Code for**: Select the top-level Subsystem or model for which you want to generate HDL code. This option corresponds to the **Generate HDL for** option in the **HDL Code Generation** pane of the Configuration Parameters dialog box.
- **Generate HDL Code**: Initiate HDL code generation; equivalent to the **Generate HDL Code** check box in the **HDL Code Generation > Global Settings > Advanced** tab of the Configuration Parameters dialog box.
- **Navigate to Code**: Select a block in your model and navigate to the HDL code generated for that block. To use this setting, you must have generated a traceability report.
- **Open Report**: Opens the Code Generation Report if this report exists on the path. Otherwise, this button opens the HDL Check Report.
- **Generate Test Bench**: Initiate test bench code generation; equivalent to the **Generate Test Bench** button in the Configuration Parameters dialog box. To use this button, you If you do not

select a subsystem in the **Generate HDL for** menu, the **Generate Test Bench** menu option is not available.

If you have HDL Verifier™ installed, you can generate a HDL cosimulation model or a SystemVerilog DPI component.

- **Share:** Generate a protected model that you can share with a third party without revealing the intellectual property of the model.

## HDL Code Options in the Block Context Menu

When you right-click a block that HDL Coder supports, the context menu for the block includes an **HDL Code** submenu. The code generator enables items in the submenu according to:

- The block type: for subsystems, the menu enables some options that are specific to subsystems.
- Whether or not code and traceability information has been generated for the block or subsystem.

**Note** You can also access the options in the context menu from the **HDL Code** tab in the Simulink Toolbar. To access this tab, open the **HDL Coder** app from the **Apps** tab.

The following summary describes the **HDL Code** submenu options.

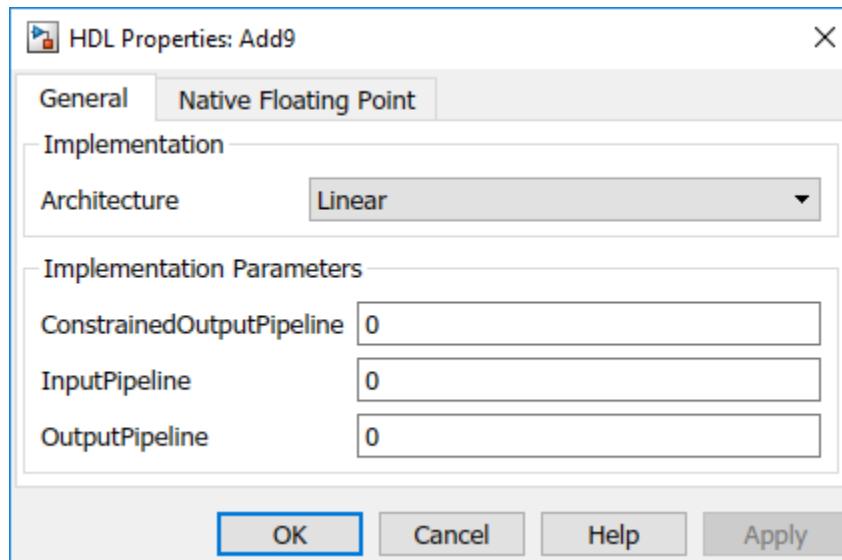
Option	Description	Availability
<b>Check Subsystem Compatibility</b>	Runs the HDL compatibility checker ( <code>checkhdl</code> ) on the subsystem.	Available only for subsystems.
<b>Generate HDL for Subsystem</b>	Runs the HDL code generator ( <code>makehdl</code> ) and generates code for the subsystem.	Available only for subsystems.
<b>HDL Coder Properties</b>	Opens the Configuration Parameters dialog box, with the top-level HDL Code Generation pane selected.	Available for blocks or subsystems.
<b>HDL Block Properties</b>	Opens a block properties dialog box for the block or subsystem. See “Set and View HDL Model and Block Parameters” on page 22-52 for more information.	Available for blocks or subsystems.
<b>HDL Workflow Advisor</b>	Opens the HDL Workflow Advisor for the subsystem.	Available only for subsystems.
<b>Navigate to Code</b>	Activates the HTML code generation report window, displaying the beginning of the code generated for the selected block or subsystem. For more information, see “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-4.	Enabled when both code and a traceability report have been generated for the block or subsystem.

## The HDL Block Properties Dialog Box

HDL Coder provides selectable alternate block implementations for many block types. Each implementation is optimized for different characteristics, such as speed or chip area. The HDL Properties dialog box lets you choose the implementation for a selected block.

Most block implementations support a number of implementation parameters that let you control further details of code generation for the block. The HDL Properties dialog box lets you set implementation parameters for a block.

The following figure shows the HDL Properties dialog box for a block.



There are a number of ways to specify implementations and implementation parameters for individual blocks or groups of blocks. See “Set and View HDL Model and Block Parameters” on page 22-52.

### See Also

`makehdltb` | `makehdl`

### More About

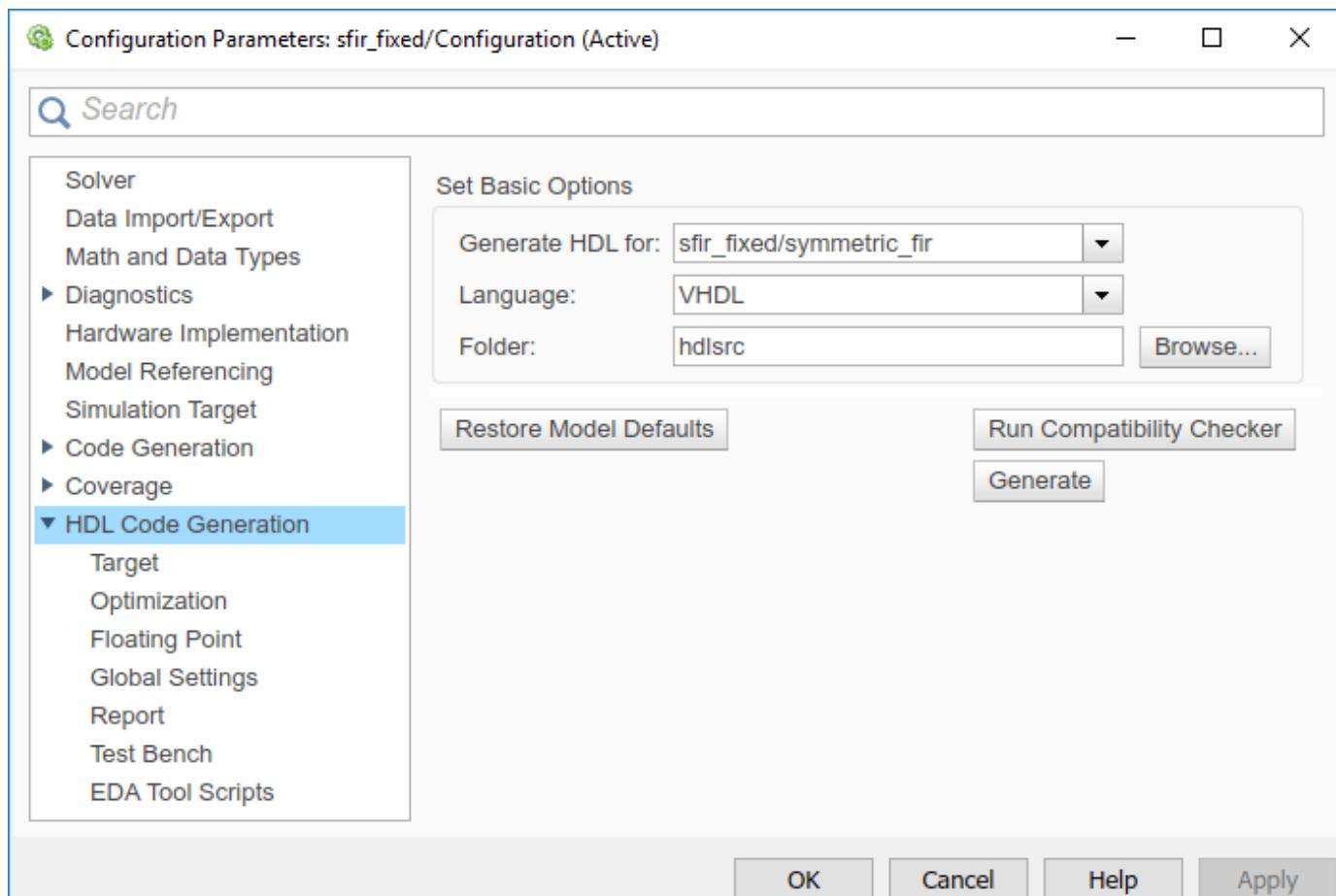
- “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6
- “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 12-11

## HDL Code Generation Options in Configuration Parameters Dialog Box

### In this section...

- “HDL Code Generation Pane: Target” on page 12-7
- “HDL Code Generation Pane: Optimization” on page 12-7
- “HDL Code Generation Pane: Floating Point” on page 12-7
- “HDL Code Generation Pane: Global Settings” on page 12-7
- “HDL Code Generation Pane: Report” on page 12-9
- “HDL Code Generation Pane: Testbench” on page 12-9
- “HDL Code Generation Pane: EDA Tool Scripts” on page 12-9

The following figure shows the top-level **HDL Code Generation** pane in the Configuration Parameters dialog box. To open this dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.



The **HDL Code Generation** pane consists of basic options that specify the DUT that you want to generate code for, target language, and folder settings. The **Generate HDL for** setting is synchronized with the **Code for** menu in the **HDL Code** tab. You can also use the buttons in this pane

to initiate code generation and perform compatibility checking. The **HDL Code Generation** pane consists of various subpanes.

To learn more about the parameters in this pane, see “Target Language and Folder Selection Parameters” on page 13-3.

## HDL Code Generation Pane: Target

The **HDL Code Generation > Target** pane consists of parameters that you can use to specify the target device and synthesis tool. You can also specify the target frequency value in MHz.

To learn more about the parameters in this pane, see “Tool and Device Parameters” on page 14-3 and “Target Frequency Parameter” on page 14-8.

## HDL Code Generation Pane: Optimization

The **HDL Code Generation > Optimization** pane consists of parameters that you can use to specify various area and speed optimizations to optimize your design. You can also specify use of multicycle path constraints as timing requirement for synthesis tools to meet.

To learn more about the parameters in this pane, see:

- “Delay Balancing and General Optimization Parameters” on page 15-3
- “RAM Mapping Parameters” on page 15-7
- “Pipelining Parameters” on page 15-9
- “Resource Sharing Settings for Various Blocks” on page 21-105
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 21-109
- “Multicycle Path Constraints Parameters” on page 15-27

## HDL Code Generation Pane: Floating Point

The **HDL Code Generation > Floating Point** pane consists of parameters that you can use to specify the floating-point IP library and additional options depending on whether to use native floating-point support and map to target floating-point IP libraries.

To learn more about the parameters in this pane, see:

- “Floating Point IP Library” on page 16-3
- “Native Floating Point Parameters” on page 16-4
- “FPGA Floating-Point Library Targeting Parameters” on page 16-8

## HDL Code Generation Pane: Global Settings

The **HDL Code Generation > Floating Point** pane consists of parameters that you can use to specify the floating-point IP library and additional options depending on whether to use native floating-point support and map to target floating-point IP libraries.

To learn more about the top-level parameters in the **Clock Settings** section of this pane, see:

- “Clock Settings and Timing Controller Postfix Parameters” on page 17-4
- “Reset Settings and Parameters” on page 17-8
- “Clock Enable Settings and Parameters” on page 17-12
- “Oversampling factor” on page 17-15

The preceding sections contain links to learn more about the parameters in various tabs of this pane.

### **General Tab**

- “Comment in header” on page 17-17
- “Language-Specific File Extension Parameters” on page 17-19
- “Language-Specific Identifiers and Postfix Parameters” on page 17-21
- “Split entity and architecture Parameters” on page 17-25
- “Complex Signals Postfix Parameters” on page 17-28
- “VHDL Architecture and Library Name and Code for Model Reference Parameters” on page 17-30
- “Generate Statement and Vector and Component Instance Label Parameters” on page 17-32

### **Ports Tab**

- “Input and Output Port and Clock Enable Output Type Parameters” on page 17-35
- “Minimize Clock Enables and Reset Signal Parameters” on page 17-37
- “Using Trigger Signals and Scalarization and Test Point DUT Port Generation Parameters” on page 17-41

### **Coding Style Tab**

- “RTL Customization Parameters for Constants and MATLAB Function Blocks” on page 17-49
- “RTL Annotation Parameters” on page 17-44
- “RTL Customization Parameters for RAMs” on page 17-51
- “No-reset registers initialization” on page 17-53
- “RTL Style Parameters” on page 17-55
- “Timing Controller Settings” on page 17-60
- “File Comment Customization Parameters” on page 17-62

### **Coding Standards Tab**

- “Choose Coding Standard and Report Option Parameters” on page 17-64
- “Basic Coding Practices Parameters” on page 17-66
- “RTL Description Rules for clock enables and resets Parameters” on page 17-71
- “RTL Description Rules for Conditional Parameters” on page 17-74
- “Other RTL Description Rule Parameters” on page 17-77
- “RTL Design Rule Parameters” on page 17-80

### **Model Generation and Advanced Tabs**

- “Model Generation Parameters for HDL Code” on page 17-82

- “Naming and Layout Options for Model Generation” on page 17-85
- “Diagnostic Parameters for Optimizations” on page 17-89
- “Diagnostic Parameters for Reals and Black Box Interfaces” on page 17-92
- “Code Generation Output Parameter” on page 17-94

## HDL Code Generation Pane: Report

The **HDL Code Generation > Report** pane consists of parameters that you can use to specify generation of a Code Generation Report with the HDL code. In addition to the summary and clock information, you can specify whether the Code Generation Report report should include sections that display high-level resource utilization, traceability information, effect of various optimizations, and floating-point resource consumption.

To learn more about the parameters in this pane, see “Code Generation Report Parameters” on page 18-3.

## HDL Code Generation Pane: Testbench

The **HDL Code Generation > Testbench** pane consists of parameters that you can use to specify generation of a testbench to verify the HDL code. You can also specify various testbench options related to clock and reset input signals, setup and hold time, and testbench tolerance parameters.

To learn more about the parameters in this pane, see:

- “Test Bench Generation Output Parameters” on page 19-3
- “Test Bench Postfix Parameters” on page 19-8
- “Clock and Reset Input Parameters for Testbench” on page 19-10
- “Setup and Hold Time Parameters for Testbench” on page 19-16
- “Test Bench Stimulus and Output Parameters” on page 19-18
- “Multi-File Testbench and Simulation Library Path Parameters” on page 19-23
- “Floating-Point Tolerance Parameters” on page 19-26

## HDL Code Generation Pane: EDA Tool Scripts

The **HDL Code Generation > EDA Tool Scripts** pane consists of parameters that you can use to specify options that control generation of script files for third-party HDL simulation and synthesis tools.

To learn more about the parameters in this pane, see:

- “Generate EDA scripts” on page 20-3
- “Compilation Script Parameters” on page 20-4
- “Simulation Script Parameters” on page 20-7
- “Synthesis Script Parameters” on page 20-11
- “Lint Script Parameters” on page 20-16

**See Also**

`makehdltb` | `makehdl`

**More About**

- “Set HDL Code Generation Options” on page 12-2
- “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 12-11

# Generate HDL Code from Simulink Model Using Configuration Parameters

## In this section...

- “FIR Filter Model” on page 12-11
- “Create a Folder and Copy Relevant Files” on page 12-12
- “Open HDL Code Generation Pane of Configuration Parameters Dialog Box” on page 12-13
- “Generate HDL Code” on page 12-13

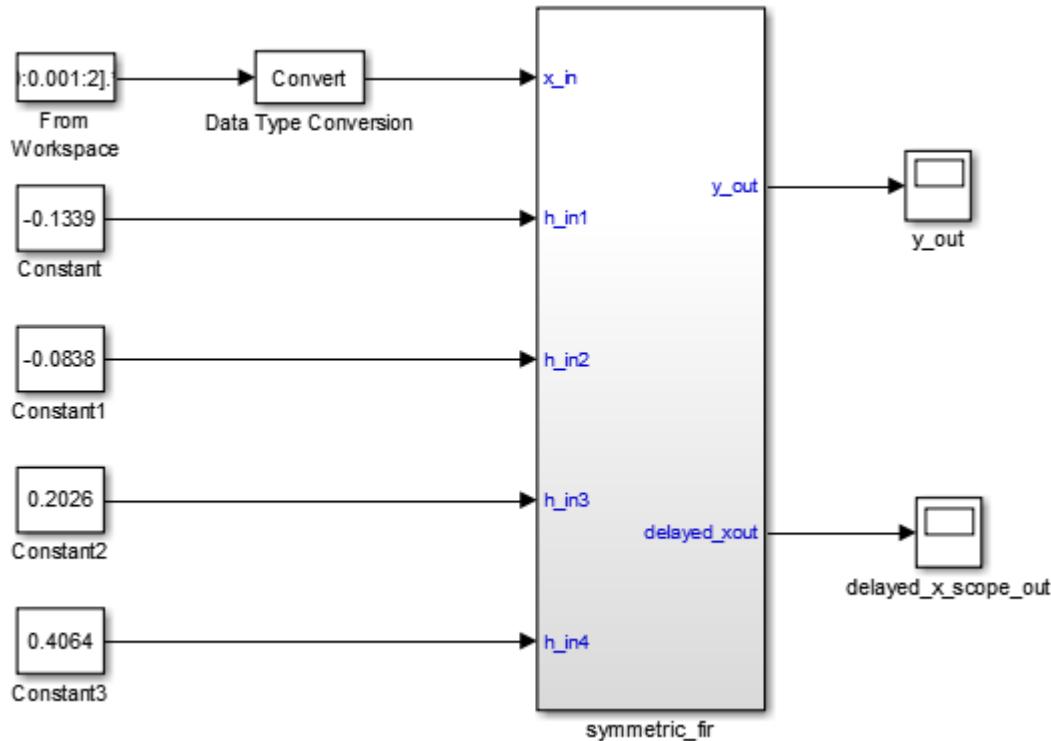
You can view and edit options and parameters that affect HDL code generation in the Configuration Parameters dialog box, or in the Model Explorer. This example illustrates how you can use the Configuration Parameters dialog box to generate HDL code for the Symmetric FIR filter model.

## FIR Filter Model

Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



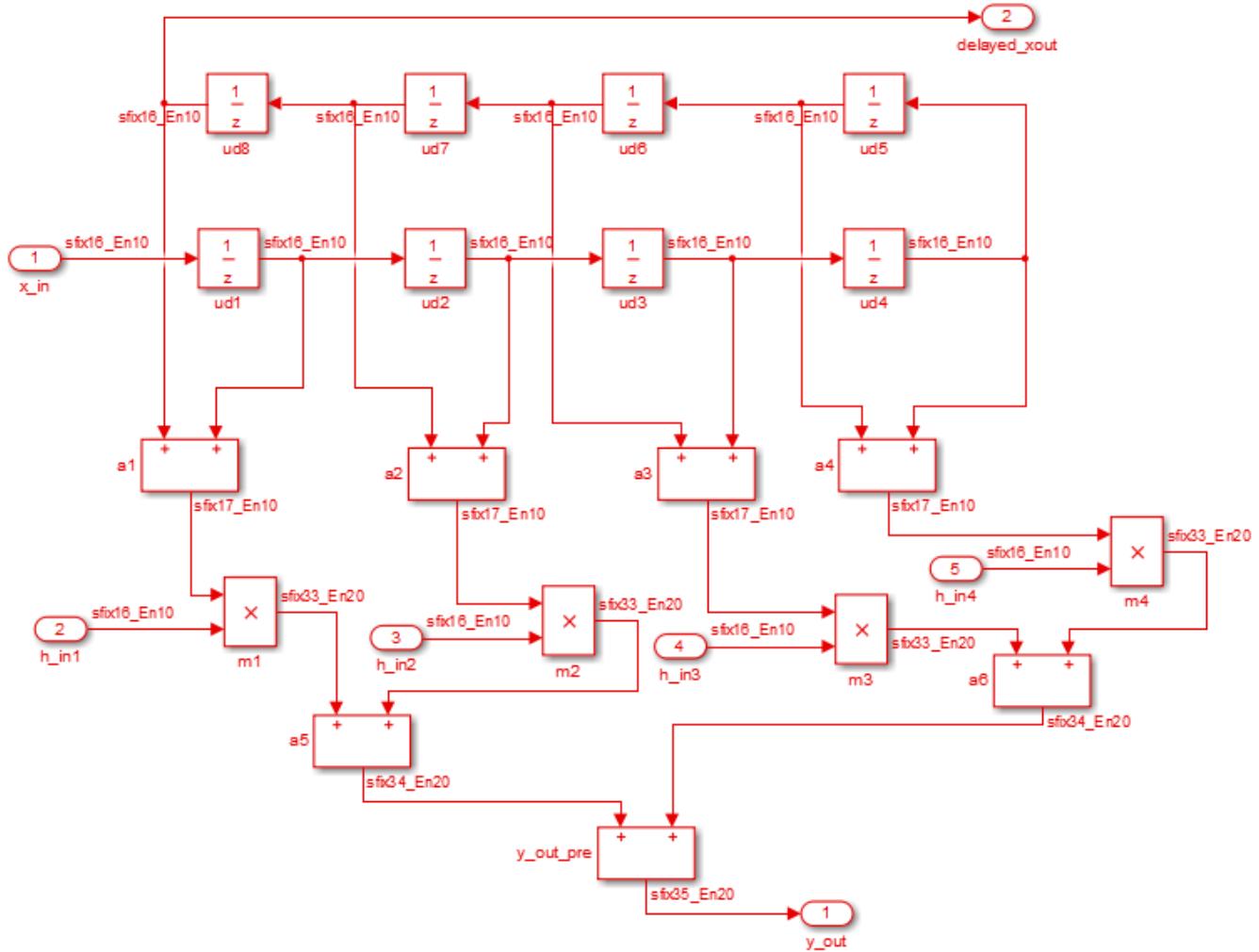
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



## Create a Folder and Copy Relevant Files

In MATLAB:

- 1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

- 2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

- 3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

## Open HDL Code Generation Pane of Configuration Parameters Dialog Box

This figure shows the top-level **HDL Code Generation** pane of the Configuration Parameters dialog box. To open this dialog box, in the Apps gallery, click **HDL Coder**. The **HDL Code** tab appears. In the **Prepare** section, click **Settings**.

The **HDL Code Generation** pane consists of basic options that specify the DUT that you want to generate code for, target language, and folder settings. The **Generate HDL for** setting is synchronized with the **Code for** menu in the **HDL Code** tab. You can also use the buttons in this pane to initiate code generation and perform compatibility checking. The **HDL Code Generation** pane consists of various subpanes that you can use to specify various settings related to clock and reset signals to reporting and optimization settings.

In the **HDL Code Generation** pane

- The **Generate HDL for** field specifies the `sfir_fixed/symmetric_fir` subsystem for code generation.
- The **Language** field specifies generation of VHDL code.
- The **Folder** field specifies a target folder that stores generated code files and scripts.

To learn more about the various parameters in the **HDL Code Generation** pane, see “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6.

## Generate HDL Code

To generate code, click the **Generate** button. By default, HDL Coder generates VHDL code in the target `hdlsrc` folder.

To generate Verilog code for the model:

- 1 In the **HDL Code** tab, click **Settings**.
- 2 In the **HDL Code Generation** pane, for **Language**, select **Verilog**. Leave other settings to the default. Click **Apply** and then click **Generate**.

HDL Coder compiles the model before generating code. Depending on model display options such as port data types, the model can change in appearance after code generation. As code generation proceeds, HDL Coder displays progress messages in the MATLAB command line with:

- Link to the Configuration Set that indicates the model for which the Configuration Parameters are applied.

- Links to the generated files. To view the files in the MATLAB Editor, click the links.
  - `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir.vhd` filter.
  - `symmetric_fir_compile.do`: Mentor Graphics ModelSim® compilation script (vcom command) to compile the generated VHDL code.
  - `symmetric_fir_synplify.tcl`: Synplify® synthesis script.
  - `symmetric_fir_map.txt`: This report maps generated entities to the subsystems that generated them. See “Trace Code Using the Mapping File” on page 25-21

The process completes with the message:

```
### HDL Code Generation Complete.
```

## See Also

`makehdltb` | `makehdl`

## More About

- “Set HDL Code Generation Options” on page 12-2
- “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6

# Generate HDL Code from Simulink Model from Command Line

## In this section...

["FIR Filter Model" on page 12-15](#)

["Create a Folder and Copy Relevant Files" on page 12-16](#)

["Generate HDL Code" on page 12-17](#)

You can customize and edit HDL code generation options and then generate code at the command line. This example illustrates how you can use the Configuration Parameters dialog box to generate HDL code for the Symmetric FIR filter model.

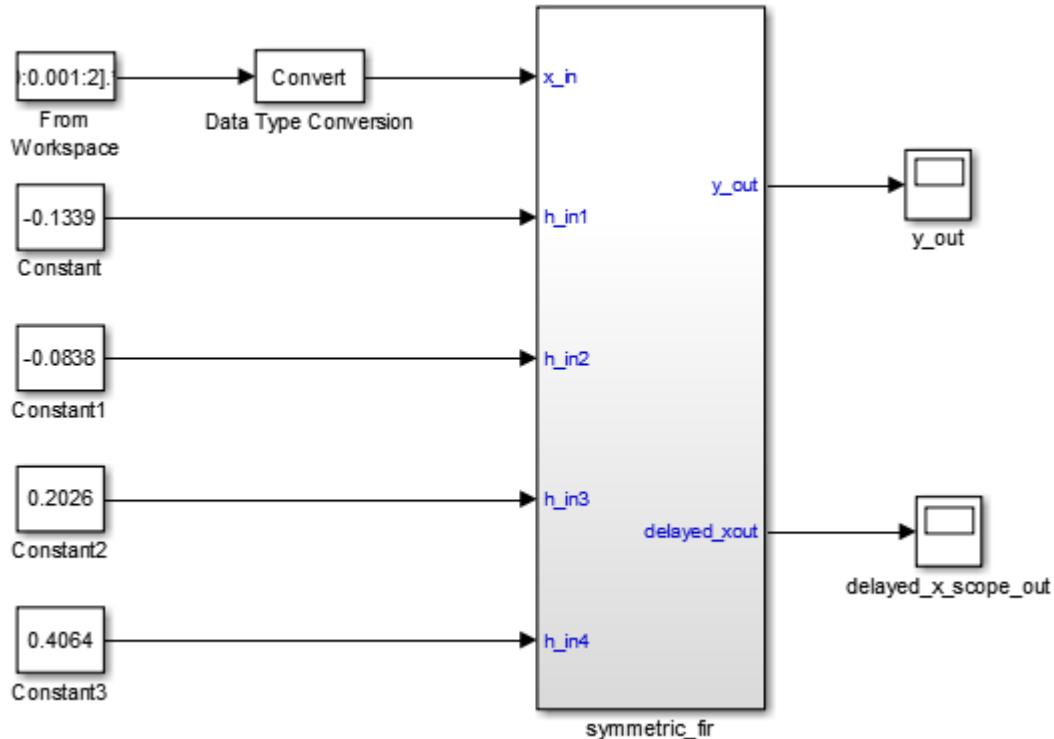
## FIR Filter Model

Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see ["Check HDL Compatibility of Simulink Model Using HDL Code Advisor" on page 39-2](#). You can also customize the model parameters by using the `hdlsetup` function.

```
hdlsetup(gcs)
```

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



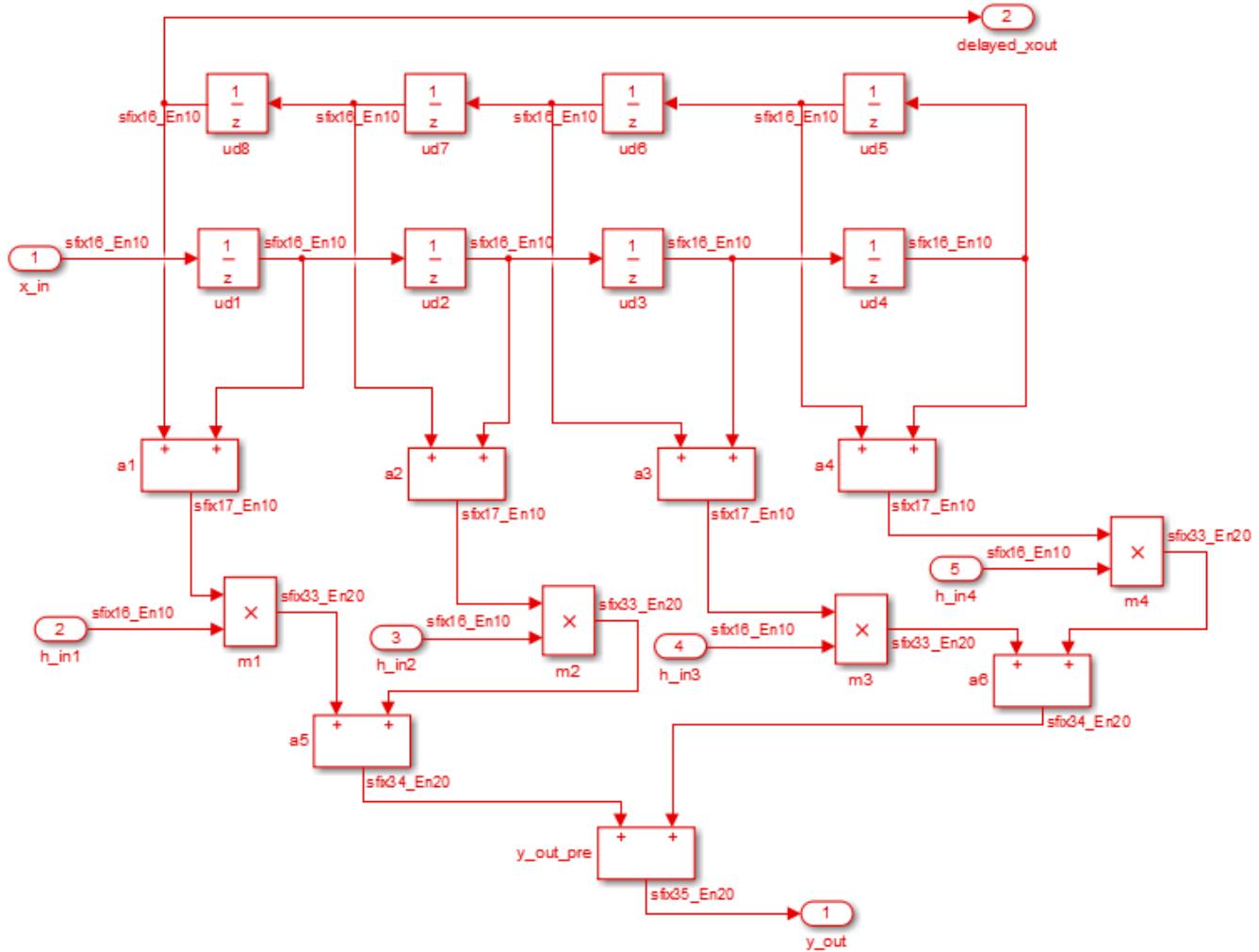
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



## Create a Folder and Copy Relevant Files

In MATLAB:

- 1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

- 2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

- 3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

## Generate HDL Code

To generate HDL code for the DUT, you use the `makehdl` function. For example, to generate HDL code for the `symmetric_fir` subsystem, enter:

```
makehdl('sfir_fixed/symmetric_fir')
```

To specify the customizations before you generate HDL code, use the `hdlset_param` function. You can also specify various name-value pair arguments with the `makehdl` function to customize HDL code generation options while generating HDL code. For example, to generate Verilog code, use the `TargetLanguage` property.

```
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog')
```

Alternatively, if you are using `hdlset_param`, set this parameter on the model and then run the `makehdl` function.

```
hdlset_param('sfir_fixed', 'TargetLanguage', 'Verilog')
makehdl('sfir_fixed/symmetric_fir')
```

HDL Coder compiles the model before generating code. Depending on model display options such as port data types, the model can change in appearance after code generation. As code generation proceeds, HDL Coder displays progress messages in the MATLAB command line with:

- Link to the Configuration Set that indicates the model for which the Configuration Parameters are applied.
- Links to the generated files. To view the files in the MATLAB Editor, click the links.
  - `symmetric_fir.vhd`: VHDL code. This file contains an entity definition and RTL architecture implementing the `symmetric_fir.vhd` filter.
  - `symmetric_fir_compile.do`: Mentor Graphics ModelSim compilation script (vcom command) to compile the generated VHDL code.
  - `symmetric_fir_synplify.tcl`: Synplify synthesis script.
  - `symmetric_fir_map.txt`: This report maps generated entities to the subsystems that generated them. See “Trace Code Using the Mapping File” on page 25-21

The process completes with the message:

```
### HDL Code Generation Complete.
```

## See Also

`makehdltb` | `makehdl`

## More About

- “Set HDL Code Generation Options” on page 12-2
- “HDL Code Generation Options in Configuration Parameters Dialog Box” on page 12-6

# HDL Code Generation Pane: General

---

- “HDL Code Generation Top-Level Pane Overview” on page 13-2
- “Target Language and Folder Selection Parameters” on page 13-3

## HDL Code Generation Top-Level Pane Overview

The top-level **HDL Code Generation** pane contains settings for target language and the model that you want to generate code for, and buttons that initiate code generation and compatibility checking, and sets code generation parameters.

### Buttons in the HDL Code Generation Top-Level Pane

The buttons in the **HDL Code Generation** pane perform functions related to code generation. These buttons are:

**Generate:** Initiates code generation for the system selected in the **Generate HDL for** menu. See also `makehdl`.

**Run Compatibility Checker:** Invokes the compatibility checker to examine the system selected in the **Generate HDL for** menu for compatibility problems. See also `checkhdl`.

**Browse:** Lets you navigate to and select the target folder to which generated code and script files are written. The path to the target folder is entered into the **Folder** field.

**Restore Model Defaults:** Sets model parameters to their default values.

# Target Language and Folder Selection Parameters

## In this section...

- “Generate HDL for” on page 13-3
- “Language” on page 13-3
- “Folder” on page 13-4
- “Restore Model Defaults” on page 13-5
- “Run Compatibility Checker” on page 13-5
- “Generate” on page 13-6

This page describes configuration parameters in the **HDL Code Generation** pane of the Configuration Parameters dialog box. By using these parameters, you can specify the Subsystem that you want to generate HDL code for, the target HDL language, and the target folder into which code is generated.

## Generate HDL for

Select the subsystem or model from which code is generated. The list includes the path to the root model and to subsystems in the model. When you specify this parameter and click the **Generate** button, HDL Coder generates code for the Subsystem that you specify. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

### Settings

**Default:** The top level subsystem in the root model is selected.

#### Command-Line Information

**Property:** `HDLSubsystem`

**Type:** character vector

**Value:** A valid path to your subsystem

**Default:** Path to the top level subsystem in root model

For example, you can generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Specify the subsystem using the property `HDLSubsystem` as an argument to `makehdl`.
 

```
makehdl('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir')
```
- Pass in the path to the subsystem as an first argument to `makehdl`.
 

```
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

## Language

Select the language (VHDL or Verilog) in which code is generated. The selected language is referred to as the target language. When you specify the **Language** and click the **Generate** button, HDL Coder generates code in that language for the Subsystem that is specified by the **Generate HDL for** parameter. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

The generated HDL code complies with these standards:

- VHDL-1993 (IEEE® 1076-1993)
- Verilog-2001 (IEEE 1364-2001)

## Settings

### Default: VHDL

#### VHDL

Generate VHDL code.

#### Verilog

Generate Verilog code.

## Command-Line Information

**Property:** TargetLanguage

**Type:** character vector

**Value:** 'VHDL' | 'Verilog'

**Default:** 'VHDL'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate Verilog code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage','Verilog')
```

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TargetLanguage','Verilog')
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

## Folder

Enter a path to the folder into which code is generated. Alternatively, click **Browse** to navigate to and select a folder. The selected folder is referred to as the target folder. When you specify the **Folder** and click the **Generate** button, HDL Coder generates code into that folder for the Subsystem that is specified by the **Generate HDL for** parameter. By default, the HDL code is generated in VHDL language and into the `hdlsrc` folder.

## Settings

**Default:** The default target folder is a subfolder of your working folder, named `hdlsrc`. HDL Coder writes the generated files into this subfolder. The folder name can be a complete path name, specified as a character vector.

## Command-Line Information

**Property:** TargetDirectory

**Type:** character vector

**Value:** A valid path to your target folder  
**Default:** 'hdlsrc'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate HDL code into a custom target folder for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','TargetDirectory','C:/Temp/hdlsrc')
```

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TargetDirectory','C:/Temp/hdlsrc')
makehdl('sfir_fixed/symmetric_fir')
```

See also `makehdl`.

## Restore Model Defaults

This button resets the model-level HDL settings to the default values. The block settings are not changed. To clear the block settings, use `hdlrestoreparams`.

---

**Note** If you clear the model-level settings, you cannot restore the previous settings. To restore the settings, close the model without saving and then reopen the model.

---

### Command-Line Information

**Function:** `hdlrestoreparams`

**Type:** character vector

**Value:** model name

**Default:** ''

## Run Compatibility Checker

This setting checks whether the Subsystem that you specify by using **Generate HDL for** is compatible for HDL code generation. The setting generates a HDL Check Report that displays errors, warnings, and messages. See “Check Subsystem for HDL Compatibility” on page 21-19.

### Command-Line Information

**Function:** `checkhdl`

**Type:** character vector

**Value:** subsystem or model name

**Default:** ''

### See Also

`checkhdl`

## Generate

This setting generates HDL code for the Subsystem that you specify by using **Generate HDL for**. If the Subsystem is not HDL-compatible, the code generator displays errors in the HDL Check Report.

### Command-Line Information

**Function:** makehdl

**Type:** character vector

**Value:** subsystem or model name

**Default:** ''

### See Also

`makehdl`

# HDL Code Generation Pane: Target

---

- “Target Overview” on page 14-2
- “Tool and Device Parameters” on page 14-3
- “Target Frequency Parameter” on page 14-8

## Target Overview

The **Target** pane enables you to specify the target hardware settings. You can specify the tool and device settings and the target frequency.

# Tool and Device Parameters

## In this section...

- “Synthesis Tool” on page 14-3
- “Family” on page 14-4
- “Device” on page 14-5
- “Package” on page 14-5
- “Speed” on page 14-6

This page describes configuration parameters in the **Tool and Device** section of the **HDL Code Generation > Target** pane of the Configuration Parameters dialog box. By using the parameters in this section, you can specify the synthesis tool, and then select the **Family**, **Device**, **Package**, and **Speed** for your synthesis target.

## Synthesis Tool

Specify the synthesis tool for targeting the generated HDL code. To use HDL Coder with one of the supported third-party FPGA synthesis tools, add the tool to the system path using the `hdlsetuptoolpath` function. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool.

### Settings

**Default:** No synthesis tool specified

The options are:

No synthesis tool specified

Select this option if you do not want to perform logic synthesis. You can generate HDL code from your design.

Xilinx Vivado

Specify Xilinx Vivado as the synthesis tool.

Xilinx ISE

Specify Xilinx ISE as the synthesis tool.

Altera Quartus II

Specify Altera Quartus II as the synthesis tool.

Microsemi Libero SoC

Specify Microsemi® Libero® SoC as the synthesis tool.

If your synthesis tool is not one of the **Synthesis tool** options, see “[Synthesis Tool Path Setup](#)”.

### Command-Line Information

**Property:** `SynthesisTool`

**Type:** character vector

**Value:** '' | 'Xilinx Vivado' | 'Xilinx ISE' | 'Altera Quartus II'

**Default:** ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `Altera Quartus II` as the `SynthesisTool` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'SynthesisTool','Altera Quartus II')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisTool','Altera Quartus II')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- `hdlsetupoolpath`
- “Tool Setup”

## Family

Specify the target device chip family for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the chip family for your target device, at the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

### Settings

**Default:** ''

Specify the target device chip family for your Simulink model as a character vector.

#### Command-Line Information

**Property:** `SynthesisToolChipFamily`

**Type:** character vector

**Value:** A valid chip family for the target device

**Default:** ''

For example, if your `SynthesisTool` is `Xilinx Vivado`, you can specify `Virtex7` as the `SynthesisToolChipFamily` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'SynthesisToolChipFamily', 'Virtex7')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolChipFamily', 'Virtex7')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

**Device**

Specify the target device name for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the device name for your target device, at the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

**Settings**

**Default:** ‘’

Specify the target device name for your Simulink model as a character vector.

**Command-Line Information**

**Property:** `SynthesisToolDeviceName`

**Type:** character vector

**Value:** A valid device name for the synthesis tool

**Default:** ‘’

You can get the `SynthesisToolDeviceName` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...
    'SynthesisToolDeviceName')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolDeviceName', 'xc7v2000t')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

**Package**

Specify the target device package name for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the device name for your target device, at the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

## Settings

**Default:** ''

Specify the target device package name for your Simulink model as a character vector.

### Command-Line Information

**Property:** SynthesisToolPackageName

**Type:** character vector

**Value:** A valid package name for the synthesis tool

**Default:** ''

You can get the `SynthesisToolPackageName` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...
    'SynthesisToolPackageName')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolPackageName', 'fgh1761')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

## Speed

Specify the target device speed value for your model as a character vector. When you specify the **Synthesis Tool**, HDL Coder populates the **Family**, **Device**, **Package**, and **Speed** with default values for that tool. To find the chip family for your target device, at the MATLAB command line, enter `hdlcoder.supportedDevices`. Then, open the linked report and find your target device details.

## Settings

**Default:** ''

Specify the target device speed value for your Simulink model as a character vector.

### Command-Line Information

**Property:** SynthesisToolSpeedValue

**Type:** character vector

**Value:** A valid speed value for the target device

**Default:** ''

You can get the `SynthesisToolSpeedValue` when you specify the `SynthesisTool` for your model. Consider that the `SynthesisTool` is set to `Xilinx Vivado` and the `SynthesisToolChipFamily` is set to `Virtex7`.

- To get the default device name, pass the property as an argument to the `hdlget_param` function.

```
hdlget_param('sfir_fixed', ...
    'SynthesisToolSpeedValue')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','SynthesisToolSpeedValue', '-1')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- `hdlsetuptoolpath`
- “Tool Setup”

## Target Frequency Parameter

This configuration parameter resides in the **Objectives Settings** section of the **HDL Code Generation > Target** pane of the Configuration Parameters dialog box. By using this parameter, you can specify the target frequency in MHz for multiple features and workflows. Before setting the target frequency, make sure that you specify the **Synthesis Tool**.

### Settings

**Default:** 0

This setting is the target frequency in MHz for multiple features and workflows that HDL Coder supports. The supported features are:

- **FPGA floating-point target library mapping:** Specify the target frequency that you want the IP to achieve when you use **ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)**. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of 200 MHz. See also “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20.
- **Adaptive pipelining:** If your design uses multipliers, specify the synthesis tool and the target frequency. Based on these settings, HDL Coder estimates the number of pipelines that can be inserted to improve area and timing on the target platform. If you do not specify the target frequency, HDL Coder uses a target frequency of 0 MHz and does not insert adaptive pipelines. See also “Adaptive Pipelining” on page 24-130.

You can also set the target frequency by using the **Target Frequency (MHz)** setting in the **Set Target Frequency** task in the HDL Workflow Advisor.

Specify the target frequency for these workflows-

- **Generic ASIC/FPGA:** To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency. It adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error. Target frequency is not supported with Microsemi Libero SoC.
- **IP Core Generation:** To specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- **Simulink Real-Time FPGA I/O:** For Speedgoat I/O modules that are supported with Xilinx ISE, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

The Speedgoat I/O modules that are supported with Xilinx Vivado use the **IP Core Generation** workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.

- **FPGA Turnkey:** To generate the clock module to produce the clock signal with that frequency automatically.

## Command-Line Information

**Property:** TargetFrequency

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify the `TargetFrequency` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'TargetFrequency', '300')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TargetFrequency','300')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Set Target Frequency” on page 37-6
- “Customize Floating-Point IP Configuration” on page 31-39



# HDL Code Generation Pane: Optimization

---

- “Optimization Overview” on page 15-2
- “Delay Balancing and General Optimization Parameters” on page 15-3
- “RAM Mapping Parameters” on page 15-7
- “Pipelining Parameters” on page 15-9
- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15
- “Resource Sharing Parameters for Subsystems and Floating-Point IPs” on page 15-23
- “Multicycle Path Constraints Parameters” on page 15-27

## Optimization Overview

The **Optimization** pane enables you to specify various optimizations such as delay balancing, resource sharing and pipelining. To improve the area and timing of your design on the target hardware, specify these settings. This pane also contains a **Multicycle Path Constraints** section. Use the settings in this section to specify the timing requirements that the Synthesis tool must meet for your design.

# Delay Balancing and General Optimization Parameters

## In this section...

- “Balance delays” on page 15-3
- “Transform non zero initial value delay” on page 15-4
- “Multiplier partitioning threshold” on page 15-5
- “Remove Unused Ports” on page 15-6

This page describes configuration parameters that reside in the **HDL Code Generation > Target > General** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can use delay balancing to match delays introduced by optimizations, and partition multipliers based on a threshold value.

## Balance delays

When you enable certain optimizations such as pipelining or resource sharing, or specify certain block implementations and generate code, HDL Coder introduces pipeline delays along certain signal paths in your model. By default, the **Balance delays** setting is enabled. The code generator detects these pipeline delays introduced along one path and then inserts matching delays on other paths.

To make sure that the generated model after HDL code generation is functionally equivalent to the original Simulink model, leave this setting enabled. If you disable this setting, HDL Coder generates a warning that numerical differences can occur in the validation model. To fix this warning, enable **Balance delays** on the model or run the model check “Check delay balancing setting” on page 38-11.

### Settings

**Default:** On



On

Enables delay balancing on your model. If HDL Coder detects introduction of new delays along one path, matching delays are inserted on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model.



Off

The latency along signal paths might not be balanced, and the generated model might not be functionally equivalent to the original model.

### Command-Line Information

**Property:** BalanceDelays

**Type:** character vector

**Value:** ‘on’ | ‘off’

**Default:** ‘on’

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `BalanceDelays` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'BalanceDelays', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','BalanceDelays','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Delay Balancing” on page 24-63
- “BalanceDelays” on page 22-5

## Transform non zero initial value delay

### Settings

**Default:** On



On

Transform Delay blocks with nonzero **Initial condition** in your Simulink model to Delay blocks with zero **Initial condition** and some additional logic in the generated HDL code.

By using this transformation, HDL Coder can perform optimizations such as sharing, distributed pipelining, and clock-rate pipelining more effectively, and prevent an assertion from being triggered in the validation model.



Off

Do not transform Delay blocks with nonzero **Initial condition** in your Simulink model.

### Command-Line Information

**Property:** `TransformNonZeroInitValDelay`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'on'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `TransformNonZeroInitValDelay` property to `on` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'TransformNonZeroInitValDelay', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TransformNonZeroInitValDelay','on')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**`makehdl`**Multiplier partitioning threshold****Settings****Default:** `Inf`

N, where N is an integer greater than or equal to 2

Partition multipliers so that N is the maximum multiplier input bit width.

This parameter specifies the maximum input bit width for a multiplier. If at least one of the inputs to the multiplier has a bit width greater than the threshold value, the code generator splits the multiplier into smaller multipliers.

To improve hardware mapping results, set the multiplier partitioning threshold to the input bit width of the DSP or multiplier hardware on your target device.

`Inf`

Do not partition multipliers.

**Command-Line Information****Property:** `MultiplierPartitioningThreshold`**Type:** integer**Value:** integer greater than or equal to 0**Default:** `Inf`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `MultiplierPartitioningThreshold` to 16 when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MultiplierPartitioningThreshold','16')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'MultiplierPartitioningThreshold','16')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- “Multiplier promotion threshold” on page 15-19

## Remove Unused Ports

### Settings

**Default:** On

On

Removes ports in your design that are unused from the generated HDL code. This optimization preserves unused ports at the top-level DUT subsystem. All other unused ports are removed from the HDL code.

Off

Do not remove unused ports from the HDL code.

### Command-Line Information

**Property:** DeleteUnusedPorts

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `DeleteUnusedPorts` property to `off` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'DeleteUnusedPorts','off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DeleteUnusedPorts','on')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166
- `makehdl`

# RAM Mapping Parameters

## In this section...

- “Map pipeline delays to RAM” on page 15-7
- “RAM mapping threshold (bits)” on page 15-8

This page describes configuration parameters that reside in the **HDL Code Generation > Optimization > General** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can reduce the area usage on the target device by trading-off block RAMs for registers. The parameters specify whether you want to map pipeline registers in the generated code to RAM, and the minimum RAM size for mapping to block RAMs on the FPGA.

## Map pipeline delays to RAM

Map pipeline registers in the generated HDL code to RAM. Certain speed or area optimizations such as pipelining and resource sharing, or certain block implementations that you specify can insert pipeline registers in the generated HDL code. You can save area on the target device by mapping these pipeline registers to RAM.

### Settings

**Default:** Off



On

Map pipeline registers in the generated HDL code to RAM. To map these registers to block RAMs, the RAM size must be greater than or equal to the RAM mapping threshold in bits. RAM size is the product Delay length \* Word length \* Vector length \* Complex length.



Off

Do not map pipeline registers in the generated HDL code to RAM.

### Command-Line Information

**Property:** MapPipelineDelaysToRAM

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MapPipelineDelaysToRAM` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'MapPipelineDelaysToRAM','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MapPipelineDelaysToRAM','on')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- “UseRAM” on page 22-25
- “RAM Mapping for MATLAB Code” on page 8-2

**RAM mapping threshold (bits)**

Specify the minimum RAM size in bits for mapping to block RAMs. The code generator determines whether to use registers or RAM resources on the FPGA by comparing the RAM size of your design with the RAM mapping threshold that you specify.

**Settings**

**Default:** 256

The RAM mapping threshold must be an integer greater than or equal to zero. HDL Coder uses the threshold to determine whether or not to map the following elements to block RAMs instead of to registers:

- Delay blocks
- Persistent arrays in MATLAB Function blocks

**Command-Line Information**

**Property:** `RAMMappingThreshold`

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 256

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can set the `RAMMappingThreshold` to 1024 when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'RAMMappingThreshold','1024')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','RAMMappingThreshold','1024')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- “UseRAM” on page 22-25
- “RAM Mapping for MATLAB Code” on page 8-2

# Pipelining Parameters

## In this section...

- “Hierarchical distributed pipelining” on page 15-9
- “Distributed pipelining priority” on page 15-10
- “Clock-rate pipelining” on page 15-11
- “Allow clock-rate pipelining of DUT output ports” on page 15-12
- “Adaptive pipelining” on page 15-13
- “Preserve design delays” on page 15-14

This documentation page describes configuration parameters that reside in the **HDL Code Generation > Optimization > Pipelining** tab of the Configuration Parameters dialog box. Using the parameters in this section, you can improve the timing of your design on the target device. .

## Hierarchical distributed pipelining

Hierarchical distributed pipelining extends the scope of distributed pipelining by distributing delays across subsystem hierarchies. This optimization moves the delays within a Subsystem while preserving the hierarchy.

### Settings

**Default:** Off

On

Enable retiming across a subsystem hierarchy. HDL Coder applies retiming hierarchically from the top-level Subsystem. To move delays inside a Subsystem, in the HDL Block Properties for that Subsystem, set **DistributedPipelining** to on. Hierarchical distributed pipelining stops distributing delays when it reaches a Subsystem that has **DistributedPipelining** set to off.

Off

Distributes pipelines within a Subsystem, if you have **DistributedPipelining** set to on for that Subsystem.

### Dependency

If you select the **Preserve design delays** check box, distributed pipelining does not move the design delays.

### Command-Line Information

**Property:** `HierarchicalDistPipelining`

**Type:** character vector

**Value:** ‘on’ | ‘off’

**Default:** ‘off’

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `HierarchicalDistPipelining` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'HierarchicalDistPipelining','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HierarchicalDistPipelining','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Distributed Pipelining” on page 24-101
- “DistributedPipelining” on page 22-8

## Distributed pipelining priority

Specify the priority for your distributed pipelining algorithm.

### Settings

#### Default: Numerical Integrity

##### Numerical Integrity

Prioritize numerical integrity when distributing pipeline registers.

This option uses a conservative retiming algorithm that does not move registers across a component if the functional equivalence to the original design is unknown.

##### Performance

Prioritize performance over numerical integrity.

Use this option if your design requires a higher clock frequency and the Simulink behavior does not need to strictly match the generated code behavior. This option uses a more aggressive retiming algorithm that moves registers across a component even if the modified design’s functional equivalence to the original design is unknown.

### Command-Line Information

#### Property: DistributedPipeliningPriority

Type: character vector

Value: ‘NumericalIntegrity’ | ‘Performance’

Default: ‘NumericalIntegrity’

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `DistributedPipeliningPriority` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'DistributedPipeliningPriority','Performance')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DistributedPipeliningPriority','Performance')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Distributed Pipelining” on page 24-101
- “DistributedPipelining” on page 22-8

## Clock-rate pipelining

If your design contains multicycle paths, use clock-rate pipelining to insert pipeline registers at a clock rate that is faster than the data rate. This optimization improves the clock frequency and reduces the area usage without introducing additional latency. Clock-rate pipelining does not affect existing design delays in your model. It is an alternative to using multicycle path constraints with your synthesis tool.

### Settings

**Default:** On



On

Insert pipeline registers at the clock rate for multi-cycle paths.



Off

Insert pipeline registers at the data rate for multi-cycle paths.

### Dependency

If you specify an **Oversampling factor** greater than one, make sure that you select the **Clock-rate pipelining** check box. Clock-rate pipelining identifies regions in your model that run at the same slow data rate and are delimited by Delay blocks or blocks that introduce a rate transition. The code generator converts these regions to the faster clock rate by introducing Repeat blocks at the input of the region and Rate Transition blocks at the output of the region.

### Command-Line Information

**Property:** `ClockRatePipelining`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'on'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ClockRatePipelining` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ClockRatePipelining','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockRatePipelining','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Oversampling factor” on page 17-15
- “ClockRatePipelining” on page 22-5
- “Clock-Rate Pipelining” on page 24-114

## Allow clock-rate pipelining of DUT output ports

For DUT output ports, insert pipeline registers at the clock rate instead of the data rate.

### Settings

**Default:** Off

On

At DUT output ports, insert pipeline registers at clock rate.

Off

At DUT output ports, insert pipeline registers at data rate.

### Dependency

When you specify this parameter, make sure that you select the **Clock-rate pipelining** check box.

### Command-Line Information

**Property:** `ClockRatePipelineOutputPorts`

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ClockRatePipelineOutputPorts` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ClockRatePipelineOutputPorts','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockRatePipelineOutputPorts','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “ClockRatePipelining” on page 22-5
- “Clock-Rate Pipelining” on page 24-114
- “Oversampling factor” on page 17-15

## Adaptive pipelining

Use this parameter to insert pipeline registers to the blocks in your design, reduce the area usage, and maximize the achievable clock frequency on the target FPGA device.

### Settings

**Default:** On



Insert adaptive pipeline registers in your design. For HDL Coder to insert adaptive pipelines, you must specify the synthesis tool.



Do not insert adaptive pipeline registers.

### Dependency

When you specify this parameter, in the **HDL Code Generation > Target** pane, specify the **Synthesis Tool**. If your design has multipliers, specify the **Synthesis Tool** and the **Target Frequency (MHz)** for adaptive pipeline insertion.

### Command-Line Information

**Property:** AdaptivePipelining

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `AdaptivePipelining` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'AdaptivePipelining','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','AdaptivePipelining','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Adaptive Pipelining” on page 24-130
- “AdaptivePipelining” on page 22-4

## Preserve design delays

### Settings

**Default:** Off

On

Prevent distributed pipelining from moving design delays.

Off

Do not prevent distributed pipelining from moving design delays.

### Command-Line Information

**Property:** PreserveDesignDelays

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `PreserveDesignDelays` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'PreserveDesignDelays','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','PreserveDesignDelays','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Distributed Pipelining” on page 24-101
- “DistributedPipelining” on page 22-8

# Resource Sharing Parameters for Adders and Multipliers

## In this section...

- “Share Adders” on page 15-15
- “Adder sharing minimum bitwidth” on page 15-16
- “Share Multipliers” on page 15-17
- “Multiplier sharing minimum bitwidth” on page 15-18
- “Multiplier promotion threshold” on page 15-19
- “Share Multiply-Add blocks” on page 15-20
- “Multiply-Add block sharing minimum bitwidth” on page 15-21

This page describes configuration parameters that reside in the **HDL Code Generation > Optimization > Resource sharing** tab of the Configuration Parameters dialog box. Enable these parameters to save resources on the target device by specifying whether to share adders and multipliers in your design, and the minimum sharing bitwidth.

## Share Adders

Enable this parameter to share adders with the resource sharing optimization. Resource sharing identifies Add or Sum blocks in your design that have two inputs and replaces them with a single Add or Sum block. This optimization saves area on the target FPGA device.

### Settings

**Default:** Off

On

When resource sharing is enabled, this optimization shares adders with a bit width greater than or equal to the **Adder sharing minimum bitwidth**.

Off

Do not share adders.

### Dependency

- To share adders in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Adder sharing minimum bitwidth**, the code generator shares adders that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing adders is zero.

### Command-Line Information

**Property:** ShareAdders

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareAdders` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ShareAdders','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareAdders','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “SharingFactor” on page 22-23
- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Subsystems and Floating-Point IPs” on page 15-23

## Adder sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share adders with the resource sharing optimization.

### Settings

**Default:** 0

01

No minimum bit width for shared adders.

N, where N is an integer greater than 1

When resource sharing and adder sharing are enabled, share adders with a bit width greater than or equal to N.

### Dependency

To share adders in your design:

- In the **Resource Sharing** tab, enable the **Adders** setting.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** `AdderSharingMinimumBitwidth`

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `AdderSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'AdderSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','AdderSharingMinimumBitwidth',16)
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “SharingFactor” on page 22-23
- “Resource Sharing” on page 24-32

## Share Multipliers

Enable this parameter to share multipliers with the resource sharing optimization. Resource sharing identifies Product or Gain blocks in your design that have two inputs and replaces them with a single Product or Gain block. This optimization saves area on the target FPGA device. Share multipliers with the resource sharing optimization.

### Settings

**Default:** On



On

When resource sharing is enabled, share multipliers with a bit width greater than or equal to the **Multiplier sharing minimum bitwidth**. For successfully sharing multipliers, the input fixed-point data types must have the same wordlength. The fraction lengths and signs of the fixed-point data types can be different.



Off

Do not share multipliers.

### Dependency

- To share multipliers in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Multiplier sharing minimum bitwidth**, the code generator shares multipliers that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing multipliers is zero.

### Command-Line Information

**Property:** `ShareMultipliers`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'on'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultipliers` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ShareMultipliers','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareMultipliers','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “SharingFactor” on page 22-23
- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Subsystems and Floating-Point IPs” on page 15-23

## Multiplier sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share multipliers with the resource sharing optimization.

### Settings

**Default:** 0

01

No minimum bit width for shared multipliers.

N, where N is an integer greater than 1

When resource sharing and multiplier sharing are enabled, share multipliers with a bit width greater than or equal to N.

### Dependency

To share multipliers in your design:

- In the **Resource Sharing** tab, make sure that the **Multipliers** check box is selected.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** `MultiplierSharingMinimumBitwidth`

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'MultiplierSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MultiplierSharingMinimumBitwidth',16)
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “SharingFactor” on page 22-23
- “Resource Sharing” on page 24-32

## Multiplier promotion threshold

To share smaller multipliers with other larger multipliers by using the resource sharing optimization, specify the multiplier promotion threshold. This threshold specifies the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

### Settings

**Default:** 0

0

No difference in word-length between the multipliers. In other words, HDL Coder shares multipliers that have the same word-lengths.

N, where N is an integer greater than 0

Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers. HDL Coder promotes and shares multipliers with different word-lengths, if the difference in word-lengths is less than or equal to N.

### Dependency

To share multipliers in your design:

- In the **Resource Sharing** tab, make sure that the **Multipliers** check box is selected.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** `MultiplierPromotionThreshold`

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierPromotionThreshold` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'MultiplierPromotionThreshold',8)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MultiplierPromotionThreshold',8)
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “SharingFactor” on page 22-23
- “Resource Sharing” on page 24-32

## Share Multiply-Add blocks

Share Multiply-Add blocks with the resource sharing optimization.

### Settings

**Default:** On



On

When resource sharing is enabled, share Multiply-Add blocks with a bit width greater than or equal to **Multiply-Add block sharing minimum bitwidth**.



Off

Do not share Multiply-Add blocks.

### Dependency

- To share Multiply-Add blocks in your design, in the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.
- When you specify the **Multiply-Add block sharing minimum bitwidth**, the code generator shares Multiply-Add blocks that have a bit width greater than or equal to the minimum bit width. The default minimum bit width for sharing Multiply-Add blocks is zero.

### Command-Line Information

**Property:** ShareMultiplyAdds

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultiplyAdds` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'ShareMultiplyAdds','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareMultiplyAdds','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15
- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15

## Multiply-Add block sharing minimum bitwidth

Use this parameter to specify the minimum bit width that is required to share Multiply-Add with the resource sharing optimization.

### Settings

**Default:** 0

01

No minimum bit width for shared Multiply-Add blocks.

N, where N is an integer greater than 1

When resource sharing and Multiply-Add block sharing are enabled, share Multiply-Add blocks with a bit width greater than or equal to N.

### Dependency

To share Multiply-Add blocks in your design:

- In the **Resource Sharing** tab, make sure that the **Multiply-Add blocks** check box is selected.
- In the HDL Block Properties for the DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** MultiplierAddSharingMinimumBitwidth

**Type:** integer

**Value:** integer greater than or equal to 0

**Default:** 0

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `MultiplierAddSharingMinimumBitwidth` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'MultiplierAddSharingMinimumBitwidth',16)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MultiplierAddSharingMinimumBitwidth',16)
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15
- “Resource Sharing” on page 24-32

# Resource Sharing Parameters for Subsystems and Floating-Point IPs

## In this section...

- “Share Atomic subsystems” on page 15-23
- “Share MATLAB Function blocks” on page 15-24
- “Share Floating-Point IPs” on page 15-25

This page describes configuration parameters that reside in the **HDL Code Generation > Optimization > Resource sharing** tab of the Configuration Parameters dialog box. Enable these parameters to save resources on the target device by specifying whether to share atomic subsystems, MATLAB Function blocks, and floating-point IPs in your design.

## Share Atomic subsystems

Share atomic subsystems with the resource sharing optimization.

### Settings

**Default:** On



On

When resource sharing is enabled, share atomic subsystems.



Off

Do not share atomic subsystems.

### Dependency

To share Atomic Subsystem blocks in your design, in the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** ShareAtomicSubsystems

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMultiplyAdds` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'ShareAtomicSubsystems','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareAtomicSubsystems','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15
- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15

## Share MATLAB Function blocks

Share MATLAB Function blocks with the resource sharing optimization.

### Settings

**Default:** On



On

When resource sharing is enabled, share MATLAB Function blocks.



Off

Do not share MATLAB Function blocks.

### Dependency

To share MATLAB Function blocks in your design, in the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**.

### Command-Line Information

**Property:** ShareMATLABBlocks

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareMATLABBlocks` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'ShareMATLABBlocks','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ShareMATLABBlocks','on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15

- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15

## Share Floating-Point IPs

### Settings

**Default:** On



On  
When you enable resource sharing, HDL Coder shares floating-point IP blocks.



Off  
Do not share floating-point IP blocks.

### Dependency

To share floating-point IPs:

- In the HDL Block Properties for the parent DUT Subsystem, specify the **SharingFactor**. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.
- In the **HDL Code Generation > Global Settings > Floating Point Target** tab, set the **Floating Point IP Library** to a value other than None.

### Command-Line Information

**Property:** ShareFloatingPointIP

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can use the `ShareFloatingPointIP` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ShareFloatingPointIP', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ShareFloatingPointIP', 'on')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15
- “Resource Sharing” on page 24-32
- “Resource Sharing Parameters for Adders and Multipliers” on page 15-15

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80

# Multicycle Path Constraints Parameters

## In this section...

["Enable based constraints" on page 15-27](#)

["Register-to-register path info" on page 15-28](#)

This section contains parameters in the **Multicycle Path Constraints** section of the **HDL Code Generation > Optimization** pane of the Configuration Parameters dialog box.

Synthesis tools require that data propagates from a source register to a destination register within one clock cycle. However, multicycle paths cannot complete their execution within one clock cycle and therefore cannot meet the timing requirements. To meet the timing requirement of multicycle paths, use the parameters in this section to generate a register-to-register path information file or to generate enable-based constraints that uses the timing controller enable signals.

## Enable based constraints

To meet the timing requirement of multicycle paths in your Simulink design in single clock mode, use enable-based constraints. Enable-based constraints relax the timing requirement by allowing multiple clock cycles for data to propagate between the registers. The constraints use the timing controller enable signals to create enable-based register groups, with registers in each group driven by the same clock enable.

### Settings

**Default:** Off

On

When you enable this setting and generate HDL code, HDL Coder generates a constraints file with the naming convention `dutname_constraints`. The format of the file name depends on the synthesis tool that you specify. The constraints file defines the timing requirements of multicycle paths and contains information about the clock multiples for calculating the setup and hold time information.

Off

Do not generate a multicycle path constraints file.

### Dependency

If you select the **Enable based constraints** check box, make sure that you clear the **Clock-rate pipelining** check box. Using enable-based multicycle path constraints is an alternative to the clock-rate pipelining optimization. You can clear the **Clock-rate pipelining** check box in the **HDL Code Generation > Target > Pipelining** tab.

### Command-Line Information

**Parameter:** MulticyclePathConstraints

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MulticyclePathConstraints` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MulticyclePathConstraints', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'MulticyclePathConstraints', 'on')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 23-26
- “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 23-32

## Register-to-register path info

Generate a text file that reports multicycle path constraint information. The text file describes one or more multicycle path constraints that is agnostic to the synthesis tool. You must convert this information to the format required by the synthesis tool. It is recommended that you use the enable-based constraints setting instead to meet the timing requirements of multicycle paths. When you use that setting, the generated constraints are more robust to name changes in synthesis tools, and are supported with Xilinx Vivado, Xilinx ISE, and Altera Quartus II.

### Settings

**Default:** Off

On

Generate a text file that reports multicycle path constraint information, for use with synthesis tools.

The file name for the multicycle path information file derives from the name of the DUT and the postfix '`_constraints`', as follows:

*DUTname\_constraints.txt*

For example, if the DUT name is `symmetric_fir`, the name of the multicycle path information file is `symmetric_fir_constraints.txt`.

Off

Do not generate a multicycle path information file.

**Command-Line Information**

**Parameter:** MulticyclePathInfo

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can enable the `MulticyclePathInfo` setting when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MulticyclePathInfo','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MulticyclePathInfo','on')
makehdl('sfir_fixed/symmetric_fir')
```

**See Also**

- `makehdl`
- “Generate Multicycle Path Information Files” on page 23-19



# HDL Code Generation Pane: Floating Point

---

- “Floating Point Overview” on page 16-2
- “Floating Point IP Library” on page 16-3
- “Native Floating Point Parameters” on page 16-4
- “FPGA Floating-Point Library Targeting Parameters” on page 16-8

## Floating Point Overview

The **Floating Point** pane enables you to specify the floating point IP library. You can specify whether to generate code the native floating point support in HDL Coder or by instantiating the third-party Intel or Xilinx floating-point libraries.

# Floating Point IP Library

This parameter resides in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use this parameter to specify the floating-point target library.

## Settings

**Default:** NONE

The options are:

None

Select this option if you do not want the design to map to floating-point target libraries.

**Native Floating Point**

Specify native floating-point as the library. You can specify the latency strategy and whether to handle denormal numbers in your design.

**Altera Megafunctions (ALTERA FP FUNCTIONS)**

Specify Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library. You can provide the IP Target frequency.

**Altera Megafunctions (ALTFP)**

Specify Altera Megafunctions (ALTFP) as the floating-point target library. You can provide the objective and latency strategy for the IP.

**Xilinx LogiCORE**

Specify Xilinx LogiCORE® as the floating-point target library. You can provide the objective and latency strategy for the IP.

## Command-Line Information

To set the floating-point library:

- Create a floating-point target configuration object for the floating-point library. This example shows how to create an `hdlcoder.FloatingPointTargetConfig` object for the Native Floating Point library:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

## See Also

- `createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20

## Native Floating Point Parameters

This section contains parameters in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use these parameters to specify the latency strategy, whether to handle denormal numbers in your design, and how to perform mantissa multiplication. To specify these settings, **Floating Point IP Library** must be set to Native Floating Point.

### Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with native floating-point libraries.

#### Settings

**Default:** MAX

The options are:

MIN

Maps to minimum latency for the native floating-point libraries.

MAX

Maps to maximum latency for the native floating-point libraries.

ZERO

Does not use any latency for the native floating-point libraries.

#### Dependency

To specify this parameter, set the **Floating Point IP Library** to Native Floating Point.

#### Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for Native Floating Point as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- 2 Specify the **LatencyStrategy** property of the **LibrarySettings** attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the **sfir\_single** model and generate HDL code for the **symmetric\_fir** subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

#### See also

- “LatencyStrategy” on page 22-33
- `createFloatingPointTargetConfig`

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Latency Considerations with Native Floating Point” on page 10-96

## Handle Denormals

Specify whether you want to handle denormal numbers in your design. Denormal numbers are nonzero numbers that are smaller than the smallest normal number.

### Settings

**Default:** Off



On

Inserts additional logic to handle the denormal numbers in your design.



Off

Does not add additional logic to handle the denormal numbers in your design. If the input is a denormal value, HDL Coder treats the value as zero before performing computations.

### Dependency

To specify this parameter, set the **Floating Point IP Library** to Native Floating Point.

### Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for Native Floating Point as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```

- 2 Specify the `HandleDenormals` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.HandleDenormals = 'on';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

### See also

- “HandleDenormals” on page 22-31
- `createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Numeric Considerations with Native Floating-Point” on page 10-84

## Mantissa Multiplier Strategy

Specify how you want HDL Coder to implement the mantissa multiplication operation when you have Product blocks in your design.

### Settings

**Default:** Auto

The options are:

#### Auto

This default option automatically determines how to implement the mantissa multiplication depending on the **Synthesis tool** that you specify.

- If you do not specify a **Synthesis tool**, this setting selects the **Full Multiplier** implementation by default.
- If you specify **Altera Quartus II** as the **Synthesis tool**, this setting selects the **Full Multiplier** implementation.
- If you specify **Xilinx Vivado** or **Xilinx ISE** as the **Synthesis tool**, this setting selects the **Part Multiplier Part AddShift** implementation.

#### Full Multiplier

Specify this option to use only multipliers for implementing the mantissa multiplication. The multipliers can utilize DSP units on the target device.

#### Part Multiplier Part AddShift

Specify this option to split the implementation into two parts. One part is implemented with multipliers. The other part is implemented with a combination of adders and shifters. The multipliers can utilize the DSP units on the target device. The combination of adders and shifters does not utilize the DSP.

#### No Multiplier Full AddShift

Select this option to use a combination of adders and multipliers to implement the mantissa multiplication. This option does not utilize DSP units on the target device. You can also use this option if your target device does not contain DSP units.

### Dependency

To specify this parameter, set the **Floating Point IP Library** to **Native Floating Point**.

### Command-Line Information

To specify the latency strategy:

- 1 Create a floating-point target configuration object for **Native Floating Point** as the floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
```
- 2 Specify the **MantissaMultiplyStrategy** property of the **LibrarySettings** attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'PartMultiplierPartAddShift';
```

- 3** Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

**See also**

- “MantissaMultiplyStrategy” on page 22-36
- `createFloatingPointTargetConfig`
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

## FPGA Floating-Point Library Targeting Parameters

This section contains parameters in the **HDL Code Generation > Floating Point** pane of the Configuration Parameters dialog box. Use these parameters to specify the latency strategy, objective, and whether to initialize the pipeline registers in the floating-point target IP to zero.

### Initialize IP Pipelines To Zero

Inserts additional logic during HDL code generation to initialize the values of pipeline registers in the Altera floating-point target IP to zero. If you do not select this check box, HDL Coder reports a warning during HDL code generation.

#### Settings

**Default:** On

On

Inserts additional logic to initialize pipeline registers in the floating-point target IP to zero.

Off

Does not add additional logic to initialize pipeline registers in the floating-point target IP to zero.

#### Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTERA FP FUNCTIONS). Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

#### Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('AlteraFFunctions');
```

- 2 Specify the `InitializeIPPIPelinesToZero` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.InitializeIPPIPelinesToZero = 0;
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

#### See Also

- `createFloatingPointTargetConfig`
- “Target Frequency Parameter” on page 14-8
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20

## Latency Strategy

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or ALTFP Altera megafunction IPs.

### Settings

**Default:** MIN

The options are:

MIN

Maps to minimum latency for the specified floating-point target IP.

MAX

Maps to maximum latency for the specified floating-point target IP.

### Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTFP) or Xilinx LogiCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

### Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

- 2 Specify the `LatencyStrategy` property of the `LibrarySettings` attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

### See also

- `createFloatingPointTargetConfig`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “Customize Floating-Point IP Configuration” on page 31-39

## Objective

Specify whether you want to optimize the design for speed or area when mapping to floating-point target libraries.

## Settings

**Default:** SPEED

The options are:

NONE

Select this option if you do not want to optimize the design for speed or area.

SPEED

Select this option to optimize the design for speed.

AREA

Select this option to optimize the design for area.

## Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafunctions (ALTFP) or Xilinx LogicCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

## Command-Line Information

To specify this setting:

- 1 Create a floating-point target configuration object with Altera Megafunctions (ALTERA FP FUNCTIONS) as the floating-point target library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

- 2 Specify the **Objective** property of the **LibrarySettings** attribute of the floating-point target configuration object.

```
fpconfig.LibrarySettings.Objective = 'AREA';
```

- 3 Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

## See also

- `createFloatingPointTargetConfig`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “Customize Floating-Point IP Configuration” on page 31-39

## IP Settings

The **IP Settings** section has an IP configuration table with the IP names and data types and additional options to specify a custom latency and any extra arguments.

The options in the IP configuration table depend on the library that you specify.

- If you specify the ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS) library, HDL Coder infers the latency value from the **Target Frequency (MHz)** value.

- If you specify the ALTERA MEGAFUNCTION (ALTFP) or XILINX LOGICORE libraries, HDL Coder infers the IP latency from the **Latency Strategy** setting. The IP configuration table has two additional columns, **MinLatency** and **MaxLatency**, that contain the minimum and maximum latency values for each IP in the table.

The IP configuration table has these sections:

- Name**: Contains a list of IP names that HDL Coder map the Simulink blocks to, such as ABS, ADDSUB, and CONVERT.
- DataType**: Contains a list of IP data types for each IP in the table. These are mostly SINGLE and DOUBLE data types. The CONVERT IP blocks can have DOUBLE\_TO\_NUMERICTYPE, NUMERICTYPE\_TO\_DOUBLE data types, and so on.
- Latency**: The default latency value of -1 means that the IP inherits the latency value from the target frequency or the latency strategy setting depending on the library that you choose. To customize the latency of the IP that your Simulink blocks map to, enter your own custom value for the latency.
- ExtraArgs**: Specify any additional settings that is specific to the IP.

For example, if you have an Add block with Single data types in your Simulink model, HDL Coder maps the block to the **ADDSUB** IP. If you want to specify a custom latency value, say 8, for the IP, enter the value in the **Latency** column for the IP.

#### IP Settings

Customize data type conversion IP for: `SINGLE_TO_NUMERICTYPE(1, 32, 16)`

Name	DataType	MinLatency	MaxLatency	Latency	ExtraArgs
ADDSUB	DOUBLE	12	12	-1	
ADDSUB	SINGLE	12	12	8	<code>CSET c_mult_usage=...</code>
CONVERT	DOUBLE_TO_N...	6	6	-1	
CONVERT	NUMERICTYPE_...	6	6	-1	

`cmultusage` is a parameter that you can specify with the Xilinx LogiCORE libraries.

#### Dependency

To specify this parameter, set the **Floating Point IP Library** to Altera Megafuctions (ALTFP) or Xilinx LogiCORE. Before you set the floating-point library, specify the path to your synthesis tool by using the `hdlsetuptoolpath` function.

#### Command-Line Information

To specify this setting:

- Create a floating-point target configuration object with Altera Megafuctions (ALTERA FP FUNCTIONS) as the floating-point target library.  

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```
- To view the floating-point IP configuration, use the `IPConfig` object.  

```
fpconfig.IPConfig
```

- 3** To customize the latency or specify additional arguments, use the `customize` method.

```
fpconfig.IPCConfig.customize('ADDSUB','Single','Latency',6);
```

- 4** Set the floating-point target configuration on the model and then generate HDL code. This example shows how to set the configuration on the `sfir_single` model and generate HDL code for the `symmetric_fir` subsystem:

```
hdlset_param('sfir_single','FloatingPointTargetConfig',fpconfig)
makehdl('sfir_single/symmetric_fir')
```

#### See also

- `createFloatingPointTargetConfig`
- `customize`
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “Customize Floating-Point IP Configuration” on page 31-39

# HDL Code Generation Pane: Global Settings

---

- “Global Settings Overview” on page 17-3
- “Clock Settings and Timing Controller Postfix Parameters” on page 17-4
- “Reset Settings and Parameters” on page 17-8
- “Clock Enable Settings and Parameters” on page 17-12
- “Oversampling factor” on page 17-15
- “Comment in header” on page 17-17
- “Language-Specific File Extension Parameters” on page 17-19
- “Language-Specific Identifiers and Postfix Parameters” on page 17-21
- “Split entity and architecture Parameters” on page 17-25
- “Complex Signals Postfix Parameters” on page 17-28
- “VHDL Architecture and Library Name and Code for Model Reference Parameters” on page 17-30
- “Generate Statement and Vector and Component Instance Label Parameters” on page 17-32
- “Input and Output Port and Clock Enable Output Type Parameters” on page 17-35
- “Minimize Clock Enables and Reset Signal Parameters” on page 17-37
- “Using Trigger Signals and Scalarization and Test Point DUT Port Generation Parameters” on page 17-41
- “RTL Annotation Parameters” on page 17-44
- “RTL Customization Parameters for Constants and MATLAB Function Blocks” on page 17-49
- “RTL Customization Parameters for RAMs” on page 17-51
- “No-reset registers initialization” on page 17-53
- “RTL Style Parameters” on page 17-55
- “Timing Controller Settings” on page 17-60
- “File Comment Customization Parameters” on page 17-62
- “Choose Coding Standard and Report Option Parameters” on page 17-64
- “Basic Coding Practices Parameters” on page 17-66
- “RTL Description Rules for clock enables and resets Parameters” on page 17-71
- “RTL Description Rules for Conditional Parameters” on page 17-74
- “Other RTL Description Rule Parameters” on page 17-77
- “RTL Design Rule Parameters” on page 17-80
- “Model Generation Parameters for HDL Code” on page 17-82
- “Naming and Layout Options for Model Generation” on page 17-85
- “Diagnostic Parameters for Optimizations” on page 17-89
- “Diagnostic Parameters for Reals and Black Box Interfaces” on page 17-92

- “Code Generation Output Parameter” on page 17-94

## Global Settings Overview

The **Global Settings** pane enables you to specify detailed characteristics of the generated code, such as HDL element naming, coding style, whether you want the HDL code to conform to coding standards, and diagnostics and additional options for model generation and HDL code generation.

## Clock Settings and Timing Controller Postfix Parameters

This page describes configuration parameters that reside in the **Clock settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to specify the clock signal name, the number of clock inputs, the active clock edge, and the postfix for the clock process and the timing controller.

### Clock input port

Specify the name for the clock input port in generated HDL code.

#### Settings

**Default:** clk

Enter the clock signal name in generated HDL code as a character vector.

For a generated entity `my_filter`, if you specify '`filter_clock`' as the clock signal name, the entity declaration is as shown in this code snippet:

```
ENTITY my_filter IS
  PORT( filter_clock : IN std_logic;
        clk_enable   : IN std_logic;
        reset        : IN std_logic;
        my_filter_in : IN std_logic_vector (15 DOWNTO 0); -- sfix16_En15
        my_filter_out: OUT std_logic_vector (15 DOWNTO 0); -- sfix16_En15
      );
END my_filter;
```

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

#### Command-Line Information

**Property:** ClockInputPort

**Type:** character vector

**Value:** A valid identifier in the target language

**Default:** 'clk'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockInputPort','system_clk')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockInputPort','system_clk')
```

**See Also**`makehdl`

## Clock inputs

Specify generation of single or multiple clock inputs.

**Settings**

**Default:** Single

Single

Generates a single clock input for the DUT. If the DUT is multirate, the input clock is the master clock rate, and a timing controller is synthesized to generate additional clocks as required. It is recommended that you use a single clock signal in your design.

Multiple

Generates a unique clock for each Simulink rate in the DUT. The number of timing controllers generated depends on the contents of the DUT. The oversample factor must be 1 (default) to specify multiple clocks.

**Command-Line Information**

**Property:** `ClockInputs`

**Type:** character vector

**Value:** 'Single' | 'Multiple'

**Default:** 'Single'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockInputs','Multiple')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockInputs','Multiple')
```

**See Also**

- `makehdl`
- “Check clock settings” on page 38-43

## Clock edge

Specify the active clock edge that triggers Verilog `always` blocks or VHDL `process` blocks in the generated HDL code.

## Settings

### Default: Rising.

#### Rising

The rising edge, or 0-to-1 transition, is the active clock edge.

#### Falling

The falling edge, or 1-to-0 transition, is the active clock edge.

### Command-Line Information

**Property:** ClockEdge

**Type:** character vector

**Value:** 'Rising' | 'Falling'

**Default:** 'Rising'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockEdge','Falling')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockEdge','Falling')
```

## See Also

- `makehdl`
- “Check clock settings” on page 38-43

## Clocked process postfix

Specify the postfix as a character vector. The code generator appends this postfix to HDL clock process names.

## Settings

### Default: \_process

HDL Coder uses `process` blocks for register operations. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the code generator derives the label `delay_pipeline_process` in the following block declaration from the register name `delay_pipeline` and the default postfix `_process`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
    .
    .
    .
```

**Command-Line Information****Property:** ClockProcessPostfix**Type:** character vector**Default:** '\_process'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockProcessPostfix','delay_postfix')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ClockProcessPostfix','delay_postfix')
```

**See Also**`makehdl`**Timing controller postfix**

Specify the postfix as a character vector. The code generator appends this suffix to the DUT name to form the timing controller name.

**Settings****Default:** '\_tc'

A timing controller file is generated if the design uses multiple rates, for example:

- When code is generated for a multirate model.
- When an area or speed optimization, or block architecture, introduces local multirate.

The timing controller name is based on the name of the DUT. For example, if the name of your DUT is `my_test`, by default, HDL Coder adds the postfix `_tc` to form the timing controller name, `my_test_tc`.

**Command-Line Information****Property:** TimingControllerPostfix**Type:** character vector**Default:** '\_tc'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Reset Settings and Parameters

This page describes parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the reset name, whether to use a synchronous or asynchronous reset, and whether the reset is asserted active-high or active-low.

### Reset type

Specify whether to use asynchronous or synchronous reset logic when generating HDL code for registers. It is recommended that you specify the **Reset type** as **Synchronous** when you use a Xilinx device and **Asynchronous** when you use an Altera device.

#### Settings

**Default:** Asynchronous

##### Asynchronous

Use asynchronous reset logic. This reset logic samples the reset independent of the clock signal.

The following process block, generated by a Unit Delay block, illustrates the use of asynchronous resets. When the reset signal is asserted, the process block performs a reset, without checking for a clock event.

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
        IF clk_enable = '1' THEN
            Unit_Delay1_out1 <= signed(x_in);
        END IF;
    END IF;
END PROCESS Unit_Delay1_process;
```

##### Synchronous

Use synchronous reset logic. This reset logic samples the reset with respect to the clock signal.

The following process block, generated by a Unit Delay block, checks for a clock event, the rising edge, before performing a reset:

```
Unit_Delay1_process : PROCESS (clk)
BEGIN
    IF rising_edge(clk) THEN
        IF reset = '1' THEN
            Unit_Delay1_out1 <= (OTHERS => '0');
        ELSIF clk_enable = '1' THEN
            Unit_Delay1_out1 <= signed(x_in);
        END IF;
    END IF;
END PROCESS Unit_Delay1_process;
```

### Command-Line Information

**Property:** ResetType

**Type:** character vector

**Value:** 'async' | 'sync'

**Default:** 'async'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `sync` as the `ResetType` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ResetType','async')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ResetType','async')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Check for global reset setting for Xilinx and Altera devices” on page 38-7

## Reset asserted level

Specify whether the asserted or active level of the reset input signal is active-high or active-low.

### Settings

**Default:** Active-high

Active-high

Specify that the asserted level of reset input signal is active-high. For example, the following code fragment checks whether `reset` is active high before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .

```

Active-low

Specify that the asserted level of reset input signal is active-low. For example, the following code fragment checks whether `reset` is active low before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '0' THEN
        delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    .
    .

```

## Dependency

If you input a logic high value to the **Reset input port**, to reset the registers in your design, set **Reset asserted level** to Active-high. If you input a logic low value to the **Reset input port**, to reset the registers in your design, set **Reset asserted level** to Active-low.

### Command-Line Information

**Property:** ResetAssertedLevel

**Type:** character vector

**Value:** 'active-high' | 'active-low'

**Default:** 'active-high'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ResetAssertedLevel','active-high')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','ResetAssertedLevel','active-high')
```

### See Also

- `makehdl`
- “Reset input port” on page 17-10

## Reset input port

Enter the name for the reset input port in generated HDL code.

### Settings

**Default:** `reset`

Enter a character vector for the reset input port name in generated HDL code.

For example, if you override the default with '`chip_reset`' for the generating system `myfilter`, the generated entity declaration might look as follows:

```
ENTITY myfilter IS
  PORT( clk           : IN  std_logic;
        clk_enable    : IN  std_logic;
        chip_reset    : IN  std_logic;
        myfilter_in   : IN  std_logic_vector (15 DOWNTO 0);
        myfilter_out  : OUT std_logic_vector (15 DOWNTO 0);
      );
END myfilter;
```

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

## Dependency

If you specify active-high for **Reset asserted level**, the reset input signal is asserted active-high. To reset the registers in the entity, the input value to the **Reset input port** must be high. If you specify active-low for **Reset asserted level**, the reset input signal is asserted active-low. To reset the registers in the entity, the input value to the **Reset input port** must be low.

## Command-Line Information

**Property:** ResetInputPort

**Type:** character vector

**Value:** A valid identifier in the target language

**Default:** 'reset'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify `sync` as the `ResetType` when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'ResetInputPort','rstx')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ResetInputPort','rstx')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Reset asserted level” on page 17-9

## Clock Enable Settings and Parameters

This page describes configuration parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the name of the clock enable input port and for internal clock enable signals in the generated code.

### Clock enable input port

Specify the name for the clock enable input port in generated HDL code.

#### Settings

**Default:** `clk_enable`

Enter the clock enable input port name in generated HDL code as a character vector.

For example, if you specify '`filter_clock_enable`' for the generating subsystem `filter_subsys`, the generated entity declaration might look as follows:

```
ENTITY filter_subsys IS
  PORT( clk           : IN  std_logic;
        filter_clock_enable : IN  std_logic;
        reset            : IN  std_logic;
        filter_subsys_in   : IN  std_logic_vector (15 DOWNTO 0);
        filter_subsys_out  : OUT std_logic_vector (15 DOWNTO 0);
      );
END filter_subsys;
```

The clock enable input signal is asserted active-high (1). Thus, the input value must be high for the generated entity's registers to be updated.

If you specify a VHDL or Verilog reserved word, the code generator appends a reserved word postfix string to form a valid VHDL or Verilog identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`.

#### Command-Line Information

**Property:** `ClockEnableInputPort`

**Type:** character vector

**Value:** A valid identifier in the target language

**Default:** '`clk_enable`'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'ClockEnableInputPort','clken')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','ClockEnableInputPort','clken')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`
- “Clock Enable output port” on page 17-36
- “Clock input port” on page 17-4
- “Reset input port” on page 17-10

## Enable prefix

Specify the base name as a character vector for internal clock enables and other flow control signals in generated code.

### Settings

**Default:** 'enb'

Where only a single clock enable is generated, **Enable prefix** specifies the signal name for the internal clock enable signal.

In some cases, the code generator can generate multiple clock enable signals. For example, if you specify a cascade block implementation for certain blocks, multiple clock enable signals are generated. In such cases, **Enable prefix** specifies a base signal name for the first clock enable that is generated. For other clock enable signals, numeric tags are appended to **Enable prefix** to form unique signal names. For example, the following code fragment illustrates two clock enables that were generated when **Enable prefix** was set to 'test\_clk\_enable':

```
COMPONENT mysys_tc
  PORT( clk : IN std_logic;
        reset : IN std_logic;
        clk_enable : IN std_logic;
        test_clk_enable : OUT std_logic;
        test_clk_enable_5_1_0 : OUT std_logic
      );
END COMPONENT;
```

### Command-Line Information

**Property:** `EnablePrefix`

**Type:** character vector

**Default:** 'enb'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'EnablePrefix','int_enable')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EnablePrefix','int_enable')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

- [makehdl](#)
- “Clock Enable output port” on page 17-36
- “Clock enable input port” on page 17-12

# Oversampling factor

This configuration parameter resides in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use this parameter to specify the frequency of the global oversampling clock as a multiple of the model's base rate.

## Settings

**Default:** 1.

**Oversampling factor** specifies the factor by which the global clock signal is a multiple of the base rate at which the model operates. Use the **Oversampling factor** to integrate the DUT with a larger system that supplies timing signals to other components in the system at the global oversampling clock.

By default, HDL Coder does not generate a global oversampling clock. To generate a global oversampling clock, specify an integer greater than one. If you use a multirate DUT, make sure that other rates in the DUT divide evenly into the global oversampling rate.

Generation of the global oversampling clock affects the generated HDL code and does not affect the simulation behavior of your model.

## Dependency

- if you use multiple clocks, the **Oversampling factor** must be set to one. If you want to use an **Oversampling factor** greater than one, set **ClockInputs** to **Single**.
- If you specify an **Oversampling factor** greater than one, make sure that the clock-rate pipelining optimization is enabled. You can specify this setting in the **HDL Code Generation > Target and Optimizations > Pipelining** tab.

Clock-rate pipelining uses the **Oversampling factor** to convert the slow regions in your model that operate at the base sample rate to the faster clock rate.

## Command-Line Information

**Property:** Oversampling

**Type:** int

**Value:** integer greater than or equal to 1

**Default:** 1

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'Oversampling',5)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','Oversampling',5)
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- [makehdl](#)
- “Generate a Global Oversampling Clock” on page 23-8
- “Pipelining Parameters” on page 15-9

# Comment in header

This parameter resides in the **General** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use this parameter to specify comment lines in header of generated HDL and test bench files.

## Settings

**Default:** None

Text entered in this field generates a comment line in the header of generated model and test bench files. The code generator adds leading comment characters for the target language. When newlines or linefeeds are included, the code generator emits single-line comments for each newline.

For example, if you specify this comment '`This is a comment line.\nThis is a second line.'` for the `symmetric_fir` subsystem inside the `sfir_fixed` model and generate HDL code, the resulting header comment block appears as follows:

```
-- 
-- Module: symmetric_fir
-- Simulink Path: sfir_fixed/symmetric_fir
-- Created: 2006-11-20 15:55:25
-- Hierarchy Level: 0
--
-- This is a comment line.
-- This is a second line.
--
-- Simulink model description for sfir_fixed:
-- This model shows how to use HDL Coder to check, generate,
-- and verify HDL for a fixed-point symmetric FIR filter.
--
```

## Command-Line Information

**Property:** UserComment

**Type:** character vector

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'UserComment','This is a comment line.\nThis is a second line.')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', ...
    'UserComment','This is a comment line.\nThis is a second line.')
makehdl('sfir_fixed/symmetric_fir')
```

## See Also

- `makehdl`

- “File Comment Customization Parameters” on page 17-62
- “Generate Code with Annotations or Comments” on page 25-13

# Language-Specific File Extension Parameters

## Verilog file extension

Specify the file name extension for generated Verilog files.

### Settings

**Default:** `.v`

This field specifies the file name extension for generated Verilog files.

### Dependency

To enable this option, set the target language to Verilog. You can specify the target language by using the **Language** parameter in the **HDL Code Generation** pane.

### Command-Line Information

**Property:** `VerilogFileExtension`

**Type:** character vector

**Default:** `'.v'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir',...
    'VerilogFileExtension','.v')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','VerilogFileExtension','.v')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

- `makehdl`
- “Target Language and Folder Selection Parameters” on page 13-3

## VHDL file extension

Specify the file name extension for generated VHDL files.

### Settings

**Default:** `.vhd`

This field specifies the file name extension for generated VHDL files.

## Dependency

To enable this option, set the target language to VHDL. You can specify the target language by using the **Language** parameter in the **HDL Code Generation** pane.

### Command-Line Information

**Property:** `VHDLFileExtension`

**Type:** character vector

**Default:** '`.vhd`'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'VHDLFileExtension','.vhd')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','VHDLFileExtension','.vhd')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

- `makehdl`
- “Target Language and Folder Selection Parameters” on page 13-3

# Language-Specific Identifiers and Postfix Parameters

This section contains parameters in the **Clock Settings** section of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Using these parameters, you can specify the entity, module, and package name postfix, and the prefix for module names.

## Entity conflict postfix

Specify the text as a character vector to resolve duplicate VHDL entity or Verilog module names in generated code.

### Settings

**Default:** `_block`

The specified postfix resolves duplicate VHDL entity or Verilog module names.

For example, if HDL Coder detects two entities with the name `MyFilter`, the coder names the first entity `MyFilter` and the second entity `MyFilter_block`.

### Command-Line Information

**Property:** `EntityConflictPostfix`

**Type:** character vector

**Value:** A valid character vector in the target language

**Default:** `'_block'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'EntityConflictPostfix', '_entity')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EntityConflictPostfix','_entity')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

`makehdl`

## Package postfix

Specify a text as a character vector to append to the model or subsystem name to form name of a package file.

### Settings

**Default:** `_pkg`

HDL Coder applies this option only if a package file is required for the design.

### Dependency

This option is enabled when:

The target language (specified by the **Language** option) is VHDL.

The target language (specified by the **Language** option) is Verilog, and the **Multi-file test bench** option is selected.

### Command-Line Information

**Property:** PackagePostfix

**Type:** character vector

**Value:** A character vector that is legal in a VHDL package file name

**Default:** '\_pkg'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'PackagePostfix','_pkg')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','PackagePostfix','_pkg')
makehdl('sfir_fixed/symmetric_fir')
```

### Reserved word postfix

Specify a text as a character vector to append to value names, postfix values, or labels that are VHDL or Verilog reserved words.

### Settings

**Default:** \_rsvd

The reserved word postfix is applied to identifiers (for entities, signals, constants, or other model elements) that conflict with VHDL or Verilog reserved words. For example, if your generating model contains a signal named `mod`, HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

### Command-Line Information

**Property:** ReservedWordPostfix

**Type:** character vector

**Default:** '\_rsvd'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, you can specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.
- ```
makehdl('sfir_fixed/symmetric_fir', ...
        'ReservedWordPostfix','_reserved')
```
- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.
- ```
hdlset_param('sfir_fixed','ReservedWordPostfix','_reserved')
makehdl('sfir_fixed/symmetric_fir')
```

## Module name prefix

Specify a prefix for every module or entity name in the generated HDL code.

### Settings

**Default:** ''

Specify a prefix for every module or entity name in the generated HDL code. HDL Coder also applies this prefix to generated script file names.

You can specify the module name prefix to avoid name collisions if you plan to instantiate the generated HDL code multiple times in a larger system.

### Command-Line Information

**Property:** `ModulePrefix`

**Type:** character vector

**Default:** ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Suppose you have a DUT, `myDut`, containing an internal module, `myUnit`. You can prefix the modules within your design with `unit1_` by using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('myDUT', ...
        'ModulePrefix','unit1_')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('myUnit/myDUT','ModulePrefix','unit1_')
makehdl('myDUT')
```

In the generated code, your HDL module names are `unit1_myDut` and `unit1_myUnit`, with corresponding HDL file names. Generated script file names also have the `unit1_` prefix.

## Pipeline postfix

Specify the postfix as a character vector to append to names of input or output pipeline registers generated for pipelined block implementations.

### Settings

**Default:** '\_pipe'

You can specify a generation of input and/or output pipeline registers for selected blocks. The **Pipeline postfix** option defines a character vector that HDL Coder appends to names of input or output pipeline registers when generating code.

### Command-Line Information

**Property:** PipelinePostfix

**Type:** character vector

**Default:** '\_pipe'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Suppose you specify a pipelined output implementation for a Product block in a model, as in the following code:

```
hdlset_param('sfir_fixed/symmetric_fir/Product', 'OutputPipeline', 2)
```

To append a postfix 'testpipe' to the generated pipeline register names, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'PipelinePostfix', 'testpipe')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'PipelinePostfix', 'testpipe')
makehdl('myDUT')
```

The following excerpt from generated VHDL code shows process the PROCESS code, with postfixed identifiers, that implements two pipeline stages:

```
Product_outtestpipe_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Product_outtestpipe_reg <= (OTHERS => to_signed(0, 33));
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Product_outtestpipe_reg(0) <= Product_out1;
      Product_outtestpipe_reg(1) <= Product_outtestpipe_reg(0);
    END IF;
  END IF;
END PROCESS Product_outtestpipe_process;
```

# Split entity and architecture Parameters

These settings correspond to the parameters in the **HDL Code Generation > Global Settings > General** tab of the Configuration Parameters dialog box. The parameters determine whether to split the entity and architecture into separate files.

## Split entity file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL entity file.

You can specify an empty character vector for either the **Split entity file postfix** or the **Split arch file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for **Split entity file postfix**, the code generator changes the entity name to a valid HDL name before generating code.

### Settings

**Default:** `_entity`

### Dependency

This parameter is enabled by selecting the **Split entity and architecture** check box. When you select this check box, HDL Coder places the VHDL entity and architecture code in separate files.

### Command-Line Information

**Property:** `SplitEntityFilePostfix`

**Type:** character vector

**Default:** `'_entity'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'SplitEntityFilePostfix', '_ent')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'SplitEntityFilePostfix', '_ent')
makehdl('myDUT')
```

## Split arch file postfix

Enter a character vector to be appended to the model name to form the name of a generated VHDL architecture file.

You can specify an empty character vector for either the **Split arch file postfix** or the **Split entity file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you

specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for **Split arch file postfix**, the code generator changes the architecture name to a valid HDL name before generating code.

### Settings

**Default:** `_arch`

### Dependency

This parameter is enabled by selecting the **Split entity and architecture** check box. When you select this check box, HDL Coder places the VHDL entity and architecture code in separate files.

### Command-Line Information

**Property:** `SplitArchFilePostfix`

**Type:** character vector

**Default:** `'_arch'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'SplitArchFilePostfix', '_arch1')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'SplitArchFilePostfix', '_arch1')
makehdl('myDUT')
```

## Split entity and architecture

Specify whether generated VHDL entity and architecture code is written to a single VHDL file or to separate files.

### Settings

**Default:** Off



On

VHDL entity and architecture definitions are written to separate files.



Off

VHDL entity and architecture code is written to a single VHDL file.

### Tips

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. You can override the default and specify your own postfix as a character vector.

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

## Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Selecting this option enables the following parameters:

- **Split entity file postfix**
- **Split architecture file postfix**

You can specify an empty character vector for either the **Split arch file postfix** or the **Split entity file postfix**. Both VHDL entity and architecture files cannot have empty postfix values. When you specify both values, make sure that you use different values for the **Split entity file postfix** and the **Split arch file postfix**.

If you input special characters for the **Split entity file postfix** or the **Split arch file postfix**, the code generator changes the entity name or the architecture name to a valid HDL name before generating code.

## Command-Line Information

**Property:** `SplitEntityArch`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.
- ```
makehdl(gcb, 'SplitEntityArch', 'on')
```
- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'SplitEntityArch', 'on')
makehdl('myDUT')
```

## Complex Signals Postfix Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > General** tab of the Configuration Parameters dialog box.

### Complex real part postfix

Specify the character vector to append to real part of complex signal names.

#### Settings

**Default:** '\_re'

Enter a text to be appended to the names generated for the real part of complex signals.

#### Command-Line Information

**Property:** ComplexRealPostfix

**Type:** character vector

**Default:** '\_re'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

To append a postfix '`_repart`' to the generated pipeline register names, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'ComplexRealPostfix', '_repart')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'ComplexRealPostfix', '_repart')
makehdl('myDUT')
```

### Complex imaginary part postfix

Specify character vector to append to imaginary part of complex signal names.

#### Settings

**Default:** '\_im'

Enter a character vector to be appended to the names generated for the imaginary part of complex signals.

#### Command-Line Information

**Property:** ComplexImagPostfix

**Type:** character vector

**Default:** '\_im'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

To append a postfix '\_imp' to the generated pipeline register names, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb,'ComplexImagePostfix','_imp')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs,'ComplexImagePostfix','_imp')
makehdl('myDUT')
```

## VHDL Architecture and Library Name and Code for Model Reference Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > General** tab of the Configuration Parameters dialog box.

### VHDL architecture name

Specify the architecture name for your DUT in the generated HDL code.

#### Settings

**Default:** 'rtl'

Specify the VHDL architecture name for your DUT in the generated HDL code as a character vector.

#### Command-Line Information

**Property:** VHDLArchitectureName

**Type:** character vector

**Default:** 'rtl'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'VHDLArchitectureName', '_rtl2')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'VHDLArchitectureName', '_rtl2')
makehdl('myDUT')
```

### VHDL library name

Specify the target library name for the generated VHDL code.

#### Settings

**Default:** 'work'

Target library name for generated VHDL code.

#### Command-Line Information

**Property:** VHDLLibNameName

**Type:** character vector

**Default:** 'work'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'VHDLLibraryName', '__work1')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'VHDLLibraryName', '__work1')
makehdl('myDUT')
```

## Generate VHDL code for model references into a single library

Specify whether VHDL code generated for model references is in a single library, or in separate libraries.

### Settings

**Default:** Off



On

Generate VHDL code for model references into a single library.



Off

For each model reference, generate a separate VHDL library.

### Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

#### Command-Line Information

**Property:** `UseSingleLibrary`

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example:

- Pass the property as an argument to the `makehdl` function.

```
makehdl(gcb, 'UseSingleLibrary', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param(gcs, 'UseSingleLibrary', 'on')
makehdl('myDUT')
```

## Generate Statement and Vector and Component Instance Label Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > General** tab of the Configuration Parameters dialog box.

### Block generate label

Specify postfix to block labels used for HDL GENERATE statements.

#### Settings

**Default:** '\_gen'

Specify the postfix as a character vector. HDL Coder appends the postfix to block labels used for HDL GENERATE statements.

#### Command-Line Information

**Property:** BlockGenerateLabel

**Type:** character vector

**Default:** '\_gen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Output generate label

Specify postfix to output assignment block labels for VHDL GENERATE statements.

#### Settings

**Default:** 'outputgen'

Specify the postfix as a character vector. HDL Coder appends this postfix to output assignment block labels in VHDL GENERATE statements.

#### Command-Line Information

**Property:** OutputGenerateLabel

**Type:** character vector

**Default:** 'outputgen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Instance generate label

Specify text to append to instance section labels in VHDL GENERATE statements.

#### Settings

**Default:** '\_gen'

Specify the postfix as a character vector. HDL Coder appends the postfix to instance section labels in VHDL GENERATE statements.

#### Command-Line Information

**Property:** InstanceGenerateLabel

**Type:** character vector

**Default:** '\_gen'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Vector prefix

Specify prefix to vector names in generated code.

#### Settings

**Default:** 'vector\_of\_'

Specify the prefix as a character vector. HDL Coder appends this prefix to vector names in generated code.

#### Command-Line Information

**Property:** VectorPrefix

**Type:** character vector

**Default:** 'vector\_of\_'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Instance postfix

Specify postfix to generated component instance names.

#### Settings

**Default:** '' (no postfix appended)

Specify the postfix as a character vector. HDL Coder appends the postfix to component instance names in generated code.

#### Command-Line Information

**Property:** InstancePostfix

**Type:** character vector

**Default:** ''

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Instance prefix

Specify prefix to generated component instance names.

**Settings****Default:** 'u\_'

Specify the prefix as a character vector. HDL Coder appends the prefix to component instance names in generated code.

**Command-Line Information****Property:** InstancePrefix**Type:** character vector**Default:** 'u\_'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

**Map file postfix**

Specify postfix appended to file name for generated mapping file.

**Settings****Default:** '\_map.txt'

Specify the postfix as a character vector. HDL Coder appends the postfix to file name for generated mapping file.

For example, if the name of the device under test is `my_design`, HDL Coder adds the postfix `_map.txt` to form the name `my_design_map.txt`.

**Command-Line Information****Property:** HDLMMapFilePostfix**Type:** character vector**Default:** '\_map.txt'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# Input and Output Port and Clock Enable Output Type Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Ports** tab of the Configuration Parameters dialog box.

## Input data type

Specify the HDL data type for the input ports of the model.

### Settings

For VHDL, the options are:

**Default:** std\_logic\_vector

std\_logic\_vector

Specifies VHDL type STD\_LOGIC\_VECTOR.

signed/unsigned

Specifies VHDL type SIGNED or UNSIGNED.

For Verilog, the options are:

**Default:** wire

In generated Verilog code, the data type for all ports is 'wire', and cannot be modified. Therefore, **Input data type** is disabled when the target language is Verilog.

### Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

#### Command-Line Information

**Property:** InputType

**Type:** character vector

**Value:** (for VHDL) 'std\_logic\_vector' | 'signed/unsigned'

(for Verilog) 'wire'

**Default:** (for VHDL) 'std\_logic\_vector'

(for Verilog) 'wire'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Output data type

Specify the HDL data type for the output ports of the model.

### Settings

For VHDL, the options are:

**Default:** Same as input data type

Same as input data type

Specifies that output ports of the model have the same type specified by **Input data type**.

`std_logic_vector`

Specifies VHDL type `STD_LOGIC_VECTOR` as the data type of the output port.

`signed/unsigned`

Specifies VHDL type `SIGNED` or `UNSIGNED` as the data type of the output port.

For Verilog, the options are:

**Default:** `wire`

In generated Verilog code, the data type for all ports is '`wire`', and cannot be modified. Therefore, **Output data type** is disabled when the target language is Verilog.

**Dependency**

This option is enabled when the target language (specified by the **Language** option) is VHDL.

**Command-Line Information**

**Property:** `OutputType`

**Type:** character vector

**Value:** (for VHDL) '`std_logic_vector`' | '`signed/unsigned`'

(for Verilog) '`wire`'

**Default:** If the property is left unspecified, output ports have the same type specified by `InputType`.

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Clock Enable output port

Specify the name for the generated clock enable output port as a character vector.

**Settings**

**Default:** `ce_out`

A clock enable output is generated when the design requires one.

**Command-Line Information**

**Property:** `ClockEnableOutputPort`

**Type:** character vector

**Default:** '`ce_out`'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

**See Also**

"Clock Enable Settings and Parameters" on page 17-12

# Minimize Clock Enables and Reset Signal Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Ports** tab of the Configuration Parameters dialog box.

## Minimize clock enables

Omit generation of clock enable logic for single-rate designs.

### Settings

**Default:** Off



On

For single-rate models, omit generation of clock enable logic wherever possible. The following VHDL code example does not define or examine a clock enable signal. When the clock signal (`clk`) goes high, the current signal value is output.

```
Unit_Delay_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
        Unit_Delay_out1 <= In1_signed;
    END IF;
END PROCESS Unit_Delay_process;
```



Off

Generate clock enable logic. The following VHDL code extract represents a register with a clock enable (`enb`)

```
Unit_Delay_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
        IF enb = '1' THEN
            Unit_Delay_out1 <= In1_signed;
        END IF;
    END IF;
END PROCESS Unit_Delay_process;
```

### Exceptions

In some cases, HDL Coder emits clock enables even when **Minimize clock enables** is selected. These cases are:

- Registers inside Enabled, State-Enabled, and Triggered subsystems.
- Multirate models.
- The coder always emits clock enables for the following blocks:
  - commseqgen2/PN Sequence Generator

- `dspsigops/NCO`

---

**Note** HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.

---

- `dpsrcs4/Sine Wave`
- `hdldemolib/HDL FFT`
- `built-in/DiscreteFir`
- `dspmlti4/CIC Decimation`
- `dspmlti4/CIC Interpolation`
- `dspmlti4/FIR Decimation`
- `dspmlti4/FIR Interpolation`
- `dspadpt3/LMS Filter`
- `dsparc4/Biquad Filter`

---

**Note** If your design uses a RAM block such as a Dual Rate Dual Port RAM with the **RAM Architecture** set to **Generic RAM without Clock Enable**, the code generator ignores the **Minimize clock enables** setting.

---

### Command-Line Information

**Property:** `MinimizeClockEnables`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to minimize Clock Enable signals when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'MinimizeClockEnables','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MinimizeClockEnables','on')
makehdl('sfir_fixed/symmetric_fir')
```

### Minimize global resets

Omit generation of reset logic in the HDL code.

#### Settings

**Default:** Off

On

When you enable this setting, the code generator tries to minimize or remove the global reset logic from the HDL code. This code snippet corresponds to the Verilog code generated for a Delay block in the Simulink model. The code snippet shows that HDL Coder removed the reset logic.

```
always @(posedge clk)
begin : Delay_Synchronous_process
  if (enb) begin
    Delay_Synchronous_out1 <= DataIn;
  end
end
```

Off

When you disable this parameter, HDL Coder generates the global reset logic in the HDL code. This Verilog code snippet shows the reset logic generated for the Delay block.

```
always @ (posedge clk or posedge reset)
begin : Delay_Synchronous_process
  if (reset == 1'b1) begin
    Delay_Synchronous_out1 <= 1'b0;
  end
  else begin
    if (enb) begin
      Delay_Synchronous_out1 <= DataIn;
    end
  end
end
```

## Dependency

If you select **Minimize global resets**, the generated HDL code contains registers that do not have a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results. To avoid simulation mismatches, you can initialize the registers by using the “No-reset registers initialization” on page 17-53 setting.

By default, the **No-reset registers initialization** setting has the value `Generate initialization inside module`, which means that the code generator initializes the registers as part of the HDL code generated for the DUT. To initialize the registers with the script, set **No-reset registers initialization** to `Generate an external script`. You must use a zero initial value for the blocks in your Simulink model.

## Exceptions

Sometimes, when you select **Minimize global resets**, HDL Coder generates the reset logic, if you have:

- Blocks with state that have a nonzero initial value, such as a Delay block with non-zero **Initial Condition**.
- Enumerated data types for blocks with state.
- Subsystem blocks with BlackBox HDL architecture where you request a reset signal.
- Multirate models with **Timing controller architecture** set to `default`.

If you set **Timing controller architecture** to resettable, HDL Coder generates a reset port for the timing controller. If you set **Minimize global reset signals** to 'on', the code generator removes this reset port.

- Truth Table
- Chart
- MATLAB Function block

### Command-Line Information

**Property:** MinimizeGlobalResets

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to minimize global reset signals when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model, use either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'MinimizeGlobalResets','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','MinimizeGlobalResets','on')
makehdl('sfir_fixed/symmetric_fir')
```

# Using Trigger Signals and Scalarization and Test Point DUT Port Generation Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Ports** tab of the Configuration Parameters dialog box.

## Use trigger signal as clock

This setting is a parameter in the **HDL Code Generation > Global Settings > Ports** tab of the Configuration Parameters dialog box.

### Settings

**Default:** Off



On

For triggered subsystems, use the trigger input signal as a clock in the generated HDL code. Make sure that the **Clock edge** setting in the Configuration Parameters dialog box matches the **Trigger type** of the Trigger block inside the triggered subsystem.



Off

For triggered subsystems, do not use the trigger input signal as a clock in the generated HDL code.

### Command-Line Information

**Property:** `TriggerAsClock`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to generate HDL code that uses the trigger signal as clock for triggered subsystems within the `sfir_fixed/symmetric_sfir` DUT subsystem, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl ('sfir_fixed/symmetric_sfir','TriggerAsClock','on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','TriggerAsClock','on')
makehdl('sfir_fixed/symmetric_sfir')
```

## Enable HDL DUT port generation for test points

Enable this setting to create DUT output ports for the test point signals in the generated HDL code.

### Settings

**Default:** Off

On

When you enable this setting, the code generator creates DUT output ports for the test point signals in the generated HDL code. You can observe the test point signals and debug your design by connecting a Scope block to the output ports corresponding to these signals.

 Off

When you disable this setting, the code generator preserves the test point signals and does not create DUT output ports in the generated HDL code.

---

**Note** The code generator ignores this setting when you designate test points for states inside a Stateflow Chart.

---

### Command-Line Information

**Property:** `EnableTestpoints`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, after you designate signals as testpoints for the `sfir_fixed/symmetric_fir` DUT subsystem, to generate DUT output ports in the HDL code, use either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl ('sfir_fixed/symmetric_sfir', 'EnableTestpoints', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','EnableTestpoints','on')
makehdl('sfir_fixed/symmetric_fir')
```

### See Also

“Model and Debug Test Point Signals with HDL Coder” on page 10-59

## Scalarize ports

Flatten vector ports into a structure of scalar ports in VHDL code.

### Settings

**Default:** Off

On

When generating code for a vector port, generate a structure of scalar ports.

Off

When generating code for a vector port, generate a type definition and port declaration for the vector port.

**dutlevel**

When generating code for a vector port, generate a structure of scalar ports for vector ports that are only at DUT level. The DUT subsystem does not have to be at the top level of your model.

**Dependency**

This option is enabled when the target language (specified by the **Language** option) is VHDL.

**Command-Line Information**

**Property:** ScalarizePorts

**Type:** character vector

**Value:** 'on' | 'off' | 'dutlevel'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

**See Also**

"Scalarization of Vector Ports in Generated VHDL Code" on page 27-24

## RTL Annotation Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

### Use Verilog `timescale directives

Specify use of compiler `timescale directives in generated Verilog code.

#### Settings

**Default:** On



On

Use compiler `timescale directives in generated Verilog code.



Off

Suppress the use of compiler `timescale directives in generated Verilog code.

#### Tip

The `timescale directive provides a way of specifying different delay values for multiple modules in a Verilog file. This setting does not affect the generated test bench.

#### Dependency

This option is enabled when the target language (specified by the **Language** option) is Verilog.

#### Command-Line Information

**Property:** UseVerilogTimescale

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Verilog timescale specification

Specify the timescale that you want to use in the generated Verilog code.

#### Settings

**Default:** `timescale 1 ns/1 ns

HDL Coder applies this option to the timescale directive in the generated Verilog code. You can customize the default timescale and specify a valid, compilable timescale directive. The Verilog language uses this directive to determine the time units and the precision for calculating delay values.

By default, both the time units and precision are 1ns. For example, if you customized the timescale to `timescale 1 ns/1 ps, a delay unit becomes 1ns and the value is precise to the nearest 1 ps.

## Dependency

This option is enabled when:

- The target language (specified by the **Language** option) is Verilog.
- The **Use Verilog `timescale directives** option is enabled.

## Command-Line Information

**Property:** Timescale

**Type:** character vector

**Value:** A character vector that is a valid timescale value

**Default:** `timescale 1 ns/1 ns

## Inline VHDL configuration

Specify whether generated VHDL code includes inline configurations.

### Settings

**Default:** On



On

Include VHDL configurations in files that instantiate a component.



Off

Suppress the generation of configurations and require user-supplied external configurations. Use this setting if you are creating your own VHDL configuration files.

### Tip

HDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, HDL Coder includes configurations for a model within the generated VHDL code. If you are creating your own VHDL configuration files, suppress the generation of inline configurations.

## Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

## Command-Line Information

**Property:** InlineConfigurations

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Concatenate type safe zeros

Specify use of syntax for concatenated zeros in generated VHDL code.

## Settings

**Default:** On

On

Use the type-safe syntax, '0' & '0', for concatenated zeros. Typically, this syntax is preferred.

Off

Use the syntax "000000..." for concatenated zeros. This syntax can be easier to read and more compact, but it can lead to ambiguous types.

## Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

### Command-Line Information

**Property:** SafeZeroConcat

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Generate obfuscated HDL code

Specify generation of obfuscated HDL code. By using obfuscation, you can share the HDL code with a third-party without revealing the intellectual property. Obfuscation reduces readability of the code.

The generated HDL code does not have comments, newlines or spaces, and replaces identifier names with other random names.

## Settings

**Default:** Off

On

Generate obfuscated HDL code.

Off

Do not generate obfuscated HDL code.

## Dependency

To enable this parameter, the **Generate HDL Code** check box must be selected.

### Command-Line Information

**Property:** ObfuscateGeneratedHDLCode

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

- To generate obfuscated HDL code by using `makehdl`:
 

```
makehdl('dutname', 'ObfuscateGeneratedHDLCode', 'on')
```
- To generate obfuscated HDL code by using `hdlset_param`:
 

```
hdlset_param('modelname', 'ObfuscateGeneratedHDLCode', 'on')
makehdl('dutname')
```

## Emit time/date stamp in header

Specify whether or not to include time and date information in the generated HDL file header.

### Settings

**Default:** On



On

Include time/date stamp in the generated HDL file header.

```
-- -----
-- 
-- File Name: hdlsrc\symmetric_fir.vhd
-- Created: 2011-02-14 07:21:36
--
```



Off

Omit time/date stamp in the generated HDL file header.

```
-- -----
-- 
-- File Name: hdlsrc\symmetric_fir.vhd
--
```

By omitting the time/date stamp in the file header, you can more easily determine if two HDL files contain identical code. You can also avoid redundant revisions of the same file when checking in HDL files to a source code management (SCM) system.

### Command-Line Information

**Property:** DateComment

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Include requirements in block comments

Enable or disable generation of requirements comments as comments in code or code generation reports.

### Settings

**Default:** On

On

If the model contains requirements comments, include them as comments in code or code generation reports. See “Requirements Comments and Hyperlinks” on page 25-14.

Off

Do not include requirements as comments in code or code generation reports.

#### **Command-Line Information**

**Property:** RequirementComments

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# RTL Customization Parameters for Constants and MATLAB Function Blocks

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

## Inline MATLAB Function block code

Inline HDL code for MATLAB Function blocks.

### Settings

**Default:** Off



On

Inline HDL code for MATLAB Function blocks to avoid instantiation of code for custom blocks.



Off

Instantiate HDL code for MATLAB Function blocks and do not inline.

### Command-Line Information

**Property:** `InlineMATLABBlockCode`

**Type:** character vector

**Value:** `'on'` | `'off'`

**Default:** `'off'`

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

For example, to enable inlining of the code:

```
mdl = 'my_custom_block_model';
hdlset_param(mdl, 'InlineMATLABBlockCode', 'on');
```

For example, to enable instantiation of HDL code:

```
mdl = 'my_custom_block_model';
hdlset_param(mdl, 'InlineMATLABBlockCode', 'off');
```

## Represent constant values by aggregates

Specify whether constants in VHDL code are represented by aggregates, including constants that are less than 32 bits. This option does not affect generated HDL code for MATLAB Function blocks.

### Settings

**Default:** Off



On

HDL Coder represents constants as aggregates. The following VHDL constant declarations show a scalar less than 32 bits represented as an aggregate:

```
GainFactor_gainparam <= (14 => '1', OTHERS => '0');
```

**Off**

The coder represents constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. The following VHDL code was generated by default for a value less than 32 bits:

```
GainFactor_gainparam <= to_signed(16384, 16);
```

### Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

#### Command-Line Information

**Property:** UseAggregatesForConst

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# RTL Customization Parameters for RAMs

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

## Initialize all RAM blocks

Enable or suppress generation of initial signal value for RAM blocks. If you specify a nonzero initial value for the RAM, this setting is ignored.

### Settings

**Default:** On



On

For RAM blocks, generate initial values of '0' for both the RAM signal and the output temporary signal.



Off

For RAM blocks, do not generate initial values for either the RAM signal or the output temporary signal.

### Tip

This parameter applies to these RAM blocks in the **HDL Coder > HDL RAMs** Block Library in the Simulink Library Browser:

- Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM
- Dual Rate Dual Port RAM

### Command-Line Information

**Property:** InitializeBlockRAM

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## RAM Architecture

Select RAM architecture with clock enable, or without clock enable, for all RAMs in DUT subsystem.

### Settings

**Default:** RAM with clock enable

Select one of the following options from the menu:

- RAM with `clock enable`: Generate RAMs with clock enable.
- Generic RAM without `clock enable`: Generate RAMs without clock enable.

**Command-Line Information**

**Property:** RAMArchitecture

**Type:** character vector

**Value:** 'WithClockEnable' | 'WithoutClockEnable'

**Default:** 'WithClockEnable'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# No-reset registers initialization

Specify whether you want to initialize registers without reset and the mode of initialization.

## Settings

**Default:** Generate initialization inside module

The options are:

**Do not initialize**

HDL Coder does not initialize the registers without a reset port.

**Generate an external script**

HDL Coder generates a script to initialize registers that do not have a reset port in the generated code.

**Generate initialization inside module**

HDL Coder initializes the registers that do not have a reset port as part of the HDL code generated for the DUT. In Verilog, an `initial` construct in the corresponding module definition initializes the no-reset registers. In VHDL, the initialization code is part of the signal declaration statements.

## Usage Notes

If you have blocks with **ResetType** on page 22-21 set to none in your Simulink model or specify the adaptive pipelining optimization, the generated HDL code can contain registers without a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results, because the registers are insensitive to the global reset logic. To avoid simulation mismatches, use this setting to initialize these registers in the generated code. For better simulation results, if you have registers without a reset port at the boundaries of the DUT, select **Initialize test bench inputs** in the **Test Bench** pane. Setting this property provides an initial value for the data driven to the DUT, and initializes the registers with these values.

| Functionality              | Script                                                                                                                                                                                                                                                                         | None value                                                                                                                                          | InsideModule                                                                                                        |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| Generated HDL code for DUT | The script is generated externally and does not affect the HDL code for the DUT.                                                                                                                                                                                               | HDL Coder does not initialize the registers in the generated code.                                                                                  | The code for initializing the registers is part of the HDL code for the DUT.                                        |
| HDL simulator support      | The syntax of the script is compliant with ModelSim 10.2c or later. Other HDL simulators or older ModelSim versions do not support the syntax of the initialization script. This mode does not support enumeration types, and initializing the registers with non zero values. | There can be numerical mismatches in the HDL simulation results, because this mode does not initialize the registers that do not have a reset port. | All HDL simulators support this initialization mode, and initialize the no-reset registers with appropriate values. |

| Functionality          | Script                                                                                                                  | None value                                                             | InsideModule                                                                                                                                                                                                                                                                        |
|------------------------|-------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Synthesis tool support | As the script does not affect the HDL code generated for the DUT, all synthesis tools support this initialization mode. | Synthesis tools do not initialize the no-reset registers in this mode. | Later versions of synthesis tools support the initialization constructs in the generated code. However, it is possible that older versions do not synthesize the initialization constructs. To avoid such issues, make sure that synthesis tools can synthesize the generated code. |

## Command-Line Information

**Property:** NoResetInitializationMode

**Type:** character vector

**Value:** 'InsideModule' | 'None' | 'Script'

**Default:** 'InsideModule'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

"Minimize global resets" on page 17-38

# RTL Style Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

## Use “`rising_edge/falling_edge`” style for registers

Specify whether generated code uses the VHDL `rising_edge` or `falling_edge` function to detect clock transitions.

### Settings

**Default:** Off



On

Generated code uses the VHDL `rising_edge` or `falling_edge` function.

For example, the following code, generated from a Unit Delay block, uses `rising_edge` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        IF clk_enable = '1' THEN
            Unit_Delay1_out1 <= signed(x_in);
        END IF;
    END IF;
END PROCESS Unit_Delay1_process;
```



Off

Generated code uses the 'event syntax.

For example, the following code, generated from a Unit Delay block, uses `clk'event AND clk = '1'` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
        IF clk_enable = '1' THEN
            Unit_Delay1_out1 <= signed(x_in);
        END IF;
    END IF;
END PROCESS Unit_Delay1_process;
```

### Dependency

This option is enabled when the target language is VHDL.

### Command-Line Information

**Property:** UseRisingEdge

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Minimize intermediate signals

Specify whether to optimize HDL code for debuggability or code coverage.

### Settings

**Default:** Off

On

Optimize for code coverage by minimizing intermediate signals. For example, suppose that the generated code with this setting *off* is:

```
const3 <= to_signed(24, 7);
subtractor_sub_cast <= resize(const3, 8);
subtractor_sub_cast_1 <= resize(delayout, 8);
subtractor_sub_temp <= subtractor_sub_cast - subtractor_sub_cast_1;
```

With this setting *on*, HDL Coder optimizes the output to:

```
subtractor_sub_temp <= 24 - (resize(delayout, 8));
```

The code generator removes the intermediate signals `const3`, `subtractor_sub_cast`, and `subtractor_sub_cast_1`.

Off

Optimize for debuggability by preserving intermediate signals.

### Command-Line Information

**Property:** `MinimizeIntermediateSignals`

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'off'

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Unroll for Generate Loops in VHDL code

Specify whether VHDL FOR and GENERATE loops are unrolled and omitted from generated VHDL code.

### Settings

**Default:** Off

On

Unroll and omit FOR and GENERATE loops from the generated VHDL code. (In Verilog code, loops are always unrolled.)