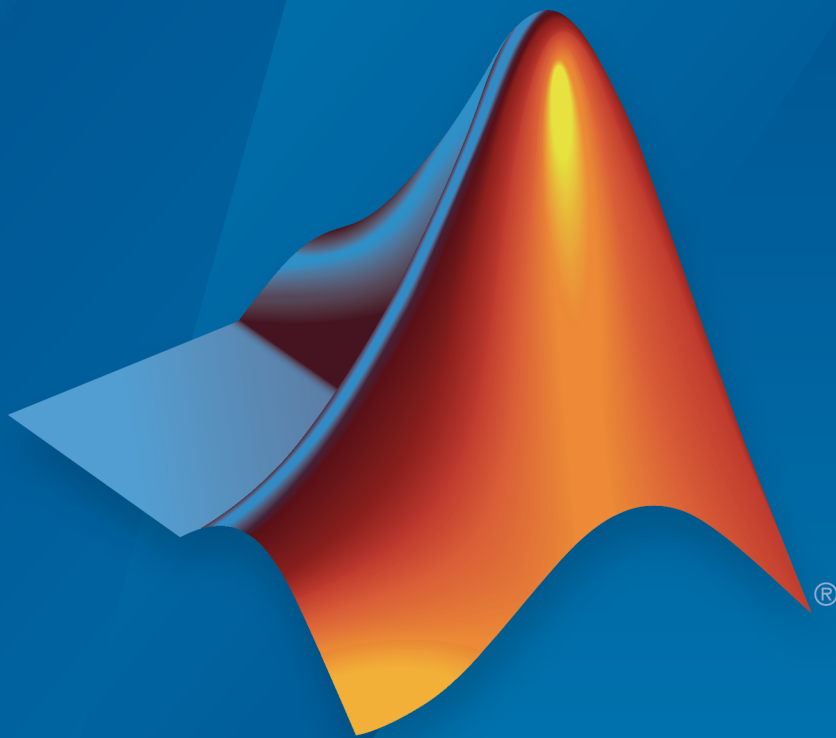


Filter Design HDL Coder™

User's Guide



MATLAB®

R2015a

 MathWorks®

How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Filter Design HDL Coder™ User's Guide

© COPYRIGHT 2004–2015 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 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 for more information.

Revision History

June 2004	Online only	New for Version 1.0 (Release 14)
October 2004	Online only	Revised for Version 1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.2 (Release 14SP2)
September 2005	Online only	Revised for Version 1.3 (Release 14SP3)
March 2006	Online only	Revised for Version 1.4 (Release 2006a)
September 2006	Online only	Revised for Version 1.5 (Release 2006b)
March 2007	Online only	Revised for Version 2.0 (Release 2007a)
September 2007	Online only	Revised for Version 2.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.2 (Release 2008a)
October 2008	Online only	Revised for Version 2.3 (Release 2008b)
March 2009	Online only	Revised for Version 2.4 (Release 2009a)
September 2009	Online only	Revised for Version 2.5 (Release 2009b)
March 2010	Online only	Revised for Version 2.6 (Release 2010a)
September 2010	Online only	Revised for Version 2.7 (Release 2010b)
April 2011	Online only	Revised for Version 2.8 (Release 2011a)
September 2011	Online only	Revised for Version 2.9 (Release 2011b)
March 2012	Online only	Revised for Version 2.9.1 (Release 2012a)
September 2012	Online only	Revised for Version 2.9.2 (Release 2012b)
March 2013	Online only	Revised for Version 2.9.3 (Release 2013a)
September 2013	Online only	Revised for Version 2.9.4 (Release 2013b)
March 2014	Online only	Revised for Version 2.9.5 (Release 2014a)
October 2014	Online only	Revised for Version 2.9.6 (Release 2014b)
March 2015	Online only	Revised for Version 2.9.7 (Release 2015a)

1 Getting Started

Filter Design HDL Coder Product Description	1-2
Key Features	1-2
Automated HDL Code Generation	1-3
Product Requirements	1-4
Required Products	1-4
VHDL and Verilog Language Support	1-4
User Experience	1-4
Tutorials on Generating HDL Code for Filters	1-6
Creating a Folder for Your Tutorial Files	1-6
Basic FIR Filter	1-6
Optimized FIR Filter	1-24
IIR Filter	1-45

2 HDL Filter Code Generation Fundamentals

Starting Filter Design HDL Coder	2-2
Opening the Filter Design HDL Coder GUI From FDATool ..	2-2
Opening the Filter Design HDL Coder GUI From the filterbuilder GUI	2-6
Opening the Filter Design HDL Coder GUI Using the fdhdltool Command	2-10
Selecting Target Language	2-12

Generating HDL Code	2-13
Applying Your Settings	2-13
Generating HDL Code from the GUI	2-13
Generating HDL Code Using generatehdl	2-14
Capturing Code Generation Settings	2-15
Closing Code Generation Session	2-16

HDL Code for Supported Filter Structures

3

Multirate Filters	3-2
Supported Multirate Filter Types	3-2
Generating Multirate Filter Code	3-2
Code Generation Options for Multirate Filters	3-2
Variable Rate CIC Filters	3-8
Supported Variable Rate CIC Filter Types	3-8
Code Generation Options for Variable Rate CIC Filters	3-8
Cascade Filters	3-11
Supported Cascade Filter Types	3-11
Generating Cascade Filter Code	3-11
Polyphase Sample Rate Converters	3-14
About Code Generation for Direct-Form FIR Polyphase Sample Rate Converters	3-14
HDL Implementation for Polyphase Sample Rate Converter	3-14
Multirate Farrow Sample Rate Converters	3-17
About Code Generation for Multirate Farrow Sample Rate Converters	3-17
Generating Code for mfilter.farrowsrc Filters at the Command Line	3-17
Generating Code for mfilter.farrowsrc Filters in the GUI	3-18
Single-Rate Farrow Filters	3-20
About Code Generation for Single-Rate Farrow Filters	3-20
Code Generation Properties for Farrow Filters	3-20

GUI Options for Farrow Filters	3-22
Farrow Filter Code Generation Mechanics	3-24
Programmable Filter Coefficients for FIR Filters	3-27
About Programmable Filter Coefficients for FIR Filters	3-27
Supported FIR Filter Types	3-28
Supported Parallel and Serial Filter Architectures	3-28
Generating a Port Interface for Programmable FIR Coefficients	3-28
Generating a Test Bench for Programmable FIR Coefficients	3-29
GUI Options for Programmable Coefficients	3-31
Using Programmable Coefficients with Serial FIR Filter Architectures	3-32
Programmable Filter Coefficients for IIR Filters	3-39
About Programmable Filter Coefficients for IIR Filters	3-39
Supported IIR Filter Types	3-39
Generating a Port Interface for Programmable IIR Filter Coefficients	3-40
Generating a Test Bench for Programmable IIR Coefficients	3-41
Addressing Scheme for Loading IIR Coefficients	3-42
GUI Options for Programmable Coefficients	3-44
DUC and DDC System Objects	3-47
Limitations	3-47

Optimization of HDL Filter Code

4

Speed vs. Area Tradeoffs	4-2
Overview of Speed vs. Area Optimizations	4-2
Parallel and Serial Architectures	4-3
Specifying Speed vs. Area Tradeoffs via generatehdl Properties	4-6
Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box	4-11
Distributed Arithmetic for FIR Filters	4-24
Distributed Arithmetic Overview	4-24

Requirements and Considerations for Generating Distributed Arithmetic Code	4-26
DALUTPartition Property	4-27
DARadix Property	4-29
Specifying Distributed Arithmetic for Cascaded Filters	4-30
Special Cases	4-31
Distributed Arithmetic Options in the Generate HDL Dialog Box	4-31
Architecture Options for Cascaded Filters	4-37
CSD Optimizations for Coefficient Multipliers	4-39
Improving Filter Performance with Pipelining	4-40
Optimizing the Clock Rate with Pipeline Registers	4-40
Multiplier Input and Output Pipelining for FIR Filters	4-41
Optimizing Final Summation for FIR Filters	4-42
Specifying or Suppressing Registered Input and Output	4-44
Overall HDL Filter Code Optimization	4-46
Optimize for HDL	4-46
Set Error Margin for Test Bench	4-47

Customization of HDL Filter Code

5

HDL File Names and Locations	5-2
Setting the Location of Generated Files	5-2
Naming the Generated Files and Filter Entity	5-3
Set HDL File Name Extensions	5-4
Splitting Entity and Architecture Code Into Separate Files ..	5-8
HDL Identifiers and Comments	5-10
Specifying a Header Comment	5-10
Resolving Entity or Module Name Conflicts	5-12
Resolving HDL Reserved Word Conflicts	5-13
Setting the Postfix String for VHDL Package Files	5-16
Specifying a Prefix for Filter Coefficients	5-17
Specifying a Postfix String for Process Block Labels	5-18
Setting a Prefix for Component Instance Names	5-19

Setting a Prefix for Vector Names	5-20
Ports and Resets	5-22
Naming HDL Ports	5-22
Specifying the HDL Data Type for Data Ports	5-23
Selecting Asynchronous or Synchronous Reset Logic	5-24
Setting the Asserted Level for the Reset Input Signal	5-25
Suppressing Generation of Reset Logic	5-27
HDL Language Constructs	5-29
Representing VHDL Constants with Aggregates	5-29
Unrolling and Removing VHDL Loops	5-30
Using the VHDL rising_edge Function	5-31
Suppressing the Generation of VHDL Inline Configurations	5-32
Specifying VHDL Syntax for Concatenated Zeros	5-33
Specifying Input Type Treatment for Addition and Subtraction Operations	5-34
Suppressing Verilog Time Scale Directives	5-35
Using Complex Data and Coefficients	5-36

Verification of Generated HDL Filter Code

6

Testing with an HDL Test Bench	6-2
Workflow for Testing With an HDL Test Bench	6-2
Enabling Test Bench Generation	6-9
Renaming the Test Bench	6-11
Specifying a Test Bench Type	6-12
Splitting Test Bench Code and Data into Separate Files ...	6-14
Configuring the Clock	6-15
Configuring Resets	6-17
Setting a Hold Time for Data Input Signals	6-20
Setting an Error Margin for Optimized Filter Code	6-22
Setting an Initial Value for Test Bench Inputs	6-24
Setting Test Bench Stimuli	6-25
Setting a Postfix for Reference Signal Names	6-26
Cosimulation of HDL Code with HDL Simulators	6-27
Generating HDL Cosimulation Blocks for Use with HDL Simulators	6-27

Generating a Simulink Model for Cosimulation with an HDL Simulator	6-29
Integration With Third-Party EDA Tools	6-36
Generating a Default Script	6-36
Customizing Script Generation Using CLI Properties	6-37
Customizing Script Generation with the EDA Tool Scripts Dialog Box	6-40

Synthesis and Workflow Automation

7

Automation Scripts for Third-Party Synthesis Tools	7-2
Selecting a Synthesis Tool	7-2
Customizing Synthesis Script Generation Using CLI Properties	7-3
Customizing Synthesis Script Generation with the EDA Tool Scripts Dialog Box	7-4

Properties — Alphabetical List

8

Function Reference

9

Getting Started

- “Filter Design HDL Coder Product Description” on page 1-2
- “Automated HDL Code Generation” on page 1-3
- “Product Requirements” on page 1-4
- “Tutorials on Generating HDL Code for Filters” on page 1-6

Filter Design HDL Coder Product Description

Generate HDL code for fixed-point filters

The Filter Design HDL Coder™ product adds hardware implementation capability to MATLAB®. It lets you generate efficient, synthesizable, and portable VHDL® and Verilog® code for fixed-point filters that are designed with DSP System Toolbox™ software, for implementation in ASICs or FPGAs. It also automatically creates VHDL and Verilog test benches for quickly simulating, testing, and verifying the generated code.

Key Features

- Generates synthesizable IEEE® 1076 compliant VHDL code and IEEE 1364-2001 compliant Verilog code for implementing fixed-point filters in ASICs and FPGAs
- Controls the content, optimization, and style of generated code
- Provides options for speed vs. area tradeoffs and architecture exploration, including distributed arithmetic
- Generates VHDL and Verilog test benches for quick verification and validation of generated HDL filter code
- Generates simulation and synthesis scripts

Automated HDL Code Generation

Hardware description language (HDL) code generation accelerates the development of application-specific integrated circuit (ASIC) and field programmable gate array (FPGA) designs and bridges the gap between system-level design and hardware development.

Traditionally, system designers and hardware developers use HDLs, such as very high speed integrated circuit (VHSIC) hardware description language (VHDL) and Verilog, to develop hardware designs. Although HDLs provide a proven method for hardware design, the task of coding filter designs, and hardware designs in general, is labor intensive and the use of these languages for algorithm and system-level design is not optimal. Users of the Filter Design HDL Coder product can spend more time on fine-tuning algorithms and models through rapid prototyping and experimentation and less time on HDL coding. Architects and designers can efficiently design, analyze, simulate, and transfer system designs to hardware developers.

In a typical use scenario, an architect or designer uses DSP System Toolbox GUIs (FDATool or `filterbuilder`) to design a filter. Then, a designer uses the Filter Design HDL Coder GUI or command-line interface to configure code generation options and generate a VHDL or Verilog implementation of the design and a corresponding test bench. The generated code adheres to a clean HDL coding style that enables architects and designers to quickly address customizations. The test bench feature allows independent confirmation that the generated code is as expected and may accelerate your test bench implementation.

Product Requirements

In this section...
“Required Products” on page 1-4
“VHDL and Verilog Language Support” on page 1-4
“User Experience” on page 1-4

Required Products

- MATLAB
- Fixed-Point Designer™
- DSP System Toolbox

For information on installing the required software listed above, and optional software, see the individual product’s Installation Guide.

VHDL and Verilog Language Support

The coder generates code that is compatible with HDL compilers, simulators and other tools that support

- VHDL versions 87, 93, and 02.

Exception: VHDL test benches using double precision data types do not support VHDL version 87.

- Verilog-2001 (IEEE 1364-2001) or later.

User Experience

The Filter Design HDL Coder software is a tool for system and hardware architects and designers who develop, optimize, and verify hardware signal filters. These designers are experienced with VHDL or Verilog, but can benefit greatly from a tool that automates HDL code generation. The Filter Design HDL Coder interface provides designers with efficient means for creating test signals and test benches that verify algorithms, validating models against standard reference designs, and translate legacy HDL descriptions into system-level views.

Users are expected to have prerequisite knowledge in the following subject areas:

- Hardware design and system integration
- VHDL or Verilog
- HDL simulators

Users are also expected to have experience with the following products:

- MATLAB
- DSP System Toolbox

Tutorials on Generating HDL Code for Filters

In this section...

“Creating a Folder for Your Tutorial Files” on page 1-6

“Basic FIR Filter” on page 1-6

“Optimized FIR Filter” on page 1-24

“IIR Filter” on page 1-45

Creating a Folder for Your Tutorial Files

Set up a writable working folder outside your MATLAB installation folder to store files that will be generated as you complete your tutorial work. The tutorial instructions assume that you create the folder `hdlfilter_tutorials` on drive C.

Basic FIR Filter

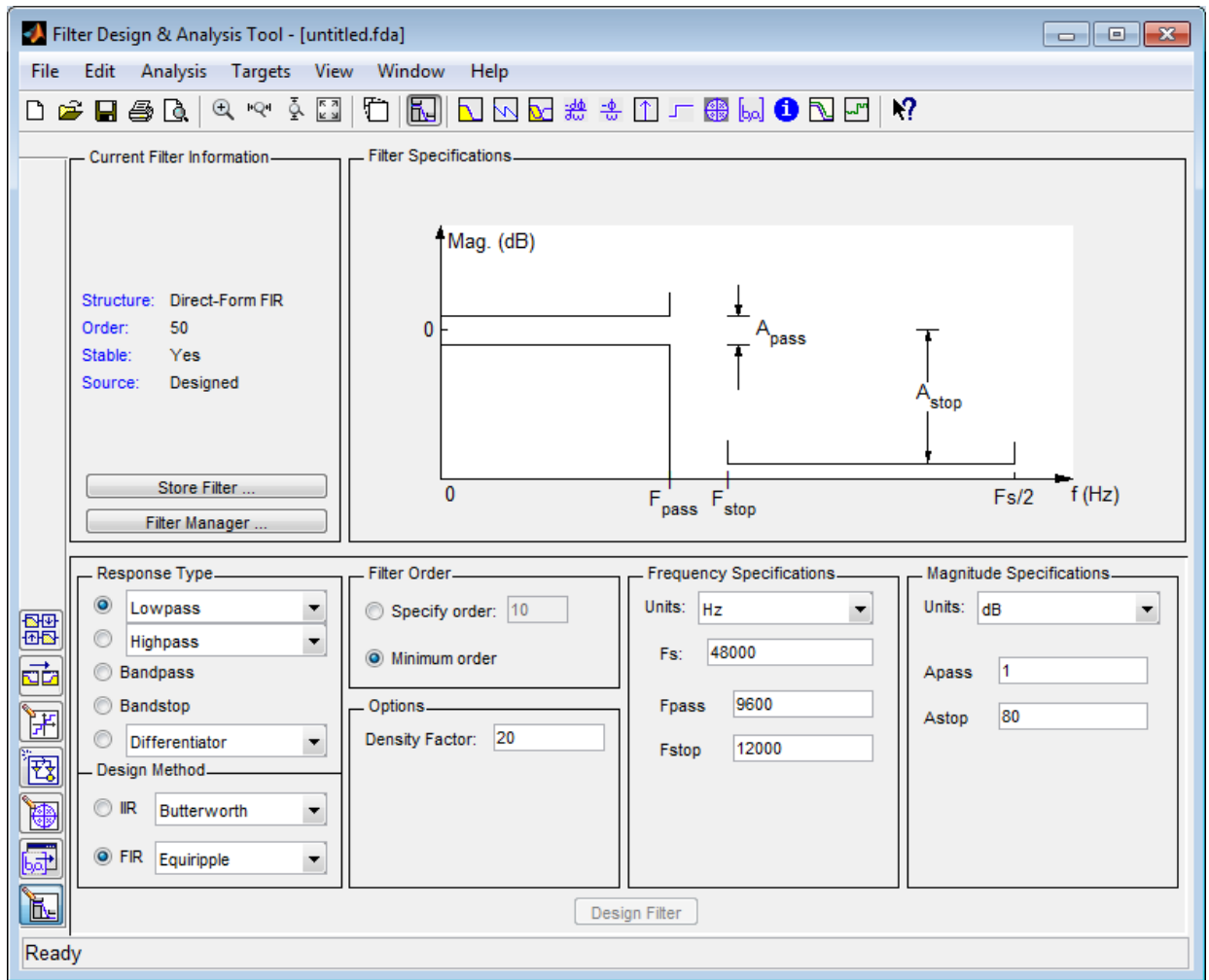
- “Designing a Basic FIR Filter in FDATool” on page 1-6
- “Quantizing the Basic FIR Filter” on page 1-8
- “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11
- “Getting Familiar with the Basic FIR Filter's Generated VHDL Code” on page 1-18
- “Verifying the Basic FIR Filter's Generated VHDL Code” on page 1-19

Designing a Basic FIR Filter in FDATool

This tutorial guides you through the steps for designing a basic quantized discrete-time FIR filter, generating VHDL code for the filter, and verifying the VHDL code with a generated test bench.

This section assumes you are familiar with the MATLAB user interface and the Filter Design & Analysis Tool (FDATool). The following instructions guide you through the procedure of designing and creating a basic FIR filter using FDATool:

- 1** Start the MATLAB software.
- 2** Set your current folder to the folder you created in “Creating a Folder for Your Tutorial Files” on page 1-6.
- 3** Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



- 4 In the Filter Design & Analysis Tool dialog box, check that the following filter options are set:

Option	Value
Response Type	Lowpass
Design Method	FIR Equiripple

Option	Value
Filter Order	Minimum order
Options	Density Factor: 20
Frequency Specifications	Units: Hz
	Fs: 48000
	Fpass: 9600
	Fstop: 12000
Magnitude Specifications	Units: dB
	Apass: 1
	Astop: 80

These settings are for the default filter design that the FDATool creates for you. If you do not have to make changes and **Design Filter** is grayed out, you are done and can skip to “Quantizing the FIR Filter” on page 1-27.


- 5 If you modified options listed in step 4, click **Design Filter**. The FDATool creates a filter for the specified design and displays the following message in the FDATool status bar when the task is complete.

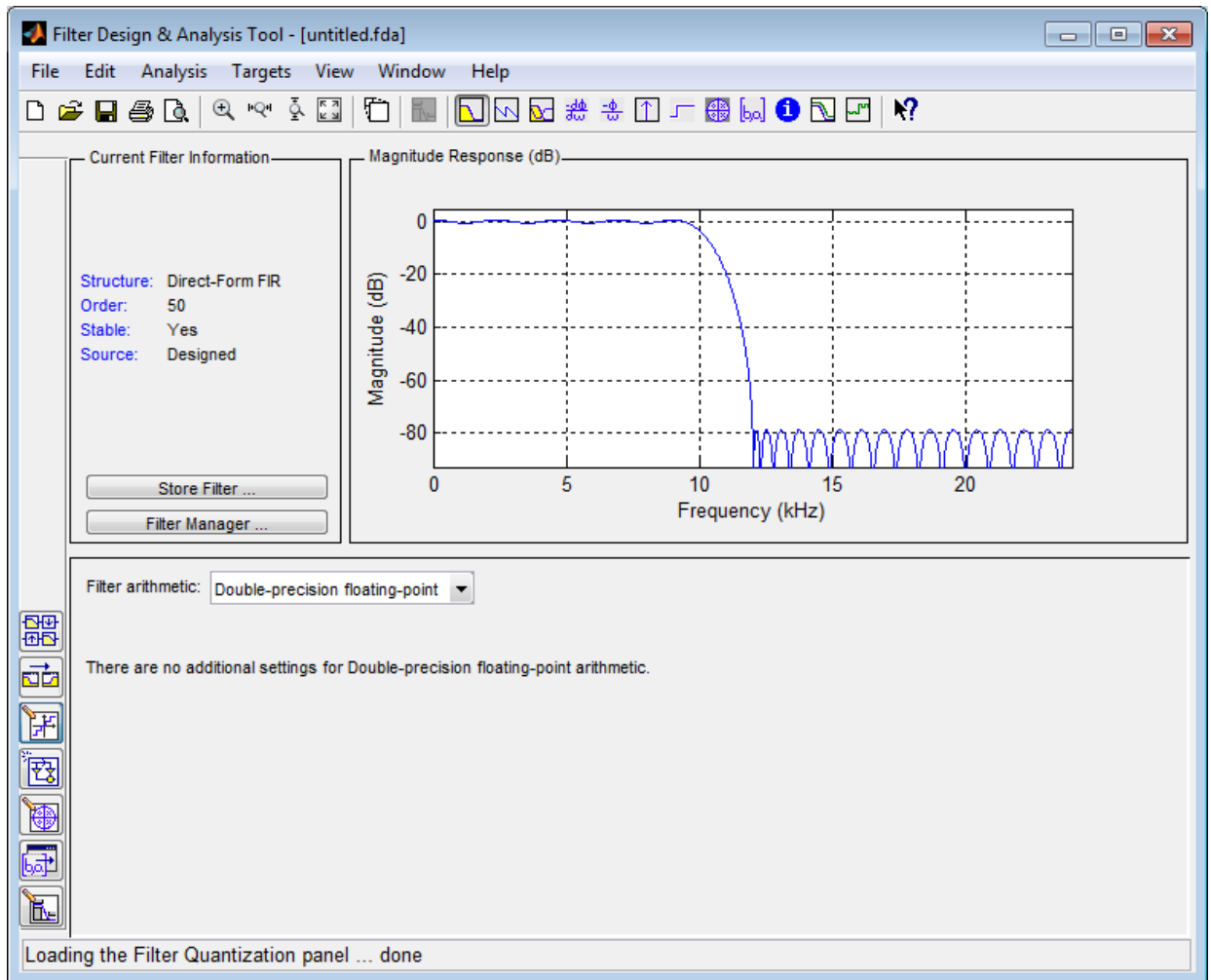
Designing Filter... Done

For more information on designing filters with the FDATool, see the DSP System Toolbox documentation.

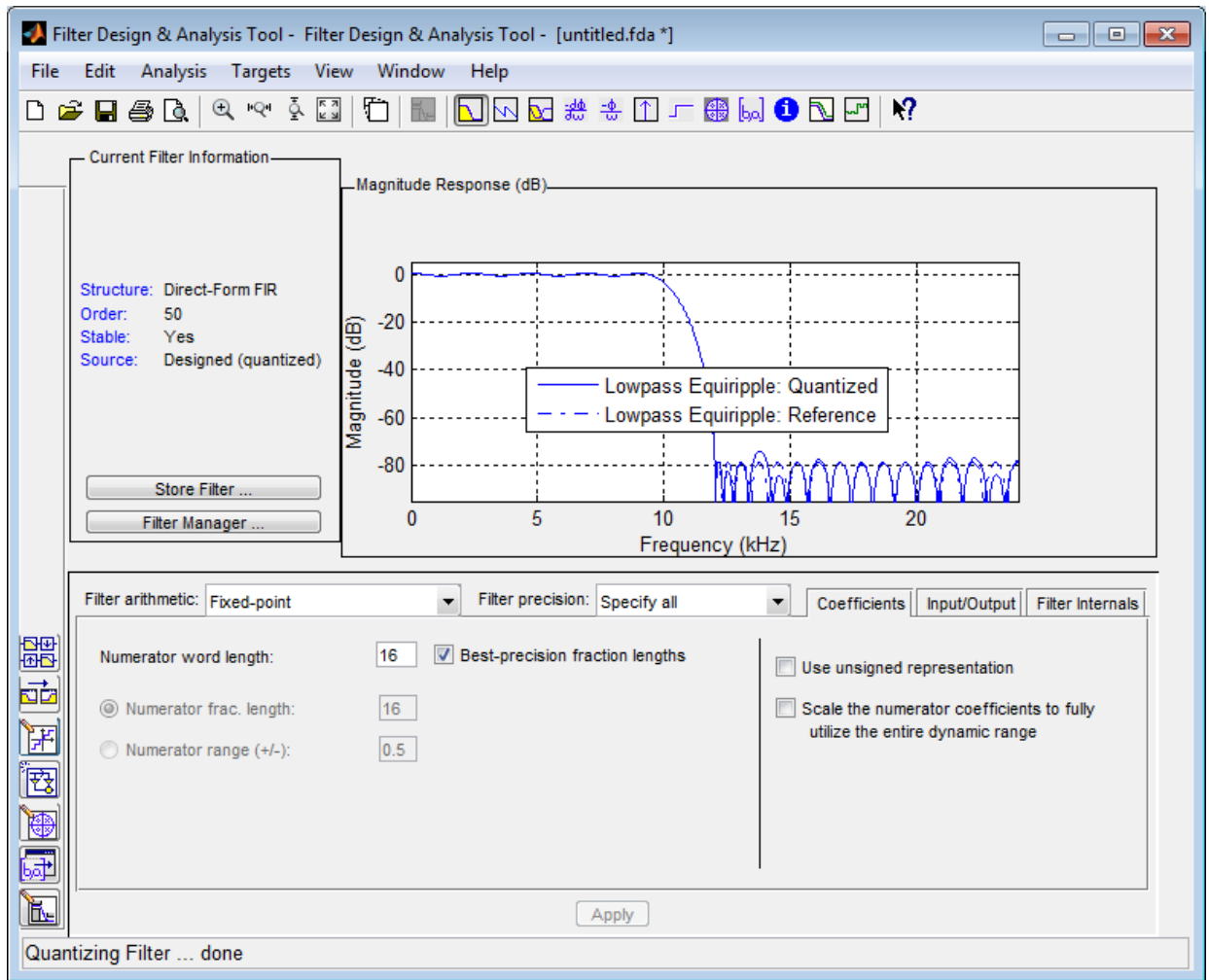
Quantizing the Basic FIR Filter

You should quantize filters for HDL code generation. To quantize your filter,

- 1 Open the basic FIR filter design you created in “Designing a Basic FIR Filter in FDATool” on page 1-6.
- 2 Click the Set Quantization Parameters button  in the left-side toolbar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog box.



- 3 Select **Fixed-point** from the **Filter arithmetic** list. Then select **Specify all** from the **Filter precision** list. The FDATool displays the first of three tabbed panels of quantization parameters across the bottom half of its dialog box.



You use the quantization options to test the effects of various settings with a goal of optimizing the quantized filter's performance and accuracy

- 4 Set the quantization parameters as follows:

Tab	Parameter	Setting
Coefficients	Numerator word length	16
	Best-precision fraction lengths	Selected
	Use unsigned representation	Cleared
	Scale the numerator coefficients to fully utilize the entire dynamic range	Cleared
Input/Output	Input word length	16
	Input fraction length	15
	Output word length	16
Filter Internals	Rounding mode	Floor
	Overflow mode	Saturate
	Accum. word length	40

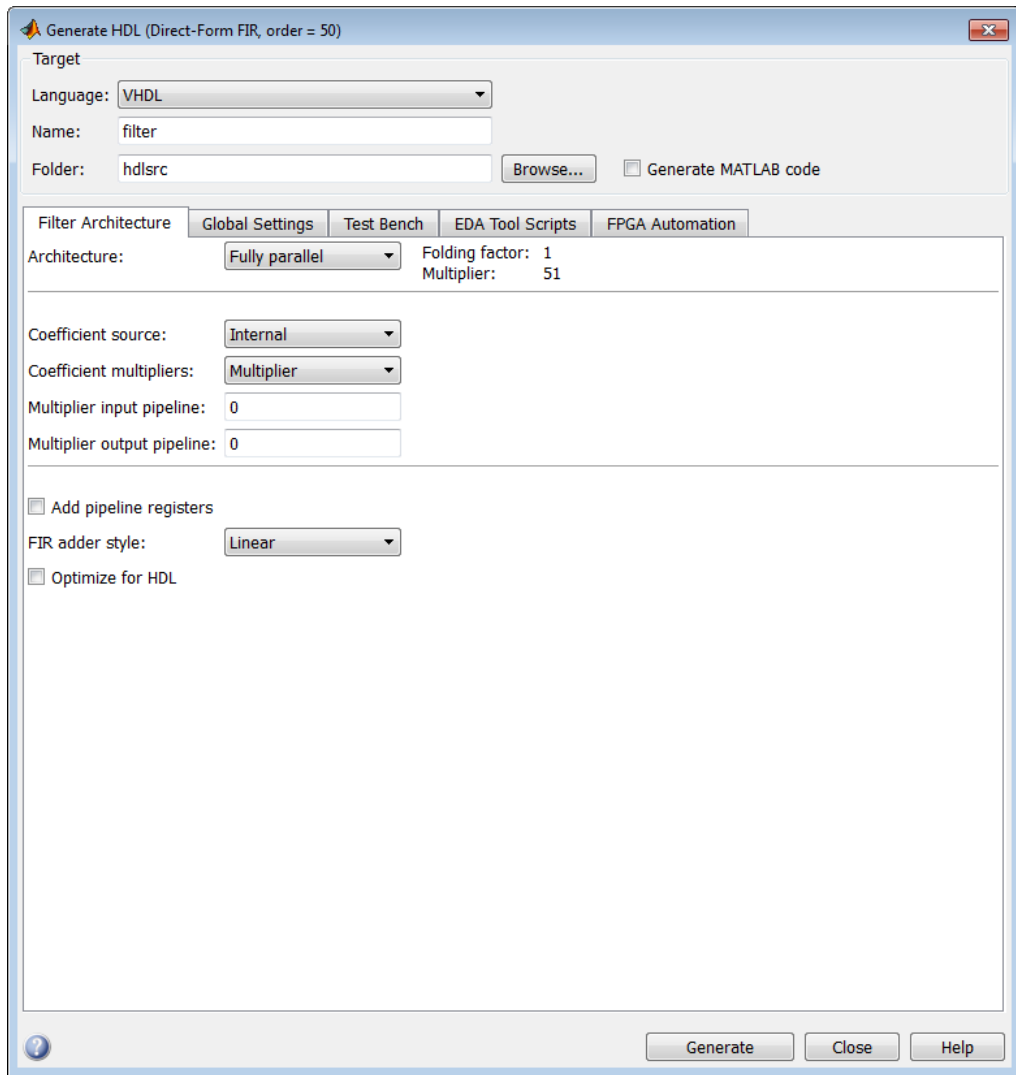
5 Click **Apply**.

For more information on quantizing filters with the FDATool, see the DSP System Toolbox documentation.

Configuring and Generating the Basic FIR Filter's VHDL Code

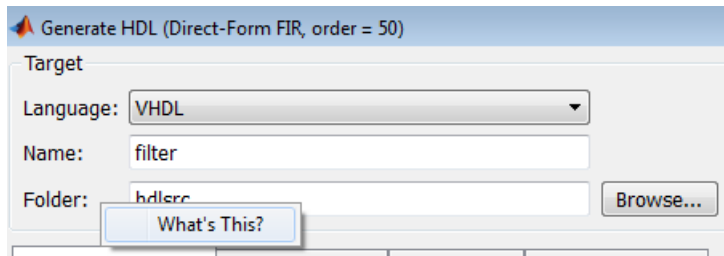
After you quantize your filter, you are ready to configure coder options and generate the filter's VHDL code. This section guides you through the procedure for starting the Filter Design HDL Coder GUI, setting some options, and generating the VHDL code and a test bench for the basic FIR filter you designed and quantized in “Designing a Basic FIR Filter in FDATool” on page 1-6 and “Quantizing the Basic FIR Filter” on page 1-8

1 Start the Filter Design HDL Coder GUI by selecting **Targets > Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog box.

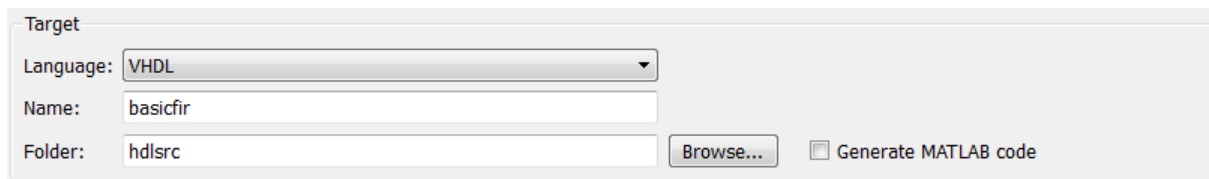


- 2 Find the Filter Design HDL Coder online help. Use the online help to learn about product details or to get answers to questions as you work with the designer.
 - a In the MATLAB window, click the **Help** button in the toolbar or click **Help > Product Help**.

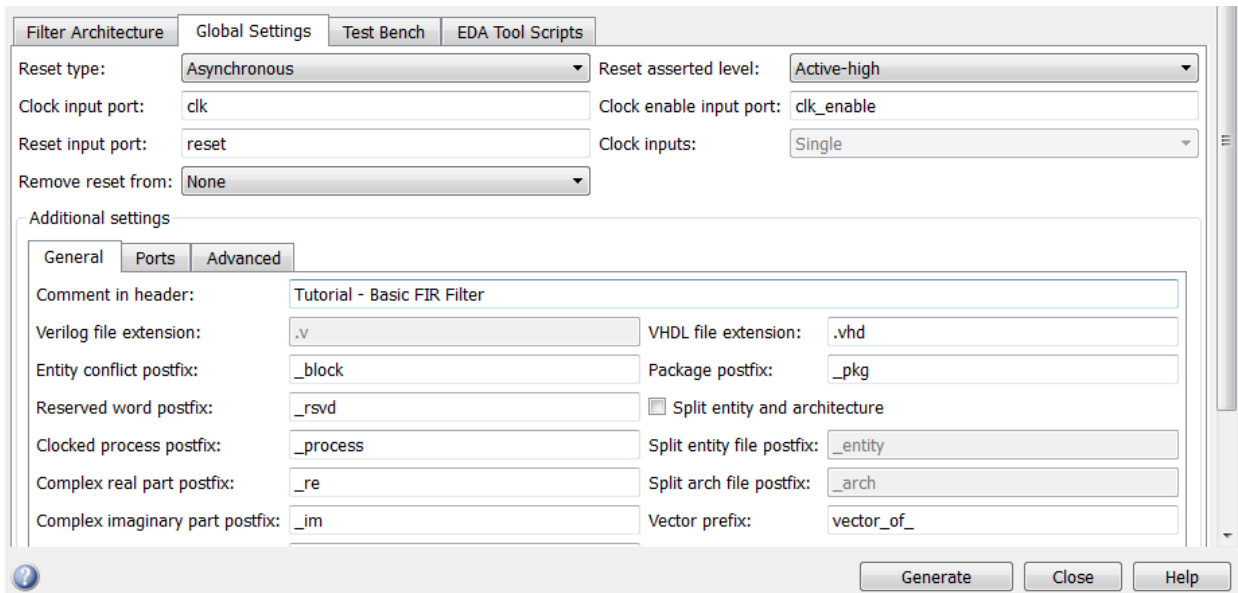
- b In the Help browser's **Contents** pane, select the **Filter Design HDL Coder** entry.
 - c Minimize the Help browser.
- 3 In the Generate HDL dialog box, click the **Help** button. A small context-sensitive help window opens. The window displays information about the dialog box.
- 4 Close the Help window.
- 5 Place your cursor over the **Folder** label or text box in the **Target** pane of the Generate HDL dialog box, and right-click. A What's This? button appears.



- 6 Click **What's This?** The context-sensitive help window displays information describing the **Folder** option. Use the context-sensitive help while using the GUI to configure the contents and style of the generated HDL code. A help topic is available for each option.
- 7 In the **Name** text box of the **Target** pane, replace the default name with `basicfir`. This option names the VHDL entity and the file that is to contain the filter's VHDL code.



- 8 Select the **Global settings** tab of the GUI. Then select the **General** tab of the **Additional settings** section of the GUI. Type `Tutorial - Basic FIR Filter` in the **Comment in header** text box. The coder adds the comment to the end of the header comment block in each generated file.



9 Select the **Ports** tab of the **Additional settings** section of the GUI.



10 Change the names of the input and output ports. In the **Input port** text box, replace **filter_in** with **data_in**. In the **Output port** text box, replace **filter_out** with **data_out**.

Additional settings

General Ports Advanced

Input data type:

Output data type:

Clock enable output port:

Input port:

Output port:

Input complexity:

Add input register

Add output register

- 11** Clear the check box for the **Add input register** option. The **Ports** pane should now look like the following.

Additional settings

General Ports Advanced

Input data type:

Output data type:

Clock enable output port:

Input port:

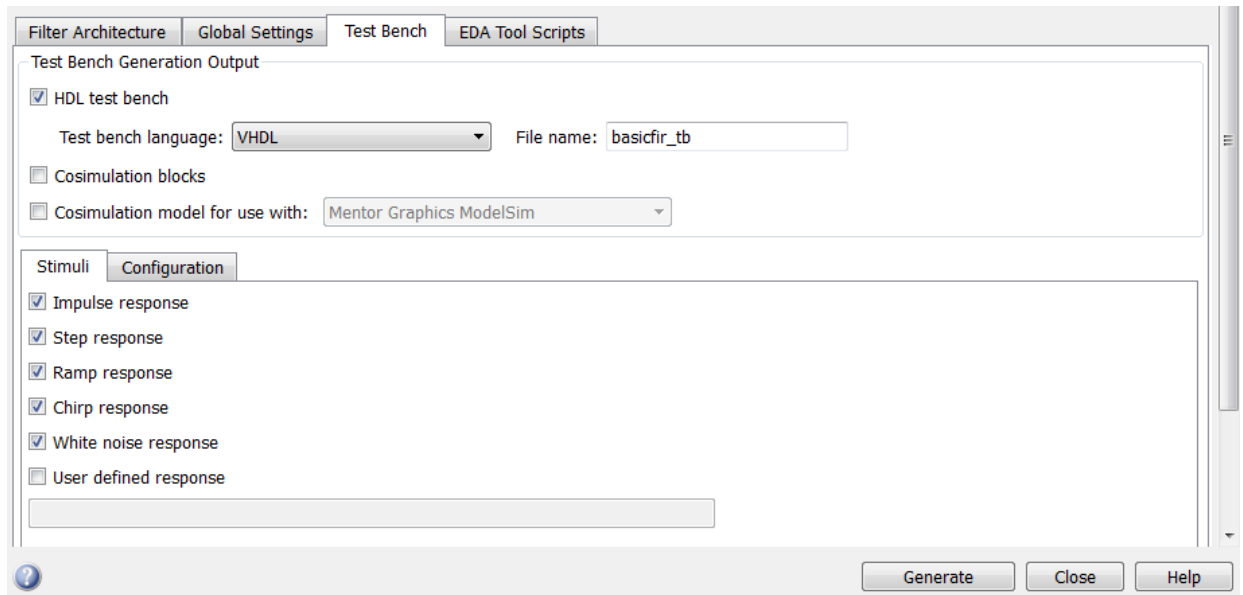
Output port:

Input complexity:

Add input register

Add output register

- 12** Click on the **Test Bench** tab in the Generate HDL dialog box. In the **File name** text box, replace the default name with `basicfir_tb`. This option names the generated test bench file.



13 Click **Generate** to start the code generation process.

The coder displays messages in the MATLAB Command Window as it generates the filter and test bench VHDL files:

```

### Starting VHDL code generation process for filter: basicfir
### Generating: C:\hdlfilter_tutorials\hdlsrc\basicfir.vhd
### Starting generation of basicfir VHDL entity
### Starting generation of basicfir VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: basicfir

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating Test bench: C:\hdlfilter_tutorials\hdlsrc\basicfir_tb.vhd
### Please wait ...
### Done generating VHDL Test Bench
>>

```

As the messages indicate, the coder creates the folder `hdlsrc` under your current working folder and places the files `basicfir.vhd` and `basicfir_tb.vhd` in that folder.

Observe that the messages include hyperlinks to the generated code and test bench files. By clicking on these hyperlinks, you can open the code files directly into the MATLAB Editor.

The generated VHDL code has the following characteristics:

- VHDL entity named `basicfir`.
- Registers that use asynchronous resets when the reset signal is active high (1).
- Ports have the following names:

VHDL Port	Name
Input	<code>data_in</code>
Output	<code>data_out</code>
Clock input	<code>clk</code>
Clock enable input	<code>clk_enable</code>
Reset input	<code>reset</code>

- An extra register for handling filter output.
- Clock input, clock enable input and reset ports are of type `STD_LOGIC` and data input and output ports are of type `STD_LOGIC_VECTOR`.
- Coefficients are named `coeffn`, where n is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'...`
- Registers are generated with the statement `ELSIF clk'event AND clk='1' THEN` rather than with the `rising_edge` function.
- The postfix string `_process` is appended to process names.

The generated test bench:

- Is a portable VHDL file.
- Forces clock, clock enable, and reset input signals.
- Forces the clock enable input signal to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.

- Applies a hold time of 2 nanoseconds to data input signals.
 - For a FIR filter, applies impulse, step, ramp, chirp, and white noise stimulus types.
- 14** When you have finished generating code, click **Close** to close the Generate HDL dialog box.

Getting Familiar with the Basic FIR Filter's Generated VHDL Code

Get familiar with the filter's generated VHDL code by opening and browsing through the file `basicfir.vhd` in an ASCII or HDL simulator editor:

- 1** Open the generated VHDL filter file `basicfir.vhd`.
- 2** Search for `basicfir`. This line identifies the VHDL module, using the string you specified for the **Name** option in the **Target** pane. See step 5 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 3** Search for `Tutorial`. This is where the coder places the text you entered for the **Comment in header** option. See step 10 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 4** Search for `HDL Code`. This section lists coder options you modified in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 5** Search for `Filter Settings`. This section describes the filter design and quantization settings as you specified in “Designing a Basic FIR Filter in FDATool” on page 1-6 and “Quantizing the Basic FIR Filter” on page 1-8.
- 6** Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **Target** pane. See step 5 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 7** Search for `PORT`. This `PORT` declaration defines the filter's clock, clock enable, reset, and data input and output ports. The ports for clock, clock enable, and reset signals are named with default strings. The ports for data input and output are named with the strings you specified for the **Input port** and **Output port** options on the **Ports** tab of the Generate HDL dialog box. See step 12 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 8** Search for `Constants`. This is where the coefficients are defined. They are named using the default naming scheme, `coeffn`, where n is the coefficient number, starting with 1.
- 9** Search for `Signals`. This is where the filter's signals are defined.

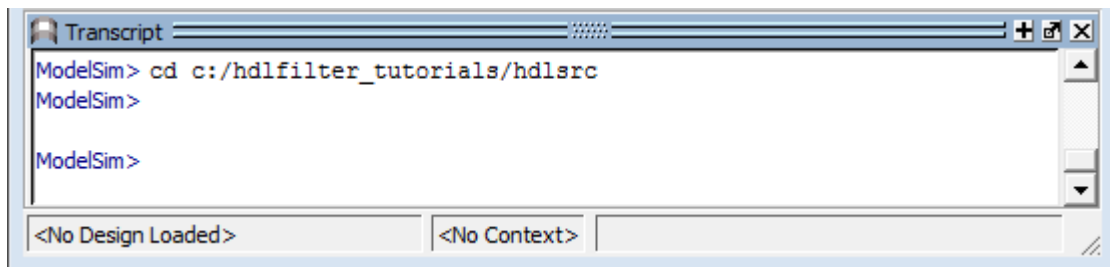
- 10 Search for `process`. The `PROCESS` block name `Delay_Pipeline_process` includes the default `PROCESS` block postfix string `_process`.
- 11 Search for `IF reset`. This is where the reset signal is asserted. The default, active high (1), was specified. Also note that the `PROCESS` block applies the default asynchronous reset style when generating VHDL code for registers.
- 12 Search for `ELSIF`. This is where the VHDL code checks for rising edges when the filter operates on registers. The default `ELSIF clk 'event` statement is used instead of the optional `rising_edge` function.
- 13 Search for `Output_Register`. This is where filter output is written to an output register. Code for this register is generated by default. In step 13 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11, you cleared the **Add input register** option, but left the **Add output register** selected. Also note that the `PROCESS` block name `Output_Register_process` includes the default `PROCESS` block postfix string `_process`.
- 14 Search for `data_out`. This is where the filter writes its output data.

Verifying the Basic FIR Filter's Generated VHDL Code

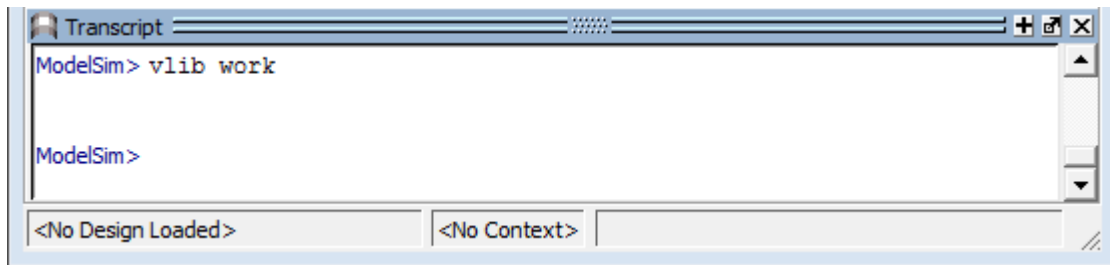
This section explains how to verify the basic FIR filter's generated VHDL code with the generated VHDL test bench. Although this tutorial uses the Mentor Graphics® ModelSim® software as the tool for compiling and simulating the VHDL code, you can use another VHDL simulation tool package.

To verify the filter code, complete the following steps:

- 1 Start your Mentor Graphics ModelSim simulator.
- 2 Set the current folder to the folder that contains your generated VHDL files. For example:



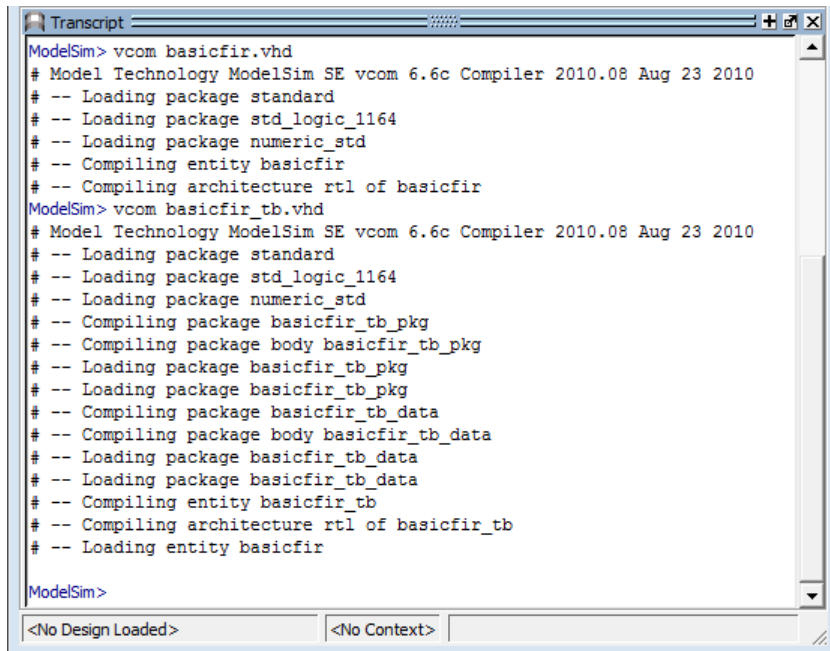
- 3 If desired, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.



- 4 Compile the generated filter and test bench VHDL files. In the Mentor Graphics ModelSim simulator, you compile VHDL code with the `vcom` command. The following commands compile the filter and filter test bench VHDL code.

```
vcom basicfir.vhd  
vcom basicfir_tb.vhd
```

The following screen display shows this command sequence and informational messages displayed during compilation.



```

ModelSim> vcom basicfir.vhd
# Model Technology ModelSim SE vcom 6.6c Compiler 2010.08 Aug 23 2010
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity basicfir
# -- Compiling architecture rtl of basicfir
ModelSim> vcom basicfir_tb.vhd
# Model Technology ModelSim SE vcom 6.6c Compiler 2010.08 Aug 23 2010
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling package basicfir_tb_pkg
# -- Compiling package body basicfir_tb_pkg
# -- Loading package basicfir_tb_pkg
# -- Loading package basicfir_tb_pkg
# -- Compiling package basicfir_tb_data
# -- Compiling package body basicfir_tb_data
# -- Loading package basicfir_tb_data
# -- Loading package basicfir_tb_data
# -- Compiling entity basicfir_tb
# -- Compiling architecture rtl of basicfir_tb
# -- Loading entity basicfir

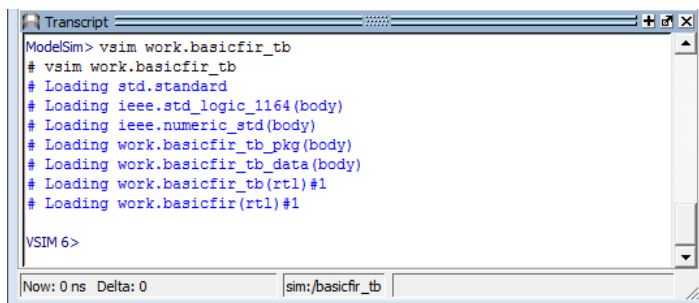
ModelSim>
<No Design Loaded> <No Context>

```

- 5 Load the test bench for simulation. The procedure for doing this varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.basicfir_tb
```

The following figure shows the results of loading `work.basicfir_tb` with the `vsim` command.



```

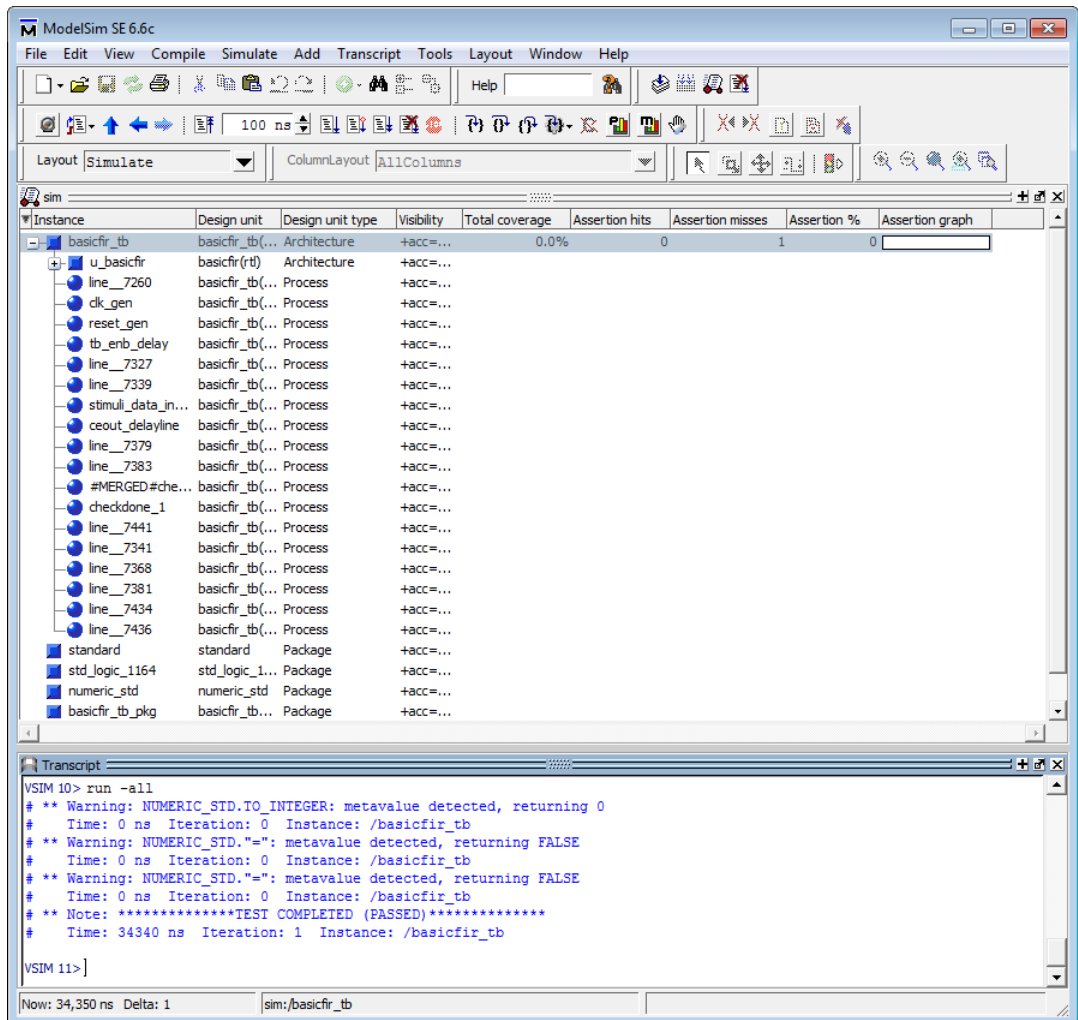
ModelSim> vsim work.basicfir_tb
# vsim work.basicfir_tb
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.basicfir_tb_pkg(body)
# Loading work.basicfir_tb_data(body)
# Loading work.basicfir_tb(rtl)#1
# Loading work.basicfir(rtl)#1

VSIM 6>
Now: 0 ns Delta: 0 sim:/basicfir_tb

```

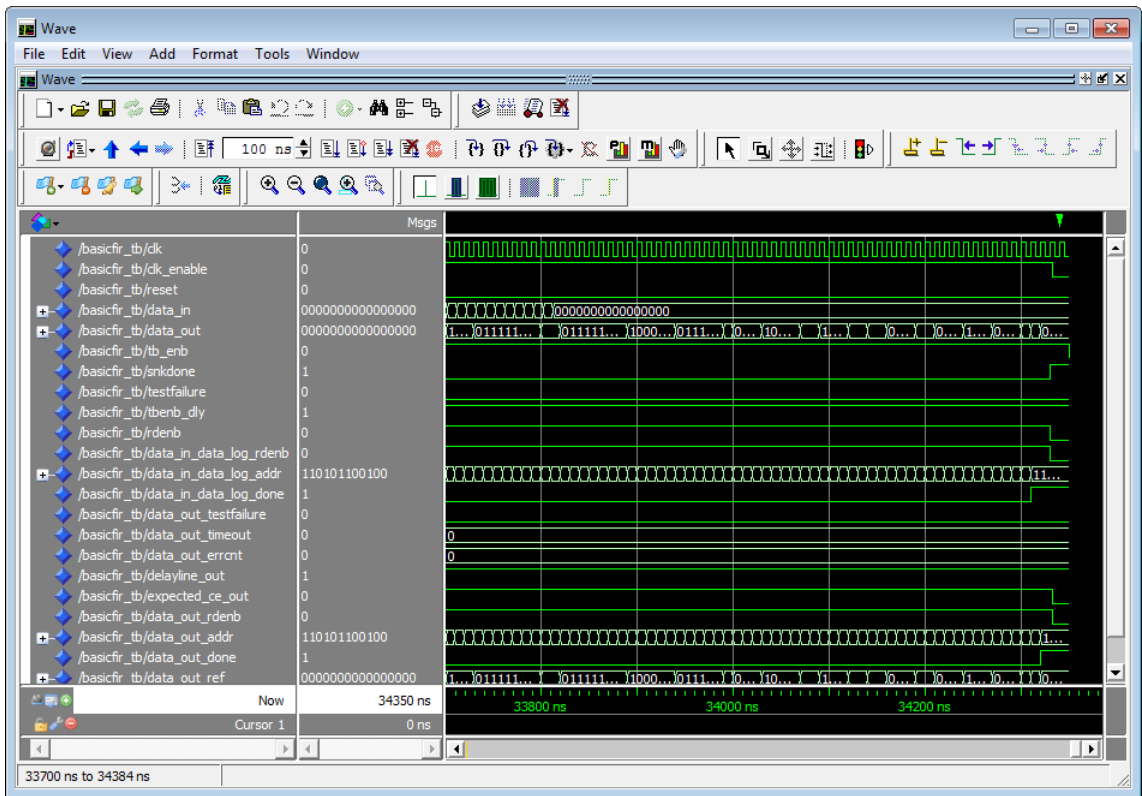

- 7 To start running the simulation, issue the start simulation command for your simulator. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the `run` command.

The following display shows the `run -all` command being used to start a simulation.



As your test bench simulation runs, watch for error messages. If error messages appear, you must interpret them as they pertain to your filter design and the HDL code generation options you selected. You must determine whether the results are expected based on the customizations you specified when generating the filter VHDL code.

The following **wave** window shows the simulation results as HDL waveforms.



Optimized FIR Filter

- “Designing the FIR Filter in FDATool” on page 1-25
- “Quantizing the FIR Filter” on page 1-27

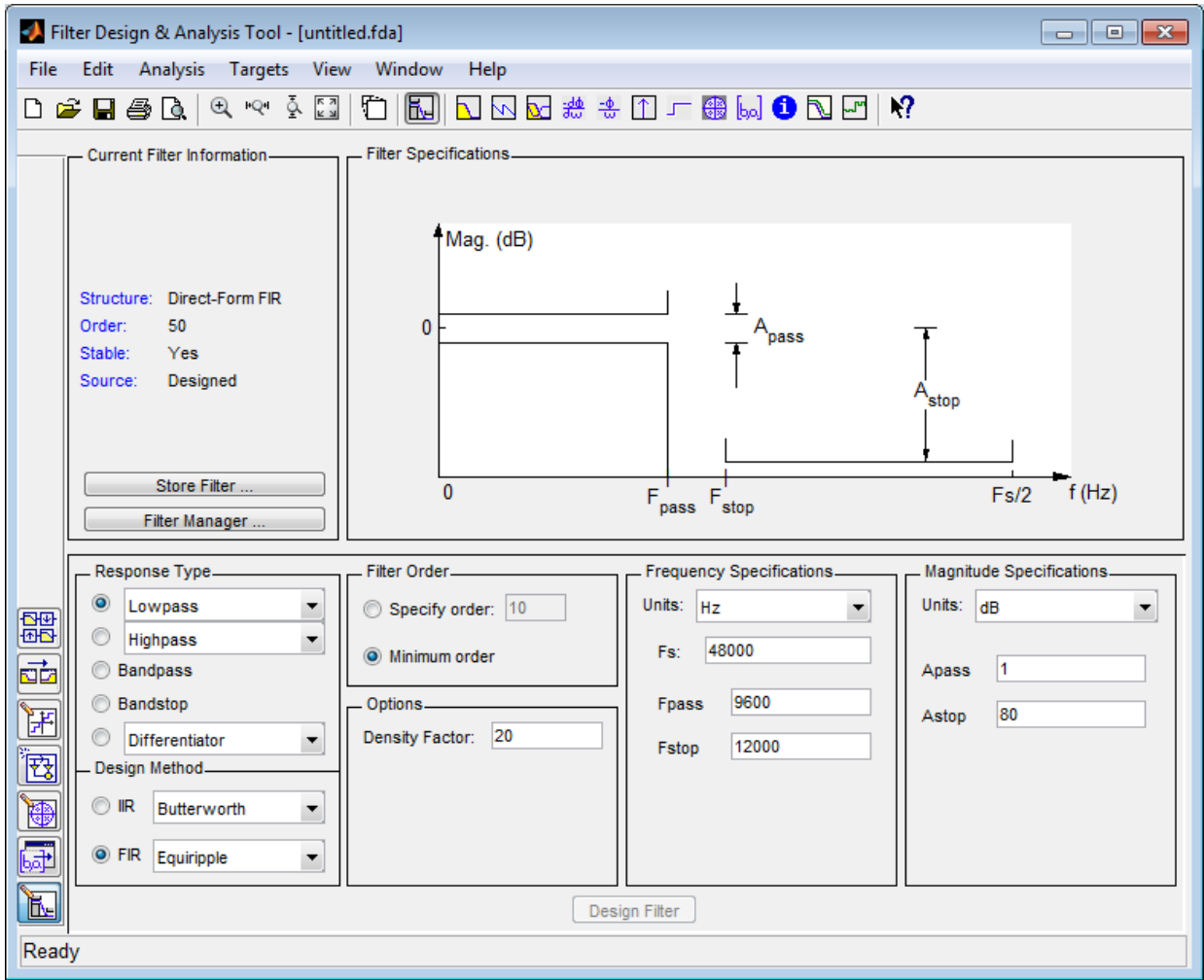
- “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30
- “Getting Familiar with the FIR Filter's Optimized Generated Verilog Code” on page 1-39
- “Verifying the FIR Filter's Optimized Generated Verilog Code” on page 1-40

Designing the FIR Filter in FDATool

This tutorial guides you through the steps for designing an optimized quantized discrete-time FIR filter, generating Verilog code for the filter, and verifying the Verilog code with a generated test bench.

This section assumes you are familiar with the MATLAB user interface and the Filter Design & Analysis Tool (FDATool).

- 1** Start the MATLAB software.
- 2** Set your current folder to the folder you created in “Creating a Folder for Your Tutorial Files” on page 1-6.
- 3** Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



4 In the Filter Design & Analysis Tool dialog box, set the following filter options:

Option	Value
Response Type	Lowpass
Design Method	FIR Equiripple
Filter Order	Minimum order

Option	Value
Options	Density Factor: 20
Frequency Specifications	Units: Hz
	Fs: 48000
	Fpass: 9600
	Fstop: 12000
Magnitude Specifications	Units: dB
	Apass: 1
	Astop: 80

These settings are for the default filter design that the FDATool creates for you. If you do not have to make changes and **Design Filter** is grayed out, you are done and can skip to “Quantizing the Basic FIR Filter” on page 1-8.

- 5 Click **Design Filter**. The FDATool creates a filter for the specified design. The following message appears in the FDATool status bar when the task is complete.

Designing Filter... Done


For more information on designing filters with the FDATool, see the DSP System Toolbox documentation.

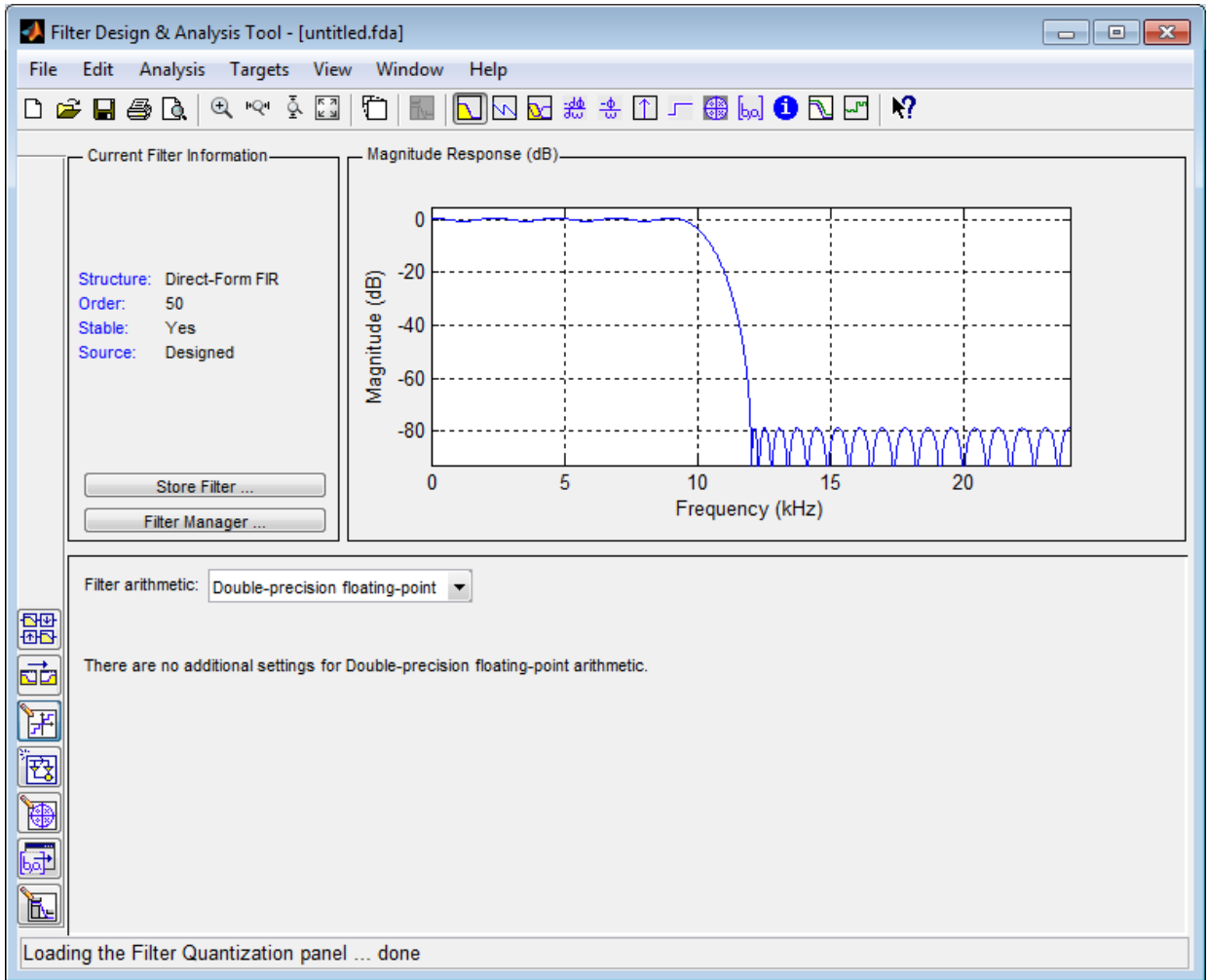
Quantizing the FIR Filter

You should quantize filters for HDL code generation. To quantize your filter,

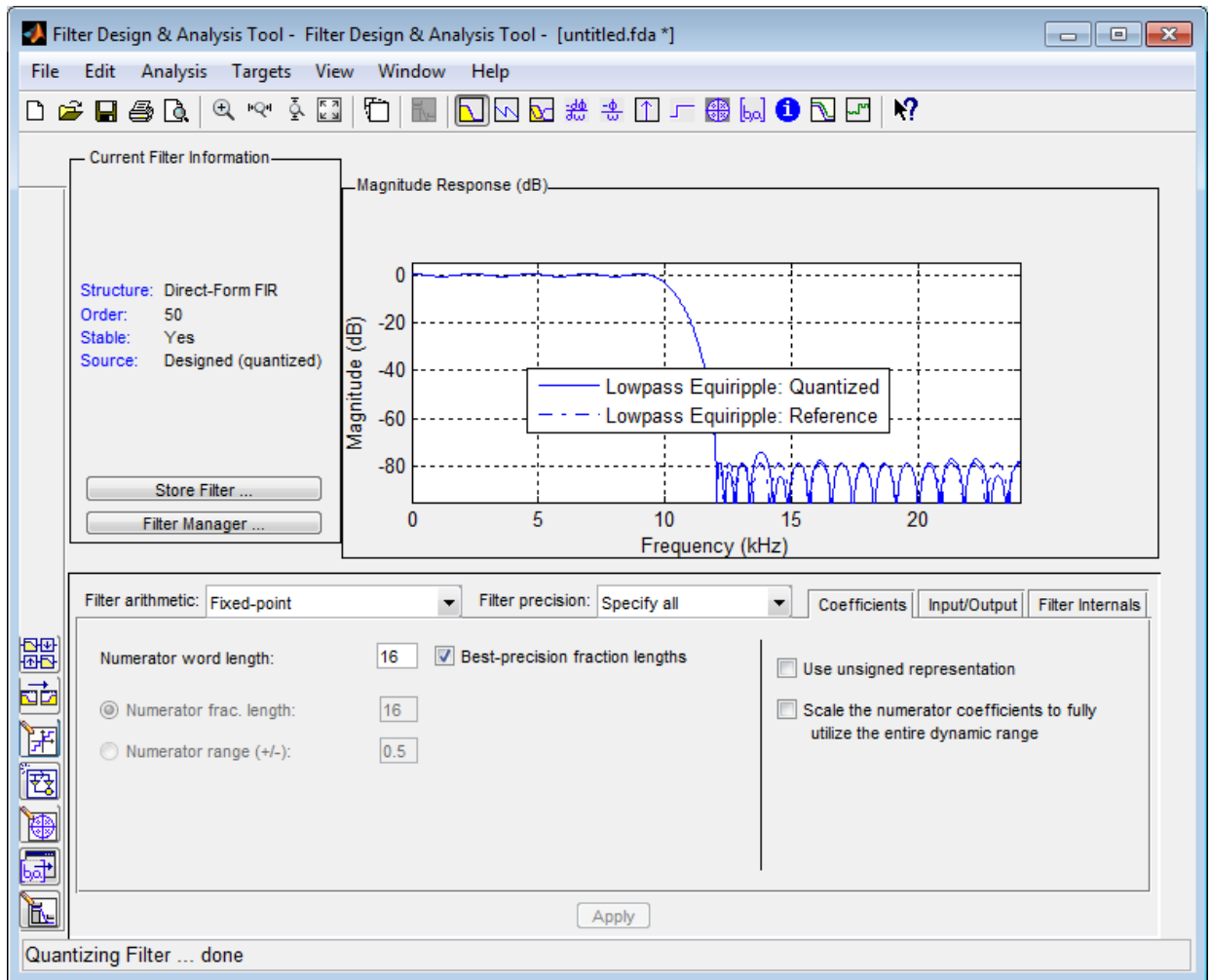
- 1 Open the FIR filter design you created in “Optimized FIR Filter” on page 1-24 if it is not already open.

2

Click the Set Quantization Parameters button  in the left-side toolbar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog box.



- 3 Select **Fixed-point** from the list. Then select **Specify all** from the **Filter precision** list. The FDATool displays the first of three tabbed panels of quantization parameters across the bottom half of its dialog box.



You use the quantization options to test the effects of various settings with a goal of optimizing the quantized filter's performance and accuracy.

- 4 Set the quantization parameters as follows:

Tab	Parameter	Setting
Coefficients	Numerator word length	16
	Best-precision fraction lengths	Selected
	Use unsigned representation	Cleared
	Scale the numerator coefficients to fully utilize the entire dynamic range	Cleared
Input/Output	Input word length	16
	Input fraction length	15
	Output word length	16
Filter Internals	Rounding mode	Floor
	Overflow mode	Saturate
	Accum. word length	40

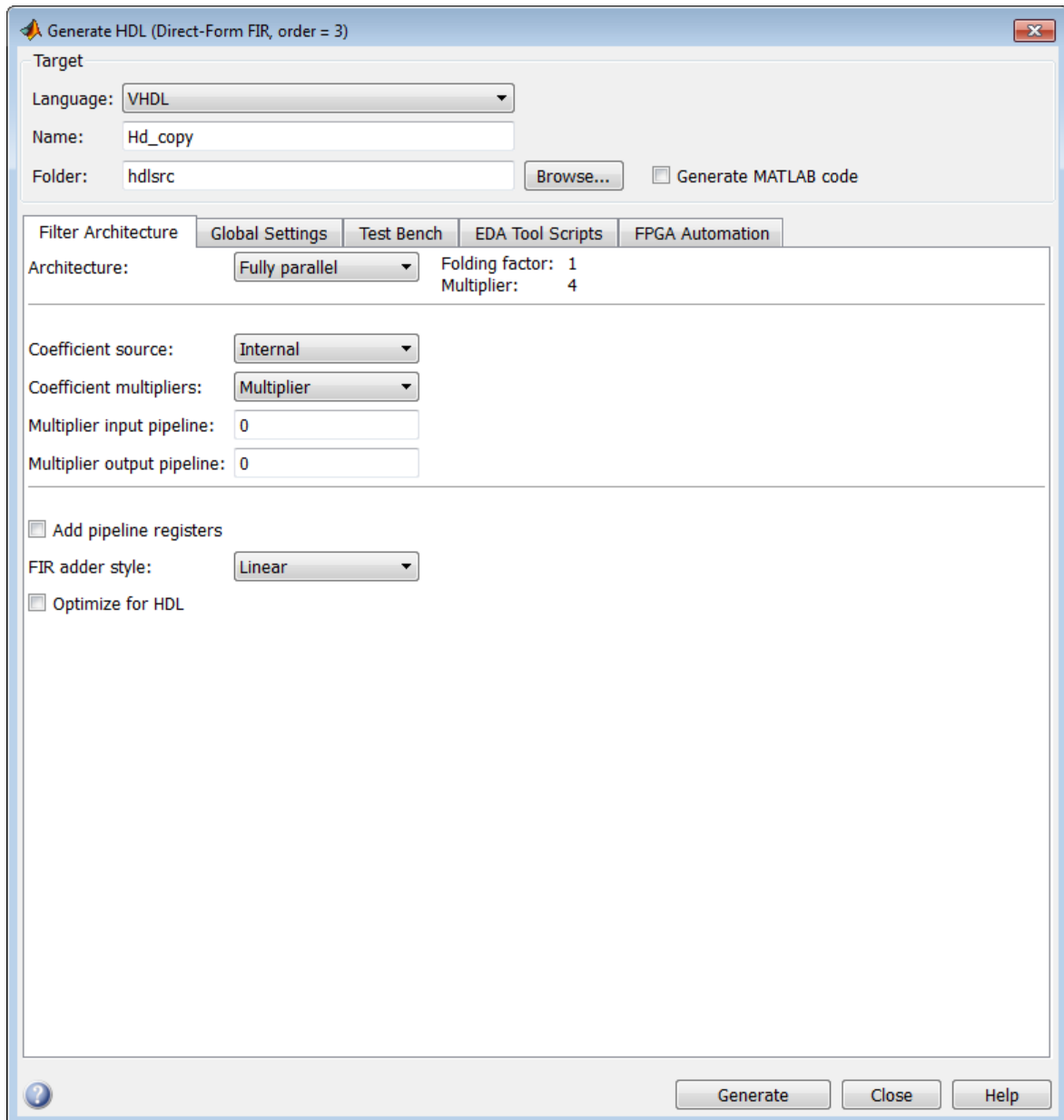
5 Click **Apply**.

For more information on quantizing filters with the FDATool, see the DSP System Toolbox documentation.

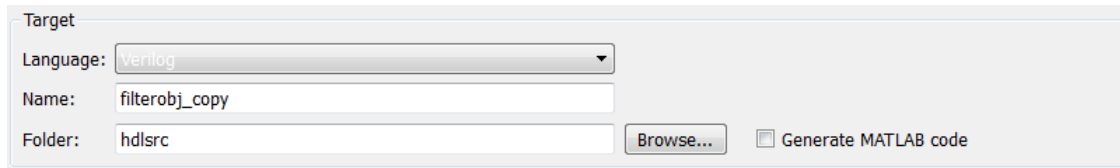
Configuring and Generating the FIR Filter's Optimized Verilog Code

After you quantize your filter, you are ready to configure coder options and generate the filter's Verilog code. This section guides you through the process for starting the GUI, setting some options, and generating the Verilog code and a test bench for the FIR filter you designed and quantized in “Designing the FIR Filter in FDATool” on page 1-25 and “Quantizing the FIR Filter” on page 1-27.

- 1 Start the Filter Design HDL Coder GUI by selecting **Targets > Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog box.

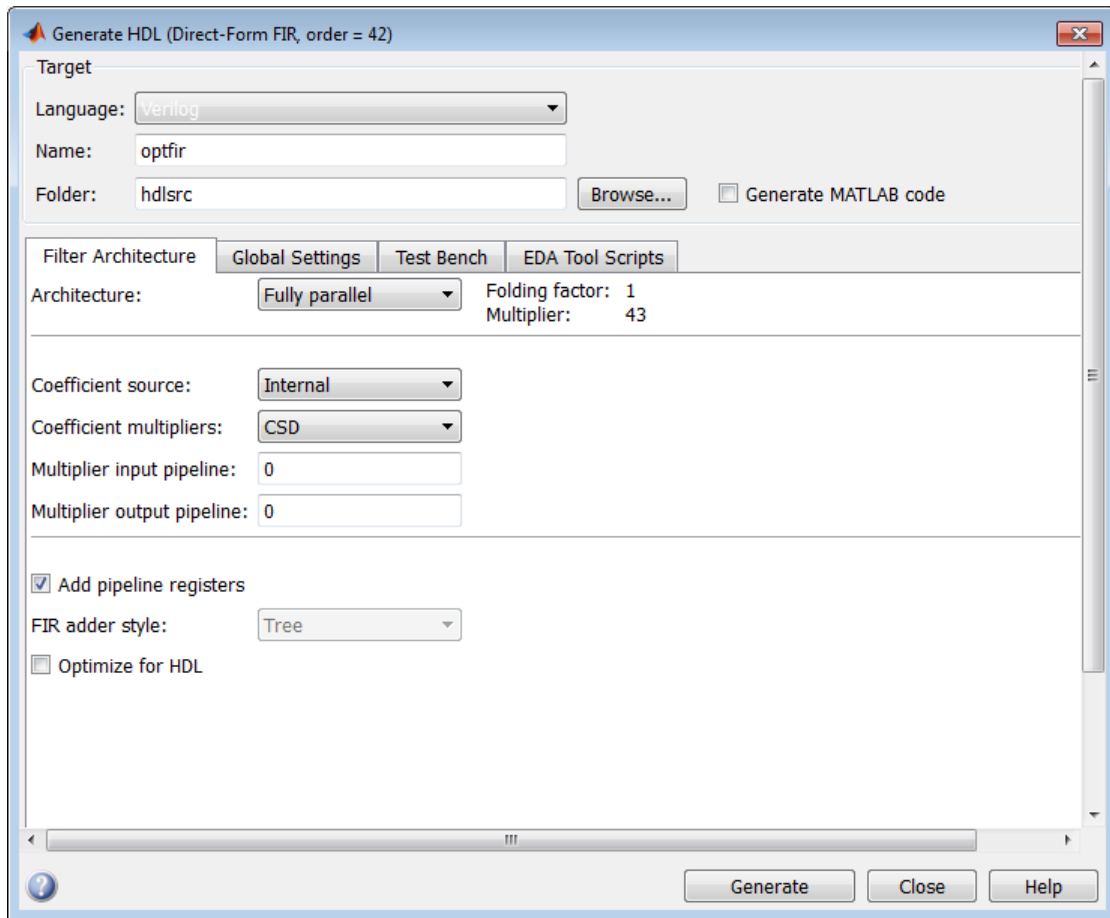


- 2 Select Verilog for the **Language** option, as shown in the following figure.



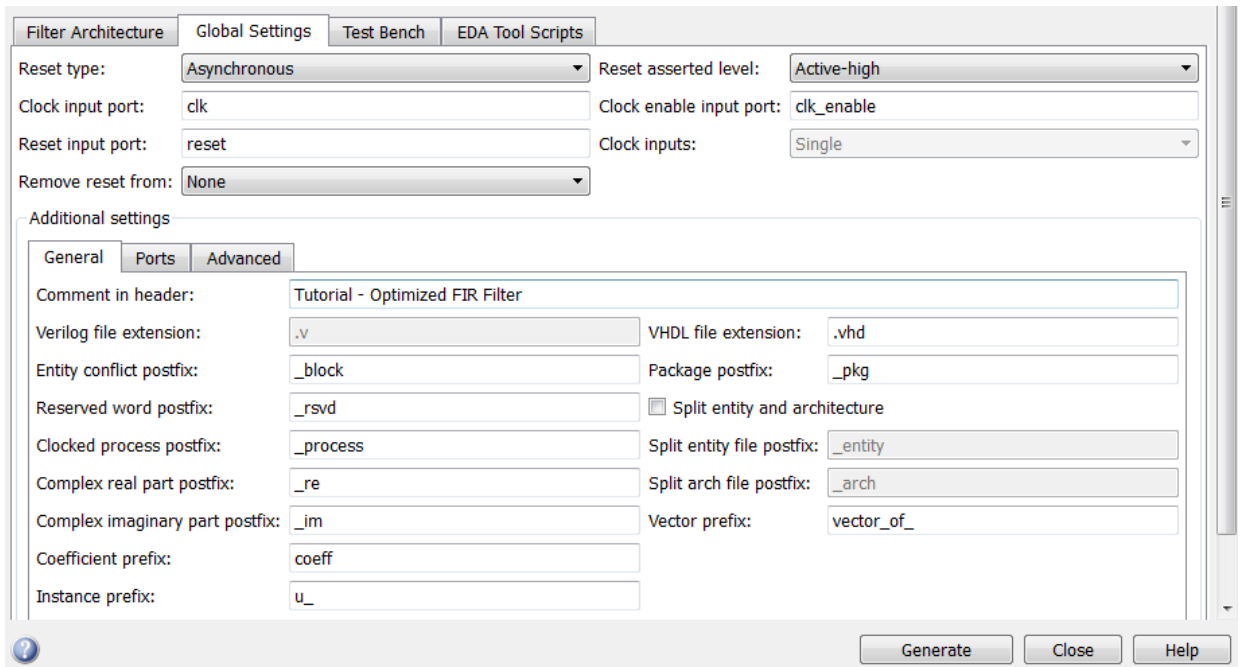
The image shows a 'Target' configuration dialog box. It has three main input fields: 'Language' is a dropdown menu currently showing 'Verilog'; 'Name' is a text box containing 'filterobj_copy'; and 'Folder' is a text box containing 'hdlsrc'. To the right of the 'Folder' field is a 'Browse...' button. Further to the right is a checkbox labeled 'Generate MATLAB code', which is currently unchecked.

- 3 In the **Name** text box of the **Target** pane, replace the default name with `optfir`. This option names the Verilog module and the file that is to contain the filter's Verilog code.
- 4 In the **Filter architecture** pane, select the **Optimize for HDL** option. This option is for generating HDL code that is optimized for performance or space requirements. When this option is enabled, the coder makes tradeoffs concerning data types and might ignore your quantization settings to achieve optimizations. When you use the option, keep in mind that you do so at the cost of potential numeric differences between filter results produced by the original filter object and the simulated results for the optimized HDL code.
- 5 Select **CSD** for the **Coefficient multipliers** option. This option optimizes coefficient multiplier operations by instructing the coder to replace them with additions of partial products produced by a canonical signed digit (CSD) technique. This technique minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
- 6 Select the **Add pipeline registers** option. For FIR filters, this option optimizes final summation. The coder creates a final adder that performs pair-wise addition on successive products and includes a stage of pipeline registers after each level of the tree. When used for FIR filters, this option also has the potential for producing numeric differences between results produced by the original filter object and the simulated results for the optimized HDL code.
- 7 The Generate HDL dialog box should now appear as shown in the following figure.



- 8 Select the **Global settings** tab of the GUI. Then select the **General** tab of the **Additional settings** section.

In the **Comment in header** text box, type `Tutorial - Optimized FIR Filter`. The coder adds the comment to the end of the header comment block in each generated file.



9 Select the **Ports** tab of the **Additional settings** section of the GUI.



- 10** Change the names of the input and output ports. In the **Input port** text box, replace `filter_in` with `data_in`. In the **Output port** text box, replace `filter_out` with `data_out`.

Additional settings

General Ports Advanced

Input data type:

Output data type:

Clock enable output port:

Input port:

Output port:

Input complexity:

Add input register

Add output register

- 11** Clear the check box for the **Add input register** option. The **Ports** pane should now look like the following.

Additional settings

General Ports Advanced

Input data type:

Output data type:

Clock enable output port:

Input port:

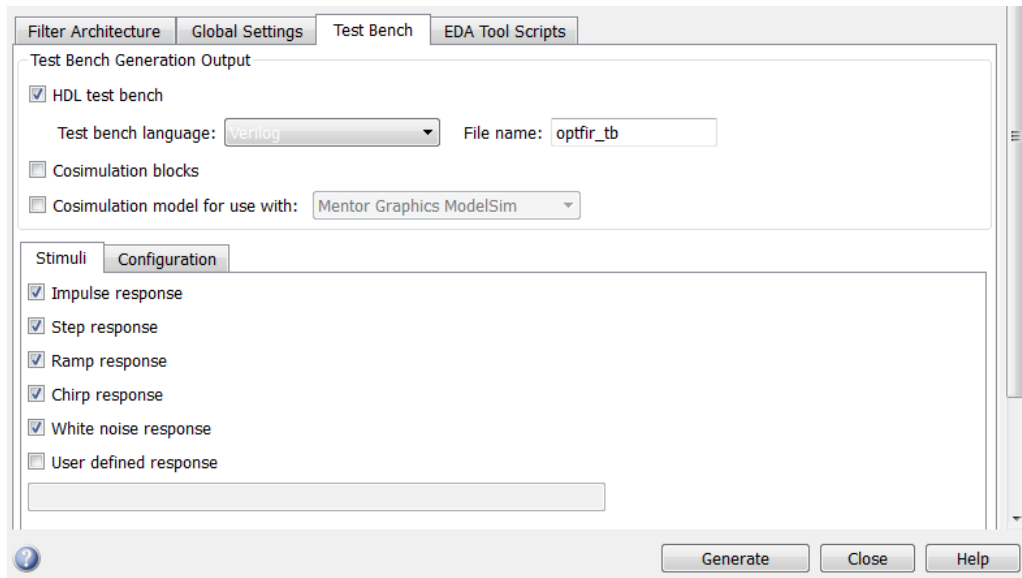
Output port:

Input complexity:

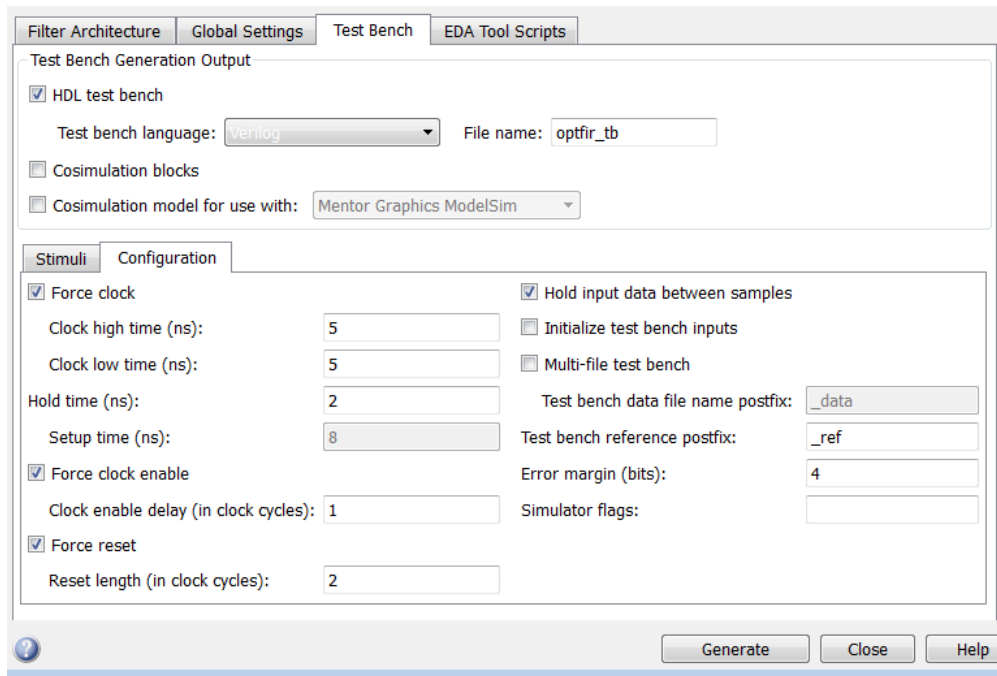
Add input register

Add output register

- 12** Click on the **Test Bench** tab in the Generate HDL dialog box. In the **File name** text box, replace the default name with `optfir_tb`. This option names the generated test bench file.



- 13** In the Test Bench pane, click the **Configuration** tab. Observe that the **Error margin (bits)** option is enabled. This option is enabled because previously selected optimization options (such as **Add pipeline registers**) can potentially produce numeric results that differ from the results produced by the original filter object. You can use this option to adjust the number of least significant bits the test bench will ignore during comparisons before generating a warning.



- 14** In the Generate HDL dialog box, click **Generate** to start the code generation process. When code generation completes, click **Close** to close the dialog box.

The coder displays the following messages in the MATLAB Command Window as it generates the filter and test bench Verilog files:

```
### Starting Verilog code generation process for filter: optfir
### Generating: C:\hdlfilter_tutorials\hdlsrc\optfir.v
### Starting generation of optfir Verilog module
### Starting generation of optfir Verilog module body
### HDL latency is 8 samples
### Successful completion of Verilog code generation process for filter: optfir

### Starting generation of VERILOG Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating Test bench: C:\hdlfilter_tutorials\hdlsrc\optfir_tb.v
### Please wait ...
### Done generating VERILOG Test Bench
```

As the messages indicate, the coder creates the folder `hdlsrc` under your current working folder and places the files `optfir.v` and `optfir_tb.v` in that folder.

Observe that the messages include hyperlinks to the generated code and test bench files. By clicking on these hyperlinks, you can open the code files directly into the MATLAB Editor.

The generated Verilog code has the following characteristics:

- Verilog module named `optfir`.
- Registers that use asynchronous resets when the reset signal is active high (1).
- Generated code that optimizes its use of data types and eliminates redundant operations.
- Coefficient multipliers optimized with the CSD technique.
- Final summations optimized using a pipelined technique.
- Ports that have the following names:

Verilog Port	Name
Input	<code>data_in</code>
Output	<code>data_out</code>
Clock input	<code>clk</code>
Clock enable input	<code>clk_enable</code>
Reset input	<code>reset</code>

- An extra register for handling filter output.
- Coefficients named `coeffn`, where n is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'...`
- The postfix string `_process` is appended to sequential (`begin`) block names.

The generated test bench:

- Is a portable Verilog file.
- Forces clock, clock enable, and reset input signals.
- Forces the clock enable input signal to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.

- Applies a hold time of 2 nanoseconds to data input signals.
- Applies an error margin of 4 bits.
- For a FIR filter, applies impulse, step, ramp, chirp, and white noise stimulus types.

Getting Familiar with the FIR Filter's Optimized Generated Verilog Code

Get familiar with the filter's optimized generated Verilog code by opening and browsing through the file `optfir.v` in an ASCII or HDL simulator editor:

- 1 Open the generated Verilog filter file `optcfir.v`.
- 2 Search for `optfir`. This line identifies the Verilog module, using the string you specified for the **Name** option in the **Target** pane. See step 3 in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 3 Search for `Tutorial`. This is where the coder places the text you entered for the **Comment in header** option. See step 9 in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 4 Search for `HDL Code`. This section lists the coder options you modified in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 5 Search for `Filter Settings`. This section of the VHDL code describes the filter design and quantization settings as you specified in “Designing the FIR Filter in FDATool” on page 1-25 and “Quantizing the FIR Filter” on page 1-27.
- 6 Search for `module`. This line names the Verilog module, using the string you specified for the **Name** option in the **Target** pane. This line also declares the list of ports, as defined by options on the **Ports** pane of the Generate HDL dialog box. The ports for data input and output are named with the strings you specified for the **Input port** and **Output port** options on the **Ports** tab of the Generate HDL dialog box. See steps 3 and 11 in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 7 Search for `input`. This line and the four lines that follow, declare the direction mode of each port.
- 8 Search for `Constants`. This is where the coefficients are defined. They are named using the default naming scheme, `coeffn`, where *n* is the coefficient number, starting with 1.
- 9 Search for `Signals`. This is where the filter's signals are defined.

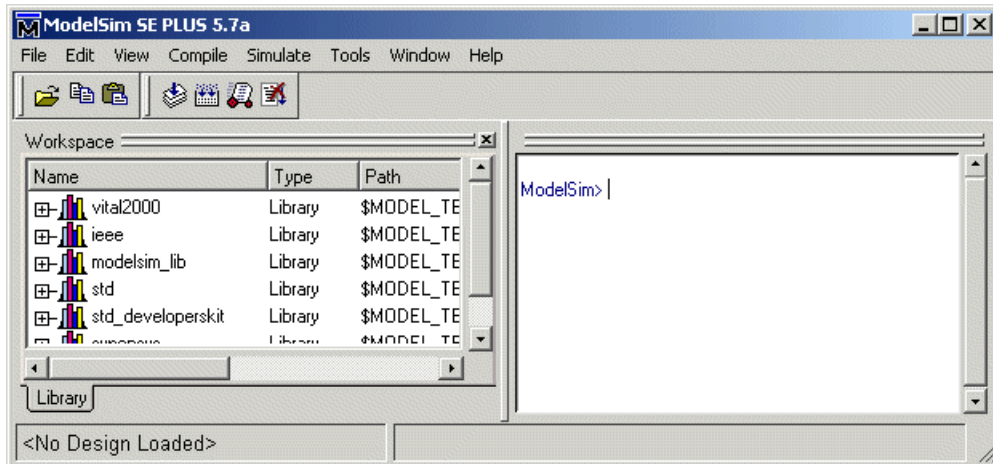
- 10 Search for `sumvector1`. This area of code declares the signals for implementing an instance of a pipelined final adder. Signal declarations for four additional pipelined final adders are also included. These signals are used to implement the pipelined FIR adder style optimization specified with the **Add pipeline registers** option. See step 7 in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 11 Search for `process`. The block name `Delay_Pipeline_process` includes the default block postfix string `_process`.
- 12 Search for `reset`. This is where the reset signal is asserted. The default, active high (1), was specified. Also note that the `process` applies the default asynchronous reset style when generating code for registers.
- 13 Search for `posedge`. This Verilog code checks for rising edges when the filter operates on registers.
- 14 Search for `sumdelay_pipeline_process1`. This block implements the pipeline register stage of the pipeline FIR adder style you specified in step 7 of “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30.
- 15 Search for `output_register`. This is where filter output is written to an output register. The code for this register is generated by default. In step 12 in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 1-30 , you cleared the **Add input register** option, but left the **Add output register** selected. Also note that the process name `Output_Register_process` includes the default `process` postfix string `_process`.
- 16 Search for `data_out`. This is where the filter writes its output data.

Verifying the FIR Filter's Optimized Generated Verilog Code

This section explains how to verify the FIR filter's optimized generated Verilog code with the generated Verilog test bench. Although this tutorial uses the Mentor Graphics ModelSim simulator as the tool for compiling and simulating the Verilog code, you can use another HDL simulation tool package.

To verify the filter code, complete the following steps:

- 1 Start your simulator. When you start the Mentor Graphics ModelSim simulator, a screen display similar to the following appears.



- 2 Set the current folder to the folder that contains your generated Verilog files. For example:

```
cd hdlsrc
```

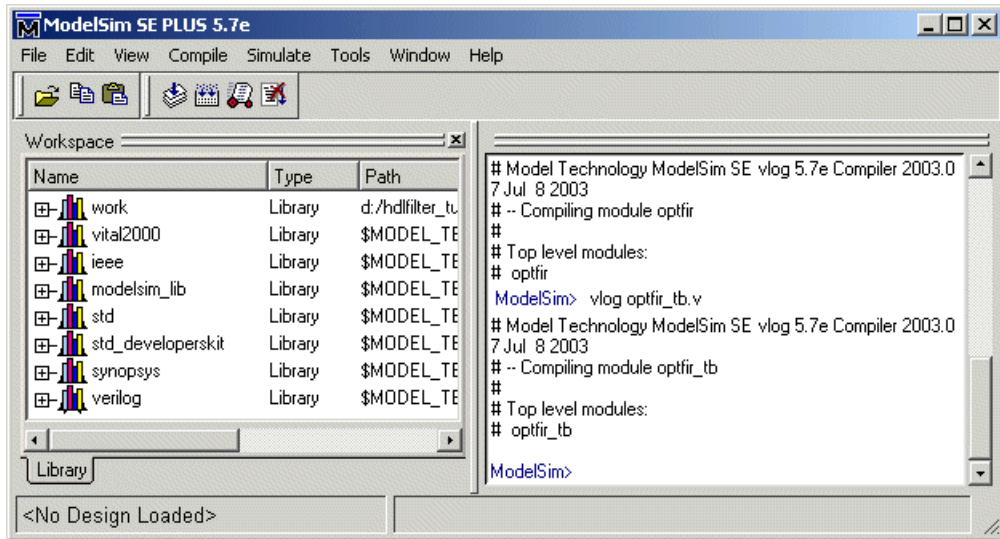
- 3 If desired, create a design library to store the compiled Verilog modules. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

- 4 Compile the generated filter and test bench Verilog files. In the Mentor Graphics ModelSim simulator, you compile Verilog code with the `vlog` command. The following commands compile the filter and filter test bench Verilog code.

```
vlog optfir.v
vlog optfir_tb.v
```

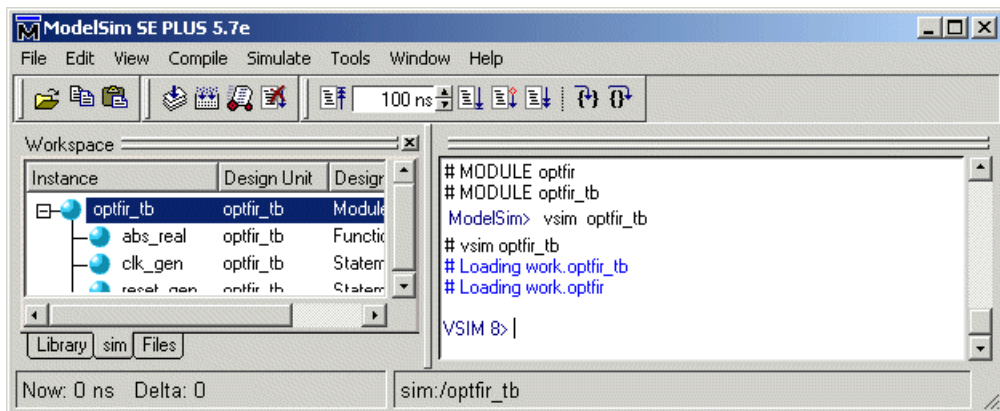
The following screen display shows this command sequence and informational messages displayed during compilation.



- 5 Load the test bench for simulation. The procedure for doing this varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim optfir_tb
```

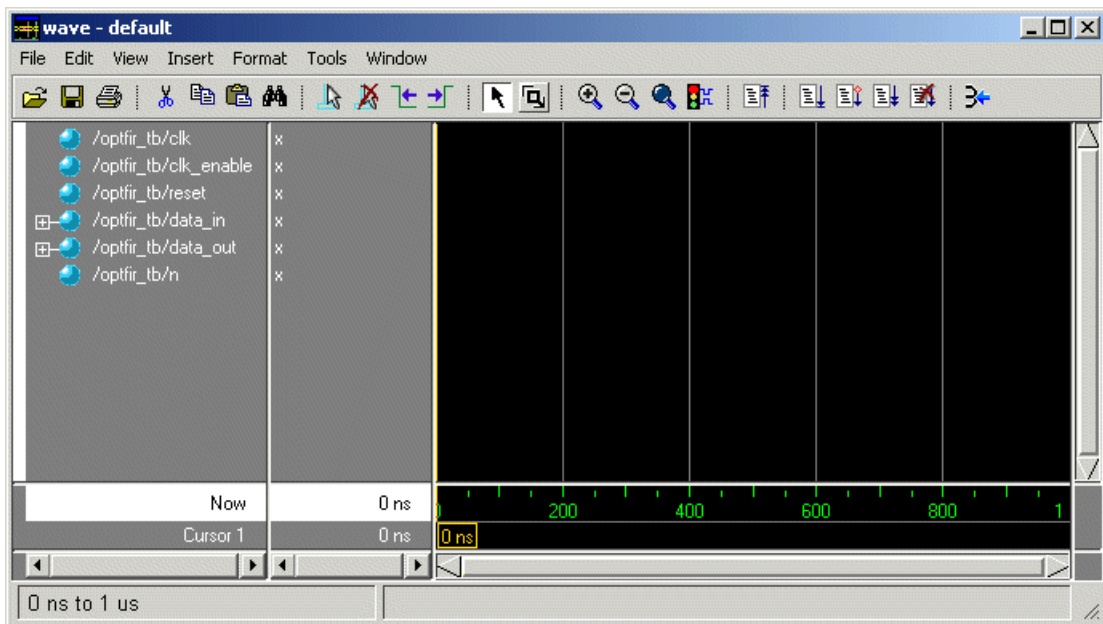
The following display shows the results of loading `optfir_tb` with the `vsim` command.



- 6 Open a display window for monitoring the simulation as the test bench runs. For example, in the Mentor Graphics ModelSim simulator, you can use the following command to open a **wave** window to view the results of the simulation as HDL waveforms:

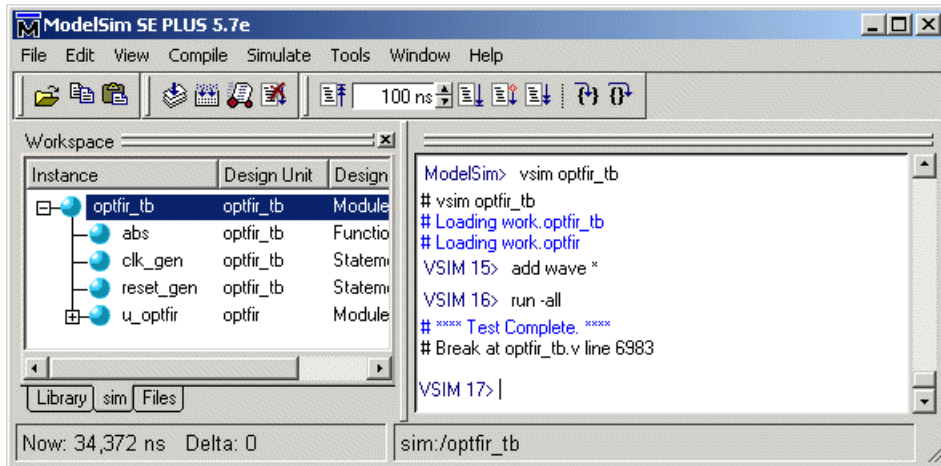
```
add wave *
```

The following **wave** window opens:



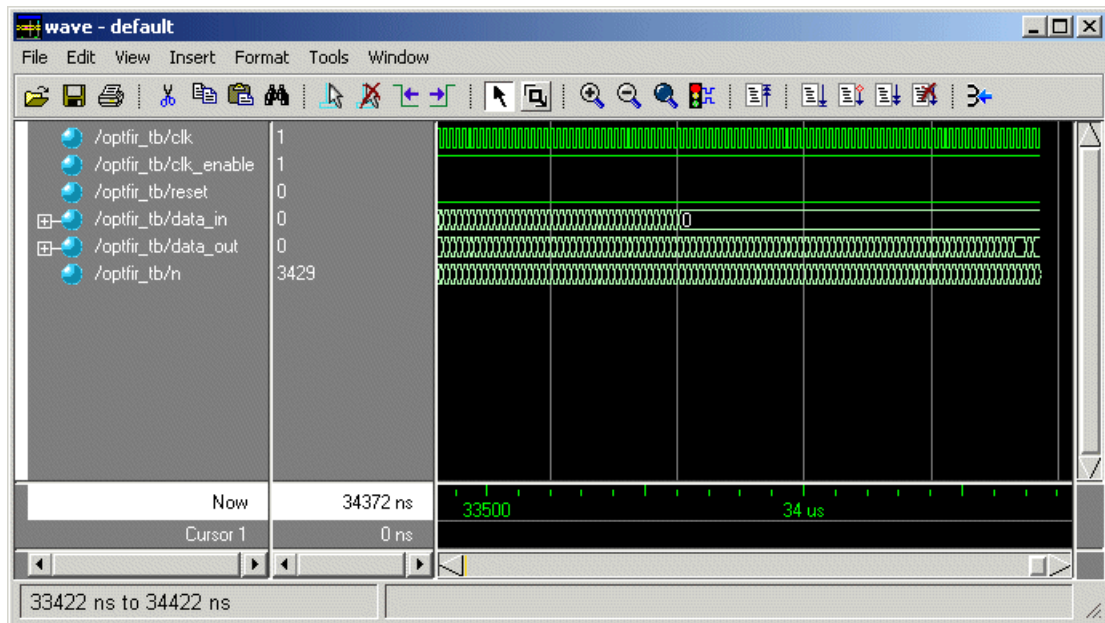
- 7 To start running the simulation, issue the start simulation command for your simulator. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the **run** command.

The following display shows the `run -all` command being used to start a simulation.



As your test bench simulation runs, watch for error messages. If error messages appear, you must interpret them as they pertain to your filter design and the HDL code generation options you selected. You must determine whether the results are expected based on the customizations you specified when generating the filter Verilog code.

The following **wave** window shows the simulation results as HDL waveforms.



IIR Filter

- “Designing an IIR Filter in FDATool” on page 1-45
- “Quantizing the IIR Filter” on page 1-47
- “Configuring and Generating the IIR Filter's VHDL Code” on page 1-50
- “Getting Familiar with the IIR Filter's Generated VHDL Code” on page 1-56
- “Verifying the IIR Filter's Generated VHDL Code” on page 1-57

Designing an IIR Filter in FDATool

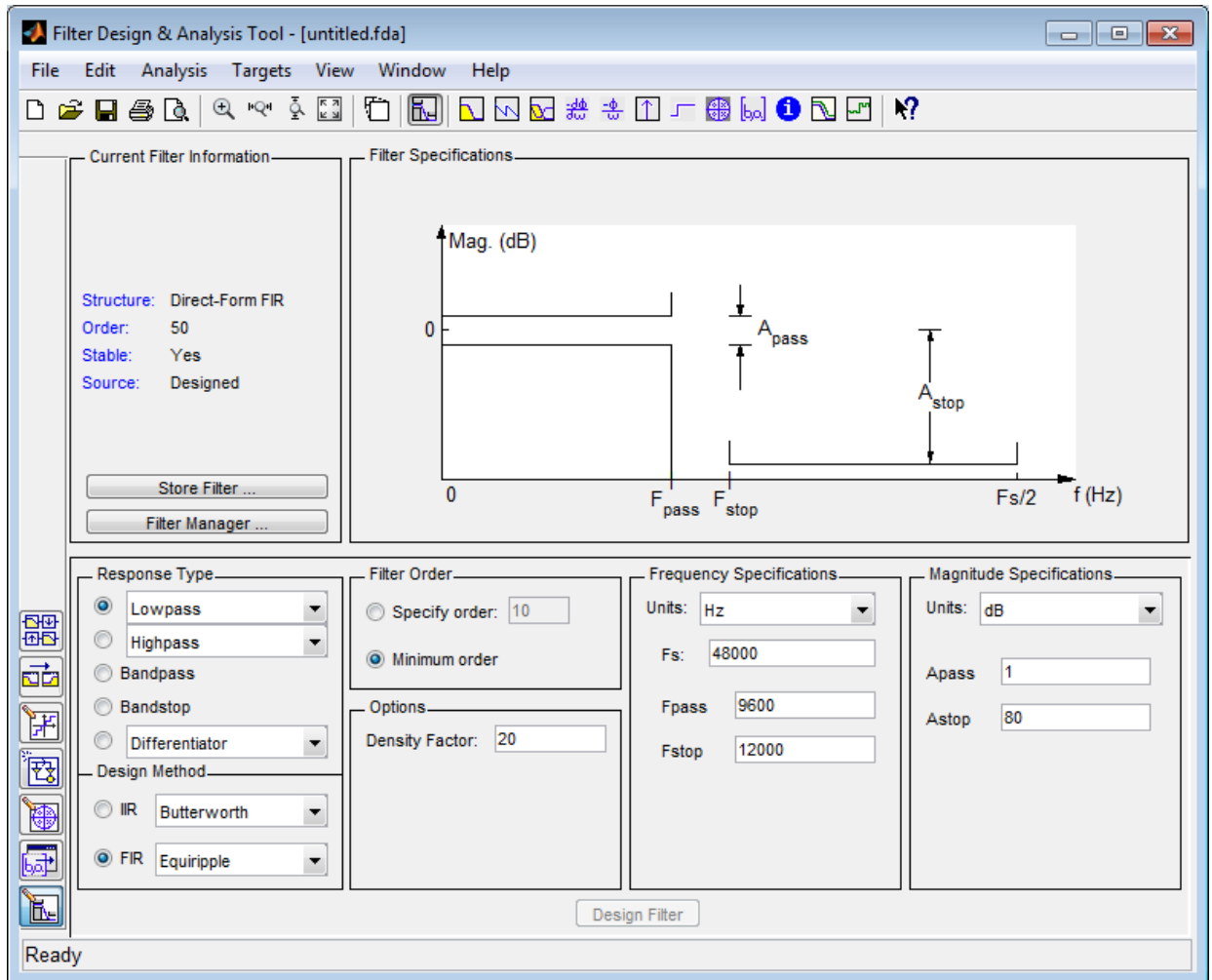
This tutorial guides you through the steps for designing an IIR filter, generating Verilog code for the filter, and verifying the Verilog code with a generated test bench.

This section guides you through the procedure of designing and creating a filter for an IIR filter. This section assumes you are familiar with the MATLAB user interface and the Filter Design & Analysis Tool (FDATool).

- 1 Start the MATLAB software.

1 Getting Started

- 2 Set your current folder to the folder you created in “Creating a Folder for Your Tutorial Files” on page 1-6.
- 3 Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



- 4 In the Filter Design & Analysis Tool dialog box, set the following filter options:

Option	Value
Response Type	Highpass
Design Method	IIR Butterworth
Filter Order	Specify order: 5
Frequency Specifications	Units: Hz
	F_s: 48000
	F_c: 10800


- 5 Click **Design Filter**. The FDATool creates a filter for the specified design. The following message appears in the FDATool status bar when the task is complete.

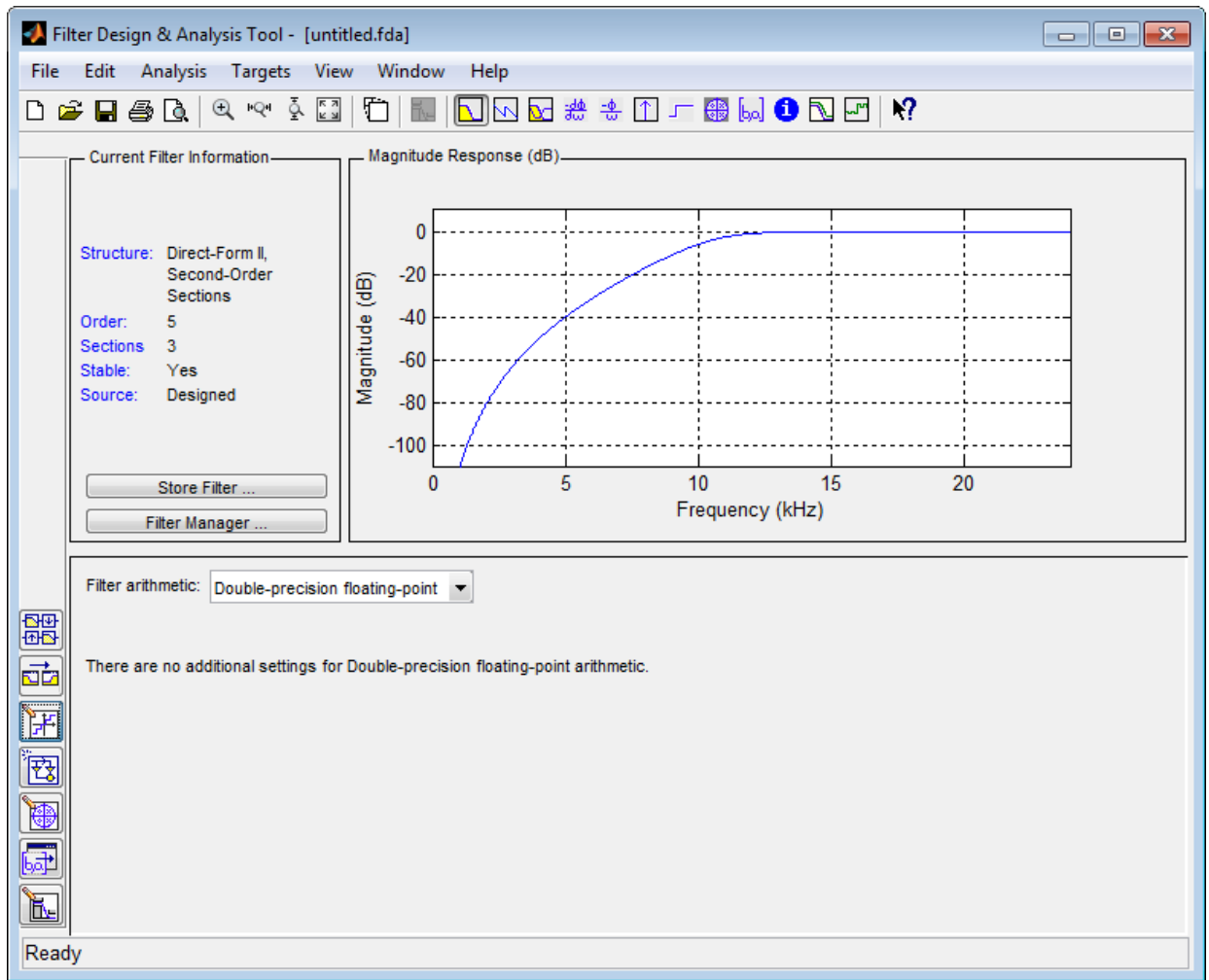
Designing Filter... Done

For more information on designing filters with the FDATool, see “Use FDATool with DSP System Toolbox Software” in the DSP System Toolbox documentation.

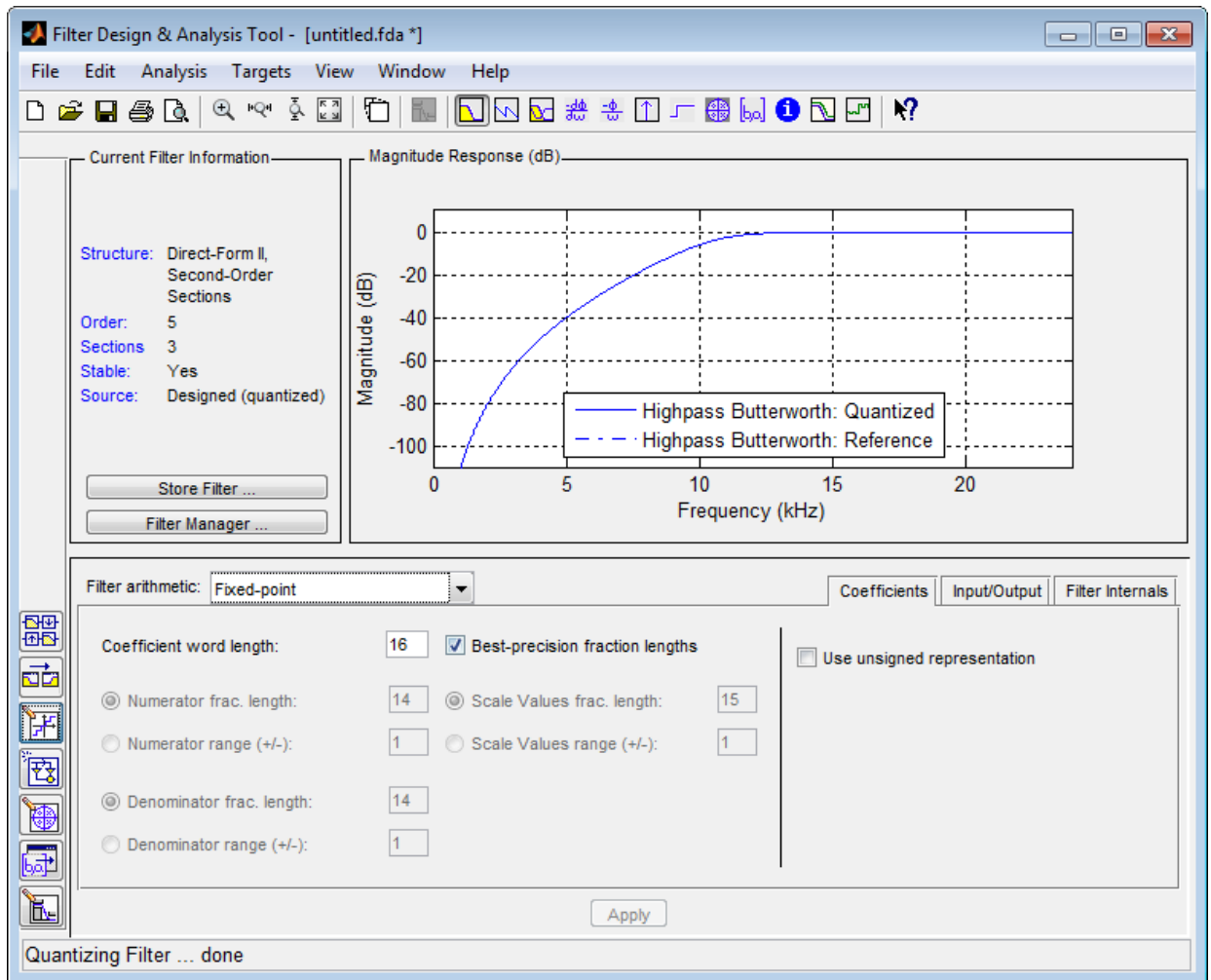
Quantizing the IIR Filter

You should quantize filters for HDL code generation. To quantize your filter,

- 1 Open the IIR filter design you created in “Designing an IIR Filter in FDATool” on page 1-45 if it is not already open.
- 2 Click the Set Quantization Parameters button  in the left-side toolbar. The FDATool displays the **Filter arithmetic** list in the bottom half of its dialog box.

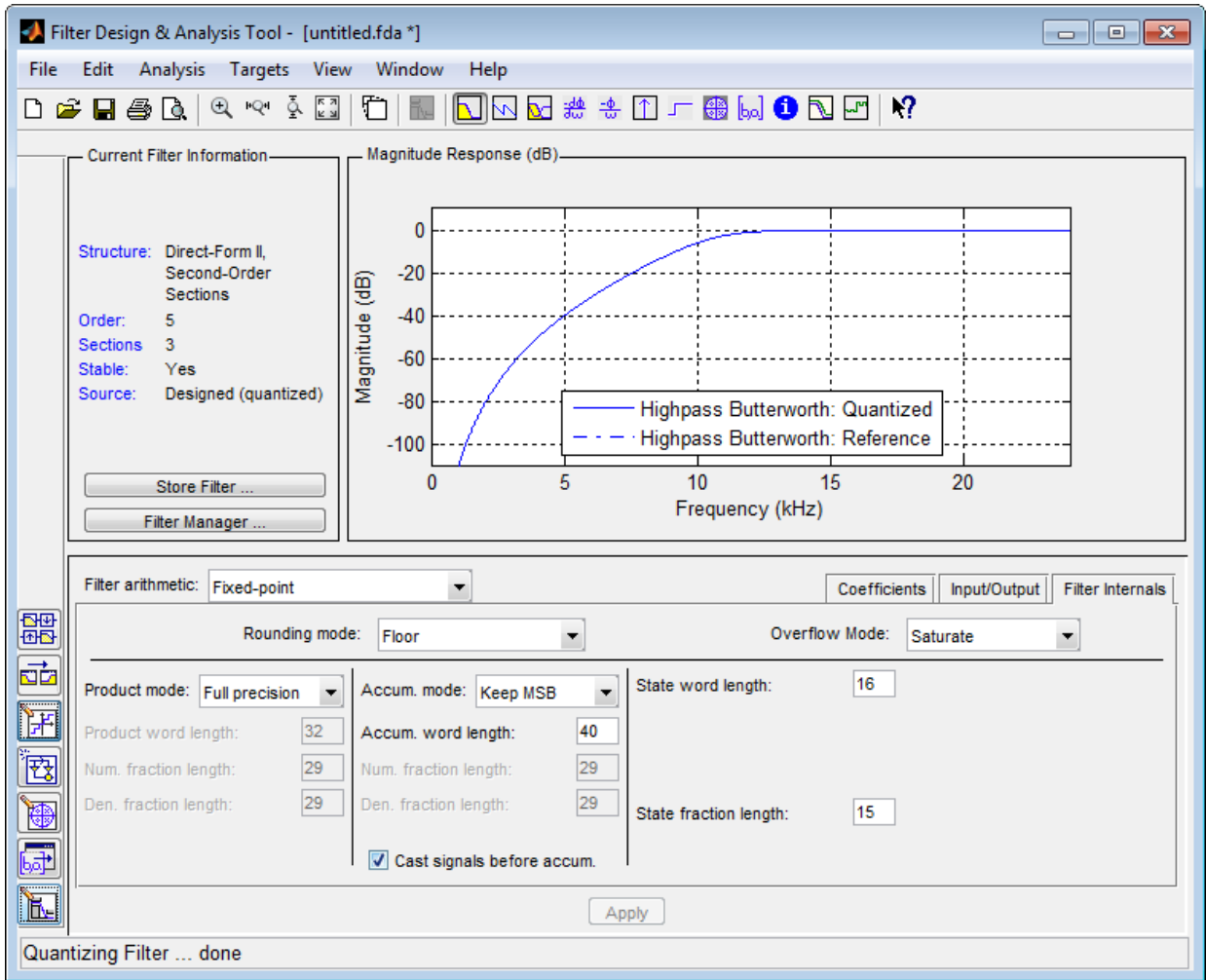


- 3 Select **Fixed-point** from the list. The FDATool displays the first of three tabbed panels of its dialog box.



You use the quantization options to test the effects of various settings with a goal of optimizing the quantized filter's performance and accuracy.

- 4 Select the **Filter Internals** tab and set **Rounding mode** to **Floor** and **Overflow Mode** to **Saturate**.
- 5 Click **Apply**. The quantized filter appears as follows.



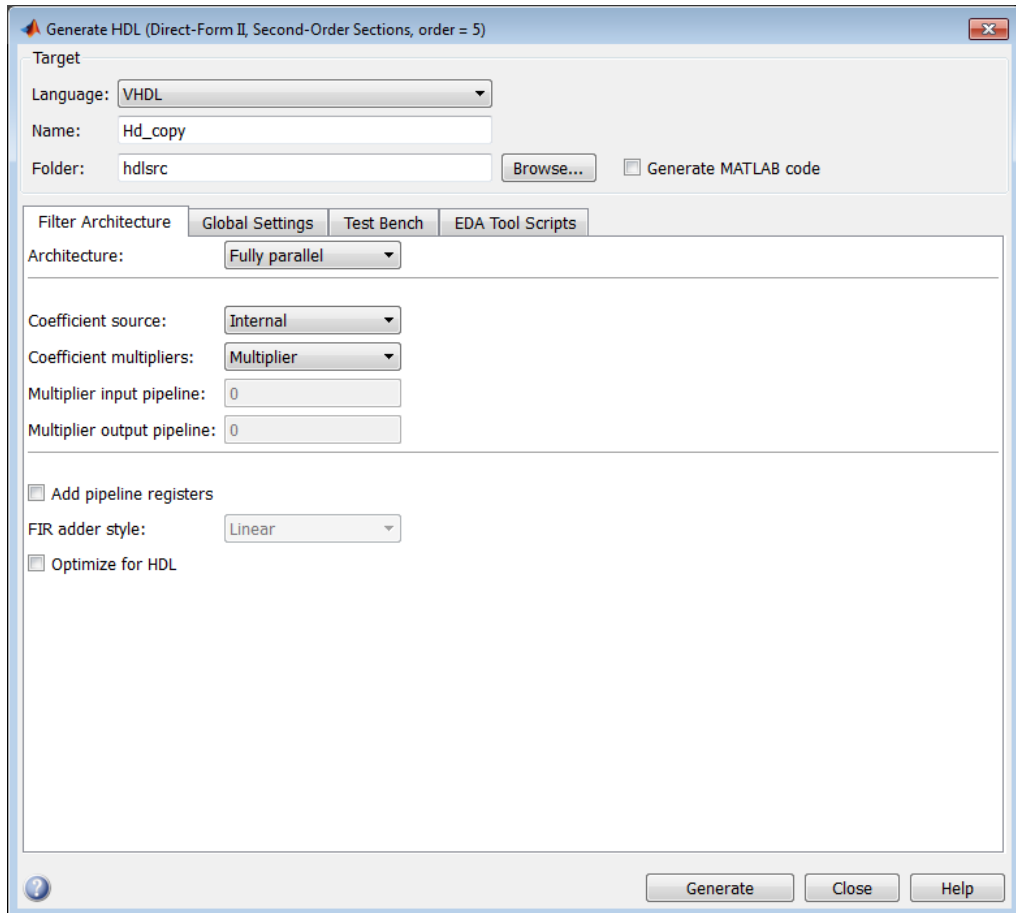
For more information on quantizing filters with the FDATool, see “Use FDATool with DSP System Toolbox Software” in the DSP System Toolbox documentation.

Configuring and Generating the IIR Filter's VHDL Code

After you quantize your filter, you are ready to configure coder options and generate the filter's VHDL code. This section guides you through the procedure for starting the

Filter Design HDL Coder GUI, setting some options, and generating the VHDL code and a test bench for the IIR filter you designed and quantized in “Designing an IIR Filter in FDATool” on page 1-45 and “Quantizing the IIR Filter” on page 1-47:

- 1 Start the Filter Design HDL Coder GUI by selecting **Targets > Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog box.



- 2 In the **Name** text box of the **Target** pane, type `iir`. This option names the VHDL entity and the file that will contain the filter's VHDL code.
- 3 Select the **Global settings** tab of the GUI. Then select the **General** tab of the **Additional settings** section.

In the **Comment in header** text box, type `Tutorial - IIR Filter`. The coder adds the comment to the end of the header comment block in each generated file.

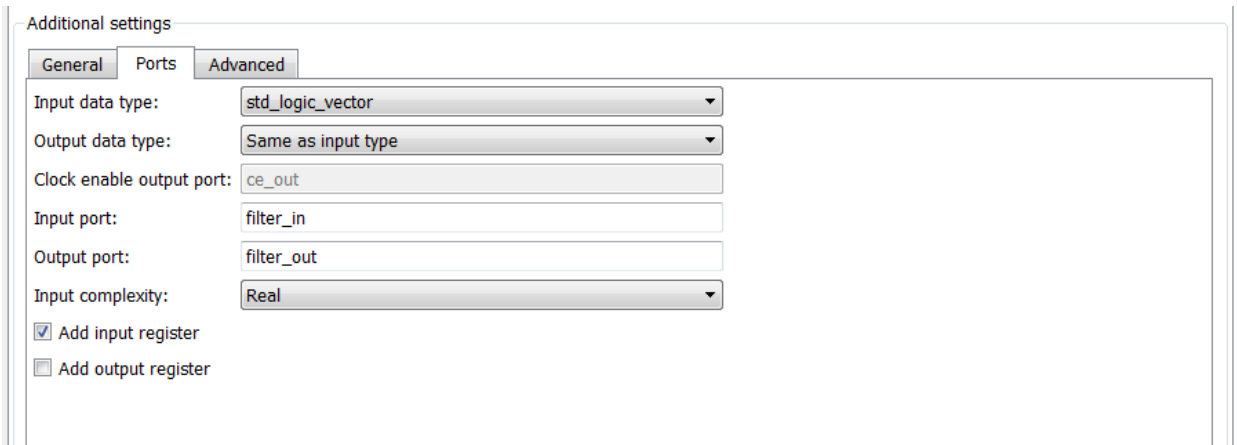
- 4 Select the **Ports** tab. The **Ports** pane appears.



The screenshot shows the 'Additional settings' pane with the 'Ports' tab selected. The settings are as follows:

- Input data type: `std_logic_vector`
- Output data type: `Same as input type`
- Clock enable output port: `ce_out`
- Input port: `filter_in`
- Output port: `filter_out`
- Input complexity: `Real`
- Add input register
- Add output register

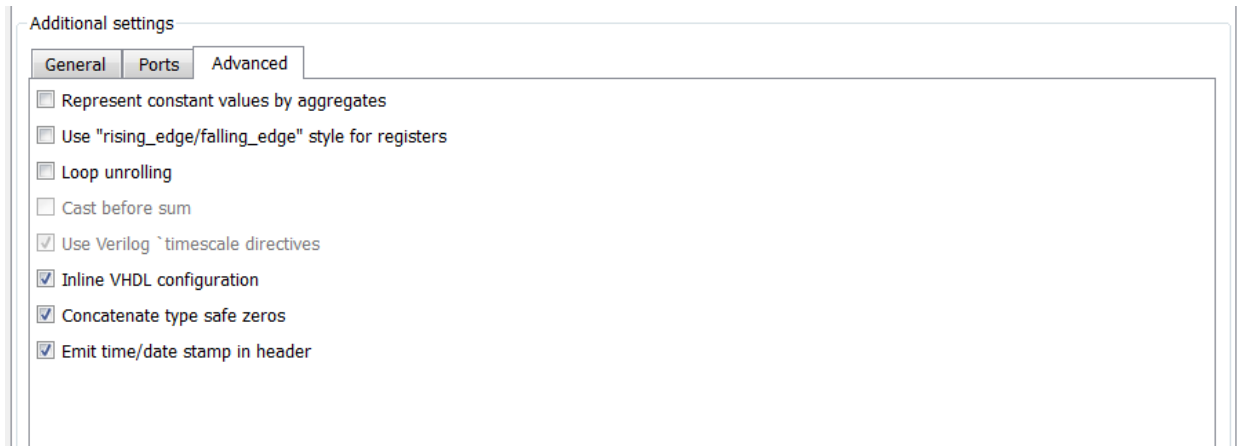
- 5 Clear the check box for the **Add output register** option. The **Ports** pane should now appear as in the following figure.



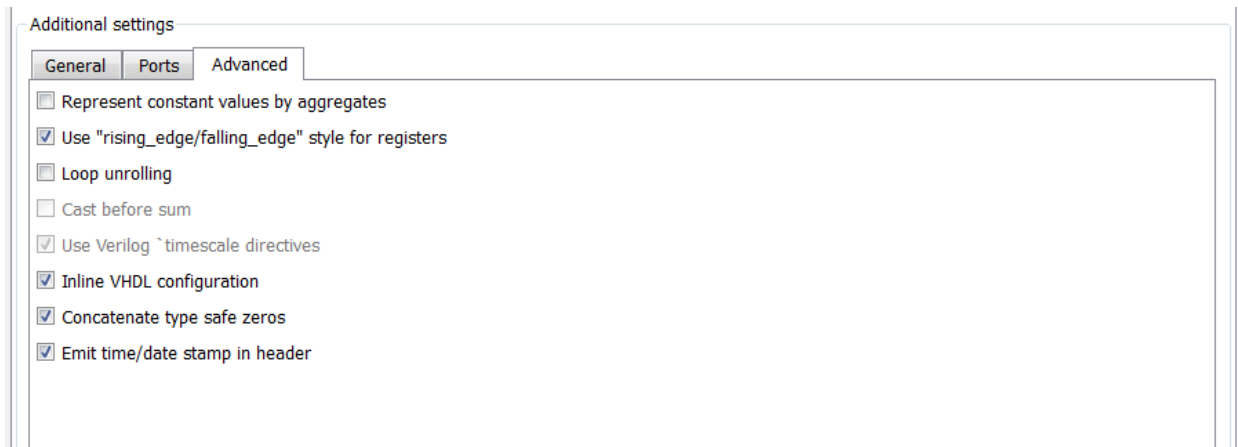
The screenshot shows the 'Additional settings' pane with the 'Ports' tab selected. The settings are as follows:

- Input data type: `std_logic_vector`
- Output data type: `Same as input type`
- Clock enable output port: `ce_out`
- Input port: `filter_in`
- Output port: `filter_out`
- Input complexity: `Real`
- Add input register
- Add output register

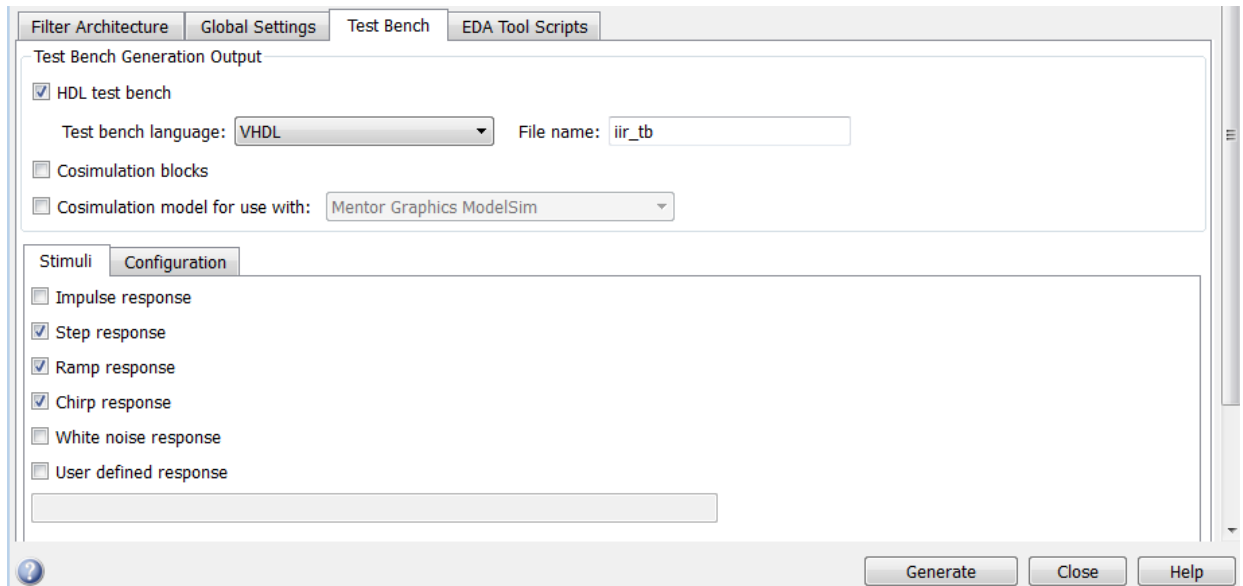
- 6 Select the **Advanced** tab. The **Advanced** pane appears.



- 7 Select the **Use 'rising_edge' for registers** option. The **Advanced** pane should now appear as in the following figure.



- 8 Click on the **Test bench** tab in the Generate HDL dialog box. In the **File name** text box, replace the default name with `iir_tb`. This option names the generated test bench file.



- 9 In the Generate HDL dialog box, click **Generate** to start the code generation process. When code generation completes, click **OK** to close the dialog box.

The coder displays the following messages in the MATLAB Command Window as it generates the filter and test bench VHDL files:

```

### Starting VHDL code generation process for filter: iir
### Starting VHDL code generation process for filter: iir
### Generating: H:\hdlsrc\iir.vhd
### Starting generation of iir VHDL entity
### Starting generation of iir VHDL architecture
### Second-order section, # 1
### Second-order section, # 2
### First-order section, # 3
### HDL latency is 1 samples
### Successful completion of VHDL code generation process for filter: iir

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 2172 samples.
### Generating Test bench: H:\hdlsrc\filter_tb.vhd
### Please wait ...
### Done generating VHDL Test Bench
### Starting VHDL code generation process for filter: iir
### Starting VHDL code generation process for filter: iir
### Generating: H:\hdlsrc\iir.vhd
### Starting generation of iir VHDL entity
### Starting generation of iir VHDL architecture
### Second-order section, # 1
    
```



```

### Second-order section, # 2
### First-order section, # 3
### HDL latency is 1 samples
### Successful completion of VHDL code generation process for filter: iir

```

As the messages indicate, the coder creates the folder `hdlsrc` under your current working folder and places the files `iir.vhd` and `iir_tb.vhd` in that folder.

Observe that the messages include hyperlinks to the generated code and test bench files. By clicking on these hyperlinks, you can open the code files directly into the MATLAB Editor.

The generated VHDL code has the following characteristics:

- VHDL entity named `iir`.
- Registers that use asynchronous resets when the reset signal is active high (1).
- Ports have the following default names:

VHDL Port	Name
Input	<code>filter_in</code>
Output	<code>filter_out</code>
Clock input	<code>clk</code>
Clock enable input	<code>clk_enable</code>
Reset input	<code>reset</code>

- An extra register for handling filter input.
- Clock input, clock enable input and reset ports are of type `STD_LOGIC` and data input and output ports are of type `STD_LOGIC_VECTOR`.
- Coefficients are named `coeffn`, where n is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'...`
- Registers are generated with the `rising_edge` function rather than the statement `ELSIF clk'event AND clk='1' THEN`.
- The postfix string `_process` is appended to process names.

The generated test bench:

- Is a portable VHDL file.

- Forces clock, clock enable, and reset input signals.
- Forces the clock enable input signal to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.
- Applies a hold time of 2 nanoseconds to data input signals.
- For an IIR filter, applies impulse, step, ramp, chirp, and white noise stimulus types.

Getting Familiar with the IIR Filter's Generated VHDL Code

Get familiar with the filter's generated VHDL code by opening and browsing through the file `iir.vhd` in an ASCII or HDL simulator editor:

- 1 Open the generated VHDL filter file `iir.vhd`.
- 2 Search for `iir`. This line identifies the VHDL module, using the string you specified for the **Name** option in the **Target** pane. See step 2 in “Configuring and Generating the IIR Filter's VHDL Code” on page 1-50.
- 3 Search for `Tutorial`. This is where the coder places the text you entered for the **Comment in header** option. See step 5 in “Configuring and Generating the IIR Filter's VHDL Code” on page 1-50.
- 4 Search for `HDL Code`. This section lists coder options you modified in “Configuring and Generating the IIR Filter's VHDL Code” on page 1-50.
- 5 Search for `Filter Settings`. This section of the VHDL code describes the filter design and quantization settings as you specified in “Designing an IIR Filter in FDATool” on page 1-45 and “Quantizing the IIR Filter” on page 1-47.
- 6 Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **Target** pane. See step 2 in “Configuring and Generating the IIR Filter's VHDL Code” on page 1-50.
- 7 Search for `PORT`. This `PORT` declaration defines the filter's clock, clock enable, reset, and data input and output ports. The ports for clock, clock enable, reset, and data input and output signals are named with default strings.
- 8 Search for `CONSTANT`. This is where the coefficients are defined. They are named using the default naming scheme, `coeff_xm_sectionn`, where *x* is `a` or `b`, *m* is the coefficient number, and *n* is the section number.
- 9 Search for `SIGNAL`. This is where the filter's signals are defined.

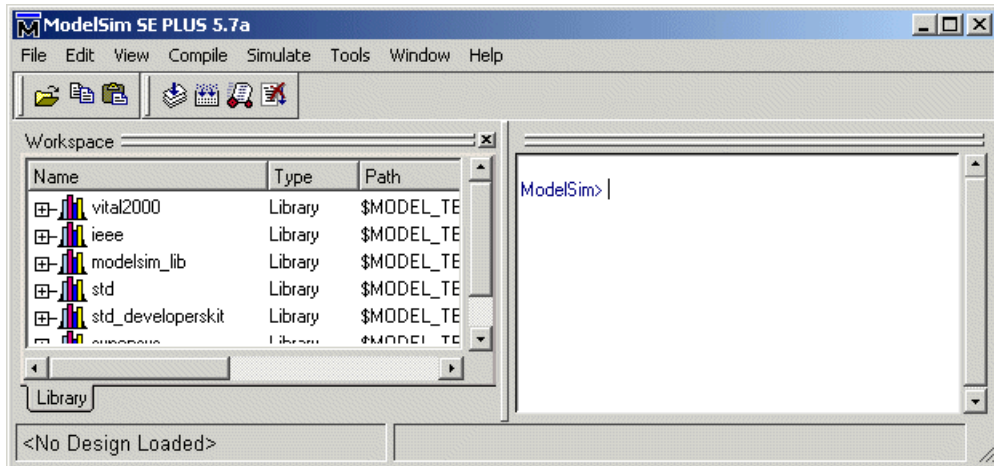
- 10** Search for `input_reg_process`. The `PROCESS` block name `input_reg_process` includes the default `PROCESS` block postfix string `_process`. This is where filter input is read from an input register. Code for this register is generated by default. In step 7 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11, you cleared the **Add output register** option, but left the **Add input register** option selected.
- 11** Search for `IF reset`. This is where the reset signal is asserted. The default, active high (1), was specified. Also note that the `PROCESS` block applies the default asynchronous reset style when generating VHDL code for registers.
- 12** Search for `ELSIF`. This is where the VHDL code checks for rising edges when the filter operates on registers. The `rising_edge` function is used as you specified in the **Advanced** pane of the Generate HDL dialog box. See step 10 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 1-11.
- 13** Search for `Section 1`. This is where second-order section 1 data is filtered. Similar sections of VHDL code apply to another second-order section and a first-order section.
- 14** Search for `filter_out`. This is where the filter writes its output data.

Verifying the IIR Filter's Generated VHDL Code

This sections explains how to verify the IIR filter's generated VHDL code with the generated VHDL test bench. Although this tutorial uses the Mentor Graphics ModelSim simulator as the tool for compiling and simulating the VHDL code, you can use another HDL simulation tool package.

To verify the filter code, complete the following steps:

- 1** Start your simulator. When you start the Mentor Graphics ModelSim simulator, a screen display similar to the following appears.



- 2 Set the current folder to the folder that contains your generated VHDL files. For example:

```
cd hdlsrc
```

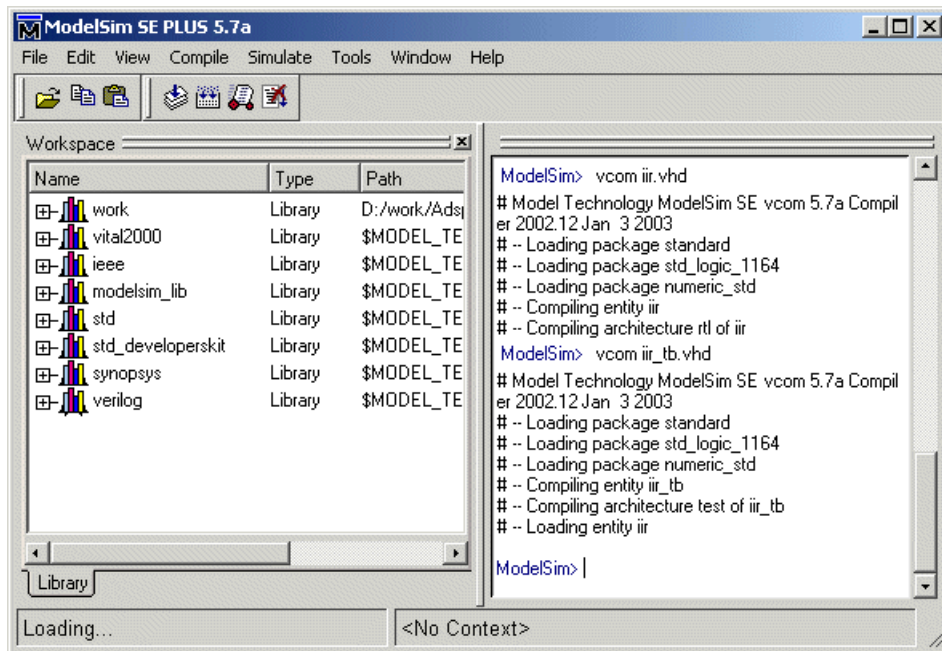
- 3 If desired, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

- 4 Compile the generated filter and test bench VHDL files. In the Mentor Graphics ModelSim simulator, you compile VHDL code with the `vcom` command. The following the commands compile the filter and filter test bench VHDL code.

```
vcom iir.vhd  
vcom iir_tb.vhd
```

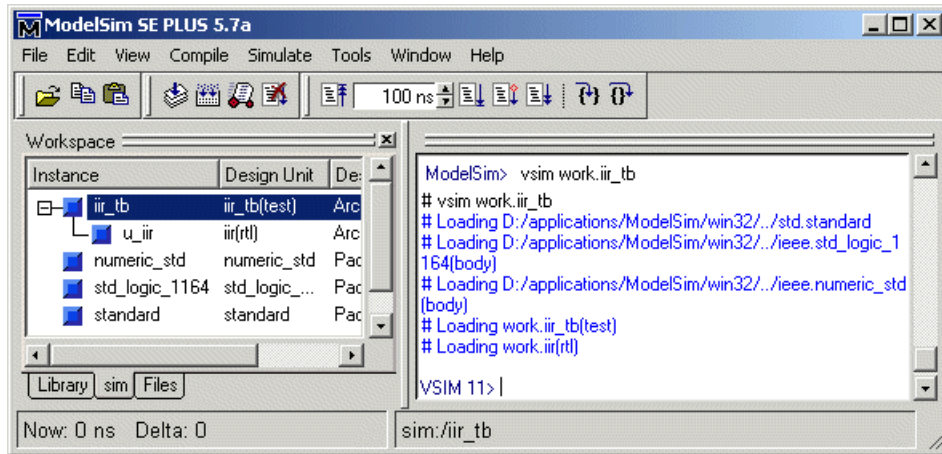
The following screen display shows this command sequence and informational messages displayed during compilation.



- 5 Load the test bench for simulation. The procedure for doing this varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.iir_tb
```

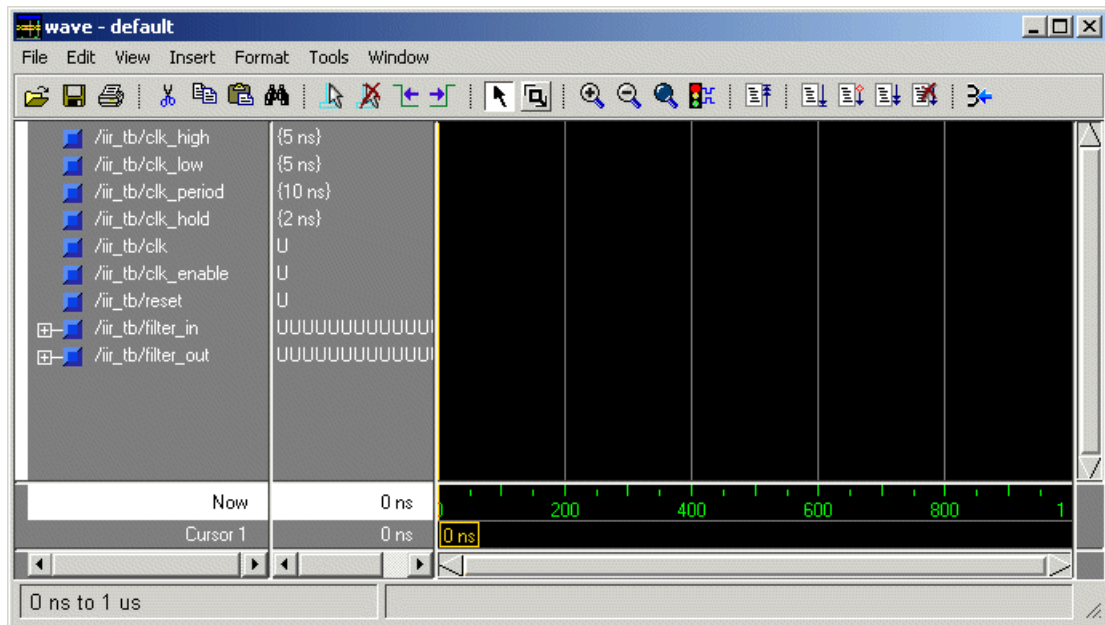
The following display shows the results of loading `work.iir_tb` with the `vsim` command.



- 6 Open a display window for monitoring the simulation as the test bench runs. For example, in the Mentor Graphics ModelSim simulator, you can use the following command to open a **wave** window to view the results of the simulation as HDL waveforms.

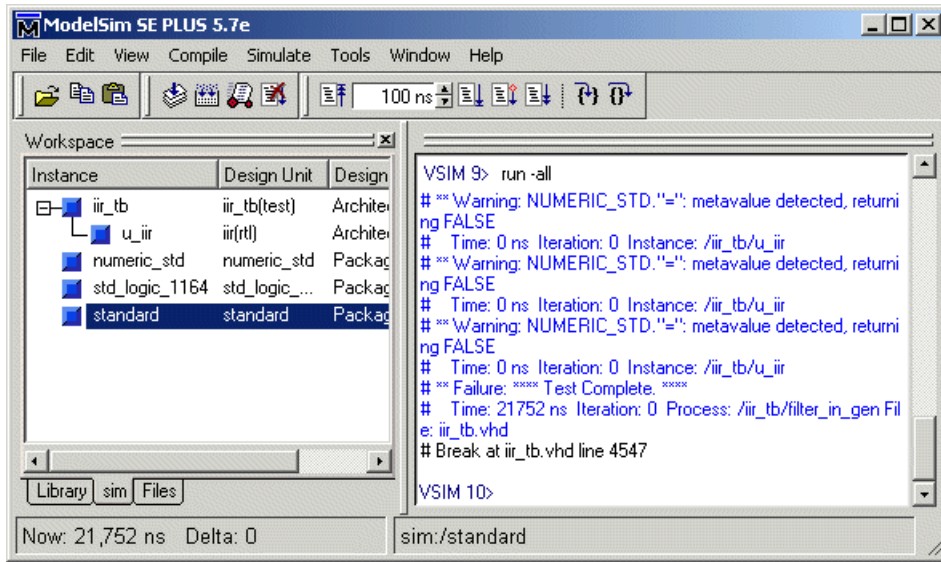
```
add wave *
```

The following **wave** window displays.



- 7 To start running the simulation, issue the start simulation command for your simulator. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the `run` command.

The following display shows the `run -all` command being used to start a simulation.

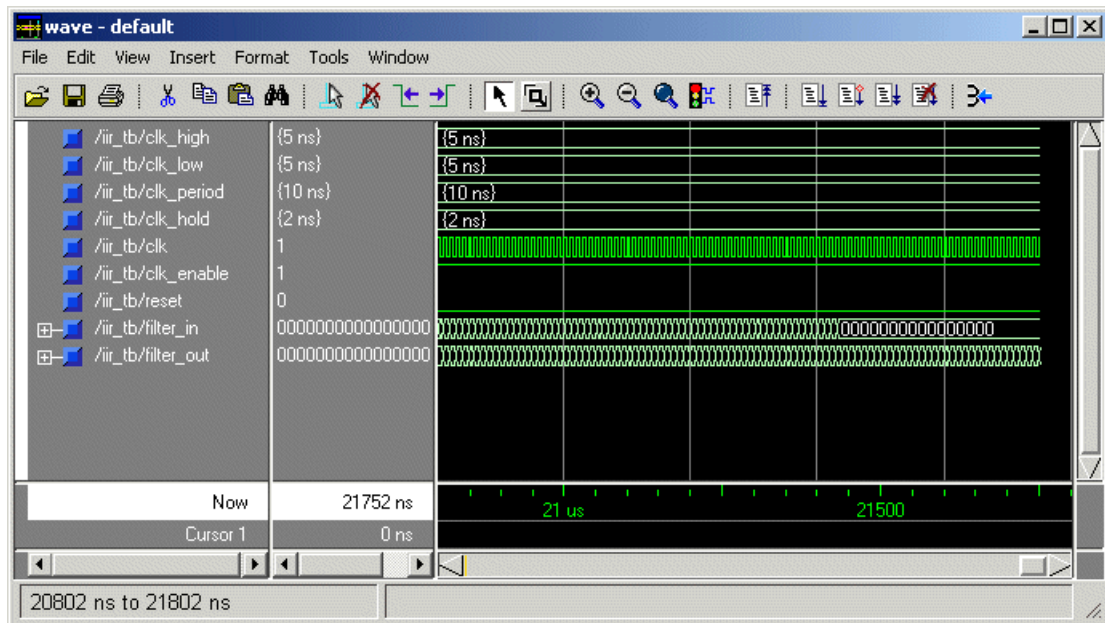


As your test bench simulation runs, watch for error messages. If error messages appear, you must interpret them as they pertain to your filter design and the HDL code generation options you selected. You must determine whether the results are expected based on the customizations you specified when generating the filter VHDL code.

Note:

- The warning messages that note **Time: 0 ns** in the preceding display are not errors and you can ignore them.
- The failure message that appears in the preceding display is not flagging an error. If the message includes the string **Test Complete**, the test bench has run to completion without encountering an error. The **Failure** part of the message is tied to the mechanism that the coder uses to end the simulation.

The following **wave** window shows the simulation results as HDL waveforms.



HDL Filter Code Generation Fundamentals

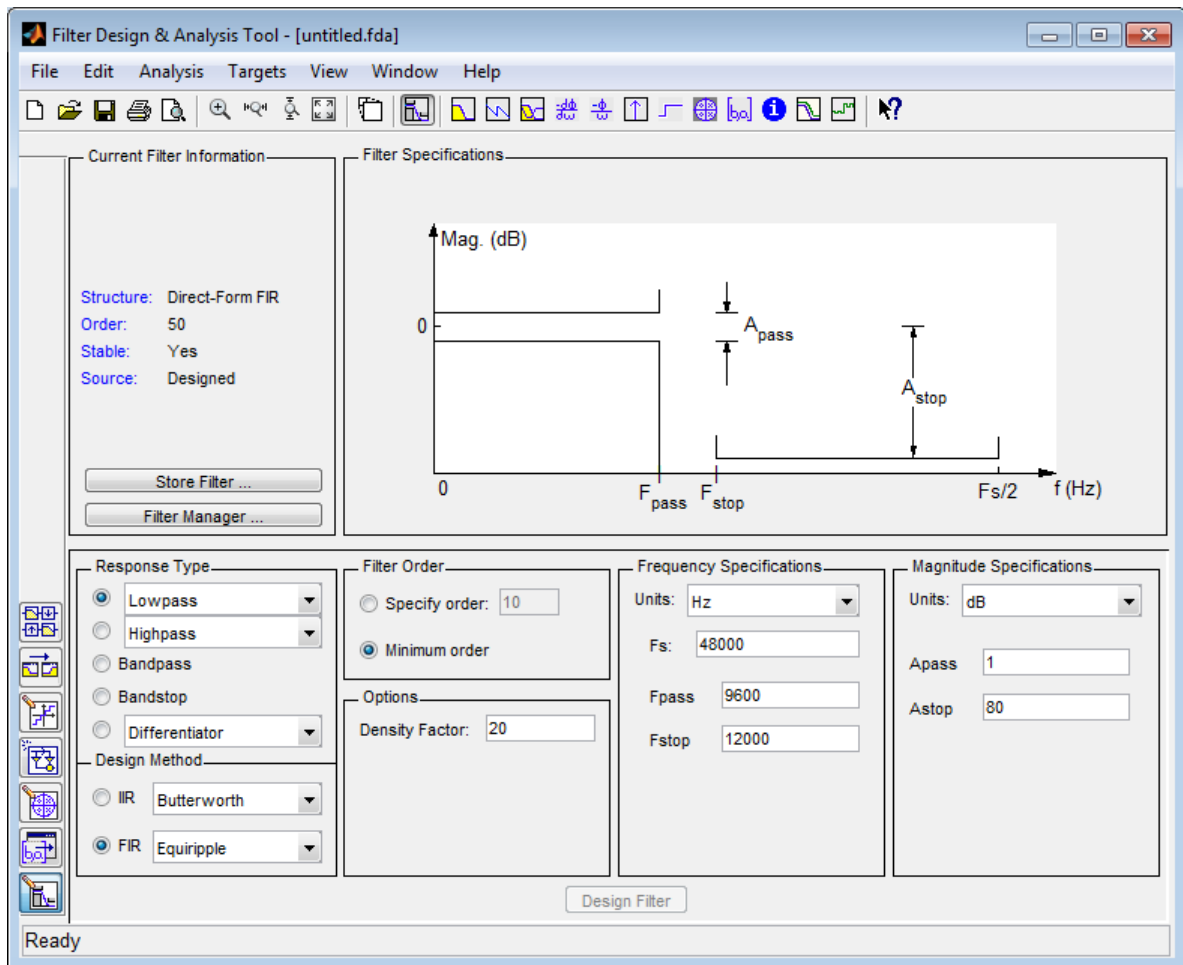
- “Starting Filter Design HDL Coder” on page 2-2
- “Selecting Target Language” on page 2-12
- “Generating HDL Code” on page 2-13
- “Capturing Code Generation Settings” on page 2-15
- “Closing Code Generation Session” on page 2-16

Starting Filter Design HDL Coder


Opening the Filter Design HDL Coder GUI From FDATool

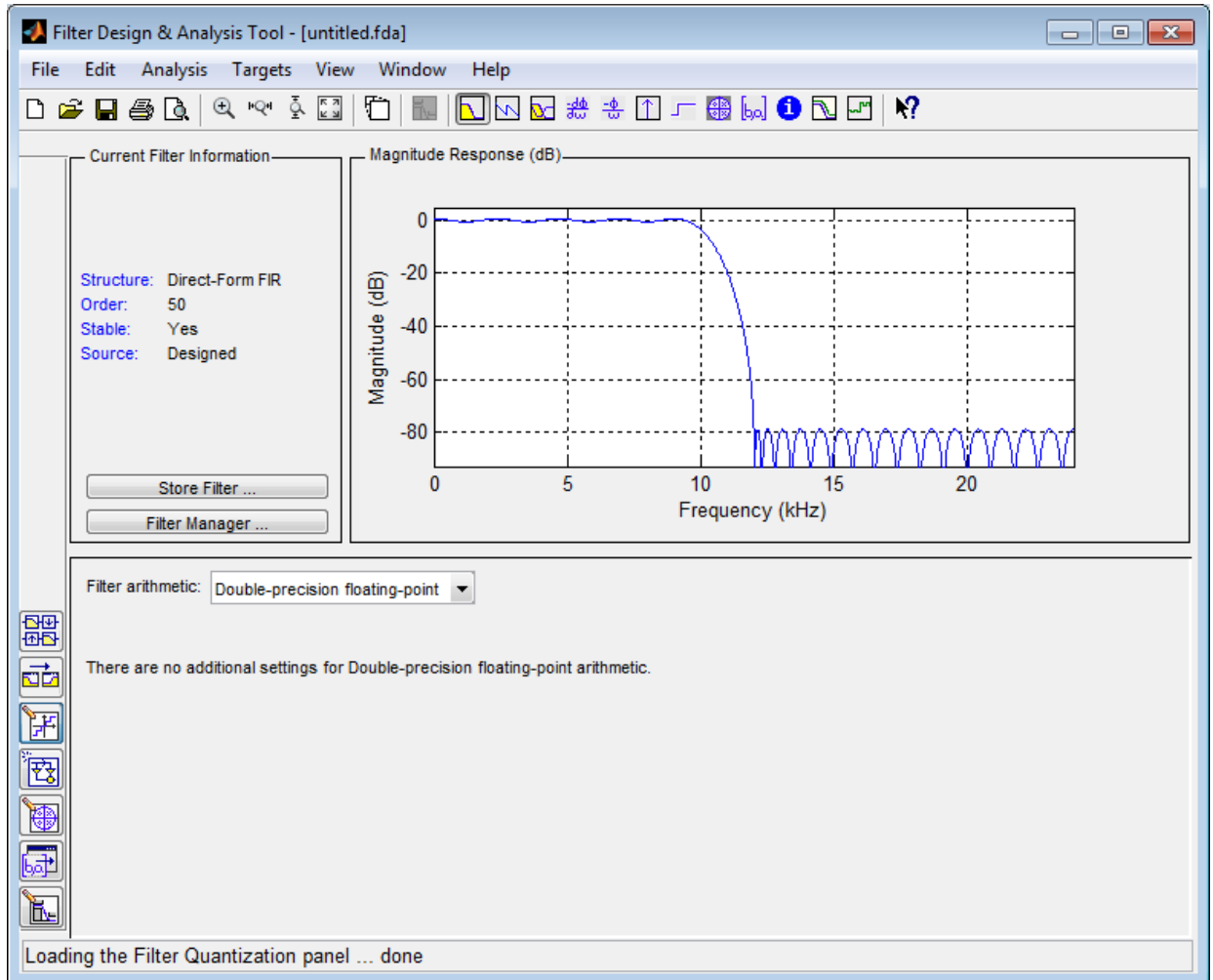
To open the initial Generate HDL dialog box from FDATool, do the following:

- 1 Enter the `fdatool` command at the MATLAB command prompt. The FDATool displays its initial dialog box.



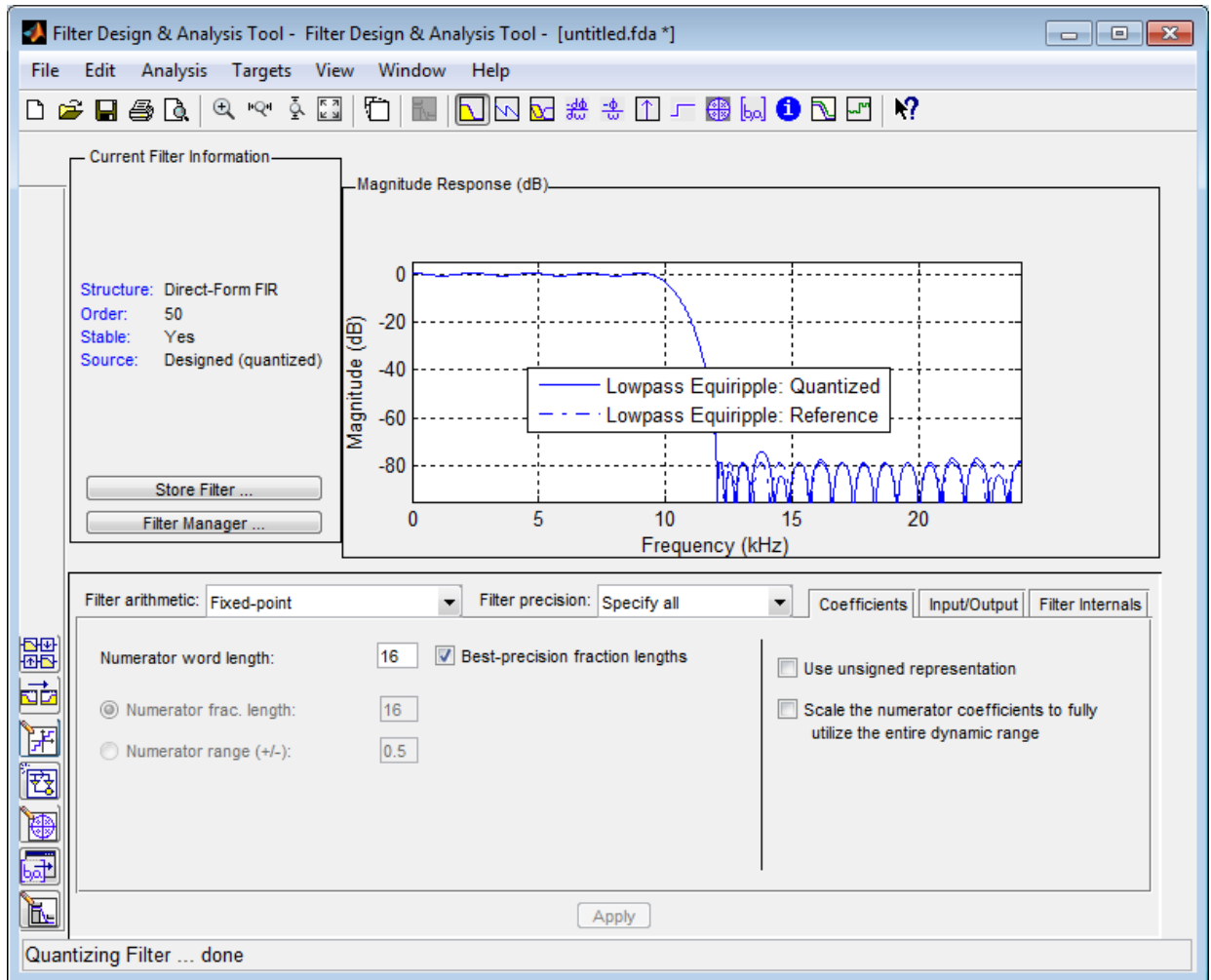
- 2 If the filter design is quantized, skip to step 3. Otherwise, quantize the filter by

clicking the **Set Quantization Parameters** button . The **Filter arithmetic** menu appears in the bottom half of the dialog box.

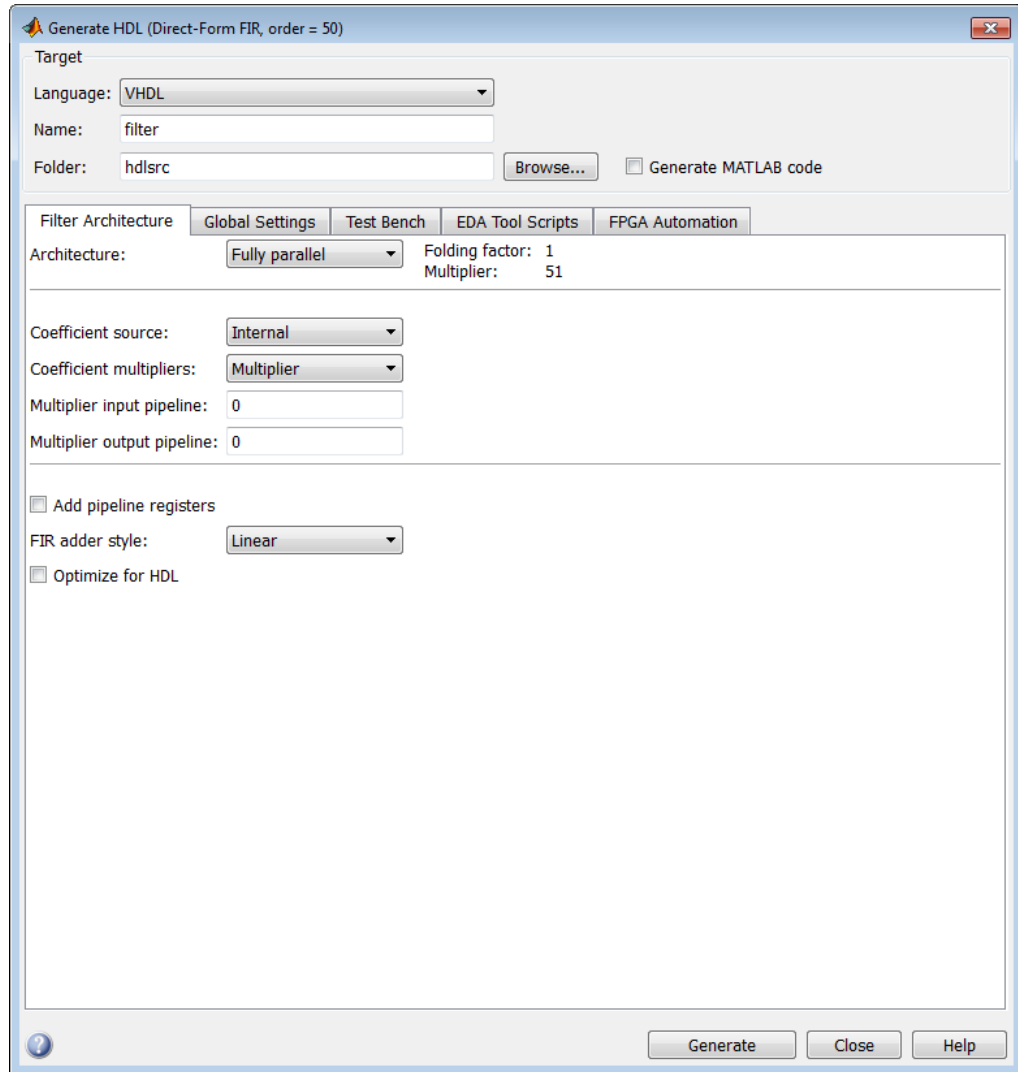


Note: Supported filter structures allow both fixed-point and floating-point (double) realizations.

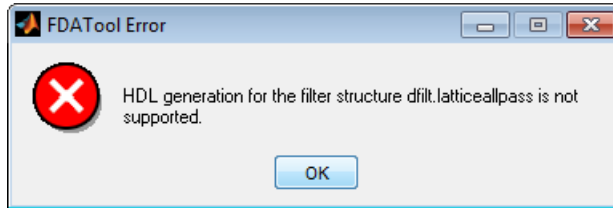
- 3 If desired, adjust the setting of the **Filter arithmetic** option. The FDATool displays the first of three tabbed panes of its dialog.



- 4 Select **Targets > Generate HDL**. The FDATool displays the Generate HDL dialog box.



If the coder does not support the structure of the current filter in the FDATool, an error message appears. For example, if the current filter is a quantized, lattice-coupled, allpass filter, the following message appears.



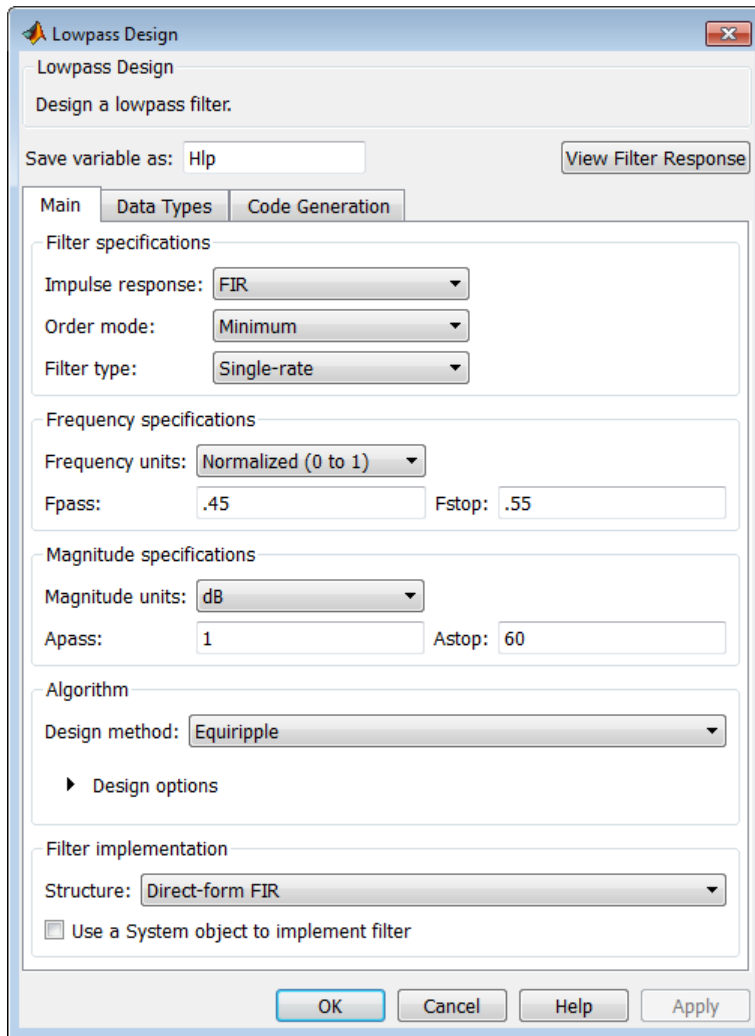
Opening the Filter Design HDL Coder GUI From the filterbuilder GUI

If you are not familiar with the `filterbuilder` GUI, see the DSP System Toolbox documentation.

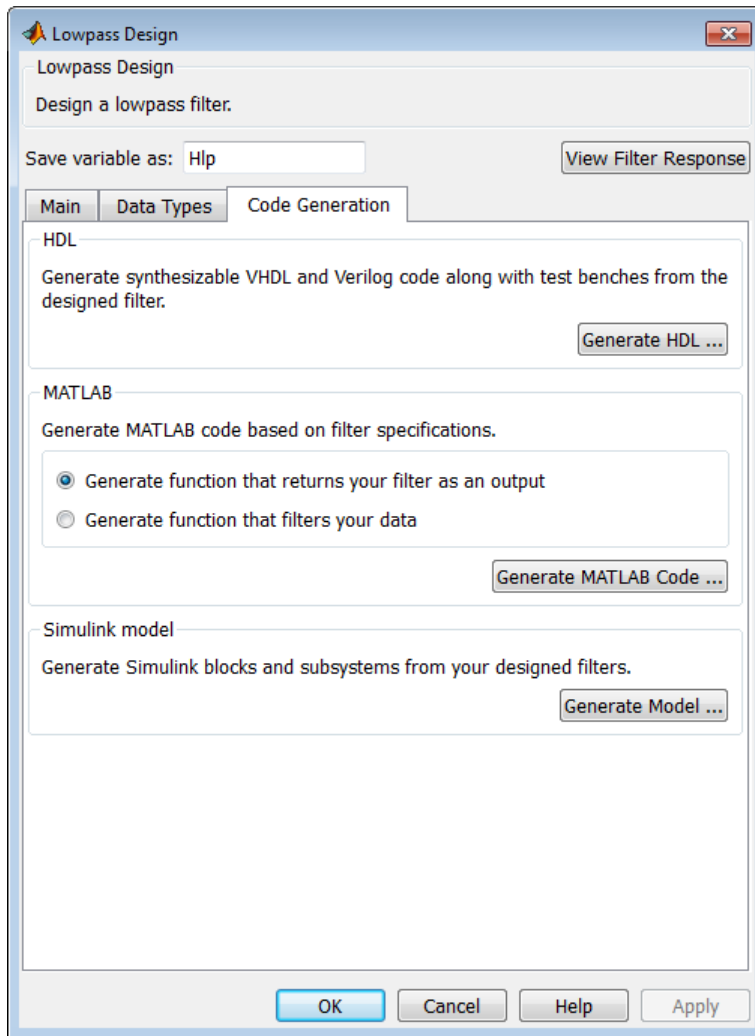
To open the initial Generate HDL dialog box from the `filterbuilder` GUI, do the following:

- 1 At the MATLAB command prompt, type a `filterbuilder` command that corresponds to the filter response or filter object you want to design.

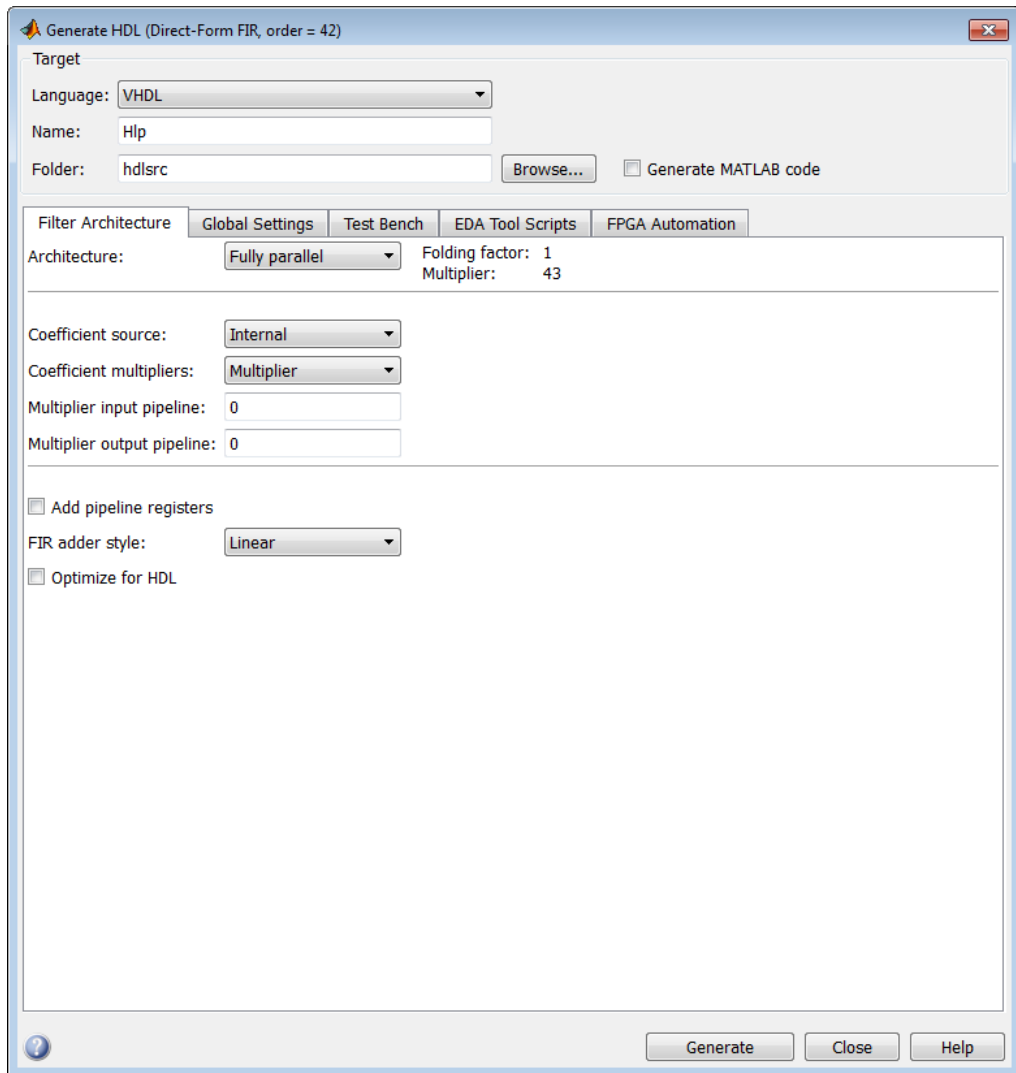
The following figure shows the default settings of the main pane of the `filterbuilder` Lowpass Filter Design dialog box.



- 2 Set the filter design parameters as required.
- 3 Click the **Code Generation** tab. This activates the **Code Generation** pane, shown in the following figure.



- 4 In the **Code Generation** pane, click the **Generate HDL** button. This opens the Generate HDL dialog box, passing in the current filter object from `filterbuilder`.



- 5 Set the desired code generation and test bench options and generate code in the Generate HDL dialog box.

Opening the Filter Design HDL Coder GUI Using the `fdhdltool` Command

Opening the Generate HDL Dialog Box Using the `fdhdltool` Command

You can use the `fdhdltool` command to open the Generate HDL dialog box directly from the MATLAB command line. The syntax is:

```
fdhdltool(Hd)
```

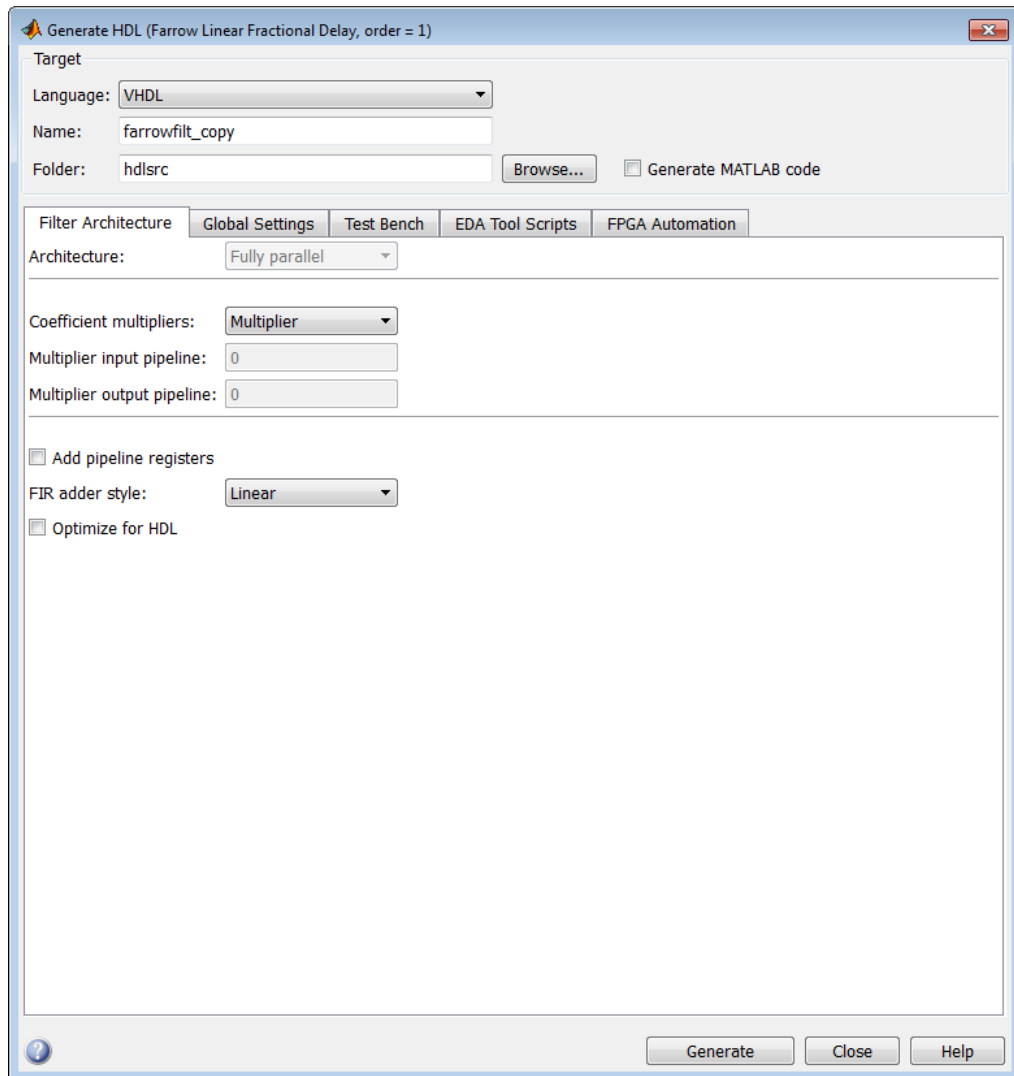
where `Hd` is a type of filter object that is supported for HDL code generation.

The `fdhdltool` function is particularly useful when you must use the Filter Design HDL Coder GUI to generate HDL code for filter structures that are not supported by `FDATool` or `filterbuilder`. For example, the following commands create a Farrow linear fractional delay filter object `Hd`, which is passed in to the `fdhdltool` function:

```
D = .3  
farrowfilt = dfilt.farrowlinearfd(D)  
fdhdltool(farrowfilt)
```

`fdhdltool` operates on a copy of the filter object, rather than the original object in the MATLAB workspace. Changes made to the original filter object after invoking `fdhdltool` do not apply to the copy and do not update the Generate HDL dialog box.

The naming convention for the copied object is `filt_copy`, where `filt` is the name of the original filter object. This naming convention is reflected in the filter **Name** and test bench **File name** fields, as shown in the following figure.



Selecting Target Language

HDL code is generated in either VHDL or Verilog. The language you choose for code generation is called the *target language*. By default, the target language is VHDL. If you retain the VHDL setting, Generate HDL dialog box options that are specific to Verilog are disabled and are not selectable.

If you require or prefer to generate Verilog code, select **Verilog** for the **Language** option in the **Target** pane of the Generate HDL dialog box. This setting causes the coder to enable options that are specific to Verilog and to gray out and disable options that are specific to VHDL.

Command Line Alternative: Use the `generatehdl` function with the `TargetLanguage` property to set the language to VHDL or Verilog.

Generating HDL Code

In this section...

“Applying Your Settings” on page 2-13

“Generating HDL Code from the GUI” on page 2-13

“Generating HDL Code Using generatehdl” on page 2-14

Applying Your Settings

When you click the **Generate** button, the coder

- Applies code generation option settings that you have edited
- Generates HDL code and other requested files, such as a test bench, as described in “Generating HDL Code from the GUI” on page 2-13.

If you want to preserve your coder settings, the best practice is to select the **Generate MATLAB code** option, as described in “Capturing Code Generation Settings” on page 2-15.

Tip Generate MATLAB code is available only in the GUI. The function `generatehdl` does not have an equivalent property.

Generating HDL Code from the GUI

To initiate HDL code generation for a filter and its test bench from the GUI, click **Generate** on the Generate HDL dialog box. As code generation proceeds, a sequence of messages similar to the following appears in the MATLAB Command Window:

```
### Starting VHDL code generation process for filter: iir
### Generating: D:\hdlfilter_tutorials\hdlsrc\iir.vhd
### Starting generation of iir VHDL entity
### Starting generation of iir VHDL architecture
### First-order section, # 1
### Second-order section, # 2
### Second-order section, # 3
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: iir

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 2172 samples.
### Generating: D:\hdlfilter_tutorials\hdlsrc\iir_tb.vhd
```

```
### Please wait .....  
### Done generating VHDL test bench.
```

The messages include hyperlinks to the generated code and test bench files. Click on these hyperlinks, you can open the code files directly into the MATLAB Editor.

Generating HDL Code Using `generatehdl`

To initiate HDL code generation for a filter and its test bench from the command line, use the `generatehdl` function. When you call the `generatehdl` function, specify the filter name and (optionally) desired property name and property value pairs, as in the following example.

```
generatehdl(iir, 'TargetLanguage,' 'Verilog')
```

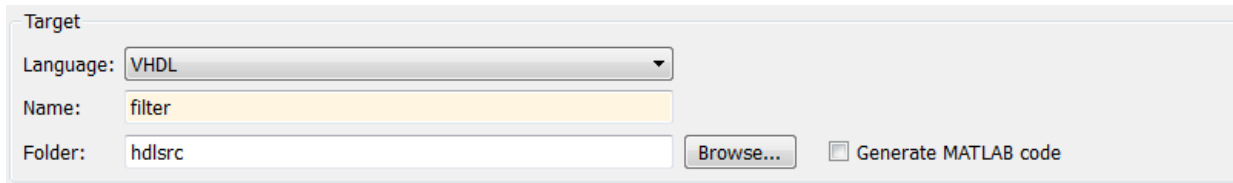
As code generation proceeds, a sequence of messages similar to the following appears in the MATLAB Command Window:

```
### Starting VHDL code generation process for filter: iir  
### Generating: D:\hdlfilter_tutorials\hdlsrc\iir.vhd  
### Starting generation of iir VHDL entity  
### Starting generation of iir VHDL architecture  
### First-order section, # 1  
### Second-order section, # 2  
### Second-order section, # 3  
### HDL latency is 3 samples  
### Successful completion of VHDL code generation process for filter: iir  
  
### Starting generation of VHDL Test Bench  
### Generating input stimulus  
### Done generating input stimulus; length 2172 samples.  
### Generating: D:\hdlfilter_tutorials\hdlsrc\iir_tb.vhd  
### Please wait .....  
### Done generating VHDL test bench.
```

The messages include hyperlinks to the generated code and test bench files. Click on these hyperlinks, you can open the code files directly into the MATLAB Editor.

Capturing Code Generation Settings

The **Generate MATLAB code** option of the Generate HDL dialog box makes command-line scripting of HDL filter code and test bench generation easier. The option is located in the **Target** section of the Generate HDL dialog box, as shown in the following figure.



By default, **Generate MATLAB code** is cleared.

When you select **Generate MATLAB code** and generate code, the coder captures nondefault HDL code and test bench generation settings from the GUI and writes out a script you can use to regenerate HDL code for the filter. The script contains:

- Header comments that document the design settings for the filter object from which code was generated.
- A function that takes a filter object as its argument, and passes the filter object in to the `generatehdl` command. The property/value pairs passed to these commands correspond to the code generation settings that applied at the time the file was generated.

The coder writes the script to the target folder. The naming convention for the file is `filter_generatehdl.m`, where `filter` is the filter name defined in the **Name** option.

When code generation completes, the generated script opens automatically for inspection and editing.

In subsequent sessions, you can use the information in the script comments to construct a filter object that is compatible with the `generatehdl` command in the script. Then you can execute the script as a function, passing in the filter object, to generate HDL code.

Tip **Generate MATLAB code** is available only in the GUI. The function `generatehdl` does not have an equivalent property.

Closing Code Generation Session

Click the **Close** to close the Generate HDL dialog box and end a session with the coder.

If you want to preserve your coder settings, the best practice is to select the **Generate MATLAB code** option, as described in “Capturing Code Generation Settings” on page 2-15.

HDL Code for Supported Filter Structures

- “Multirate Filters” on page 3-2
- “Variable Rate CIC Filters” on page 3-8
- “Cascade Filters” on page 3-11
- “Polyphase Sample Rate Converters” on page 3-14
- “Multirate Farrow Sample Rate Converters” on page 3-17
- “Single-Rate Farrow Filters” on page 3-20
- “Programmable Filter Coefficients for FIR Filters” on page 3-27
- “Programmable Filter Coefficients for IIR Filters” on page 3-39
- “DUC and DDC System Objects” on page 3-47

Multirate Filters

Supported Multirate Filter Types

HDL code generation is supported for the following types of multirate filters:

- Cascaded Integrator Comb (CIC) interpolation (`mfilt.cicdecim`)
- Cascaded Integrator Comb (CIC) decimation (`mfilt.cicinterp`)
- Direct-Form Transposed FIR Polyphase Decimator (`mfilt.firtdecim`)
- Direct-Form FIR Polyphase Interpolator (`mfilt.firinterp`)
- Direct-Form FIR Polyphase Decimator (`mfilt.firdecim`)
- Direct-Form FIR Polyphase Sample Rate Converter (`mfilt.firsrc`)
- FIR Hold Interpolator (`mfilt.holdinterp`)
- FIR Linear Interpolator (`mfilt.linearinterp`)

Generating Multirate Filter Code

To generate multirate filter code, you must first select and design one of the supported filter types using `FDATool`, `filterbuilder`, or the MATLAB command line.

After you have created the filter, open the Generate HDL dialog box, set the desired code generation properties, and generate code. GUI options that support multirate filter code generation are described in “Code Generation Options for Multirate Filters” on page 3-2.

If you prefer to generate code via the `generatehdl` function, the coder also defines multirate filter code generation properties that are functionally equivalent to the GUI options. “generatehdl Properties for Multirate Filters” on page 3-7 summarizes these properties.

Code Generation Options for Multirate Filters

When a multirate filter of a supported type (see “Supported Multirate Filter Types” on page 3-2) is designed, the enabled/disabled state of several options in the Generate HDL dialog box changes:

- The **Clock inputs** pulldown menu is enabled. This menu provides two alternatives for generating clock inputs for multirate filters.

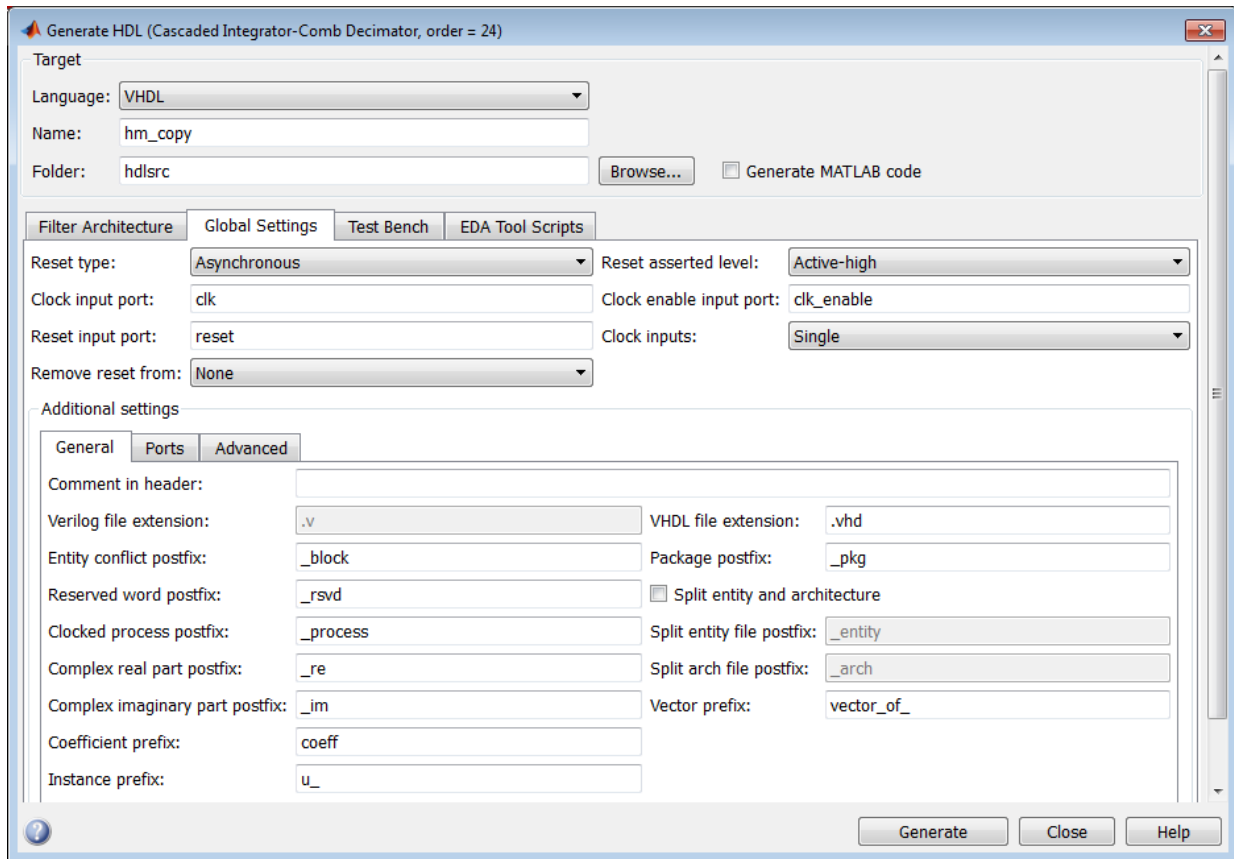
Note: For multirate filters with the **Partly serial** architecture option selected, the **Clock inputs** options is set to **Single** and disabled.

- For CIC filters, the **Coefficient multipliers** option is disabled. Coefficient multipliers are not used in CIC filters.

However, the **Coefficient multipliers** option is enabled for Direct-Form Transposed FIR Polyphase Decimator (`mfilt.firtdecim`) filters.

- For CIC filters, the **FIR adder style** option is disabled, since CIC filters do not require a final adder.

The following figure shows the default settings of the Generate HDL dialog box options for a supported CIC filter.



The **Clock inputs** options are:

- **Single:** When **Single** is selected, the ENTITY declaration for the filter defines a single clock input with an associated clock enable input and clock enable output. The generated code maintains a counter that controls the timing of data transfers to the filter output (for decimation filters) or input (for interpolation filters). The counter behaves as a secondary clock enable whose rate is determined by the filter's decimation or interpolation factor.

The **Single** option is primarily intended for FPGAs. It provides a self-contained solution for multirate filters, and does not require you to provide additional code.

A clock enable output is also generated when **Single** is selected. If you want to customize the name of this output in generated code, see “Setting the Clock Enable Output Name” on page 3-6.

The following code excerpts were generated from a CIC decimation filter having a decimation factor of 4, with **Clock inputs** set to **Single**.

The ENTITY declaration is as follows.

```
ENTITY cic_decim_4_1_single IS
  PORT( clk      : IN   std_logic;
        clk_enable : IN   std_logic;
        reset    : IN   std_logic;
        filter_in : IN   std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        filter_out : OUT  std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        ce_out   : OUT  std_logic
  );
END cic_decim_4_1_single;
```

The signal **counter** is maintained by the clock enable output process (**ce_output**). Every 4th clock cycle, **counter** is toggled to 1.

```
ce_output : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      cur_count <= to_unsigned(0, 4);
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        IF cur_count = 3 THEN
          cur_count <= to_unsigned(0, 4);
        ELSE
          cur_count <= cur_count + 1;
        END IF;
      END IF;
    END IF;
  END PROCESS ce_output;

  counter <= '1' WHEN cur_count = 1 AND clk_enable = '1' ELSE '0';
```

The following code excerpt illustrates a typical use of the **counter** signal, in this case to time the filter output.

```
output_reg_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      output_register <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF counter = '1' THEN
        output_register <= section_out4;
      END IF;
    END IF;
  END PROCESS output_reg_process;
```

- **Multiple:** When **Multiple** is selected, the ENTITY declaration for the filter defines separate clock inputs (each with an associated clock enable input) for each rate of a multirate filter. (For currently supported multirate filters, there are two such rates).

The generated code assumes that the clocks are driven at suitable rates. You are responsible for seeing that the clocks run at relative rates that correspond to the filter's decimation or interpolation factor. To see an example of such code, generate test bench code for your multirate filter and examine the `clk_gen` processes for each clock.

The **Multiple** option is intended for ASICs and FPGAs. It provides more flexibility than the **Single** option, but assumes that you will provide higher-level code for driving your filter's clocks.

Note that synchronizers between multiple clock domains are not provided.

When **Multiple** is selected, clock enable outputs are not generated; therefore the **Clock enable output port** field of the **Global Settings** pane is disabled.

The following ENTITY declaration was generated from a CIC decimation filter with **Clock inputs** set to **Multiple**.

```
ENTITY cic_decim_4_1_multi IS
  PORT( clk          : IN    std_logic;
        clk_enable   : IN    std_logic;
        reset        : IN    std_logic;
        filter_in    : IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
        clk1         : IN    std_logic;
        clk_enable1  : IN    std_logic;
        reset1       : IN    std_logic;
        filter_out   : OUT   std_logic_vector(15 DOWNTO 0) -- sfix16_En15
        );
END cic_decim_4_1_multi;
```

Setting the Clock Enable Output Name

A clock enable output is generated when **Single** is selected from the **Clock inputs** options in the Generate HDL dialog box. The default name for the clock enable output is `ce_out`.

To change the name of the clock enable output, enter the desired name into the **Clock enable output port** field of the **Ports** pane of the Generate HDL dialog box, as shown in the following figure.

The screenshot shows the 'Filter Architecture' configuration window with the following settings:

- Global Settings:**
 - Reset type: Asynchronous
 - Reset asserted level: Active-high
 - Clock input port: clk
 - Clock enable input port: clk_enable
 - Reset input port: reset
 - Clock inputs: Single
 - Remove reset from: None
- Additional settings:**
 - General tab selected.
 - Input data type: std_logic_vector
 - Output data type: Same as input type
 - Clock enable output port: ce_out
 - Input port: filter_in
 - Output port: filter_out
 - Input complexity: Real
 - Add input register
 - Add output register

Note that the coder enables the **Clock enable output port** field only when generating multiple clocks.

generatehdl Properties for Multirate Filters

If you are using `generatehdl` to generate code for a multirate filter, you can set the following properties to specify clock generation options:

- **ClockInputs:** Corresponds to the **Clock inputs** option; selects generation of single or multiple clock inputs for multirate filters.
- **ClockEnableOutputPort:** Corresponds to the **Clock enable output port** field; specifies the name of the clock enable output port.

Variable Rate CIC Filters

In this section...
“Supported Variable Rate CIC Filter Types” on page 3-8
“Code Generation Options for Variable Rate CIC Filters” on page 3-8

Supported Variable Rate CIC Filter Types

The coder supports HDL code generation for variable rate CIC filters, including the following filter types:

- CIC Decimators (`mfilt.cicdecim`)
- CIC Interpolators (`mfilt.cicinterp`)
- Multirate cascade with one CIC stage (`mfilt.cascade`)

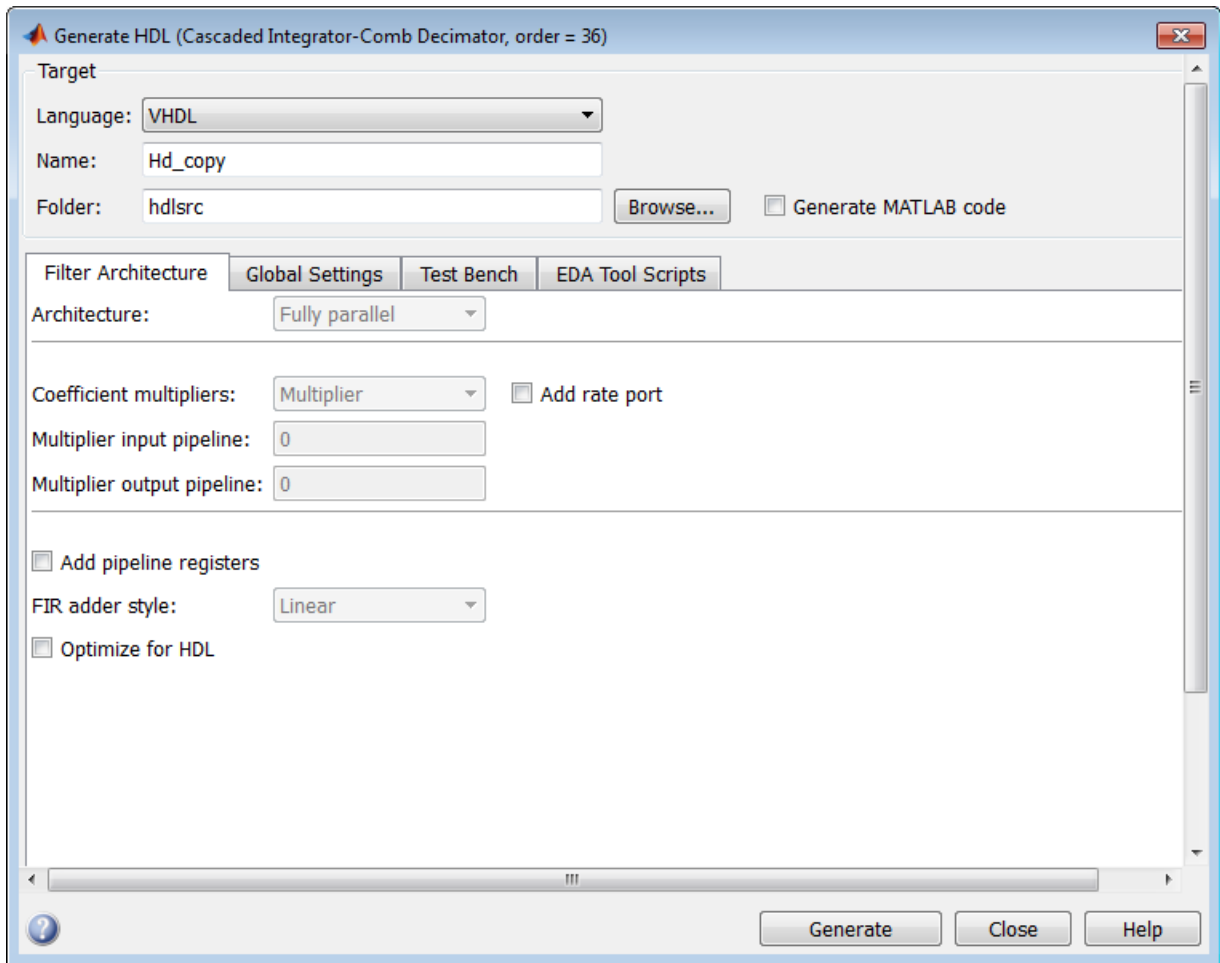
Code Generation Options for Variable Rate CIC Filters

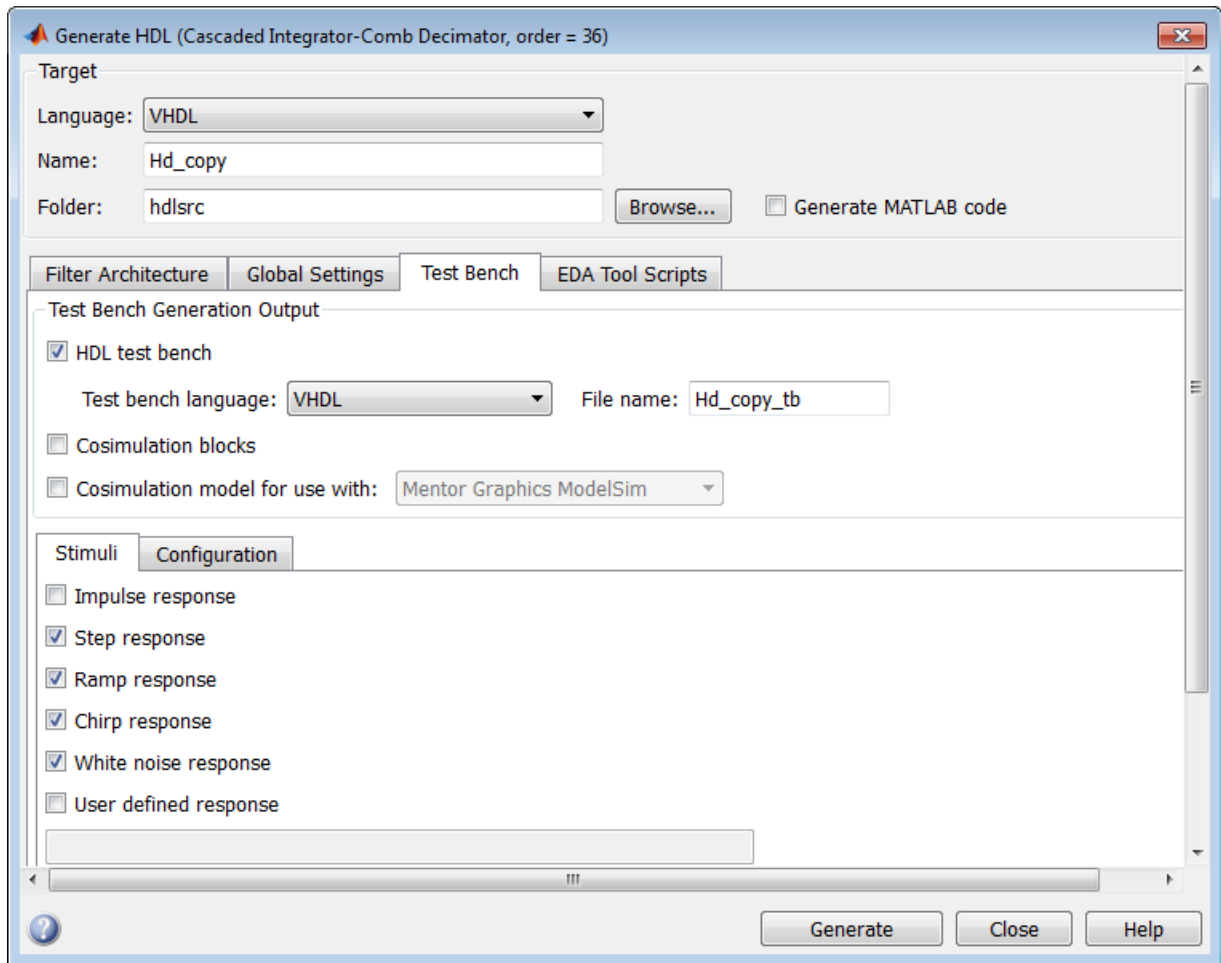
A variable rate CIC filter has a programmable rate change factor. It is assumed that the filter is designed with the maximum rate expected, and that the Decimation Factor (for CIC Decimators) or Interpolation Factor (for CIC Interpolators) is set to this maximum rate change factor.

Two properties support variable rate CIC filters:

- **AddRatePort**: When `AddRatePort` is set 'on', the coder generates `rate` and `load_rate` ports. When the `load_rate` signal is asserted, the `rate` port loads in a rate factor.
- **TestbenchRateStimulus**: Specifies the rate stimulus. If you do not specify `TestbenchRateStimulus`, the coder uses the maximum rate change factor specified in the filter object.

You can also specify these properties in the GUI using the **Add rate port** checkbox and the **Testbench rate stimulus** edit box, shown in the following figures.





Cascade Filters

In this section...
“Supported Cascade Filter Types” on page 3-11
“Generating Cascade Filter Code” on page 3-11

Supported Cascade Filter Types

The coder supports code generation for the following types of cascade filters:

- Multirate cascade of filter objects (`mfilt.cascade`)
- Cascade of discrete-time filter objects (`dfilt.cascade`)

Generating Cascade Filter Code

Generating Cascade Filter Code with FDATool

To generate cascade filter code with FDATool:

- 1 Instantiate the filter stages and cascade them in the MATLAB workspace (see the DSP System Toolbox documentation for the `dfilt.cascade` and `mfilt.cascade` filter objects).

The coder currently imposes certain limitations on the filter types allowed in a cascade filter. See “Rules and Limitations for Code Generation with Cascade Filters” on page 3-12 before creating your filter stages and cascade filter object.

- 2 Import the cascade filter object into FDATool, as described in Import and Export Quantized Filters in the DSP System Toolbox documentation.
- 3 After you have imported the filter, open the Generate HDL dialog box, set the desired code generation properties, and generate code. See “Rules and Limitations for Code Generation with Cascade Filters” on page 3-12.

Generating Cascade Filter Code with the `fdhdltool` Function

To generate cascade filter code from the command line using `fdhdltool`:

- 1 Instantiate the filter stages and cascade them in the MATLAB workspace (see the DSP System Toolbox documentation for the `dfilt.cascade` and `mfilt.cascade` filter objects).

The coder currently imposes certain limitations on the filter types allowed in a cascade filter. See “Rules and Limitations for Code Generation with Cascade Filters” on page 3-12 before creating your filter stages and cascade filter object.

- 2 Call `fdhdltool` to open the Generate HDL dialog box, passing in the cascade filter object as in the following example:

```
fdhdltool(my_cascade)
```

- 3 Set the desired code generation properties, and generate code. See .

“Rules and Limitations for Code Generation with Cascade Filters” on page 3-12

Rules and Limitations for Code Generation with Cascade Filters

The following rules and limitations apply to cascade filters when used for code generation:

- You can generate code for cascades that combine the following filter types:
 - Decimators and/or single-rate filter structures
 - Interpolators and/or single-rate filter structures

Code generation for cascades that include both decimators and interpolators is not currently supported, however. If unsupported filter structures or combinations of filter structures are included in the cascade, code generation is disallowed.

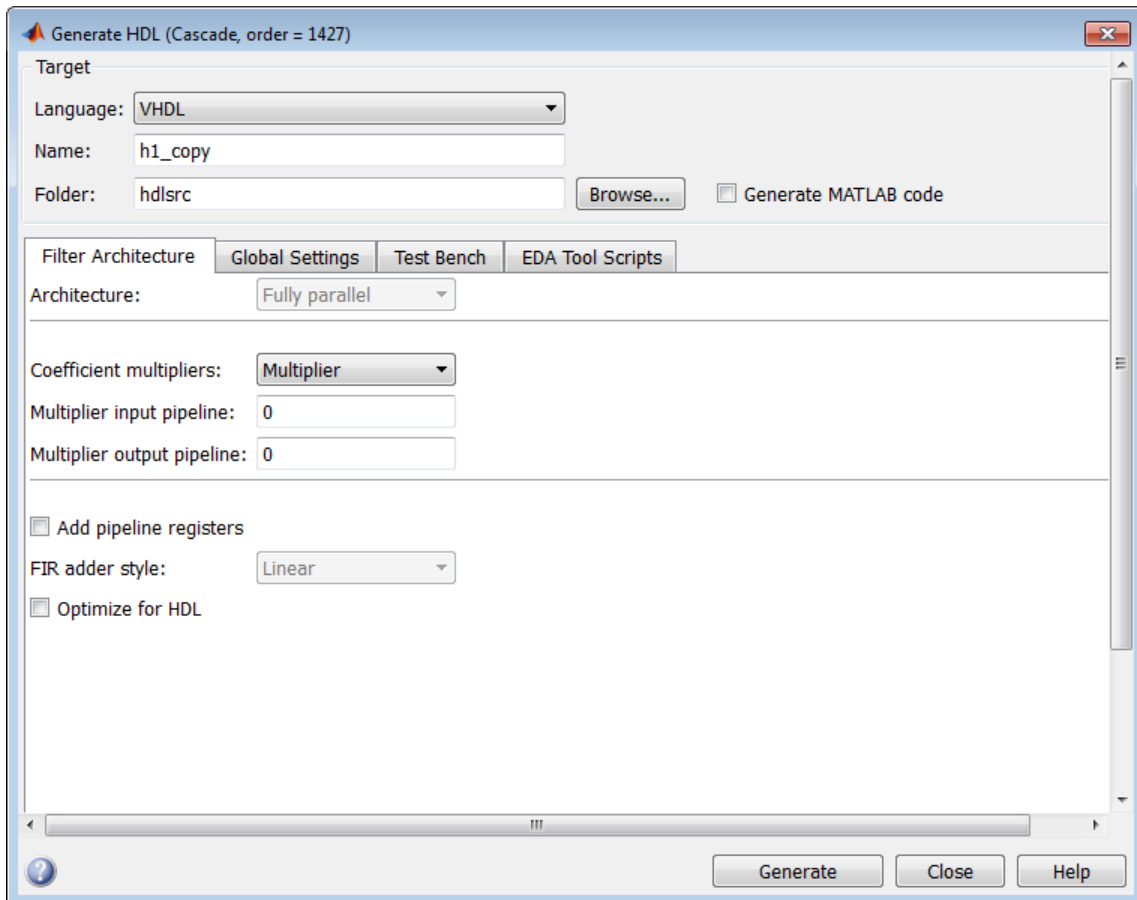
- For code generation, only a flat (single-level) cascade structure is allowed. Nesting of cascade filters is disallowed.
- By default, input and output registers are removed from the stages of the cascade in generated code, except for the input of the first stage and the output of the final stage. However, if the **Add pipeline registers** option in Generate HDL dialog box is selected, the output registers for each stage are generated, and internal pipeline registers may be added, depending on the filter structures.

Note: Code generated for interpolators within a cascade includes input registers, regardless of the setting of the **Add pipeline registers** option.

- When a cascade filter is passed in to the Generate HDL dialog box, the **FIR adder style** option is disabled. If you require tree adders for FIR filters in a cascade, select the **Add pipeline registers** option (since pipelines require tree style FIR adders).

- The coder generates separate HDL code files for each stage of the cascade, in addition to the top-level code for the cascade filter itself. The filter stage code files are identified by appending the string `_stage1`, `_stage2`, ... `_stageN` to the filter name.

The following figure shows the default settings of the Generate HDL dialog box options for a cascade filter design.



Polyphase Sample Rate Converters

In this section...

“About Code Generation for Direct-Form FIR Polyphase Sample Rate Converters” on page 3-14

“HDL Implementation for Polyphase Sample Rate Converter” on page 3-14

About Code Generation for Direct-Form FIR Polyphase Sample Rate Converters

The coder supports code generation for direct-form FIR polyphase sample rate converters (`mfilt.firsrc`). `mfilt.firsrc` is a multirate filter structure that combines an interpolation factor and a decimation factor, allowing you to perform fractional interpolation or decimation on an input signal.

The interpolation factor (l) and decimation factor (m) for a polyphase sample rate converter are specified as integers in the `RateChangeFactors` property of an `mfilt.firsrc` object. The following example code constructs an `mfilt.firsrc` object with a resampling ratio of $5/3$:

```
frac_cvrter = mfilt.firsrc
frac_cvrter.RateChangeFactors = [5 3]
```

Fractional rate resampling can be visualized as a two step process: an interpolation by the factor l , followed by a decimation by the factor m . For example, given a resampling ratio of $5/3$, a fractional sample rate converter raises the sample rate by a factor of 5, using a standard five-path polyphase filter. A resampling switch then reduces the new rate by a factor of 3. This process extracts five output samples for every three input samples.

For general information on this filter structure, see the `mfilt.firsrc` reference page in the DSP System Toolbox documentation.

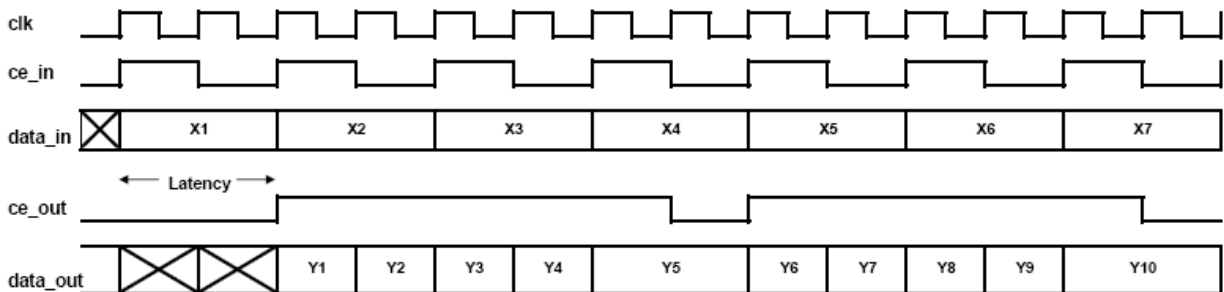
HDL Implementation for Polyphase Sample Rate Converter

Signal Flow, Latency and Timing

The signal flow for the `mfilt.firsrc` filter is similar to the polyphase FIR interpolator (`mfilt.firinterp`). The delay line is advanced such that the inputs are delivered after the required set of polyphase coefficients are processed.

The following diagram illustrates the timing of the HDL implementation for `mfilt.firsrc`. A clock enable input (`ce_in`) enables the inputs into the filter. The outputs, and a clock enable output (`ce_out`) are produced and delivered simultaneously, which results in a nonperiodic output.

Sample rate conversion filter timing diagram for $L/M = 5/3$



Clock Rate

The clock rate required to process the hardware logic is related to the input rate as:

$$\text{ceil}(\text{Hm.RateChangeFactors}(1) / \text{Hm.RateChangeFactors}(2))$$

For example, for a resampling ratio of $5/3$, the clock rate is $\text{ceil}(5/3) = 2$, or twice the input sample rate. The inputs are delivered at every other clock cycle. The outputs are delivered as soon as they are produced and therefore are nonperiodic.

Note: When the generated code or hardware logic is deployed, the outputs must be taken into a FIFO designed with outputs occurring at the desired sampling rate.

Clock Enable Ports

The generated HDL entity or module the `mfilt.firsrc` filter has two clock enable ports:

- Clock enable output: the default clock enable output port name is `ce_out`. As with other multirate filters, you can use the **Clock enable output port** field of the Generate HDL dialog box to specify the port name. Alternatively, you can use the `ClockEnableOutputPort` property to set the port name in the `generatehdl` command.

- Clock enable input: the default clock enable input port name is `ce_in`. In the current release, there isn't an option to change the name of this port.

Test Bench Generation

Generated test benches use the input and output clock enables to force in and verify the test vectors.

Generating Code for `mfilt.firsrc` filters at the Command Line

The following example constructs a fixed-point `mfilt.firsrc` object with a resampling ratio of 5/3, and generates VHDL filter code.

```
frac_cvrter = mfilt.firsrc;
frac_cvrter.arithmetic = 'fixed';
frac_cvrter.RateChangeFactors = [5 3];
generatehdl(frac_cvrter)

### Starting VHDL code generation process for filter: frac_cvrter
### Generating: D:\Work\post_2006b_Adsp_sbox\hdlsrc\frac_cvrter.vhd
### Starting generation of frac_cvrter VHDL entity
### Starting generation of frac_cvrter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: frac_cvrter
```

Filter Design HDL Coder does not require special code generation properties when generating code for `mfilt.firsrc` filters. However, the following code generation options are not supported for `mfilt.firsrc` filters:

- Use of pipeline registers (`AddPipelineRegisters`)
- Distributed Arithmetic architecture
- Fully or partially serial architectures
- Multiple clock inputs

Multirate Farrow Sample Rate Converters

In this section...

“About Code Generation for Multirate Farrow Sample Rate Converters” on page 3-17

“Generating Code for `mfilt.farrowsrc` Filters at the Command Line” on page 3-17

“Generating Code for `mfilt.farrowsrc` Filters in the GUI” on page 3-18

About Code Generation for Multirate Farrow Sample Rate Converters

The coder supports code generation for multirate Farrow sample rate converters (`mfilt.farrowsrc`). `mfilt.farrowsrc` is a multirate filter structure that implements a sample rate converter with an arbitrary conversion factor determined by its interpolation and decimation factors.

Unlike a single-rate Farrow filter (see “Single-Rate Farrow Filters” on page 3-20), a multirate Farrow sample rate converter does not have a fractional delay input.

For general information on this filter structure, see the `mfilt.farrowsrc` reference page in the DSP System Toolbox documentation.

Generating Code for `mfilt.farrowsrc` Filters at the Command Line

You can generate HDL code for either a standalone `mfilt.farrowsrc` object, or a cascade that includes a `mfilt.farrowsrc` object. This section provides simple examples for each case.

The following example instantiates a standalone fixed-point Farrow sample rate converter. The converter performs a conversion between two standard audio rates, from 44.1 kHz to 48 kHz. The example generates both VHDL code and a VHDL test bench.

```
[L,M] = rat(48/44.1);  
Hm = mfilt.farrowsrc(L,M);  
Hm.arithmetic = 'fixed';  
generatehdl(Hm, 'GenerateHDLTestbench', 'on')
```

The following example illustrates code generation for a cascade that include a `mfilt.farrowsrc` filter. The coder requires that the `mfilt.farrowsrc` filter is in the last position of the cascade.

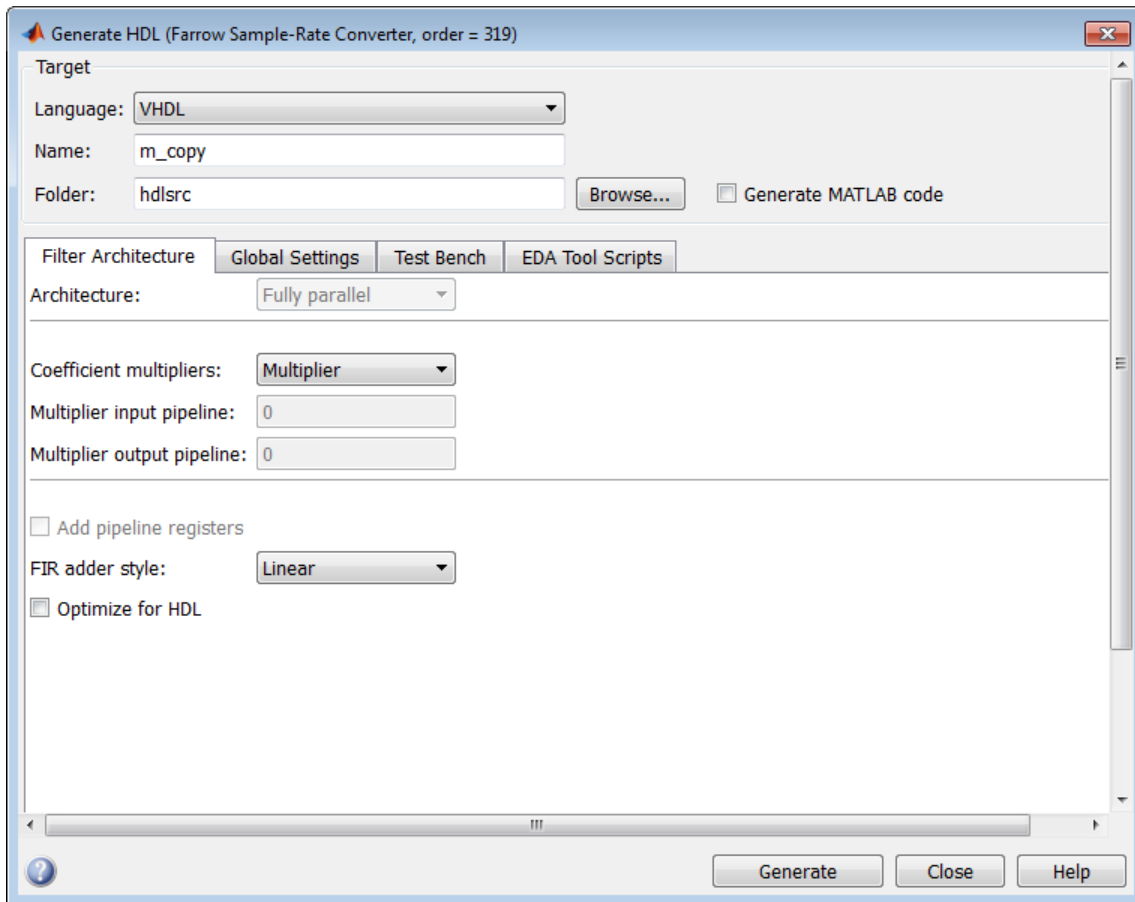
```
Astop = 50; % Minimum stopband attenuation
% First interpolate the original 8 kHz signal by 4 using a
% cascade of FIR halfband filters.
TW = .125; % Transition Width
f2 = fdesign.interpolator(4,'Nyquist',4,'TW,Ast',TW,Astop);
hfir = design(f2,'multistage','HalfbandDesignMethod','equiripple');
% Then, interpolate the intermediate 8x4=32 kHz signal by 44.1/32 =
% 1.378125 to get the desired 44.1 kHz final sampling frequency. We use a
% cubic Lagrange polynomial-based filter for this purpose. We first design
% a single rate Farrow filter, then convert it to a multirate Farrow filter.
N = 3; % Polynomial Order
ffar = fdesign.fracdelay(0,'N',N);
hfar = design(ffar,'lagrange');
[L,M] = rat(1.378125); % Interpolation and decimation factors
hfir = mfilt.farrowsrc(L,M,hfar.Coefficients);
% The overall filter is obtained by cascading the two filters.
% For HDL code generation compatibility, the cascade of FIR filters is flattened.
% Note that the mfilt.farrowsrc filter is at the end of the cascade.
h2 = cascade(hfir.Stage(1), hfir.Stage(2),hfar);
generatehdl(h2,'GenerateHDLTestbench','on');
```

Generating Code for mfilt.farrowsrc Filters in the GUI

FDATool and filterbuilder do not currently support mfilt.farrowsrc filters. If you want to generate code an mfilt.farrowsrc filter in the HDL code generation GUI, you can use the fdhdltool command, as in the following example:

```
[L,M] = rat(48/44.1);
m = mfilt.farrowsrc(L,M);
fdhdltool(m);
```

fdhdltool opens the Generate HDL dialog box for the mfilt.farrowsrc filter, as shown in the following figure.



The following code generation options are not supported for `mfilt.farrowsrc` filters and are disabled in the GUI:

- **Add pipeline registers**
- Distributed Arithmetic architecture
- Fully or partially serial architectures
- Multiple clock inputs

Single-Rate Farrow Filters

In this section...

“About Code Generation for Single-Rate Farrow Filters” on page 3-20

“Code Generation Properties for Farrow Filters” on page 3-20

“GUI Options for Farrow Filters” on page 3-22

“Farrow Filter Code Generation Mechanics” on page 3-24

About Code Generation for Single-Rate Farrow Filters

The coder supports HDL code generation for the following single-rate Farrow filter structures:

- `dfilt.farrowlinearfd`
- `dfilt.farrowfd`

A Farrow filter differs from a conventional filter because it has a fractional delay input in addition to a signal input. The fractional delay input enables the use of time-varying delays, as the filter operates. The fractional delay input receives a signal taking on values between 0 and 1.0. For general information how to construct and use Farrow filter objects, see the DSP System Toolbox documentation.

The coder provides `generatehdl` properties and equivalent GUI options that let you:

- Define the fractional delay port name used in generated code.
- Apply a variety of test bench stimulus signals to the fractional delay port, or define your own stimulus signal.

Code Generation Properties for Farrow Filters

The following properties support Farrow filter code generation:

- `FracDelayPort` (string). This property specifies the name of the fractional delay port in generated code. The default name is `'filter_fd'`. In the following example, the name `'FractionalDelay'` is assigned to the fractional delay port.

```
D = .3;
hd = dfilt.farrowfd(D);
generatehdl(hd, 'FracDelayPort', 'FractionalDelay');
```

- `TestBenchFracDelayStimulus` (string). This property specifies a stimulus signal applied to the fractional delay port in test bench code.

By default, a constant value is obtained from the `FracDelay` property of the Farrow filter object, and applied to the fractional delay port. To use the default, leave the `TestBenchFracDelayStimulus` property unspecified, or pass in the empty string (''). In the following example, the filter's `FracDelay` property is set to 0.6, and this value is used (by default) as the fractional delay stimulus.

```
D = .3;
hd = dfilt.farrowfd(D);
hd.Fracdelay = 0.6;
generatehdl(hd, 'GenerateHDLTestbench', 'on');
```

Alternatively, you can specify generation of the following types of stimulus vectors:

- `'RandSweep'`: A vector of random values within the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.
- `'RampSweep'`: A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.
- A user-defined stimulus vector. You can pass in a call to a function that returns a vector. Alternatively, create the vector in the workspace and pass it in as shown in the following code example:

```
D = .3;
hd = dfilt.farrowfd(D);
inputdata = generatetbstimulus(hd, 'TestBenchStimulus', {'ramp'});
mytestv = [0.5*ones(1, length(inputdata)/2), 0.2*ones(1, length(inputdata)/2)];
generatehdl(hd, 'GenerateHDLTestbench', 'on', 'TestBenchStimulus', {'noise'},...
'TestbenchFracDelayStimulus',mytestv);
```

Note: A user-defined fractional delay stimulus signal must have the same length as the test bench input signal. If the two signals do not have equal length, test bench generation terminates with an error message. The error message displays the signal lengths, as shown in the following example:

```
D = .3;
hd = dfilt.farrowfd(D);
inputdata = generatetbstimulus(hd, 'TestBenchStimulus', {'ramp'});
mytestv = [0.5*ones(1, length(inputdata)/2), 0.2*ones(1, length(inputdata)/2)];
```

```
generatehdl(hd, 'GenerateHDLTestbench', 'on', 'TestBenchStimulus', {'noise' 'chirp'},...
'TestbenchFracDelayStimulus',mytestv)
```

??? Error using ==> generatevhdltb
The lengths of specified vectors for FracDelay (1026) and Input (2052) do not match.

GUI Options for Farrow Filters

This section describes Farrow filter code generation options that are available in the Filter Design HDL Coder GUI. These options correspond to the properties described in “Code Generation Properties for Farrow Filters” on page 3-20.

Note: The Farrow filter options are displayed only when the a Farrow filter is selected for HDL code generation.

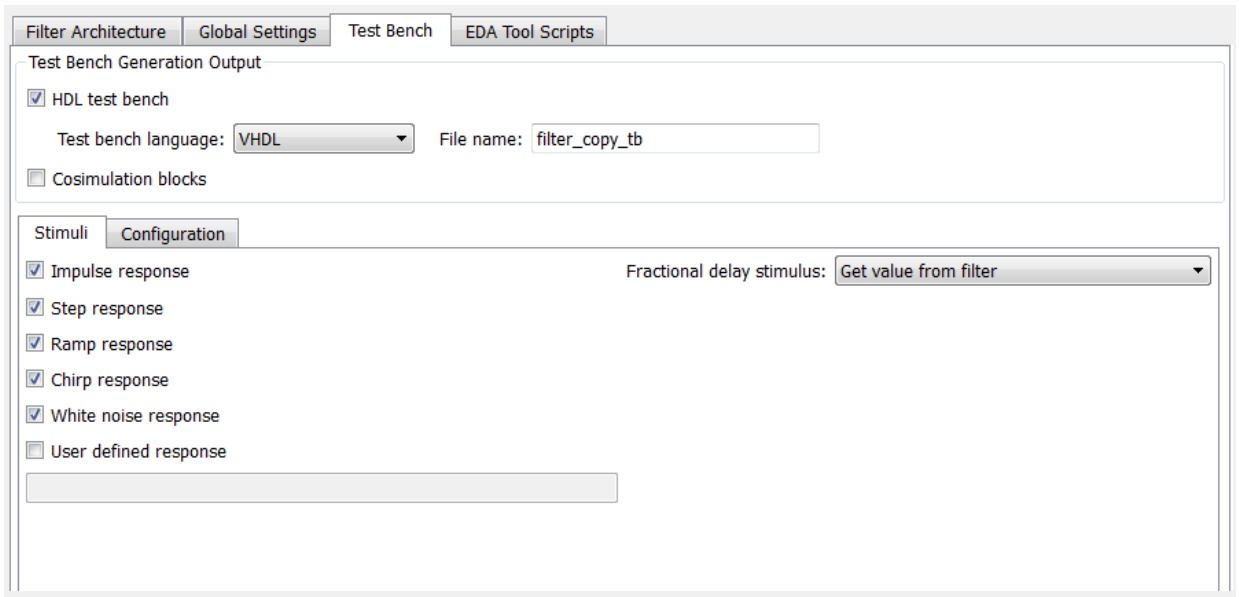
The Farrow filter options are:

- The **Fractional delay port** field in the **Ports** pane of the Generate HDL dialog box (shown in the following figure) specifies the name of the fractional delay port in generated code. The default name is `filter_fd`.

The screenshot shows the 'Filter Architecture' dialog box with the 'Additional settings' section expanded. The 'Ports' tab is selected, showing the following configuration:

- Reset type: Asynchronous
- Reset asserted level: Active-high
- Clock input port: clk
- Reset input port: reset
- Clock enable input port: clk_enable
- Remove reset from: None
- Clock inputs: Single
- Input data type: std_logic_vector
- Output data type: Same as input type
- Clock enable output port: ce_out
- Input port: filter_in
- Output port: filter_out
- Fractional delay port: filter_fd
- Add input register
- Add output register

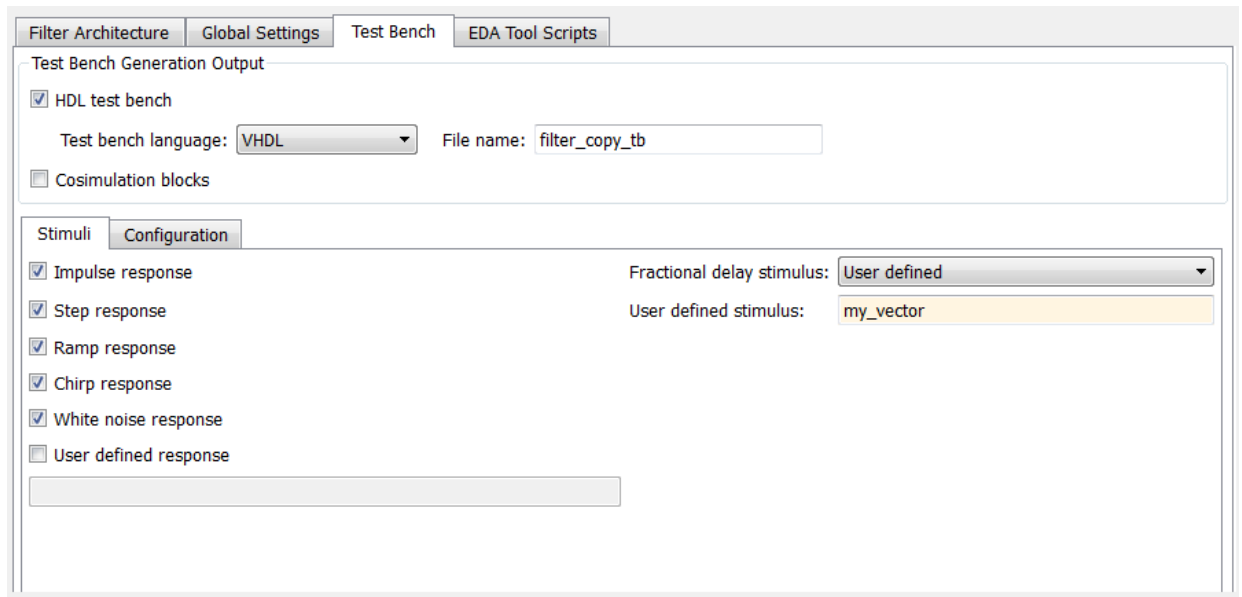
- The **Fractional delay stimulus** pop-up list in the **Test Bench** pane of the Generate HDL dialog box (shown in the following figure) lets you select a stimulus signal applied to the fractional delay port in the generated test bench.



The **Fractional delay stimulus** list lets you select generation of the following types of stimulus signals:

- **Get value from filter:** (default). A constant value is obtained from the `FracDelay` property of the Farrow filter object, and applied to the fractional delay port.
- **Ramp sweep.** A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.
- **Random sweep.** A vector of random values in the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.

- **User defined.** When you select this option, the **User defined stimulus** field is enabled. You can enter a call to a function that returns a vector in the **User defined stimulus** field. Alternatively, create the vector as a workspace variable and enter the variable name, as shown in the following figure.



Farrow Filter Code Generation Mechanics

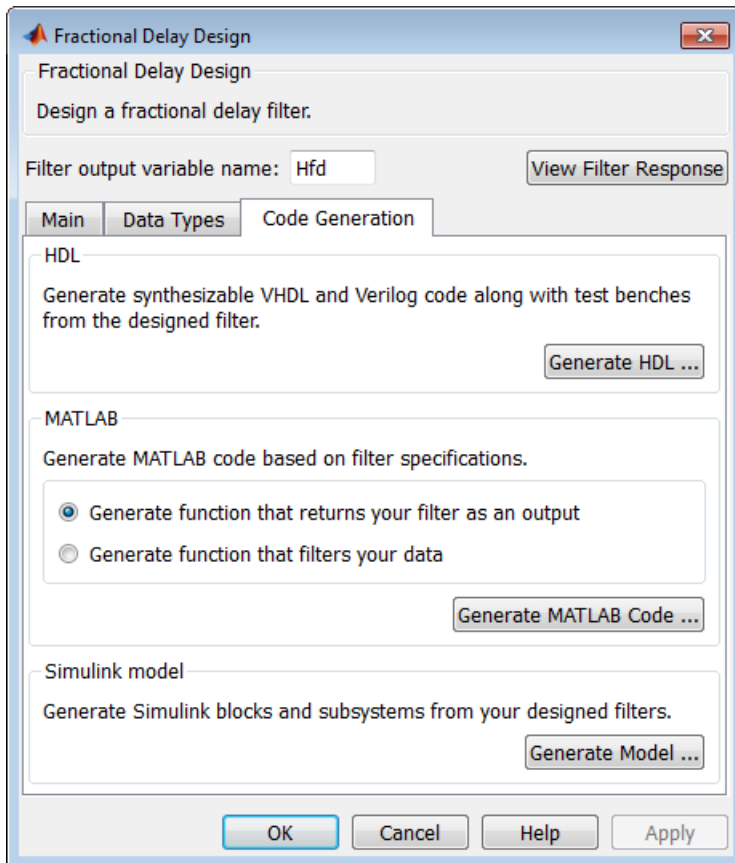
FDATool does not support design or import of Farrow filters. To generate HDL code for a Farrow filter, use one of the following methods:

- Use the MATLAB command line to create a Farrow filter object and initiate code generation, and set Farrow-related properties, as in the examples shown in “Code Generation Properties for Farrow Filters” on page 3-20.
- Use the MATLAB command line to create a Farrow filter object. Then use the `fdhdltool` function to open the Generate HDL dialog box. For example, the following commands create a Farrow linear fractional delay filter object `Hd` and pass it in to `fdhdltool`:

```
D = .3
Hd = dfilt.farrowlinearfd(D)
```

```
Hd.arithmetic = 'fixed'
fdhdltool(Hd)
```

- Use `filterbuilder` to design a Farrow (fractional delay) filter object. Then, select the **Code Generation** pane of the `filterbuilder` dialog box (shown in the following figure). Click the **Generate HDL** button to open the Generate HDL dialog box, specify code generation options, and generate code.



Options Disabled for Farrow Filters

The coder disables some options or sets them to fixed default value when the Generate HDL dialog box opens with a Farrow filter. The options affected are:

Architecture. The coder sets this option to its default (**Fully parallel**) and disables it.

Clock inputs. The coder sets this option to its default (**Single**) and disables it.

Programmable Filter Coefficients for FIR Filters

In this section...

“About Programmable Filter Coefficients for FIR Filters” on page 3-27

“Supported FIR Filter Types” on page 3-28

“Supported Parallel and Serial Filter Architectures” on page 3-28

“Generating a Port Interface for Programmable FIR Coefficients” on page 3-28

“Generating a Test Bench for Programmable FIR Coefficients” on page 3-29

“GUI Options for Programmable Coefficients” on page 3-31

“Using Programmable Coefficients with Serial FIR Filter Architectures” on page 3-32

About Programmable Filter Coefficients for FIR Filters

By default, the coder obtains filter coefficients from a filter object and hard-codes them into the generated code. An HDL filter realization generated in this way cannot be used with a different set of coefficients.

For direct-form FIR filters, the coder provides GUI options and corresponding command-line properties that let you:

- Generate an interface for loading coefficients from memory. Coefficients stored in memory are called *programmable coefficients*.
- Test the interface.

To use programmable coefficients, a port interface (referred to as a *processor interface*) is generated for the filter entity or module. Coefficient loading is assumed to be under the control of a microprocessor that is external to the generated filter. The filter uses the loaded coefficients for processing input samples.

If you choose one of the serial FIR filter architectures, you can also specify storage of programmable coefficients in your choice of RAM (dual-port or single-port) or register file. See “Programmable Filter Coefficients for FIR Filters” on page 3-27.

Note: You can also generate a processor interface for loading IIR filter coefficients. See “Programmable Filter Coefficients for IIR Filters” on page 3-39 for further information.

Supported FIR Filter Types

Programmable filter coefficients are supported for the following direct-form FIR filter types:

- `dfilt.dffir`
- `dfilt.dfsymfir`
- `dfilt.dfasymfir`

Supported Parallel and Serial Filter Architectures

Programmable filter coefficients are supported for the following filter architectures:

- Fully parallel
- Fully serial
- Partly serial
- Cascade serial

If you choose one of the serial filter architectures, see “Using Programmable Coefficients with Serial FIR Filter Architectures” on page 3-32 for special considerations that apply to these architectures.

Generating a Port Interface for Programmable FIR Coefficients

This section describes how to use the `CoefficientSource` property to specify that a processor interface for loading coefficients is generated. You can also use the **Coefficient source** list on the Generate HDL dialog box for this purpose.

The valid value strings for the property are:

- `'Internal'`: (Default) Do not generate a processor interface. Coefficients are obtained from the filter object and hard-coded.
- `'ProcessorInterface'`: Generate a processor interface for coefficients.

When you specify `'ProcessorInterface'`, the generated entity or module definition for the filter includes the following port definitions:

- `coeffs_in`: Input port for coefficient data

- `write_address`: Write address for coefficient memory
- `write_enable`: Write enable signal for coefficient memory
- `write_done`: Signal to indicate completion of coefficient write operation

Example

In the following command-line example, a processor interface is generated in VHDL code for a direct-form symmetric FIR filter with fully parallel (default) architecture.

```
Hd = design(fdesign.lowpass, 'equiripple', 'FilterStructure', 'dfsymfir')
generatehdl(Hd, 'CoefficientSource', 'ProcessorInterface')
```

The following listing shows the VHDL entity definition generated for the filter object Hd.

```
ENTITY Hd IS
  PORT( clk           : IN   std_logic;
        clk_enable    : IN   std_logic;
        reset         : IN   std_logic;
        filter_in     : IN   real; -- double
        write_enable  : IN   std_logic;
        write_done    : IN   std_logic;
        write_address : IN   real; -- double
        coeffs_in     : IN   real; -- double
        filter_out    : OUT  real -- double
        );
END Hd;
```

Generating a Test Bench for Programmable FIR Coefficients

This section describes how to use the `TestbenchCoeffStimulus` property to specify how the test bench drives the coefficient ports. You can also use the **Coefficient stimulus** option for this purpose.

When a coefficient memory interface has been generated for a filter, the coefficient ports have associated test vectors. The `TestbenchCoeffStimulus` property determines how the test bench drives the coefficient ports.

The `TestBenchStimulus` property determines the filter input stimuli.

The `TestbenchCoeffStimulus` property selects from two types of test benches. `TestbenchCoeffStimulus` takes a vector argument. The valid values are:

- `[]`: Empty vector.

This is the default. When the value of `TestbenchCoeffStimulus` is unspecified (or set to the default value of `[]`), the test bench loads the coefficients from the filter object and then forces the input stimuli. This shows the response to the input stimuli and verifies that the interface writes one set of coefficients into the memory without encountering an error.

- `[coeff1, coeff2, ...coeffN]`: Vector of `N` coefficients, where `N` is determined as follows:
 - For symmetric filters, `N` must equal `ceil(length(filterObj.Numerator)/2)`.
 - For symmetric filters, `N` must equal `floor(length(filterObj.Numerator)/2)`.
 - For other filters, `N` must equal the length of the filter object.

In this case, the filter processes the input stimuli twice. First, the test bench loads the coefficients from the filter object and forces the input stimuli to show the response. Then, the filter loads the set of coefficients specified in the `TestbenchCoeffStimulus` vector, and shows the response by processing the same input stimuli for a second time. In this case, the internal states of the filter, as set by the first run of the input stimulus, are retained. The test bench verifies that the interface writes two different sets of coefficients into the coefficient memory. The test bench also provides an example of how the memory interface can be used to program the filter with different sets of coefficients.

Note: If a coefficient memory interface has not been previously generated for the filter, the `TestbenchCoeffStimulus` property is ignored.

Example

In the following example, a processor interface is generated for a direct-form symmetric FIR filter with fully parallel (default) architecture. The coefficients for the filter object are defined in the vector `b`. Test bench code is then generated, using a second set of coefficients defined in the vector `c`. Note that `c` is trimmed to the effective length of the filter.

```
b = [-0.01 0.1 0.8 0.1 -0.01]
c = [-0.03 0.5 0.7 0.5 -0.03]
c = c(1:ceil(length(c)/2))
hd = dfilt.dfsymfir(b)
generatehdl(hd, 'GenerateHDLTestbench', 'on', 'CoefficientSource', 'ProcessorInterface',...
```

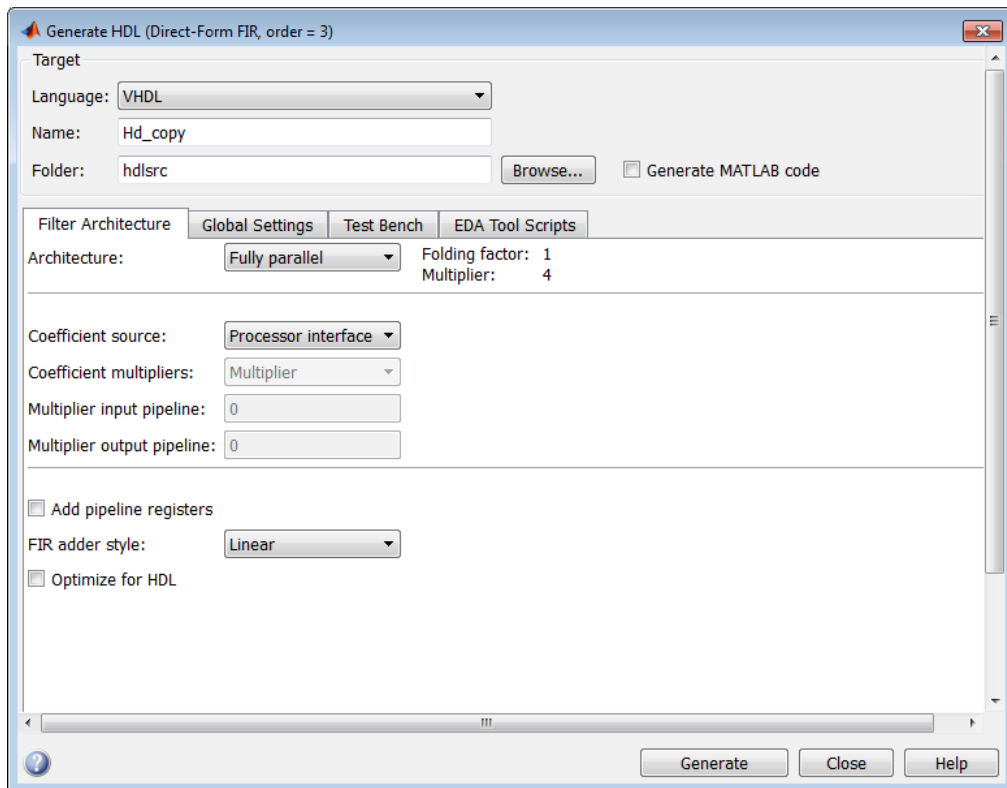

'TestbenchCoeffStimulus', c)

GUI Options for Programmable Coefficients

The following GUI options let you specify programmable coefficients:

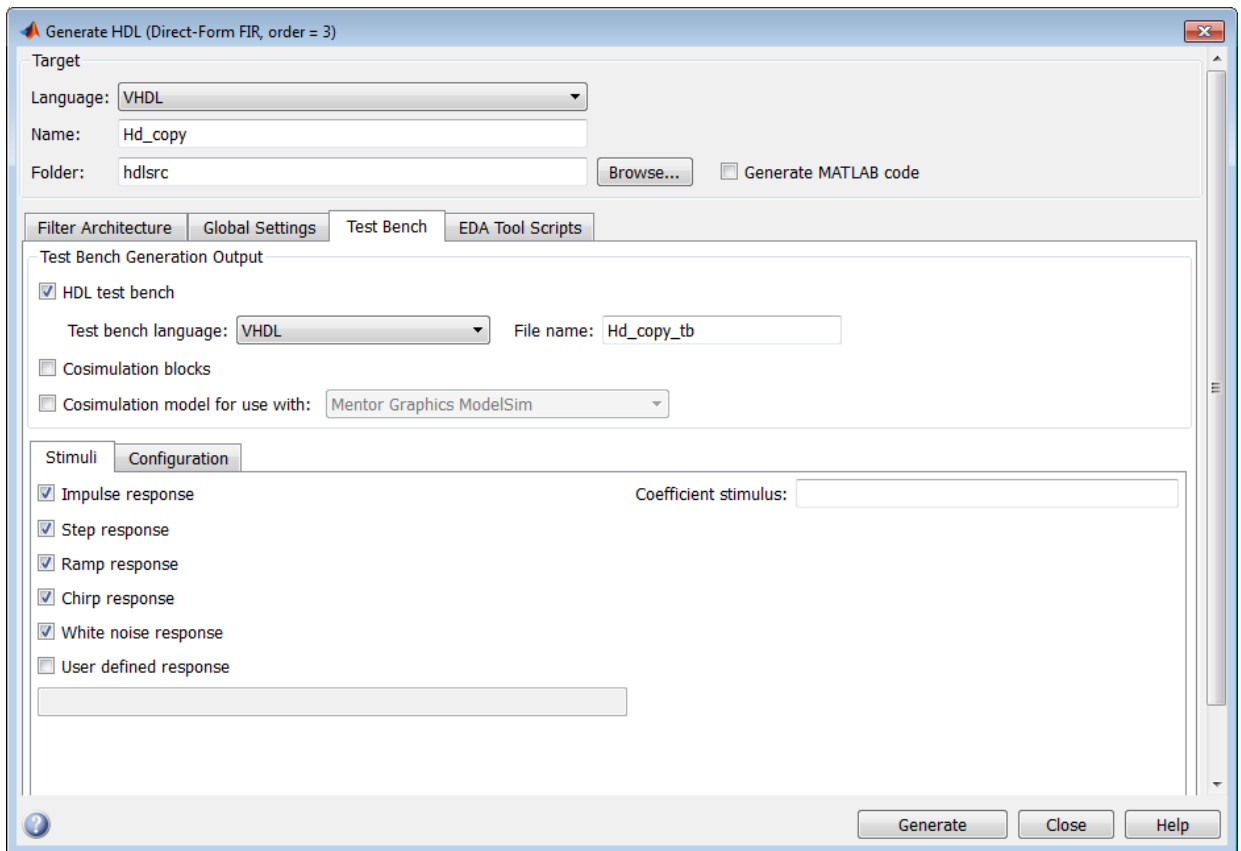
- The **Coefficient source** list on the Generate HDL dialog box lets you select whether coefficients are obtained from the filter object and hard-coded (**Internal**), or from memory (**Processor interface**). The default is **Internal**.

The corresponding command-line property is **CoefficientSource** (see “Generating a Port Interface for Programmable FIR Coefficients” on page 3-28).



- The **Coefficient stimulus** option on the **Test Bench** pane of the Generate HDL dialog box lets you specify how the test bench tests the generated memory interface.

The corresponding command-line property is `TestbenchCoeffStimulus` (see “Generating a Test Bench for Programmable FIR Coefficients” on page 3-29).



Using Programmable Coefficients with Serial FIR Filter Architectures

This section discusses special considerations for using programmable filter coefficients with FIR filters that have one of the following serial architectures:

- Fully serial
- Partly serial
- Cascade serial

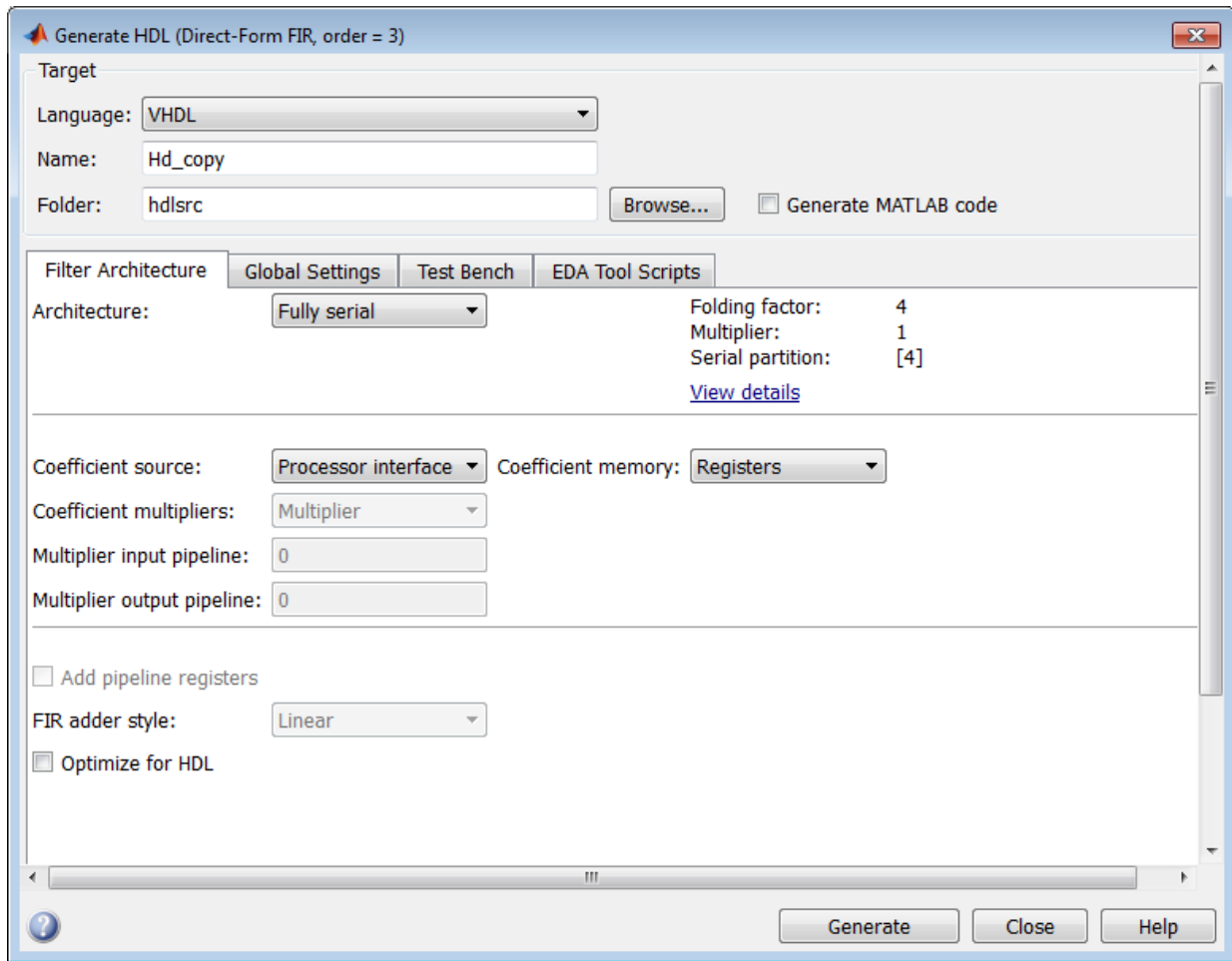
Specifying Memory for Programmable Coefficients

By default, the processor interface for programmable coefficients loads the coefficients from a register file. The **Coefficient memory** pulldown menu lets you specify alternative RAM-based storage for programmable coefficients.

Coefficient memory displays for FIR filters, when both the following conditions are met:

- The **Coefficient source** option specifies Processor interface.
- The **Architecture** option specifies a Fully serial, Partly serial, or Cascade serial architecture.

The following figure shows where you specify the **Coefficient memory** option for a fully serial FIR filter. You can select an option using the drop-down list.



The following table summarizes the **Coefficient memory** options.

Coefficient Memory Pulldown Selection	Description
Registers	<i>default:</i> Store programmable coefficients in a register file.
Single Port RAMs	Store programmable coefficients in single-port RAM.

Coefficient Memory Pulldown Selection	Description
	The coder writes RAM interface code to one or more separate files, depending on the filter partitioning.
Dual Port RAMs	<p>Store programmable coefficients in dual-port RAM.</p> <p>The coder writes RAM interface code to one or more separate files, depending on the filter partitioning.</p>

You can also use the `CoefficientMemory` command-line property to specify alternative RAM-based storage for programmable coefficients. The following table summarizes the options.

CoefficientMemory Setting	Description
'CoefficientMemory', 'Registers'	<i>default</i> : Store programmable coefficients in a register file.
'CoefficientMemory', 'SinglePortRAMs'	<p>Store programmable coefficients in single-port RAM.</p> <p>The coder writes RAM interface code to one or more separate files, depending on the filter partitioning.</p>
'CoefficientMemory', 'DualPortRAMs'	<p>Store programmable coefficients in dual-port RAM.</p> <p>The coder writes RAM interface code to one or more separate files, depending on the filter partitioning.</p>

The commands in the following example create an asymmetric filter `Hd`, and generate VHDL code for the filter using a partly serial architecture, with a dual-port RAM interface for programmable coefficients.

```

coeffs = fir1(22,0.45)
Hd = dfilt.dfasymfir(coeffs)
Hd.arithmetic = 'fixed'
generatehdl(Hd, 'SerialPartition', [7 4], 'CoefficientSource', ...
'ProcessorInterface', 'CoefficientMemory', 'DualPortRAMs')
    
```

Tip When you use this property, be sure to set `CoefficientSource` to `'ProcessorInterface'`. The coder ignores `CoefficientMemory` unless it is generating an interface for programmable coefficients.

Timing Considerations

In a serial implementation for a FIR filter, the rate of the system clock (`clk`) is generally a multiple of the filter's input data rate (i.e., the nominal sample rate of the filter). The exact relationship between the clock rate and the filter's data rate is determined by the choice of serial architecture and partitioning.

Programmable coefficients load into the `coeffs_in` port at either the system clock rate (faster) or the input data (slower) rate. If your design requires loading of coefficients at the faster rate, observe the following points:

- When `write_enable` asserts, coefficients load from the `coeffs_in` port into coefficient memory at the address specified by `write_address`.
- `write_done` can assert on clock cycles for any duration. If `write_done` asserts at least two `clk` cycles before the arrival of the next data input value, new coefficients will be applied with the next data sample. Otherwise, new coefficients will be applied for the data after the next sample.

The next two code generation examples illustrate how serial partitioning affects the timing of coefficient loading. Both examples generate code for a filter `Hd`. `Hd` is an asymmetric filter that requires 11 coefficients. The following code creates the filter `Hd`:

```
rng(13893,'v5uniform')
b = rand(1,23)
Hd = dfilt.dfasymfir(b)
Hd.Arithmetic = 'fixed'
```

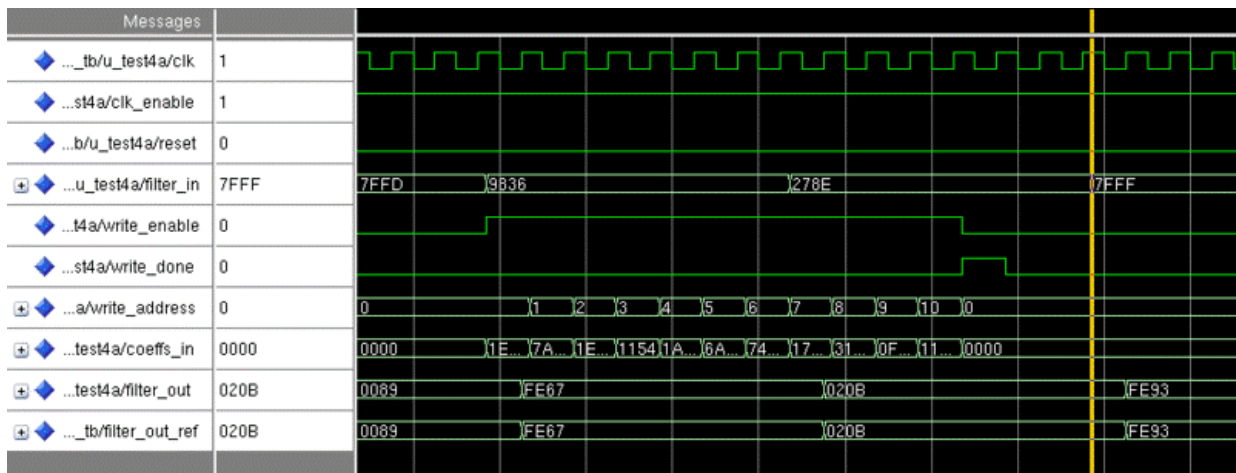
The following command generates VHDL code for `Hd`, using a partly serial architecture with the serial partition [7 4]. `CoefficientSource` specifies that a processor interface is generated, with the default `CoefficientStimulus`.

```
generatehdl(Hd,'SerialPartition',[7 4],'CoefficientSource','ProcessorInterface')

### Clock rate is 7 times the input sample rate for this architecture.
### HDL latency is 2 samples
```

As reported by the coder, this partitioning results in a clock rate that is 7 times the input sample rate.

The following timing diagram illustrates the timing of coefficient loading relative to the timing of input data samples. While `write_enable` is asserted, 11 coefficient values are loaded, via `coeffs_in`, to 11 sequential memory addresses. On the next `clk` cycle, `write_enable` is deasserted and `write_done` is asserted for one clock period. The coefficient loading operation is completed within 2 cycles of data input, allowing 2 `clk` cycles to elapse before the arrival of the data value 07FFF. Therefore the newly loaded coefficients are applied to that data sample.



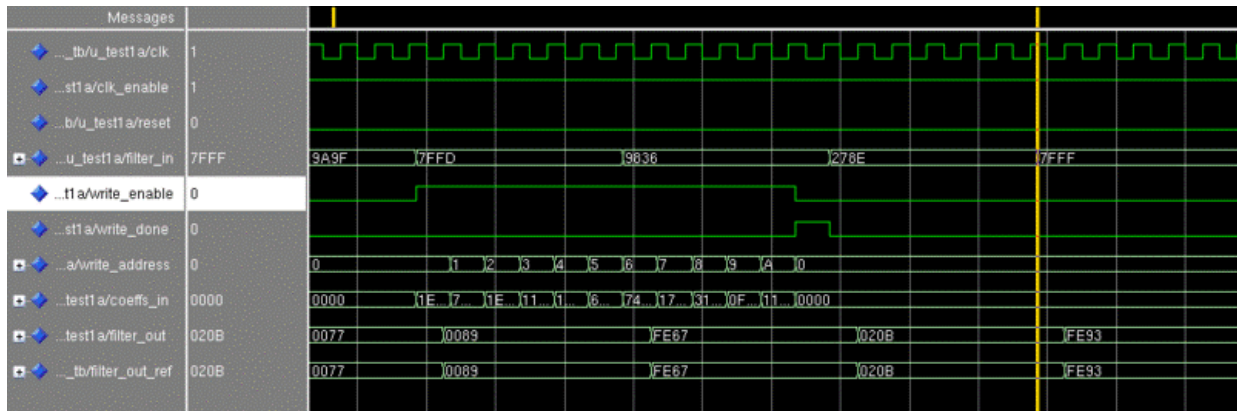
The following code generation example defines a serial partition of [6 5] for the same filter. This results in a slower clock rate, 6 times the input sample rate.

```
generatehd1(Hd, 'SerialPartition', [6 5], 'CoefficientSource', 'ProcessorInterface')

### Clock rate is 6 times the input sample rate for this architecture.
### HDL latency is 2 samples
```

The following timing diagram illustrates that `write_done` deasserts too late for the coefficients to be applied to the arriving data value 278E. They are applied instead to the next sample, 7FFF.

3 HDL Code for Supported Filter Structures



Programmable Filter Coefficients for IIR Filters

About Programmable Filter Coefficients for IIR Filters

By default, the coder obtains filter coefficients from a filter object and hard-codes them into the generated code. An HDL filter realization generated in this way cannot be used with a different set of coefficients.

For IIR filters, the coder provides GUI options and corresponding command-line properties that let you:

- Generate an interface for loading coefficients from memory. Coefficients stored in memory are called *programmable coefficients*.
- Test the interface.

To use programmable coefficients, a port interface (referred to as a *processor interface*) is generated for the filter entity or module. Coefficient loading is assumed to be under the control of a microprocessor that is external to the generated filter. The filter uses the loaded coefficients for processing input samples.

For IIR filters, the current release supports programmable coefficients that are stored in a register file.

Note: You can also generate a processor interface for loading FIR filter coefficients. “Specifying Memory for Programmable Coefficients” on page 3-33 for further information.

Supported IIR Filter Types

The following IIR filter types support programmable filter coefficients:

- Second-order section (SOS) infinite impulse response (IIR) Direct Form I (`dfilt.df1sos`)
- SOS IIR Direct Form I transposed (`dfilt.df1tsos`)
- SOS IIR Direct Form II (`dfilt.df2sos`)
- SOS IIR Direct Form II transposed (`dfilt.df2tsos`)

Note: Programmable filter coefficients are supported for IIR filters with fully parallel architectures only.

Generating a Port Interface for Programmable IIR Filter Coefficients

This section describes how to use the `CoefficientSource` property to specify that a processor interface for loading coefficients is generated. You can also use the **Coefficient source** menu on the Generate HDL dialog box for this purpose.

The valid value strings for the property are:

- `'Internal'`: (Default) Do not generate a processor interface. Coefficients are obtained from the filter object and hard-coded.
- `'ProcessorInterface'`: Generate a processor interface for coefficients.

When you specify `'ProcessorInterface'`, the generated entity or module definition for the filter includes the following port definitions:

- `coeffs_in`: Input port for coefficient data
- `write_address`: Write address for coefficient memory (See also “Addressing Scheme for Loading IIR Coefficients” on page 3-42)
- `write_enable`: Write enable signal for coefficient memory
- `write_done`: Signal to indicate completion of coefficient write operation

Example

In the following command-line example, a processor interface is generated in VHDL code for an SOS IIR Direct Form II filter.

```
Fs = 48e3           % Sampling frequency
Fc = 10.8e3        % Cut-off frequency
N = 5             % Filter Order
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs)
Hd = design(f_lp,'butter','FilterStructure','df2sos')
Hd.arithmetic = 'fixed'
Hd.OptimizeScaleValues = 0 % SEE NOTE ON LIMITATIONS FOLLOWING
generatehdl(Hd, 'CoefficientSource', 'ProcessorInterface')
```

The following listing shows the VHDL entity definition generated for the filter object Hd.

```
ENTITY Hd IS
PORT( clk          : IN    std_logic;
```

```

    clk_enable      : IN    std_logic;
    reset           : IN    std_logic;
    filter_in       : IN    std_logic_vector(15 DOWNT0 0); -- sfix16_En15
    write_enable    : IN    std_logic;
    write_done      : IN    std_logic;
    write_address   : IN    std_logic_vector(4 DOWNT0 0); -- ufix5
    coeffs_in       : IN    std_logic_vector(15 DOWNT0 0); -- sfix16
    filter_out      : OUT   std_logic_vector(15 DOWNT0 0) -- sfix16_En12
    );

```

END Hd;

Limitation

When you generate a processor interface for an IIR filter, you must set the value of the filter's `OptimizeForScaleValue` from 1 to 0. For example:

```
Hd.OptimizeScaleValues = 0
```

You should then make sure the filter still has the desired response, using the `fvtool` and `filter`, commands. The disabling of `Hd.OptimizeScaleValues` may add quantization at section inputs and outputs.

Generating a Test Bench for Programmable IIR Coefficients

This section describes how to use the `TestbenchCoeffStimulus` property to specify how the test bench drives the coefficient ports. You can also use the **Coefficient stimulus** option for this purpose.

When a coefficient memory interface has been generated for a filter, the coefficient ports have associated test vectors. The `TestbenchCoeffStimulus` property determines how the test bench drives the coefficient ports.

The `TestBenchStimulus` property determines the filter input stimuli.

The `TestbenchCoeffStimulus` specified the source of coefficients used for the test bench. The valid values for `TestbenchCoeffStimulus` are:

- `[]` : Empty vector.

This is the default. When the value of `TestbenchCoeffStimulus` is unspecified (or set to the default value of `[]`) the test bench loads the coefficients from the filter object and then forces the input stimuli. This shows the response to the input stimuli and verifies that the interface writes one set of coefficients into the memory without encountering an error.

- A cell array containing the following elements:
 - `New_Hd.ScaleValues`: column vector of scale values for the IIR filter
 - `New_Hd.sosMatrix`: second-order section (SOS) matrix for the IIR filter

You can specify the elements of the cell array in the following forms:

- `{New_Hd.ScaleValues , New_Hd.sosMatrix}`
- `{New_Hd.ScaleValues ; New_Hd.sosMatrix}`
- `{New_Hd.sosMatrix , New_Hd.ScaleValues}`
- `{New_Hd.sosMatrix ; New_Hd.ScaleValues}`
- `{New_Hd.ScaleValues}`
- `{New_Hd.sosMatrix}`

In this case, the filter processes the input stimuli twice. First, the test bench loads the coefficients from the filter object and forces the input stimuli to show the response. Then, the filter loads the set of coefficients specified in the `TestbenchCoeffStimulus` cell array, and shows the response by processing the same input stimuli for a second time. In this case, the internal states of the filter, as set by the first run of the input stimulus, are retained. The test bench verifies that the interface writes two different sets of coefficients into the register file. The test bench also provides an example of how the memory interface can be used to program the filter with different sets of coefficients.

If you omit `New_Hd.ScaleValues`, the test bench uses the scale values loaded from the filter object twice. Likewise, if you omit `New_Hd.sosMatrix`, the test bench uses the SOS matrix loaded from the filter object twice.

Addressing Scheme for Loading IIR Coefficients

The following table gives the address generation scheme for the `write_address` port when loading IIR coefficients into memory. This addressing scheme allows the different types of coefficients (scale values, numerator coefficients, and denominator coefficients) to be loaded via a single port (`coeffs_in`).

Note that each type of coefficient has the same word length, but may have different fractional lengths.

The address for each coefficient is divided into two fields:

- Section address: Width is $\text{ceil}(\log_2 N)$ bits, where N is the number of sections.
- Coefficient address : Width is 3 bits

The total width of the `write_address` port is therefore $\text{ceil}(\log_2 N) + 3$ bits.

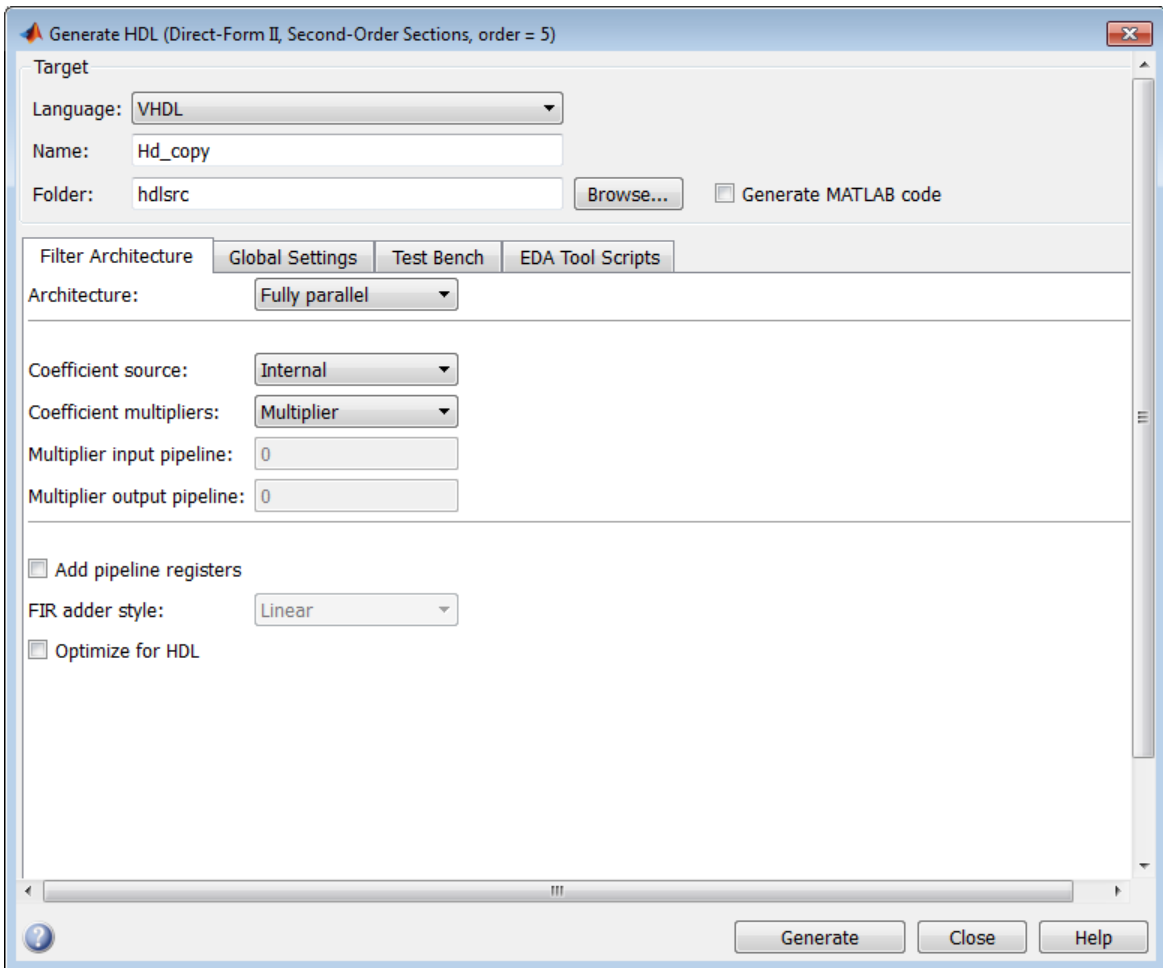
Section Address	Coefficient Address	Description
S S ... S	000	Section scale value
S S ... S	001	Numerator coefficient: b1
S S ... S	011	Numerator coefficient: b2
S S ... S	100	Numerator coefficient: b3
S S ... S	101	Denominator coefficient: a2
S S ... S	110	Denominator coefficient: a3 (if order = 2; otherwise unused)
S S ... S	110	Unused
0 0 ... 0	111	Last scale value

GUI Options for Programmable Coefficients

The following GUI options let you specify programmable coefficients:

- The **Coefficient source** list on the Generate HDL dialog box lets you select whether coefficients are obtained from the filter object and hard-coded (**Internal**), or from memory (**Processor interface**). The default is **Internal**.

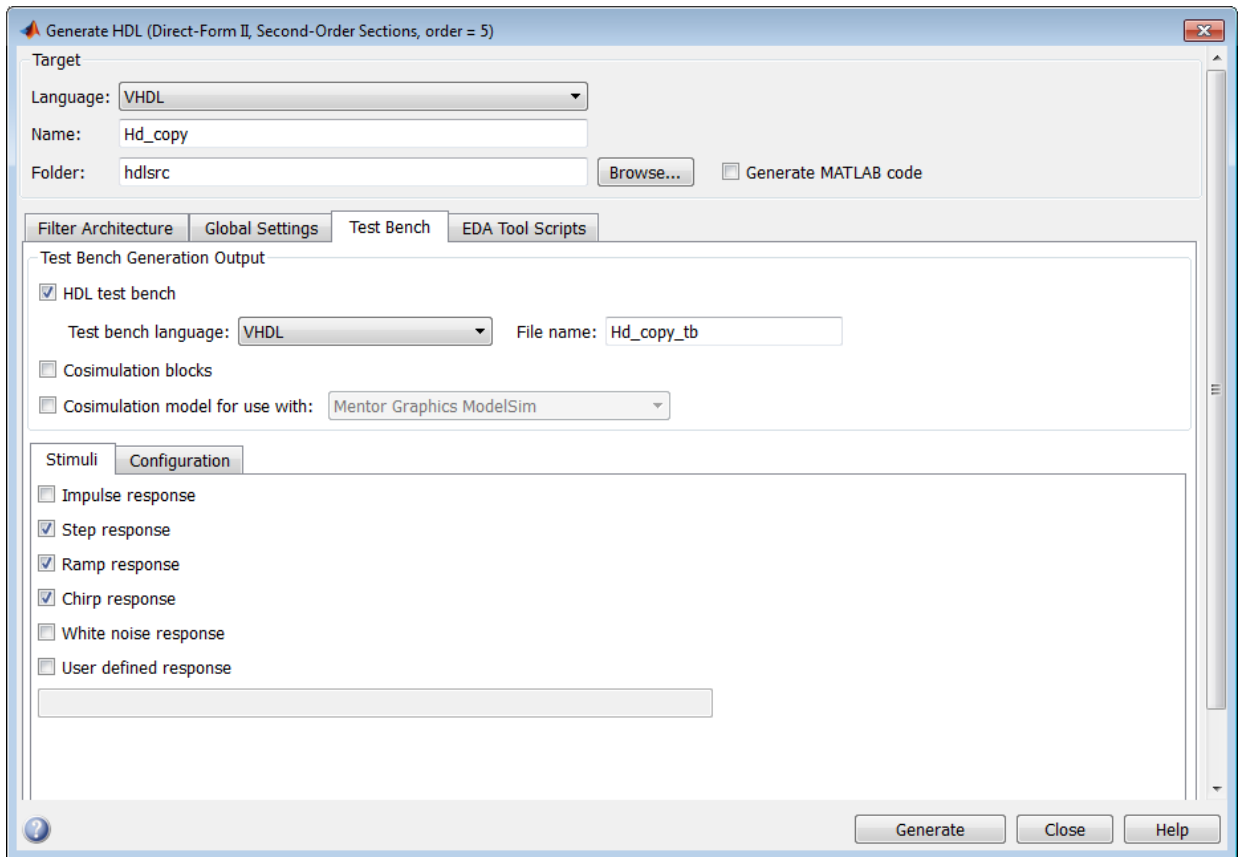
The corresponding command-line property is **CoefficientSource**. See “Generating a Port Interface for Programmable IIR Filter Coefficients” on page 3-40.



- The **Coefficient stimulus** option on the **Test Bench** pane of the Generate HDL dialog box lets you specify how the test bench tests the generated memory interface.

The corresponding command-line property is `TestbenchCoeffStimulus`. See “Generating a Test Bench for Programmable IIR Coefficients” on page 3-41.

3 HDL Code for Supported Filter Structures



DUC and DDC System Objects

You can generate HDL code for Digital Up Converter (DUC) and Digital Down Converter (DDC) System objects, giving you a direct path from System object™ to hardware. This capability is limited to code generation at the command line only.

The command line interface now includes the property `InputDataType`. This property accepts only `numerictype` values.

```
hDDC = dsp.DigitalDownConverter('Oscillator','NCO')
generatehdl(hDDC, 'InputDataType', numerictype([], 8,7))
```

The software generates a data valid signal at the top DDC or DUC level:

- For DDC, the signal is named `ce_out`. Filter Design HDL Coder™ software ties that signal to the corresponding `ce_out` signal from the decimating filtering cascade.
- For DUC, the signal is named `ce_out_valid`. The coder software ties that signal to the corresponding `ce_out_valid` signal from the interpolating filtering cascade.

Limitations

You may not set the input and output port names. These ports have the default names of `ddc_in` and `ddc_out`. Inputs and outputs are registered by the software. If you attempt to turn them off, you will get a warning.

You can implement filtering stages in DDC and DUC with the default fully parallel architecture only. For these objects, the coder software does not support optimization and architecture-specific properties such as:

- `SerialPartition`
- `DALUTPartition`
- `DARadix`
- `AddPipelineRegisters`
- `MultiplierInputPipeline`
- `MultiplierOutputPipeline`

Optimization of HDL Filter Code

- “Speed vs. Area Tradeoffs” on page 4-2
- “Distributed Arithmetic for FIR Filters” on page 4-24
- “Architecture Options for Cascaded Filters” on page 4-37
- “CSD Optimizations for Coefficient Multipliers” on page 4-39
- “Improving Filter Performance with Pipelining” on page 4-40
- “Overall HDL Filter Code Optimization” on page 4-46

Speed vs. Area Tradeoffs

In this section...

“Overview of Speed vs. Area Optimizations” on page 4-2

“Parallel and Serial Architectures” on page 4-3

“Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-6

“Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box” on page 4-11

Overview of Speed vs. Area Optimizations

The coder provides options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff, you can either specify a *fully parallel* architecture for generated HDL filter code, or choose one of several *serial* architectures. Supported architectures are described in “Parallel and Serial Architectures” on page 4-3.

The coder supports the full range of parallel and serial architecture options via properties passed in to the `generatehdl` command, as described in “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-6.

Alternatively, you can use the **Architecture** pop-up menu on the Generate HDL dialog box to choose parallel and serial architecture options, as described in “Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box” on page 4-11.

The following table summarizes the filter types that are available for parallel and serial architecture choices.

Architecture	Available for Filter Types...
Fully parallel (default)	Filter types that are supported for HDL code generation
Fully serial	<ul style="list-style-type: none"> • <code>dfilt.dffir</code> • <code>dfilt.dfsymfir</code> • <code>dfilt.dfasymfir</code> • <code>dfilt.df1sos</code> • <code>dfilt.df2sos</code>

Architecture	Available for Filter Types...
	<ul style="list-style-type: none"> • <code>mfilt.firdecim</code> • <code>mfilt.firinterp</code>
Partly serial	<ul style="list-style-type: none"> • <code>dfilt.dffir</code> • <code>dfilt.dfsymfir</code> • <code>dfilt.dfasymfir</code> • <code>dfilt.df1sos</code> • <code>dfil2.df1sos</code> • <code>mfilt.firdecim</code> • <code>mfilt.firinterp</code>
Cascade serial	<ul style="list-style-type: none"> • <code>dfilt.dffir</code> • <code>dfilt.dfsymfir</code> • <code>dfilt.dfasymfir</code>

Note: The coder also supports distributed arithmetic (DA), another highly efficient architecture for realizing filters. See “Distributed Arithmetic for FIR Filters” on page 4-24 for information about how to use this architecture.)

Parallel and Serial Architectures

Fully Parallel Architecture

This option is the default selection. A *fully parallel architecture* uses a dedicated multiplier and adder for each filter tap; the taps execute in parallel. This type of architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore consumes more chip area.

Serial Architectures

Serial architectures reuse hardware resources in time, saving chip area. The coder provides a range of serial architecture options. These architectures have a latency of one clock period (see “Latency in Serial Architectures” on page 4-5).

You can select from these serial architecture options:

- *Fully serial:* A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design would use a single multiplier and adder, executing a multiply/accumulate operation once for each tap. The multiply/accumulate section of the design runs at four times the filter's input/output sample rate. This type of architecture saves area at the cost of some speed loss and higher power consumption.

In a fully serial architecture, the system clock runs at a much higher rate than the sample rate of the filter. Thus, for a given filter design, the maximum speed achievable by a fully serial architecture will be less than that of a parallel architecture.

- *Partly serial:* Partly serial architectures cover the full range of speed vs. area tradeoffs that lie between fully parallel and fully serial architectures.

In a partly serial architecture, the filter taps are grouped into a number of serial partitions. The taps within each partition execute serially, but the partitions execute together in parallel. The outputs of the partitions are summed at the final output.

When you select a partly serial architecture for a filter, you can define the serial partitioning in the following ways:

- Define the serial partitions directly, as a vector of integers. Each element of the vector specifies the length of the corresponding partition.
- Specify the desired hardware folding factor *ff*, an integer greater than 1. Given the folding factor, the coder computes the serial partition and the number of multipliers.
- Specify the desired number of multipliers *nmults*, an integer greater than 1. Given the number of multipliers, the coder computes the serial partition and the folding factor.

The Generate HDL dialog box lets you specify a partly serial architecture in terms of these three parameters. You can then view how a change in one parameter interacts with the other two. The coder also provides `hdlfilterserialinfo`, an informational function that helps you define an optimal serial partition for a filter.

- *Cascade-serial:* A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are grouped into a number of serial partitions that execute together in parallel. However, the accumulated output of each partition cascades to the accumulator of the previous partition. The output of the partitions is therefore computed at the accumulator of the

first partition. This technique is termed *accumulator reuse*. You do not require a final adder, which saves area.

The cascade-serial architecture requires an extra cycle of the system clock to complete the final summation to the output. Therefore, the frequency of the system clock must be increased slightly with respect to the clock used in a noncascade partly serial architecture.

To generate a cascade-serial architecture, you specify a partly serial architecture with accumulator reuse enabled. (See “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-6.) If you do not specify the serial partitions, the coder automatically selects an optimal partitioning.

Latency in Serial Architectures

Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to sequentially add the products. An additional final register is used to store the summed result of each of the serial partitions. The operation requires an extra clock cycle.

Holding Input Data in a Valid State

Serial filters allow data to be delivered to the outputs N cycles ($N \geq 2$) later than the inputs. Using the **Hold input data between samples** test bench option (or the `HoldInputDataBetweenSamples` CLI property), you can determine how long (in terms of clock cycles) input data values are held in a valid state, as follows:

- When you select **Hold input data between samples** (the default), input data values are held in a valid state across N clock cycles.
- When you clear **Hold input data between samples**, data values are held in a valid state for only one clock cycle. For the next $N - 1$ cycles, data is in an unknown state (expressed as 'X') until the next input sample is clocked in.

The following figure shows the **Test Bench** pane of the Generate HDL dialog box with **Hold input data between samples** set to its default setting.

Filter Architecture Global Settings Test Bench EDA Tool Scripts

Test Bench Generation Output

HDL test bench

Test bench language: VHDL File name: Hd_copy_tb

Cosimulation blocks

Cosimulation model for use with: Mentor Graphics ModelSim

Stimuli Configuration

Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

Force clock enable

Clock enable delay (in clock cycles): 1

Force reset

Reset length (in clock cycles): 2

Hold input data between samples

Initialize test bench inputs

Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Simulator flags:

See also `HoldInputDataBetweenSamples`

Specifying Speed vs. Area Tradeoffs via `generatehdl` Properties

By default, `generatehdl` generates filter code using a fully parallel architecture. If you want to generate filter code with a fully parallel architecture, you do not have to specify this architecture explicitly.

Two properties let you specify serial architecture options when generating code via `generatehdl`:

- `'SerialPartition'`: This property specifies the serial partitioning of the filter.
- `'ReuseAccum'`: This property enables or disables accumulator reuse.

The table below summarizes how to set these properties to generate the desired architecture. The table is followed by several examples.

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Fully parallel	Omit this property	Omit this property
Fully serial	N, where N is the length of the filter	Not specified, or 'off'
Partly serial	<p>[p1 p2 p3 . . . pN] : a vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. When you define the partitioning for a partly serial architecture, consider the following:</p> <ul style="list-style-type: none"> • The filter length should be divided as uniformly as you can into a vector of length equal to the number of multipliers intended. For example, if your design requires a filter of length 9 with 2 multipliers, the recommended partition is [5 4]. If your design requires 3 multipliers, the recommended partition is [3 3 3] rather than some less uniform division such as [1 4 4] or [3 4 2]. • If your design is constrained by having to compute each output value (corresponding to each input value) in an exact number N of clock cycles, use N as the largest partition size and partition the other elements as uniformly as you can. For example, if the filter length is 9 and your design requires exactly 4 cycles to compute the output, define the partition as [4 3 2]. This partition executes in 4 clock cycles, at the cost of 3 multipliers. <p>You can also specify a serial architecture in terms of a desired hardware folding factor, or in terms of the optimal number of multipliers. See <code>hdlfilterserialinfo</code> for detailed information.</p>	'off'

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Cascade-serial with explicitly specified partitioning	[p1 p2 p3...pN]: a vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must equal the length of the filter. The values of the vector elements must appear in descending order, except that the last two elements must be equal. For example, for a filter of length 9, partitions such as [5 4] or [4 3 2] would be legal, but the partitions [3 3 3] or [3 2 4] would raise an error at code generation time.	'on'
Cascade-serial with automatically optimized partitioning	Omit this property	'on'

Specifying Parallel and Serial FIR Filter Architectures in generatehdl

The following examples show the use of the 'SerialPartition' and 'ReuseAccum' properties in generating code with the generatehdl function. The following examples assume that a direct-form FIR filter has been created in the workspace as follows:

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
```

This example generates a fully parallel architecture (by default).

```
generatehdl(Hd, 'Name', 'FullyParallel')

### Starting VHDL code generation process for filter: FullyParallel
### Generating: D:\Work\test\hdlsrc\FullyParallel.vhd
### Starting generation of FullyParallel VHDL entity
### Starting generation of FullyParallel VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: FullyParallel
```

This example generates a fully serial architecture. Notice that the system clock rate is nine times the filter's sample rate. Also, the HDL latency reported is one sample greater than in the previous (parallel) example.

```
generatehdl(Hd, 'SerialPartition',9, 'Name', 'FullySerial')
```

```

### Starting VHDL code generation process for filter: FullySerial
### Generating: D:\Work\test\hdlsrc\FullySerial.vhd
### Starting generation of FullySerial VHDL entity
### Starting generation of FullySerial VHDL architecture
### Clock rate is 9 times the input sample rate for this architecture.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: FullySerial

```

This example generates a partly serial architecture with three equal partitions.

```

generatehdl(Hd, 'SerialPartition',[3 3 3], 'Name', 'PartlySerial')

### Starting VHDL code generation process for filter: PartlySerial
### Generating: D:\Work\test\hdlsrc\PartlySerial.vhd
### Starting generation of PartlySerial VHDL entity
### Starting generation of PartlySerial VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: PartlySerial

```

This example generates a cascade-serial architecture with three partitions. The partitions appear in descending order of size. Notice that the clock rate is higher than that in the previous (partly serial without accumulator reuse) example.

```

generatehdl(Hd, 'SerialPartition',[4 3 2], 'ReuseAccum', 'on', 'Name', 'CascadeSerial')

### Starting VHDL code generation process for filter: CascadeSerial
### Generating: D:\Work\test\hdlsrc\CascadeSerial.vhd
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: CascadeSerial

```

This example generates a cascade-serial architecture, with the partitioning automatically determined by the coder .

```

generatehdl(Hd, 'ReuseAccum','on', 'Name', 'CascadeSerial')

### Starting VHDL code generation process for filter: CascadeSerial
### Generating: D:\Work\test\hdlsrc\CascadeSerial.vhd
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### Serial partition # 1 has 4 inputs.
### Serial partition # 2 has 3 inputs.
### Serial partition # 3 has 2 inputs.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: CascadeSerial

```

Serial Partitions for Cascaded Filters

Note: Filter Design HDL Coder software supports this feature for the command-line interface (`generatehdl`) only.

To specify serial partitioning for one or more cascade stages, use the `SerialPartition` property. The following example defines different serial partitions for each filter in a two-stage cascade. The partition vectors are contained within a cell array.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
Hp = design(fdesign.highpass('N,Fc',8,.4))
Hp.arithmetic = 'fixed'
Hc = cascade(Hd,Hp)
generatehdl(Hc,'SerialPartition',{[5 4],[8 1]})
```

Tip Use the `hdlfilterserialinfo` function to display the effective filter length and partitioning options for each filter stage of a cascade.

Serial Architectures for IIR SOS Filters

To specify a partly or fully serial architecture for an IIR SOS filter structure, specify either one of the following parameters:

- `'FoldingFactor'`, `ff`: Specify the desired hardware folding factor ff , an integer greater than 1. Given the folding factor, the coder computes the number of multipliers.
- `'NumMultipliers'`, `nmults`: Specify the desired number of multipliers $nmults$, an integer greater than 1. Given the number of multipliers, the coder computes the folding factor.

To obtain information about the folding factor options and the corresponding number of multipliers for a filter, call the `hdlfilterserialinfo` function. The following example creates a Direct Form I SOS (`df1sos`) filter and the calls `hdlfilterserialinfo`.

```
Fs = 48e3           % Sampling frequency
Fc = 10.8e3        % Cut-off frequency
N = 5              % Filter Order
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs)
Hd = design(f_lp,'butter','FilterStructure','df1sos')
Hd.arithmetic = 'fixed'
hdlfilterserialinfo(Hd)
```

Table of folding factors with corresponding number of multipliers for the given filter.

Folding Factor	Multipliers
6	3
9	2
18	1

The following example generates HDL code for the `df1s0s` filter, specifying a folding factor of 18.

```
generatehdl(Hd, 'FoldingFactor',18)

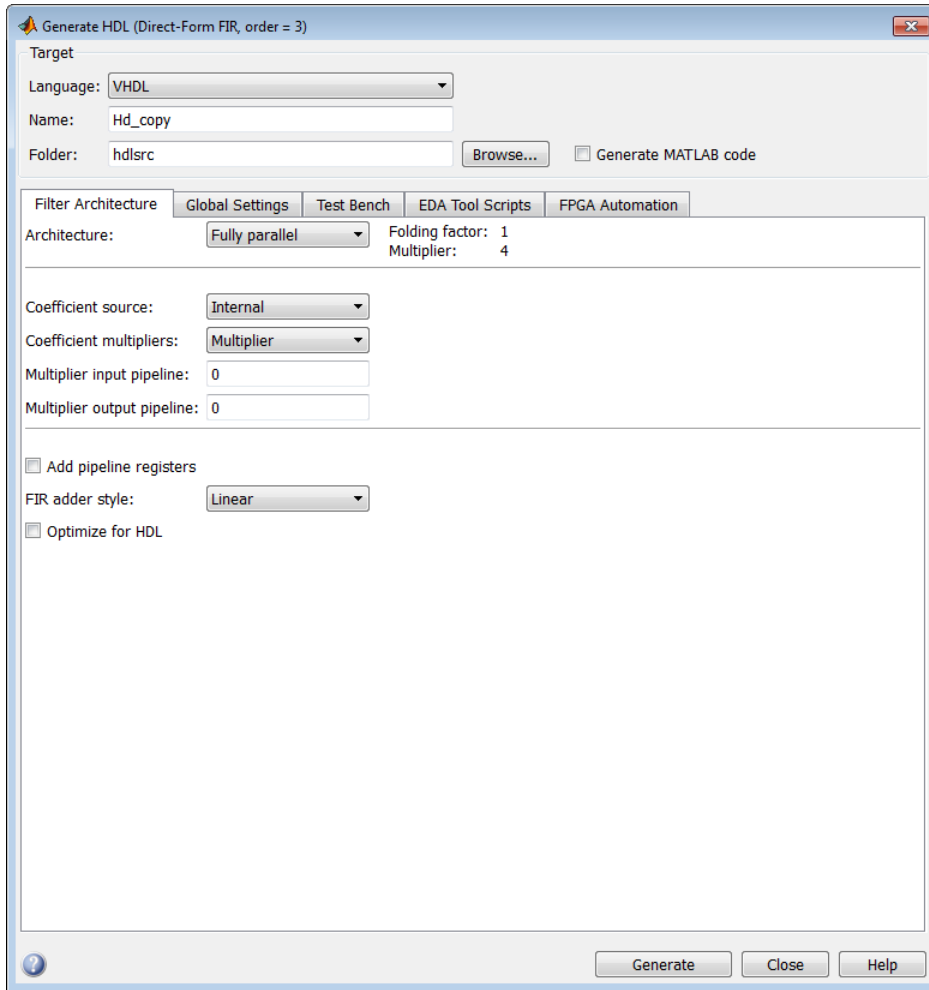
### Starting VHDL code generation process for filter: Hd
### Starting VHDL code generation process for filter: Hd
### Generating: c:\work\hdlsrc\Hd.vhd
### Starting generation of Hd VHDL entity
### Starting generation of Hd VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: Hd
```

Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box

The **Architecture** pop-up menu, located on the Generate HDL dialog box, lets you select parallel and serial architecture options corresponding to those described in “Parallel and Serial Architectures” on page 4-3. The following topics describe the GUI options you must set for each **Architecture** choice.

Specifying a Fully Parallel Architecture

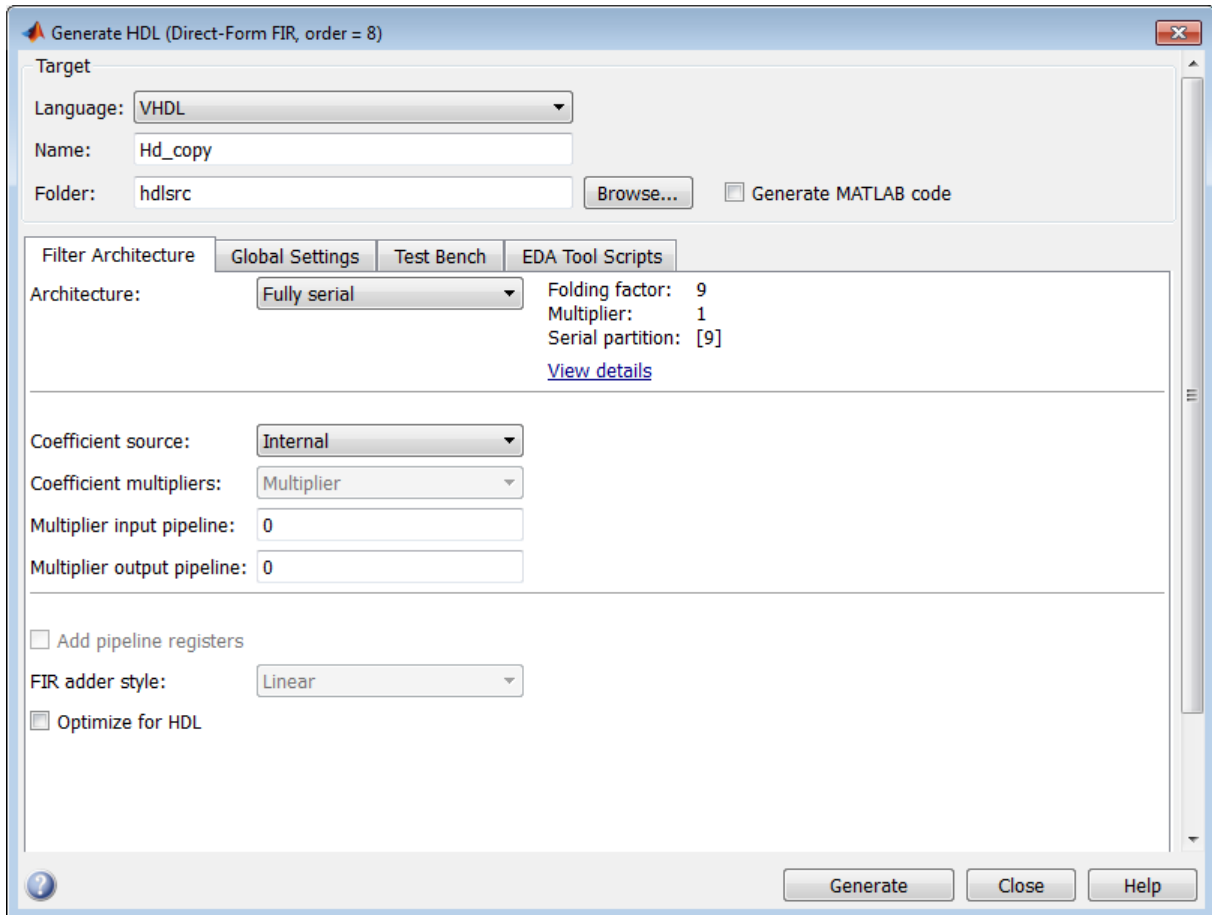
The default **Architecture** setting is Fully parallel, as shown in the following figure.



Specifying a Fully Serial Architecture

When you select the **Fully serial, Architecture** options, the Generate HDL dialog box displays additional information about the filters's folding factor, number of multipliers, and serial partitioning. Because these parameters are determined by the length of the filter, they display in a read-only format, as shown in the following figure.

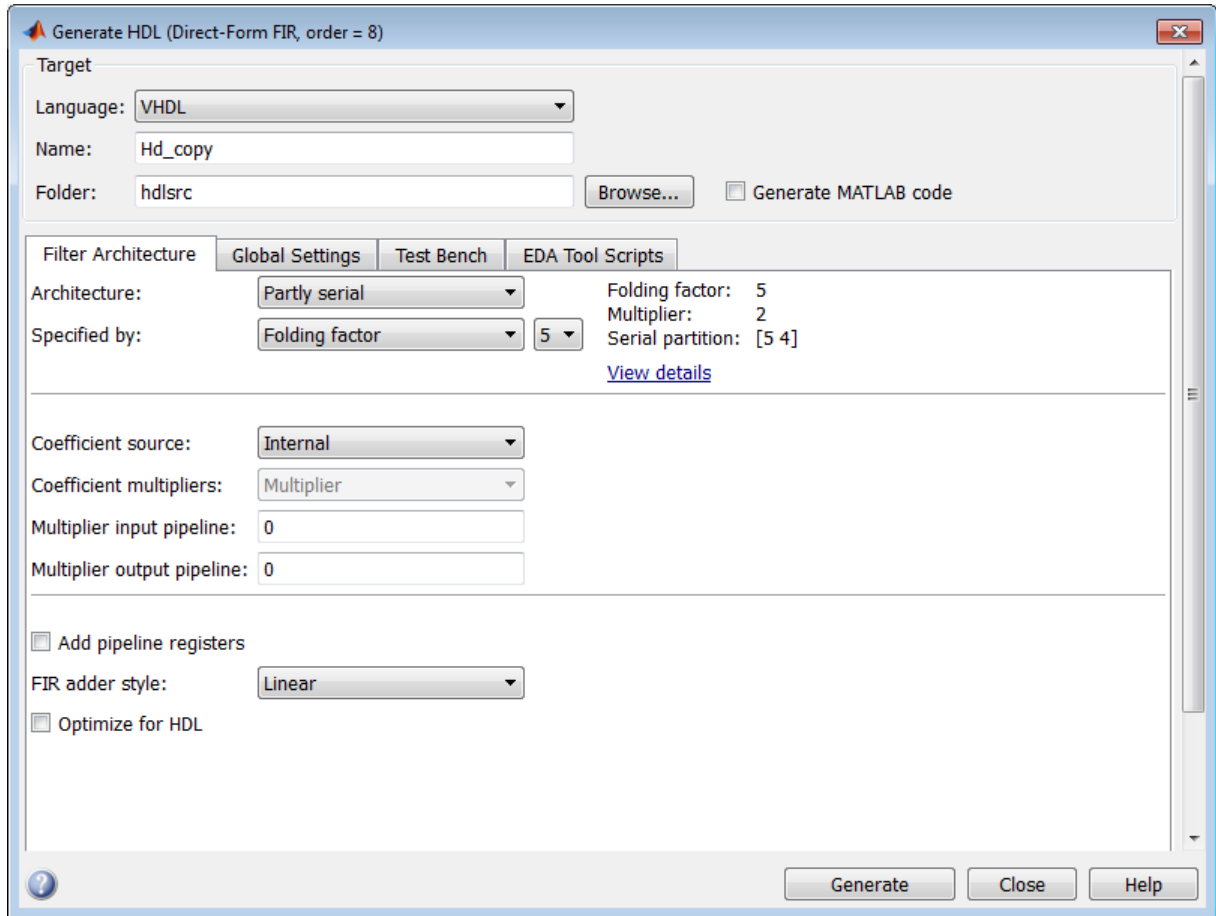
The Generate HDL dialog box also displays a **View details** link. When you click on this link, the coder displays an HTML report in a separate window. The report displays an exhaustive table of folding factor, multiplier, and serial partition settings for the current filter. You can use the table to help you choose optimal settings for your design.



Specifying Partitions for a Partly Serial Architecture

When you select the **Partly serial Architecture** option, the Generate HDL dialog box displays additional information and data entry fields related to serial partitioning. (See the following figure.)

The Generate HDL dialog box also displays a **View details** link. When you click this link, the coder displays an HTML report in a separate window. The report displays an exhaustive table of folding factor, multiplier, and serial partition settings for the current filter. You can use the table to help you choose optimal settings for your design.



The **Specified by** drop-down menu lets you decide how you define the partly serial architecture. Select one of the following options:

- **Folding factor:** The drop-down menu to the right of **Folding factor** contains an exhaustive list of folding factors for the filter. When you select a value, the display of the current folding factor, multiplier, and serial partition settings updates.

Filter Architecture Global Settings Test Bench EDA Tool Scripts

Architecture: Partly serial

Specified by: Folding factor 26

1
2
3
4
5
6
7

Folding factor: 26
Multiplier: 2
Serial partition: [26 25]

[View details](#)

Coefficient source: Internal

Coefficient multipliers: Multiplier

- **Multipliers:** The drop-down menu to the right of **Multipliers** contains an exhaustive list of value options for the number of multipliers for the filter. When you select a value, the display of the current folding factor, multiplier, and serial partition settings updates.

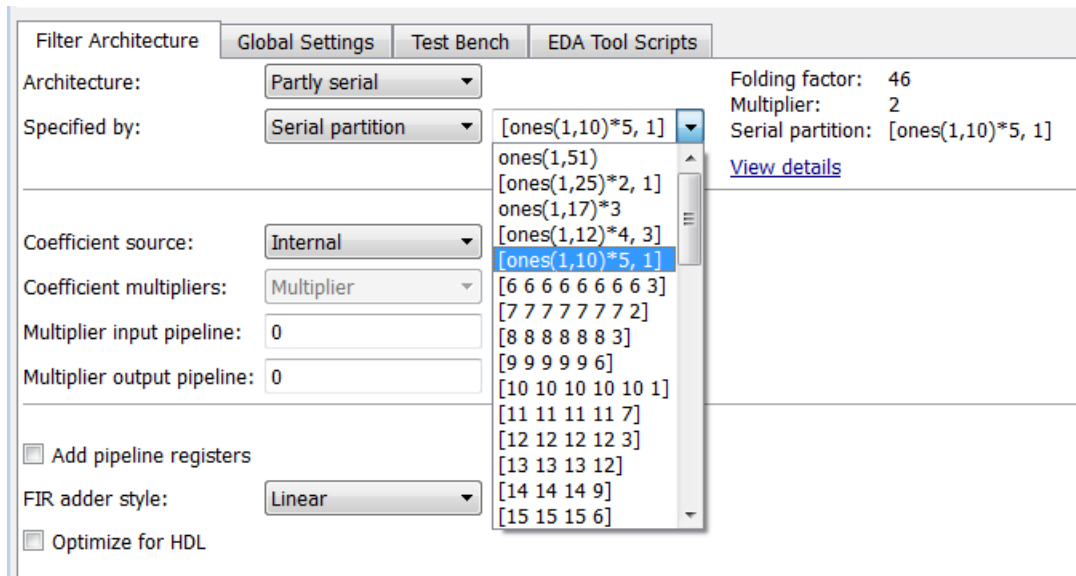
The screenshot shows the 'Global Settings' tab of the HDL Filter Code configuration window. The 'Specified by' dropdown menu is open, displaying a list of multiplier options from 1 to 51. The current settings are as follows:

Parameter	Value
Architecture	Partly serial
Specified by	Multipliers
Folding factor	5
Multiplier	11
Serial partition	[ones(1,10)*5, 1]

Other settings visible in the window include:

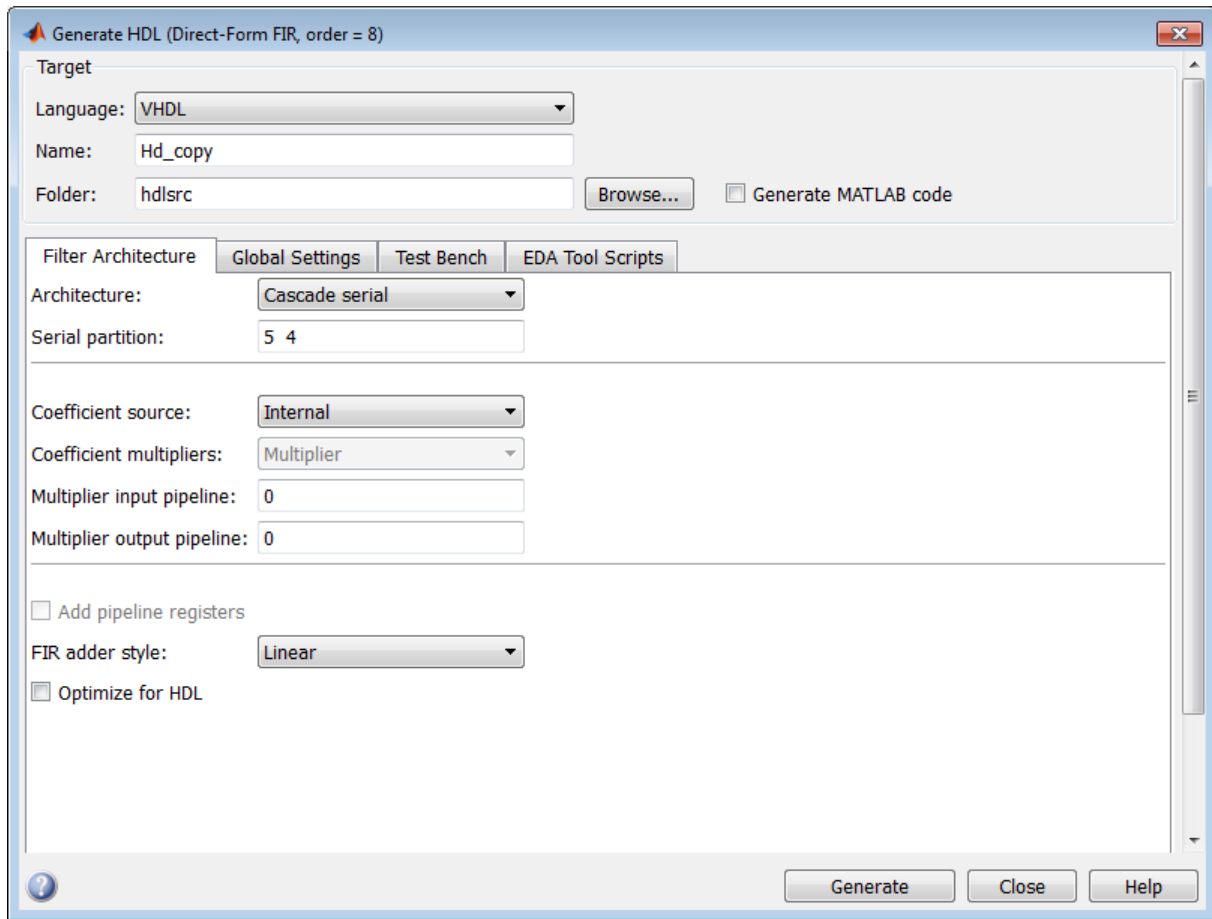
- Coefficient source: Internal
- Coefficient multipliers: Multiplier
- Multiplier input pipeline: 0
- Multiplier output pipeline: 0
- Add pipeline registers
- FIR adder style: Linear
- Optimize for HDL

- **Serial partition:** The drop-down menu to the right of **Serial partition** contains an exhaustive list of serial partition options for the filter. When you select a value, the display of the current folding factor, multiplier, and serial partition settings updates.



Specifying a Cascade Serial Architecture

When you select the **Cascade serial Architecture** option, the Generate HDL dialog box displays the **Serial partition** field, as shown in the following figure.



The **Specified by** menu lets you define the number and size of the serial partitions according to different criteria, as described in “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-6.

Specifying Serial Architectures for IIR SOS Filters

To specify a partly or fully serial architecture for an IIR SOS filter structure in the GUI, you set the following options:

- **Architecture:** Select Fully parallel (the default), Fully serial, or Partly serial.

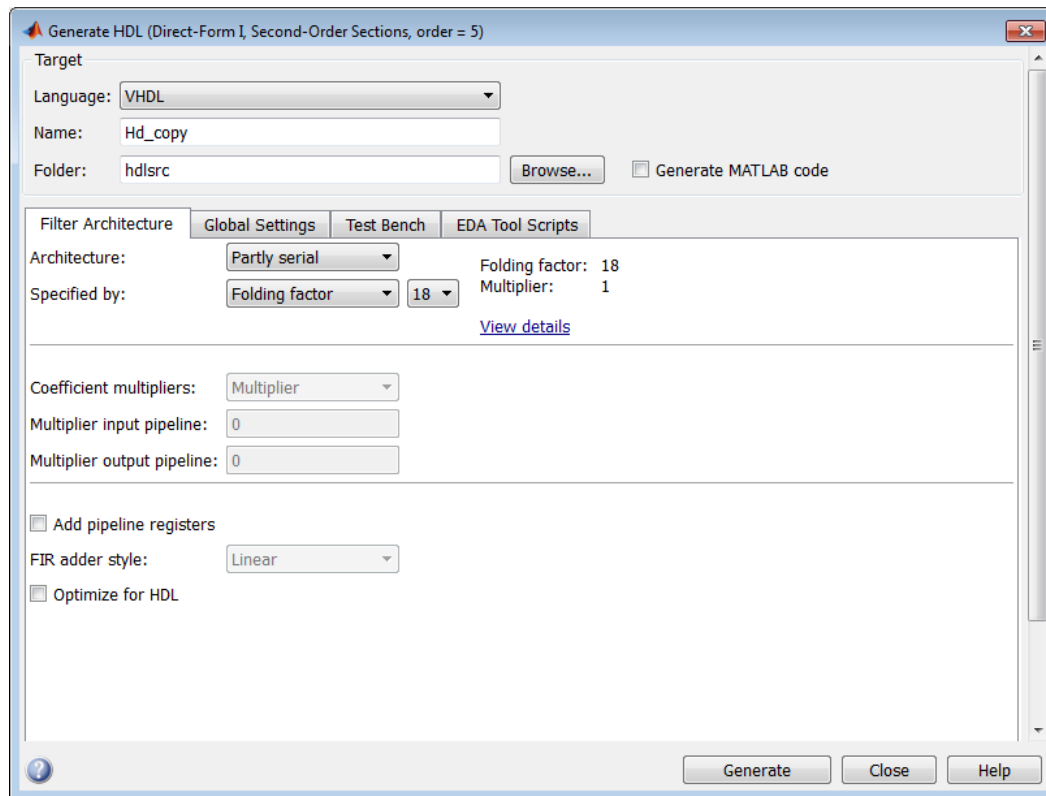
If you select **Partly serial**, the GUI displays the **Specified by** drop-down menu.

- **Specified by:** Select one of the following:
 - **Folding factor:** Specify the desired hardware folding factor ff , an integer greater than 1. Given the folding factor, the coder computes the number of multipliers.
 - **Multipliers:** Specify the desired number of multipliers $nmults$, an integer greater than 1. Given the number of multipliers, the coder computes the folding factor.

Example: Direct Form I SOS (df1sos) Filter

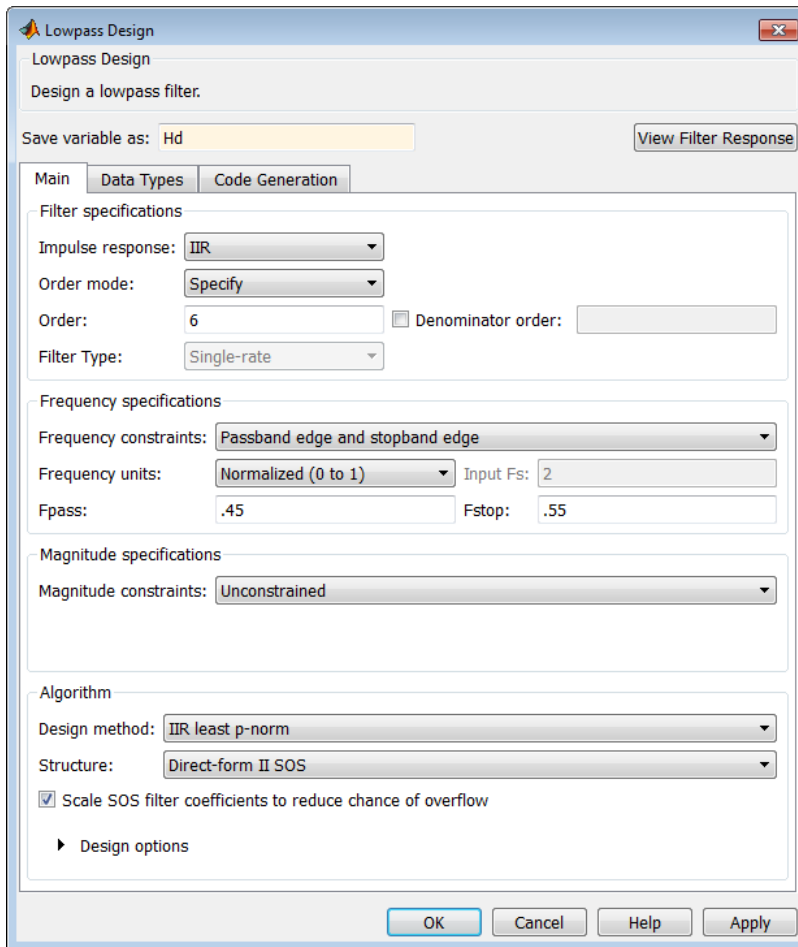
The following example creates a Direct Form I SOS (df1sos) filter design and opens the GUI. The figure following the code example shows the coder options configured for a partly serial architecture specified with a **Folding factor** of 18.

```
Fs = 48e3           % Sampling frequency
Fc = 10.8e3        % Cut-off frequency
N = 5              % Filter Order
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs)
Hd = design(f_lp,'butter','FilterStructure','df1sos')
Hd.arithmetic = 'fixed'
fdhdltool(Hd)
```



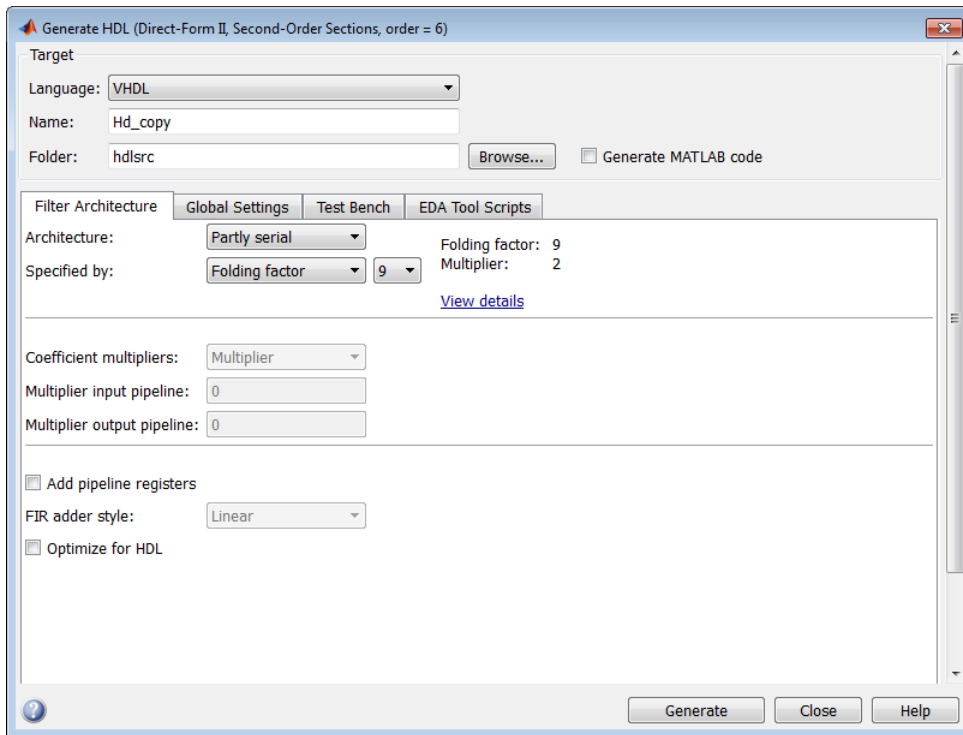
Example: Direct Form II SOS (df2sos) Filter

The following example creates a Direct Form II SOS (df2sos) filter design using filterbuilder.



The filter is a lowpass `df2sos` filter with a filter order of 6. The filter arithmetic is set to Fixed-point.

On the **Code Generation** tab, the **Generate HDL** button activates the Filter Design HDL Coder GUI. The following figure shows the HDL coder options configured for this filter, using party serial architecture with a **Folding** factor of 9.



Specifying a Distributed Arithmetic Architecture

The **Architecture** pop-up menu also includes the **Distributed arithmetic (DA)** option. See “Distributed Arithmetic for FIR Filters” on page 4-24) for information about this architecture.

Interactions Between Architecture Options and Other HDL Options

Selecting some **Architecture** menu options may change or disable other options.

- When the **Fully serial** option is selected, the following options are set to their default values and disabled:
 - **Coefficient multipliers**
 - **Add pipeline registers**
 - **FIR adder style**

- When the **Partly serial** option is selected:
 - The **Coefficient multipliers** option is set to its default value and disabled.
 - If the filter is multirate, the **Clock inputs** options is set to **Single** and disabled.
- When the **Cascade serial** option is selected, the following options are set to their default values and disabled:
 - **Coefficient multipliers**
 - **Add pipeline registers**
 - **FIR adder style**

Distributed Arithmetic for FIR Filters

In this section...

“Distributed Arithmetic Overview” on page 4-24

“Requirements and Considerations for Generating Distributed Arithmetic Code” on page 4-26

“DALUTPartition Property” on page 4-27

“DARadix Property” on page 4-29

“Specifying Distributed Arithmetic for Cascaded Filters” on page 4-30

“Special Cases” on page 4-31

“Distributed Arithmetic Options in the Generate HDL Dialog Box” on page 4-31

Distributed Arithmetic Overview

Distributed Arithmetic (DA) is a widely-used technique for implementing sum-of-products computations without the use of multipliers. Designers frequently use DA to build efficient Multiply-Accumulate Circuitry (MAC) for filters and other DSP applications.

The main advantage of DA is its high computational efficiency. DA distributes multiply and accumulate operations across shifters, lookup tables (LUTs) and adders in such a way that conventional multipliers are not required.

The coder supports DA in HDL code generated for several single-rate and multirate FIR filter structures for fixed-point filter designs. (See “Requirements and Considerations for Generating Distributed Arithmetic Code” on page 4-26.)

This section briefly summarizes of the operation of DA. Detailed discussions of the theoretical foundations of DA appear in the following publications:

- Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Second Edition, Springer, pp 88–94, 128–143.
- White, S.A., *Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review*. IEEE ASSP Magazine, Vol. 6, No. 3 .

In a DA realization of a FIR filter structure, a sequence of input data words of width W is fed through a parallel to serial shift register. This feedthrough produces a serialized

stream of bits. The serialized data is then fed to a bit-wide shift register. This shift register serves as a delay line, storing the bit serial data samples.

The delay line is tapped (based on the input word size W), to form a W -bit address that indexes into a lookup table (LUT). The LUT stores the possible sums of partial products over the filter coefficients space. The LUT is followed by a shift and adder (scaling accumulator) that adds the values obtained from the LUT sequentially.

A table lookup is performed sequentially for each bit (in order of significance starting from the LSB). On each clock cycle, the LUT result is added to the accumulated and shifted result from the previous cycle. For the last bit (MSB), the table lookup result is subtracted, accounting for the sign of the operand.

This basic form of DA is fully serial, operating on one bit at a time. If the input data sequence is W bits wide, then a FIR structure takes W clock cycles to compute the output. Symmetric and asymmetric FIR structures are an exception, requiring $W+1$ cycles, because one additional clock cycle is required to process the carry bit of the preadders.

Improving Performance with Parallelism

The inherently bit serial nature of DA can limit throughput. To improve throughput, the basic DA algorithm can be modified to compute more than one bit sum at a time. The number of simultaneously computed bit sums is expressed as a power of two called the *DA radix*. For example, a DA radix of 2 (2^1) indicates that one bit sum is computed at a time; a DA radix of 4 (2^2) indicates that two bit sums are computed at a time, and so on.

To compute more than one bit sum at a time, the LUT is replicated. For example, to perform DA on 2 bits at a time (radix 4), the odd bits are fed to one LUT and the even bits are simultaneously fed to an identical LUT. The LUT results corresponding to odd bits are left-shifted before they are added to the LUT results corresponding to even bits. This result is then fed into a scaling accumulator that shifts its feedback value by 2 places.

Processing more than one bit at a time introduces a degree of parallelism into the operation, possibly improving performance at the expense of area. The *DARadix* property lets you specify the number of bits processed simultaneously in DA (see “*DARadix* Property” on page 4-29).

Reducing LUT Size

The size of the LUT grows exponentially with the order of the filter. For a filter with N coefficients, the LUT must have 2^N values. For higher-order filters, LUT size must

be reduced to reasonable levels. To reduce the size, you can subdivide the LUT into a number of LUTs, called *LUT partitions*. Each LUT partition operates on a different set of taps. The results obtained from the partitions are summed.

For example, for a 160 tap filter, the LUT size is $(2^{160}) * W$ bits, where W is the word size of the LUT data. Dividing this value into 16 LUT partitions, each taking 10 inputs (taps), the total LUT size is reduced to $16 * (2^{10}) * W$ bits, a significant reduction.

Although LUT partitioning reduces LUT size, you must have more adders to sum the LUT data.

The `DALUTPartition` property lets you specify how the LUT is partitioned in DA (see “`DALUTPartition` Property” on page 4-27).

Requirements and Considerations for Generating Distributed Arithmetic Code

The coder lets you control how DA code is generated using the `DALUTPartition` and `DARadix` properties (or equivalent Generate HDL dialog box options). Before using these properties, review the following general requirements, restrictions, and other considerations for generation of DA code.

Supported Filter Types

The coder supports DA in HDL code generated for the following single-rate and multirate FIR filter structures:

- `dfilt.dffir`
- `dfilt.dfsymfir`
- `dfilt.dfasymfir`
- `mfilt.firdecim`
- `mfilt.firinterp`

Requirements Specific to Filter Type

The `DALUTPartition` and `DARadix` properties have certain requirements and restrictions that are specific to different filter types. These requirements are included in the discussions of each property:

- “`DALUTPartition` Property” on page 4-27

- “DARadix Property” on page 4-29

Fixed-Point Quantization Required

Generation of DA code is supported only for fixed-point filter designs.

Specifying Filter Precision

The data path in HDL code generated for the DA architecture is carefully optimized for full precision computations. The filter result is cast to the output data size only at the final stage when it is presented to the output. If you leave the `FilterInternals` property set to the default, (`FullPrecision`), numeric results obtained from simulation of the generated HDL code are bit-true to filter results produced by the original filter object.

If the `FilterInternals` property is set to `SpecifyPrecision` and you change filter word or fraction lengths, generated DA code may produce numeric results that are different than the filter results produced by the original filter object.

DALUTPartition Property

Syntax: `'DALUTPartition', [p1 p2... pN]`

`DALUTPartition` enables DA code generation and specifies the number and size of LUT partitions used for DA.

Specify LUT partitions as a vector of integers `[p1 p2... pN]` where

- `N` is the number of partitions.
- Each vector element specifies the size of a partition. The maximum size for an individual partition is 12.
- The sum of the vector elements equals the filter length `FL`. `FL` is calculated differently depending on the filter type. For more information, see:
 - “Specifying `DALUTPartition` for Single-Rate Filters” on page 4-28
 - “Specifying `DALUTPartition` for Multirate Filters” on page 4-28

To enable generation of DA code for your filter design without LUT partitioning, specify a vector of one element, whose value equals the filter length, as in the following example:

```
filtDES = fdesign.lowpass('N,Fc,Ap,Ast',4,0.4,0.05,0.03,'linear')
```

```
Hd = design(filtDES)
Hd.arithmetic = 'fixed'
generatehdl(Hd, 'DALUTPartition', 5)
```

Specifying DALUTPartition for Single-Rate Filters

To determine the LUT partition for one of the supported single-rate filter types, calculate FL as shown in the following table. Then, specify the partition as a vector whose elements sum to FL.

Filter Type	Filter Length (FL) Calculation
dfilt.dffir	FL = length(find(Hd.numerator~= 0))
dfilt.dfsymfir dfilt.dfasymfir	FL = ceil(length(find(Hd.numerator~= 0))/2)

The following example shows the FL calculation and one partitioning option for a direct form FIR filter:

```
filtDES = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear')
Hd = design(filtDES,'filterstructure','dffir')
Hd.arithmetic = 'fixed'
FL = length(find(Hd.numerator~= 0))

FL =

    31

generatehdl(Hd, 'DALUTPartition',[8 8 8 7])
```

The following example shows the FL calculation and one partitioning option for a direct-form symmetric FIR filter:

```
filtDES = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear')
Hd = design(filtDES,'filterstructure','dfsymfir')
Hd.arithmetic = 'fixed'
FL = ceil(length(find(Hd.numerator~= 0))/2)

FL =

    16

generatehdl(Hd, 'DALUTPartition',[8 8])
```

Specifying DALUTPartition for Multirate Filters

For supported multirate filters (`mfilt.firdecim` and `mfilt.firinterp`), you can specify the LUT partition as

- A vector defining a partition for LUTs for the polyphase subfilters.

- A matrix of LUT partitions, where each row vector specifies a LUT partition for a corresponding polyphase subfilter. In this case, the FL is uniform for the subfilters. This approach provides a fine control for partitioning each subfilter.

The following table shows the FL calculations for each type of LUT partition.

LUT Partition Specified As...	Filter Length (FL) Calculation
<i>Vector</i> : determine FL as shown in the Filter Length (FL) Calculation column to the right. Specify the LUT partition as a vector of integers whose elements sum to FL.	$FL = \text{size}(\text{polyphase}(H_m), 2)$
<i>Matrix</i> : determine the subfilter length FL_i based on the polyphase decomposition of the filter, as shown in the Filter Length (FL) Calculation column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to FL_i .	$p = \text{polyphase}(H_m)$ $FL_i = \text{length}(\text{find}(p(i, :)))$ where i is the index to the i th row of the polyphase matrix of the multirate filter. The i th row of the matrix p represents the i th subfilter.

The following example shows the FL calculation for a direct-form FIR polyphase decimator, with the LUT partition specified as a vector:

```
d = fdesign.decimator(4)
Hm = design(d)
Hm.arithmetic = 'fixed'
FL = size(polyphase(Hm),2)
```

```
FL =
```

```
    27
```

```
generatehdl(Hm, 'DALUTPartition',[8 8 8 3])
```

The following example shows the LUT partition specified as a matrix for the same direct-form FIR polyphase decimator. The length of the first subfilter is 1, and the other subfilters have length 26.

```
d = fdesign.decimator(4)
Hm = design(d)
Hm.arithmetic = 'fixed'
generatehdl(Hm, 'DALUTPartition',[1 0 0 0; 8 8 8 2; 8 8 6 4; 8 8 8 2])
```

DARadix Property

Syntax: 'DARadix', N

DARadix specifies the number of bits processed simultaneously in DA. The number of bits is expressed as N, which must be

- A nonzero positive integer that is a power of two
- Such that $\text{mod}(W, \log_2(N)) = 0$ where W is the input word size of the filter.

The default value for N is 2, specifying processing of one bit at a time, or fully serial DA, which is slow but low in area. The maximum value for N is 2^W , where W is the input word size of the filter. This maximum specifies fully parallel DA, which is fast but high in area. Values of N between these extrema specify partly serial DA.

Note: When setting a DARadix value for symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 4-31.

Specifying Distributed Arithmetic for Cascaded Filters

Note: Filter Design HDL Coder software supports this feature for the command-line interface (`generatehdl`) only.

To specify a DA architecture for one or more cascade stages, you use the `DALUTPartition` property, optionally specifying the `DARadix` property. The following example defines LUT partitions for each filter in a two-stage cascade. The partition vectors are contained within a cell array.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
Hp = design(fdesign.highpass('N,Fc',8,.4))
Hp.arithmetic = 'fixed'
Hc = cascade(Hd,Hp)
generatehdl(Hc,'DALUTPartition',{[5 4],[3 3 3]})
```

The following example defines LUT partitions and `DARadix` values for each filter in a two-stage cascade. The partition vectors are contained within a cell array.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
Hp = design(fdesign.highpass('N,Fc',8,.4))
```



```

Hp.arithmetic = 'fixed'
Hc = cascade(Hd,Hp)
generatehdl(Hc, 'DALUTPartition', {[5 4],[3 3 3]}, 'DARadix', {2^8,2^4})

```

Tip Use the `hdlfilterdainfo` function to display the effective filter length and LUT partitioning options and `DARadix` values for each filter stage of a cascade.

Special Cases

Coefficients with Zero Values

DA ignores taps that have zero-valued coefficients and reduces the size of the DA LUT accordingly.

Considerations for Symmetrical and Asymmetrical Filters

For symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters:

- A bit-level preadder or presubtractor is required to add tap data values that have coefficients of equal value and/or opposite sign. One extra clock cycle is required to compute the result because of the additional carry bit.
- The coder takes advantage of filter symmetry. This symmetry reduces the DA LUT size substantially, because the effective filter length for these filter types is halved.

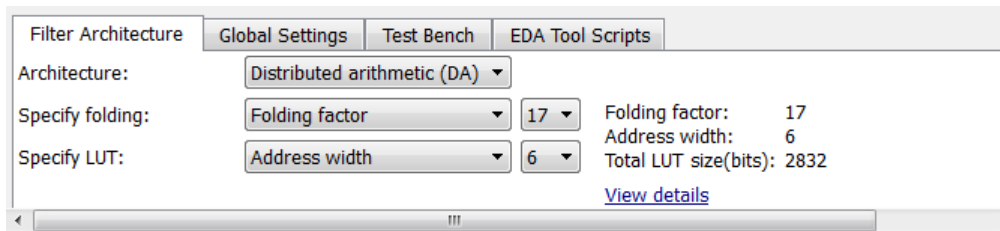
Holding Input Data in a Valid State

Filters with a DA architecture, allow data to be delivered to the outputs N cycles ($N \geq 2$) later than the inputs. You can use the `HoldInputDataBetweenSamples` property to determine how long (in terms of clock cycles) input data values are held in a valid state, as follows:

- When `HoldInputDataBetweenSamples` is set 'on' (the default), input data values are held in a valid state across N clock cycles.
- When `HoldInputDataBetweenSamples` is set 'off', data values are held in a valid state for only one clock cycle. For the next $N - 1$ cycles, data is in an unknown state (expressed as 'X') until the next input sample is clocked in.

Distributed Arithmetic Options in the Generate HDL Dialog Box

The Generate HDL dialog box provides several options related to DA code generation.



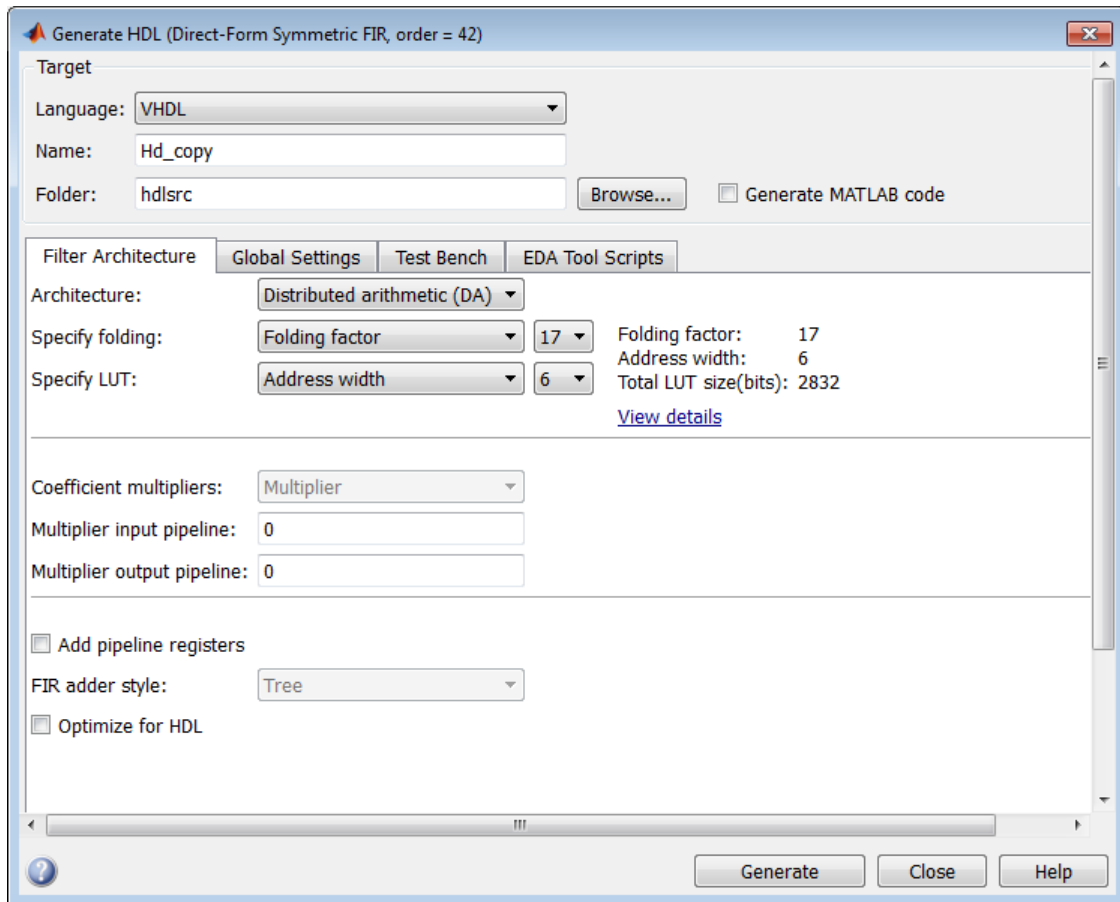
- The **Architecture** pop-up menu, which lets you enable DA code generation and displays related options
- The **Specify folding** drop-down menu, which lets you directly specify the filter's folding factor, or set a value for the **DARadix** property (see “DARadix Property” on page 4-29).
- The **Specify LUT** drop-down menu, which lets you directly set a value for the **DALUTPartition** property (see “DALUTPartition Property” on page 4-27). You can also select an address width for the LUT. If you specify an address width, the coder uses input LUTs as required.

The Generate HDL dialog box initially displays default DA related option values that correspond to the current filter design. Otherwise, the requirements for setting these options are identical to those described in “DALUTPartition Property” on page 4-27 and “DARadix Property” on page 4-29.

To specify DA code generation using the Generate HDL dialog box, follow these steps:

- 1 Design a FIR filter (using **FDATool**, **filterbuilder**, or **MATLAB** commands) that meets the requirements described in “Requirements and Considerations for Generating Distributed Arithmetic Code” on page 4-26.
- 2 Open the Generate HDL dialog box.
- 3 Select **Distributed Arithmetic (DA)** from the **Architecture** pop-up menu.

When you select this option, the related **Specify folding** and **Specify LUT** options are displayed below the **Architecture** menu. The following figure shows the default DA options for a Direct Form FIR filter.



- 4 Select one of the following options from the **Specify folding** drop-down menu:
 - **Folding factor** (default): Select a folding factor from the drop-down menu to the right of **Specify folding**. The menu contains an exhaustive list of folding factor options for the filter.
 - **DA radix**: Select the number of bits processed simultaneously, expressed as a power of 2. The default **DA radix** value is 2, specifying processing of one bit at a time, or fully serial DA. If desired, set the **DA radix** field to a nondefault value.
- 5 Select one of the following options from the **Specify LUT** drop-down menu:

- **Address width** (default): Select from the drop-down menu to the right of **Specify LUT**. The menu contains an exhaustive list of LUT address widths for the filter.
 - **Partition**: Select, or enter, a vector specifying the number and size of LUT partitions.
- 6 Set other HDL options as required, and generate code. Invalid or illegal values for **LUT Partition** or **DA Radix** are reported at code generation time.

Viewing Detailed DA Options

As you interact with the **Specify folding** and **Specify LUT** options you can see the results of your choice in three display-only fields: **Folding factor**, **Address width**, and **Total LUT size (bits)**.

In addition, when you click the **View details** hyperlink, the coder displays a report showing complete DA architectural details for the current filter, including:

- Filter lengths
- Complete list of applicable folding factors and how they apply to the sets of LUTs
- Tabulation of the configurations of LUTs with total LUT Size and LUT details

The following figure shows a typical report.

Architecture Details

--- Distributed Arithmetic (DA) ---

The following table is the summary of various filter lengths:

Total Coefficients	Zeros	A/Symm	Effective
43	0	21	22

Effective filter length for SerialPartition value is 22.

Table of 'DARadix' values with corresponding values of folding factor and multiple for LUT sets for the given filter:

Folding Factor	LUT-Sets Multiple	DARadix
1	16	2^{16}
3	8	2^8
5	4	2^4
9	2	2^2
17	1	2^1

Details of LUTs with corresponding 'DALUTPartition' values:

Max Address Width	Size(bits)	LUT Details	DALUTPartition
12	74752	1x1024x17, 1x4096x14	[12 10]
11	61440	1x2048x13, 1x2048x17	[11 11]
10	28740	1x1024x13, 1x1024x15, 1x4x17	[10 10 2]
9	14096	1x16x17, 1x512x13, 1x512x14	[9 9 4]
8	8000	1x256x13, 1x256x14, 1x64x17	[8 8 6]
7	5408	2x128x13, 1x128x16, 1x2x16	[7 7 7 1]
6	2832	1x16x17, 2x64x13, 1x64x14	[6 6 6 4]
5	1764	1x32x12, 1x32x13, 2x32x14, 1x4x17	[5 5 5 5 2]
4	1076	3x16x12, 1x16x13, 1x16x14, 1x4x17	[4 4 4 4 4 2]
3	744	1x2x16, 1x8x10, 3x8x12, 1x8x13, 1x8x14, 1x8x16	[3 3 3 3 3 3 1]
2	544	1x4x10, 3x4x11, 3x4x12, 2x4x13, 1x4x14, 1x4x17	ones(1,11)*2

Notes:

- LUT Details indicates number of LUTs with their sizes. e.g. 1x1024x18 implies 1 LUT of 1024 18-bit wide locations.

OK

DA Interactions with Other HDL Options

When **Distributed Arithmetic (DA)** is selected in the **Architecture** menu, some other HDL options change automatically to settings that correspond to DA code generation:

- **Coefficient multipliers** is set to **Multiplier** and disabled.
- **FIR adder style** is set to **Tree** and disabled.
- **Add input register** in the Ports pane of the Generate HDL dialog box is selected and disabled. (An input register, used as part of a shift register, is used in DA code.)
- **Add output register** in the Ports pane of the Generate HDL dialog box is selected and disabled.

Architecture Options for Cascaded Filters

You can specify serial, DA, or parallel architectures for individual stages of cascade filters. These options lead to an area efficient implementation of cascade filters, such as Digital Down Converter (DDC), Digital Up Converter (DUC), and so on. You can use this feature only with the command-line interface (`generatehdl`).

You can pass a cell array of values to the 'SerialPartition', 'DALUTPartition', and 'DARadix' properties, with each element corresponding to its respective stage. You can also pass the default values if you want to skip the corresponding specification for a stage:

- -1 for SerialPartition
- -1 for DALUTPartition
- 2 for DARadix

When you create a cascaded filter, Filter Design HDL Coder software performs the following actions:

- Generates code for each stage as per the inferred architecture.
- Generates an overall timing controller at the top level. This controller then produces clock enables for the module in each stage, which corresponds to that module's rate and folding factor.

Tip Use the `hdlfilterserialinfo` function to display the effective filter length and partitioning options for each filter stage of a cascade.

Cascaded Filter with Serial Partitioning

To specify serial partitioning for one or more cascade stages, use the `SerialPartition` property. The following example defines different serial partitions for each filter in a two-stage cascade. The code implements each stage using two multipliers per stage. The partition vectors reside within a cell array.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
Hp = design(fdesign.highpass('N,Fc',8,.4))
Hp.arithmetic = 'fixed'
Hc = cascade(Hd,Hp)
```

```
sp1 = hdlfilterserialinfo(Hc.stage(1), 'Multiplier', 2)
sp2 = hdlfilterserialinfo(Hc.stage(2), 'Multiplier', 2)
generatehdl(Hc, 'serialpartition', {sp2, sp2})
```

Cascaded Filter with DA Architecture

To specify a DA architecture for one or more cascade stages, use the `DALUTPartition` property. Optionally, you can specify the `DARadix` property. The following example defines LUT partitions for each filter in a two-stage cascade. The first stage uses LUTs with a maximum address size of 5 bits and the second stage uses LUTs with a maximum address size of 3 bits. The `DALUT` partition vectors and `DARadix` values reside within a cell array for each property value.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4))
Hd.arithmetic = 'fixed'
Hp = design(fdesign.highpass('N,Fc',8,.4))
Hp.arithmetic = 'fixed'
Hc = cascade(Hd,Hp)
dp1 = hdlfilterdainfo(Hc.stage(1), 'LUTInputs', 5)
dp2 = hdlfilterdainfo(Hc.stage(2), 'LUTInputs', 3)
generatehdl(Hc, 'DALUTPartition', {dp1, dp2})
```

Tip Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and `DARadix` values for each filter stage of a cascade.

Cascaded Filter with Multiple Architectures

You can specify a mix of serial, DA, and parallel architectures depending upon your hardware constraints. This example implements DA, serial, and parallel architectures for the first, second, and third stages, respectively.

```
% create a filter object
h1 = dfilt.dffir([0.05 -.25 .88 0.9 .88 -.25 0.05])
h1.Arithmetic = 'fixed'
h2 = dfilt.dfasymfir([-0.008 0.06 -0.44 0.44 -0.06 0.008])
h2.Arithmetic = 'fixed'
h3 = dfilt.dfsymfir([-0.008 0.06 0.44 0.44 0.06 -0.008])
h3.Arithmetic = 'fixed'
% generate HDL with mixed architectures
Generatehdl(Hd, 'serialpartition',{-1,3,-1},...
    'DaLUTPartition', {[4 3],-1,-1},...
    'DaRadix', {2^8,2,2})
```


CSD Optimizations for Coefficient Multipliers

By default, the coder produces code that includes coefficient multiplier operations. You can optimize these operations such that they decrease the area used and maintain or increase clock speed. You do this by instructing the coder to replace multiplier operations with additions of partial products produced by canonical signed digit (CSD) or factored CSD techniques. These techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. The amount of optimization you can achieve is dependent on the binary representation of the coefficients used.

Note: The coder does not use coefficient multiplier operations for multirate filters. Therefore, the **Coefficient multipliers** options described below are disabled for multirate filters.

To optimize coefficient multipliers (for nonmultirate filter types),

- 1** Select **CSD** or **Factored-CSD** from the **Coefficient multipliers** menu in the **Filter architecture** pane of the Generate HDL dialog box.
- 2** Consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,
 - a** Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.
 - b** Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
- 3** Continue setting other options or click **Generate** to initiate code generation.

If you are generating code for an FIR filter, see “Multiplier Input and Output Pipelining for FIR Filters” on page 4-41 for information on a related optimization.

Command Line Alternative: Use the `generatehdl` function with the property `CoeffMultipliers` to optimize coefficient multipliers with CSD techniques.

Improving Filter Performance with Pipelining

In this section...

“Optimizing the Clock Rate with Pipeline Registers” on page 4-40

“Multiplier Input and Output Pipelining for FIR Filters” on page 4-41

“Optimizing Final Summation for FIR Filters” on page 4-42

“Specifying or Suppressing Registered Input and Output” on page 4-44

Optimizing the Clock Rate with Pipeline Registers

You can optimize the clock rate used by filter code by applying pipeline registers. Although the registers increase the overall filter latency and space used, they provide significant improvements to the clock rate. These registers are disabled by default. When you enable them, the coder adds registers between stages of computation in a filter.

For...	Pipeline Registers Are Added Between...
FIR, antisymmetric FIR, and symmetric FIR filters	Each level of the final summation tree
Transposed FIR filters	Coefficient multipliers and adders
IIR filters	Sections

For example, for a sixth order IIR filter, the coder adds two pipeline registers, one between the first and second section and one between the second and third section.

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see “Optimizing Final Summation for FIR Filters” on page 4-42

Note: The use of pipeline registers in FIR, antisymmetric FIR, and symmetric FIR filters can produce numeric results that differ from those produced by the original filter object because they force the tree mode of final summation.

To use pipeline registers,

- 1 Select the **Add pipeline registers** option in the **Filter architecture** pane of the Generate HDL dialog box.

- 2 For FIR, antisymmetric FIR, and symmetric FIR filters, consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin:
 - a Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.
 - b Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numerical results can differ before the coder issues a warning.
- 3 Continue setting other options or click **Generate** to initiate code generation.

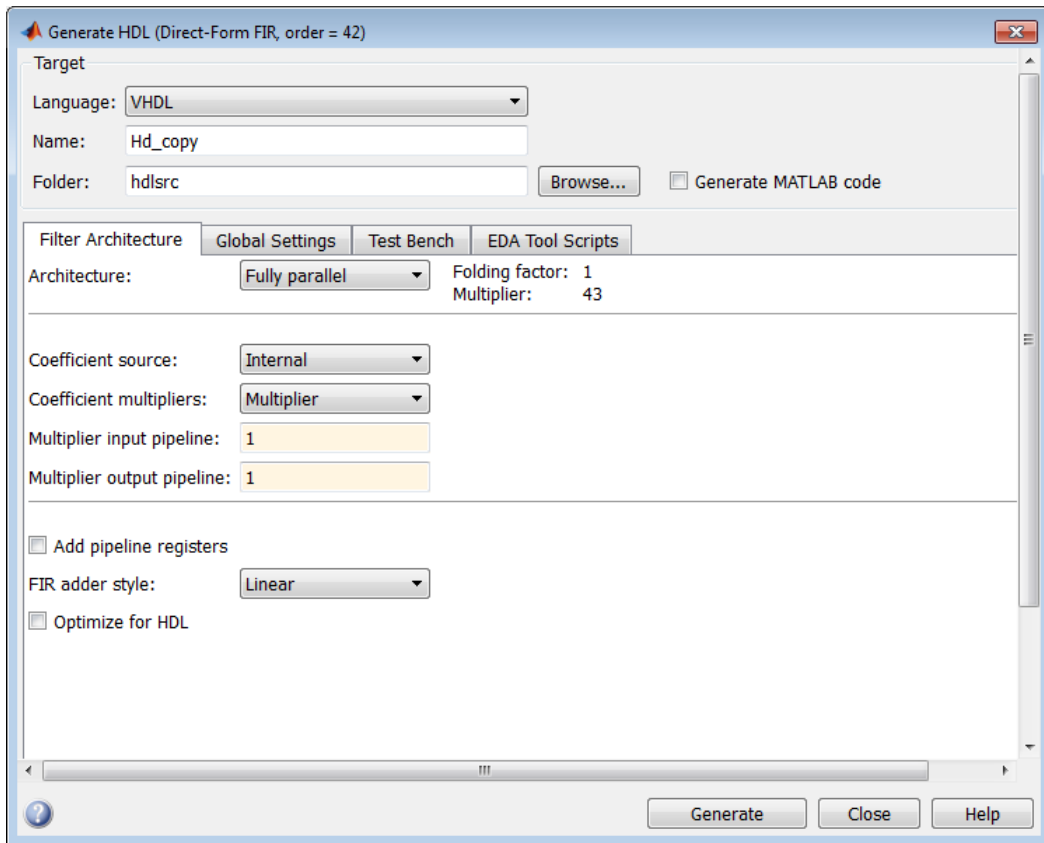
Command Line Alternative: Use the `generatehdl` function with the property `AddPipelineRegisters` to optimize the filters with pipeline registers.

Multiplier Input and Output Pipelining for FIR Filters

If you choose to retain multiplier operations for a FIR filter, you can achieve significantly higher clock rates by adding pipeline stages at multiplier inputs or outputs.

The following figure shows the GUI options for multiplier pipelining options. To enable these options, **Coefficient multipliers** to **Multiplier**.

- **Multiplier input pipeline:** To add pipeline stages before each multiplier, enter the desired number of stages as an integer greater than or equal to 0.
- **Multiplier output pipeline:** To add pipeline stages after each multiplier, enter the desired number of stages as an integer greater than or equal to 0.



Command Line Alternative: Use the `generatehdl` function with the `MultiplierInputPipeline` and `MultiplierOutputPipeline` properties to specify multiplier pipelining for FIR filters.

Optimizing Final Summation for FIR Filters

If you are generating HDL code for an FIR filter, consider optimizing the final summation technique to be applied to the filter. By default, the coder applies linear adder summation, which is the final summation technique discussed in most DSP text books. Alternatively, you can instruct the coder to apply tree or pipeline final summation. When set to tree mode, the coder creates a final adder that performs pair-wise addition on successive products that execute in parallel, rather than sequentially. Pipeline mode

produces results similar to tree mode with the addition of a stage of pipeline registers after processing each level of the tree.

In comparison,

- The number of adder operations for linear and tree mode are the same, but the timing for tree mode might be significantly better due to summations occurring in parallel.
- Pipeline mode optimizes the clock rate, but increases the filter latency by the base 2 logarithm of the number of products to be added, rounded up to the nearest integer.
- Linear mode helps attain numeric accuracy in comparison to the original filter object. Tree and pipeline modes can produce numeric results that differ from those produced by the filter object.

To change the final summation to be applied to an FIR filter,

- 1 Select one of the following options in the **Filter architecture** pane of the Generate HDL dialog box:

For...	Select...
Linear mode (the default)	Linear from the FIR adder style menu
Tree mode	Tree from the FIR adder style menu
Pipeline mode	The Add pipeline registers check box

- 2 If you specify tree or pipelined mode, consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,
 - a Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.
 - b Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
- 3 Continue setting other options or click **Generate** to initiate code generation.

Command Line Alternative: Use the `generatehdl` function with the property `FIRAdderStyle` or `AddPipelineRegisters` to optimize the final summation for FIR filters.

Specifying or Suppressing Registered Input and Output

The coder adds an extra input register (`input_register`) and an extra output register (`output_register`) during HDL code generation. These extra registers can be useful for timing purposes, but they add to the filter's overall latency. The following process block writes to extra output register `output_register` when a clock event occurs and `clk` is active high (1):

```
Output_Register_Process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        output_register <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
        IF clk_enable = '1' THEN
            output_register <= output_typeconvert;
        END IF;
    END IF;
END PROCESS Output_Register_Process;
```

If overall latency is a concern for your application and you do not have timing requirements, you can suppress generation of the extra registers as follows:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Ports** tab in the **Additional settings** pane.
- 3 Clear **Add input register** and **Add output register** as required. The following figure shows the setting for suppressing the generation of an extra input register.



Command Line Alternative: Use the `generatehdl` and function with the properties `AddInputRegister` and `AddOutputRegister` to add an extra input or output register.

Overall HDL Filter Code Optimization

In this section...
“Optimize for HDL” on page 4-46
“Set Error Margin for Test Bench” on page 4-47

Optimize for HDL

By default, generated HDL code is bit-compatible with the numeric results produced by the original filter object. The **Optimize for HDL** option generates HDL code that is slightly optimized for clock speed or space requirements. However, note that this optimization causes the coder to

- Make tradeoffs concerning data types
- Avoid extra quantization
- Generate code that produces numeric results that are different than the results produced by the original filter object

To optimize generated code for clock speed or space requirements:

- 1 Select **Optimize for HDL** in the **Filter architecture** pane of the Generate HDL dialog box.
- 2 Consider setting an error margin for the generated test bench. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,
 - a Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.
 - b Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
- 3 Continue setting other options or click **Generate** to initiate code generation.

Command Line Alternative: Use the `generatehdl` function with the property `OptimizeForHDL` to enable the optimizations described above.

Set Error Margin for Test Bench

Customizations that provide optimizations can generate test bench code that produces numeric results that differ from results produced by the original filter object. These options include:

- **Optimize for HDL**
- **FIR adder style** set to **Tree**
- **Add pipeline registers** for FIR, asymmetric FIR, and symmetric FIR filters

If you choose to use these options, consider setting an error margin for the generated test bench to account for differences in numeric results. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin:

- 1** Select the **Test Bench** pane in the Generate HDL dialog box.
- 2** Within the **Test Bench** pane, select the **Configuration** subpane.
- 3** For fixed-point filters, the initial **Error margin (bits)** field has a default value of 4. If you wish to change the error margin, enter an integer in the **Error margin (bits)** field. In the following figure, the error margin is set to 4 bits.

4 Optimization of HDL Filter Code

Filter Architecture Global Settings **Test Bench** EDA Tool Scripts

Test Bench Generation Output

HDL test bench

Test bench language: **VHDL** File name: **Hd_copy_tb**

Cosimulation blocks

Cosimulation model for use with: **Mentor Graphics ModelSim**

Stimuli Configuration

Force clock

Clock high time (ns): **5**

Clock low time (ns): **5**

Hold time (ns): **2**

Setup time (ns): **8**

Force clock enable

Clock enable delay (in clock cycles): **1**

Force reset

Reset length (in clock cycles): **2**

Hold input data between samples

Initialize test bench inputs

Multi-file test bench

Test bench data file name postfix: **_data**

Test bench reference postfix: **_ref**

Error margin (bits): **4**

Simulator flags:

Generate **Close** **Help**

Customization of HDL Filter Code

- “HDL File Names and Locations” on page 5-2
- “HDL Identifiers and Comments” on page 5-10
- “Ports and Resets” on page 5-22
- “HDL Language Constructs” on page 5-29

HDL File Names and Locations

In this section...
“Setting the Location of Generated Files” on page 5-2
“Naming the Generated Files and Filter Entity” on page 5-3
“Set HDL File Name Extensions” on page 5-4
“Splitting Entity and Architecture Code Into Separate Files” on page 5-8

Setting the Location of Generated Files

By default, the coder places generated HDL files in the subfolder `hdlsrc` under your current working folder. To direct the coder output to a folder other than the default target folder, you can use either the **Folder** field or the **Browse** button in the **Target** pane of the Generate HDL dialog box.

Clicking on the **Browse** button opens a browser window that lets you select (or create) the folder to which generated code will be written. When the folder is selected, the full path and folder name are automatically entered into the **Folder** field.

Alternatively, you can enter the folder specification directly into the **Folder** field. If you specify a folder that does not exist, the coder creates the folder for you before writing the generated files. Your folder specification can be one of the following:

- Folder name. In this case, the coder looks for the subfolder under your current working folder. If it cannot find the specified folder, the coder creates it.
- An absolute path to a folder under your current working folder. If the coder cannot find the specified folder, the coder creates it.
- A relative path to a higher level folder under your current working folder. For example, if you specify `../../../../../myfiltvhd`, the coder checks whether a folder named `myfiltvhd` exists three levels up from your current working folder, creates the folder if it does not exist, and writes generated HDL files to that folder.

In the following figure, the folder is set to `MyFIRBetaVHDL`.

Given this setting, the coder creates the subfolder `MyFIRBetaVHDL` under the current working folder and writes generated HDL files to that folder.

Command Line Alternative: Use the `generatehdl` function with the `TargetDirectory` property to redirect coder output.

Naming the Generated Files and Filter Entity

To set the string that the coder uses to name the filter entity or module and generated files, specify a new value in the **Name** field of the **Filter settings** pane of the Generate HDL dialog box. The coder uses the **Name** string to

- Label the VHDL entity or Verilog module for your filter
- Name the file containing the HDL code for your filter
- Derive names for the filter's test bench and package files

Derivation of File Names

By default, the coder creates the HDL files listed in the following table. File names in generated HDL code derive from the name of the filter for which the HDL code is being generated and the file type extension `.vhd` or `.v` for VHDL and Verilog, respectively. The table lists example file names based on filter name `Hq`.

Language	Generated File	File Name	Example
Verilog	Source file for the quantized filter	<code>dfilt_name.v</code>	<code>Hq.v</code>
	Source file for the filter's test bench	<code>dfilt_name_tb.v</code>	<code>Hq_tb.v</code>
VHDL	Source file for the quantized filter	<code>dfilt_name.vhd</code>	<code>Hq.vhd</code>
	Source file for the filter's test bench	<code>dfilt_name_tb.vhd</code>	<code>Hq_tb.vhd</code>

Language	Generated File	File Name	Example
	Package file, if required by the filter design	<i>dfilt_name_pkg.vhd</i>	Hq_pkg.vhd

By default, the coder generates a single test bench file, containing test bench helper functions, data, and test bench code. You can split these elements into separate files, as described in “Splitting Test Bench Code and Data into Separate Files” on page 6-14.

By default, the code for a filter's VHDL entity and architectures is written to a single VHDL source file. Alternatively, you can specify that the coder write the generated code for the entity and architectures to separate files. For example, if the filter name is Hd, the coder writes the VHDL code for the filter to files Hd_entity.vhd and Hd_arch.vhd (see “Splitting Entity and Architecture Code Into Separate Files” on page 5-8).

Derivation of Entity Names

The coder also uses the filter name to name the VHDL entity or Verilog module that represents the quantized filter in the HDL code. Assuming a filter name of Hd, the name of the filter entity or module in the HDL code is Hd.

Set HDL File Name Extensions

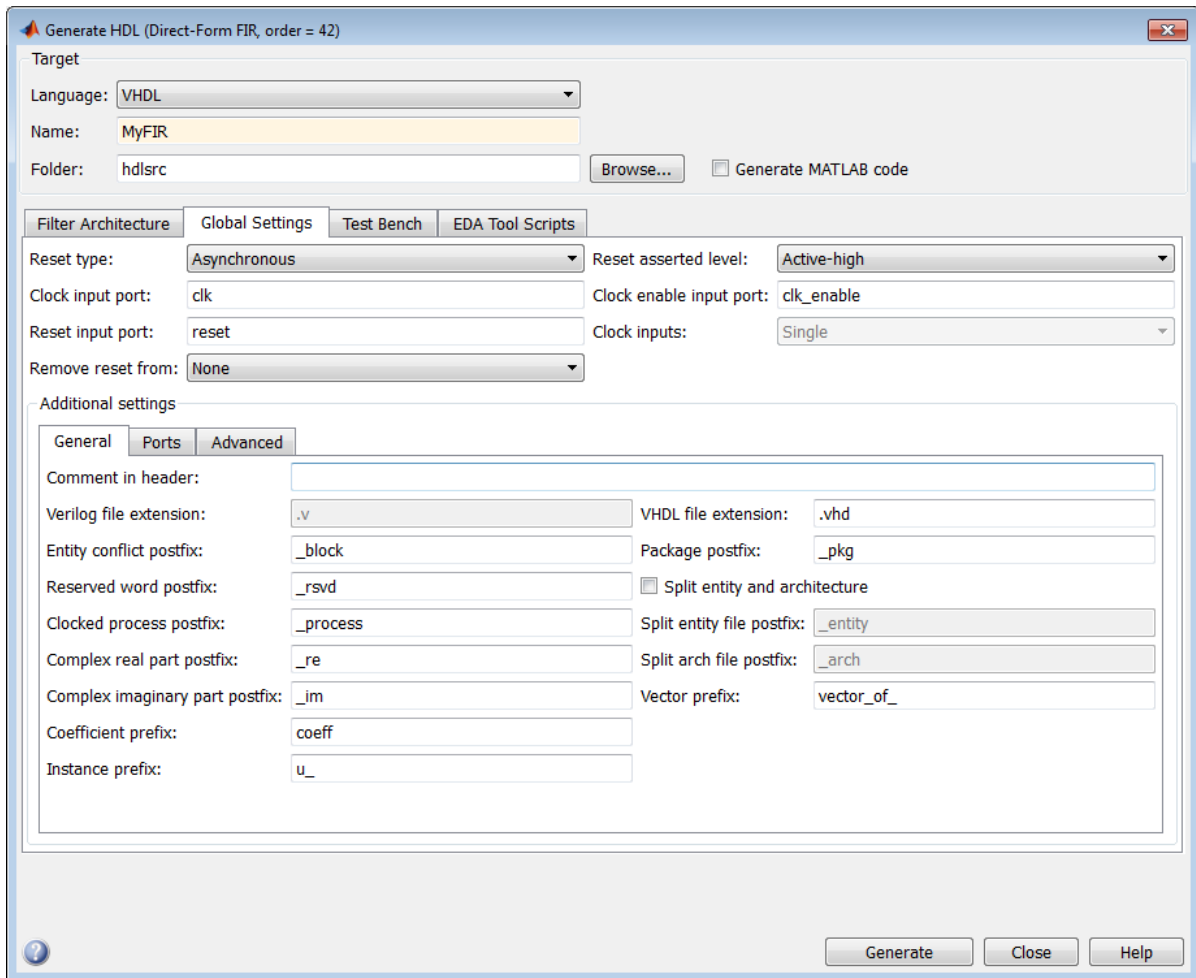
- “Set VHDL File Name Extension Via the Generate HDL Tool” on page 5-4
- “Set Verilog File Name Extension Via the Generate HDL Tool” on page 5-6
- “Set HDL File Name Extensions Via the Command Line” on page 5-8

Set VHDL File Name Extension Via the Generate HDL Tool

By default, the filter HDL files are generated with a .vhd file extension when the Language is specified as VHDL. To change the VHDL file extension,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Type the new file extension in the **VHDL file extension** field.

Based on the following settings, the coder generates the filter file MyFIR.vhd1.



Note: When specifying strings for file names and file type extensions, consider platform-specific requirements and restrictions. Also consider postfix strings that the coder appends to the **Name** string, such as **_tb** and **_pkg**.

Command Line Alternative: Use the `generatehdl` function with the `Name` property to set the name of your filter entity and the base string for generated HDL file names.

Specify the function with the `VerilogFileExtension` or `VHDLFileExtension` property to specify a file type extension for generated HDL files.

Set Verilog File Name Extension Via the Generate HDL Tool

By default, the filter HDL files are generated with `.v` file extension when Language is specified as Verilog. To change the file extension for a Verilog file,

- 1** Select the **Global Settings** tab on the Generate HDL dialog box.
- 2** Select the **General** tab in the **Additional settings** pane.
- 3** Type the new file extension in the **Verilog file extension** field.

Based on the following settings, the coder generates the filter file `MyFIR.v`.

Generate HDL (Direct-Form FIR, order = 42)

Target

Language: Verilog

Name: MyFIR

Folder: hdsrc Generate MATLAB code

Filter Architecture Global Settings Test Bench EDA Tool Scripts

Reset type: Asynchronous Reset asserted level: Active-high

Clock input port: clk Clock enable input port: clk_enable

Reset input port: reset Clock inputs: Single

Remove reset from: None

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: .v VHDL file extension: .vhd

Entity conflict postfix: _block Package postfix: _pkg

Reserved word postfix: _rsvd Split entity and architecture

Clocked process postfix: _process Split entity file postfix: _entity

Complex real part postfix: _re Split arch file postfix: _arch

Complex imaginary part postfix: _im Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

Note: When specifying strings for file names and file type extensions, consider platform-specific requirements and restrictions. Also consider postfix strings that the coder appends to the **Name** string, such as **_tb** and **_pkg**.

Set HDL File Name Extensions Via the Command Line

Command Line Alternative: Use the `generatehdl` function with the `Name` property to set the name of your filter entity and the base string for generated HDL file names. Specify the function with the `VerilogFileExtension` or `VHDLFileExtension` property to specify a file type extension for generated HDL files.

Splitting Entity and Architecture Code Into Separate Files

By default, the coder includes a filter's VHDL entity and architecture code in the same generated VHDL file. Alternatively, you can instruct the coder to place the entity and architecture code in separate files. For example, instead of generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

The names of the entity and architecture files derive from

- The base file name, as specified by the **Name** field in the **Target** pane of the Generate HDL dialog box
- Default postfix string values `_entity` and `_arch`
- The VHDL file type extension, as specified by the **VHDL file extension** field on the **General** pane of the Generate HDL dialog box

To split the filter source file, do the following:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Select **Split entity and architecture**. The **Split entity file postfix** and **Split arch. file postfix** fields are now enabled.

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: VHDL file extension:

Entity conflict postfix: Package postfix:

Reserved word postfix: Split entity and architecture

Clocked process postfix: Split entity file postfix:

Complex real part postfix: Split arch file postfix:

Complex imaginary part postfix: Vector prefix:

Coefficient prefix:

Instance prefix:

- 4 Specify new strings in the postfix fields if you want to use postfix string values other than `_entity` and `_arch` to identify the generated VHDL files.

Note: When specifying a string for use as a postfix value in file names, consider the size of the base name and platform-specific file naming requirements and restrictions.

Command Line Alternative: Use the `generatehdl` function with the property `SplitEntityArch` to split the filter's VHDL code into separate files. Use properties `SplitEntityFilePostfix` and `SplitArchFilePostfix` to rename the file name postfix for VHDL entity and architecture code components.

HDL Identifiers and Comments

In this section...

- “Specifying a Header Comment” on page 5-10
- “Resolving Entity or Module Name Conflicts” on page 5-12
- “Resolving HDL Reserved Word Conflicts” on page 5-13
- “Setting the Postfix String for VHDL Package Files” on page 5-16
- “Specifying a Prefix for Filter Coefficients” on page 5-17
- “Specifying a Postfix String for Process Block Labels” on page 5-18
- “Setting a Prefix for Component Instance Names” on page 5-19
- “Setting a Prefix for Vector Names” on page 5-20

Specifying a Header Comment

The coder includes a header comment block at the top of the files it generates. The header comment block contains information about the specifications of the generating filter and about the coder options that were selected at the time HDL code was generated.

You can use the **Comment in header** option to add a comment string, to the end of the header comment block in each generated file. For example, you might use this option to add the comment `This module was automatically generated`. With this change, the preceding header comment block would appear as follows:

```
-- .....
--
-- Module: Hlp
--
-- Generated by MATLAB(R) 7.11 and the Filter Design HDL Coder 2.7.
--
-- Generated on: 2010-08-31 13:32:16
--
-- This module was automatically generated
--
-- .....

-- .....
-- HDL Code Generation Options:
--
-- TargetLanguage: VHDL
-- Name: Hlp
-- UserComment: User data, length 47
```

```

-- Filter Specifications:
--
-- Sampling Frequency : N/A (normalized frequency)
-- Response           : Lowpass
-- Specification      : Fp,Fst,Ap,Ast
-- Passband Edge      : 0.45
-- Stopband Edge      : 0.55
-- Passband Ripple    : 1 dB
-- Stopband Atten.    : 60 dB
-----

-- HDL Implementation : Fully parallel
-- Multipliers         : 43
-- Folding Factor      : 1
-----

-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-----
-- Filter Structure   : Direct-Form FIR
-- Filter Length      : 43
-- Stable             : Yes
-- Linear Phase       : Yes (Type 1)
-- Arithmetic         : fixed
-- Numerator          : s16,16 -> [-5.000000e-001 5.000000e-001)
-- Input              : s16,15 -> [-1 1)
-- Filter Internals   : Full Precision
-- Output             : s33,31 -> [-2 2) (auto determined)
-- Product            : s31,31 -> [-5.000000e-001 5.000000e-001) (auto determined)
-- Accumulator        : s33,31 -> [-2 2) (auto determined)
-- Round Mode         : No rounding
-- Overflow Mode      : No overflow
-----

```

To add a header comment,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Type the comment string in the **Comment in header** field, as shown in the following figure.

Additional settings

General Ports Advanced

Comment in header: This module was automatically generated

Verilog file extension: .v VHDL file extension: .vhd

Entity conflict postfix: _block Package postfix: _pkg

Reserved word postfix: _rsvd Split entity and architecture

Clocked process postfix: _process Split entity file postfix: _entity

Complex real part postfix: _re Split arch file postfix: _arch

Complex imaginary part postfix: _im Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

Unapplied changes

Command Line Alternative: Use the `generatehdl` function with the property `UserComment` to add a comment string to the end of the header comment block in each generated HDL file.

Resolving Entity or Module Name Conflicts

The coder checks whether multiple entities in VHDL or multiple modules in Verilog share the same name. If a name conflict exists, the coder appends the postfix `_block` to the second of the two matching strings.

To change the postfix string:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Entity conflict postfix** field, as shown in the following figure.

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: VHDL file extension:

Entity conflict postfix: Package postfix:

Reserved word postfix: Split entity and architecture

Clocked process postfix: Split entity file postfix:

Complex real part postfix: Split arch file postfix:

Complex imaginary part postfix: Vector prefix:

Coefficient prefix:

Instance prefix:

Command Line Alternative: Use the `generatehdl` function with the property `EntityConflictPostfix` to change the entity or module conflict postfix string.

Resolving HDL Reserved Word Conflicts

The coder checks whether strings that you specify as names, postfix values, or labels are VHDL or Verilog reserved words. See “Reserved Word Tables” on page 5-14 for listings of VHDL and Verilog reserved words.

If you specify a reserved word, the coder appends the postfix `_rsvd` to the string. For example, if you try to name your filter `mod`, for VHDL code, the coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

To change the postfix string:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Reserved word postfix** field, as shown in the following figure.

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: VHDL file extension:

Entity conflict postfix: Package postfix:

Reserved word postfix: Split entity and architecture

Clocked process postfix: Split entity file postfix:

Complex real part postfix: Split arch file postfix:

Complex imaginary part postfix: Vector prefix:

Coefficient prefix:

Instance prefix:

Command Line Alternative: Use the `generatehdl` function with the property `ReservedWordPostfix` to change the reserved word postfix string.

Reserved Word Tables

The following tables list VHDL and Verilog reserved words.

VHDL Reserved Words

abs	access	after	alias	all
and	architecture	array	assert	attribute
begin	block	body	buffer	bus
case	component	configuration	constant	disconnect
downto	else	elsif	end	entity
exit	file	for	function	generate
generic	group	guarded	if	impure
in	inertial	inout	is	label
library	linkage	literal	loop	map
mod	nand	new	next	nor
not	null	of	on	open
or	others	out	package	port

postponed	procedure	process	pure	range
record	register	reject	rem	report
return	rol	ror	select	severity
signal	shared	sla	sll	sra
srl	subtype	then	to	transport
type	unaffected	units	until	use
variable	wait	when	while	with
xnor	xor			

Verilog Reserved Words

always	and	assign	automatic	begin
buf	bufif0	bufif1	case	casex
casez	cell	cmos	config	deassign
default	defparam	design	disable	edge
else	end	endcase	endconfig	endfunction
endgenerate	endmodule	endprimitive	endspecify	endtable
endtask	event	for	force	forever
fork	function	generate	genvar	highz0
highz1	if	ifnone	incdir	include
initial	inout	input	instance	integer
join	large	liblist	library	localparam
macromodule	medium	module	nand	negedge
nmos	nor	noshowcancelled	not	notif0
notif1	or	output	parameter	pmos
posedge	primitive	pull0	pull1	pulldown
pullup	pulsetype_onevent	pulsetype_ondetect	cmos	real
realtime	reg	release	repeat	rnmos
rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparam
strong0	strong1	supply0	supply1	table

task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
trireg	unsigned	use	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

Setting the Postfix String for VHDL Package Files

By default, the coder appends the postfix `_pkg` to the base file name when generating a VHDL package file. To rename the postfix string for package files, do the following:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Specify a new value in the **Package postfix** field.

The screenshot shows the 'Global Settings' tab of the 'Generate HDL' dialog box. Under the 'Additional settings' section, the 'General' sub-tab is selected. The 'Package postfix' field is highlighted in yellow and contains the text '_filtpkg'. Other fields include 'Verilog file extension' (.v), 'VHDL file extension' (.vhd), 'Entity conflict postfix' (_block), 'Reserved word postfix' (_rsvd), 'Clocked process postfix' (_process), 'Complex real part postfix' (_re), 'Complex imaginary part postfix' (_im), 'Coefficient prefix' (coeff), 'Instance prefix' (u_), 'Reset type' (Asynchronous), 'Reset asserted level' (Active-high), 'Clock input port' (clk), 'Clock enable input port' (clk_enable), 'Reset input port' (reset), 'Clock inputs' (Single), 'Remove reset from' (None), 'Split entity and architecture' (checked), 'Split entity file postfix' (_entity), 'Split arch file postfix' (_arch), and 'Vector prefix' (vector_of_).

Note: When specifying a string for use as a postfix in file names, consider the size of the base name and platform-specific file naming requirements and restrictions.

Command Line Alternative: Use the `generatehdl` function with the `PackagePostfix` property to rename the file name postfix for VHDL package files.

Specifying a Prefix for Filter Coefficients

The coder declares a filter's coefficients as constants within an `rtl` architecture. The coder derives the constant names adding the prefix `coeff` to the following:

For...	The Prefix Is Concatenated with...
FIR filters	Each coefficient number, starting with 1. Examples: <code>coeff1</code> , <code>coeff22</code>
IIR filters	An underscore (<code>_</code>) and an <code>a</code> or <code>b</code> coefficient name (for example, <code>_a2</code> , <code>_b1</code> , or <code>_b2</code>) followed by the string <code>_sectionn</code> , where <code>n</code> is the section number. Example: <code>coeff_b1_section3</code> (first numerator coefficient of the third section)

For example:

```
ARCHITECTURE rtl OF Hd IS
-- Type Definitions
TYPE delay_pipeline_type IS ARRAY(NATURAL range <>) OF signed(15 DOWNTO 0);-- sfix16_En15
CONSTANT coeff1          : signed(15 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
CONSTANT coeff2          : signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
CONSTANT coeff3          : signed(15 DOWNTO 0) := to_signed(-81, 16); -- sfix16_En15
CONSTANT coeff4          : signed(15 DOWNTO 0) := to_signed(120, 16); -- sfix16_En15
```

To use a prefix other than `coeff`,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Coefficient prefix** field, as shown in the following figure.

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: VHDL file extension:

Entity conflict postfix: Package postfix:

Reserved word postfix: Split entity and architecture

Clocked process postfix: Split entity file postfix:

Complex real part postfix: Split arch file postfix:

Complex imaginary part postfix: Vector prefix:

Coefficient prefix:

Instance prefix:

The string that you specify

- Must start with a letter
- Cannot include a double underscore (__)

Note: If you specify a VHDL or Verilog reserved word, the coder appends a reserved word postfix to the string to form a valid identifier. If you specify a prefix that ends with an underscore, the coder replaces the underscore character with **under**. For example, if you specify `coef_`, the coder generates coefficient names such as `coefunder1`.

Command Line Alternative: Use the `generatehdl` function with the property `CoeffPrefix` to change the base name for filter coefficients.

Specifying a Postfix String for Process Block Labels

The coder generates process blocks to modify the content of a filter's registers. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the coder derives the label `delay_pipeline_process` in the following block from the register name `delay_pipeline` and the postfix string `_process`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
```

```

IF reset = '1' THEN
  delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
ELSIF clk'event AND clk = '1' THEN
  IF clk_enable = '1' THEN
    delay_pipeline(0) <= signed(filter_in)
    delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
  END IF;
END IF;
END PROCESS delay_pipeline_process;

```

The **Clocked process postfix** property lets you change the postfix string to a value other than `_process`. For example, to change the postfix string to `_clkproc`, do the following:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Clocked process postfix** field, as shown in the following figure.

The screenshot shows the 'Additional settings' dialog box with the 'General' tab selected. The 'Clocked process postfix' field is highlighted in yellow and contains the text '_clkproc'. Other fields include 'Verilog file extension' (.v), 'VHDL file extension' (.vhd), 'Entity conflict postfix' (_block), 'Package postfix' (_pkg), 'Reserved word postfix' (_rsvd), 'Split entity and architecture' (checkbox), 'Split entity file postfix' (_entity), 'Complex real part postfix' (_re), 'Split arch file postfix' (_arch), 'Complex imaginary part postfix' (_im), 'Vector prefix' (vector_of_), 'Coefficient prefix' (coeff), and 'Instance prefix' (u_).

Command Line Alternative: Use the `generatehdl` function with the property `ClockProcessPostfix` to change the postfix string appended to process labels.

Setting a Prefix for Component Instance Names

Instance prefix specifies a string to be prefixed to component instance names in generated code. The default string is `u_`.

You can set the postfix string to a value other than `u_`. To change the string:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Instance prefix** field, as shown in the following figure.

The screenshot shows the 'Additional settings' dialog box with the 'General' tab selected. The 'Instance prefix' field is highlighted in yellow and contains the text 'u_tst'. Other fields include 'Verilog file extension' (.v), 'VHDL file extension' (.vhd), 'Entity conflict postfix' (_block), 'Package postfix' (_pkg), 'Reserved word postfix' (_rsvd), 'Split entity and architecture' (checkbox), 'Clocked process postfix' (_process), 'Split entity file postfix' (_entity), 'Complex real part postfix' (_re), 'Split arch file postfix' (_arch), 'Complex imaginary part postfix' (_im), 'Coefficient prefix' (coeff), and 'Vector prefix' (vector_of_).

Command Line Alternative: Use the `generatehdl` function with the property `InstancePrefix` to change the instance prefix string.

Setting a Prefix for Vector Names

Vector prefix specifies a string to be prefixed to vector names in generated VHDL code. The default string is `vector_of_`.

Note: `VectorPrefix` is supported only for VHDL code generation.

You can set the prefix string to a value other than `vector_of_`. To change the string:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **General** tab in the **Additional settings** pane.
- 3 Enter a new string in the **Vector prefix** field, as shown in the following figure.

Additional settings

General Ports Advanced

Comment in header:

Verilog file extension: VHDL file extension:

Entity conflict postfix: Package postfix:

Reserved word postfix: Split entity and architecture

Clocked process postfix: Split entity file postfix:

Complex real part postfix: Split arch file postfix:

Complex imaginary part postfix: Vector prefix:

Coefficient prefix:

Instance prefix:

Command Line Alternative: Use the `generatehdl` function with the property `VectorPrefix` to change the instance prefix string.

Ports and Resets

In this section...

“Naming HDL Ports” on page 5-22

“Specifying the HDL Data Type for Data Ports” on page 5-23

“Selecting Asynchronous or Synchronous Reset Logic” on page 5-24

“Setting the Asserted Level for the Reset Input Signal” on page 5-25

“Suppressing Generation of Reset Logic” on page 5-27

Naming HDL Ports

The default names for filter HDL ports are as follows:

HDL Port	Default Port Name
Input port	filter_in
Output port	filter_out
Clock port	clk
Clock enable port	clk_enable
Reset port	reset
Fractional delay port (Farrow filters only)	filter_fd

For example, the default VHDL declaration for entity `Hd` looks like the following.

```
ENTITY Hd IS
  PORT( clk           : IN   std_logic;
        clk_enable    : IN   std_logic;
        reset         : IN   std_logic;
        filter_in     : IN   std_logic_vector (15 DOWNTO 0); -- sfix16_En15
        filter_out    : OUT  std_logic_vector (15 DOWNTO 0); -- sfix16_En15
        );
END Hd;
```

To change port names,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Ports** tab in the **Additional settings** pane. The following figure highlights the port name fields for **Input port**, **Output port**, **Clock input port**, **Reset input port**, and **Clock enable output port**.

The screenshot shows the 'Global Settings' tab of the 'Generate HDL' dialog box. The 'Filter Architecture' tab is selected. The 'Reset type' is set to 'Asynchronous', 'Reset asserted level' is 'Active-high', 'Clock input port' is 'clk', 'Reset input port' is 'reset', 'Clock enable input port' is 'clk_enable', and 'Remove reset from' is 'None'. The 'Clock inputs' are set to 'Single'. The 'Additional settings' pane is open to the 'Advanced' sub-tab, showing 'Input data type' as 'std_logic_vector', 'Output data type' as 'Same as input type', 'Clock enable output port' as 'ce_out', 'Input port' as 'filter_in', 'Output port' as 'filter_out', and 'Input complexity' as 'Real'. Both 'Add input register' and 'Add output register' checkboxes are checked.

- 3 Enter new strings in the port name fields, if you wish.

Command Line Alternative: Use the `generatehdl` function with the properties `InputPort`, `OutputPort`, `ClockInputPort`, `ClockEnableInputPort`, and `ResetInputPort` to change the names of a filter's VHDL ports.

Specifying the HDL Data Type for Data Ports

By default, filter input and output data ports have data type `std_logic_vector` in VHDL and type `wire` in Verilog. If you are generating VHDL code, alternatively, you can specify `signed/unsigned`, and for output data ports, `Same as input data type`. The coder applies type `SIGNED` or `UNSIGNED` based on the data type specified in the filter design.

To change the VHDL data type setting for the input and output data ports,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Ports** tab in the **Additional settings** pane.
- 3 Select a data type from the **Input data type** or **Output data type** menu identified in the following figure.

By default, the output data type is the same as the input data type.

The type for Verilog ports is `wire`, and cannot be changed.

Additional settings

General Ports **Advanced**

Input data type:

Output data type:

Clock enable output port:

Input port:

Output port:

Input complexity:

Add input register

Add output register

Note: The setting of **Input data type** does not apply to double-precision input, which is generated as type `REAL` for VHDL and `wire[63:0]` for Verilog.

Command Line Alternative: Use the `generatehdl` function with the properties `InputType` and `OutputType` to change the VHDL data type for a filter's input and output ports.

Selecting Asynchronous or Synchronous Reset Logic

By default, generated HDL code for registers uses a asynchronous reset logic. Whether you should set the reset type to asynchronous or synchronous depends on the type of device you are designing (for example, FPGA or ASIC) and preference.

The following code fragment illustrates the use of asynchronous resets. Note that the process block does not check for an active clock before performing a reset.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
  
```

```

ELSIF clk'event AND clk = '1' THEN
  IF clk_enable = '1' THEN
    delay_pipeline(0) <= signed(filter_in)
    delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
  END IF;
END IF;
END PROCESS delay_pipeline_process;

```

To change the reset type to synchronous, select **Synchronous** from the **Reset type** menu in the **Global settings** pane of the Generate HDL dialog box.

Filter Architecture	Global Settings	Test Bench	EDA Tool Scripts
Reset type:	Synchronous	Reset asserted level:	Active-high
Clock input port:	clk	Clock enable input port:	clk_enable
Reset input port:	reset	Clock inputs:	Single
Remove reset from:	None		

Code for a synchronous reset follows. This process block checks for a clock event, the rising edge, before performing a reset.

```

delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF rising_edge(clk) THEN
    IF reset = '1' THEN
      delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
    ELSIF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in)
      delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS delay_pipeline_process;

```

Command Line Alternative: Use the `generatehdl` function with the property `ResetType` to set the reset style for your filter's registers.

Setting the Asserted Level for the Reset Input Signal

The asserted level for the reset input signal determines whether that signal must be driven to active high (1) or active low (0) for registers to be reset in the filter design. By default, the coder sets the asserted level to 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'));
    .
    .
    .

```

To change the setting to active low, select **Active-low** from the **Reset asserted level** menu in the **Global settings** pane of the Generate HDL dialog box.

Filter Architecture	Global Settings	Test Bench	EDA Tool Scripts
Reset type:	Synchronous	Reset asserted level:	Active-low
Clock input port:	clk	Clock enable input port:	clk_enable
Reset input port:	reset	Clock inputs:	Single
Remove reset from:	None		

With this change, the IF statement in the preceding generated code changes to

```

IF reset = '0' THEN

```

Note: The **Reset asserted level** setting also determines the reset level for test bench reset input signals.

Command Line Alternative: Use the `generatehdl` function with the property `ResetAssertedLevel` to set the asserted level for the filter's reset input signal.

Suppressing Generation of Reset Logic

For some FPGA applications, it is desirable to avoid generation of resets. The **Remove reset from** option in the **Global settings** pane of the Generate HDL dialog box lets you suppress generation of resets from shift registers.

To suppress generation of resets from shift registers, select **Shift register** from the **Remove reset from** pull-down menu in the **Global settings** pane of the Generate HDL dialog box.

Filter Architecture	Global Settings	Test Bench	EDA Tool Scripts
Reset type:	<input type="text" value="Synchronous"/>	Reset asserted level:	<input type="text" value="Active-low"/>
Clock input port:	<input type="text" value="clk"/>	Clock enable input port:	<input type="text" value="clk_enable"/>
Reset input port:	<input type="text" value="reset"/>	Clock inputs:	<input type="text" value="Single"/>
Remove reset from:	<input type="text" value="Shift register"/>		

If you do not want to suppress generation of resets from shift registers, leave **Remove reset from** set to its default, which is **None**.

Command Line Alternative: Use the `generatehdl` function with the property `RemoveResetFrom` to suppress generation of resets from shift registers.

HDL Language Constructs

In this section...

“Representing VHDL Constants with Aggregates” on page 5-29

“Unrolling and Removing VHDL Loops” on page 5-30

“Using the VHDL rising_edge Function” on page 5-31

“Suppressing the Generation of VHDL Inline Configurations” on page 5-32

“Specifying VHDL Syntax for Concatenated Zeros” on page 5-33

“Specifying Input Type Treatment for Addition and Subtraction Operations” on page 5-34

“Suppressing Verilog Time Scale Directives” on page 5-35

“Using Complex Data and Coefficients” on page 5-36

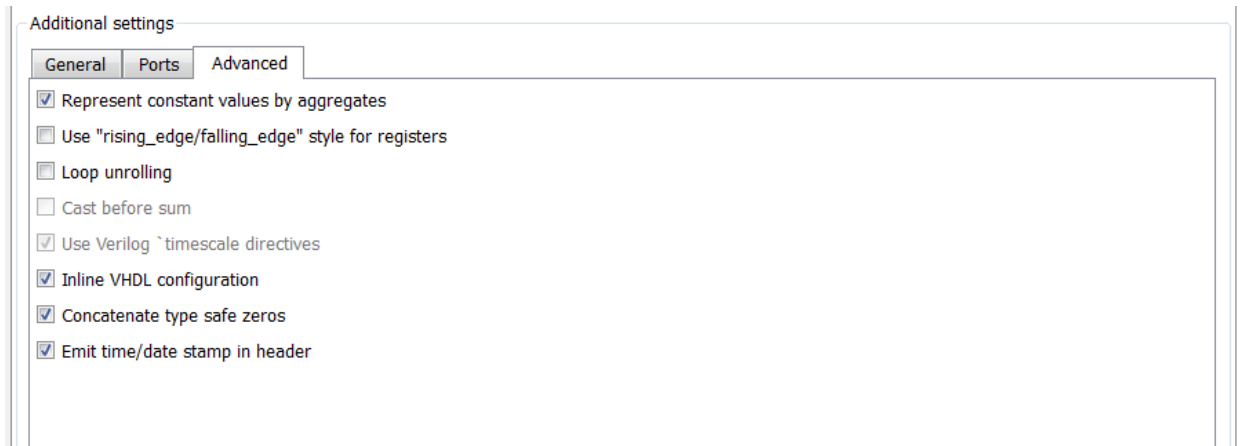
Representing VHDL Constants with Aggregates

By default, the coder represents constants as scalars or aggregates depending on the size and type of the data. The coder represents values that are less than $2^{32} - 1$ as integers and values greater than or equal to $2^{32} - 1$ as aggregates. The following VHDL constant declarations are examples of declarations generated by default for values less than 32 bits:

```
CONSTANT coeff1: signed(15 DOWNT0 0) := to_signed(-60, 16); -- sfix16_En16
CONSTANT coeff2: signed(15 DOWNT0 0) := to_signed(-178, 16); -- sfix16_En16
```

If you prefer that constant values be represented as aggregates, set the **Represent constant values by aggregates** as follows:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab.
- 3 Select **Represent constant values by aggregates**, as shown the following figure.



The preceding constant declarations would now appear as follows:

```
CONSTANT coeff1: signed(15 DOWNT0 0) := (5 DOWNT0 3 => '0', 1 DOWNT0 0 => '0', OTHERS => '1');  
CONSTANT coeff2: signed(15 DOWNT0 0) := (7 => '0', 5 DOWNT0 4 => '0', 0 => '0', OTHERS => '1');
```

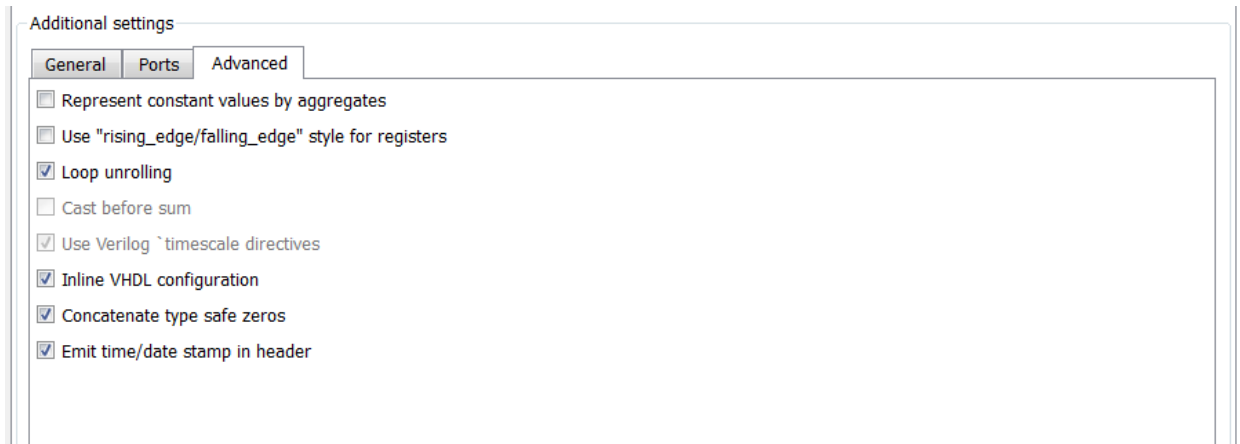
Command Line Alternative: Use the `generatehdl` function with the property `UseAggregatesForConst` to represent constants in the HDL code as aggregates.

Unrolling and Removing VHDL Loops

By default, the coder supports VHDL loops. However, some EDA tools do not support them. If you are using such a tool along with VHDL, you may have to unroll and remove FOR and GENERATE loops from your filter's generated VHDL code. Verilog code is already unrolled.

To unroll and remove FOR and GENERATE loops,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Select **Loop unrolling**, as shown in the following figure.



Command Line Alternative: Use the `generatehdl` function with the property `LoopUnrolling` to unroll and remove loops from generated VHDL code.

Using the VHDL `rising_edge` Function

The coder can generate two styles of VHDL code for checking for rising edges when the filter operates on registers. By default, the generated code checks for a clock event, as shown in the `ELSIF` statement of the following VHDL process block.

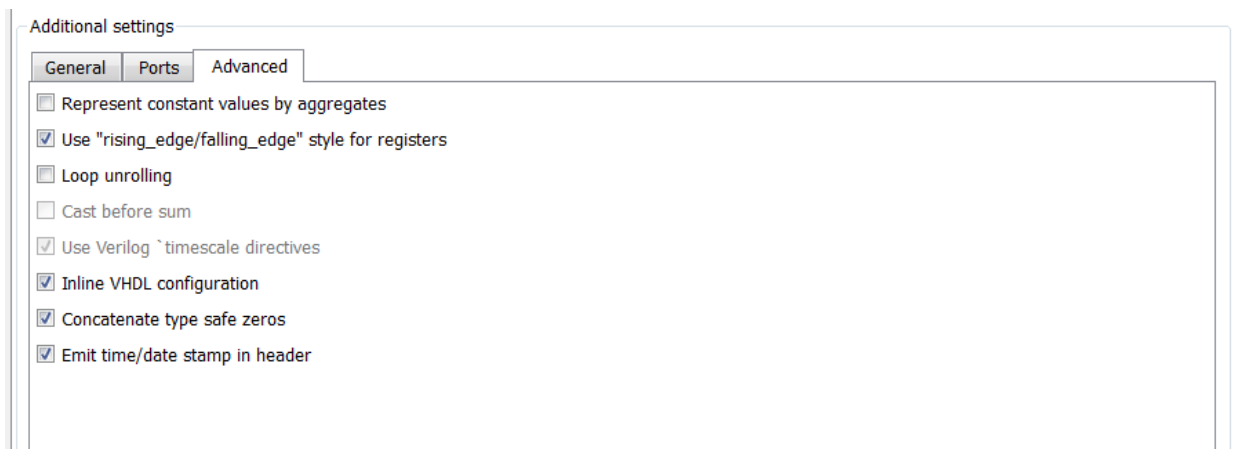
```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
    delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS Delay_Pipeline_Process ;
```

If you prefer, the coder can produce VHDL code that applies the VHDL `rising_edge` function instead. For example, the `ELSIF` statement in the preceding process block would be replaced with the following statement:

```
ELSIF rising_edge(clk) THEN
```

To use the `rising_edge` function,

- 1 Click **Global Settings** in the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Select **Use 'rising_edge' for registers**, as shown in the following dialog box.



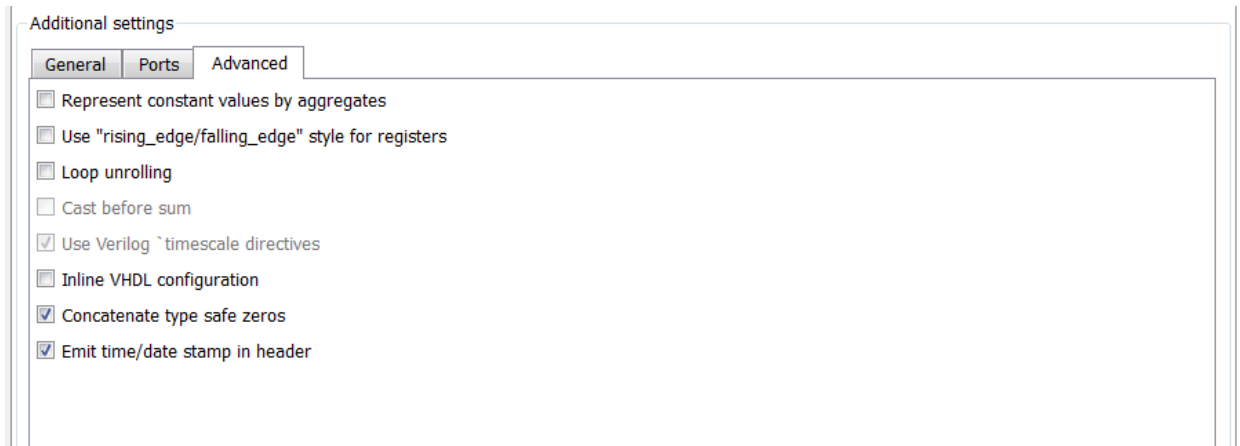
Command Line Alternative: Use the `generatehdl` function with the property `UseRisingEdge` to use the VHDL `rising_edge` function to check for rising edges during register operations.

Suppressing the Generation of VHDL Inline Configurations

VHDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, the coder includes configurations for a filter within the generated VHDL code. If you are creating your own VHDL configuration files, you should suppress the generation of inline configurations.

To suppress the generation of inline configurations,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Clear **Inline VHDL configuration**, as shown in the following figure.



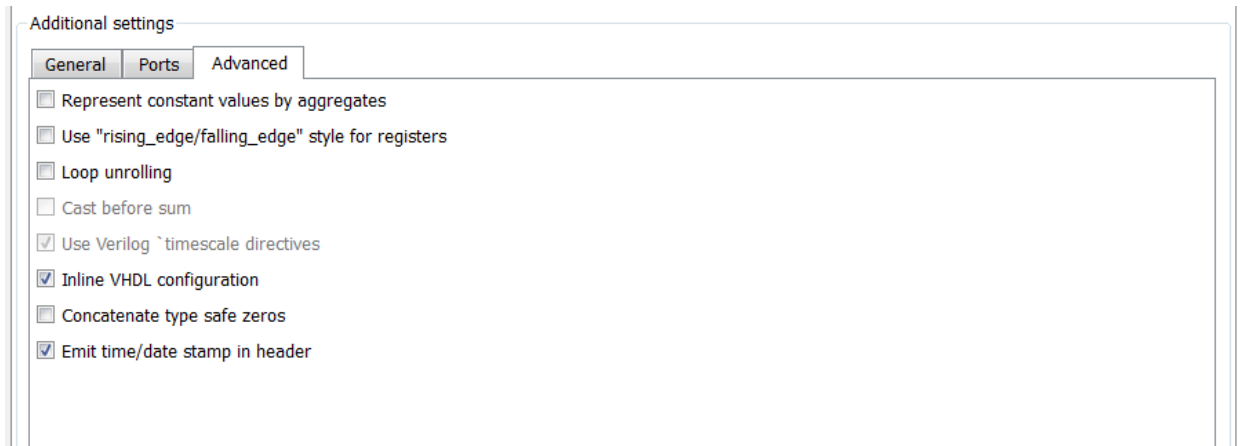
Command Line Alternative: Use the `generatehdl` function with the property `InlineConfigurations` to suppress the generation of inline configurations.

Specifying VHDL Syntax for Concatenated Zeros

In VHDL, the concatenation of zeros can be represented in two syntax forms. One form, `'0' & '0'`, is type-safe. This is the default. The alternative syntax, `"000000..."`, can be easier to read and is more compact, but can lead to ambiguous types.

To use the syntax `"000000..."` for concatenated zeros,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Clear **Concatenate type safe zeros**, as shown in the following figure.



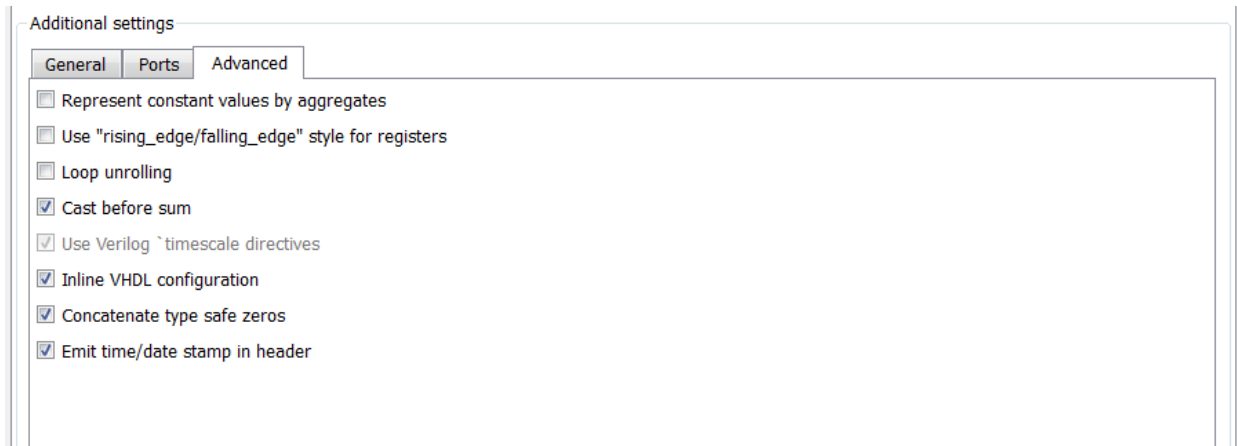
Command Line Alternative: Use the `generatehdl` function with the property `SafeZeroConcat` to use the syntax "000000...", for concatenated zeros.

Specifying Input Type Treatment for Addition and Subtraction Operations

By default, generated HDL code operates on input data using data types as specified by the filter design, and then converts the result to the specified result type.

Typical DSP processors type cast input data to the result type *before* operating on the data. Depending on the operation, the results can be very different. If you want generated HDL code to handle result typing in this way, use the **Cast before sum** option as follows:

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Select **Cast before sum**, as shown in the following figure.



Command Line Alternative: Use the `generatehdl` function with the property `CastBeforeSum` to cast input values to the result type for addition and subtraction operations.

Relationship Between Cast Before Sum and Cast Before Accum.

The **Cast before sum** property is related to the FDATool setting for the quantization property **Cast signals before accum.** as follows:

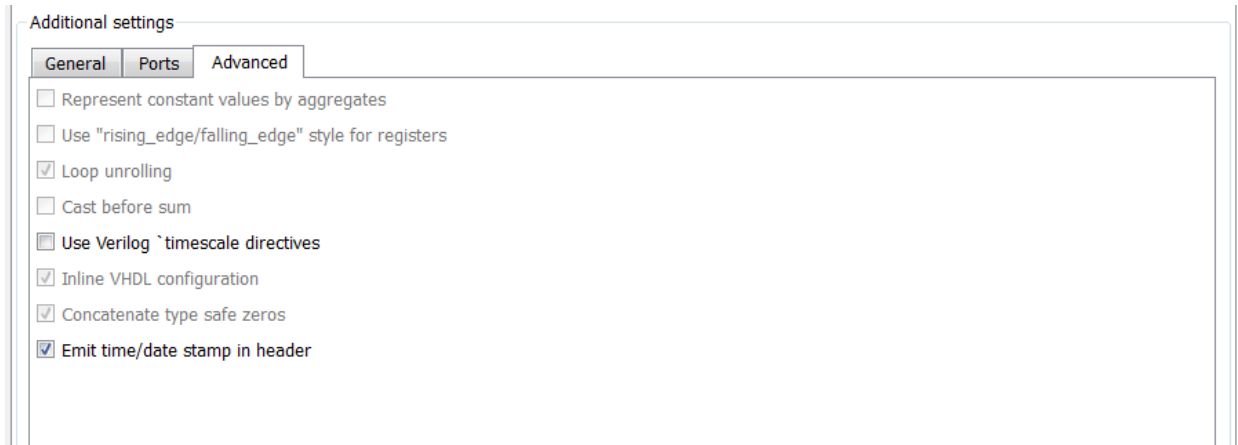
- Some filter object types do not have the **Cast signals before accum.** property. For such filter objects, **Cast before sum** is effectively off when HDL code is generated; it is not relevant to the filter.
- Where the filter object does have the **Cast signals before accum.** property, the coder by default follows the setting of **Cast signals before accum.** in the filter object. This is visible in the GUI. If you change the setting of **Cast signals before accum.**, the coder updates the setting of **Cast before sum.**
- However, by explicitly setting **Cast before sum**, you can override the **Cast signals before accum.** setting passed in from FDATool.

Suppressing Verilog Time Scale Directives

In Verilog, the coder generates time scale directives (``timescale`) by default. This compiler directive provides a way of specifying different delay values for multiple modules in a Verilog file.

To suppress the use of ``timescale` directives,

- 1 Select the **Global Settings** tab on the Generate HDL dialog box.
- 2 Select the **Advanced** tab. The **Advanced** pane appears.
- 3 Clear **Use Verilog ``timescale` directives**, as shown in the following figure.



Command Line Alternative: Use the `generatehdl` function with the property `UseVerilogTimescale` to suppress the use of time scale directives.

Using Complex Data and Coefficients

The coder supports use of complex coefficients and complex input signals for fully parallel FIR, CIC, and some other filter structures. In many cases, you can use complex data and complex coefficients in combination. The following table shows the filter structures that support complex data and/or coefficients, and the permitted combinations.

Filter Structure	Complex Data	Complex Coefficients	Both Complex Data and Coefficients
<code>dfilt.dffir</code>	Y	Y	Y
<code>dfilt.dfsymfir</code>	Y	Y	Y
<code>dfilt.dfasymfir</code>	Y	Y	Y
<code>dfilt.dffirt</code>	Y	Y	Y

Filter Structure	Complex Data	Complex Coefficients	Both Complex Data and Coefficients
dfilt.scalar	Y	Y	Y
dfilt.delay	Y	N/A	N/A
mfilt.cicdecim	Y	N/A	N/A
mfilt.cicinterp	Y	N/A	N/A
mfilt.firdecim	Y	Y	Y
mfilt.firinterp	Y	Y	Y
mfilt.firsrc	Y	Y	Y
mfilt.firtdecim	Y	Y	Y
mfilt.linearinterp	Y	N/A	N/A
mfilt.holdinterp	Y	Y	N/A
dfilt.df1sos	Y	Y	Y
dfilt.df1tsos	Y	Y	Y
dfilt.df2sos	Y	Y	Y
dfilt.df2tsos	Y	Y	Y

Enabling Code Generation for Complex Data

The option you choose from the **Input complexity** menu instructs the coder whether or not to generate corresponding ports and signal paths for the real and imaginary components of a complex signal. The default setting for **Input complexity** is **Real**, disabling generation of ports for complex input data. To enable generation of ports for complex input data, set **Input complexity** to **Complex**.

The corresponding command line property is `InputComplex`. By default, `InputComplex` is set to `'off'`, disabling generation of ports for complex input data. To enable generation of ports for complex input data, set `InputComplex` to `'on'`, as in the following code example:

```
Hd = design(fdesign.Lowpass,'equiripple','Filterstructure','dffir')
generatehdl(Hd, 'InputComplex', 'on')
```

The following VHDL code excerpt shows the entity definition generated by the preceding commands:

```
ENTITY Hd IS
  PORT( clk           : IN   std_logic;
        clk_enable    : IN   std_logic;
        reset         : IN   std_logic;
        filter_in_re   : IN   real; -- double
        filter_in_im   : IN   real; -- double
        filter_out_re  : OUT  real; -- double
        filter_out_im  : OUT  real  -- double
        );
END Hd;
```

In the code excerpt, the port names generated for the real components of complex signals are identified by the default postfix string `'_re'`, and port names generated for the imaginary components of complex signals are identified by the default postfix string `'_im'`.

Setting the Port Name Postfix for Complex Ports

Two code generation properties let you customize naming conventions for the real and imaginary components of complex signals in generated HDL code. These properties are:

- The **Complex real part postfix** option (corresponding to the `ComplexRealPostfix` command line property) specifies a string to be appended to the names generated for the real part of complex signals. The default postfix is `'_re'`. See also `ComplexRealPostfix`.
- The **Complex imaginary part postfix** option (corresponding to the `ComplexImagPostfix` command line property) specifies a string to be appended to the names generated for the imaginary part of complex signals. The default postfix is `'_im'`. See also `ComplexImagPostfix`.

Verification of Generated HDL Filter Code

- “Testing with an HDL Test Bench” on page 6-2
- “Cosimulation of HDL Code with HDL Simulators” on page 6-27
- “Integration With Third-Party EDA Tools” on page 6-36

Testing with an HDL Test Bench

In this section...

“Workflow for Testing With an HDL Test Bench” on page 6-2

“Enabling Test Bench Generation” on page 6-9

“Renaming the Test Bench” on page 6-11

“Specifying a Test Bench Type” on page 6-12

“Splitting Test Bench Code and Data into Separate Files” on page 6-14

“Configuring the Clock” on page 6-15

“Configuring Resets” on page 6-17

“Setting a Hold Time for Data Input Signals” on page 6-20

“Setting an Error Margin for Optimized Filter Code” on page 6-22

“Setting an Initial Value for Test Bench Inputs” on page 6-24

“Setting Test Bench Stimuli” on page 6-25

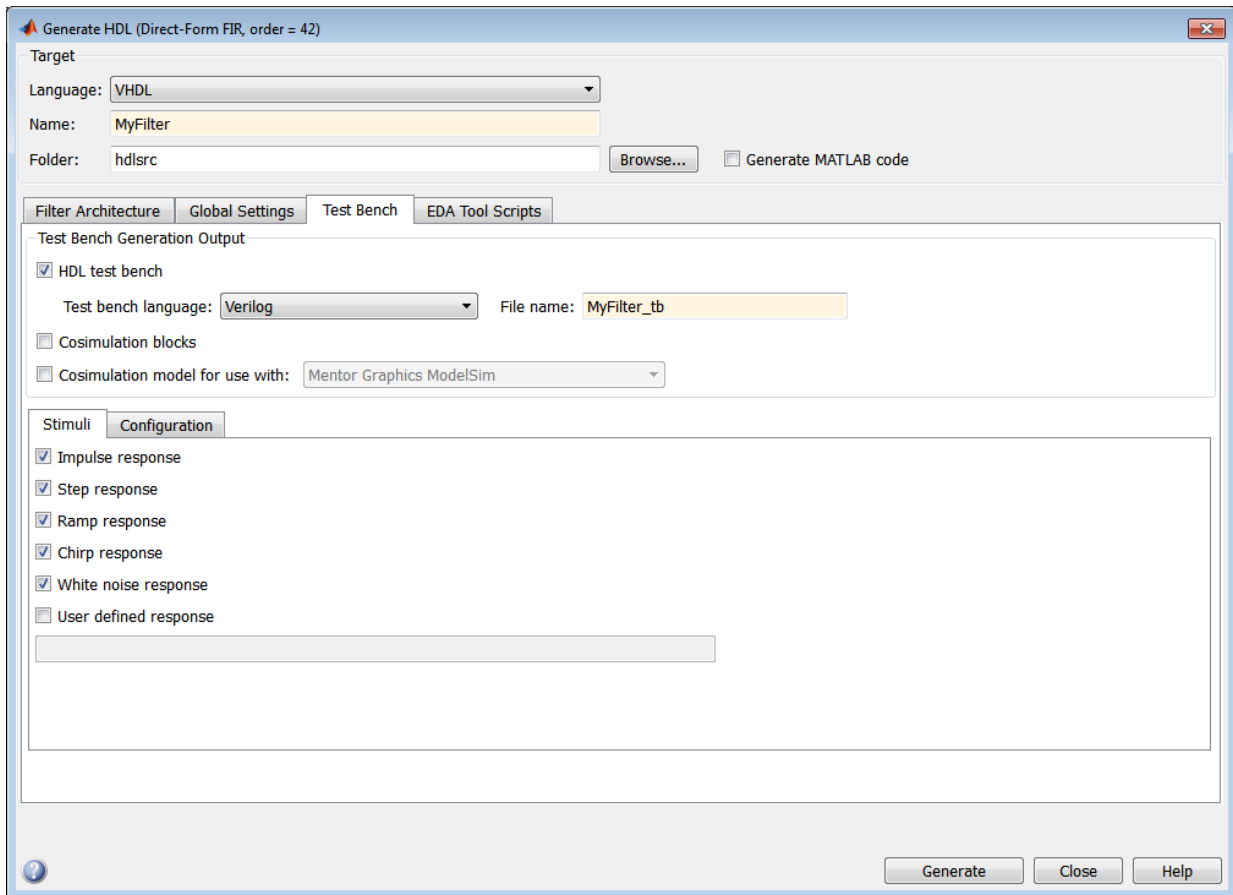
“Setting a Postfix for Reference Signal Names” on page 6-26

Workflow for Testing With an HDL Test Bench

Generating the Filter and Test Bench HDL Code

Use the Filter Design HDL Coder GUI or command line interface to generate the HDL code for your filter design and test bench. As explained in “Specifying a Test Bench Type” on page 6-12, the GUI generates a VHDL or Verilog test bench file by default, depending on your language selection. To specify a language-specific test bench type explicitly, select the **Test bench language** option in the **Test Bench** pane of the Generate HDL dialog box.

The following figure shows settings for generating the filter (VHDL) and test bench (Verilog) files `MyFilter.vhd`, and `MyFilter_tb.v`. The dialog box also specifies that the generated files are to be placed in the default target folder `hdlsrc` under the current working folder.



After you click **Generate**, the coder displays progress information similar to the following in the MATLAB Command Window:

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\Work\sl_hdlcoder_work\hdlsrc\MyFilter.vhd
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: MyFilter

### Starting generation of VERILOG Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating Test bench: C:\Work\sl_hdlcoder_work\hdlsrc\MyFilter_tb.v
```

```
### Please wait ...  
### Done generating VERILOG Test Bench
```

Note: The length of the input stimulus samples varies from filter to filter. For example, the value 3429 in the preceding message sequence is not fixed; the value is dependent on the filter under test.

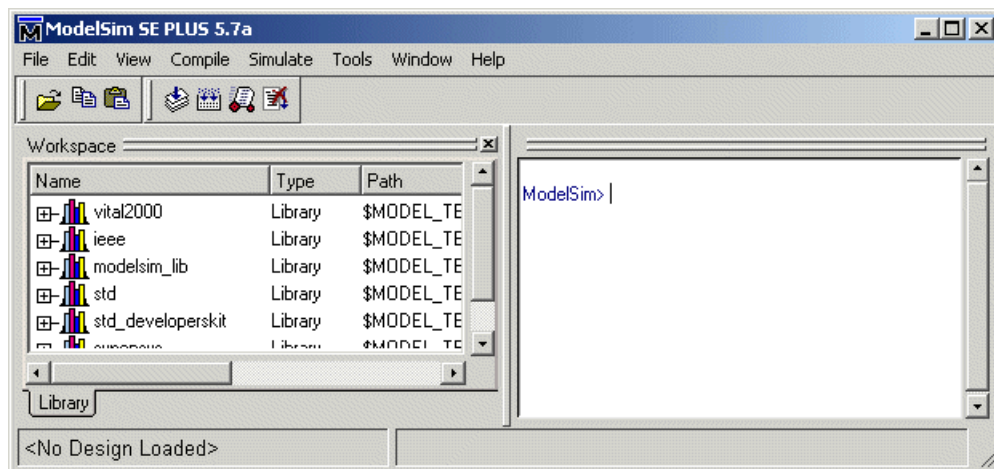
If you call the `generatehdl` function from the command line interface, you must

- Specify 'VHDL' or 'Verilog' for the `TbType` parameter.
- For double-precision filters, you must specify the type that matches the target language specified for your filter code.
- You can use the function `generatetbstimulus` to return the test bench stimulus to the MATLAB Command Window.

See `generatehdl` function reference for details on the property name and property value pairs that you can specify for customizing the output.

Starting the Simulator

After you generate your filter and test bench HDL files, start your simulator. When you start the Mentor Graphics ModelSim simulator, a screen display similar to the following appears:



After starting the simulator, set the current folder to the folder that contains your generated HDL files.

Compiling the Generated Filter and Test Bench Files

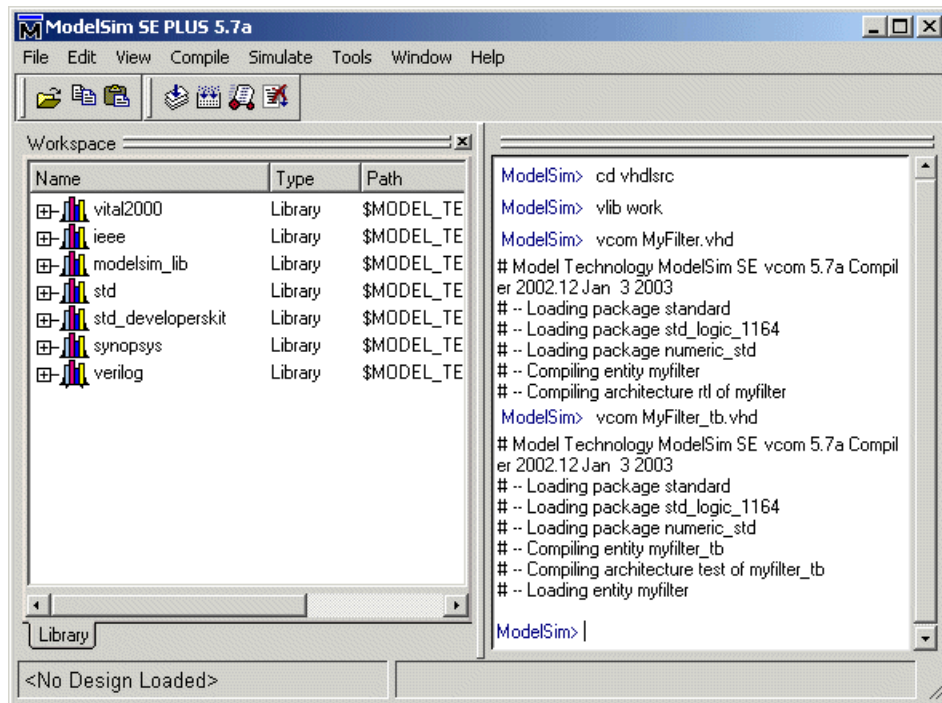
Using your choice HDL compiler, compile the generated filter and test bench HDL files. Depending on the language of the generated test bench and the simulator you are using, you may have to complete some precompilation setup. For example, in the Mentor Graphics ModelSim simulator, you might choose to create a design library to store compiled VHDL entities, packages, architectures, and configurations.

The following Mentor Graphics ModelSim command sequence changes the current folder to `hdlsrc`, creates the design library `work`, and compiles VHDL filter and filter test bench code. The `vlib` command creates the design library `work` and the `vcom` commands initiate the compilations.

```
cd hdlsrc
vlib work
vcom MyFilter.vhd
vcom MyFilter_tb.vhd
```

Note: For VHDL test bench code that has floating-point (double) realizations, use a compiler that supports VHDL-93 or VHDL-02 (for example, in the Mentor Graphics ModelSim simulator, specify the `vcom` command with the `-93` option). Do not compile the generated test bench code with a VHDL-87 compiler. VHDL test benches using double-precision data types do not support VHDL-87, because test bench code uses the `image` attribute, which is available only in VHDL-93 or higher.

The following screen display shows this command sequence and informational messages displayed during compilation.

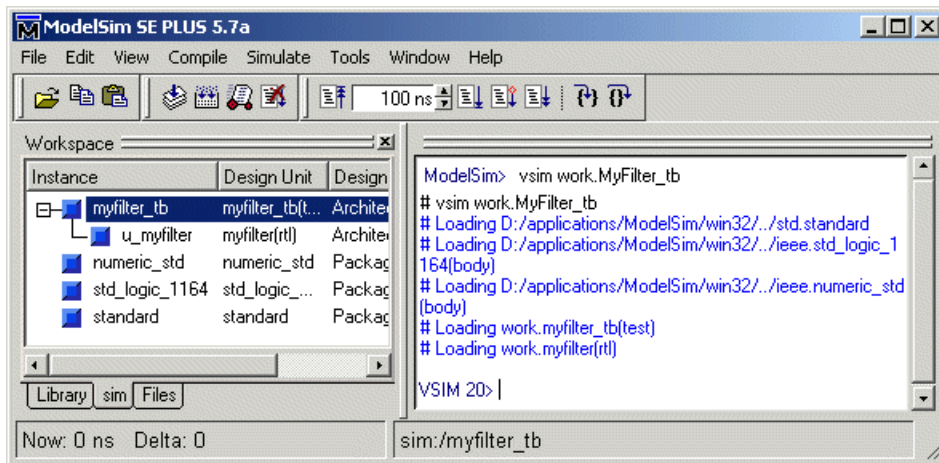


Running the Test Bench Simulation

Once your generated HDL files are compiled, load and run the test bench. The procedure for doing this varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.MyFilter_tb
```

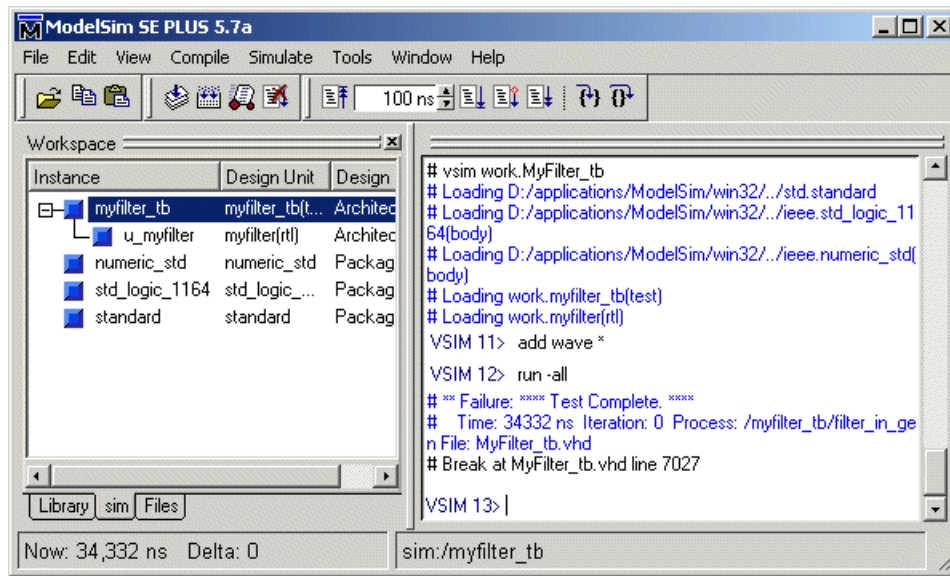
The following display shows the results of loading `work.MyFilter_tb` with the `vsim` command.



Once the design is loaded into the simulator, consider opening a display window for monitoring the simulation as the test bench runs. For example, in the Mentor Graphics ModelSim simulator, you might use the `add wave *` command to open a **wave** window to view the results of the simulation as HDL waveforms.

To start running the simulation, issue the start simulator command. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the `run -all` command.

The following display shows the `add wave *` command being used to open a **wave** window and the `-run all` command being used to start a simulation.



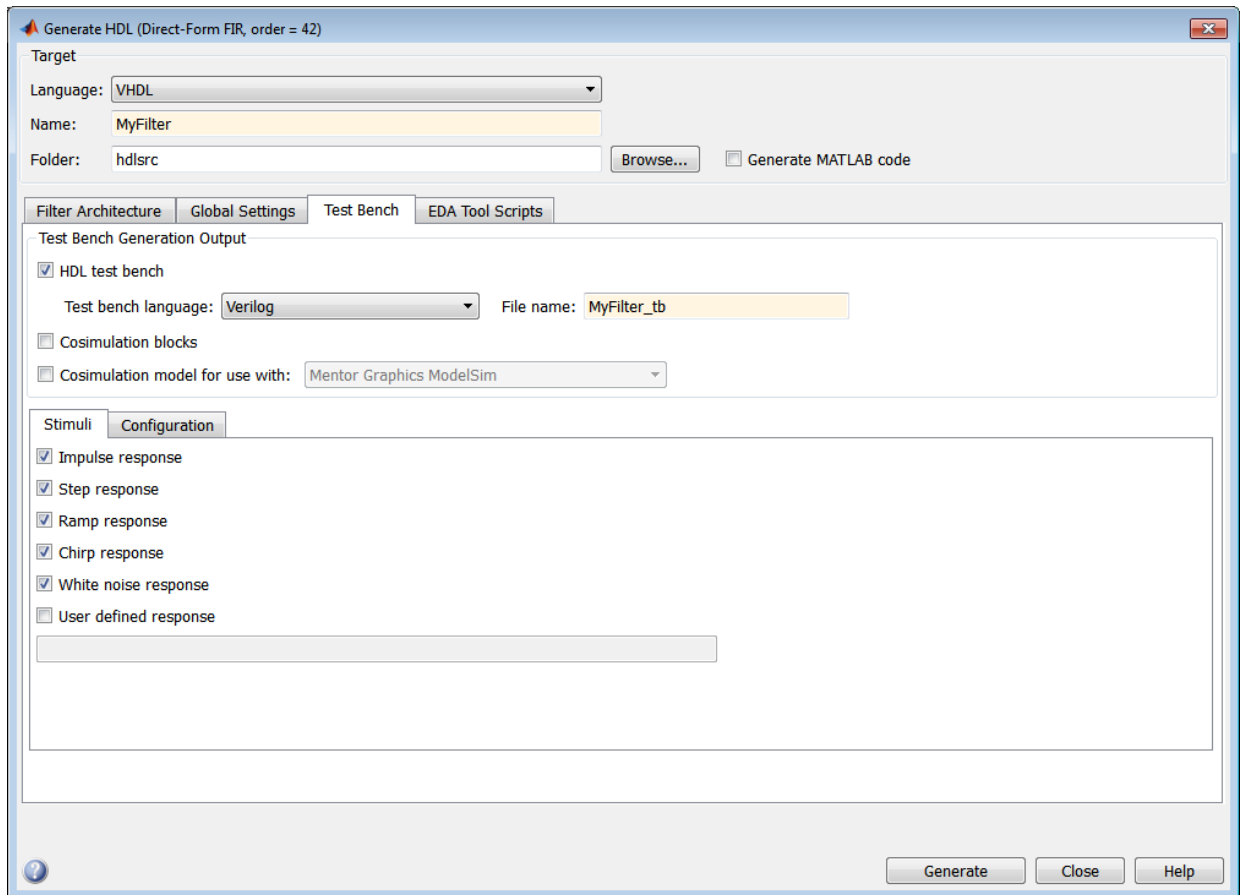
As your test bench simulation runs, watch for error messages. If error messages appear, you must interpret them as they pertain to your filter design and the code generation options you applied. For example, a number of HDL customization options allow you to specify settings that can produce numeric results that differ from those produced by the original filter object. For HDL test benches, expected and actual results are compared. If they differ (excluding the specified error margin), an error message similar to the following is returned:

```
Error in filter test: Expected xxxxxxxx Actual xxxxxxxx
```

You must determine whether the actual results are expected based on the customizations you specified when generating the filter HDL code.

Note: The failure message that appears in the preceding display is not flagging an error. If the message includes the string `Test Complete`, the test bench has run to completion without encountering an error. The `Failure` part of the message is tied to the mechanism the coder uses to end the simulation.

The following **wave** window shows the simulation results as HDL waveforms.



3 Click **Generate** to generate HDL and test bench code.

Tip By default, **HDL test bench** is selected.

Command Line Alternative: Use the `generatehdl` function with the property `GenerateHDLTestbench` to generate an HDL test bench.

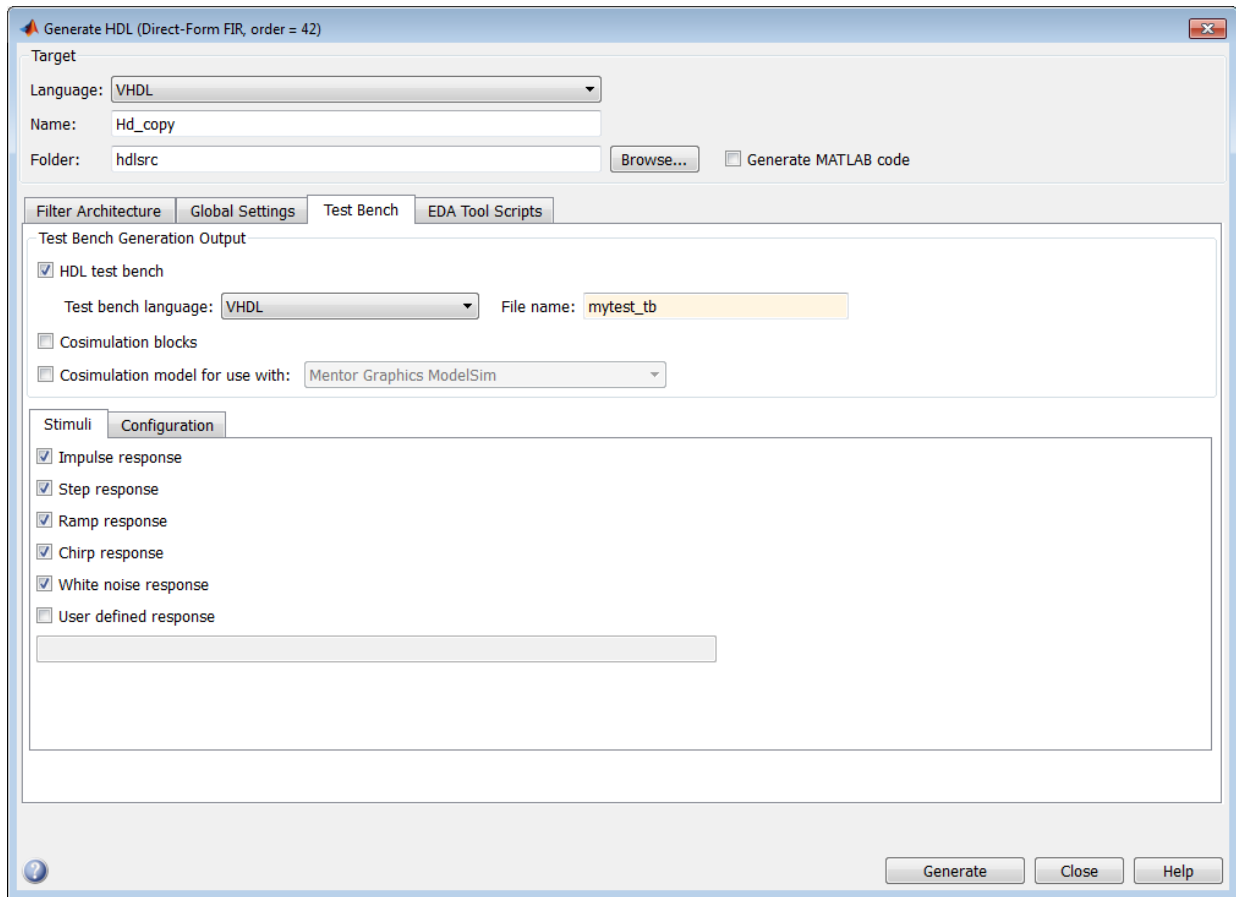
Renaming the Test Bench

The coder derives the name of the test bench file from the name of the quantized filter for which the HDL code is being generated and the postfix `_tb`. The file type extension depends on the type of test bench that is being generated.

If the Test Bench Is a...	The Extension Is...
Verilog file	Defined by the Verilog file extension field in the General subpane of the Global Settings pane of the Generate HDL dialog box
VHDL file	Defined by the VHDL file extension field in the Global Settings pane of the Generate HDL dialog box

The file is placed in the folder defined by the **Folder** option in the **Target** pane of the Generate HDL dialog box.

To specify a test bench name, enter the name in the **Name** field of the **Test bench settings** pane, as shown in the following figure.



Note: If you enter a string that is a VHDL or Verilog reserved word, the coder appends the reserved word postfix to the string to form a valid identifier.

Command Line Alternative: Use the `TestBenchName` to property to specify a name for your filter's test bench.

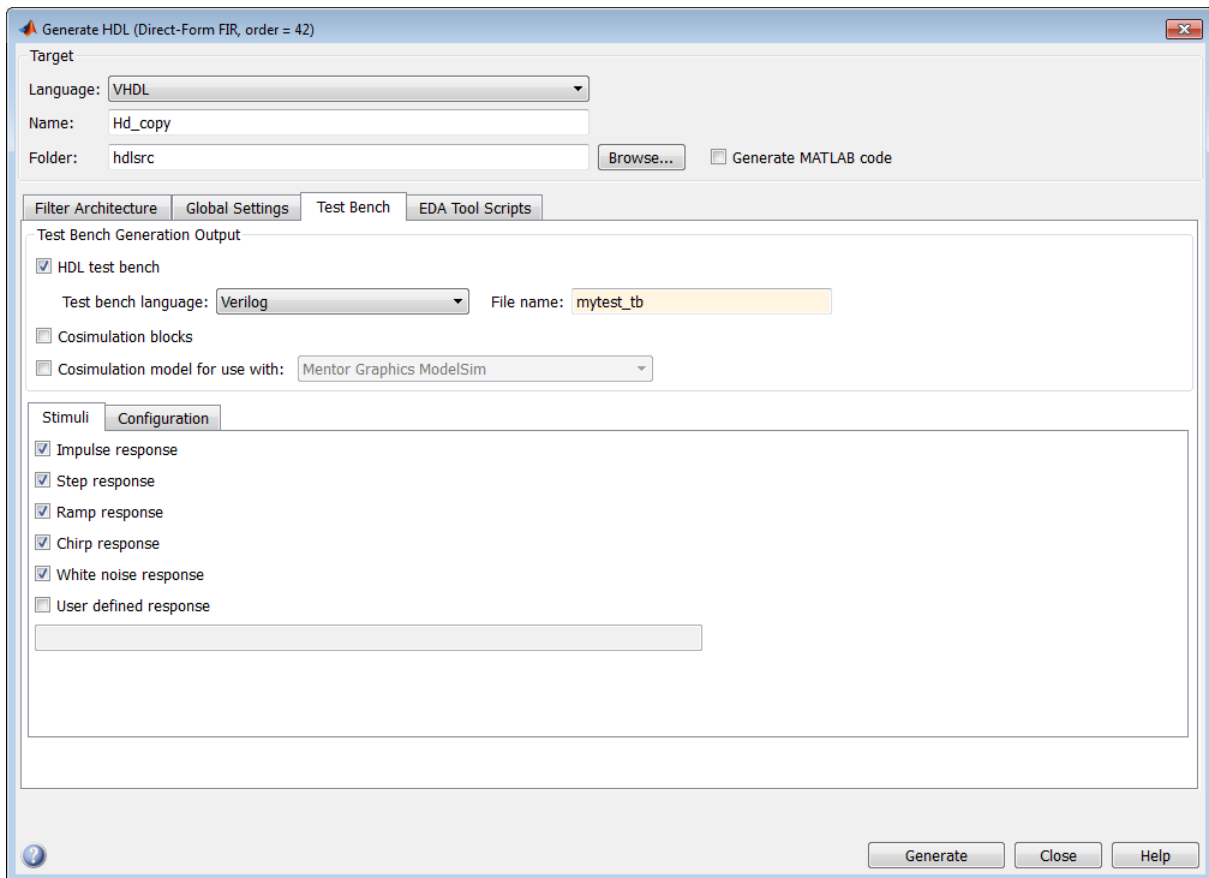
Specifying a Test Bench Type

Specifying a Test Bench Type

The coder can generate two types of test benches:

- A VHDL file that you can simulate in a simulator of choice
- A Verilog file that you can simulate in a simulator of choice

By default, the coder produces a VHDL or Verilog file, depending on your **Language** selection. Use the **Test bench language** pulldown menu in the **Test bench generate output** panel of the Generate HDL dialog box. to select a test bench language that differs from the **Language** selection. In the following figure, the **Language** is VHDL, while the **Test bench language** is Verilog.



Command Line Alternative: Use the `TbType` parameter to specify the type of test bench files to be generated.

Splitting Test Bench Code and Data into Separate Files

By default, the coder generates a single test bench file, containing test bench helper functions, data, and test bench code. You can split these elements into separate files by selecting the **Multi-file test bench** option in the **Configuration** subpane of the **Test Bench** pane of the Generate HDL dialog box, as shown below.

The screenshot shows the 'Test Bench' configuration window with the following settings:

- Test Bench Generation Output:**
 - HDL test bench
 - Test bench language: **VHDL**
 - File name: **mytest_tb**
 - Cosimulation blocks
 - Cosimulation model for use with: **Mentor Graphics ModelSim**
- Configuration:**
 - Force clock
 - Clock high time (ns): **5**
 - Clock low time (ns): **5**
 - Hold time (ns): **2**
 - Setup time (ns): **8**
 - Force clock enable
 - Clock enable delay (in clock cycles): **1**
 - Force reset
 - Reset length (in clock cycles): **2**
 - Hold input data between samples
 - Initialize test bench inputs
 - Multi-file test bench
 - Test bench data file name postfix: **_data**
 - Test bench reference postfix: **_ref**
 - Simulator flags:

When you select the **Multi-file test bench** option, the **Test bench data file name postfix** option is enabled. The test bench file names are then derived from the name of the test bench, the **Test bench name** option, and the **Test bench data file name postfix** option as follows:

TestBenchName_TestBenchDataPostfix

For example, if the test bench name is `my_fir_filt`, and the target language is VHDL, the default test bench file names are:

- `my_fir_filt_tb.vhd`: test bench code
- `my_fir_filt_tb_pkg.vhd`: helper functions package
- `my_fir_filt_tb_data.vhd`: data package

If the filter name is `my_fir_filt` and the target language is Verilog, the default test bench file names are:

- `my_fir_filt_tb.v`: test bench code
- `my_fir_filt_tb_pkg.v`: helper functions package
- `my_fir_filt_tb_data.v`: test bench data

Command Line Alternative: Use the `generatehdl` properties `MultifileTestBench`, `TestBenchDataPostFix`, and `TestBenchName` to generate and name separate test bench helper functions, data, and test bench code files.

Configuring the Clock

Based on default settings, the coder configures the clock for a filter test bench such that it

- Forces clock enable input signals to active high (1).
- Asserts the clock enable signal 1 clock cycle after deassertion of the reset signal.
- Forces clock input signals low (0) for a duration of 5 nanoseconds and high (1) for a duration of 5 nanoseconds.

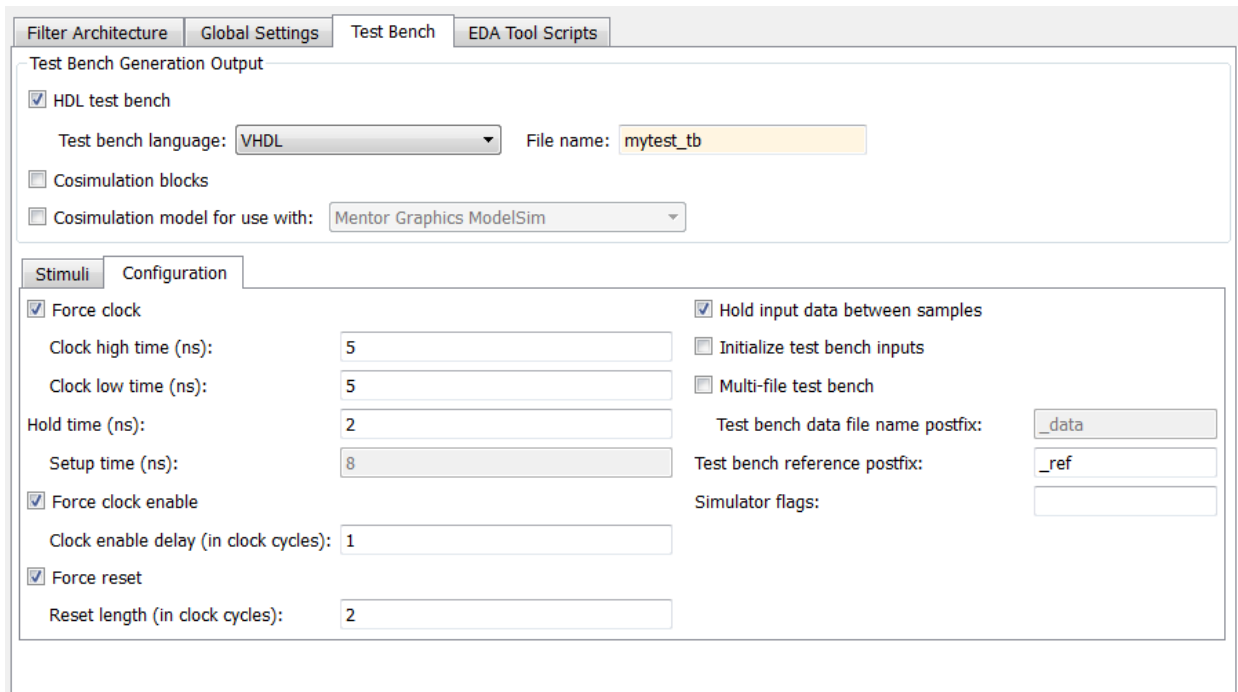
To change these clock configuration settings:

- 1 Click **Configuration** in the **Test bench** pane of the Generate HDL dialog box.
- 2 Within the **Test Bench** pane, select the **Configuration** subpane.
- 3 Make the following configuration changes as described in the following table:

If You Want to...	Then...
Disable the forcing of clock enable input signals	Clear Force clock enable .
Disable the forcing of clock input signals	Clear Force clock .

If You Want to...	Then...
Reset the number of nanoseconds during which clock input signals are to be driven low (0)	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point) in the Clock low time field.
Reset the number of nanoseconds during which clock input signals are to be driven high (1)	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point) in the Clock high time field.
Change the delay time elapsed between the deassertion of the reset signal and the assertion of clock enable signal.	Specify a positive integer in the Clock enable delay field.

The following figure highlights the applicable options.



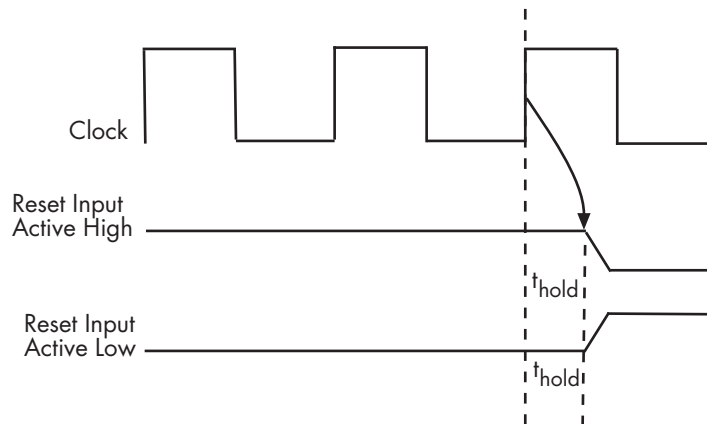
Command Line Alternative: Use the `generatehdl` properties `ForceClock`, `ClockHighTime`, `ClockLowTime`, `ForceClockEnable`, and `TestBenchClockEnableDelay` to reconfigure the test bench clock.

Configuring Resets

Based on default settings, the coder configures the reset for a filter test bench such that it

- Forces reset input signals to active high (1). (Test bench reset input levels are set by the **Reset asserted level** option).
- Asserts reset input signals for a duration of 2 clock cycles.
- Applies a hold time of 2 nanoseconds for reset input signals.

The hold time is the amount of time (after some number of initial clock cycles defined by the **Reset length** option) that reset input signals are to be held past the clock rising edge. The following figure shows the application of a hold time (t_{hold}) for reset input signals when the signals are forced to active high and active low. The default **Reset length** of 2 clock cycles is shown.



Note: The hold time applies to reset input signals only if the forcing of reset input signals is enabled.

The following table summarizes the reset configuration settings,

If You Want to...	Then...
Disable the forcing of reset input signals	Clear Force reset in the Test Bench pane of the Generate HDL dialog box.
Change the length of time (in clock cycles) during which reset is asserted	Set Reset length (in clock cycles) to an integer greater than or equal to 0. This option is located in the Test Bench pane of the Generate HDL dialog box.
Change the reset value to active low (0)	Select Active - low from the Reset asserted level menu in the Global Settings pane of the Generate HDL dialog box (see “Setting the Asserted Level for the Reset Input Signal”)
Set the hold time	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point), representing nanoseconds, in the Hold time field. When the Hold time changes, the Setup time (ns) value (displayed below Hold time) is updated. The Setup time (ns) value computed as (clock period - HoldTime) in nanoseconds. These options are in the Test Bench pane of the Generate HDL dialog box.

The following figures highlight the applicable options.

Filter Architecture	Global Settings	Test Bench	EDA Tool Scripts
Reset type:	Asynchronous	Reset asserted level:	Active-high
Clock input port:	clk	Clock enable input port:	clk_enable
Reset input port:	reset	Clock inputs:	Single
Remove reset from:	None		
Additional settings			
General Ports Advanced			
Comment in header:			
Verilog file extension:	.v	VHDL file extension:	.vhd
Entity conflict postfix:	_block	Package postfix:	_pkg
Reserved word postfix:	_rsvd	<input type="checkbox"/> Split entity and architecture	
Clocked process postfix:	_process	Split entity file postfix:	_entity
Complex real part postfix:	_re	Split arch file postfix:	_arch
Complex imaginary part postfix:	_im	Vector prefix:	vector_of_
Coefficient prefix:	coeff		
Instance prefix:	u_		

Filter Architecture Global Settings Test Bench EDA Tool Scripts

Test Bench Generation Output

HDL test bench

Test bench language: File name:

Cosimulation blocks

Cosimulation model for use with:

Stimuli Configuration

Force clock

Clock high time (ns):

Clock low time (ns):

Hold time (ns):

Setup time (ns):

Force clock enable

Clock enable delay (in clock cycles):

Force reset

Reset length (in clock cycles):

Hold input data between samples

Initialize test bench inputs

Multi-file test bench

Test bench data file name postfix:

Test bench reference postfix:

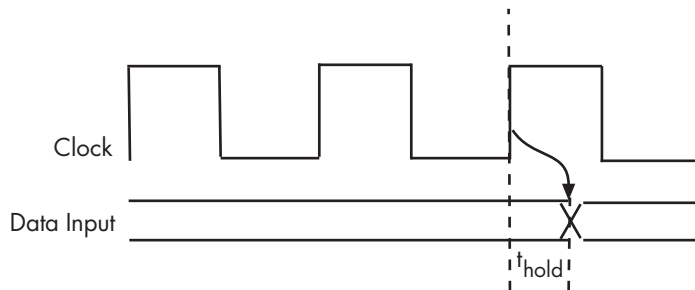
Simulator flags:

Note: The hold time and setup time settings also apply to data input signals.

Command Line Alternative: Use the `generatehdl` properties `ForceReset`, `ResetLength`, and `HoldTime` to reconfigure test bench resets.

Setting a Hold Time for Data Input Signals

By default, the coder applies a hold time of 2 nanoseconds for filter data input signals. The hold time is the amount of time that data input signals are to be held past the clock rising edge. The following figure shows the application of a hold time (t_{hold}) for data input signals.



To change the hold time setting,

- 1 Click the **Test Bench** tab in the Generate HDL dialog box.
- 2 Within the **Test Bench** pane, select the **Configuration** subpane.
- 3 Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point), representing nanoseconds, in the **Hold time** field. In the following figure, the hold time is set to 2 nanoseconds.

When the **Hold time** changes, the **Setup time (ns)** value (displayed below **Hold time**) updates. The coder computes the **Setup time (ns)** value as (clock period - HoldTime) in nanoseconds. **Setup time (ns)** is a display-only field.

The screenshot shows a software interface with two tabs: "Test Bench Generation Output" and "Stimuli Configuration".

Test Bench Generation Output:

- HDL test bench
 - Test bench language: VHDL (dropdown)
 - File name: mytest_tb (text field)
- Cosimulation blocks
- Cosimulation model for use with: Mentor Graphics ModelSim (dropdown)

Stimuli Configuration:

- Force clock
 - Clock high time (ns): 5 (text field)
 - Clock low time (ns): 5 (text field)
 - Hold time (ns): 2 (text field)
 - Setup time (ns): 8 (text field)
- Force clock enable
 - Clock enable delay (in clock cycles): 1 (text field)
- Force reset
 - Reset length (in clock cycles): 2 (text field)
- Hold input data between samples
- Initialize test bench inputs
- Multi-file test bench
 - Test bench data file name postfix: _data (text field)
 - Test bench reference postfix: _ref (text field)
- Simulator flags: (empty text field)

Note: The hold time and setup time settings also apply to reset input signals, if the forcing of such signals is enabled.

Command Line Alternative: Use the `generatehdl` property `HoldTime` to adjust the hold time setting.

Setting an Error Margin for Optimized Filter Code

Customizations that provide optimizations can generate test bench code that produces numeric results that differ from results produced by the original filter object. These options include:

- **Optimize for HDL**
- **FIR adder style** set to Tree

- **Add pipeline registers** for FIR, asymmetric FIR, and symmetric FIR filters

If you choose to use these options, consider setting an error margin for the generated test bench to account for differences in numeric results. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin:

- 1 Select the **Test Bench** pane in the Generate HDL dialog box.
- 2 Within the **Test Bench** pane, select the **Configuration** subpane.
- 3 For fixed-point filters, the initial **Error margin (bits)** field has a default value of 4. If you wish to change the error margin, enter an integer in the **Error margin (bits)** field. In the following figure, the error margin is set to 4 bits.

The screenshot shows the 'Test Bench' configuration window with the following settings:

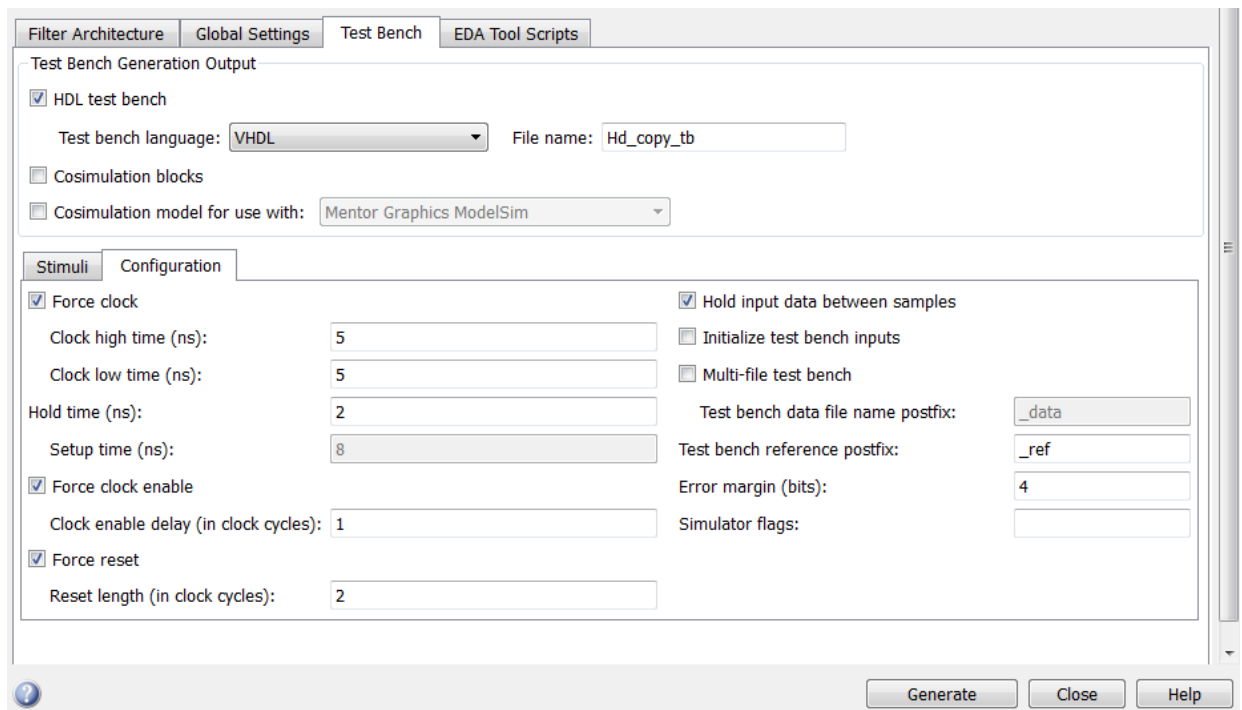
- Test Bench Generation Output:**
 - HDL test bench
 - Test bench language: VHDL
 - File name: Hd_copy_tb
 - Cosimulation blocks
 - Cosimulation model for use with: Mentor Graphics ModelSim
- Configuration Sub-pane:**
 - Force clock
 - Clock high time (ns): 5
 - Clock low time (ns): 5
 - Force clock enable
 - Clock enable delay (in clock cycles): 1
 - Force reset
 - Reset length (in clock cycles): 2
 - Hold input data between samples
 - Initialize test bench inputs
 - Multi-file test bench
 - Test bench data file name postfix: _data
 - Test bench reference postfix: _ref
 - Error margin (bits): 4
 - Simulator flags: (empty)

Buttons at the bottom: Generate, Close, Help.

Setting an Initial Value for Test Bench Inputs

By default, the initial value driven on test bench inputs is 'X' (unknown). Alternatively, you can specify that the initial value driven on test bench inputs is 0, as follows:

- 1 Select the **Test Bench** pane in the Generate HDL dialog box.
- 2 Within the **Test Bench** pane, select the **Configuration** subpane.



- 3 To set an initial test bench input value of 0, select the **Initialize test bench inputs** option.

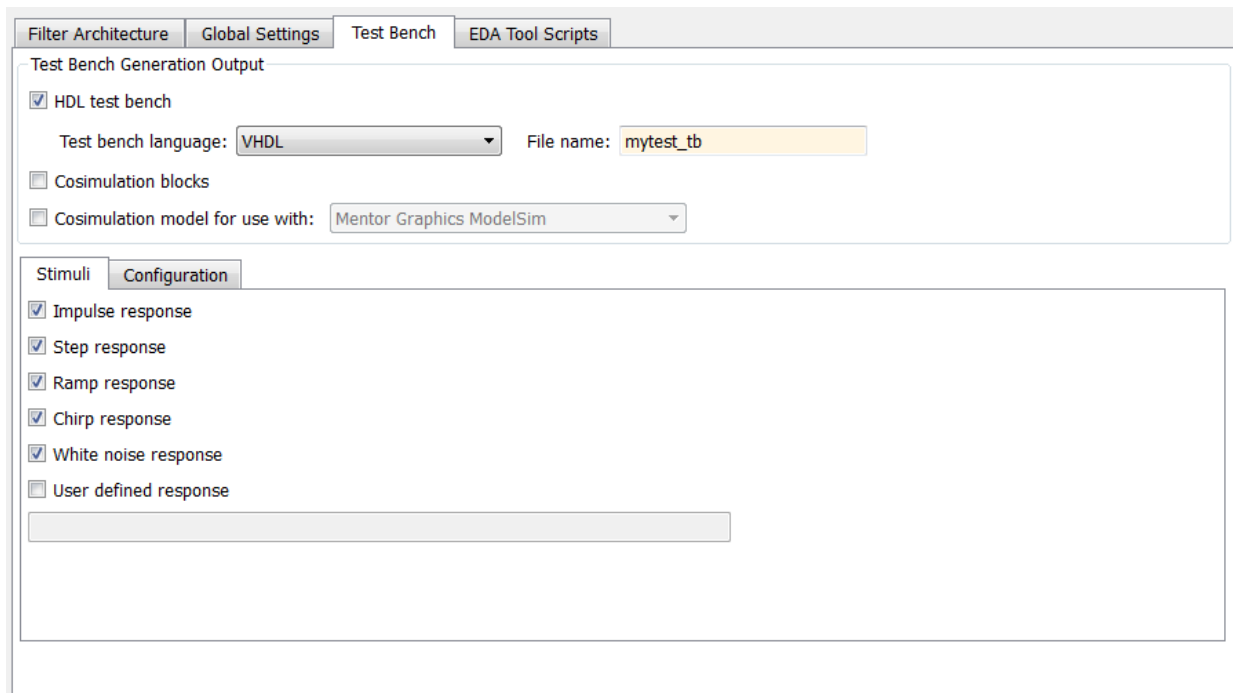
To set an initial test bench input value of 'X', clear the **Initialize test bench inputs** option.

Command Line Alternative: Use the `generatehdl` property `InitializeTestBenchInputs` to set the initial test bench input value.

Setting Test Bench Stimuli

By default, the coder generates a filter test bench that includes stimuli that correspond to the given filter type. However, you can adjust the stimuli settings or specify user defined stimuli, if desired.

To modify the stimuli that the coder is to include in a test bench, select one or more response types listed in the **Stimuli** subpane of the **Test bench settings** pane of the Generate HDL dialog box. The following figure highlights this pane of the dialog box.



If you select **User defined response**, you must also specify an expression or function that returns a vector of values to be applied to the filter. The values specified in the vector are quantized and scaled based on the filter's quantization settings.

Command Line Alternative: Use the `generatehdl` properties `TestBenchStimulus` and `TestBenchUserStimulus` to adjust stimuli settings.

Setting a Postfix for Reference Signal Names

Reference signal data is represented as arrays in the generated test bench code. The string specified by **Test bench reference postfix** is appended to the generated signal names. The default string is `_ref`.

You can set the postfix string to a value other than `_ref`. To change the string:

- 1 Select the **Test Bench** pane in the Generate HDL dialog box.
- 2 Within the **Test Bench** pane, select the **Configuration** subpane.
- 3 Enter a new string in the **Test bench reference postfix** field, as shown in the following figure.

The screenshot shows the 'Test Bench' tab of a configuration dialog. The 'Test Bench Generation Output' section is active, showing 'HDL test bench' selected, 'Test bench language' set to 'VHDL', and 'File name' set to 'mytest_tb'. Below this, the 'Configuration' sub-tab is selected. In the 'Configuration' section, the 'Test bench reference postfix' field is highlighted and contains the text 'Ref'. Other fields include 'Test bench data file name postfix' set to '_data', and various timing parameters like 'Clock high time (ns)' set to 5, 'Clock low time (ns)' set to 5, 'Hold time (ns)' set to 2, 'Setup time (ns)' set to 8, 'Clock enable delay (in clock cycles)' set to 1, and 'Reset length (in clock cycles)' set to 2. Checkboxes for 'Force clock', 'Force clock enable', 'Force reset', 'Hold input data between samples', 'Initialize test bench inputs', and 'Multi-file test bench' are also visible.

Command Line Alternative: Use the function with the property `TestBenchReferencePostFix` to change the postfix string.

Cosimulation of HDL Code with HDL Simulators

In this section...

“Generating HDL Cosimulation Blocks for Use with HDL Simulators” on page 6-27

“Generating a Simulink Model for Cosimulation with an HDL Simulator” on page 6-29

Generating HDL Cosimulation Blocks for Use with HDL Simulators

The coder supports generation of Simulink[®] HDL Cosimulation block(s). You can use the generated HDL Cosimulation blocks to cosimulate your filter design using Simulink and HDL Verifier[™], in conjunction with an HDL simulator. To use this feature, your installation must include one or more of the following:

- HDL Verifier for use with Mentor Graphics ModelSim
- HDL Verifier for use with Cadence Incisive[®]

The generated HDL Cosimulation blocks are configured to conform to the port and data type interface of the filter selected for code generation. By connecting an HDL Cosimulation block to a Simulink model in place of the filter, you can cosimulate your design with the desired HDL simulator.

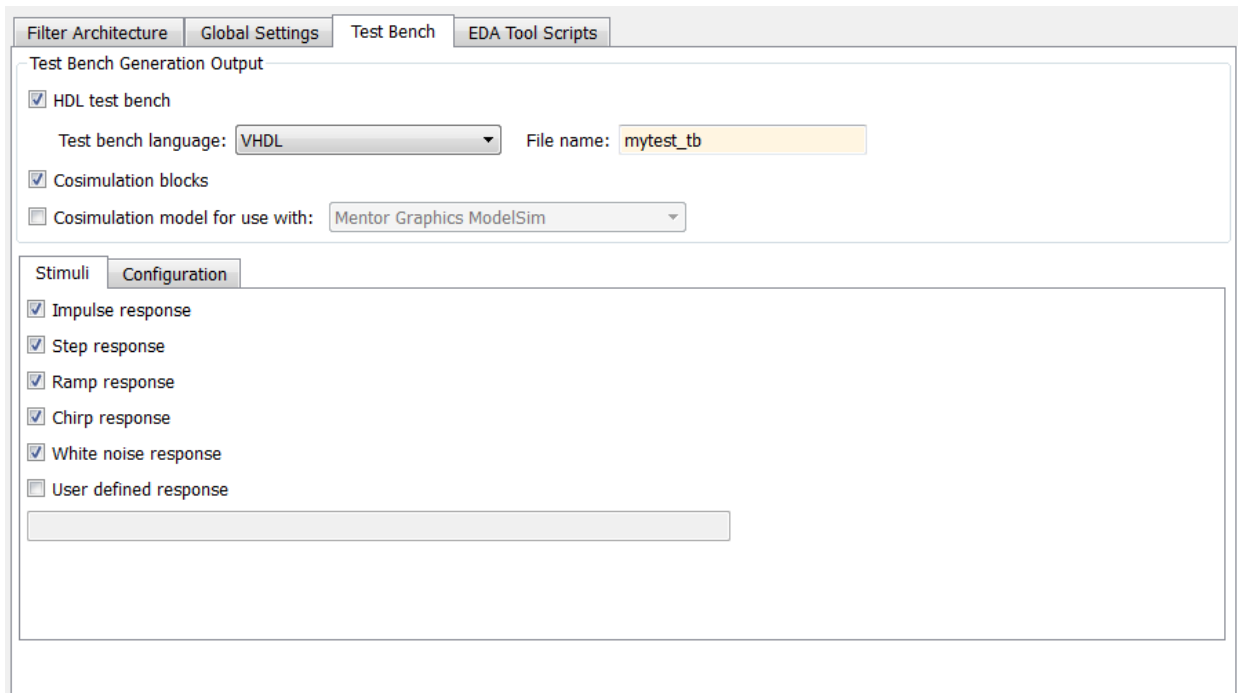
To generate HDL Cosimulation blocks:

- 1 Select the **Test Bench** pane in the Generate HDL dialog box.
- 2 Select the **Cosimulation blocks** option.

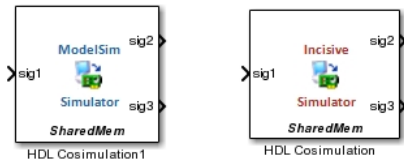
When this option is selected, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for each supported HDL simulator.

- 3 If you want to generate HDL Cosimulation blocks only (without generating HDL code), deselect **HDL test bench**.

The following figure shows both **HDL test bench** and **Cosimulation blocks** selected.



- 4 In the Generate HDL dialog box, click **Generate** to generate HDL and test bench code.
- 5 In addition to the usual code files, the coder generates a Simulink model containing one or more HDL Cosimulation blocks. The coder generates an HDL Cosimulation block for each HDL simulator supported by HDL Verifier, as shown in the following figure.



- 6 The generated model is untitled and exists in memory only. Be sure to save it to a destination folder if you want to preserve the model and blocks for use in future sessions.

See “Define HDL Cosimulation Block Interface ” for information on timing, latency, data typing, frame-based processing, and other issues that may be of concern to you when setting up an HDL cosimulation.

Command Line Alternative: Use the `generatehdl` function with the property `GenerateCoSimBlock` to generate HDL Cosimulation blocks.

Generating a Simulink Model for Cosimulation with an HDL Simulator

Note: To use this feature, your installation must include for one or both of the following:

- HDL Verifier for use with Mentor Graphics ModelSim
 - HDL Verifier for use with Cadence Incisive
-

The coder supports generation of a Simulink model, configured for both Simulink simulation of your filter design, and cosimulation of your design with an HDL simulator.

The generated model includes:

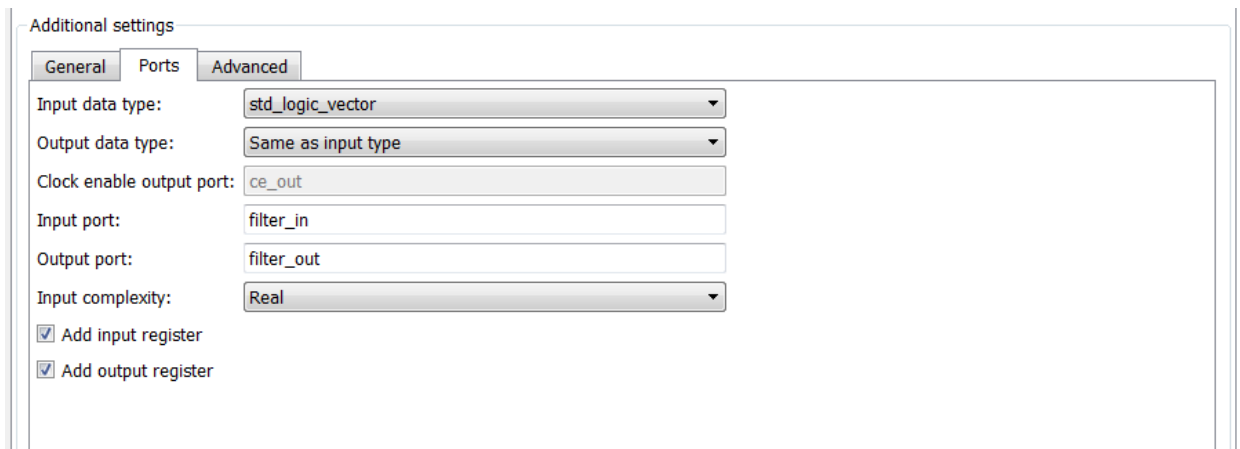
- A behavioral model of the filter design, realized in a Simulink subsystem. The subsystem includes a Digital Filter block, generated by the `realizemodel` function of the DSP System Toolbox.
- A corresponding HDL Cosimulation block. The coder configures this block to cosimulate the filter design using Simulink with either of the following:
 - HDL Verifier for use with Mentor Graphics ModelSim
 - HDL Verifier for use with Cadence Incisive
- Test input data, calculated from the test bench stimulus you specify. The coder stores the test data in the model workspace variable `inputdata`. A From Workspace block routes test data to the filter subsystem and HDL Cosimulation blocks.
- A Scope block that lets you observe and compare the test input signal, the outputs of the Filter block and the HDL cosimulation, and the difference (error) between these two outputs.

Generating the Model

Generation of a cosimulation model requires registered inputs and/or outputs (see “Limitations” on page 6-35). Before generating the model, make sure your model meets this requirement, as follows:

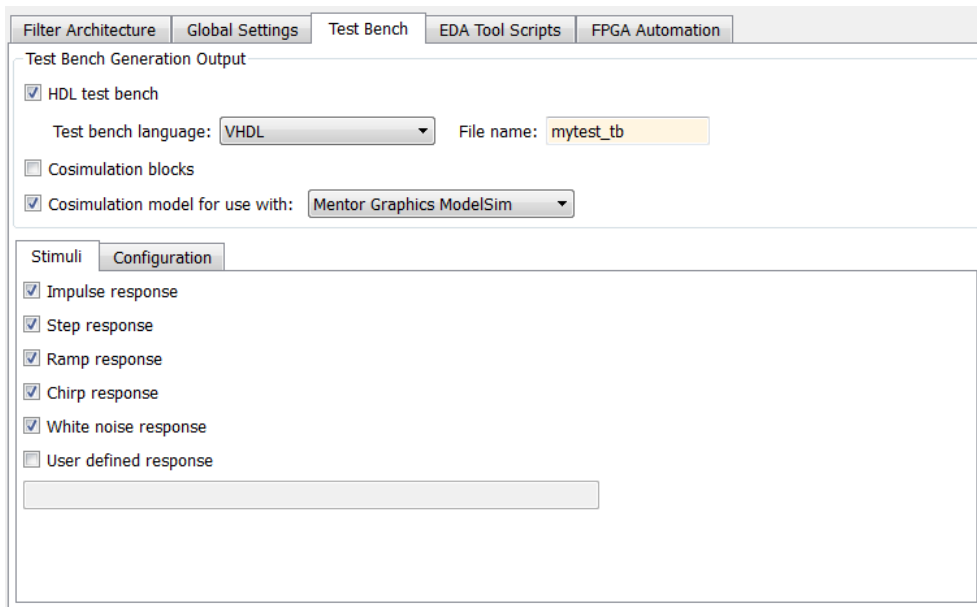
- 1 Select the **Global Settings** pane the Generate HDL dialog box.
- 2 In the **Global Settings** pane, click on the **Ports** tab. Port options appear.
- 3 Select one or both of the following options:
 - **Add input register**
 - **Add output register**

The coder selects both options, as shown in the following figure.



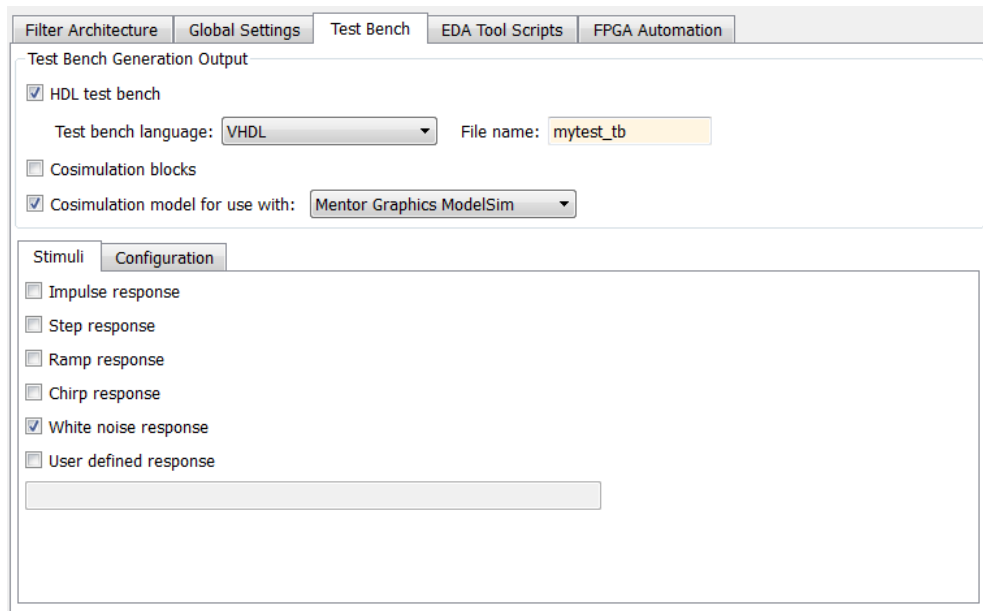
To generate the model:

- 1 In the Generate HDL dialog box, configure other code generation and test bench parameters as required by your design.
- 2 Select the **Test bench** pane of the Generate HDL dialog box.
- 3 Select the **Cosimulation model for use with:** option. Selecting this option enables the adjacent drop-down menu, where you can select Mentor Graphics ModelSim or Cadence Incisive.



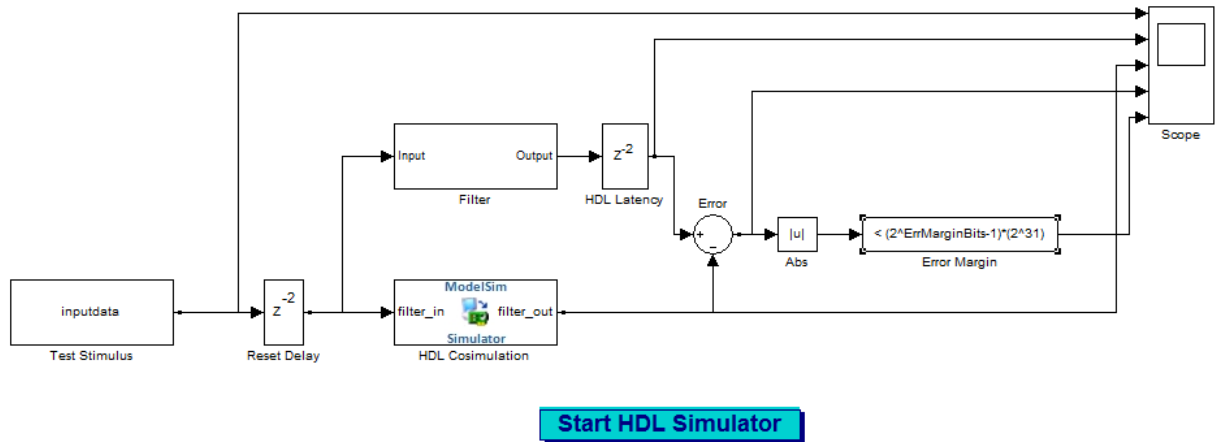
- 4 Using the drop-down menu, select which type of HDL Cosimulation block you want in the generated model. Select either **Mentor Graphics ModelSim** (the default) or **Cadence Incisive**.

In the following figure, the cosimulation model type is **Mentor Graphics ModelSim**, and the stimulus signal is **White noise response** .



- 5 In the Generate HDL dialog box, click **Generate** to generate HDL and test bench code.

In addition to the usual code files, the coder generates and opens a Simulink model. The following figure shows the model generated from the coder configuration shown in the previous step.



- 6 The generated model is untitled and exists in memory only. Be sure to save it to a destination folder if you want to preserve the model and blocks for use in future sessions.

See “Define HDL Cosimulation Block Interface ” for information on timing, latency, data typing, frame-based processing, and other issues that may be of concern to you when setting up an HDL cosimulation.

Details of the Generated Model

The generated model contains the following blocks:

- **Test Stimulus:** This From Workspace block routes test data in the model workspace variable `inputdata` to the filter subsystem and HDL Cosimulation blocks.
- **Filter:** This subsystem realizes a behavioral model of the filter design.
- **HDL Cosimulation:** This block cosimulates the generated HDL code. The table HDL Cosimulation Block Settings describes how the block parameters are configured by HDL Coder™.
- **Reset Delay:** The Tcl commands specified in the HDL Cosimulation block apply the reset signal. Reset is high at 0 ns and low at 22 ns (before the third rising clock edge). The Simulink simulation starts feeding the input at 0, 10, 20 ns. The Reset Delay block adds a delay such that the first sample is available to the RTL simulation when it is ready after the reset is applied.

- **HDL Latency:** This represents the difference between the latency of the RTL simulation and the Simulink behavioral block.
- **Error:** Computes the difference between the outputs of the **Filter** block and the **HDL Cosimulation** block .
- **Abs:** Absolute value of the error computation.
- **Error margin::** Indicator comparing the Absolute value of the error with the test bench error margin value (see “Setting an Error Margin for Optimized Filter Code” on page 6-22).
- **Scope:** Displays the input signal, outputs from the **Filter** block and the **HDL Cosimulation** blocks, and the difference (if one exists) between the two.
- **Start HDL Simulator** button: Launches your HDL cosimulation software.

HDL Cosimulation Block Settings

Pane	Settings
Ports	Port names: same as those in the generated code for the filter. Input/Output data types: Inherit Input sample time: Inherit Output sample time: Same as Simulink fixed step size.
Clocks	Clock port name: same as that in the generated code for the filter. Active clock edge: Rising Period: same as the Simulink sample time.
Timescales	1 second in Simulink corresponds to 1 tick in the HDL simulator
Connection	Connection Mode: Full Simulation Connection Method: Shared memory
Tcl (Pre-simulation commands)	<pre>force /Hlp/clock_enable 1; force /Hlp/reset 1 0 ns, 0 22 ns; puts ----- puts "Running Simulink Cosimulation block."; puts [clock format [clock seconds]]</pre>

Pane	Settings
Tcl (Post-simulation commands)	force /Hlp/reset 1 puts [clock format [clock seconds]]

Generated Model Settings

The generated model has the following nondefault settings:

- **Solver:** Discrete (no continuous states).
- **Solver Type:** Fixed-step.
- **Stop Time:** $T_s * StimLen$, where T_s is the Simulink sample time and $StimLen$ is the stimulus length.
- **Sample Time Colors:** enabled
- **Port Data Types:** enabled
- **Hardware Implementation:** ASIC/FPGA

Limitations

- A cosimulation that runs without encountering errors requires that outputs from the generated HDL code are synchronous with the clock. Before generating code, make sure that one or both of the following options are selected:
 - **Add input register**
 - **Add output register**

If you do not select either of these options, the coder terminates model generation with an error. The process of code generation will complete, however.

- The coder does not support generation of a cosimulation model when the target language is Verilog and data of type double is generated.

Command Line Alternative

Use the `generatehdl` function, passing in one of the following values for the property `GenerateCosimModel`:

- `generatehdl(filterObj, 'GenerateCosimModel', 'Incisive');`
- `generatehdl(filterObj, 'GenerateCosimModel', 'ModelSim');`

Integration With Third-Party EDA Tools

In this section...

“Generating a Default Script” on page 6-36

“Customizing Script Generation Using CLI Properties” on page 6-37

“Customizing Script Generation with the EDA Tool Scripts Dialog Box” on page 6-40

Generating a Default Script

By default, script generation takes place automatically, as part of the code and test bench generation process. Script files are generated in the target folder.

When HDL code is generated for a filter *Hd*, the coder writes the following script files:

- *Hd_compile.do*: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated filter code, but not to simulate it.
- *Hd_synplify.tcl*: Synplify[®] synthesis script

When test bench code is generated for a filter *Hd*, the coder writes the following script files:

- *Hd_tb_compile.do*: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
- *Hd_tb_sim.do*: Mentor Graphics ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.

You can enable or disable script generation and customize the names and content of generated script files using either of the following methods:

- Use the `generatehdl` function, and pass in the specified property name/property value arguments, as described in “Customizing Script Generation Using CLI Properties” on page 6-37.
- Set script generation options in the EDA Tool Scripts dialog box, as described in “Customizing Script Generation with the EDA Tool Scripts Dialog Box” on page 6-40.

Structure of Generated Script Files

A generated EDA script consists of three sections, which are generated and executed in the following order:

- 1 An initialization (`Init`) phase. The `Init` phase performs required setup actions, such as creating a design library or a project file. Some arguments to the `Init` phase are implicit, for example, the top-level entity or module name.
- 2 A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file or once per signal. On each call, a different file or signal name is passed in.
- 3 A termination phase (`Term`). This is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase does not take arguments.

The coder generates scripts by passing format strings to the `fprintf` function. Using the GUI options (or `generatehdl` properties) summarized in the following sections, you can pass in customized format strings to the script generator. Some of these format strings take arguments, such as the top-level entity or module name, or the names of the VHDL or Verilog files in the design.

You can use legal `fprintf` formatting characters. For example, `'\n'` inserts a newline into the script file.

Customizing Script Generation Using CLI Properties

This section describes how to set properties in the `generatehdl` function to enable or disable script generation and customize the names and content of generated script files.

Enabling and Disabling Script Generation

The `EDAScriptGeneration` property controls the generation of script files. By default, `EDAScriptGeneration` is set `'on'`. To disable script generation, set `EDAScriptGeneration` to `'off'`, as in the following example.

```
generatehdl(Hd, 'EDAScriptGeneration', 'off')
```

Customizing Script Names

When HDL code is generated, the code generator forms script names by appending a postfix string to the filter name `Hd`.

When test bench code is generated, the code generator forms script names by appending a postfix string to the test bench name `testbench_tb`.

The postfix string depends on the type of script (compilation, simulation, or synthesis) being generated. The default postfix strings are shown in the following table. For each type of script, you can define your own postfix using the associated property.

Script Type	Property	Default Value
Compilation	'HDLCompileFilePostfix'	'_compile.do'
Simulation	'HDLSimFilePostfix'	'_sim.do'
Synthesis	'HDLSynthFilePostfix'	See “Automation Scripts for Third-Party Synthesis Tools” on page 7-2

The following command generates VHDL code for the filter object `myfilt`, specifying a custom postfix string for the compilation script. The name of the generated compilation script will be `myfilt_test_compilation.do`.

```
generatehdl(myfilt, 'HDLCompileFilePostfix', '_test_compilation.do')
```

Customizing Script Code

Using the property name/property value pairs summarized in the following table, you can pass in customized format strings to `generatehdl`. The properties are named according to the following conventions:

- Properties that apply to the initialization (`Init`) phase are identified by the substring `Init` in the property name.
- Properties that apply to the command-per-file phase (`Cmd`) are identified by the substring `Cmd` in the property name.
- Properties that apply to the termination (`Term`) phase are identified by the substring `Term` in the property name.

Property Name and Default	Description
Name: 'HDLCompileInit' Default: 'vlib work\n'	Format string passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script.
Name: 'HDLCompileVHDLCmd' Default: 'vcom %s %s\n'	Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for VHDL files. The two arguments are the contents of the 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).

Property Name and Default	Description
Name: 'HDLCompileVerilogCmd' Default: 'vlog %s %s\n'	Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for Verilog files. The two arguments are the contents of the 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: 'HDLCompileTerm' Default: ''	Format string passed to <code>fprintf</code> to write the termination portion of the compilation script.
Name: 'HDLSimInit' Default: ['onbreak resume\n',... 'onerror resume\n']	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.
Name: 'HDLSimCmd' Default: 'vsim -novopt %s.%s\n'	Format string passed to <code>fprintf</code> to write the simulation command. The implicit arguments are replaced with your library name and top-level module or entity name. If you are using VHDL, you can set the library name in <code>VHDLLibraryName</code> . If you are using Verilog it is set to <code>work</code> .
Name: 'HDLSimViewWaveCmd' Default: 'add wave sim:%s\n'	Format string passed to <code>fprintf</code> to write the simulation script waveform viewing command. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.
Name: 'HDLSimTerm' Default: 'run -all\n'	Format string passed to <code>fprintf</code> to write the <code>Term</code> portion of the simulation script.
Name: 'HDLSynthInit' Name: 'HDLSynthCmd' Name: 'HDLSynthTerm'	These format strings apply to generation of synthesis scripts. See “Automation Scripts for Third-Party Synthesis Tools” on page 7-2.

generatehdl Example

The following example specifies a Mentor Graphics ModelSim command for the `Init` phase of a compilation script for VHDL code generated from the filter `myfilt`.

```
generatehdl(myfilt, 'HDLCompileInit', 'vlib mydesignlib\n')
```

The following code shows the resultant script, `myfilt_compile.do`.

```
vlib mydesignlib  
vcom myfilt.vhd
```

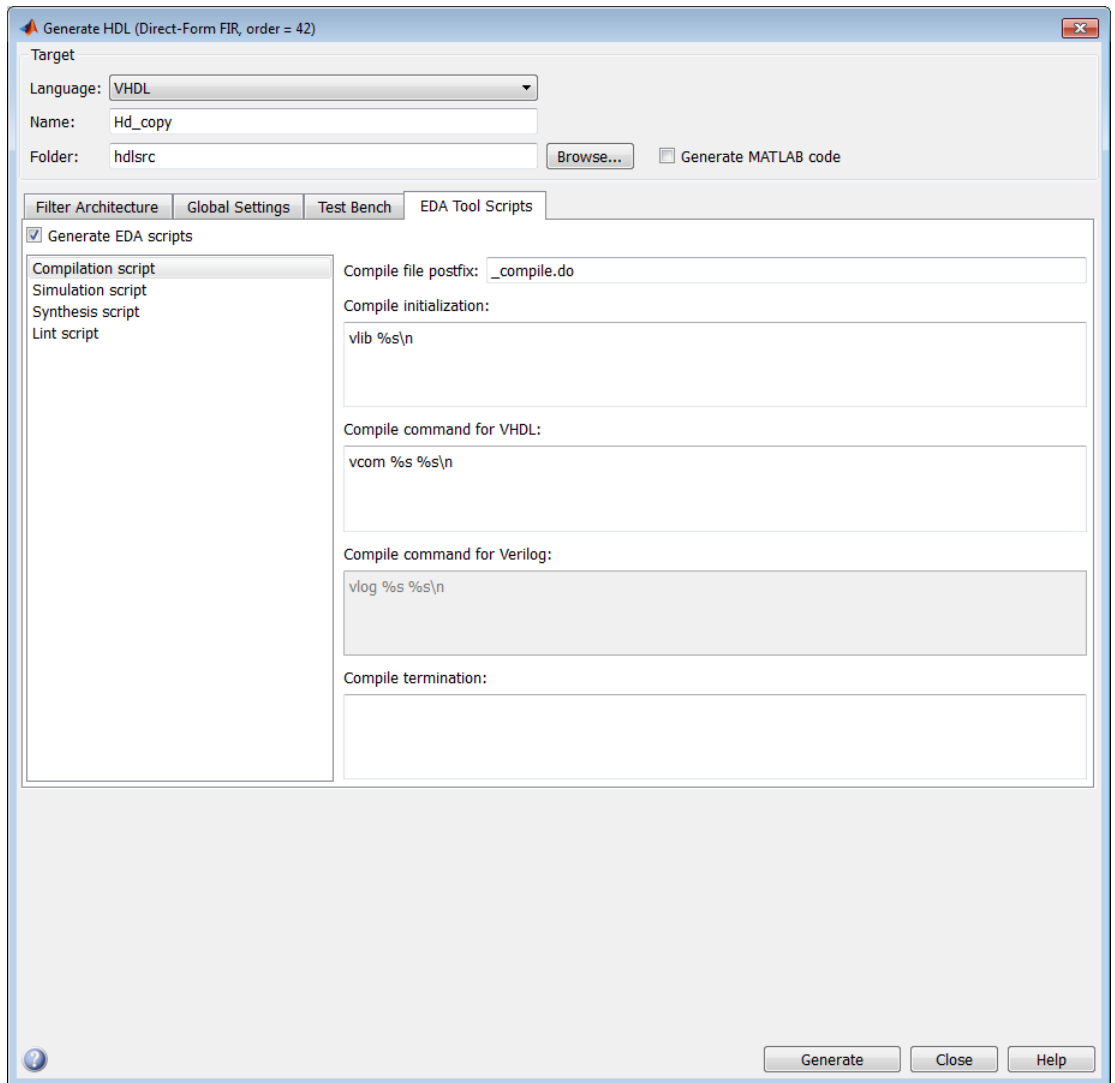
Customizing Script Generation with the EDA Tool Scripts Dialog Box

The EDA Tool Scripts dialog box, a subdialog of the Generate HDL dialog box, lets you set the options that control generation of script files. These options correspond to the properties described in “Customizing Script Generation Using CLI Properties” on page 6-37.

To view and set options in the EDA Tool Scripts dialog box:

- 1 Open the Generate HDL dialog box.
- 2 Click the **EDA Tool Scripts** tab in the Generate HDL dialog box.

The EDA Tool scripts dialog box displays, with the **Compilation script** options group selected, as shown in the following figure.



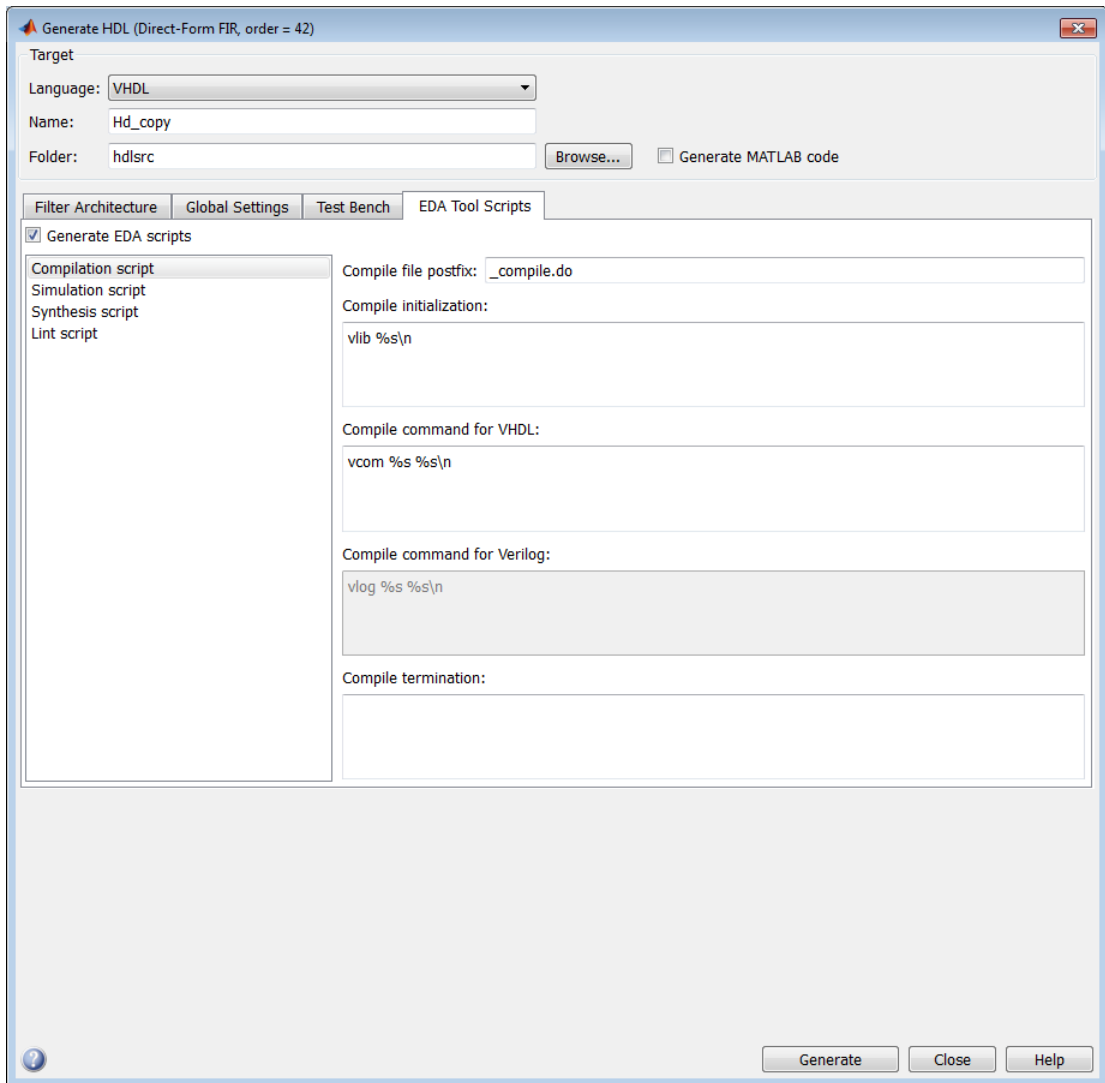
- 3 The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected, as shown in the preceding image.

If you want to disable script generation, clear this option.

- 4 The list on the left of the dialog box lets you select from several categories of options. Select a category and set the options as desired. The categories are
- **Compilation script:** options related to customizing scripts for compilation of generated VHDL or Verilog code. See “Compilation Script Options” on page 6-42 for further information.
 - **Simulation script:** options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 6-45 for further information.
 - **Synthesis script:** options related to customizing scripts for synthesis tools. For information about synthesis script options, see “Automation Scripts for Third-Party Synthesis Tools” on page 7-2 .

Compilation Script Options

The following figure shows the **Compilation script** pane, with the options set to their default values.



The following table summarizes the **Compilation script** options.

Option and Default	Description
Compile file postfix	Postfix string appended to the filter name or test bench name to form the script file name.

Option and Default	Description
'_compile.do'	
Name: Compile initialization Default: 'vlib work\n'	Format string passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script.
Name: Compile command for VHDL Default: 'vcom %s %s\n'	Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for VHDL files. The two arguments are the contents of the Simulator flags option and the file name of the current entity or module. To omit the flags, set Simulator flags to '' (the default). See also “Setting Simulator Flags for Compilation Scripts” on page 6-44.
Name: Compile command for Verilog Default: 'vlog %s %s\n'	Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for Verilog files. The two arguments are the contents of the Simulator flags option and the file name of the current entity or module. To omit the flags, set Simulator flags to '' (the default). See also “Setting Simulator Flags for Compilation Scripts” on page 6-44.
Name: Compile termination Default: ''	Format string passed to <code>fprintf</code> to write the termination portion of the compilation script.

Setting Simulator Flags for Compilation Scripts

You have the option of inserting simulator flags into your generated compilation scripts. For example, you may want to specify a specific compiler version. To specify the flags:

- 1 Click **Test Bench** in the Generate HDL dialog box.
- 2 Type the flags of interest in the **Simulator flags** field. In the following figure, the dialog box specifies that the Mentor Graphics ModelSim simulator use the `-93` compiler option for compilation.

Filter Architecture Global Settings **Test Bench** EDA Tool Scripts

Test Bench Generation Output

HDL test bench

Test bench language: File name:

Cosimulation blocks

Cosimulation model for use with:

Stimuli Configuration

Force clock

Clock high time (ns):

Clock low time (ns):

Hold time (ns):

Setup time (ns):

Force clock enable

Clock enable delay (in clock cycles):

Force reset

Reset length (in clock cycles):

Hold input data between samples

Initialize test bench inputs

Multi-file test bench

Test bench data file name postfix:

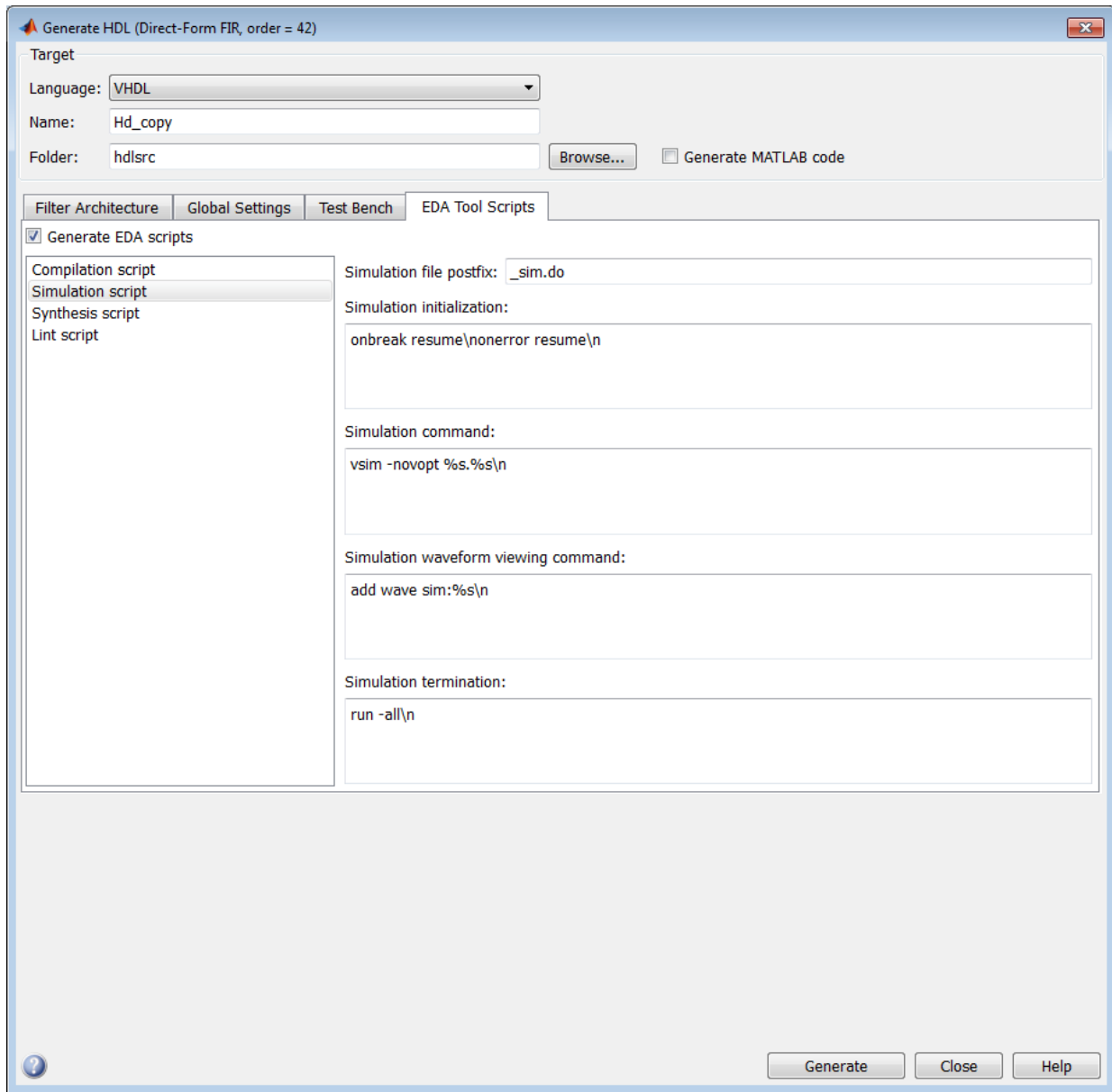
Test bench reference postfix:

Simulator flags:

Command Line Alternative: Use the `generatehdl` function's `SimulatorFlags` property to specify the type of test bench files to be generated.

Simulation Script Options

The following figure shows the **Simulation script** pane, with the options set to their default values.



The following table summarizes the **Simulation script** options.

Option and Default	Description
Simulation file postfix '_sim.do'	Postfix string appended to the filter name or test bench name to form the script file name.
Simulation initialization Default: ['onbreak resume\n',... 'onerror resume\n']	Format string passed to <code>fprintf</code> to write the initialization section of the simulation script.
Simulation command Default: 'vsim -novopt %s.%s\n'	Format string passed to <code>fprintf</code> to write the simulation command. The implicit arguments are replaced with your library name and top-level module or entity name. If you are using VHDL, you can set the library name with <code>VHDLLibraryName</code> . If you are using Verilog it is set to "work".
Simulation waveform viewing command Default: 'add wave sim:%s\n'	Format string passed to <code>fprintf</code> to write the simulation script waveform viewing command. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.
Simulation termination Default: 'run -all\n'	Format string passed to <code>fprintf</code> to write the Term portion of the simulation script.

Synthesis Script Options

For information about synthesis script options, see "Automation Scripts for Third-Party Synthesis Tools" on page 7-2.

Synthesis and Workflow Automation

Automation Scripts for Third-Party Synthesis Tools

In this section...

“Selecting a Synthesis Tool” on page 7-2

“Customizing Synthesis Script Generation Using CLI Properties” on page 7-3

“Customizing Synthesis Script Generation with the EDA Tool Scripts Dialog Box” on page 7-4

Selecting a Synthesis Tool

With Filter Design HDL Coder, you can enable or disable generation of synthesis scripts, and select the synthesis tool for which the coder generates scripts. To do so, click the **Choose synthesis tool** drop-down menu and select one of the following options:

None

This option is the default value. When you select **None**, the coder does not generate a synthesis script. The coder clears and disables the fields in the **Synthesis script** pane.

Altera Quartus II

Generate a synthesis script for Altera® Quartus II. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_quartus.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with Tcl script code for the tool.

Mentor Graphics Precision

Generate a synthesis script for Mentor Graphics Precision. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_precision.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with Tcl script code for the tool.

Synopsys Synplify Pro

Generate a synthesis script for Synopsys® Synplify Pro®. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_synplify.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with Tcl script code for the tool.

Xilinx ISE

Generate a synthesis script for Xilinx® ISE. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_ise.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with Tcl script code for the tool.

Customizing Synthesis Script Generation Using CLI Properties

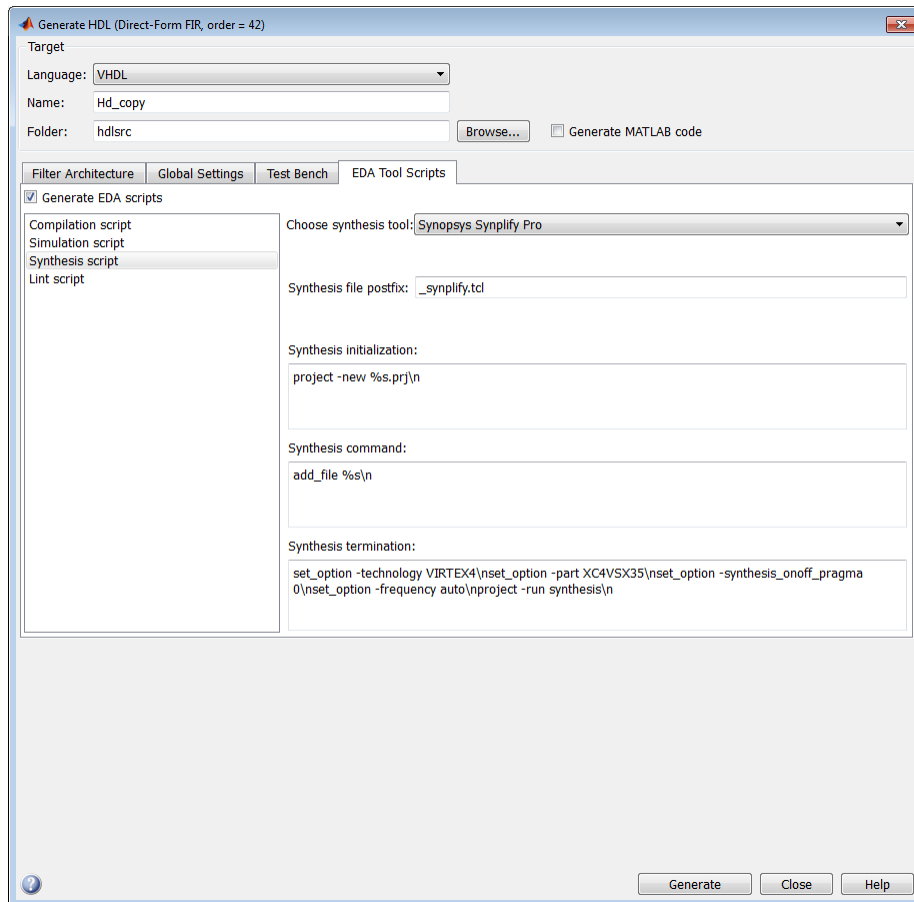
Using the property name/property value pairs summarized in the following table, you can pass in customized format strings to `generatehdl`. These format strings are passed on to `fprintf` to write each section of the synthesis script.

Property Name and Default	Description
Name: 'HDLSynthInit'	Format string passed to <code>fprintf</code> to write the Init section of the synthesis script. The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code> .
Name: 'HDLSynthCmd'	Format string passed to <code>fprintf</code> to write the Cmd section of the synthesis script. The argument uses the file name of the entity or module. The content of the string is specific to the selected synthesis tool. See <code>HDLSynthTool</code> .
Name: 'HDLSynthTerm'	Format string passed to <code>fprintf</code> to write the Term section of the synthesis script.

Property Name and Default	Description
	The content of the string is specific to the selected synthesis tool. See HDLSynthTool.

Customizing Synthesis Script Generation with the EDA Tool Scripts Dialog Box

The following figure shows the **Synthesis script** pane, with the options set to their default values.



The following table summarizes the **Synthesis script** options.

Option Name and Default	Description
Choose synthesis tool	<ul style="list-style-type: none"> • None (default): Do not generate a synthesis script. • ISE: Generate a synthesis script for Xilinx ISE software. • Precision: Generate a synthesis script for Mentor Graphics Precision software. • Quartus: Generate a synthesis script for Altera Quartus II software. • Synplify: Generate a synthesis script for Synopsys Synplify Pro software.
Synthesis file postfix	<p>Your choice of synthesis tool sets the postfix for generated synthesis file names to one of the following:</p> <ul style="list-style-type: none"> • <code>_ise.tcl</code> • <code>_precision.tcl</code> • <code>_quartus.tcl</code> • <code>_synplify.tcl</code>
Synthesis initialization	<p>Format string passed to <code>fprintf</code> to write the <code>Init</code> section of the synthesis script. The default string contains a synthesis project creation command. The implicit argument uses the top-level module or entity name.</p> <p>The content of the string is specific to the selected synthesis tool.</p>
Synthesis command	<p>Format string passed to <code>fprintf</code> to write the <code>Cmd</code> section of the synthesis script. The argument uses the filename of the entity or module.</p> <p>The content of the string is specific to the selected synthesis tool.</p>
Synthesis termination	<p>Format string passed to <code>fprintf</code> to write the <code>Term</code> section of the synthesis script.</p>

Option Name and Default	Description
	The content of the string is specific to the selected synthesis tool.

Properties — Alphabetical List

AddInputRegister

Generate extra register in HDL code for filter input

Settings

'on' (default)

Add an extra input register to the filter's generated HDL code.

The code declares a signal named `input_register` and includes a `PROCESS` block similar to the block below. Names and meanings of the timing parameters (clock, clock enable, and reset) and the coding style that checks for clock events may vary depending on other property settings.

```
Input_Register_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    input_register <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      input_register <= input_typeconvert;
    END IF;
  END IF;
END PROCESS Input_Register_Process ;
```

'off'

Omit the extra input register from the filter's generated HDL code.

Consider omitting the extra register if you are incorporating the filter into HDL code that already has a source for driving the filter. You might also consider omitting the extra register if the latency it introduces to the filter is not tolerable.

See Also

AddOutputRegister

AddOutputRegister

Generate extra register in HDL code for filter output

Settings

'on' (default)

Add an extra output register to the filter's generated HDL code.

The code declares a signal named `output_register` and includes a `PROCESS` block similar to the block below. Names and meanings of the timing parameters (clock, clock enable, and reset) and the coding style that checks for clock events may vary depending on other property settings.

```
Output_Register_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    output_register <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      output_register <= output_typeconvert;
    END IF;
  END IF;
END PROCESS Output_Register_Process ;
```

'off'

Omit the extra output register from the filter's generated HDL code.

Consider omitting the extra register if you are incorporating the filter into HDL code that has its own input register. You might also consider omitting the extra register if the latency it introduces to the filter is not tolerable.

See Also

AddInputRegister

AddPipelineRegisters

Optimize clock rate used by filter code by adding pipeline registers

Settings

'on'

Add a pipeline register between stages of computation in a filter. For example, for a sixth-order IIR filter, the coder adds two pipeline registers, one between the first and second sections and one between the second and third sections. Although the registers add to the overall filter latency, they provide significant improvements to the clock rate.

For...	A Pipeline Register Is Added Between...
FIR Transposed filters	Coefficient multipliers and adders
FIR, Asymmetric FIR, and Symmetric FIR filters	Levels of a tree-based final adder
IIR filters	Sections

'off' (default)

Suppress the use of pipeline registers.

Usage Notes

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see “Optimizing Final Summation for FIR Filters” on page 4-42.

Note: The use of pipeline registers in FIR, antisymmetric FIR, and symmetric FIR filters can produce numeric results that differ from those produced by the original filter object because they force the tree mode of final summation. In such cases, consider adjusting the test bench error margin.

See Also

CoeffMultipliers, FIRAdderStyle, OptimizeForHDL

AddRatePort

Generate rate ports for variable-rate CIC filter

Settings

'off' (default)

Do not generate rate ports.

'on'

Generate `rate` and `load_rate` ports. When the `load_rate` signal is asserted, the `rate` port loads in a rate factor.

Usage Notes

AddRatePort is specifically for use with variable rate CIC filters. A variable rate CIC filter has a programmable rate change factor. It is assumed that the filter is designed with the maximum rate expected, and that the Decimation Factor (for CIC Decimators) or Interpolation Factor (for CIC Interpolators) is set to this maximum rate change factor.

See Also

TestbenchRateStimulus, “Variable Rate CIC Filters” on page 3-8

BlockGenerateLabel

Specify string to append to block labels used for HDL GENERATE statements

Settings

'string'

Specify a postfix string to append to block labels used for HDL GENERATE statements. The default string is `_gen`.

See Also

InstanceGenerateLabel, OutputGenerateLabel

CastBeforeSum

Enable or disable type casting of input values for addition and subtraction operations

Settings

'off' (default)

Preserve the types of input values during addition and subtraction operations and then convert the result to the result type.

'on'

Type cast input values in addition and subtraction operations to the result type before operating on the values. This setting produces numeric results that are typical of DSP processors.

The `CastBeforeSum` property is related to the `FDATool` setting for the quantization property **Cast signals before accum.** as follows:

- Some filter object types do not have the **Cast signals before accum.** property. For such filter objects, `CastBeforeSum` is effectively off when HDL code is generated; it is not relevant to the filter.
- Where the filter object does have the **Cast signals before accum.** property, the coder by default sets `CastBeforeSum` following the setting of **Cast signals before accum.** in the filter object. This is visible in the GUI. If you change the setting of **Cast signals before accum.**, the coder updates the setting of **Cast before sum.**
- However, by explicitly setting the `CastBeforeSum` property, you can override the **Cast signals before accum.** setting passed in from `FDATool`.

See Also

“Specifying Input Type Treatment for Addition and Subtraction Operations” on page 5-34, `InlineConfigurations`, `LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`, `UseVerilogTimescale`

ClockEnableInputPort

Name HDL port for filter's clock enable input signals

Settings

'string'

The default name for the filter's clock enable input port is `clk_enable`.

For example, if you specify the string `'filter_clock_enable'` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( clk
        filter_clock_enable : IN std_logic;
        reset                : IN std_logic;
        filter_in            : IN std_logic_vector (15 DOWNT0 0);
        filter_out          : OUT std_logic_vector (15 DOWNT0 0);
  );
END Hd;
```

If you specify a string that is a VHDL or Verilog reserved word, a reserved word postfix string is appended 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`. See `ReservedWordPostfix` for more information.

Usage Notes

The clock enable signal is asserted active high (1). Thus, the input value must be high for the filter entity's registers to be updated.

See Also

`ClockInputPort`, `InputPort`, `InputType`, `OutputPort`, `OutputType`, `ResetInputPort`

ClockEnableOutputPort

For multirate filters (with single clock), specify name of clock enable output port

Settings

'string'

The default name for the generated clock enable output port is `ce_out`.

Usage Notes

For multirate filters, a clock enable output is generated when **Single** is selected from the **Clock inputs** menu in the Generate HDL dialog box. In this case only, the **Clock enable output port** option is enabled.

See Also

ClockInputs

ClockHighTime

Specify period, in nanoseconds, during which test bench drives clock input signals high (1)

Settings

ns

The default is 5.

The clock high time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

Usage Notes

The coder ignores this property if `ForceClock` is set to `'off'`.

See Also

`ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockInputPort

Name HDL port for filter's clock input signals

Settings

'string'

The default clock input port name is `clk`.

For example, if you specify the string `'filter_clock'` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( filter_clock : IN  std_logic;
        clk_enable   : IN  std_logic;
        reset        : IN  std_logic;
        filter_in    : IN  std_logic_vector (15 DOWNTO 0); -- sfix16_En15
        filter_out   : OUT std_logic_vector (15 DOWNTO 0); -- sfix16_En15
        );
ENDHd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See for more information.

See Also

`ClockEnableInputPort`, `InputPort`, `InputType`, `OutputPort`, `OutputType`, `ResetInputPort`

ClockInputs

For multirate filters, specify generation of single or multiple clock inputs

Settings

'Single' (default)

Generate a single clock input for a multirate filter. When this option is selected, the ENTITY declaration for the filter defines a single clock input with an associated clock enable input and clock enable output. The generated code maintains a counter that controls the timing of data transfers to the filter output (for decimation filters) or input (for interpolation filters). The counter behaves as a secondary clock whose rate is determined by the filter's decimation or interpolation factor.

'Multiple'

Generate multiple clock inputs for a multirate filter. When this option is selected, the ENTITY declaration for the filter defines separate clock inputs (each with an associated clock enable input) for each rate of a multirate filter. (For currently supported multirate filters, there are two such rates.)

Usage Notes

The **Clock inputs** menu is enabled only when a multirate filter (of one of the types supported for code generation) has been designed in `fdatool`.

The generated code assumes that the clocks are driven at suitable rates. You are responsible for seeing that the clocks run at relative rates that correspond to the filter's decimation or interpolation factor. To see an example, generate test bench code for your multirate filter and examine the `clk_gen` processes for each clock.

See Also

ClockEnableOutputPort

ClockLowTime

Specify period, in nanoseconds, during which test bench drives clock input signals low (0)

Settings

ns

The default is 5.

The clock low time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

Usage Notes

The coder ignores this property if `ForceClock` is set to `'off'`.

See Also

`ClockHighTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockProcessPostfix

Specify string to append to HDL clock process names

Settings

'string'

The default postfix is `_process`.

The coder uses process blocks to modify the content of a filter's registers. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the coder derives the label `delay_pipeline_process` in the following block declaration from the register name `delay_pipeline` and the default postfix string `_process`:

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
    .
    .
    .
```

See Also

`PackagePostfix`, `ReservedWordPostfix`

CoefficientMemory

Specify type of memory for storage of programmable coefficients for serial FIR filters

Settings

'Registers' (default)

Store programmable coefficients in a register file.

'DualPortRAMs'

Store programmable coefficients in dual-port RAM.

'SinglePortRAMs'

Usage Notes

This property applies only to FIR filters that have one of the following serial architectures:

- Fully serial
- Partly serial
- Cascade serial

When you use this property, be sure to set `CoefficientSource` to `'ProcessorInterface'`. The coder ignores `CoefficientMemory` unless it is generating an interface for programmable coefficients.

See Also

“Programmable Filter Coefficients for FIR Filters” on page 3-27, , `CoefficientSource`, `TestbenchCoeffStimulus`

CoefficientSource

Specify source for FIR or IIR filter coefficients

Settings

'Internal' (default)

Coefficients are obtained from the filter object and hard coded.

ProcessorInterface

Generate a memory interface, which can be driven by an external microprocessor, for coefficients.

When you specify 'ProcessorInterface', the generated entity or module definition for the filter includes the following port definitions:

- `coeffs_in`: Input port for coefficient data
- `write_address`: Write address for coefficient memory
- `write_enable`: Write enable signal for coefficient memory
- `write_done`: Signal to indicate completion of coefficient write operation

See Also

“Programmable Filter Coefficients for FIR Filters” on page 3-27, “Programmable Filter Coefficients for IIR Filters” on page 3-39, `TestbenchCoeffStimulus`

CoeffMultipliers

Specify technique used for processing coefficient multiplier operations

Settings

'multiplier' (default)

Retain multiplier operations in the generated HDL code.

'csd'

This option uses canonical signed digit (CSD) techniques, which replace multiplier operations with shift and add operations. CSD techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This decreases the area used by the filter while maintaining or increasing clock speed.

'factored-csd'

This option uses factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the coefficients. This option lets you achieve a greater filter area reduction than CSD, at the cost of decreasing clock speed.

See Also

AddPipelineRegisters, FIRAdderStyle, OptimizeForHDL

CoeffPrefix

Specify prefix (string) for filter coefficient names

Settings

'string'

The default prefix for filter coefficient names is `coeff`.

For...	The Prefix Is Concatenated with...
FIR filters	Each coefficient number, starting with 1. For example, the default for the first coefficient would be <code>coeff1</code> .
IIR filters	An underscore (<code>_</code>) and an <code>a</code> or <code>b</code> coefficient name (for example, <code>_a2</code> , <code>_b1</code> , or <code>_b2</code>) followed by the string <code>_sectionn</code> , where <code>n</code> is the section number. For example, the default for the first numerator coefficient of the third section is <code>coeff_b1_section3</code> .

For example:

```
ARCHITECTURE rtl OF Hd IS
  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>)
    OF signed(15 DOWNTO 0); -- sfix16_En15
  CONSTANT coeff1 : signed(15 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
  CONSTANT coeff2 : signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
  CONSTANT coeff3 : signed(15 DOWNTO 0) := to_signed(-81, 16); -- sfix16_En15
  CONSTANT coeff4 : signed(15 DOWNTO 0) := to_signed(120, 16); -- sfix16_En15
  .
  .
  .
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See [ReservedWordPostfix](#) for more information.

See Also

[ClockProcessPostfix](#), [EntityConflictPostfix](#), [PackagePostfix](#)

ComplexImagPostfix

Specify string to append to imaginary part of complex signal names

Settings

'string'

Default: '_im'

See Also

ComplexRealPostfix, InputComplex

“Using Complex Data and Coefficients” on page 5-36

ComplexRealPostfix

Specify string to append to real part of complex signal names

Settings

'string'

Default: 're'

See Also

ComplexImagPostfix, InputComplex

“Using Complex Data and Coefficients” on page 5-36

DALUTPartition

Specify number and size of LUT partitions for distributed arithmetic architecture

Settings

[p1 p2...pN]

Where [p1 p2 p3...pN] is a vector of N integers, divides the LUT used in distributed arithmetic (DA) into N partitions. Each vector element specifies the size of a partition. The maximum size for an individual partition is 12. The sum of the vector elements must be equal to the filter length. The filter length is calculated differently depending on the filter type (see “Distributed Arithmetic for FIR Filters” on page 4-24).

Usage Notes

To enable generation of DA code for your filter design without LUT partitioning, specify a vector of one element, whose value is equal to the filter length, as in the following example:

```
fdes = fdesign.lowpass('N,Fc,Ap,Ast',4,0.4,0.05,0.03,'linear');  
Hd = design(fdse);  
Hd.arithmetic = 'fixed';  
generatehdl(Hd, 'DALUTPartition', 5)
```

See Also

DARadix

DARadix

Specify number of bits processed simultaneously in distributed arithmetic architecture

Settings

N

N specifies the number of bits processed simultaneously in a distributed arithmetic (DA) architecture. N must be

- A nonzero positive integer that is a power of two
- Such that $\text{mod}(W, \log_2(N)) = 0$, where W is the input word size of the filter.

The default value for N is 2, specifying processing of one bit at a time, or fully serial DA. The maximum value for N is 2^W , where W is the input word size of the filter. This maximum specifies fully parallel DA. Values of N between these extrema specify partly serial DA.

Usage Notes

The `DARadix` property lets you introduce a degree of parallelism into the operation of DA, possibly resulting in performance improvement at the expense of area. See “Distributed Arithmetic for FIR Filters” on page 4-24 for a complete description of DA.

See Also

DALUTPartition

EDAScriptGeneration

Enable or disable generation of script files for third-party tools

Settings

'on' (default)

Enable generation of script files.

'off'

Disable generation of script files.

See Also

“Integration With Third-Party EDA Tools” on page 6-36

EntityConflictPostfix

Specify string to append to duplicate VHDL entity or Verilog module names

Settings

'string'

The specified postfix resolves duplicate VHDL entity or Verilog module names. The default string is `_block`.

For example, if the coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_block`.

See Also

`ClockProcessPostfix`, `CoeffPrefix`, `PackagePostfix`, `ReservedWordPostfix`

ErrorMargin

Specify error margin for HDL language-based test benches

Settings

n

Some HDL optimizations can generate test bench code that produces numeric results that differ from those produced by the original filter function. By specifying an error margin, you can specify an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.

Specify the error margin as an integer number of bits.

Usage Notes

Optimizations that can generate test bench code that produces numeric results that differ from those produced by the original filter function include

- `CastBeforeSum` (qfilters only)
- `OptimizeForHDL`
- `FIRAdderStyle ('Tree')`
- `AddPipelineRegisters` (for FIR, Asymmetric FIR, and Symmetric FIR filters)

The error margin is the number of least significant bits a Verilog or VHDL language-based test bench can ignore when comparing the numeric results before generating a warning.

For fixed-point filters, the **Error margin (bits)** value is initialized to a default value of 4.

See Also

`AddPipelineRegisters`, `CastBeforeSum`, `CoeffMultipliers`, `FIRAdderStyle`, `OptimizeForHDL`

FIRAdderStyle

Specify final summation technique used for FIR filters

Settings

'linear' (default)

Apply linear adder summation. This technique is discussed in most DSP text books.

'tree'

Increase clock speed while maintaining the area used. This option creates a final adder that performs pair-wise addition on successive products that execute in parallel, rather than sequentially.

Usage Notes

If you are generating HDL code for a FIR filter, consider optimizing the final summation technique by applying tree or pipeline final summation techniques. Pipeline mode produces results similar to tree mode with the addition of a stage of pipeline registers after processing each level of the tree.

For information on applying pipeline mode, see [AddPipelineRegisters](#).

Consider the following tradeoffs when selecting the final summation technique for your filter:

- The number of adder operations for linear and tree mode are the same, but the timing for tree mode might be significantly better due to summations occurring in parallel.
- Pipeline mode optimizes the clock rate, but increases the filter latency by the base 2 logarithm of the number of products to be added, rounded up to the nearest integer.
- Linear mode can help attain numeric accuracy in comparison to the original filter function. Tree and pipeline modes can produce numeric results that differ from those produced by the original filter function.

See Also

AddPipelineRegisters, CoeffMultipliers, OptimizeForHDL

FoldingFactor

Specify folding factor for IIR SOS filter with serial architecture

Settings

N

A value must be specified. N must be an integer greater than 1.

FoldingFactor defines N, the total number of clock cycles taken for the computation of filter output in an IIR SOS filter with serial architecture.

Usage Notes

FoldingFactor is available for **df1sos** and **df2sos** filters with serial architecture only. Both **FoldingFactor** and **NumMultipliers** generate HDL for an IIR SOS filter with serial architecture, but you must select one property or the other; you may not use both.

If neither **NumMultipliers** or **FoldingFactor** is specified, HDL code for the filter is generated with Fully Parallel architecture.

The following example demonstrates generating HDL for a **df2sos** filter with serial architecture and a folding factor of 9:

```
Hd = design(fdesign.lowpass, 'ellip', 'FilterStructure', 'df2sos');  
Hd.arithmetic = 'fixed';  
generatehdl(Hd, 'foldingfactor', 9)
```

For the legal values of **FoldingFactor**, use the helper function:

```
hdlfilterserialinfo(Hd)
```

Table of folding factors with corresponding number of multipliers for the given filter.

Folding Factor	Multipliers
6	3
9	2
18	1

See Also

NumMultipliers

ForceClock

Specify whether test bench forces clock input signals

Settings

'on' (default)

Specify that the test bench forces the clock input signals. When this option is set, the clock high and low time settings control the clock waveform.

'off'

Specify that a user-defined external source forces the clock input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ForceClockEnable

Specify whether test bench forces clock enable input signals

Settings

'on' (default)

Specify that the test bench forces the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value.

'off'

Specify that a user-defined external source forces the clock enable input signals.

See Also

ClockHighTime, ClockLowTime, ForceClock, ForceReset, HoldTime

ForceReset

Specify whether test bench forces reset input signals

Settings

'on' (default)

Specify that the test bench forces the reset input signals. If you enable this option, you can also specify a hold time to control the timing of a reset.

'off'

Specify that a user-defined external source forces the reset input signals.

See Also

ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, HoldTime

FracDelayPort

Name port for Farrow filter's fractional delay input signal

Settings

'string'

The default string is `filter_fd`.

For example, if you specify the string `FractionalDelay` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( clk           : IN  std_logic;
        clk_enable   : IN  std_logic;
        reset        : IN  std_logic;
        filter_in    : IN  std_logic_vector (15 DOWNTO 0);
        FractionalDelay : IN  std_logic_vector (5 DOWNTO 0);
        filter_out   : OUT std_logic_vector (15 DOWNTO 0);
        );
END Hd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See `ReservedWordPostfix` for more information.

See Also

“Single-Rate Farrow Filters” on page 3-20, `TestBenchFracDelayStimulus`

GenerateCoSimBlock

Generate model containing HDL Cosimulation block(s) for simulation of filter in Simulink

Settings

'off' (default)

Do not generate HDL Cosimulation blocks.

'on'

If your installation is licensed for one or more of the following HDL simulation products, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for each licensed product:

- HDL Verifier for use with Mentor Graphics ModelSim
- HDL Verifier for use with Cadence Incisive

The coder configures the generated HDL Cosimulation blocks to conform to the port and data type interface of the filter selected for code generation. By connecting an HDL Cosimulation block to a Simulink model in place of the filter, you can cosimulate your design with the desired HDL simulator.

The coder appends the string (if one exists) specified by the `CosimLibPostfix` property to the names of the generated HDL Cosimulation blocks.

GenerateCosimModel

Generate model containing realized filter and HDL Cosimulation block for simulation of filter in Simulink

Settings

'none' (default)

Do not generate a cosimulation model.

'ModelSim

If your installation includes HDL Verifier for use with Mentor Graphics ModelSim, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for that simulator.

'Incisive'

If your installation is licensed for HDL Verifier for use with Cadence Incisive, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for that simulator.

See Also

“Generating a Simulink Model for Cosimulation with an HDL Simulator” on page 6-29

GenerateHDLTestbench

Enable generation of a test bench

Settings

'on'

Generate test bench code.

'off' (default)

Do not generate test bench code

Description

To generate a test bench for your HDL filter code, use the `generatehdl` function and set the `GenerateHDLTestbench` property to `'on'`, as shown in the following example.

```
generatehdl(Hlp, 'GenerateHDLTestbench', 'on')
```

Note: You should replace calls to `generatetb` in your scripts with calls to `generatehdl`, enabling test bench generation with the `GenerateHDLTestbench` property, as shown in the preceding example.

In Release R2011a, the `generatetb` function will continue to operate, but will display a warning message when it is called.

See Also

“Enabling Test Bench Generation” on page 6-9, `generatetb`

HDLCompileFilePostfix

Specify postfix string appended to file name for generated Mentor Graphics ModelSim compilation scripts

Settings

The default postfix is `_compile.do`.

For example, if the name of the filter or test bench is `my_design`, the coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLCompileInit

Specify string written to initialization section of compilation script

Settings

'string'

The default string is 'vlib %s\n'.

At script generation time, the string you specify as `VHDLLibraryName` substitutes into the `HDLCompileInit` string value. By default, this generates the library specification 'vlib work/n'

You can use `VHDLLibraryName` to avoid library name conflicts.

See Also

`VHDLLibraryName`

“Integration With Third-Party EDA Tools” on page 6-36

HDLCompileTerm

Specify string written to termination section of compilation script

Settings

'string'

The default is the null string (' ').

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLCompileVerilogCmd

Specify command string written to compilation script for Verilog files

Settings

'string'

The default string is 'vlog %s %s\n'.

The two arguments are the contents of the 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLCompileVHDLCmd

Specify command string written to compilation script for VHDL files

Settings

'string'

The default string is 'vcom %s %s\n'.

The two arguments are the contents of the 'SimulatorFlags' property and the file name of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLsimCmd

Specify simulation command written to simulation script

Settings

'string'

The default string is 'vsim -novopt %s.%s\n'.

The implicit arguments are replaced with your library name and top-level module or entity name. If you are using VHDL, you can set the library name in VHDLLibraryName. If you are using Verilog it is set to work.

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLsimFilePostfix

Specify postfix string appended to file name for generated Mentor Graphics ModelSim simulation scripts

Settings

'string'

The default postfix is `_sim.do`.

For example, if the name of your test bench file is `my_design`, the coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

HDLSimInit

Specify string written to initialization section of simulation script

Settings

'string'

The default string is

```
['onbreak resume\n', ...  
'onerror resume\n']
```

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLsimTerm

Specify string written to termination section of simulation script

Settings

'string'

The default string is 'run -all\n'.

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLSimViewWaveCmd

Specify waveform viewing command written to simulation script

Settings

'string'

The default string is 'add wave sim:%s\n'.

The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.

See Also

“Integration With Third-Party EDA Tools” on page 6-36

HDLSynthCmd

Specify command written to synthesis script

Settings

'string'

The default string is 'add_file %s\n'.

The implicit argument is the file name of the entity or module.

See Also

“Automation Scripts for Third-Party Synthesis Tools” on page 7-2

HDLSynthFilePostfix

Specify postfix string appended to file name for generated Synplify synthesis scripts

Settings

'string'

Default: The value of `HDLSynthFilePostfix` normally defaults to a string that corresponds to the synthesis tool specified by `HDLSynthTool` (see `HDLSynthTool`).

For example, if the value of `HDLSynthTool` is 'Synplify', `HDLSynthFilePostfix` defaults to the string `'_synplify.tcl'`. Then, if the name of the device under test is `my_design`, the coder adds the postfix `_synplify.tcl` to form the synthesis script file name `my_design_synplify.tcl`.

See Also

“Automation Scripts for Third-Party Synthesis Tools” on page 7-2

HDLSynthInit

Specify string written to initialization section of synthesis script

Settings

'string'

The default string is 'project -new %s.prj\n', which is a synthesis project creation command.

The implicit argument is the top-level module or entity name.

See Also

“Automation Scripts for Third-Party Synthesis Tools” on page 7-2

HDLSynthTerm

Specify string written to termination section of synthesis script

Settings

'string'

The default string is

```
['set_option -technology VIRTEX4\n',...  
'set_option -part XC4VSX35\n',...  
'set_option -synthesis_onoff_pragma 0\n',...  
'set_option -frequency auto\n',...  
'project -run synthesis\n']
```

See Also

“Automation Scripts for Third-Party Synthesis Tools” on page 7-2

HDLSynthTool

Select synthesis tool for which the coder generates scripts.

Settings

'string'

Default: 'none'.

HDLSynthTool enables or disables generation of scripts for third-party synthesis tools. By default, the coder does not generate a synthesis script. To generate a script for one of the supported synthesis tools, set HDLSynthTool to one of the strings given in the following table.

Tip The value of HDLSynthTool also sets the postfix string (HDLSynthFilePostfix) that the coder appends to generated synthesis script file names.

Choice of HDLSynthTool Value...	Generates Script For...	Sets HDLSynthFilePostfix To...
none	N/A; script generation disabled	N/A
'ISE'	Xilinx ISE	'_ise.tcl'
'Precision'	Mentor Graphics Precision	'_precision.tcl'
'Quartus'	Altera Quartus II	'_quartus.tcl'
'Synplify'	Synopsys Synplify Pro	'_synplify.tcl'

See Also

HDLSynthFilePostfix, “Automation Scripts for Third-Party Synthesis Tools” on page 7-2

HoldInputDataBetweenSamples

Specify how long input data values are held in valid state

Settings

This property can be applied to filters that do not have fully parallel implementations. (See “Parallel and Serial Architectures” on page 4-3 and “Distributed Arithmetic for FIR Filters” on page 4-24.)

In such filter implementations, data can be delivered to the outputs N cycles ($N \geq 2$) later than the inputs. The `HoldInputDataBetweenSamples` property determines how long (in terms of clock cycles) input data values for these signals are held in a valid state, as follows:

- When `HoldInputDataBetweenSamples` is set to 'on' (the default), input data values are held in a valid state across N clock cycles.
- When `HoldInputDataBetweenSamples` is set to 'off', data values are held in a valid state for only one clock cycle. For the next $N-1$ cycles, data is in an unknown state (expressed as 'X') until the next input sample is clocked in.

See Also

HoldTime

HoldTime

Specify hold time for filter data input signals and forced reset input signals

Settings

ns

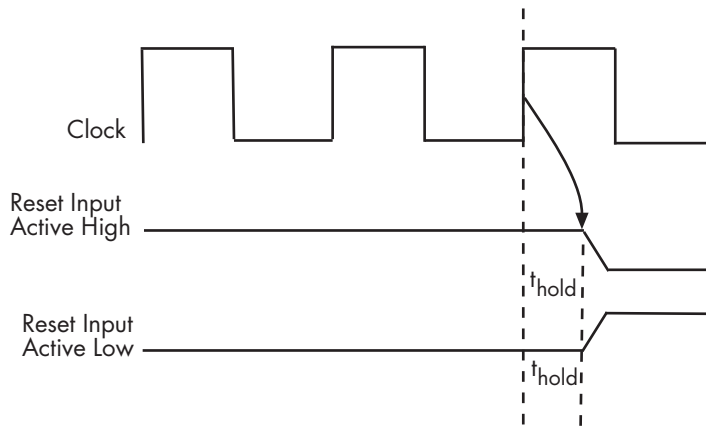
Specify the number of nanoseconds during which filter data input signals and forced reset input signals are held past the clock rising edge. The default is 2.

The hold time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

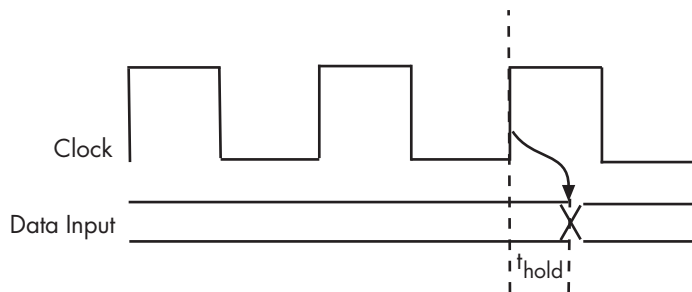
This option applies to reset input signals only if forced resets are enabled.

Usage Notes

The hold time is the amount of time that reset input signals and input data are held past the clock rising edge. The following figures show the application of a hold time (t_{hold}) for reset and data input signals when the signals are forced to active high and active low. The `ResetLength` property is set to its default of 2 cycles, and the reset signal is asserted for a total of two cycles plus t_{hold} .



Hold Time for Reset Input Signals



Hold Time for Data Input Signals

See Also

ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, ForceReset, HoldInputDataBetweenSamples

InitializeTestBenchInputs

Specify initial value driven on test bench inputs before data is asserted to filter

Settings

'on'

Initial value driven on test bench inputs is '0'.

'off' (default)

Initial value driven on test bench inputs is 'X' (unknown).

InlineConfigurations

Specify whether generated VHDL code includes inline configurations

Settings

'on' (default)

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.

Usage Notes

VHDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, the coder includes configurations for a filter within the generated VHDL code. If you are creating your own VHDL configuration files, you should suppress the generation of inline configurations.

See Also

CastBeforeSum, LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

InputComplex

Enable generation ports and signal paths that correspond to filters with complex input data

Settings

'on'

Use this option when your filter design requires complex input data. To enable generation of ports and signal paths for the real and imaginary components of a complex signal, set `InputComplex` to 'on', as in the following code example.

```
Hd = design(fdesign.lowpass,'equiripple','Filterstructure','dffir');  
generatehdl(Hd, 'InputComplex', 'on')
```

'off' (default)

Do not generate ports for complex input data

See Also

`ComplexImagPostfix`

`ComplexRealPostfix`

“Using Complex Data and Coefficients” on page 5-36

InputDataType

Specify input data type for system objects for HDL code generation

Settings

This property accepts an object of `numericType` class only.

Usage Notes

When you generate code for System objects, consider the following points:

- This capability is limited to code generation at the command line only, as shown in the following code example:

```
hDDC = dsp.DigitalDownConverter('Oscillator','NCO');
generatehdl(hDDC, 'InputDataType', numericType([], 8,7))
```
- You may not set the input and output port names. Filter Design HDL Coder assigns the port names.
- Inputs and outputs are registered by default.

InputPort

Name HDL port for filter's input signals

Settings

'string'

The default string is `filter_in`.

For example, if you specify the string `'filter_data_in'` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( clk           : IN  std_logic;
        clk_enable   : IN  std_logic;
        reset        : IN  std_logic;
        filter_data_in : IN  std_logic_vector (15 DOWNTO 0);
        filter_out    : OUT std_logic_vector (15 DOWNTO 0);
  );
END Hd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See [ReservedWordPostfix](#) for more information.

See Also

[ClockEnableInputPort](#), [ClockInputPort](#), [OutputPort](#), [OutputType](#), [ResetInputPort](#)

InputType

Specify HDL data type for filter's input port

Settings

'std_logic_vector'

Specifies VHDL type `STD_LOGIC_VECTOR` for the filter input port.

'signed/unsigned'

Specifies VHDL type `SIGNED` or `UNSIGNED` for the filter input port.

'wire' (Verilog)

If the target language is Verilog, the data type for ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, `ClockInputPort`, `InputPort`, `OutputPort`, `OutputType`, `ResetInputPort`

InstanceGenerateLabel

Specify string to append to instance section labels in VHDL GENERATE statements

Settings

'string'

Specify a postfix string to append to instance section labels in VHDL GENERATE statements. The default string is `_gen`.

See Also

BlockGenerateLabel, OutputGenerateLabel

InstancePrefix

Specify string prefixed to generated component instance names

Settings

'string'

Specify a string to be prefixed to component instance names in generated code. The default string is `u_`.

LoopUnrolling

Specify whether VHDL FOR and GENERATE loops are unrolled and omitted from generated VHDL code

Settings

'on'

Unroll and omit FOR and GENERATE loops from the generated VHDL code. Verilog is already unrolled.

This option takes into account that some EDA tools do not support GENERATE loops. If you are using such a tool, enable this option to omit loops from your generated VHDL code.

'off' (default)

Include FOR and GENERATE loops in the generated VHDL code.

Usage Notes

The setting of this option does not apply to generated VHDL code during simulation or synthesis.

See Also

CastBeforeSum, InlineConfigurations, LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

MultifileTestBench

Divide generated test bench into helper functions, data, and HDL test bench code files

Settings

'on'

Write separate files for test bench code, helper functions, and test bench data. The file names are derived from the test bench name and the `TestBenchDataPostfix` property as follows:

TestBenchName_TestBenchDataPostfix

For example, if the test bench name is `my_fir_filt`, and the target language is VHDL, the default test bench file names are:

- `my_fir_filt_tb.vhd`: test bench code
- `my_fir_filt_tb_pkg.vhd`: helper functions package
- `my_fir_filt_tb_data.vhd`: test vector data package

If the test bench name is `my_fir_filt` and the target language is Verilog, the default test bench file names are:

- `my_fir_filt_tb.v`: test bench code
- `my_fir_filt_tb_pkg.v`: helper functions package
- `my_fir_filt_tb_data.v`: test bench data

'off' (default)

Write a single test bench file containing HDL test bench code and helper functions and test bench data.

See Also

`TestBenchName`, `TestBenchDataPostFix`

MultiplierInputPipeline

Specify number of pipeline stages at multiplier inputs for FIR filters

Settings

nStages

Default: 0. nStages must be an integer greater than or equal to 0.

MultiplierInputPipeline lets you specify generation of pipeline stages at multiplier inputs for FIR filter structures. Multiplier pipelining can help you achieve significantly higher clock rates.

Usage Notes

The coder ignores this property if `CoeffMultipliers` is not set to `'multipliers'`.

See Also

`CoeffMultipliers`

MultiplierOutputPipeline

Specify number of pipeline stages at multiplier outputs for FIR filters

Settings

nStages

Default: 0. nStages must be an integer greater than or equal to 0.

MultiplierOutputPipeline lets you specify generation of pipeline stages at multiplier outputs for FIR filter structures. Multiplier pipelining can help you achieve significantly higher clock rates.

Usage Notes

The coder ignores this property if `CoeffMultipliers` is not set to `'multipliers'`.

See Also

`CoeffMultipliers`

Name

Specify file name for generated HDL code and for filter VHDL entity or Verilog module

Settings

'string'

The defaults take the name of the filter as defined in the FDATool.

The file type extension for the generated file is the string specified for the file type extension option for the selected language.

The generated file is placed in the folder or path specified by `TargetDirectory`.

If you specify a string that is a reserved word in the selected language, the coder appends the string specified by `ReservedWordPostfix`. For a list of reserved words, see “Resolving HDL Reserved Word Conflicts” on page 5-13.

See Also

`TargetDirectory`, `VerilogFileExtension`, `VHDLFileExtension`

NumMultipliers

Specify multipliers for IIR SOS filter with serial architecture

Settings

N

A value must be specified. N must be an integer greater than 1.

`NumMultipliers` defines N, the total number of multipliers used for the filter implementation in an IIR SOS filter with serial architecture.

Usage Notes

`NumMultipliers` is available for `df1sos` and `df2sos` filters with serial architecture only. Both `NumMultipliers` and `FoldingFactor` generate HDL for an IIR SOS filter with serial architecture, but you must select one property or the other; you may not use both.

If neither `NumMultipliers` or `FoldingFactor` is specified, HDL code for the filter is generated with Fully Parallel architecture.

The following example demonstrates generating HDL code for a `df2sos` filter with serial architecture and 2 multipliers:

```
Hd = design(fdesign.lowpass, 'ellip', 'FilterStructure', 'df2sos');
Hd.arithmetic = 'fixed';
generatehdl(Hd, 'NumMultipliers', 2)
```

For the legal values of `NumMultipliers`, use the helper function:

```
hdlfilterserialinfo(Hd)
```

Table of folding factors with corresponding number of multipliers for the given filter.

Folding Factor	Multipliers
6	3
9	2
18	1

See Also

FoldingFactor

OptimizeForHDL

Specify whether generated HDL code is optimized for specific performance or space requirements

Settings

'on'

Generate HDL code that is optimized for specific performance or space requirements. As a result of these optimizations, the coder may

- Make tradeoffs concerning data types
- Avoid excessive quantization
- Generate code that produces numeric results that differ from results produced by the original filter function

'off' (default)

Generate HDL code that maintains bit compatibility with the numeric results produced by the original filter function.

See Also

AddPipelineRegisters, CoeffMultipliers, FIRAdderStyle

OutputGenerateLabel

Specify string that labels output assignment block for VHDL GENERATE statements

Settings

'string'

Specify a postfix string to append to output assignment block labels in VHDL GENERATE statements. The default string is `outputgen`.

See Also

BlockGenerateLabel, InstanceGenerateLabel

OutputPort

Name HDL port for filter's output signals

Settings

'string'

The default is `filter_out`.

For example, if you specify `'filter_data_out'` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( clk           : IN  std_logic;
        clk_enable    : IN  std_logic;
        reset         : IN  std_logic;
        filter_in     : IN  std_logic_vector (15 DOWNTO 0);
        filter_data_out : OUT std_logic_vector (15 DOWNTO 0);
  );
ENDHd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See [ReservedWordPostfix](#) for more information.

See Also

[ClockEnableInputPort](#), [ClockInputPort](#), [InputPort](#), [InputType](#), [OutputType](#), [ResetInputPort](#)

OutputType

Specify HDL data type for filter's output port

Settings

'Same as input data type' (VHDL default)

The filter's output port has the same type as the specified input port type.

'std_logic_vector'

The filter's output port has VHDL type `STD_LOGIC_VECTOR`.

'signed/unsigned'

The filter's input port has type `SIGNED` or `UNSIGNED`.

'wire' (Verilog)

If the target language is Verilog, the data type for ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, `ClockInputPort`, `InputPort`, `InputType`, `OutputPort`, `ResetInputPort`

PackagePostfix

Specify string to append to specified filter name to form name of VHDL package file

Settings

'string'

The coder applies this option only if a package file is required for the design. The default string is `_pkg`.

See Also

`ClockProcessPostfix`, `CoeffPrefix`, `EntityConflictPostfix`,
`ReservedWordPostfix`

RemoveResetFrom

Suppress generation of resets from shift registers

Settings

'none' (default)

Do not suppress generation of resets from shift registers.

'ShiftRegister'

Suppress generation of resets from shift registers.

See Also

“Suppressing Generation of Reset Logic” on page 5-27

ReservedWordPostfix

Specify string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words

Settings

'string'

The default postfix is `_rsvd`.

For example, if you name your filter `mod`, the coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

See Also

`ClockProcessPostfix`, `CoeffPrefix`, `EntityConflictPostfix`, `PackagePostfix`

ResetAssertedLevel

Specify asserted (active) level of reset input signal

Settings

'active-high' (default)

Specify that the reset input signal must be driven high (1) to reset registers in the filter design. 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 reset input signal must be driven low (0) to reset registers in the filter design. 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'));
    .
    .
    .
```

See Also

ResetType

ResetInputPort

Name HDL port for filter's reset input signals

Settings

'string'

The default name for the filter's reset input port is `reset`.

For example, if you specify the string `'filter_reset'` for filter entity `Hd`, the generated entity declaration might look as follows:

```
ENTITY Hd IS
  PORT( clk           : IN  std_logic;
        clk_enable    : IN  std_logic;
        filter_reset   : IN  std_logic;
        filter_in      : IN  std_logic_vector (15 DOWNTO 0);
        filter_out     : OUT std_logic_vector (15 DOWNTO 0);
  );
END Hd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See [ReservedWordPostfix](#) for more information.

Usage Notes

If the reset asserted level is set to active high, the reset input signal is asserted active high (1) and the input value must be high (1) for the entity's registers to be reset. If the reset asserted level is set to active low, the reset input signal is asserted active low (0) and the input value must be low (0) for the entity's registers to be reset.

See Also

[ClockEnableInputPort](#), [ClockInputPort](#), [InputPort](#), [InputType](#), [OutputPort](#), [OutputType](#)

ResetLength

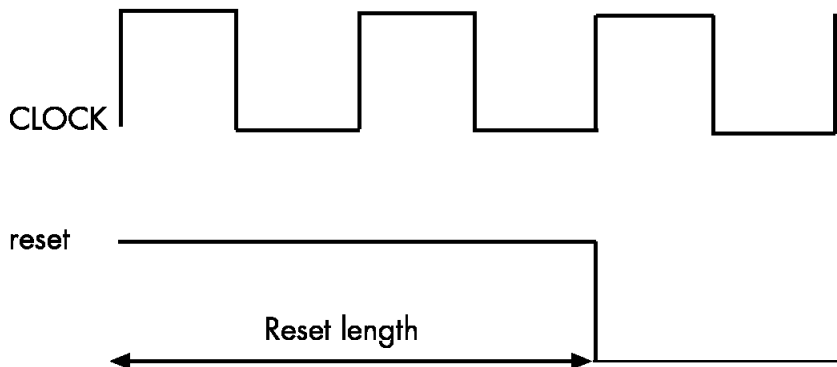
Define length of time (in clock cycles) during which reset is asserted

Settings

N

Default: 2. N must be an integer greater than or equal to 0.

Reset length defines N, the number of clock cycles during which reset is asserted. The following figure illustrates the default case, in which the reset signal (active-high) is asserted for 2 clock cycles.



ResetType

Specify whether to use asynchronous or synchronous reset style when generating HDL code for registers

Settings

'async' (default)

Use an asynchronous reset style. The following generated code fragment illustrates the use of asynchronous resets. Note that the process block does not check for an active clock before performing a reset.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF Reset_Port = '1' THEN
    delay_pipeline (0 To 50) <= (OTHERS =>(OTHERS => '0'));
  ELSIF Clock_Port'event AND Clock_Port = '1' THEN
    IF ClockEnable_Port = '1' THEN
      delay_pipeline(0) <= signed(Fin_Port)
      delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS delay_pipeline_process;
```

'sync'

Use a synchronous reset style. Code for a synchronous reset follows. This process block checks for a clock event, the rising edge, before performing a reset.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF rising_edge(Clock_Port) THEN
    IF Reset_Port = '0' THEN
      delay_pipeline(0 To 50) <= (OTHERS =>(OTHERS => '0'));
    ELSIF ClockEnable_Port = '1' THEN
      delay_pipeline(0) <= signed(Fin_Port)
      delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS delay_pipeline_process;
```

See Also

ResetAssertedLevel

ReuseAccum

Enable accumulator reuse, generating cascade-serial architecture for FIR filters

Settings

'off' (default)

Disable accumulator reuse.

'on'

Enable accumulator reuse when generating a partly serial architecture. (i.e., a cascade-serial architecture). If the number and size of serial partitions is not specified (see `SerialPartition`), the coder generates an optimal partition.

Usage Notes

In a cascade-serial architecture, filter taps are grouped into a number of serial partitions, and the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of the partitions is therefore computed at the accumulator of the first partition. This technique, termed *accumulator reuse*, saves chip area.

See “Speed vs. Area Tradeoffs” on page 4-2 for a complete description of parallel and serial architectures and a list of filter types supported for each architecture.

See Also

`SerialPartition`

SafeZeroConcat

Specify syntax used in generated VHDL code for concatenated zeros

Settings

'on' (default)

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 is more compact, but can lead to ambiguous types.

See Also

CastBeforeSum, InlineConfigurations, LoopUnrolling,
UseAggregatesForConst, UseRisingEdge

SerialPartition

Specify number and size of partitions generated for serial filter architectures

Settings

N

Generate a fully serial architecture for a filter of length N.

[p1 p2 p3...pN]

Where [p1 p2 p3...pN] is a vector of N integers, generate a partly serial architecture with N partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter.

{[p1 p2 p3...pNa], [p1 p2 p3...pNb], ...}

Where each vector in a cell array represents a serial partitioning of an individual filter within a cascade of filters.

Usage Notes

To save chip area in a partly serial architecture, you can enable the `ReuseAccum` property.

See “Speed vs. Area Tradeoffs” on page 4-2 for a complete description of parallel and serial architectures and a list of filter types supported for each architecture.

See Also

`ReuseAccum`

SimulatorFlags

Specify simulator flags applied to generated test bench

Settings

'string'

Specify options that are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag `-93`.

Usage Notes

The flags you specify with this option are added to the compilation command in generated EDA tool scripts. The compilation command string is specified by the `HDLCompileVHDLCmd` or `HDLCompileVerilogCmd` properties.

See Also

`HDLCompileVerilogCmd`, `HDLCompileVHDLCmd`

SplitArchFilePostfix

Specify string to append to specified name to form name of file containing filter's VHDL architecture

Settings

'string'

The default is `_arch`. This option applies only if you direct the coder to place the filter's entity and architecture in separate files.

Usage Notes

The option applies only if you direct the coder to place the filter's entity and architecture in separate files.

See Also

`SplitEntityArch`, `SplitEntityFilePostfix`

SplitEntityArch

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

'on'

Write the code for the filter VHDL entity and architecture to separate files.

The names of the entity and architecture files derive from the base file name (as specified by the filter 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 string. The file type extension is specified by the **VHDL file extension** option.

For example, instead of the generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

'off' (default)

Write the generated filter VHDL code to a single file.

See Also

`SplitArchFilePostfix`, `SplitEntityFilePostfix`

SplitEntityFilePostfix

Specify string to append to specified filter name to form name of file that contains filter's VHDL entity

Settings

'string'

The default is `_entity`. This option applies only if you direct the coder to place the filter's entity and architecture in separate files.

Usage Notes

This option applies only if you direct the coder to place the filter's entity and architecture in separate files.

See Also

SplitArchFilePostfix, SplitEntityArch

TargetDirectory

Identify folder for generated output files

Settings

Specify the subfolder under the current working folder into which generated files are written. The string can specify a complete pathname. The default string is `hdlsrc`.

See Also

Name, VerilogFileExtension, VHDLFileExtension

TargetLanguage

Specify HDL language to use for generated filter code

Settings

'VHDL' (default)

Generate VHDL filter code.

'verilog'

Generate Verilog filter code.

TestBenchClockEnableDelay

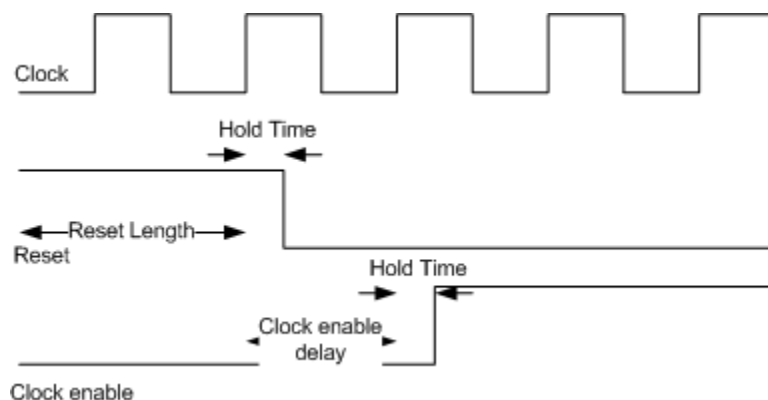
Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable

Settings

N (integer number of clock cycles) Default: 1

The `TestBenchClockEnableDelay` property specifies a delay time N, expressed in clock cycles (the default value is 1) elapsed between the time the reset signal is deasserted and the time the clock enable signal is first asserted. `TestBenchClockEnableDelay` works in conjunction with the `HoldTime` property. After deassertion of reset, the clock enable goes high after a delay of N clock cycles plus the delay specified by `HoldTime`.

In the figure below, the reset signal (active-high) deasserts after the interval labeled `Hold Time`. The clock enable asserts after a further interval labeled `Clock enable delay`.



See Also

`HoldTime`, `ResetLength`

TestbenchCoeffStimulus

Specify testing options for coefficient memory interface for FIR or IIR filters

Settings

[] : Empty vector (default)

When the value of `TestbenchCoeffStimulus` is unspecified (or set to the default value of []), the test bench loads the coefficients from the filter object and then forces the input stimuli. This shows the response to the input stimuli and verifies that the interface writes one set of coefficients into the RAM as expected.

(For FIR filters): vector of coefficient values . In this case, the filter processes the input stimuli twice. First, the test bench loads the coefficients from the filter object and forces the input stimuli to show the response. Then, the filter loads the set of coefficients specified in the `TestbenchCoeffStimulus` vector, and shows the response by processing the same input stimuli for a second time. In this case, the internal states of the filter, as set by the first run of the input stimulus, are retained. The test bench verifies that the interface writes two different sets of coefficients into the RAM. See “Programmable Filter Coefficients for FIR Filters” on page 3-27 for further information.

(for IIR filters): Cell array containing a column vector of scale values, and a second-order section (SOS) matrix for the filter. See “Programmable Filter Coefficients for IIR Filters” on page 3-39 for further information.

See Also

`CoefficientSource`

TestBenchDataPostFix

Specify suffix added to test bench data file name when generating multi-file test bench

Settings

'string'

The default postfix is '_data'.

The coder applies `TestBenchDataPostFix` only when generating a multi-file test bench (i.e., when `MultifileTestBench` is set 'on').

For example, if the name of your test bench is `test_fir_tb`, the coder adds the postfix `_data` to form the test bench data file name `test_fir_tb_data`.

See Also

`MultifileTestBench`

TestBenchFracDelayStimulus

Specify input stimulus that test bench applies to Farrow filter fractional delay port

Settings

Note: This option applies only to Farrow filters.

Default: A constant value is obtained from the `FracDelay` property of the Farrow filter object, and applied to the fractional delay port.

'RandSweep'

A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.

'RampSweep'

A vector of random values in the range from 0 to 1. This stimulus signal has the same duration as the filter's input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal before the next value is obtained.

Vector or function returning a vector

Define a vector as a workspace variable, store the stimulus signal in the vector, and pass in the vector name. Alternatively, pass in a call to a function that returns a vector.

See Also

`FracDelayPort`, “Single-Rate Farrow Filters” on page 3-20

TestBenchName

Name VHDL test bench entity or Verilog module and file that contains test bench code

Settings

'string'

The file type extension depends on the type of test bench that is being generated:

- For Verilog files, the extension is defined by the Verilog file extension option.
- For VHDL files, the extension is defined by the VHDL file extension option.

The file is placed in the target folder.

If you specify a string that is a VHDL or Verilog reserved word, a reserved word postfix string is appended to form a valid HDL identifier. For example, if you specify the reserved word `entity`, the resulting name string would be `entity_rsvd`. To set the reserved word postfix string, see `ReservedWordPostfix`.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

TestbenchRateStimulus

Specify rate stimulus for CIC filter with rate port

Settings

`TestbenchRateStimulus` specifies the rate stimulus to be loaded into the rate port for a variable rate CIC filter . If you do not specify `TestbenchRateStimulus`, the coder uses the maximum rate change factor specified in the filter object.

Usage Notes

`TestbenchRateStimulus` is specifically for use with variable rate CIC filters.

See Also

`AddRatePort`, “Variable Rate CIC Filters” on page 3-8

TestBenchReferencePostFix

Specify string appended to names of reference signals generated in test bench code

Settings

'string'

The default postfix is '_ref'.

Reference signal data is represented as arrays in the generated test bench code. The string specified by `TestBenchReferencePostFix` is appended to the generated signal names.

TestBenchStimulus

Specify input stimuli that test bench applies to filter

Settings

'impulse'

Specify that the test bench acquire an impulse stimulus response. The impulse response is output arising from the unit impulse input sequence defined such that the value of $x(n)$ is 1 when n equals 1 and $x(n)$ equals 0 when n does not equal 1.

'step'

Specify that the test bench acquire a step stimulus response.

'ramp'

Specify that the test bench acquire a ramp stimulus response, which is a constantly increasing or constantly decreasing signal.

'chirp'

Specify that the test bench acquire a chirp stimulus response, which is a linear swept-frequency cosine signal.

'noise'

Specify that the test bench acquire a white noise stimulus response.

The coder assigns default stimuli based on the filter type.

Usage Notes

You can specify combinations of stimuli in whatever order you wish. If you specify multiple stimuli, specify the corresponding strings in a cell array. For example:

```
{'impulse', 'ramp', 'noise'}
```

See Also

TestBenchUserStimulus

TestBenchUserStimulus

Specify user-defined function that returns vector of values that test bench applies to filter

Settings

function call

For example, the following function call generates a square wave with a sample frequency of 8 bits per second ($F_s/8$):

```
repmat([1 1 1 1 0 0 0 0], 1, 10)
```

See Also

TestBenchStimulus

UseAggregatesForConst

Specify whether constants are represented by aggregates, including constants that are less than 32 bits wide

Settings

'on'

Specify that constants, including constants that are less than 32 bits, be represented by aggregates. The following VHDL constant declarations show scalars less than 32 bits being declared as aggregates:

```
CONSTANT c1: signed(15 DOWNT0 0):= (5 DOWNT0 3 =>'0',1 DOWNT0 0 => '0',OTHERS =>'1');  
CONSTANT c2: signed(15 DOWNT0 0):= (7 => '0',5 DOWNT0 4 =>'0',0 => '0',OTHERS =>'1');
```

'off' (default)

Specify that the coder represent constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. This is the default. The following VHDL constant declarations are examples of declarations generated by default for values less than 32 bits:

```
CONSTANT coeff1: signed(15 DOWNT0 0) := to_signed(-60, 16); -- sfix16_En16  
CONSTANT coeff2: signed(15 DOWNT0 0) := to_signed(-178, 16); -- sfix16_En16
```

See Also

CastBeforeSum, InlineConfigurations, , LoopUnrolling, SafeZeroConcat, UseRisingEdge, UseVerilogTimescale

UserComment

Specify comment line in header of generated filter and test bench files

Settings

The coder includes a header comment block at the top of the files it generates. The header comment block contains information about the specifications of the generating filter and about the coder options that were selected at the time HDL code was generated.

You can add your own comment lines to the header comment block by setting `UserComment` to the desired string value. The code generator adds leading comment characters that correspond to the target language. When you include newlines or line feeds in the string, the coder emits single-line comments for each newline.

For example, the following `generatehdl` command adds two comment lines to the header in a generated VHDL file.

```
generatehdl(Hlp,'UserComment','This is a comment line.\nThis is a second line.')
```

The resulting header comment block for filter `Hlp` would appear as follows:

```
-- -----
--
-- Module: Hlp
--
-- Generated by MATLAB(R) 7.11 and the Filter Design HDL Coder 2.7.
--
-- Generated on: 2010-08-31 13:32:16
--
-- This is a comment line.
-- This is a second line.
--
-- -----
--
-- HDL Code Generation Options:
--
-- TargetLanguage: VHDL
-- Name: Hlp
-- UserComment: User data, length 47
--
-- Filter Specifications:
--
-- Sampling Frequency : N/A (normalized frequency)
-- Response           : Lowpass
-- Specification      : Fp,Fst,Ap,Ast
-- Passband Edge     : 0.45
```

```
-- Stopband Edge      : 0.55
-- Passband Ripple   : 1 dB
-- Stopband Atten.   : 60 dB
-----
-- HDL Implementation : Fully parallel
-- Multipliers        : 43
-- Folding Factor     : 1
-----
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-----
-- Filter Structure   : Direct-Form FIR
-- Filter Length      : 43
-- Stable             : Yes
-- Linear Phase       : Yes (Type 1)
-- Arithmetic         : fixed
-- Numerator          : s16,16 -> [-5.000000e-001 5.000000e-001)
-- Input              : s16,15 -> [-1 1)
-- Filter Internals   : Full Precision
-- Output             : s33,31 -> [-2 2) (auto determined)
-- Product            : s31,31 -> [-5.000000e-001 5.000000e-001) (auto determined)
-- Accumulator        : s33,31 -> [-2 2) (auto determined)
-- Round Mode         : No rounding
-- Overflow Mode      : No overflow
-----
```

UseRisingEdge

Specify VHDL coding style used to check for rising edges when operating on registers

Settings

'on'

Use the VHDL `rising_edge` function to check for rising edges when operating on registers. The generated code applies `rising_edge` as shown in the following `PROCESS` block:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    ELSIF rising_edge(clk) THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= signed(filter_in);
        delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_Process ;
```

'off' (default)

Check for clock events when operating on registers. The generated code checks for a clock event as shown in the `ELSIF` statement of the following `PROCESS` block:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= signed(filter_in);
        delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
      END IF;
    END IF;
  END PROCESS Delay_Pipeline_Process ;
```

Usage Notes

The two coding styles have different simulation behavior when the clock transitions from 'X' to '1'.

See Also

`CastBeforeSum`, `InlineConfigurations`, `LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`

UseVerilogTimescale

Allow or exclude use of compiler ``timescale` directives in generated Verilog code

Settings

'on' (default)

Use compiler ``timescale` directives in generated Verilog code.

'off'

Suppress the use of compiler ``timescale` directives in generated Verilog code.

Usage Notes

The ``timescale` directive provides a way of specifying different delay values for multiple modules in a Verilog file.

See Also

CastBeforeSum, InlineConfigurations, LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

VectorPrefix

Specify string prefixed to vector names in generated VHDL code

Settings

'string'

Specify a string to be prefixed to vector names in generated VHDL code. The default string is `vector_of_`.

VerilogFileExtension

Specify file type extension for generated Verilog files

Settings

'string'

The default file type extension for generated Verilog files is `.v`.

See Also

Name, TargetDirectory

VHDLArchitectureName

Specify architecture name for generated VHDL code

Settings

'string'

The default string is rtl.

VHDLFileExtension

Specify file type extension for generated VHDL files

Settings

'string'

The default file type extension for generated VHDL files is .vhd.

See Also

Name, TargetDirectory

VHDLlibraryName

Specify target library name used in initialization section of compilation script

Settings

'string'

The default string is 'work'.

At script generation time, the string you specify as VHDLlibraryName substitutes into the HDLCompileInit string value. By default, this generates the library specification 'vlib work/n'

You can use VHDLlibraryName to avoid library name conflicts.

See Also

HDLCompileInit

“Integration With Third-Party EDA Tools” on page 6-36

Function Reference

fdhdltool

Open Generate HDL dialog box

Syntax

```
fdhdltool(Hd)
```

Description

`fdhdltool(Hd)` is a convenience function that lets you open the Generate HDL dialog box from the command line, passing in the filter object handle `Hd`. When the Generate HDL dialog box opens, it displays default values for code generation options applicable to the filter object. You can then use the Generate HDL dialog box and its subordinate dialog boxes to specify code generation options and initiate generation of HDL and test bench code and scripts for third-party EDA tools.

`fdhdltool` operates on a copy of the filter object, rather than the original object in the workspace. Changes made to the original filter object after `fdhdltool` is invoked do not apply to the copy and do not update the Generate HDL dialog box.

The naming convention for the copied object is `filt_copy`, where `filt` is the name of the original filter object.

Examples

The following code example designs a lowpass filter, constructs a direct-form FIR filter object, `Hd`, and then opens the Generate HDL dialog box.

```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear')
Hd = design(filtdes,'filterstructure','dffir')
Hd.arithmetic = 'fixed'
fdhdltool(Hd)
```

More About

- “Opening the Filter Design HDL Coder GUI Using the `fdhdltool` Command” on page 2-10

generatehdl

Generate HDL code for quantized filter

Syntax

```
generatehdl(Hd)  
generatehdl(Hd,Name,Value)
```

Description

`generatehdl(Hd)` generates HDL code for a quantized filter using default settings.

- The function places generated files in a subfolder name `hdlsrc`, under your current working folder.
- The function includes the VHDL entity and architecture code in a single source file.

`generatehdl(Hd,Name,Value)` generates HDL code with additional options specified by one or more `Name,Value` pair arguments.

Examples

Generate HDL Code for FIR Equiripple Filter

Call `fdesign` to pass the specifications for designing a minimum order lowpass filter.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2, 0.22, 1, 60)
```

Those specifications determine the following characteristics for this filter:

- Normalized passband frequency of 0.2
- Stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

Call the `design` function to construct the FIR equiripple filter object `Hd`.

```
Hd = design(d, 'equiripple')
```

Set the filter arithmetic to fixed-point using the arithmetic assignment statement.

```
Hd.arithmetic = 'fixed'
```

Call function `generatehdl` to generate VHDL code for the FIR equiripple filter `Hd`.

```
generatehdl(Hd, 'Name', 'MyFilter')
```

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\Users\jhenley\hdlsrc\MyFilter.vhd
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: MyFilter
```

The function names the file `MyFilter.vhd` and places it in the default target folder `hdlsrc`.

Input Arguments

Hd — Filter object

filter object name returned by `design`

Filter object for which to generate HDL, specified as the name of a filter object, returned by the `design` function.

Example: `generatehdl(Hd, 'GenerateHDLTestbench', 'on')`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'TargetLanguage','Verilog'`

Language Selection

'TargetLanguage' — Target language

'VHDL' (default) | 'Verilog'

For more information, see TargetLanguage.

File Naming and Location

'Name' — Specify file name for generated HDL code and for filter VHDL entity or Verilog module

string

For more information, see Name.

'TargetDirectory' — Output directory

'hdlsrc' (default) | string

For more information, see TargetDirectory.

'VerilogFileExtension' — Verilog file extension

'.v' (default) | string

For more information, see VerilogFileExtension.

'VHDLFileExtension' — VHDL file extension

'.vhd' (default) | string

For more information, see VHDLFileExtension.

Resets

'RemoveResetFrom' — Suppress generation of resets from shift registers

'none' (default) | 'ShiftRegister'

For more information, see RemoveResetFrom.

'ResetAssertedLevel' — Asserted (active) level of reset

'active-high' (default) | 'active-low'

For more information, see ResetAssertedLevel.

'ResetLength' — Define length of time (in clock cycles) during which reset is asserted

2 (default) | N

For more information, see `ResetLength`.

'ResetType' — Reset type
'async' (default) | 'sync'

For more information, see `ResetType`.

Header Comment and General Naming

'ClockProcessPostfix' — Postfix for clock process names
'_process' (default) | string

For more information, see `ClockProcessPostfix`.

'CoeffPrefix' — Specify prefix (string) for filter coefficient names
coeff (default) | string

For more information, see `CoeffPrefix`.

'ComplexImagPostfix' — Postfix for imaginary part of complex signal
'_im' (default) | string

For more information, see `ComplexImagPostfix`.

'ComplexRealPostfix' — Postfix for imaginary part of complex signal names
'_re' (default) | string

For more information, see `ComplexRealPostfix`.

'EntityConflictPostfix' — Postfix for duplicate VHDL entity or Verilog module names
'_block' (default) | string

For more information, see `EntityConflictPostfix`.

'InstancePrefix' — Prefix for generated component instance names
'u_' (default) | string

For more information, see `InstancePrefix`.

'PackagePostfix' — Postfix for package file name
'_pkg' (default) | string

For more information, see PackagePostfix.

'ReservedWordPostfix' — Postfix for names conflicting with VHDL or Verilog reserved words

'_rsvd' (default) | string

For more information, see ReservedWordPostfix.

'SplitArchFilePostfix' — Postfix for VHDL architecture file names

'_arch' (default) | string

For more information, see SplitArchFilePostfix.

'SplitEntityArch' — Split VHDL entity and architecture into separate files

'off' (default) | 'on'

For more information, see SplitEntityArch.

'SplitEntityFilePostfix' — Postfix for VHDL entity file names

'_entity' (default) | string

For more information, see SplitEntityFilePostfix.

'UserComment' — HDL file header comment

string

For more information, see UserComment.

'VectorPrefix' — Prefix for vector names

'vector_of_' (default) | string

For more information, see VectorPrefix.

'VHDLArchitectureName' — VHDL architecture name

'rtl' (default) | string

For more information, see VHDLArchitectureName.

'VHDLLibraryName' — VHDL library name

'work' (default) | string

For more information, see VHDLLibraryName.

Ports

'AddInputRegister' — Extra register indicator to HDL code for filter input

'on' (default) | 'off'

For more information, see AddInputRegister.

'AddOutputRegister' — Generate extra register in HDL code for filter output

'on' (default) | 'off'

For more information, see AddOutputRegister.

'ClockEnableInputPort' — HDL port name for filter clock enable input signals

clk_enable (default) | string

For more information, see ClockEnableInputPort.

'ClockEnableOutputPort' — Name of clock enable output port for multirate filters with a single clock

ce_out (default) | string

For more information, see ClockEnableOutputPort.

'ClockInputPort' — HDL port name for filter clock input signals

clk (default) | string

For more information, see ClockInputPort.

'ClockInputs' — Generation of single or multiple clock inputs for multirate filters

'Single' (default) | 'Multiple'

For more information, see ClockInputs.

'FracDelayPort' — Name port for Farrow filter fractional delay input signal

'filter_fd' (default) | string

For more information, see FracDelayPort.

'InputComplex' — Enable generation ports and signal paths that correspond to filters with complex input data

'off' (default) | 'on'

For more information, see InputComplex.

'InputDataType' — Specify input data type for system objects for HDL code generation
object of `numeric_type` class

For more information, see `InputDataType`.

'InputPort' — Name HDL port for filter input signals
'filter_in' (default) | string

For more information, see `InputPort`.

'InputType' — Specify HDL data type for filter input port
'std_logic_vector' | 'signed/unsigned' | 'wire' (Verilog)

For more information, see `InputType`.

'OutputPort' — Name HDL port for filter output signals
'filter_out' (default) | string

For more information, see `OutputPort`.

'OutputType' — Specify HDL data type for filter output port
'Same as input data type' (VHDL default) | 'std_logic_vector' | 'signed/unsigned' | 'wire' (Verilog)

For more information, see `OutputType`.

'ResetInputPort' — Name HDL port for filter reset input signals
'reset' (default) | string

For more information, see `ResetInputPort`.

Advanced Coding

'AddRatePort' — Generate rate ports for variable-rate CIC filter
'off' (default) | 'on'

For more information, see `AddRatePort`.

'BlockGenerateLabel' — Block label for HDL GENERATE statements
_gen (default) | string

For more information, see `BlockGenerateLabel`.

'CastBeforeSum' — Type casting of input values enable or disable for addition and subtraction operations

'off' (default) | 'on'

For more information, see `CastBeforeSum`

'CoefficientMemory' — Specify type of memory for storage of programmable coefficients for serial FIR filters settings

'Registers' (default) | 'DualPortRAMs' | 'SinglePortRAMs'

For more information, see `CoefficientMemory`.

'CoefficientSource' — Specify source for FIR or IIR filter coefficients

'Internal' (default) | 'ProcessorInterface'

For more information, see `CoefficientSource`.

'InlineConfigurations' — Include VHDL configurations

'on' (default) | 'off'

For more information, see `InlineConfigurations`.

'InstanceGenerateLabel' — Instance section label postfix for VHDL GENERATE statements

'_gen' (default) | string

For more information, see `InstanceGenerateLabel`.

'LoopUnrolling' — Unroll VHDL FOR and GENERATE loops

'off' (default) | 'on'

For more information, see `LoopUnrolling`.

'OutputGenerateLabel' — Output assignment label postfix for VHDL GENERATE statements

'outputgen' (default) | string

For more information, see `OutputGenerateLabel`.

'SafeZeroConcat' — Type-safe syntax for concatenated zeros

'on' (default) | 'off'

For more information, see `SafeZeroConcat`.

'UseAggregatesForConst' — Represent constant values with aggregates
'off' (default) | 'on'

For more information, see UseAggregatesForConst.

'UseRisingEdge' — Use VHDL `rising_edge` function to clock registers
'off' (default) | 'on'

For more information, see UseRisingEdge.

'UseVerilogTimescale' — Generate `timescale` compiler directives
'on' (default) | 'off'

For more information, see UseVerilogTimescale.

Optimizations

'AddPipelineRegisters' — Add pipeline register indicator for optimizing filter code clock rate
'off' (default) | 'on'

For more information, see AddPipelineRegisters.

'CoeffMultipliers' — Specify technique used for processing coefficient multiplier operations
'multiplier' (default) | 'csd' | 'factored-csd'

For more information, see CoeffMultipliers.

'DalutPartition' — Specify number and size of LUT partitions for distributed arithmetic architecture
[p1 p2...pN], a vector of N integers

For more information, see DALUTPartition.

'DARadix' — Specify number of bits processed simultaneously in distributed arithmetic architecture
2 (default) | N, a nonzero positive integer that is a power of two

For more information, see DARadix.

'FIRAdderStyle' — Specify final summation technique used for FIR filters

'linear' (default) | 'tree'

For more information, see FIRAdderStyle.

'FoldingFactor' — Specify folding factor for IIR SOS filter with serial architecture

integer greater than 1

For more information, see FoldingFactor.

'MultiplierInputPipeline' — Specify number of pipeline stages at multiplier inputs for FIR filters

0 (default) | integer

For more information, see MultiplierInputPipeline.

'MultiplierOutputPipeline' — Specify number of pipeline stages at multiplier outputs for FIR filters

0 (default) | integer

For more information, see MultiplierOutputPipeline.

'NumMultipliers' — Specify multipliers for IIR SOS filter with serial architecture

integer greater than 1

For more information, see NumMultipliers.

'OptimizeForHDL' — Specify whether generated HDL code is optimized for specific performance or space requirements

'off' (default) | 'on'

For more information, see OptimizeForHDL.

'ReuseAccum' — Enable accumulator reuse, generating cascade-serial architecture for FIR filters

'off' (default) | 'on'

For more information, see ReuseAccum.

'SerialPartition' — Specify number and size of partitions generated for serial filter architectures

[p1 p2 p3 . . . pN], a vector of N integers

For more information, see SerialPartition.

Test Bench

'ClockHighTime' — Period during which test bench drives clock input signals high (1)
5 (default) | ns

For more information, see ClockHighTime.

'ClockLowTime' — Specify period, in nanoseconds, during which test bench drives clock input signals low (0)
5 (default) | ns

For more information, see ClockLowTime.

'ErrorMargin' — Specify error margin for HDL language-based test benches
integer number of bits

For more information, see ErrorMargin.

'ForceClock' — Specify whether test bench forces clock input signals
'on' (default) | 'off'

For more information, see ForceClock.

'ForceClockEnable' — Specify whether test bench forces input signals for clock enable
'on' (default) | 'off'

For more information, see ForceClockEnable.

'ForceReset' — Specify whether test bench forces reset input signals
'on' (default) | 'off'

For more information, see ForceReset.

'GenerateCoSimBlock' — Generate HDL Cosimulation block
'off' (default) | 'on'

For more information, see GenerateCoSimBlock.

'GenerateCoSimModel' — Generate HDL Cosimulation model
'ModelSim' (default) | 'Incisive'

For more information, see GenerateCosimModel.

'GenerateHDLTestbench' — Enable generation of a test bench

'off' (default) | 'on'

For more information, see `GenerateHDLTestbench`.

'HoldInputDataBetweenSamples' — Specify how long input data values are held in valid state

'on' (default) | 'off'

For more information, see `HoldInputDataBetweenSamples`.

'HoldTime' — Specify hold time for filter data input signals and forced reset input signals

5 (default) | ns

For more information, see `HoldTime`.

'InitializeTestBenchInputs' — Specify initial value driven on test bench inputs before data is asserted to filter

'off' (default) | 'on'

For more information, see `InitializeTestBenchInputs`.

'MultifileTestBench' — Divide generated test bench into helper functions, data, and HDL test bench code files

'off' (default) | 'on'

For more information, see `MultifileTestBench`.

'SimulatorFlags' — Specify simulator flags applied to generated test bench

string

For more information, see `SimulatorFlags`.

'TestBenchClockEnableDelay' — Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable

1 (default) | integer number of clock cycles

For more information, see `TestBenchClockEnableDelay`.

'TestbenchCoeffStimulus' — Specify testing options for coefficient memory interface for FIR or IIR filters

empty vector (default) | vector

For more information, see TestbenchCoeffStimulus.

'TestBenchDataPostFix' — Specify suffix added to test bench data file name when generating multifile test bench

'_data' (default) | string

For more information, see TestBenchDataPostFix.

'TestBenchFracDelayStimulus' — Specify input stimulus that test bench applies to Farrow filter fractional delay port

constant (either 'RandSweep' or 'RampSweep') (default) | vector or function returning a vector

For more information, see TestBenchFracDelayStimulus.

'TestBenchName' — Name VHDL test bench entity or Verilog module and file that contains test bench code

string

For more information, see TestBenchName.

'TestbenchRateStimulus' — Specify rate stimulus for CIC filter with rate port

integer

For more information, see TestbenchRateStimulus.

'TestBenchReferencePostFix' — Specify string appended to names of reference signals generated in test bench code

'_ref' (default) | string

For more information, see TestBenchReferencePostFix.

'TestBenchStimulus' — Specify input stimuli that test bench applies to filter

'impulse' | 'step' | 'ramp' | 'chirp' | 'noise'

For more information, see TestBenchStimulus.

'TestBenchUserStimulus' — Specify user-defined function that returns vector of values that test bench applies to filter

function call

For more information, see TestBenchUserStimulus.

Script Generation

'EDAScriptGeneration' — Enable or disable script generation for third-party tools
'on' (default) | 'off'

For more information, see EDAScriptGeneration.

'HDLCompileFilePostfix' — Specify postfix string appended to file name for generated Mentor Graphics ModelSim compilation scripts
'_compile.do' (default) | string

For more information, see HDLCompileFilePostfix.

'HDLCompileInit' — Compilation script initialization string
'vlib work\n' (default) | string

For more information, see HDLCompileInit.

'HDLCompileTerm' — Compilation script termination string
' ' (default) | string

For more information, see HDLCompileTerm.

'HDLCompileVerilogCmd' — Verilog compilation command
'vlog %s %s\n' (default) | string

For more information, see HDLCompileVerilogCmd.

'HDLCompileVHDLCmd' — VHDL compilation command
'vcom %s %s\n' (default) | string

For more information, see HDLCompileVHDLCmd.

'HDLSimCmd' — Specify simulation command written to simulation script
'vsim -novopt %s.%s\n' (default) | string

For more information, see HDLSimCmd.

'HDLSimFilePostfix' — Specify postfix string appended to file name for generated Mentor Graphics ModelSim simulation scripts
'_sim.d' (default) | string

For more information, see HDLSimFilePostfix.

'HDLsimInit' — Specify string written to initialization section of simulation script

```
['onbreak resume\n',...
'onerror resume\n'] (default) | string
```

For more information, see HDLSimInit.

'HDLsimTerm' — Specify string written to termination section of simulation script

```
'run -all\n' (default) | string
```

For more information, see HDLSimTerm.

'HDLsimViewWaveCmd' — Specify waveform viewing command written to simulation script

```
'add wave sim:%s\n' (default) | string
```

For more information, see HDLSimViewWaveCmd.

'HDLSynthCmd' — HDL synthesis command

```
'add_file %s\n' (default) | string
```

For more information, see HDLSynthCmd.

'HDLSynthFilePostfix' — Postfix for synthesis script file name

a string that corresponds to the synthesis tool specified by HDLSynthTool. (default) | string

For more information, see HDLSynthFilePostfix.

'HDLSynthInit' — Synthesis script initialization string

```
'project -new %s.prj\n' (default) | string
```

For more information, see HDLSynthInit.

'HDLSynthTerm' — Synthesis script termination string

```
['set_option -technology VIRTEX4\n',...
'set_option -part XC4VSX35\n',...
'set_option -synthesis_onoff_pragma 0\n',...
'set_option -frequency auto\n',...
'project -run synthesis\n'] (default) | string
```

For more information, see HDLSynthTerm.

'HDLSynthTool' — Synthesis tool

```
'None' (default) | 'ISE' | 'Precision' | 'Quartus' | 'Synplify'
```

For more information, see HDLSynthTool.

See Also

generatetbstimulus

generatetb

Generate HDL test bench for filter

Syntax

```
generatetb(Hd, 'TbType')
generatetb(Hd 'TbType', 'PropertyName', 'PropertyValue',...)
generatetb(Hd)
generatetb(Hd, 'PropertyName', 'PropertyValue',...)
```

Description

Note: In Release R2011a, the `generatetb` function is deprecated. To generate a test bench for your HDL filter code, you should use the `generatehdl` function, setting the `GenerateHDLTestbench` property to `'on'`, as in the following example.

```
generatehdl(Hlp, 'GenerateHDLTestbench', 'on')
```

In Release R2011a, the `generatetb` function will continue to operate, but will display a warning message when it is called.

You should replace calls to `generatetb` in your scripts with calls to `generatehdl`. This change enables test bench generation with the `GenerateHDLTestbench` property, as shown in the preceding example.

`generatetb(Hd, 'TbType')` generates a HDL test bench of a specified type to verify the HDL code generated for the quantized filter identified by `Hd`. The argument `Hd` must be a handle to a filter object. The value that you specify for `'TbType'` identifies the type of test bench to be generated. This value can be one of those shown in the following table or a cell array that contains one or more of these values.

If you do not specify the `'TbType'` argument, the test bench type defaults to the current setting of the `TargetLanguage` property (`'VHDL'` or `'Verilog'`).

Specify...	To Generate a Test Bench Consisting of...
<code>'Verilog'</code>	Verilog code

Specify...	To Generate a Test Bench Consisting of...
'VHDL'	VHDL code

The generated test bench applies input stimuli based on the setting of the properties `TestBenchStimulus` and `TestBenchUserStimulus`. The coder assigns default stimuli based on the filter type. The following choices of stimuli are available:

- 'impulse'
- 'step'
- 'ramp'
- 'chirp'
- 'noise'

The function uses default settings for other properties that determine test bench characteristics.

Default Settings for the Test Bench

- Places the generated test bench file in the target folder `hdlsrc` under your current working folder. This file has the name `Hd_tb` and a file type extension that is based on the type of test bench you are generating.

If the Test Bench Is a...	The Extension Is...
Verilog file	Defined by the property <code>VerilogFileExtension</code>
VHDL file	Defined by the property <code>VHDLFileExtension</code>

- Generates script files for third-party EDA tools. Where *Hd* is the name of the specified filter object, the following script files are generated:
 - `Hd_tb_compile.do`: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
 - `Hd_tb_sim.do`: Mentor Graphics ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.
- Forces clock, clock enable, and reset input signals.
- Forces clock enable and reset input to active high.

- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces reset signals.
- Applies a hold time of 2 nanoseconds to filter reset and data input signals.
- For HDL test benches, applies an error margin of 4 bits.

Default Settings for Files

- Places generated files in the target folder `hdlsrc` and names the files as follows:

File	Name
Verilog source	<i>Hd</i> .v, where <i>Hd</i> is the name of the specified filter object
VHDL source	<i>Hd</i> .vhd, where <i>Hd</i> is the name of the specified filter object
VHDL package	<i>Hd_pkg</i> .vhd, where <i>Hd</i> is the name of the specified filter object

- Places generated files in a subfolder name `hdlsrc`, under your current working folder.
- Includes VHDL entity and architecture code in a single source file.

Default Settings for Register Resets

- Uses an asynchronous reset when generating HDL code for registers.
- Asserts the reset input signal high (1) to reset registers in the design.

Default Settings for General HDL Code

- Names the generated VHDL entity or Verilog module with the name of the filter.
- Names the filter's HDL ports as follows:

HDL Port	Name
Input	<code>filter_in</code>
Output	<code>filter_out</code>
Clock input	<code>clk</code>
Clock enable input	<code>clk_enable</code>
Reset input	<code>reset</code>

- Sets the data type for clock input, clock enable input, and reset ports to `STD_LOGIC` and data input and output ports to VHDL type `STD_LOGIC_VECTOR` or Verilog type `wire`.
- Names coefficients as follows:

For...	Names Coefficients...
FIR filters	<code>coeffn</code> , where n is the coefficient number, starting with 1
IIR filters	<code>coeff_xm_sectionn</code> , where x is <code>a</code> or <code>b</code> , m is the coefficient number, and n is the section number

- When declaring signals of type `REAL`, initializes the signal with a value of 0.0.
- Places VHDL configurations in files that instantiate a component.
- Appends `_rsvd` to names that are VHDL or Verilog reserved words.
- Uses a type-safe representation when concatenating zeros: `'0' & '0'...`
- Applies the statement `IF clock'event AND clock='1' THEN` to check for clock events.
- Allows scale values to be up to 3 bits smaller than filter input values.
- Adds an extra input register and an extra output register to the filter.
- Appends `_process` to process names.
- When creating labels for VHDL `GENERATE` statements:
 - Appends `_gen` to section and block names.
 - Names output assignment blocks with the string `outputgen`

Default Settings for Code Optimizations

- Generates HDL code that is bit-true to the original filter function and is *not* optimized for performance or space requirements.
- Applies a linear final summation to FIR filters. This form of summation is explained in most DSP text books.
- Enables multiplier operations for a filter, as opposed to replacing them with additions of partial products.

`generatetb(Hd 'TbType', 'PropertyName', 'PropertyValue', ...)` generates an HDL test bench of a specified type to verify the HDL code generated for the quantized

filter identified by *Hd*, using the specified property name and property value pair settings. You can specify the function with one or more of the property name and property value pairs described in the Name-Value Pair Arguments as described in the “Input Arguments” for function `generatehdl`.

`generatetb(Hd)` generates an HDL test bench to verify the HDL code generated for the quantized filter identified by *Hd*. The `'TbType'` argument is omitted, and the test bench type defaults to the current setting of the `TargetLanguage` property (`'VHDL'` or `'Verilog'`).

`generatetb(Hd, 'PropertyName', 'PropertyValue', ...)` generates an HDL test bench to verify the HDL code generated for the quantized filter identified by *Hd*, using the specified property name and property value pair settings. The `'TbType'` argument is omitted, and the test bench type defaults to the current setting of the `TargetLanguage` property (`'VHDL'` or `'Verilog'`).

You can specify the function with one or more of the property name and property value pairs described in the Name-Value Pair Arguments as described in the “Input Arguments” for function `generatehdl`.

Examples

Generate HDL Code and Test Bench for Lowpass Filter

```
1 d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2, 0.22, 1, 60)
```

Designs a minimum order lowpass filter with the following characteristics:

- Normalized passband frequency of 0.2
- Stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

```
2 Hd = design(d, 'equiripple')
```

Creates a FIR equiripple filter using the filter data in *d*.

```
3 % Quantized filter with default settings
Hd.arithmetic = 'fixed'
```

Sets the filter arithmetic to fixed-point arithmetic.

```
4 generatehdl(Hd, 'Name', 'MyFilter')
```

Generates VHDL code for the filter Hd. The function names the file `MyFilter.vhd` and places it in the default target folder `hdlsrc`.

5 `generatetb(Hd, 'VHDL', 'TestBenchName', 'MyFilterTB')`

Generates a VHDL test bench for the filter Hd named `MyFilterTB.vhd` and places the generated test bench file in the default target folder `hdlsrc`.

More About

- `generatetbstimulus`
- `generatehdl`

generatetbstimulus

Generate and return HDL test bench stimulus

Syntax

```
generatetbstimulus(Hd)  
generatetbstimulus(Hd, 'PropertyName', 'PropertyValue'...)  
x = generatetbstimulus(Hd, 'PropertyName', 'PropertyValue'...)
```

Description

`generatetbstimulus(Hd)` generates and returns filter input stimulus for the filter `Hd`, where `Hd` is a handle to a filter object. The stimulus is generated based on the setting of the properties `TestBenchStimulus` and `TestBenchUserStimulus`.

The generated test bench applies this input stimuli, and the coder assigns the default stimuli based on the filter type. The following choices of stimuli are available:

- 'impulse'
- 'step'
- 'ramp'
- 'chirp'
- 'noise'

Note: The function quantizes the results by applying the reference coefficients of the specified quantized filter.

`generatetbstimulus(Hd, 'PropertyName', 'PropertyValue'...)` generates and returns filter input stimuli for the filter `Hd`. This value is based on specified settings for `TestbenchRateStimulus` and `TestBenchUserStimulus`.

`x = generatetbstimulus(Hd, 'PropertyName', 'PropertyValue'...)` generates and returns filter input stimuli for the filter `Hd` based on specified settings

for `TestbenchRateStimulus` and `TestBenchUserStimulus` and writes the output to `x` for future use or reference.

Examples

Generate Test Bench Stimulus for FIR Filter

This example shows how to generate ramp and chirp stimulus for a FIR filter.

Generate a lowpass filter and construct a direct-form FIR filter object, `Hd`.

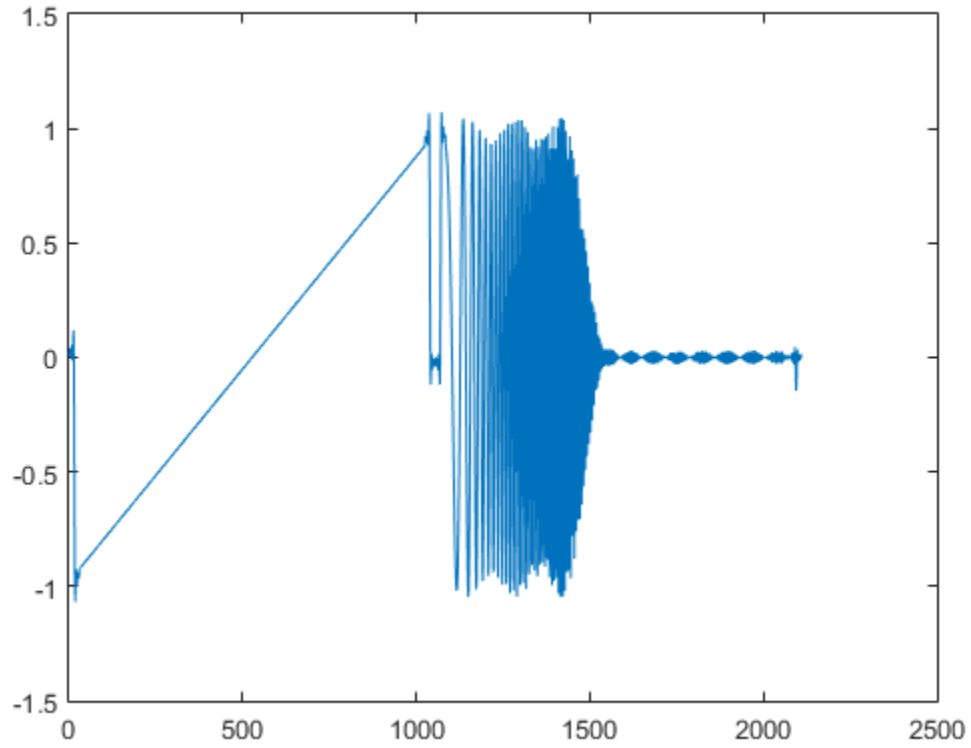
```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');  
Hd = design(filtdes,'filterstructure','dffir');
```

Generate and return test bench stimuli. The call to `generatetbstimulus` in the following command line sequence generates ramp and chirp stimuli and returns the results to `y`.

```
y = generatetbstimulus(Hd, 'TestBenchStimulus', {'ramp', 'chirp'});
```

Apply a quantized filter to the data and plot the results. The call to the `filter` function applies the quantized filter `Hd` to the data that was returned to `y` and gains access to state and filtering information. The `plot` function then plots the resulting data.

```
plot(filter(Hd,y));
```



See Also
generatetb

hdlfilterdainfo

Distributed arithmetic information for filter object

Syntax

```
hdlfilterdainfo (Hd)
hdlfilterdainfo (Hd, 'FoldingFactor', ff)
hdlfilterdainfo (Hd, 'DARadix', dr)
hdlfilterdainfo (Hd, 'LUTInputs', lutip )
hdlfilterdainfo (Hd, 'DALUTPartition', dp )
[dp, dr, lutsizes, ff] = hdlfilterdainfo(Hd)
```

Description

`hdlfilterdainfo` is an informational function that helps you define optimal distributed arithmetic (DA) settings for a filter. For general information on distributed arithmetic architectures, see “Distributed Arithmetic for FIR Filters” on page 4-24.

`hdlfilterdainfo (Hd)`: For the filter object, *Hd*, displays an exhaustive table of 'DARadix' values for the filter *Hd* with the corresponding folding factor and number of LUT sets. This option also displays a table of 'DALUTPartition' values with corresponding LUT inputs (maximum address width) and details of the LUT sets.

`hdlfilterdainfo (Hd, 'FoldingFactor', ff)`: For the filter object, *Hd*, displays a table of LUT input values with total sizes and details of the LUT for the folding factor, *ff*. *ff* must be a nonzero positive integer, or `inf` (to indicate the maximum).

`hdlfilterdainfo (Hd, 'DARadix', dr)`: for the filter object, *Hd*, displays a table of LUT input values with total sizes and details of LUT for the *DARadix* value *dr*.

`hdlfilterdainfo (Hd, 'LUTInputs', lutip)`: For the filter object, *Hd*, displays a table of folding factor values for LUT inputs (maximum address width) of *lutip*. This option also displays total LUT details for each value of folding factor.

`hdlfilterdainfo (Hd, 'DALUTPartition', dp)`: For the filter object, *Hd*, displays a table of folding factor values for *DALUTPartition* of *dp*. This option also displays the total LUT details for each value of folding factor.

`[dp, dr, lutsizes, ff] = hdlfilterdainfo(Hd)`: For the filter object, *Hd*, returns a list of output values to a cell array.

Input Arguments

Hd

A filter object. See “Distributed Arithmetic for FIR Filters” on page 4-24. for a summary of filter types that support distributed arithmetic.

Parameter Name/Value Pairs

The following parameter name/value inputs are optional. These parameters do not have a default value; you must supply a valid value.

'FoldingFactor'

Hardware folding factor, an integer greater than 1 (or `inf`). If the folding factor is `inf`, the coder uses the maximum folding factor. Given the folding factor, the coder the corresponding LUT inputs.

Default: None

'DARadix'

Desired DA Radix value. Given the DA radix, of multipliers, the coder computes a table of values of LUT inputs and sizes.

Default: None

'LUTInputs'

Displays an exhaustive table of values of folding factor corresponding to LUT inputs.

Default: None

'DALUTPartition'

Displays an exhaustive table of values of folding factor for corresponding LUT partition.

Default: None

Output Arguments

[*dp,dr,lutsizes,ff*]

Cell array. `hdlfilterdainfo` returns, in order, the DA LUT partition, *dp*, DA radix, *dr*, folding factor, *fold*, and LUT size, *lutsizes*.

Example

The following example constructs a direct-form FIR filter object, passes it to `hdlfilterdainfo`. The results then display at the MATLAB command line.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
hdlfilterdainfo(Hd)
```

```
| Total Coefficients | Zeros | Effective |
-----|-----|-----|
|           9       |    0  |     9    |
```

Effective filter length for SerialPartition value is 9.

Table of 'DARadix' values with corresponding values of folding factor and multiple for LUT sets for the given filter.

```
| Folding Factor | LUT-Sets Multiple | DARadix |
-----|-----|-----|
|         1     |          16       |    2^16  |
|         2     |           8       |    2^8   |
|         4     |           4       |    2^4   |
|         8     |           2       |    2^2   |
|        16     |           1       |    2^1   |
```

Details of LUTs with corresponding 'DALUTPartition' values.

```
| Max Address | Size(bits) | LUT Details | DALUTPartition |
-----|-----|-----|-----|
|         9   |      9216  | 1x512x18   | [9]            |
|         8   |      4628  | 1x256x18, 1x2x10 | [8 1]         |
|         7   |      2352  | 1x128x18, 1x4x12 | [7 2]         |
|         6   |      1192  | 1x64x17, 1x8x13 | [6 3]         |
|         5   |       800  | 1x16x16, 1x32x17 | [5 4]         |
|         4   |       548  | 1x16x16, 1x16x17, 1x2x10 | [4 4 1]       |
|         3   |       344  | 2x8x13, 1x8x17  | [3 3 3]       |
|         2   |       252  | 1x2x10, 1x4x12, 1x4x13, 1x4x16, 1x4x17 | [2 2 2 2 1]  |
```

Notes:

1. LUT Details indicates number of LUTs with their sizes. e.g. 1x1024x18

implies 1 LUT of 1024 18-bit wide locations.

More About

- “Distributed Arithmetic for FIR Filters” on page 4-24

hdlfilterserialinfo

Serial partition information for filter object

Syntax

```
hdlfilterserialinfo (Hd)
hdlfilterserialinfo (Hd,'FoldingFactor', ff)
hdlfilterserialinfo (Hd,'Multipliers', nmults)
hdlfilterserialinfo (Hd,'SerialPartition', [p1 p2...pN])
[sp, fold, nm] = hdlfilterserialinfo(Hd)
[sp, fold, nm] = hdlfilterserialinfo(Hd,'FoldingFactor', ff)
[sp, fold, nm] = hdlfilterserialinfo(Hd,'Multipliers', nmults)
[sp, fold, nm] = hdlfilterserialinfo(Hd,'SerialPartition', [p1
p2...pN])
```

Description

`hdlfilterserialinfo` is an informational function that helps you define an optimal serial partition for a filter. `hdlfilterserialinfo` provides answers to design questions such as:

- What is the folding factor and associated number of multipliers that results from a particular choice of serial partition?
- What are the valid serial partitions for a given filter?

When you specify a serial architecture for a filter, you can define the serial partitioning in the following ways:

- Directly specify a vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition.
- Specify the desired hardware folding factor, ff , an integer greater than 1. Given the folding factor, the coder computes the serial partition and the number of multipliers.
- Specify the desired number of multipliers, $nmults$, an integer greater than 1. Given the number of multipliers, the coder computes the serial partition and the folding factor.

By default, `hdfilterserialinfo` displays a table of serial partition vector options for a filter object, *Hd*, with corresponding values of folding factor and number of multipliers.

Optionally, you can pass in a parameter/value pair specifying a serial partition vector, a folding factor, or number of multipliers. In this case, `hdfilterserialinfo` displays the corresponding values for the other parameters.

You can also optionally specify that `hdfilterserialinfo` returns serial partition values and corresponding values of folding factor and number of multipliers to a cell array.

`hdfilterserialinfo (Hd)`: For the filter object, *Hd*, displays a table of serial partition values with corresponding values of folding factor and number of multipliers.

`hdfilterserialinfo (Hd, 'FoldingFactor', ff)`: For the filter object, *Hd*, displays the serial partition values and number of multipliers corresponding the folding factor, *ff*.

`hdfilterserialinfo (Hd, 'Multipliers', nmults)`: for the filter object, *Hd*, displays the serial partition values and folding factor corresponding to the number of multipliers, *nmults*.

`hdfilterserialinfo (Hd, 'SerialPartition', [p1 p2...pN])`: For the filter object, *Hd*, displays the folding factor and number of multipliers corresponding to the serial partition vector [*p1p2...pN*].

`[sp, fold, nm] = hdfilterserialinfo(Hd)`: For the filter object, *Hd*, returns a table of serial partition values with corresponding values of folding factor and number of multipliers to a cell array.

`[sp, fold, nm] = hdfilterserialinfo(Hd, 'FoldingFactor', ff)`: For the filter object, *Hd*, returns the serial partition values and number of multipliers corresponding to the folding factor, *ff*, to a cell array.

`[sp, fold, nm] = hdfilterserialinfo(Hd, 'Multipliers', nmults)`: For the filter object, *Hd*, displays the serial partition values and folding factor corresponding to the number of multipliers, *nmults*, to a cell array.

`[sp, fold, nm] = hdfilterserialinfo(Hd, 'SerialPartition', [p1 p2...pN])`: For the filter object, *Hd*, returns the folding factor and number of multipliers corresponding to the serial partition vector [*p1p2...pN*] to a cell array.

Input Arguments

Hd

A filter object. See “Speed vs. Area Tradeoffs” on page 4-2 for a summary of filter types that support serial architectures.

Parameter Name/Value Pairs

The following parameter name/value inputs are optional. These parameters do not have a default value; you must supply a valid value.

'FoldingFactor'

Hardware folding factor, an integer greater than 1 (or `inf`). If the folding factor is `inf`, the coder uses the maximum folding factor. Given the folding factor, the coder computes the serial partition and the number of multipliers.

Default: None

'Multipliers'

Desired number of multipliers, an integer greater than 1 (or `inf`). If the number of multipliers is `inf`, the coder uses the maximum number of multipliers. Given the number of multipliers, the coder computes the serial partition and the folding factor.

Default: None

'SerialPartition'

A vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition.

Default: None

Output Arguments

[sp, fold, nm]

Cell array. `hdlfilterserialinfo` returns, in order, the serial partition, sp , folding factor, $fold$, and number of multipliers, nm .

Examples

Each of the following examples constructs a direct-form FIR filter object, passes it to `hdlfilterserialinfo`, and shows the results as displayed at the MATLAB command line.

Pass only the filter object to `hdlfilterserialinfo` to display the valid serial partitions:

```
Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
hdlfilterserialinfo(Hd)
```

Table of 'SerialPartition' values with corresponding values of folding factor and number of multipliers for the given filter.

Folding Factor	Multipliers	SerialPartition
1	9	[[1 1 1 1 1 1 1 1 1]]
2	5	[[2 2 2 2 1]]
3	3	[[3 3 3]]
4	3	[[4 4 1]]
5	2	[[5 4]]
6	2	[[6 3]]
7	2	[[7 2]]
8	2	[[8 1]]
9	1	[[9]]

Pass the filter object to `hdlfilterserialinfo`, specifying the desired number of multipliers as 3:

```
Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
hdlfilterserialinfo(Hd, 'Multipliers', 3)
```

SerialPartition: [3 3 3], Folding Factor: 3, Multipliers: 3

Pass the filter object to `hdlfilterserialinfo`, specifying the desired folding factor as 4:

```
Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
hdlfilterserialinfo(Hd, 'FoldingFactor', 4)
```

SerialPartition: [4 4 1], Folding Factor: 4, Multipliers: 3

Pass only the filter object to `hdlfilterserialinfo`, returning the results to a cell array:

```
Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
[sp,ff,nm] = hdlfilterserialinfo(Hd)
```

sp =

```

    '1 1 1 1 1 1 1 1 1]'
    '[2 2 2 2 1]'
    '[3 3 3]'
    '[4 4 1]'
    '[5 4]'
    '[6 3]'
    '[7 2]'
    '[8 1]'
    '[9]'
ff =
    '1'
    '2'
    '3'
    '4'
    '5'
    '6'
    '7'
    '8'
    '9'
nm =
    '1'
    '2'
    '3'
    '5'
    '9'

```

Pass the filter object to `hdlfilterserialinfo`, specifying the desired folding factor as 4, returning the results to a cell array:

```

Hd = design(fdesign.lowpass('N,Fc',8,.4));
Hd.Arithmetic = 'fixed';
[sp,ff,nm] = hdlfilterserialinfo(Hd, 'FoldingFactor', 4)
sp =
    4    4    1
ff =
    4
nm =
    3

```

More About

- “Speed vs. Area Tradeoffs” on page 4-2