

# Filter Design HDL Coder

For Use with **MATLAB**®

- Computation
- Visualization
- Programming

User's Guide

*Version 1*



## How to Contact The MathWorks:



www.mathworks.com      Web  
comp.soft-sys.matlab      Newsgroup



support@mathworks.com      Technical Support  
suggest@mathworks.com      Product enhancement suggestions  
bugs@mathworks.com      Bug reports  
doc@mathworks.com      Documentation error reports  
service@mathworks.com      Order status, license renewals, passcodes  
info@mathworks.com      Sales, pricing, and general information



508-647-7000      Phone



508-647-7001      Fax



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

For contact information about worldwide offices, see the MathWorks Web site.

### *Filter Design HDL Coder User's Guide*

© COPYRIGHT 2004 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.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:

June 2004

Online only

New for Version 1.0 (Release 14)

## Getting Started

### 1

<b>What Is the Filter Design HDL Coder?</b> .....	<b>1-2</b>
Expected Users .....	<b>1-3</b>
Key Features and Components .....	<b>1-3</b>
FDATool Plug-In — the GUI .....	<b>1-4</b>
Command-Line Interface .....	<b>1-5</b>
Quantized Filters — the Input .....	<b>1-6</b>
Filter Properties — Input Parameters .....	<b>1-7</b>
Generated HDL Files — the Output .....	<b>1-8</b>
<b>Installation</b> .....	<b>1-9</b>
Checking Product Requirements .....	<b>1-9</b>
Installing the Software .....	<b>1-9</b>
<b>Getting Help with the Filter Design HDL</b>	
<b>Coder</b> .....	<b>1-10</b>
Information Overview .....	<b>1-10</b>
Online Help .....	<b>1-11</b>
Using “What’s This?” Context-Sensitive Help .....	<b>1-11</b>
Demos and Tutorials .....	<b>1-12</b>
<b>Applying the Filter Design HDL Coder to the Hardware</b>	
<b>Design Process</b> .....	<b>1-13</b>

## Tutorials — Generating HDL Code for Filters

### 2

<b>Creating a Directory for Your Tutorial Files</b> .....	<b>2-2</b>
<b>Basic FIR Filter Tutorial</b> .....	<b>2-3</b>
Designing a Basic FIR Filter .....	<b>2-3</b>

Quantizing the Basic FIR Filter .....	2-5
Configuring and Generating the Basic FIR Filter's VHDL Code .....	2-8
Getting Familiar with the Basic FIR Filter's Generated VHDL Code .....	2-15
Verifying the Basic FIR Filter's Generated VHDL Code .....	2-16
<b>Optimized FIR Filter Tutorial .....</b>	<b>2-23</b>
Designing the FIR Filter .....	2-23
Quantizing the FIR Filter .....	2-25
Configuring and Generating the FIR Filter's Optimized Verilog Code .....	2-28
Getting Familiar with the FIR Filter's Optimized Generated Verilog Code .....	2-35
Verifying the FIR Filter's Optimized Generated Verilog Code .....	2-37
<b>IIR Filter Tutorial .....</b>	<b>2-43</b>
Designing an IIR Filter .....	2-43
Quantizing the IIR Filter .....	2-45
Configuring and Generating the IIR Filter's VHDL Code .....	2-49
Getting Familiar with the IIR Filter's Generated VHDL Code .....	2-55
Verifying the IIR Filter's Generated VHDL Code .....	2-56

## Generating HDL Code for a Filter Design

### 3

<b>Overview of Generating HDL Code for a Filter Design .....</b>	<b>3-3</b>
<b>Opening the Generate HDL Dialog .....</b>	<b>3-4</b>
<b>What Is Generated by Default? .....</b>	<b>3-9</b>
Default Settings for Generated Files .....	3-9
Default Settings for Register Resets .....	3-10

Default Settings for General HDL Code .....	3-10
Default Settings for Code Optimizations .....	3-11
Default Settings for Test Benches .....	3-12
<b>What Are Your HDL Requirements?</b> .....	<b>3-13</b>
<b>Setting the Target Language</b> .....	<b>3-18</b>
<b>Setting the Names and Location for Generated HDL</b>	
<b>Files</b> .....	<b>3-19</b>
Setting Filter Entity and General File Naming	
Strings .....	<b>3-20</b>
Redirecting Filter Design HDL Coder Output .....	<b>3-21</b>
Setting the Postfix String for VHDL Package	
Files .....	<b>3-22</b>
Splitting Entity and Architecture Code into Separate	
Files .....	<b>3-23</b>
<b>Customizing Reset Specifications</b> .....	<b>3-26</b>
Setting the Reset Style for Registers .....	<b>3-26</b>
Setting the Asserted Level for the Reset Input	
Signal .....	<b>3-28</b>
<b>Customizing the HDL Code</b> .....	<b>3-29</b>
Specifying a Header Comment .....	<b>3-30</b>
Specifying a Prefix for Filter Coefficients .....	<b>3-32</b>
Setting the Postfix String for Resolving Entity or Module Name	
Conflicts .....	<b>3-33</b>
Setting the Postfix String for Resolving HDL Reserved Word	
Conflicts .....	<b>3-34</b>
Setting the Postfix String for Process Block	
Labels .....	<b>3-37</b>
Naming HDL Ports .....	<b>3-38</b>
Specifying the HDL Data Type for Data Ports .....	<b>3-40</b>
Suppressing Extra Input and Output Registers .....	<b>3-42</b>
Minimizing Quantization Noise for Fixed-Point	
Filters .....	<b>3-43</b>
Representing Constants with Aggregates .....	<b>3-45</b>
Unrolling and Removing VHDL Loops .....	<b>3-46</b>
Using the VHDL rising_edge Function .....	<b>3-47</b>
Suppressing the Generation of VHDL Inline	
Configurations .....	<b>3-48</b>

Specifying VHDL Syntax for Concatenated Zeros .....	3-49
Suppressing Verilog Time Scale Directives .....	3-50
Suppressing the Initialization of Signals of Type REAL .....	3-51
Specifying Input Type Treatment for Addition and Subtraction Operations .....	3-52
<b>Setting Optimizations .....</b>	<b>3-54</b>
Optimizing Generated Code for HDL .....	3-55
Optimizing Coefficient Multipliers .....	3-55
Optimizing Final Summation for FIR Filters .....	3-57
Optimizing the Clock Rate with Pipeline Registers .....	3-58
Setting Optimizations for Synthesis .....	3-59
<b>Customizing the Test Bench .....</b>	<b>3-61</b>
Renaming the Test Bench .....	3-61
Specifying a Test Bench Type .....	3-62
Configuring the Clock .....	3-65
Configuring Resets .....	3-67
Setting a Hold Time for Data Input Signals .....	3-69
Setting an Error Margin for Optimized Filter Code .....	3-70
Setting Test Bench Stimuli .....	3-72
<b>Generating the HDL Code .....</b>	<b>3-74</b>

## Testing a Filter Design

# 4

<b>Overview of the Test Methods .....</b>	<b>4-2</b>
<b>Testing with an HDL Test Bench .....</b>	<b>4-3</b>
Generating the Filter and Test Bench HDL Code .....	4-3
Starting the Simulator .....	4-7
Compiling the Generated Filter and Test Bench Files .....	4-7
Running the Test Bench Simulation .....	4-8

<b>Testing with a ModelSim Tcl/Tk DO File</b> .....	<b>4-12</b>
Generating the Filter HDL Code and Test Bench DO File .....	<b>4-12</b>
Starting ModelSim .....	<b>4-15</b>
Compiling the Generated Filter File .....	<b>4-16</b>
Execute the ModelSim DO File .....	<b>4-17</b>

## Properties — Categorical List

# 5

<b>Language Selection Properties</b> .....	<b>5-2</b>
<b>File Naming and Location Properties</b> .....	<b>5-2</b>
<b>Reset Properties</b> .....	<b>5-2</b>
<b>Header Comment and General Naming Properties</b> .....	<b>5-3</b>
<b>Port Properties</b> .....	<b>5-4</b>
<b>Advanced Coding Properties</b> .....	<b>5-4</b>
<b>Optimization Properties</b> .....	<b>5-6</b>
<b>Test Bench Properties</b> .....	<b>5-6</b>

## Properties — Alphabetical List

6

## Functions — Alphabetical List

7

## Examples

A

<b>Tutorials</b> .....	<b>A-2</b>
<b>Basic FIR Filter Tutorial</b> .....	<b>A-3</b>
<b>Optimized FIR Filter Tutorial</b> .....	<b>A-4</b>
<b>IIR Filter Tutorial</b> .....	<b>A-5</b>

## Index



# Getting Started

---

This chapter introduces you to the Filter Design HDL Coder by discussing the following topics:

“What Is the Filter Design HDL Coder?” (p. 1-2)	Describes key product features and components
“Installation” (p. 1-9)	Explains how to install and set up the Filter Design HDL Coder
“Getting Help with the Filter Design HDL Coder” (p. 1-10)	Discusses ways of applying the Filter Design HDL Coder to the hardware design process, including signal analysis, algorithm verification, and reference design validation
“Applying the Filter Design HDL Coder to the Hardware Design Process” (p. 1-13)	Identifies and explains how to gain access to available documentation and online help resources

## What Is the Filter Design HDL Coder?

The Filter Design HDL Coder 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 by generating hardware description language (HDL) code based on filters developed in MATLAB®. Currently, system designers and hardware developers use HDLs, such as very high speed integrated circuit (VHSIC) hardware definition 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.

Using the Filter Design HDL Coder, system architects and designers 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 the Filter Design Toolbox, its Filter Design and Analysis Tool (FDATool), and the Filter Design HDL Coder to design a filter. Then, with the click of a button, the Filter Design HDL Coder generates 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, as needed. The test bench feature increases confidence in the correctness of the generated code and saves potential time spent on test bench implementation.

The following sections discuss

- “Expected Users” on page 1-3
- “Key Features and Components” on page 1-3
- “FDATool Plug-In — the GUI ” on page 1-4
- “Command-Line Interface” on page 1-5
- “Quantized Filters — the Input” on page 1-6

- “Filter Properties — Input Parameters” on page 1-7
- “Generated HDL Files — the Output” on page 1-8

## Expected Users

Filter Design HDL Coder users are 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 also expected to have prerequisite knowledge in the following subject areas:

- Hardware design and system integration
- VHDL or Verilog
- MATLAB
- Filter Design Toolbox
- HDL simulators, such as ModelSim®

## Key Features and Components

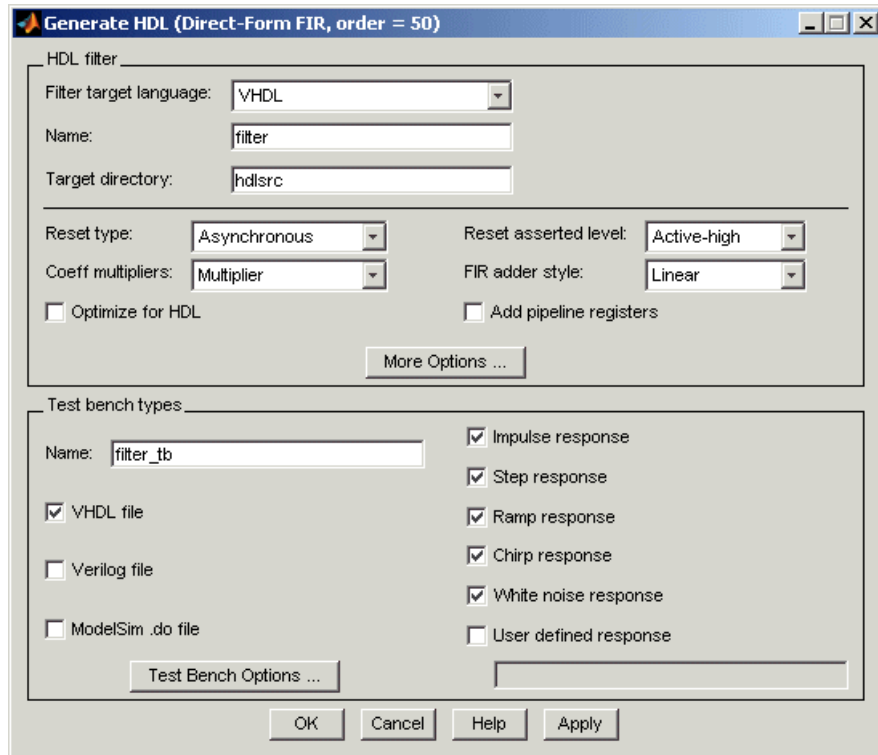
Key features and components of Filter Design HDL Coder include

- Graphical user interface (GUI) plug-in to the Filter Design and Analysis Tool (FDATool)
- MATLAB command line interface
- Support for the following filter structures:
  - Finite impulse response (FIR)
  - Antisymmetric FIR
  - Transposed FIR
  - Symmetric FIR

- Second-order section (SOS) infinite impulse response (IIR) Direct Form I
- SOS IIR Direct Form I transposed
- SOS IIR Direct Form II
- SOS IIR Direct Form II transposed
- Generation of code that adheres to a clean HDL coding style
- Options for optimizing numeric results of generated HDL code
- Options for controlling the contents and style of the generated HDL code and test bench
- Test bench generation for validating the generated HDL filter code
- VHDL, Verilog, and ModelSim Tcl/Tk DO file test bench options

## **FDATool Plug-In – the GUI**

The Filter Design HDL Coder graphical user interface (GUI) is a plug-in component of the FDATool and is accessible from the FDATool **Targets** menu. Given that you have designed, or at least opened, a quantized filter in the FDATool, you can generate HDL code for that filter with the **Generate HDL** dialog. To open this dialog, click **Targets→Generate HDL**. The main dialog appears, showing the title **Generate HDL** and the filter's structure and order. The following dialog indicates that the input is a Direct Form II transposed filter with an order of 50.



Chapter 3, “Generating HDL Code for a Filter Design” explains how to use the GUI to customize HDL code generation to meet project-specific requirements.

## Command-Line Interface

You also have the option of generating HDL code for a filter with the Filter Design HDL Coder command-line interface. You can apply functions interactively at the MATLAB command line or programmatically in an M-file. The following table lists available functions with brief descriptions. For more detail, see Chapter 7, “Functions — Alphabetical List”.

<b>Function</b>	<b>Purpose</b>
<code>generatehdl</code>	Generate HDL code for quantized filter
<code>generatetb</code>	Generate test bench for quantized filter
<code>generatetbstimulus</code>	Generate and return test bench stimuli

## **Quantized Filters – the Input**

The input to the Filter Design HDL Coder is a quantized filter that you design and quantize in one of two ways:

- Design and quantize the filter with the Filter Design Toolbox
- Design the filter with the Signal Processing Toolbox and then quantize it with the Filter Design Toolbox

The Filter Design HDL Coder supports the following filter structures:

- Finite impulse response (FIR)
- Antisymmetric FIR
- Transposed FIR
- Symmetric FIR
- Second-order section (SOS) infinite impulse response (IIR) Direct Form I
- SOS IIR Direct Form I transposed
- SOS IIR Direct Form II
- SOS IIR Direct Form II transposed

Each of these structures supports fixed-point, quantization type, and floating-point (double) realizations. The FIR structures also support unsigned fixed-point realizations.

---

**Note** Filter Design HDL Coder does not support zero order sections for IIR filters.

---

The quantized filter must have the following data format characteristics:

- Fixed-point signed or unsigned
- Double floating-point precision

For information on how to design filter objects, see the Filter Design Toolbox and Signal Processing Toolbox documentation. For information on quantizing filters, see the Filter Design Toolbox documentation.

## **Filter Properties – Input Parameters**

The Filter Design HDL Coder generates filter and test bench HDL code for a specified quantized filter based on the settings of a collection of property name and property value pairs. The properties and their values

- Contribute to the naming of language elements
- Specify port parameters
- Determine the use of advanced HDL coding features

All properties have default settings. However, you can customize the HDL output to meet project specifications by adjusting the property settings with the Filter Design HDL Coder GUI or command line interface. As an FDATool plug-in, the GUI enables you to set properties associated with

- The HDL language specification
- Filename and location specifications
- Reset specifications
- HDL code customizations
- HDL code optimizations
- Test bench customizations

You can set the same filter properties by specifying property name and property value pairs with the functions `generatehdl`, `generatetb`, and `generatetbstimulus` interactively at the MATLAB command line or in M-code.

The property names and property values are *not* case sensitive and, when specifying them, you can abbreviate them to the shortest unique string.

This chapter explains how to apply property settings to customize HDL code generation for a specific application. For lists and descriptions of the properties and functions, see Chapter 5, “Properties — Categorical List” and Chapter 7, “Functions — Alphabetical List”, respectively.

## Generated HDL Files — the Output

Based on the interface you use and the input data you specify, the Filter Design HDL Coder generates filter and filter test bench HDL files as output. If the filter design requires a VHDL package, the Filter Design HDL Coder also generates a package file.

The GUI generates all output files at the end of a dialog session. If you choose to use the command line interface, you generate the filter and test bench HDL files separately with calls to the functions `generatehdl` and `generatetb`.

By default, the Filter Design HDL Coder places the output files in a subdirectory named `hdlsrc`, under the current MATLAB directory, and names the files as follows, where *name* is the value of the Name property.

<b>Language</b>	<b>File</b>	<b>Name</b>
Verilog	Filter	<i>name</i> .v
	Filter test bench	<i>name</i> _tb.v
VHDL	Filter	<i>name</i> .vhd
	Filter test bench	<i>name</i> _tb.vhd
	Filter package (if required)	<i>name</i> _pkg.vhd



# Installation

The following sections discuss installation:

- “Checking Product Requirements” on page 1-9
- “Installing the Software” on page 1-9

## Checking Product Requirements

Filter Design HDL Coder requires the following:

- MATLAB
- Filter Design Toolbox
- Signal Processing Toolbox
- Fixed-Point Toolbox

## Installing the Software

For information on installing MATLAB, the Signal Processing Toolbox, the Filter Design Toolbox, the Filter Design HDL Coder, and optional software, see the MATLAB installation instructions.

## Getting Help with the Filter Design HDL Coder

The following sections explain how to get help with using the Filter Design HDL Coder:

- “Information Overview” on page 1-10
- “Online Help” on page 1-11
- “Using “What’s This?” Context-Sensitive Help” on page 1-11
- “Demos and Tutorials” on page 1-12

### Information Overview

The following information is available with this product:

Chapter 1, “Getting Started”	Explains what the product is, how to install it, how you might apply it to the hardware design process, and how to gain access to product documentation and online help.
Chapter 2, “Tutorials — Generating HDL Code for Filters”	Guides you through the process of generating HDL code for a sampling of filters.
Chapter 3, “Generating HDL Code for a Filter Design”	Explains how to use the Filter Design HDL Coder to generate HDL code for a filter design. Provides details on how HDL code is mapped to MATLAB code and vice versa.
Chapter 4, “Testing a Filter Design”	Explains how to apply generated test benches.
Chapter 5, “Properties — Categorical List”	Lists filter properties by category.

Chapter 6, “Properties — Alphabetical List”	Provides descriptions of properties organized alphabetically by property name.
Chapter 7, “Functions — Alphabetical List”	Provides descriptions of the functions available in the product’s command line interface.

## Online Help

The following online help is available:

- Online help in the MATLAB Help browser. Click the Filter Design HDL Coder product link in the browser’s Contents pane.
- Context-sensitive “What’s This?” help for items that appear in the Filter Design HDL Coder GUI. Click a GUI Help button or right-click on a GUI item or within a specific frame in a GUI dialog to display help on that dialog, item, or frame. For more information on using the context-sensitive help, see “Using “What’s This?” Context-Sensitive Help” on page 1-11.
- M-help for the command line interface functions `generatehdl`, `generatetb`, and `generatetbstimulus` is accessible with the MATLAB `doc` and `help` commands. For example

```
doc generatehdl
help generatehdl
```

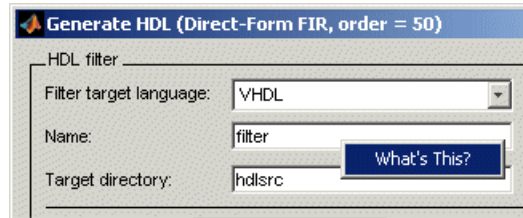
## Using “What’s This?” Context-Sensitive Help

“What’s This?” context-sensitive help topic is available for each dialog, pane, and option in the Filter Design HDL Coder GUI. Use the “What’s This?” help as needed while using the GUI to configure options that control the contents and style of the generated HDL code and test bench.

To use the “What’s This?” help, do the following:

- 1 Place your cursor over the label or control for an option or in the background for a pane or dialog.

- 2 Right-click. A **What's This?** button appears. The following display shows the **What's This?** button appearing after a right-click on the **Name** option in the **HDL filter** pane of the **Generate HDL** dialog.



- 3 Click **What's This?** The Filter Design HDL Coder opens context-sensitive help that describes the option, pane, or dialog.

## Demos and Tutorials

The Filter Design HDL Coder provides demos and tutorials to help you get started. The demos give you a quick view of the product's capabilities and examples of how you might apply the product. You can run them with limited product exposure.

The tutorials provide procedural instruction on how to apply product features. The following topics, in Chapter 2, "Tutorials — Generating HDL Code for Filters", guide you through three tutorials:

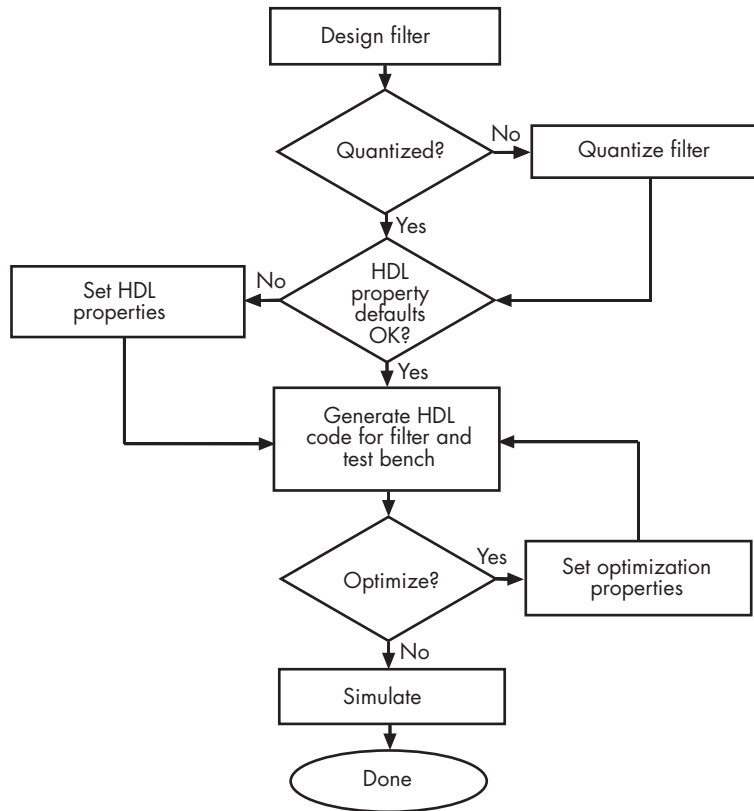
- "Basic FIR Filter Tutorial" on page 2-3
- "Optimized FIR Filter Tutorial" on page 2-23
- "IIR Filter Tutorial" on page 2-43

## Applying the Filter Design HDL Coder to the Hardware Design Process

The basic workflow for applying the Filter Design HDL Coder to the hardware design process involves the following steps:

- 1** Design a filter with the Signal Processing or Filter Design Toolbox.
- 2** Quantize the filter with the Filter Design Toolbox.
- 3** Review the property settings that the Filter Design HDL Coder applies to generated HDL code by default.
- 4** Adjust property settings to customize the generated HDL code, as necessary.
- 5** Generate the filter and test bench code.
- 6** Consider and, if appropriate, apply optimization options.
- 7** Test the generated code in a simulation.

The following figure shows these steps in a flow diagram.



# Tutorials — Generating HDL Code for Filters

---

This chapter guides you through the basic steps for generating and testing HDL code for a few filter designs. Topics include the following:

“Creating a Directory for Your Tutorial Files” (p. 2-2)

Suggests that you create a directory to store files generated as you complete the tutorials presented in this chapter

“Basic FIR Filter Tutorial” (p. 2-3)

Guides you through the steps for designing a basic FIR filter, generating VHDL code for the filter, and verifying the VHDL code with a generated test bench

“Optimized FIR Filter Tutorial” (p. 2-23)

Guides you through the steps for designing an optimized FIR filter, generating Verilog code for the filter, and verifying the Verilog code with a generated test bench

“IIR Filter Tutorial” (p. 2-43)

Guides you through the steps for designing an IIR filter, generating VHDL code for the filter, and verifying the VHDL code with a generated test bench

## Creating a Directory for Your Tutorial Files

Set up a writable working directory outside the scope of your MATLAB installation area to store files that will be generated as you complete your Filter Design HDL Coder tutorial work. The tutorial instructions assume that you create the directory `hdlfilter_tutorials` on drive D.



## Basic FIR Filter Tutorial

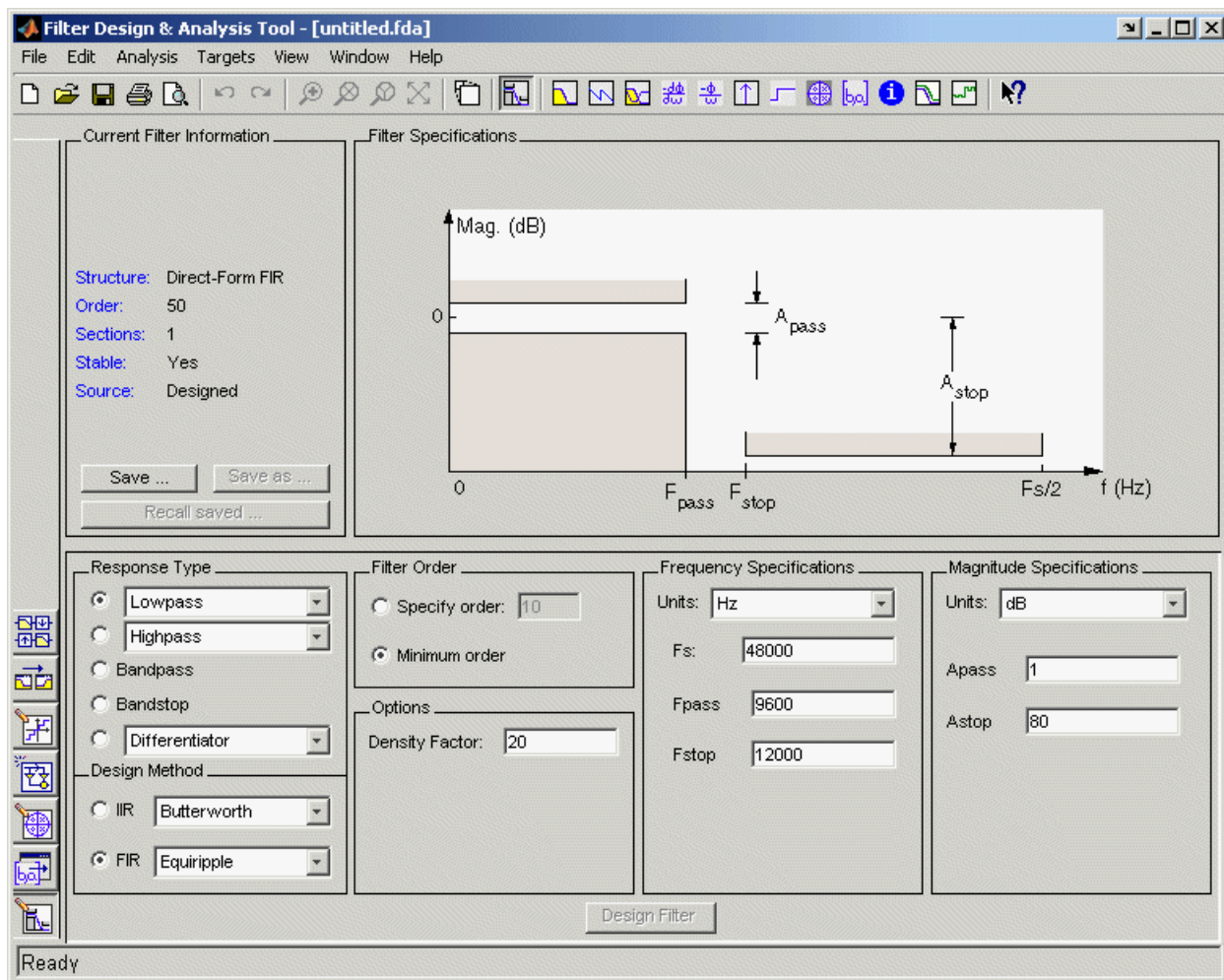
This section guides you through the steps for designing a basic quantized discrete-time FIR filter, generating VHDL code for the filter, and verifying the VHDL code with a generated test bench. The procedure is presented in the following topics:

- “Designing a Basic FIR Filter” on page 2-3
- “Quantizing the Basic FIR Filter” on page 2-5
- “Configuring and Generating the Basic FIR Filter’s VHDL Code” on page 2-8
- “Getting Familiar with the Basic FIR Filter’s Generated VHDL Code” on page 2-15
- “Verifying the Basic FIR Filter’s Generated VHDL Code” on page 2-16

### Designing a Basic FIR Filter

One way of designing a filter in the MATLAB environment is to use the FDATool. This section guides you through the procedure of designing and creating a filter for a basic FIR filter. These instructions assume you are familiar with the MATLAB user interface and the FDATool.

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



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

<b>Option</b>	<b>Value</b>
Response Type	Lowpass
Design Method	FIR Equiripple

<b>Option</b>	<b>Value</b>
Filter Order	Minimum order (50)
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 need to make any changes and **Design Filter** is greyed out, you are done and can skip to “Quantizing the Basic FIR Filter” on page 2-5.

- 5 If you modified any of the 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

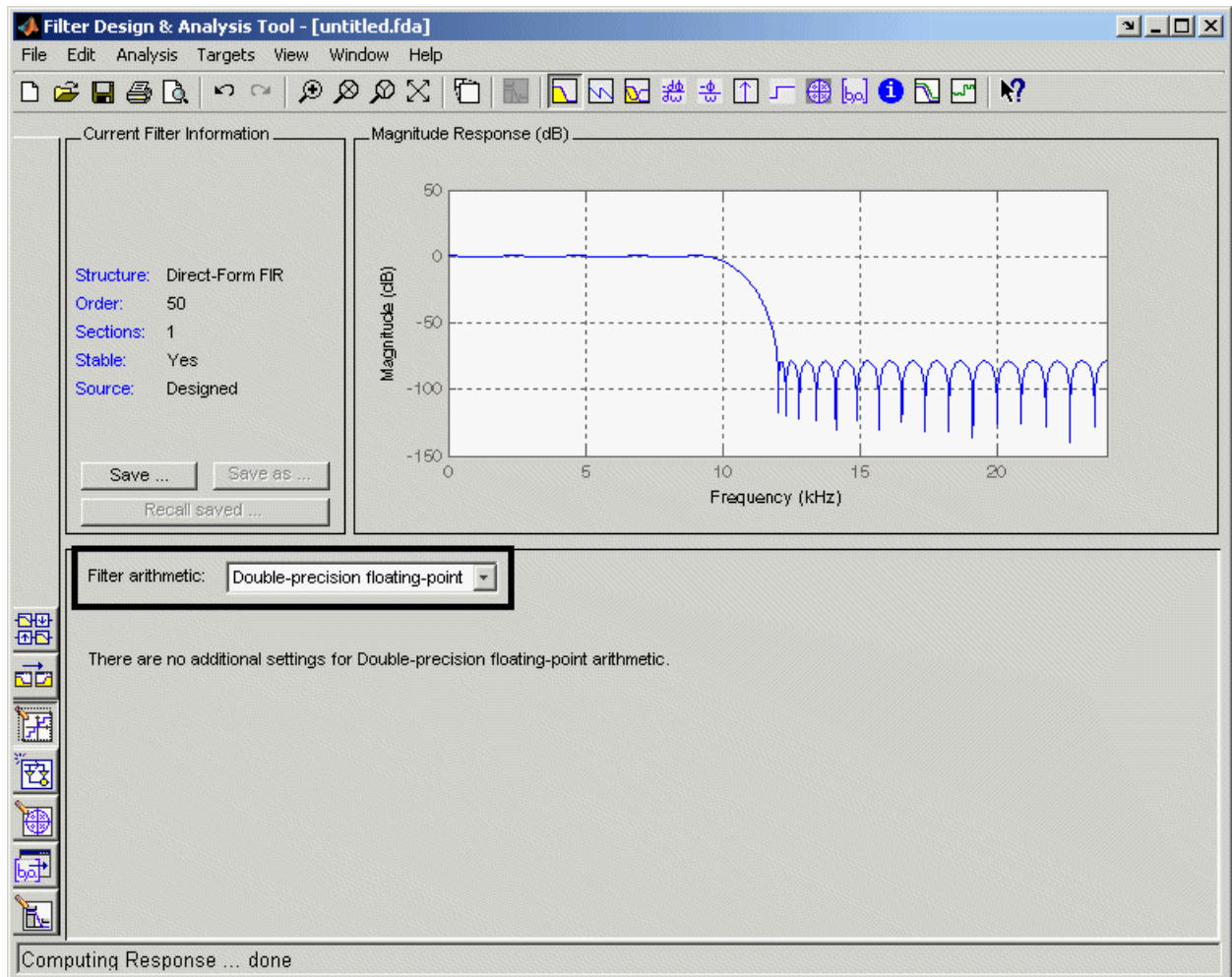
For more information on designing filters with the FDATool, see the FDATool and Filter Design Toolbox documentation.

## Quantizing the Basic FIR Filter

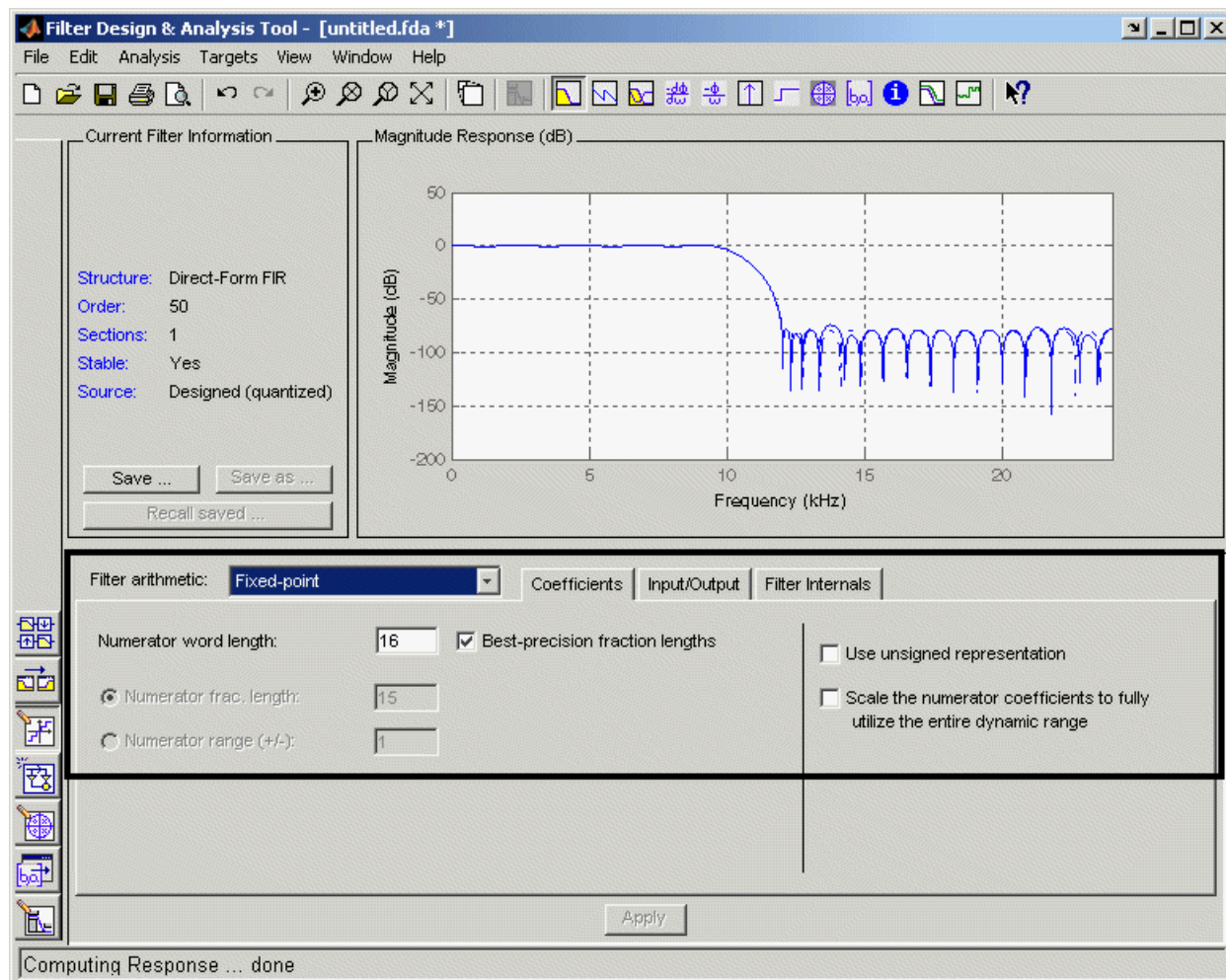
You should quantize filters for HDL code generation. To quantize your filter,

- 1 Open the basic FIR filter design you created in “Designing a Basic FIR Filter” on page 2-3 if it is not already open.

- 2 Click the **Set quantization parameters** icon  in the left-side tool bar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog.



- 3 Select Fixed-point from the **Filter arithmetic** menu. The FDATool displays the first of three tabbed panels of quantization parameters across the bottom half of its dialog.



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:

<b>Tab</b>	<b>Parameter</b>	<b>Setting</b>
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
	Avoid overflow	Selected
Filter Internals	Round towards	Floor
	Overflow mode	Saturate
	Product mode	Full precision
	Accum. mode	Keep MSB
	Accum. word length	40
	Cast signals before accum.	Selected

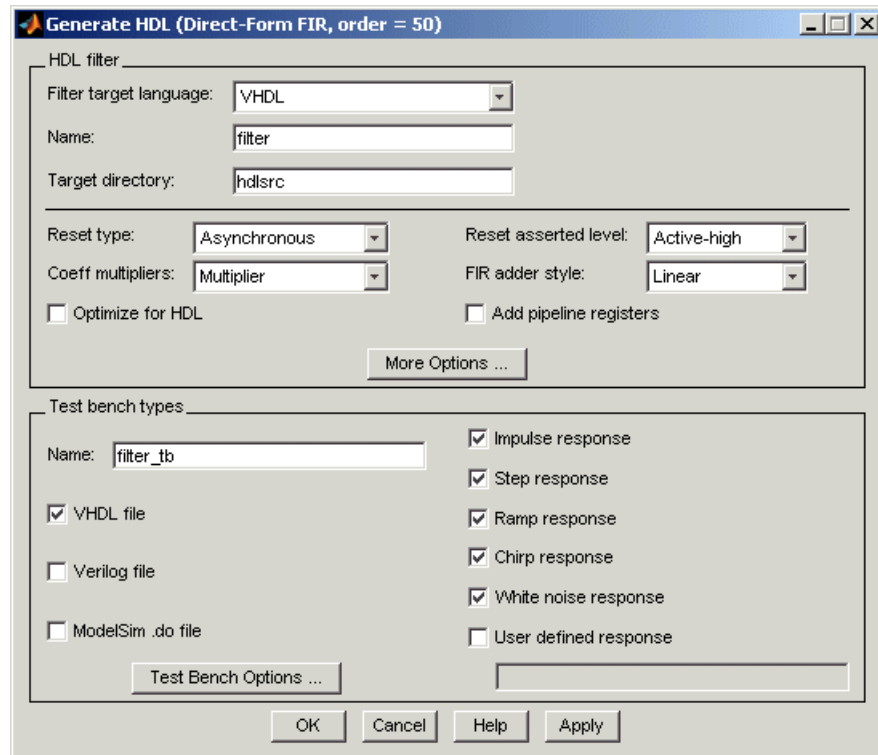
**5** Click **Apply**.

For more information on quantizing filters, see the FDATool and Filter Design Toolbox documentation.


## **Configuring and Generating the Basic FIR Filter’s VHDL Code**

After you quantize your filter, you are ready to use the Filter Design HDL Coder to configure 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” on page 2-3 and “Quantizing the Basic FIR Filter” on page 2-5.

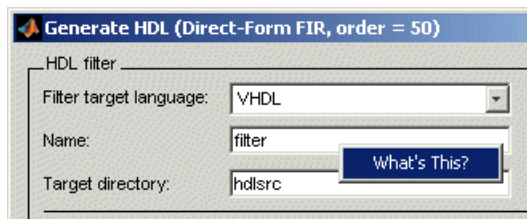
- 1 Start the Filter Design HDL Coder by clicking **Targets**→**Generate HDL** in the FDATool dialog. The FDATool displays the Filter Design HDL Coder dialog.



- 2 Find the Filter Design HDL Coder online help. Use the help to learn about product details or to get answers to questions as you work with the designer.

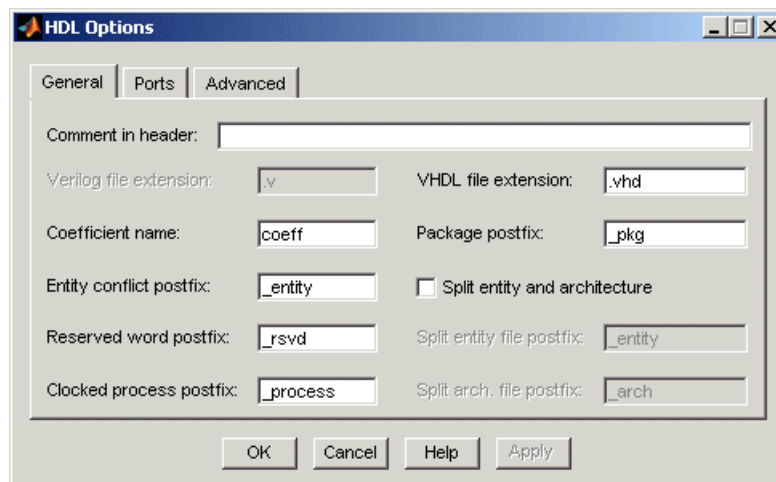
- a In the MATLAB window, click the **Help icon**  in the toolbar or click **Help**→**Full Product Family Help**.
- b In the Help browser's **Contents** pane, select **Filter Design HDL Coder**.
- c Minimize the Help browser.

- 3 Click the **Help** button. The FDATool displays context-sensitive help for the dialog. As necessary, use the **Help** button on the other Filter Design HDL Coder dialogs for context-sensitive help on those dialog views.
- 4 Close the Help window.
- 5 Place your cursor over the **Name** label or text box in the **HDL filter** pane of the **Generate HDL** dialog and right-click. A **What's This?** button appears.



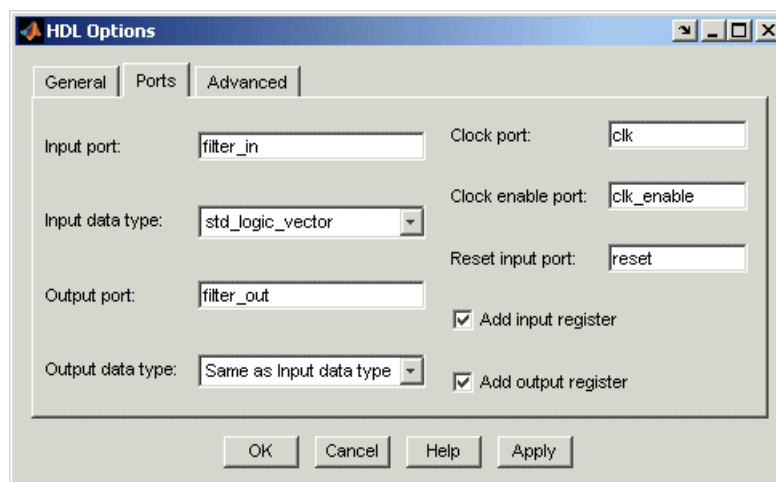
- 6 Click **What's This?** The Filter Design HDL Coder opens context-sensitive help that describes the **Name** option. Use the context-sensitive help as needed while using the GUI to configure options that control the contents and style of the generated HDL code and test bench. A help topic is available for each option and pane.
- 7 In the **Name** text box of the **HDL filter** 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 In the **Name** text box of the **Test bench types** pane, replace the default name with `basicfir_tb`. This option names the generated test bench file.
- 9 Click **More Options**. The Filter Design HDL Coder displays an **HDL Options** dialog.





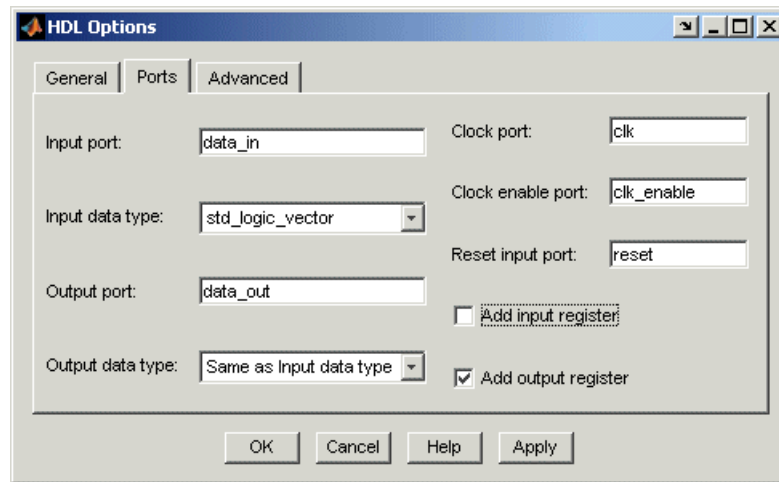
**10** In the **Comment in header** text box, type Tutorial - Basic FIR Filter and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.

**11** Click the **Ports** tab. The **Ports** pane appears.

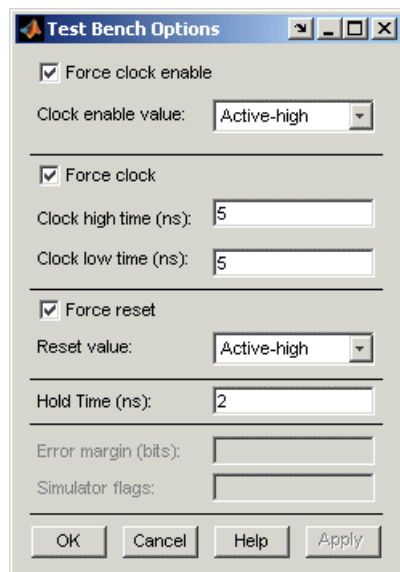


**12** Change the names of the input and output ports. Replace filter\_in with data\_in and filter\_out with data\_out.

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



- 14** Click **Apply** and then **OK** to register your changes and close the **HDL Options** dialog.
- 15** Click **Test Bench Options**. The Filter Design HDL Coder displays a **Test Bench Options** dialog.



You use this dialog to customize the generated test bench.

**16** For this tutorial, apply the default settings by clicking **OK**.

**17** In the **Generate HDL** dialog, click **Apply** or **OK** to start the code generation process. **OK** closes the dialog.

The Filter Design HDL 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: basicfir
### Generating basicfir.vhd file in: hdsrc
### Starting generation of basicfir VHDL entity
### Starting generation of basicfir VHDL architecture
### 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 VHDL file basicfir_tb.vhd in: hdsrc
### Done generating VHDL test bench.
```

As the messages indicate, the Filter Design HDL Coder creates the directory `hdlsrc` under your current working directory and places the files `basicfir.vhd` and `basicfir_tb.vhd` in that directory.

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:

<b>VHDL Port</b>	<b>Name</b>
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 `coeff $n$` , 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.
- Applies impulse, step, ramp, chirp, and white noise stimulus types.

## 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 **HDL filter** pane. See step 5 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 2-8.
- 3 Search for `Tutorial1`. This is where the Filter Design HDL 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 2-8.
- 4 Search for `HDL Code`. This section lists the Filter Design HDL Coder options you modified in “Configuring and Generating the FIR Filter's Optimized Verilog Code” on page 2-28.
- 5 Search for `Filter Settings`. This section describes the filter design and quantization settings as you specified in “Designing a Basic FIR Filter” on page 2-3 and “Quantizing the Basic FIR Filter” on page 2-5.
- 6 Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **HDL filter** pane. See step 5 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 2-8.
- 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 **HDL Options**

dialog. See step 12 in “Configuring and Generating the Basic FIR Filter’s VHDL Code” on page 2-8.

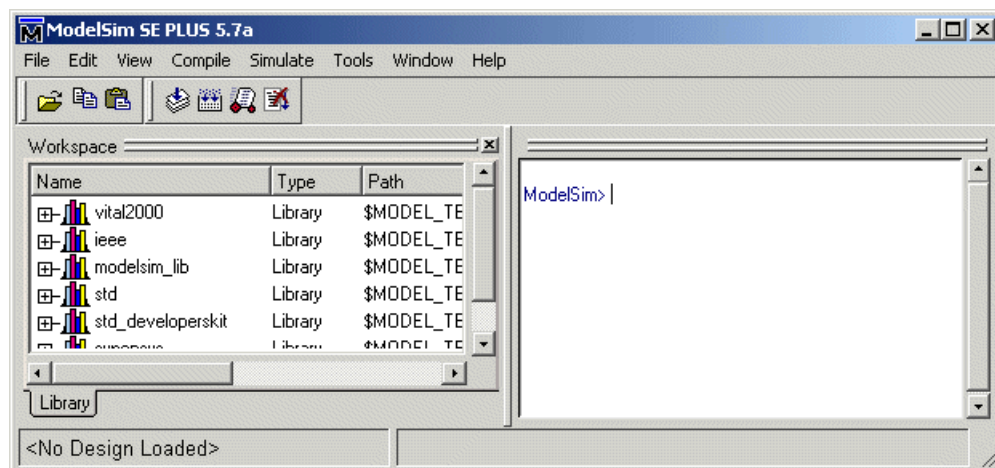
- 8 Search for Constants. This is where the coefficients are defined. They are named using the default naming scheme, `coeffn`, where  $n$  is the coefficient number, starting with 1.
- 9 Search for Signals. This is where the filter’s signals are defined.
- 10 Search for process. The PROCESS block name `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. The Filter Design HDL Coder generates the code for this register by default. In step 13 in “Configuring and Generating the Basic FIR Filter’s VHDL Code” on page 2-8, 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 ModelSim as the tool for compiling and simulating the VHDL code, you can use any VHDL simulation tool package.

To verify the filter code, complete the following steps:

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



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

```
cd d:/hdlfilter_tutorials/hdlsrc
```

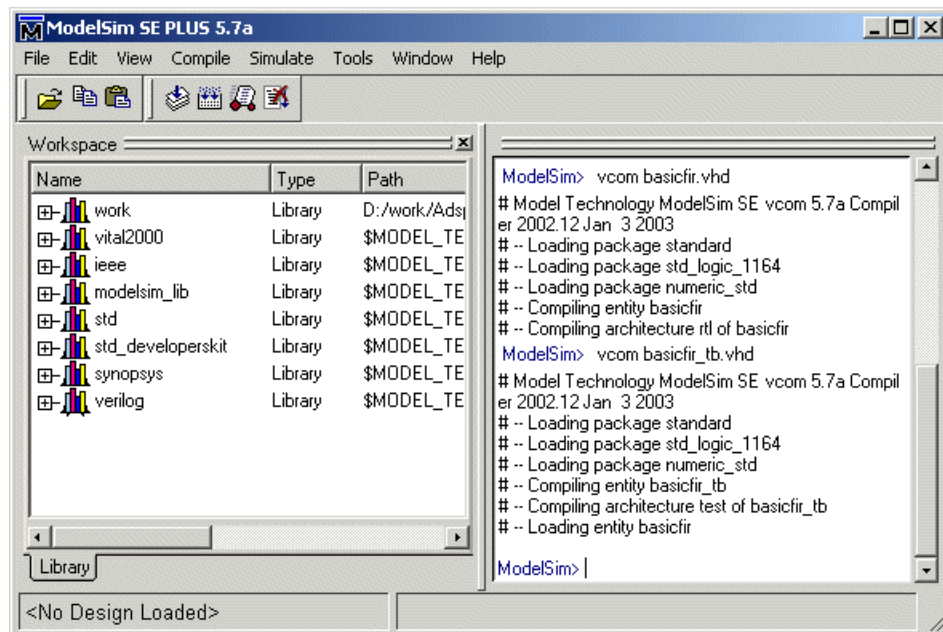
- 3** If necessary, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In ModelSim, you can create a design library with the `vlib` command.

```
vlib work
```

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

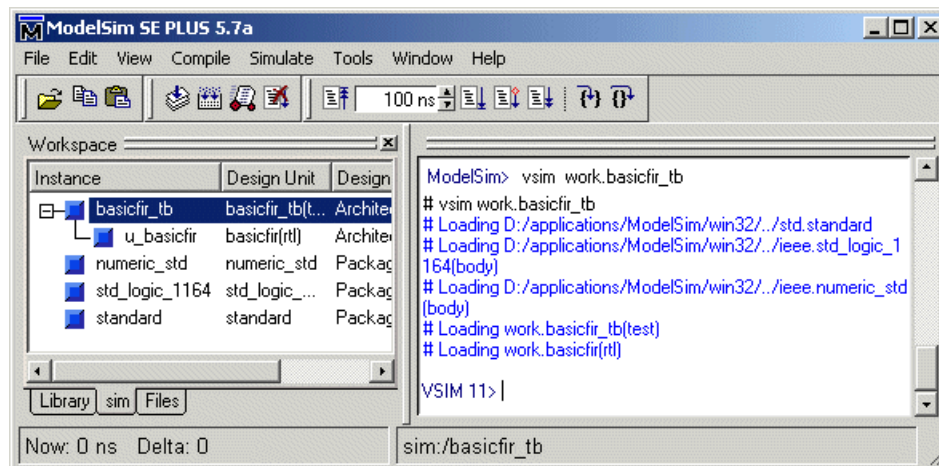


- 5 Load the test bench for simulation. The procedure for doing this varies depending on the simulator you are using. In ModelSim, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.basicfir_tb
```

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

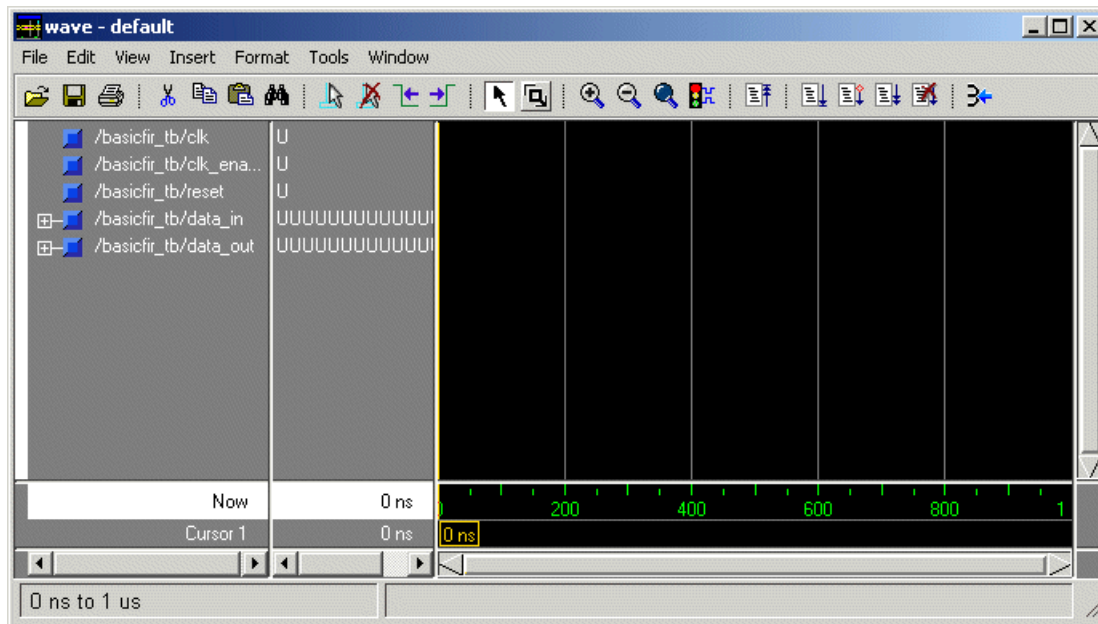




- 6 Open a display window for monitoring the simulation as the test bench runs. For example, in ModelSim, you can use the following command to open a **wave** window to view the results of the simulation as HDL waveforms:

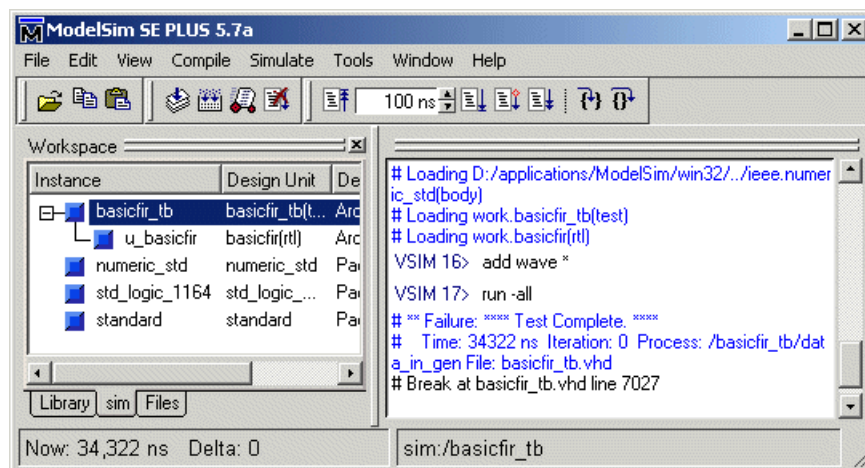
```
add wave *
```

The following Wave window displays.



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

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



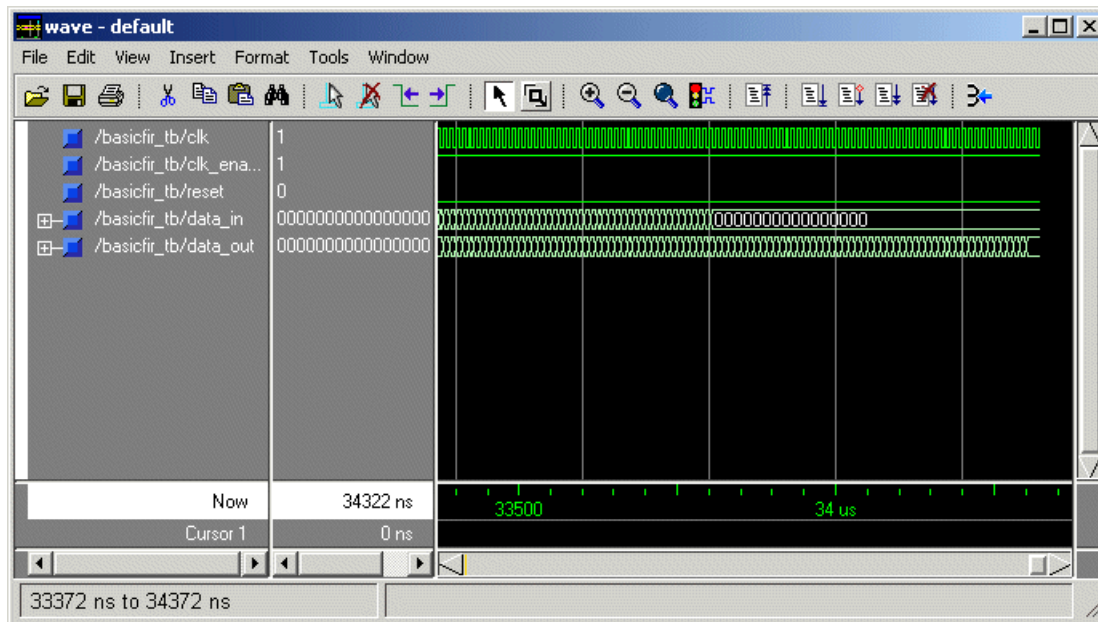
As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. You must determine whether the results are expected based on the customizations you specified when generating the filter VHDL 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 successfully run to completion. The `Failure` part of the message is tied to the mechanism the Filter Design HDL Coder uses to end the simulation.

---

The following Wave window shows the simulation results as HDL waveforms.



## Optimized FIR Filter Tutorial

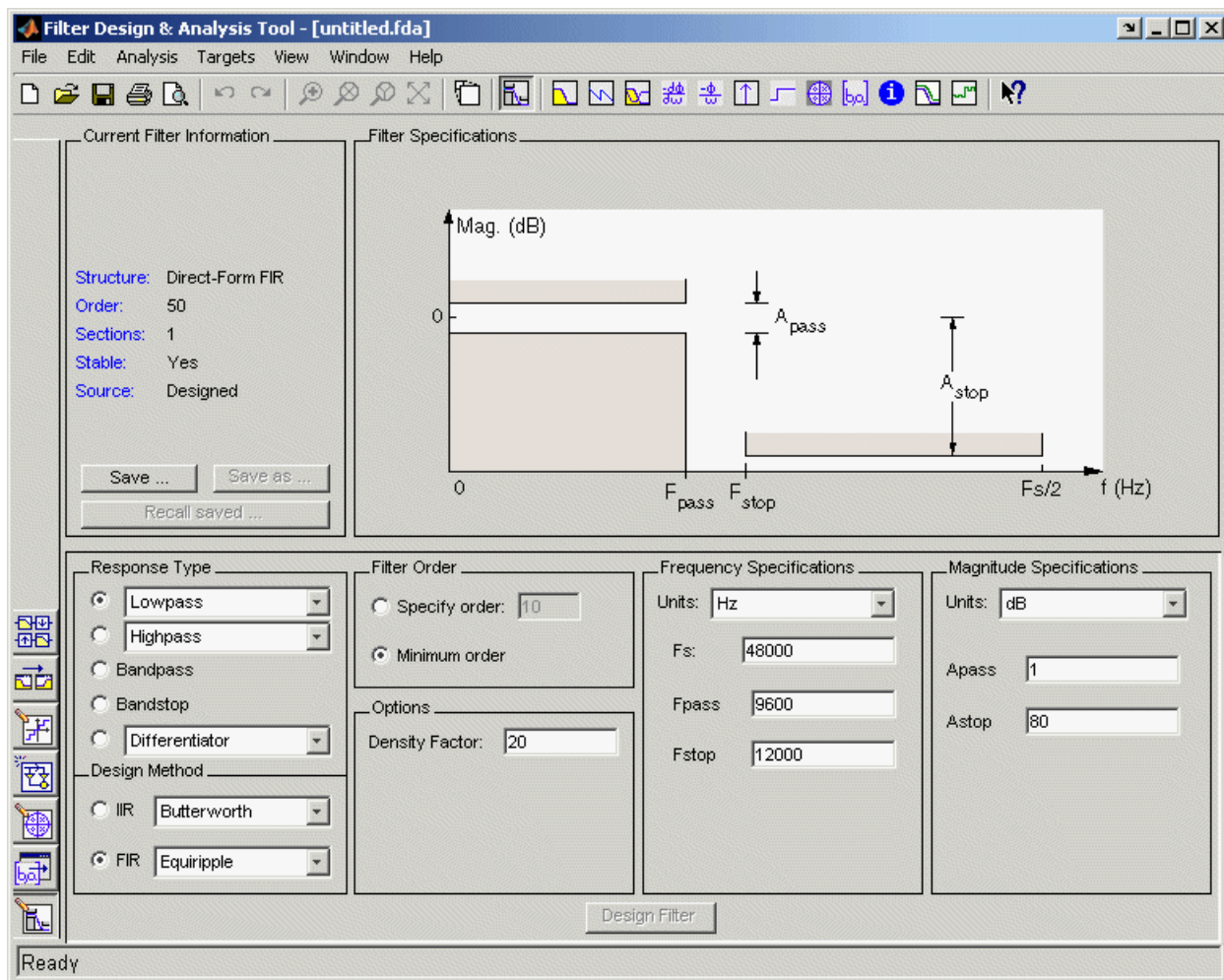
This section guides you through the steps for designing an optimized quantized discrete-time FIR filter, generating Verilog code for the filter, and verifying the Verilog code with a generated test bench. The procedure is presented in the following topics:

- “Designing the FIR Filter” on page 2-23
- “Quantizing the FIR Filter” on page 2-25
- “Configuring and Generating the FIR Filter’s Optimized Verilog Code” on page 2-28
- “Getting Familiar with the FIR Filter’s Optimized Generated Verilog Code” on page 2-35
- “Verifying the FIR Filter’s Optimized Generated Verilog Code” on page 2-37

### Designing the FIR Filter

One way of designing a filter in the MATLAB environment is to use the FDATool. This section guides you through the procedure of designing and creating a filter for an FIR filter to which you will apply VHDL optimizations. These instructions assume you are familiar with the MATLAB user interface and the FDATool.

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



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

<b>Option</b>	<b>Value</b>
Response Type	Lowpass
Design Method	FIR Equiripple

<b>Option</b>	<b>Value</b>
Filter Order	Minimum order (50)
Options	Density Factor: 20
Frequency Specifications	Units: Hz Fs: 48000 Fpass: 9600 Fstop: 12000
Magnitude Specifications	Units: dB Apass: 1 Astop: 80

These are settings are for the default filter design that the FDATool creates for you. If you do not need to make any changes and **Design Filter** is greyed out, you are done and can skip to “Quantizing the FIR Filter” on page 2-25.

- 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

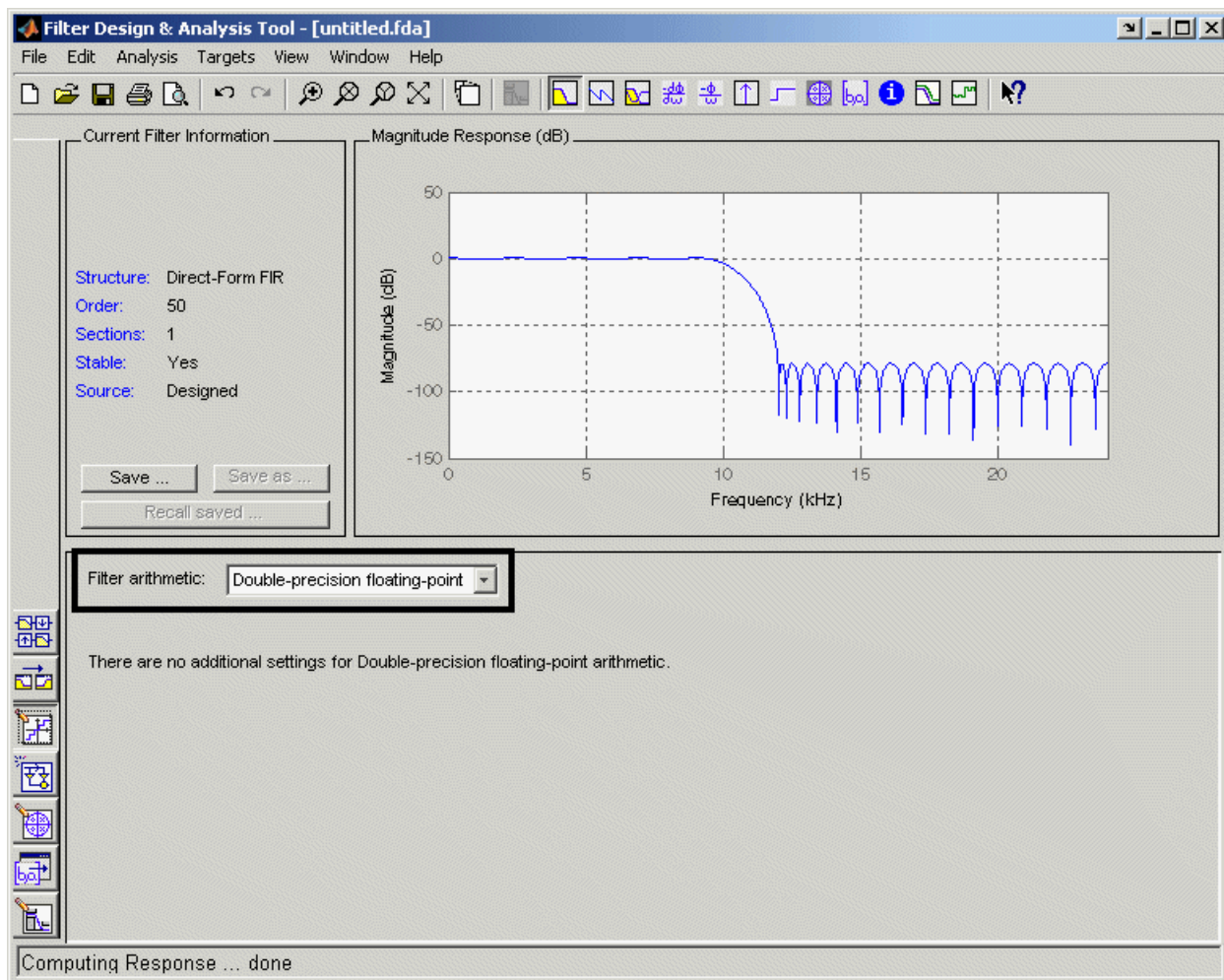
For more information on designing filters with the FDATool, see the FDATool and Filter Design Toolbox documentation.

## Quantizing the FIR Filter

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

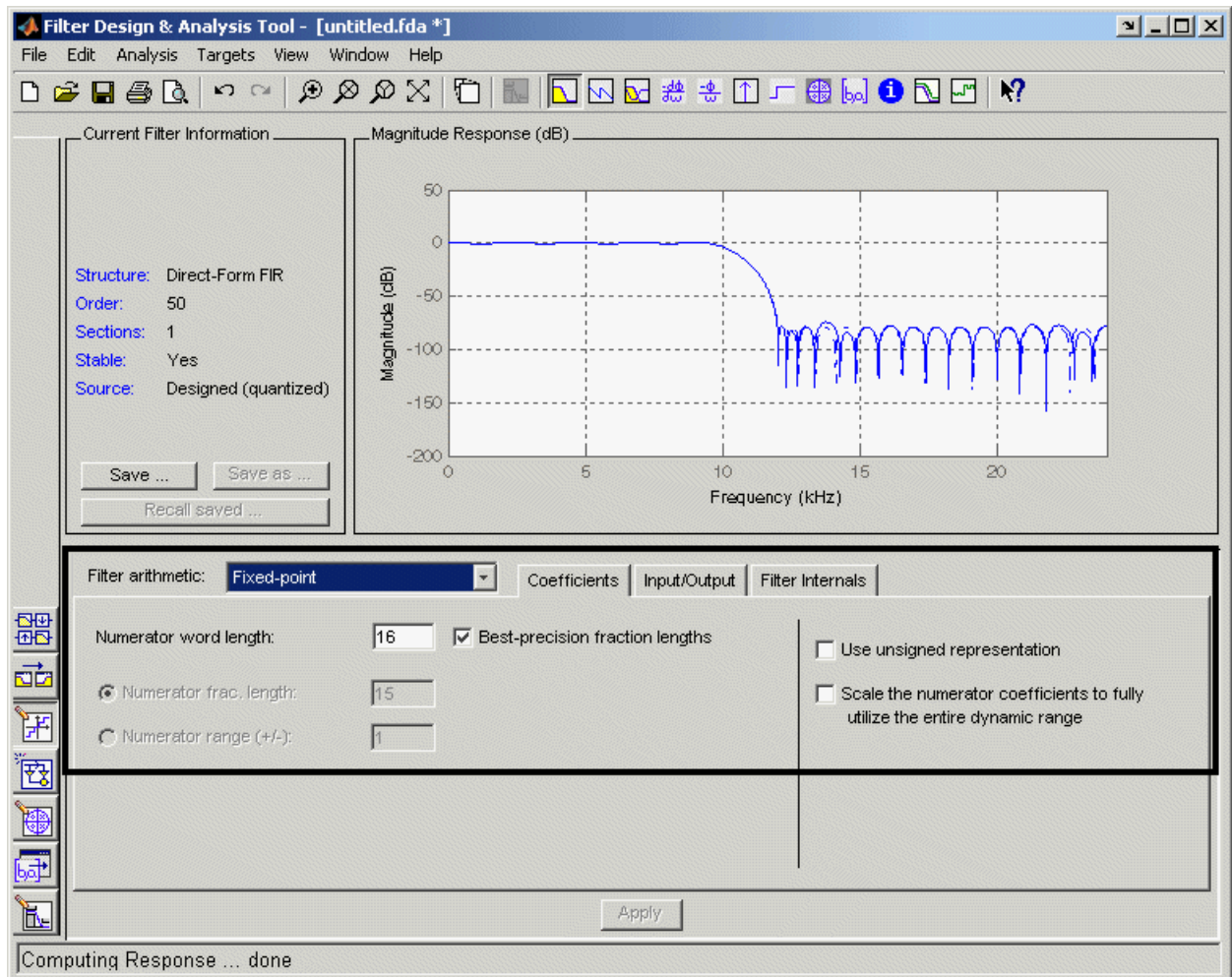
- 1 Open the FIR filter design you created in if it is not already open.

- 2 Click the **Set quantization parameters** icon  in the left-side tool bar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog.



- 3 Select Fixed-point from the menu. The FDATool displays the first of three tabbed panels of quantization parameters across the bottom half of its dialog.





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:

<b>Tab</b>	<b>Parameter</b>	<b>Setting</b>
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
	Avoid overflow	Selected
Filter Internals	Round towards	Floor
	Overflow mode	Saturate
	Product mode	Full precision
	Accum. mode	Keep MSB
	Accum. word length	40
	Cast signals before accum.	Set

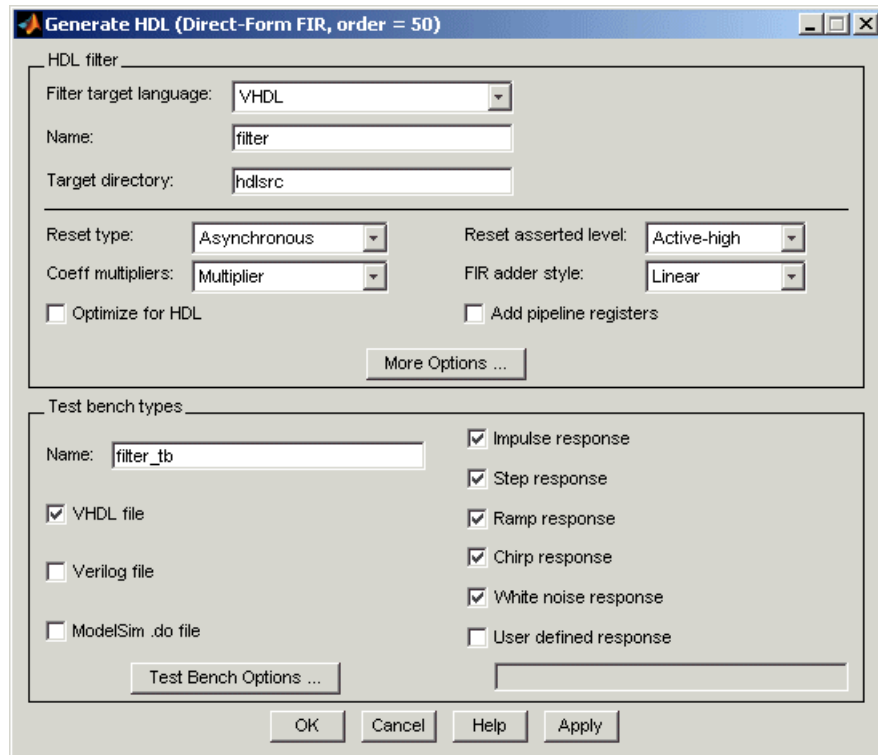
**5** Click **Apply**.

For more information on quantizing filters, see the FDATool and Filter Design Toolbox documentation.

## **Configuring and Generating the FIR Filter’s Optimized Verilog Code**

After you quantize your filter, you are ready to use the Filter Design HDL Coder to configure and generate the filter’s Verilog code. This section guides you through the process for starting the Filter Design HDL Coder 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” on page 2-23 and “Quantizing the FIR Filter” on page 2-25.

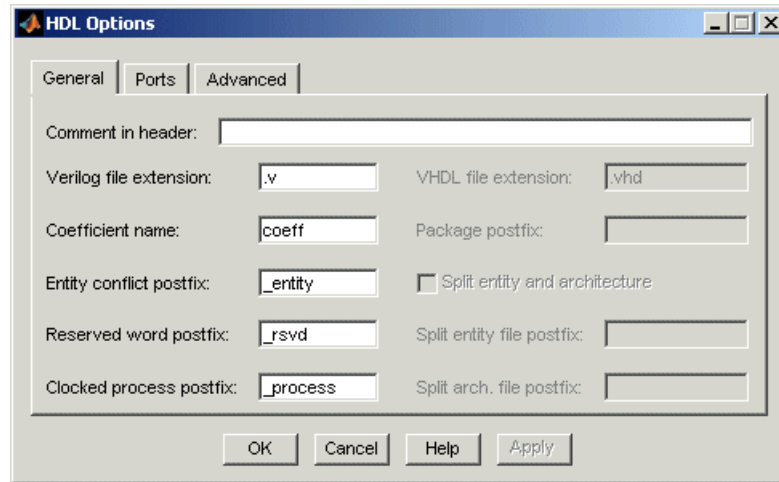
- 1 Start the Filter Design HDL Coder by clicking **Targets->Generate HDL** in the FDATool dialog. The FDATool displays the Filter Design HDL Coder dialog.



- 2 Select Verilog for the **Filter target language** option, as shown in the following dialog.

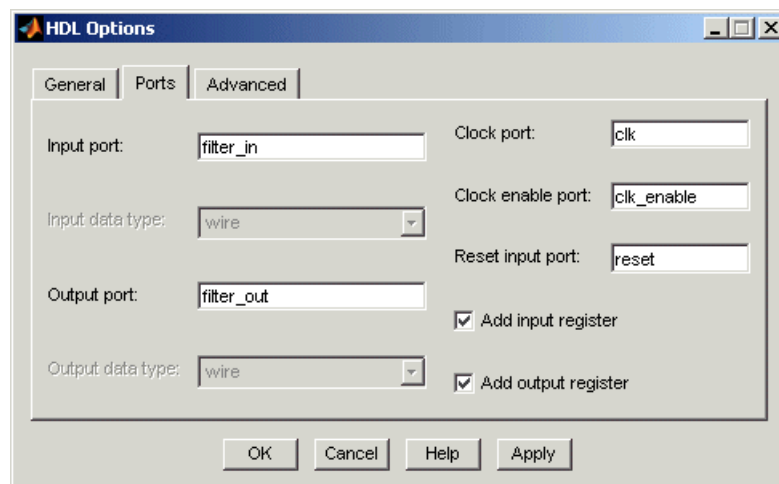


- 3 In the **Name** text box of the **HDL filter** pane, replace the default name with `optfir`. This option names the Verilog module and the file that is to contain the filter's Verilog code.
- 4 In the **Name** text box of the **Test bench types** pane, replace the default name with `optfir_tb`. This option names the generated test bench file.
- 5 In the **HDL filter** 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 Filter Design HDL 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 MATLAB and the simulated results for the optimized HDL code.
- 6 Select CSD for the **Coeff multipliers** option. This option optimizes coefficient multiplier operations by instructing the coder to replace them with additions of partial products produced by a canonic 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. This option also has the potential for producing numeric differences between MATLAB filter results and the simulated results for the optimized HDL code.
- 7 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 MATLAB filter results and the simulated results for the optimized HDL code.
- 8 Click **More Options**. The Filter Design HDL Coder displays an **HDL Options** dialog.



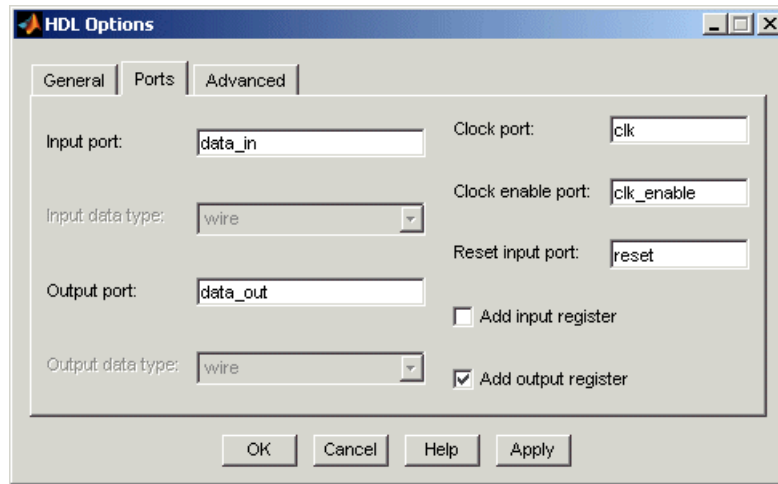
**9** In the **Comment in header** text box, type Tutorial - Optimized FIR Filter and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.

**10** Click the **Ports** tab. The **Ports** appears.

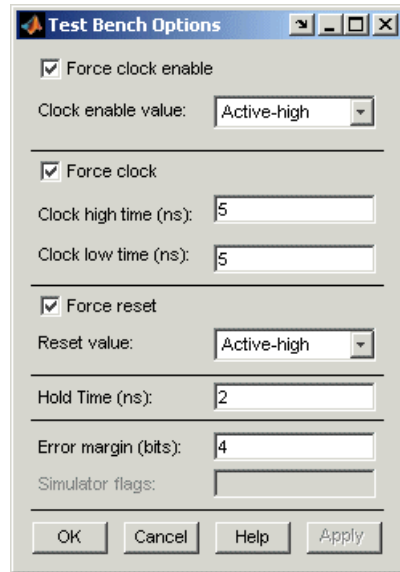


**11** Change the names of the input and output ports. Replace filter\_in with data\_in and filter\_out with data\_out.

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



- 13 Click **Apply** and then **OK** to register your changes and close the **HDL Options** dialog.
- 14 Click **Test Bench Options**. The Filter Design HDL Coder displays a **Test Bench Options** dialog.



Use this dialog to customize the generated test bench. Note that the **Error margin (bits)** option is enabled. This is due to the use of optimization options that potentially produce numeric results that differ from the results of the original MATLAB filter. You can use this option to adjust the number of least significant bits the test bench will ignore during comparisons before generating a warning.

**15** For this tutorial, apply the default settings by clicking **OK**.

**16** In the **Generate HDL** dialog, click **Apply** or **OK** to start the code generation process. **OK** closes the dialog.

The Filter Design HDL 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 optfir.v file in: hdlsrc
### Starting generation of optfir Verilog module
### Starting generation of optfir Verilog module body
### HDL latency is 6 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 Verilog file optfir_tb.v in: hdlsrc
### Done generating Verilog test bench.
```

As the messages indicate, the Filter Design HDL Coder creates the directory `hdlsrc` under your current working directory and places the files `optfir.v` and `optfir_tb.v` in that directory.

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:

<b>Verilog Port</b>	<b>Name</b>
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.
- 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 **HDL filter** pane. See step 3 in "Configuring and Generating the FIR Filter's Optimized Verilog Code" on page 2-28.
- 3 Search for `Tutorial1`. This is where the Filter Design HDL 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 2-28.
- 4 Search for `HDL Code`. This section lists the Filter Design HDL Coder options you modified in "Configuring and Generating the FIR Filter's Optimized Verilog Code" on page 2-28.
- 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" on page 2-23 and "Quantizing the FIR Filter" on page 2-25.

- 6 Search for `module`. This line names the Verilog module, using the string you specified for the **Name** option in the **HDL filter** pane. This line also declares the list of ports, as defined by options on the **Ports** pane of the **HDL Options** dialog. 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 **HDL Options** dialog. See steps 3 and 11 in “Configuring and Generating the FIR Filter’s Optimized Verilog Code” on page 2-28.
- 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 2-28.
- 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 2-28.
- 15 Search for `output_register`. This is where filter output is written to an output register. The Filter Design HDL Coder generates the code for this register by default. In step 12 in “Configuring and Generating the FIR Filter’s Optimized Verilog Code” on page 2-28, 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`.

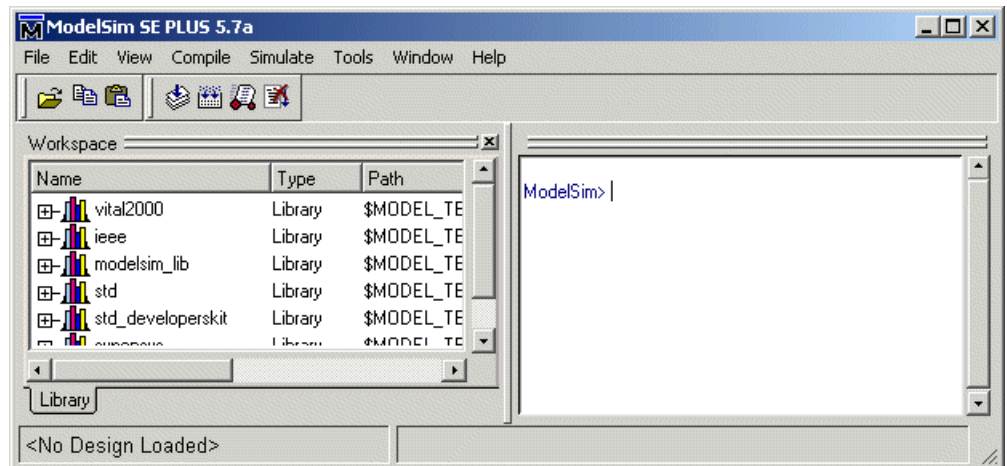
**16** Search for `data_out`. This is where the filter writes its output data.

## 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 ModelSim as the tool for compiling and simulating the Verilog code, you can use any HDL simulation tool package.

To verify the filter code, complete the following steps:

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



**2** Set the current directory to the directory that contains your generated Verilog files. For example:

```
cd hdlsrc
```

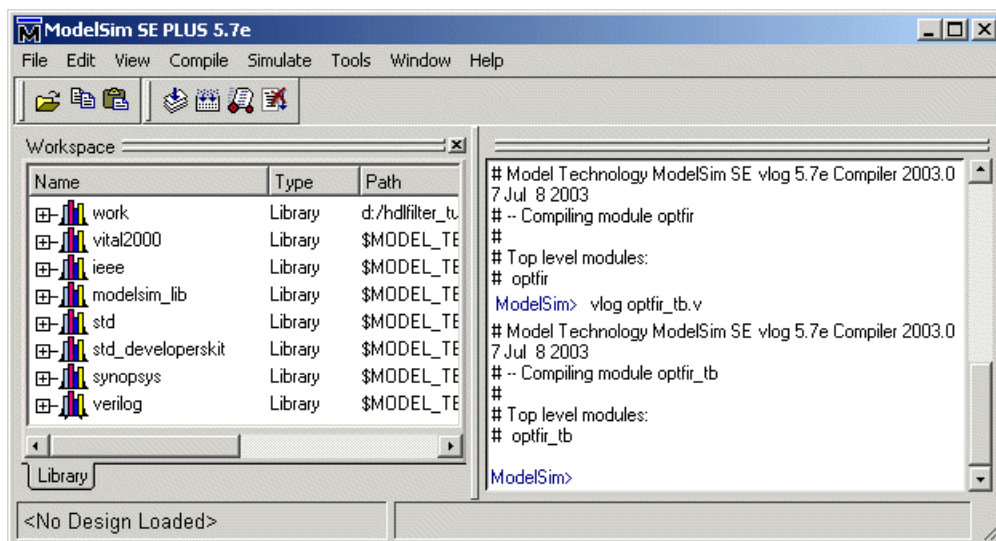
**3** If necessary, create a design library to store the compiled Verilog modules. In ModelSim, you can create a design library with the `vlib` command.

```
vlib work
```

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

```
vlog optfir.vhd
vlog optfir_tb.vhd
```

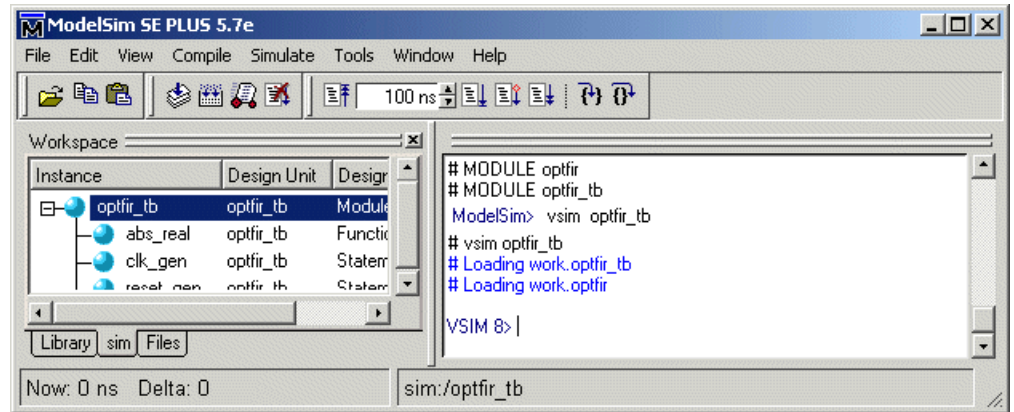
The following screen display shows this command sequence and informational messages displayed during compilation.



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

```
vsim optfir_tb
```

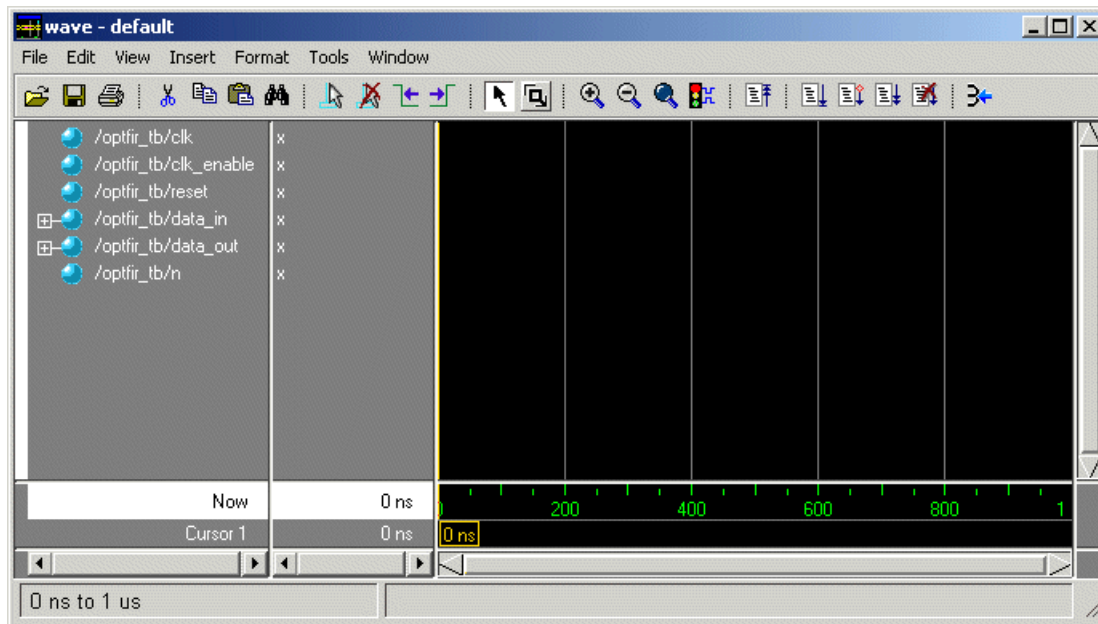
The following ModelSim display shows the results of loading `optfir_tb` with the `vsim` command.



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

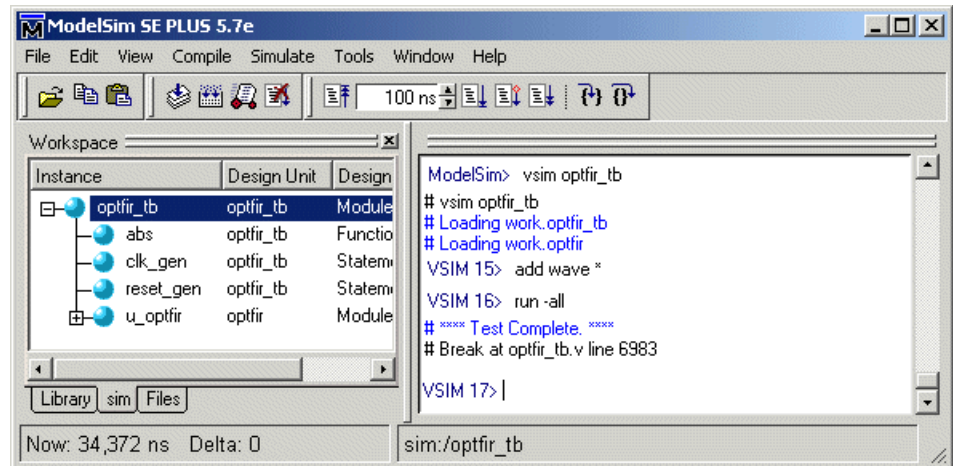
```
add wave *
```

The following Wave window displays:



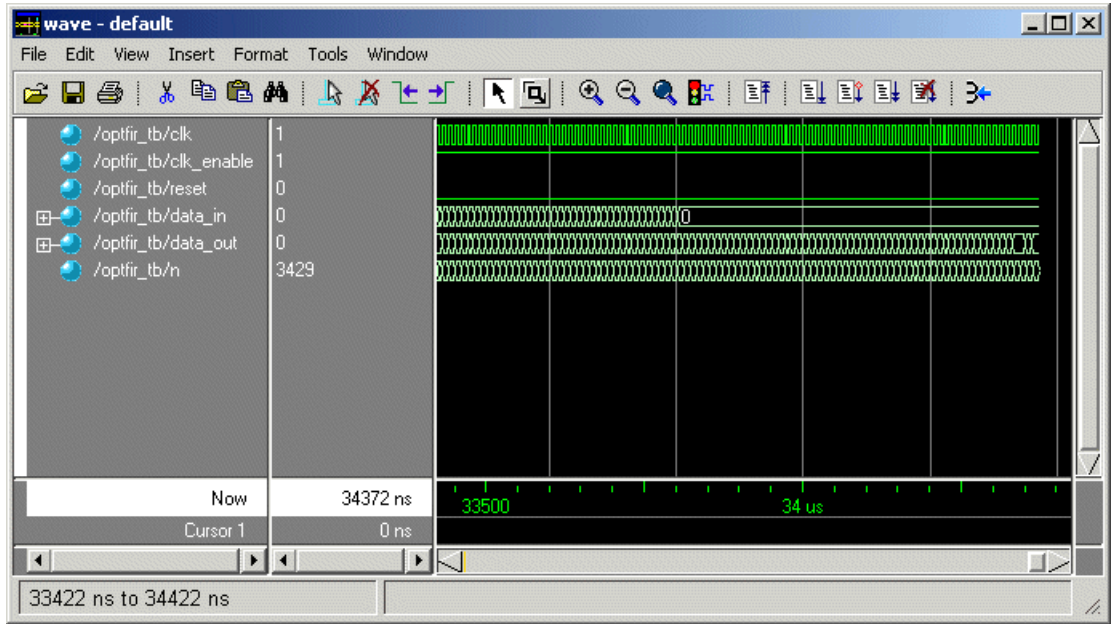
- 7 To start running the simulation, issue the appropriate command for your simulator. For example, in ModelSim, you can start a simulation with the run command.

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



As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. 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 Tutorial

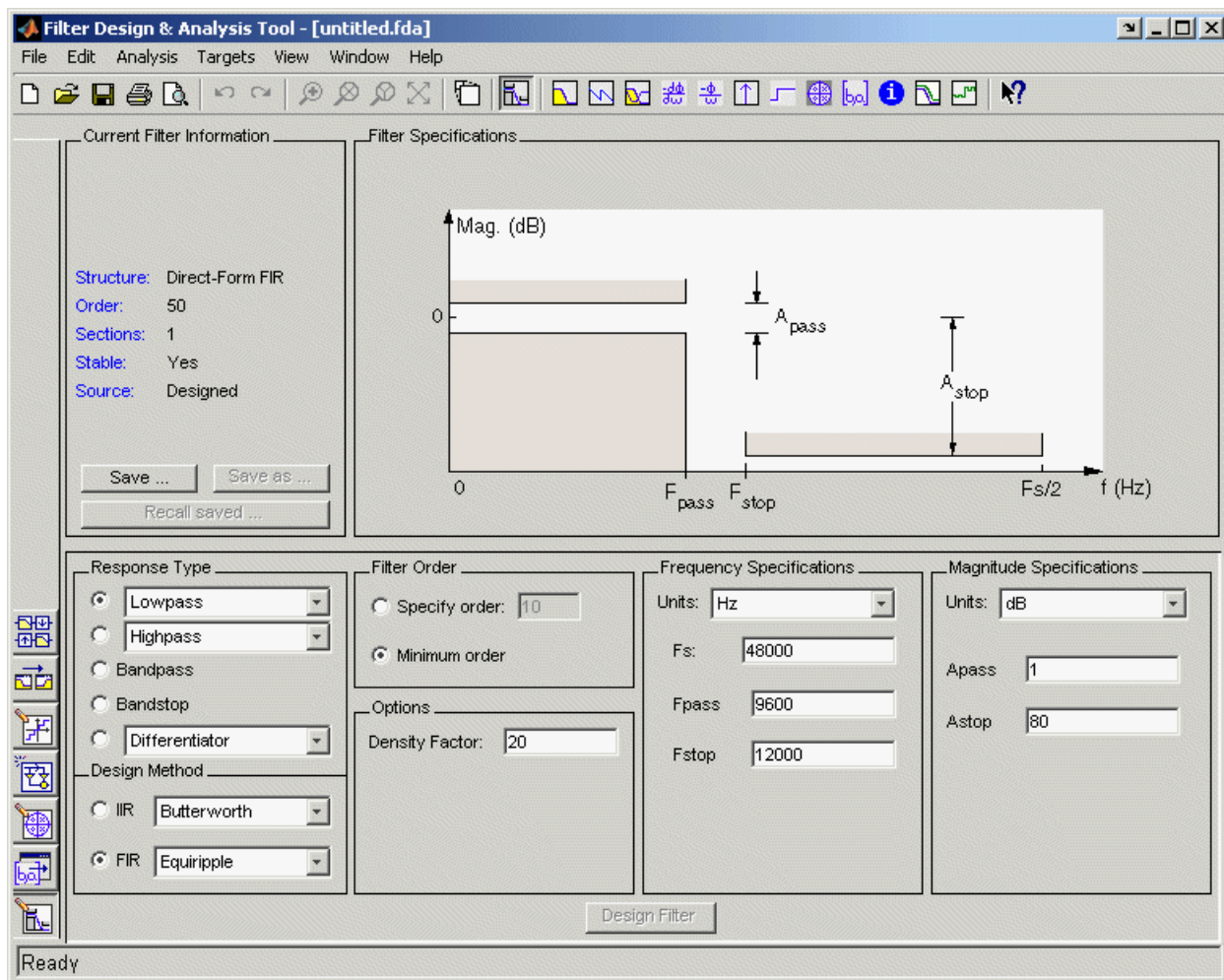
This section guides you through the steps for designing a basic quantized discrete-time IIR filter, generating VHDL code for the filter, and verifying the VHDL code with a generated test bench. The procedure is presented in the following topics:

- “Designing an IIR Filter” on page 2-43
- “Quantizing the IIR Filter” on page 2-45
- “Configuring and Generating the IIR Filter’s VHDL Code” on page 2-49
- “Getting Familiar with the IIR Filter’s Generated VHDL Code” on page 2-55
- “Verifying the IIR Filter’s Generated VHDL Code” on page 2-56

### Designing an IIR Filter

One way of designing a filter in the MATLAB environment is to use the FDATool. This section guides you through the procedure of designing and creating a filter for an IIR filter. These instructions assume you are familiar with the MATLAB user interface and the FDATool.

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



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

<b>Option</b>	<b>Value</b>
Response Type	Highpass
Design Method	IIR Butterworth

<b>Option</b>	<b>Value</b>
Filter Order	5
Frequency Specifications	Units: Hz
	Fs: 48000
	Fc: 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 the FDATool and Filter Design Toolbox documentation.

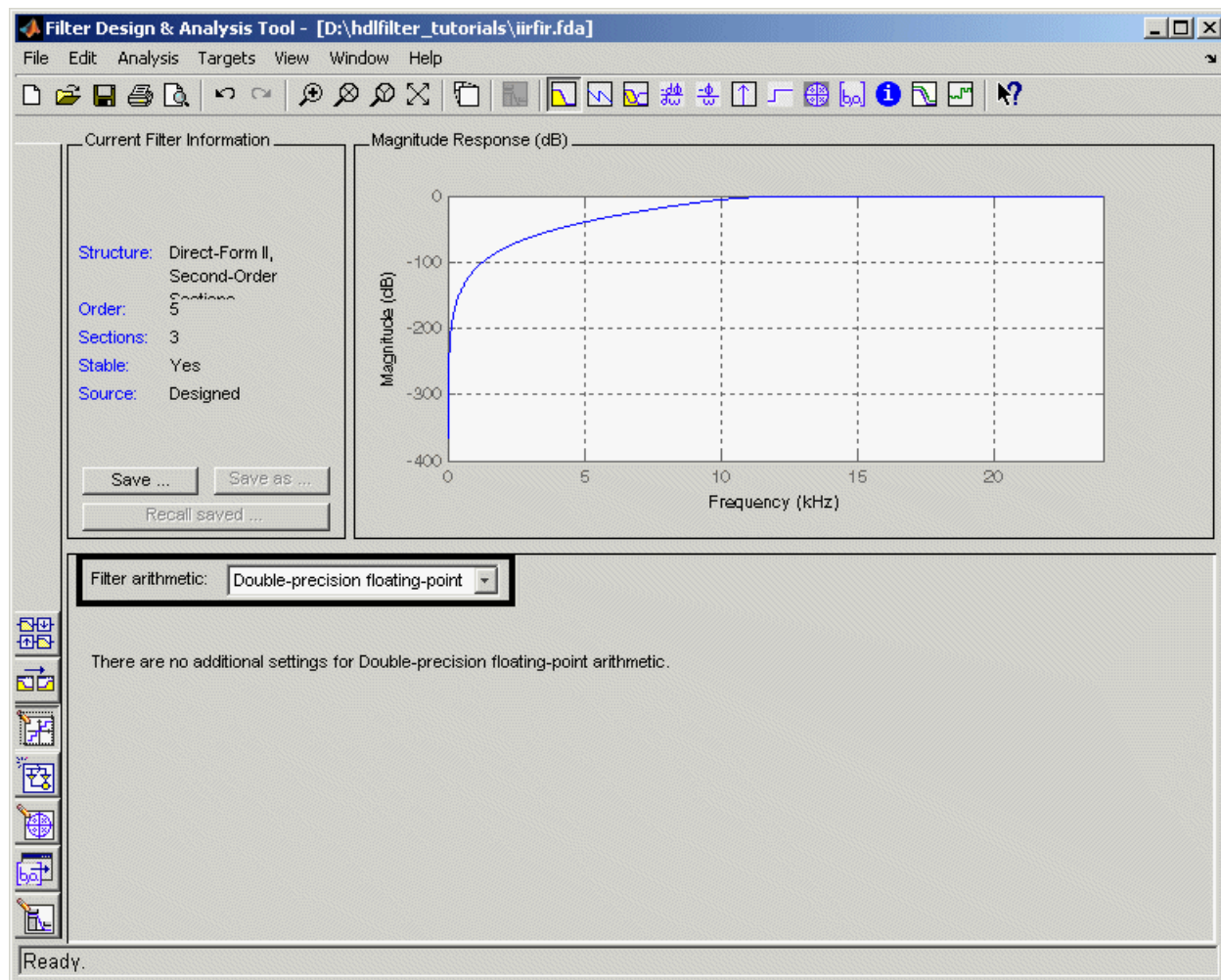
## Quantizing the IIR Filter

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

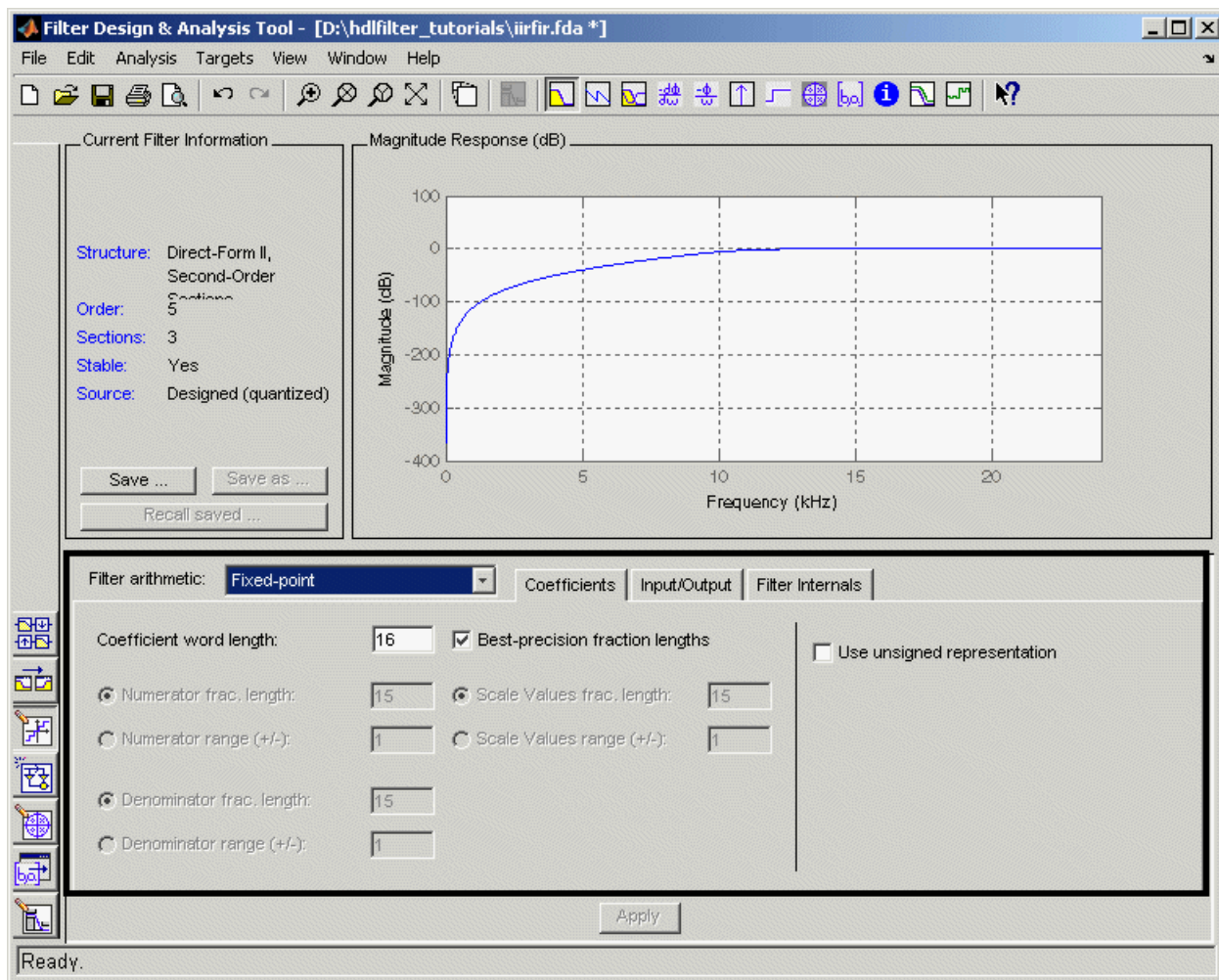
- 1** Open the IIR filter design you created in “Designing an IIR Filter” on page 2-43 if it is not already open.



- 2** Click the **Set quantization parameters** icon in the left-side tool bar. The FDATool displays the **Filter arithmetic** menu in the bottom half of its dialog.

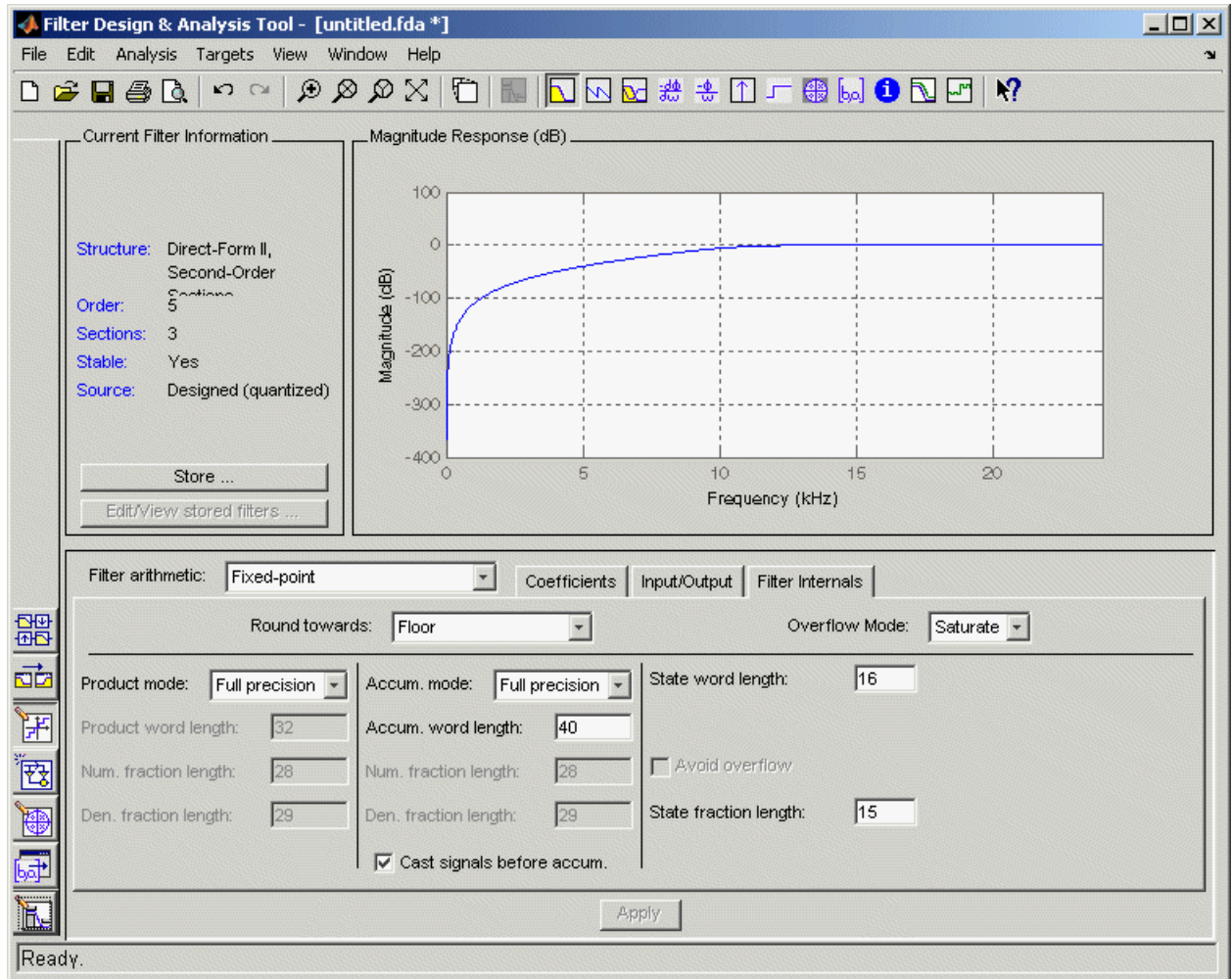


- 3 Select Fixed-point from the menu. The FDATool displays the first of three tabbed panels of its dialog.



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 Click the **Filter Internals** tab and set **Round towards** to Floor and **Overflow Mode** to Saturate.
- 5 Click **Apply**. The quantized filter appears as follows.

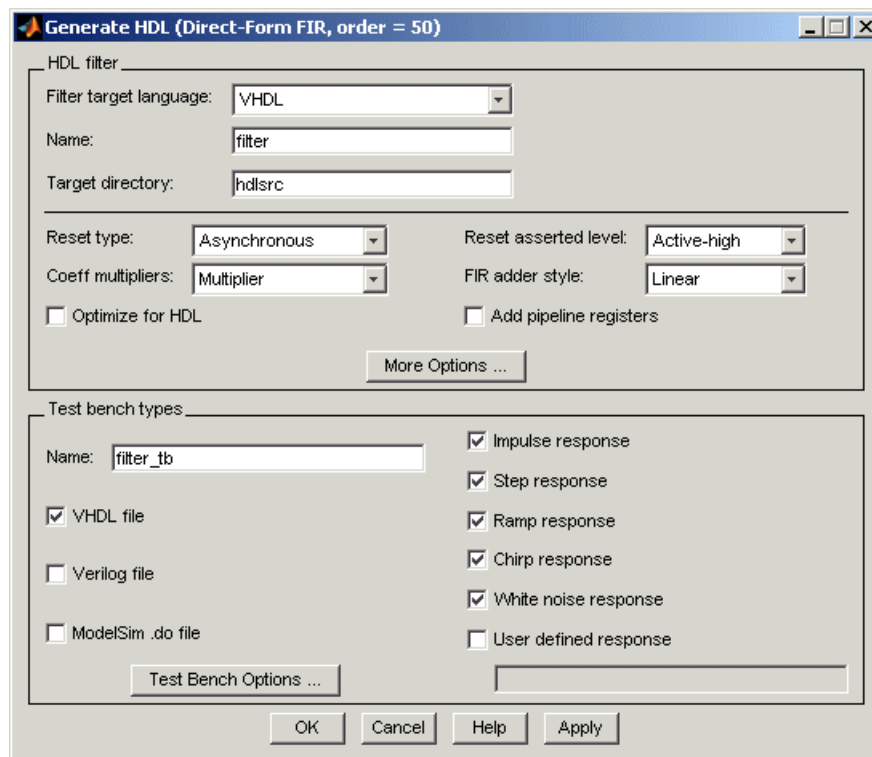


For more information on quantizing filters, see the FDATool and Filter Design Toolbox documentation.

## Configuring and Generating the IIR Filter's VHDL Code

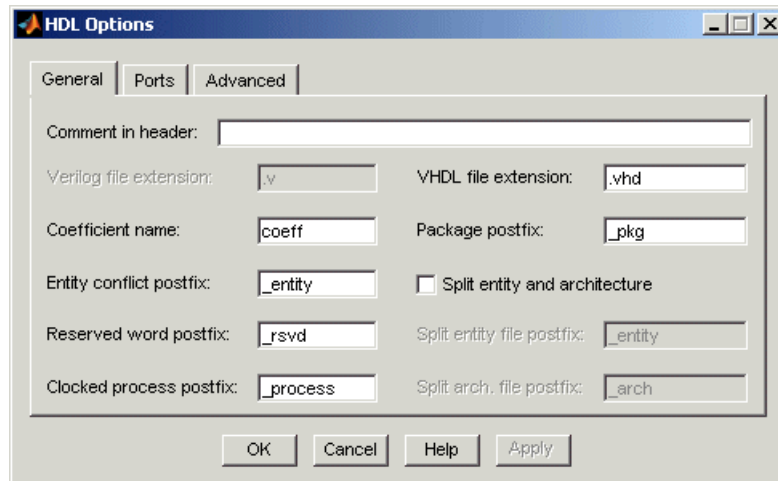
After you quantize your filter, you are ready to use the Filter Design HDL Coder to configure and generate the filter's VHDL code. This section guides you through the procedure for starting the Filter Design HDL Coder GUI, setting some options, and generating the VHDL code and a test bench for the IIR filter you designed and quantized in “Designing an IIR Filter” on page 2-43 and “Quantizing the IIR Filter” on page 2-45

- 1 Start the Filter Design HDL Coder by clicking **Targets**→**Generate HDL** in the FDATool dialog. The FDATool displays the Filter Design HDL Coder dialog.



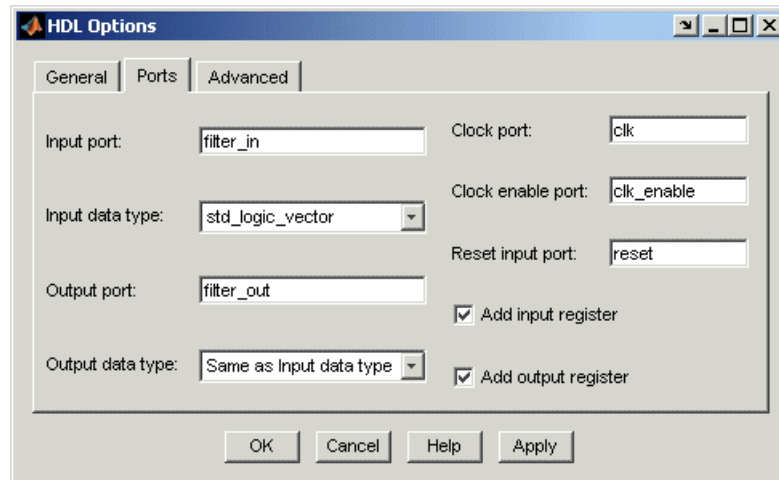
- 2 In the **Name** text box of the **HDL filter** pane, type `iir`. This option names the VHDL entity and the file that is to contain the filter's VHDL code.

- 3 In the **Name** text box of the **Test bench types** pane, type `iir_tb`. This option names the generated test bench file.
- 4 Click **More Options**. The Filter Design HDL Coder displays an **HDL Options** dialog.

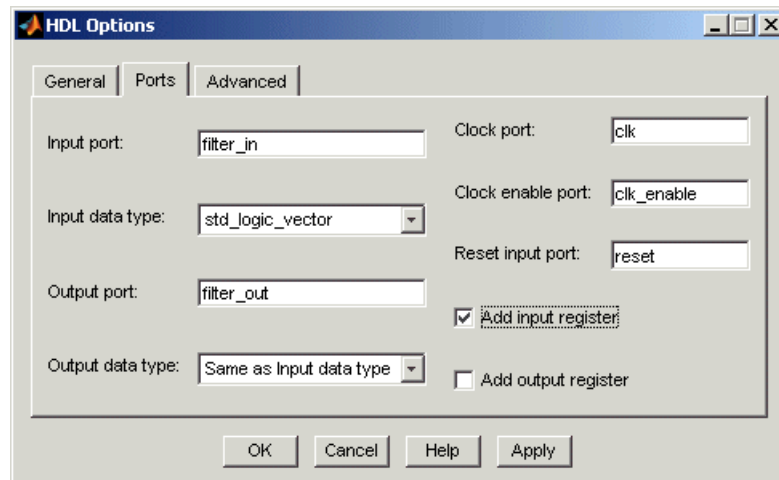


- 5 In the **Comment in header** text box, type `Tutorial - IIR Filter` and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.
- 6 Click the **Ports** tab. The **Ports** pane appears.

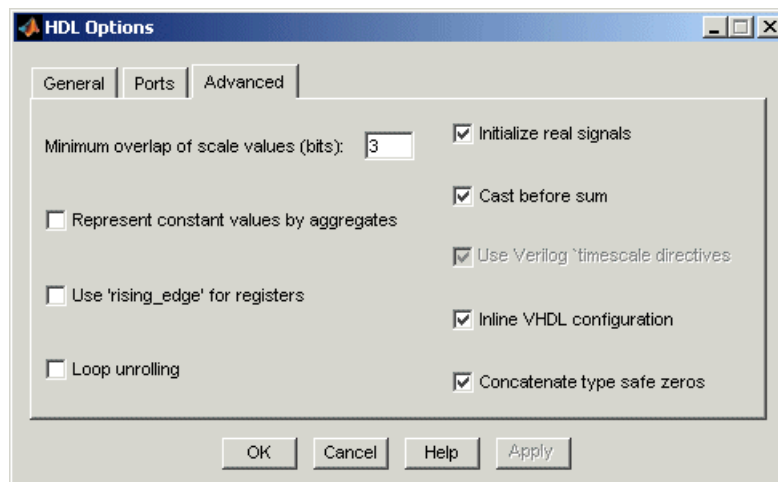




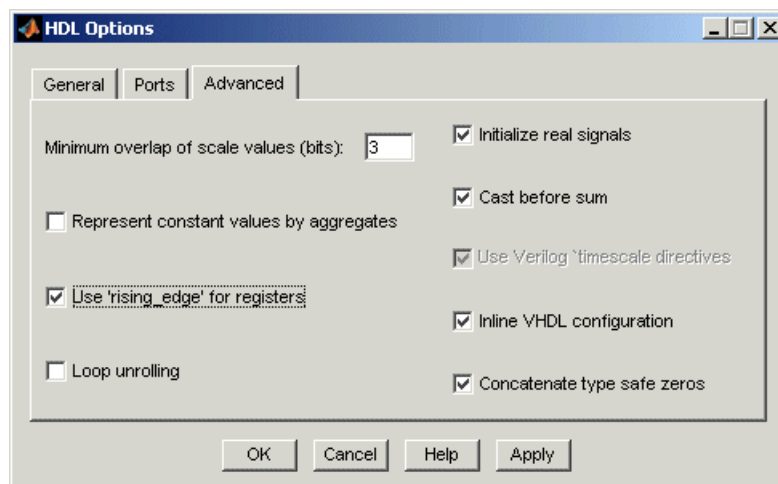
- 7** Clear the check box for the **Add output register** option. The **Ports** tab should now look like the following.



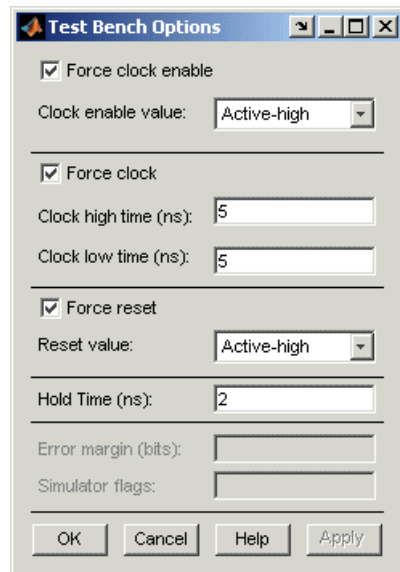
- 8** Click **Apply**.
- 9** Click the **Advanced** tab. The **Advanced** pane appears.



- 10** Select the **Use 'rising\_edge' for registers** option. The **Advanced** pane should now look like the following.



- 11** Click **Apply** to register your changes and then **OK** to close the dialog.
- 12** Click **Test Bench Options**. The Filter Design HDL Coder displays a **Test Bench Options** dialog.



You use this dialog to customize the generated test bench.

**13** For this tutorial, apply the default settings by clicking **OK**.

**14** In the **Generate HDL** dialog, click **Apply** or **OK** to start the code generation process. **OK** closes the dialog.

The Filter Design HDL 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
### Generating iir.vhd file in:hdlsrc
### Starting generation of iir VHDL entity
### Starting generation of iir VHDL architecture
### Second-order section, # 1
### Second-order section, # 2
### First-order section, # 3
### 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 VHDL file iir_tb.vhd in: hdlsrc  
### Done generating VHDL test bench.
```

As the messages indicate, the Filter Design HDL Coder creates the directory `hdlsrc` under your current working directory and places the files `iir.vhd` and `iir_tb.vhd` in that directory.

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:

<b>VHDL Port</b>	<b>Name</b>
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.
- Applies step, ramp, and chirp 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 **HDL filter** pane. See step 2 in “Configuring and Generating the IIR Filter's VHDL Code” on page 2-49.
- 3 Search for `Tutorial1`. This is where the Filter Design HDL 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 2-49.
- 4 Search for `HDL Code`. This section lists the Filter Design HDL Coder options you modified in “Configuring and Generating the IIR Filter's VHDL Code” on page 2-49.
- 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” on page 2-43 and “Quantizing the IIR Filter” on page 2-45.
- 6 Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **HDL filter** pane. See step 2 in “Configuring and Generating the IIR Filter's VHDL Code” on page 2-49.
- 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.

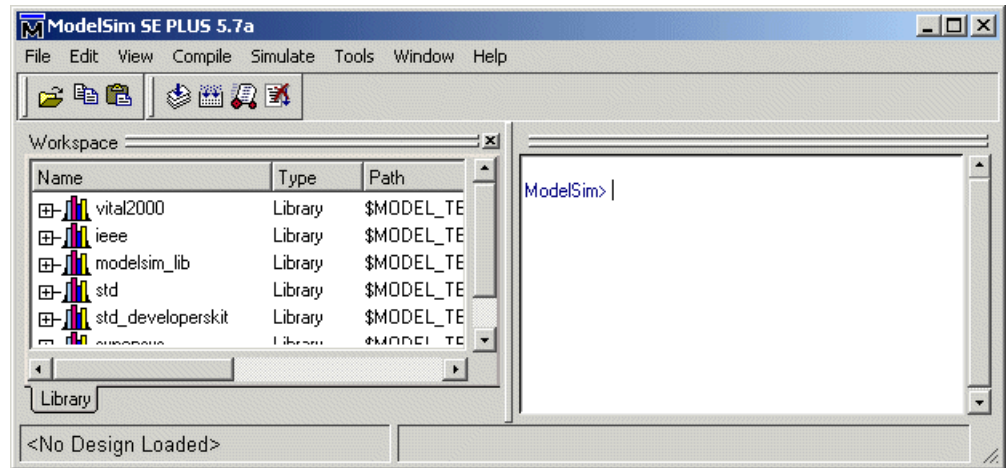
- 8 Search for `CONSTANT`. This is where the coefficients are defined. They are named using the default naming scheme, `coeff_xm_sectionn`, where  $x$  is a or b,  $m$  is the coefficient number, and  $n$  is the section number.
- 9 Search for `SIGNAL`. This is where the filter's signals are defined.
- 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. The Filter Design HDL Coder generates the code for this register by default. In step 7 in “Configuring and Generating the Basic FIR Filter's VHDL Code” on page 2-8, you cleared the **Add output register** option, but left the **Add input register** 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 **HDL Options** dialog. See step 10 in “Configuring and Generating the IIR Filter's VHDL Code” on page 2-49.
- 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 ModelSim as the tool for compiling and simulating the VHDL code, you can use any HDL simulation tool package.

To verify the filter code, complete the following steps:

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



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

```
cd hdlsrc
```

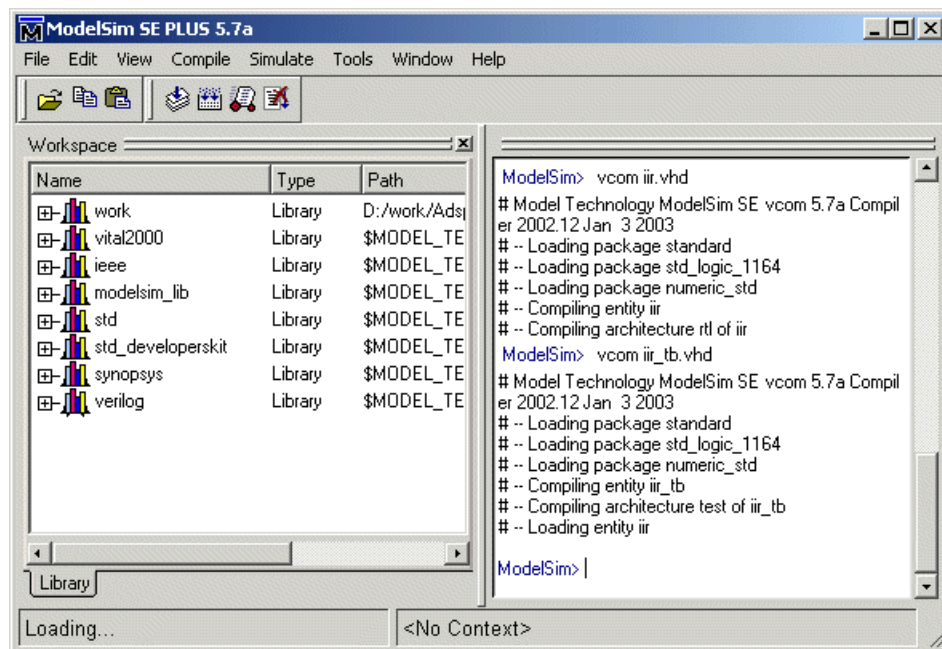
- 3 If necessary, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In ModelSim, you can create a design library with the `vlib` command.

```
vlib work
```

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

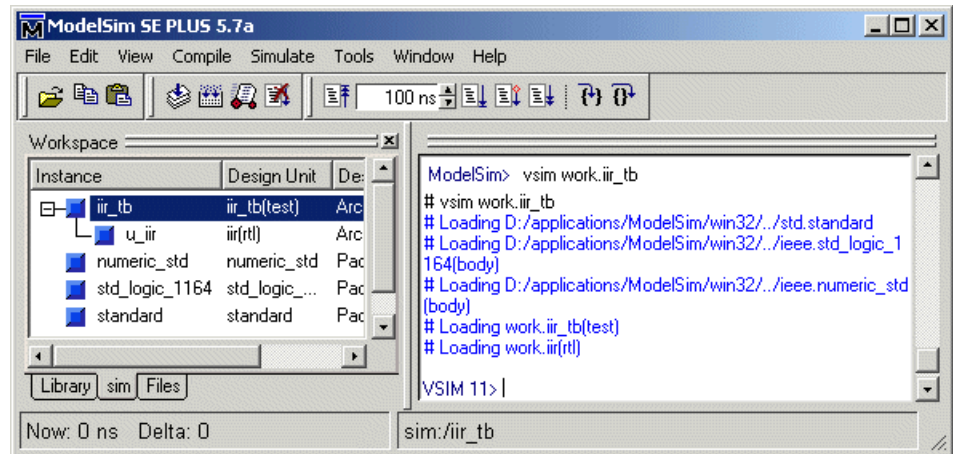


- 5 Load the test bench for simulation. The procedure for doing this varies depending on the simulator you are using. In ModelSim, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.iir_tb
```

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



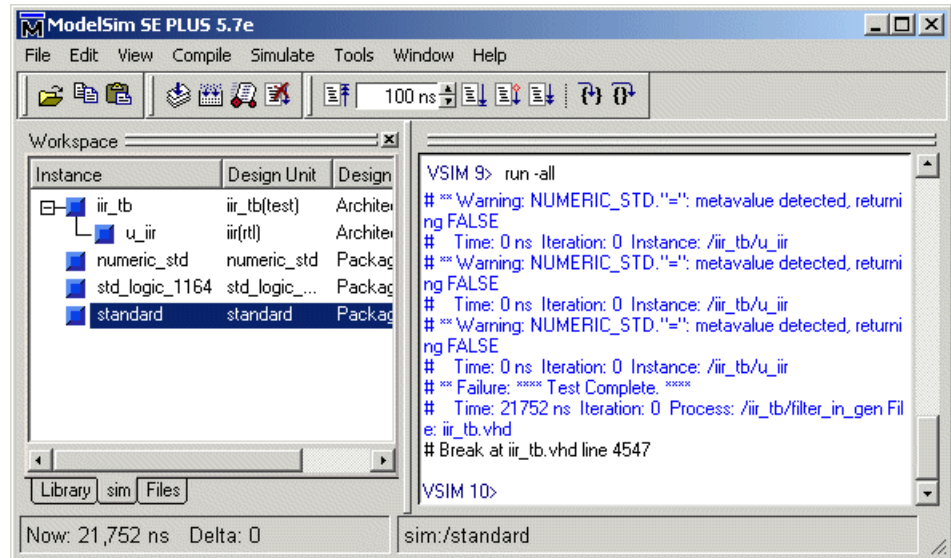


- 6 Open a display window for monitoring the simulation as the test bench runs. For example, in ModelSim, you can use the following command to open a **wave** window to view the results of the simulation as HDL waveforms:

```
add wave *
```

The following Wave window displays.



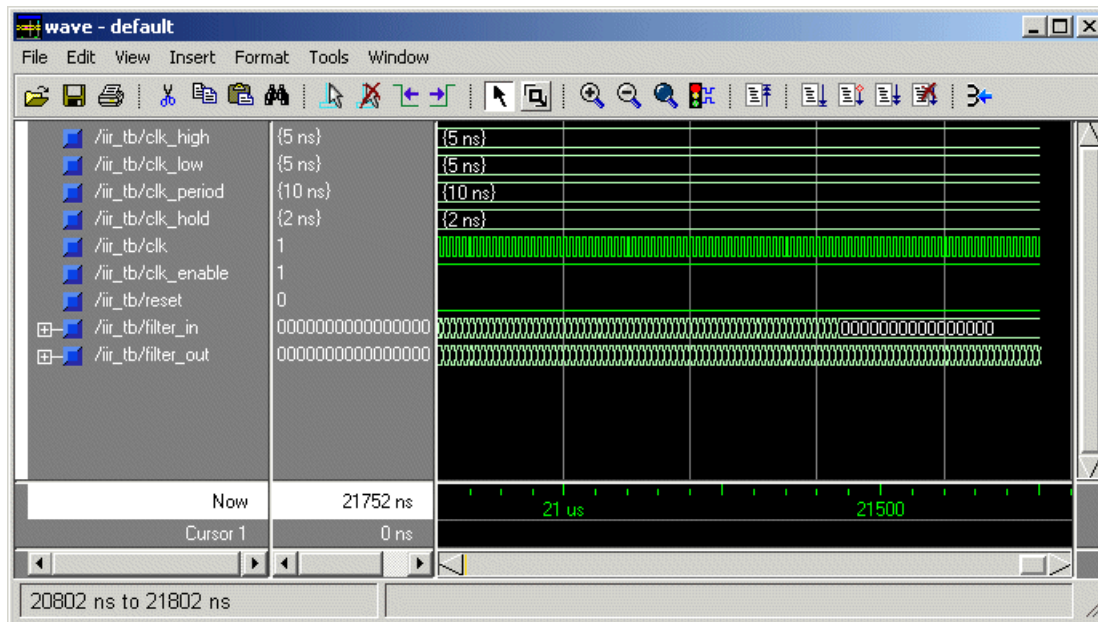


As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. You must determine whether the results are expected based on the customizations 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 successfully run to completion. The Failure part of the message is tied to the mechanism the Filter Design HDL Coder uses to end the simulation.

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



# Generating HDL Code for a Filter Design

---

The **Generate HDL** dialog is a graphical user interface (GUI) plug-in tool accessible from the Filter Design and Analysis Tool (FDATool) packaged with the Signal Processing and Filter Design Toolboxes. Using the GUI, you can quickly and easily generate HDL code and a test bench for a quantized filter you design with the FDATool. Although this chapter focuses on explaining how to use the **Generate HDL** dialog, a command line interface is also available. For descriptions of available functions and the properties you can specify in the command line, see Chapter 7, “Functions — Alphabetical List” and Chapter 6, “Properties — Alphabetical List”. Topics covered in this chapter include the following:

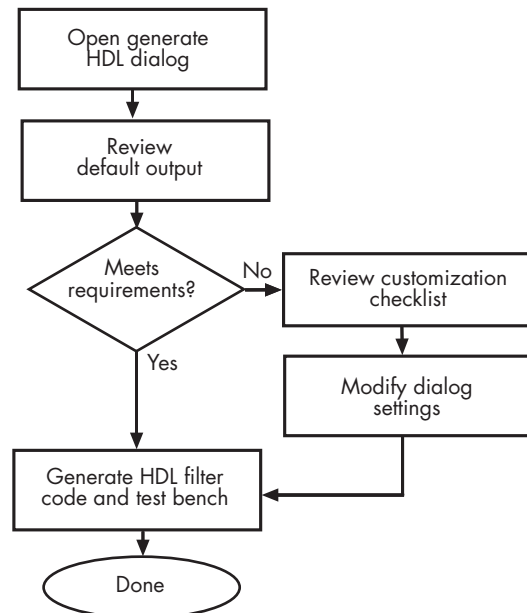
“Overview of Generating HDL Code for a Filter Design” (p. 3-3)	Provides an overview of the steps involved with using the Generate HDL dialog to generate HDL code for a filter design
“Opening the Generate HDL Dialog” (p. 3-4)	Explains how to open the Generate HDL dialog
“What Is Generated by Default?” (p. 3-9)	Describes what the Filter Design HDL Coder generates when you specify no customizations
“What Are Your HDL Requirements?” (p. 3-13)	Provides a checklist that helps you determine whether you need to specify generation customizations
“Setting the Target Language” (p. 3-18)	Explains how to specify whether VHDL or Verilog filter code is generated

“Setting the Names and Location for Generated HDL Files” (p. 3-19)	Explains how to explicitly name and specify the location for generated HDL filter and test bench files
“Customizing Reset Specifications” (p. 3-26)	Explains how to customize the names and location of generated files and specifications for resets
“Customizing the HDL Code” (p. 3-29)	Explains how to customize various elements of generated HDL code
“Setting Optimizations” (p. 3-54)	Explains how to optimize a filter’s generated HDL code, even if the resulting code might produce results that differ from results of the original MATLAB filter design
“Customizing the Test Bench” (p. 3-61)	Explains how to specify a test bench type, customize clock and reset settings, and adjust the stimulus response
“Generating the HDL Code ” (p. 3-74)	Explains how to initiate HDL code generation discusses the data type conversions that occur during the generation process

## Overview of Generating HDL Code for a Filter Design

Consider the following process as you prepare to use the **Generate HDL** dialog to generate VHDL code for your quantized filter:

- 1 Open the **Generate HDL** dialog.
- 2 Review what the Filter Design HDL Coder generates by default.
- 3 Assess whether the default settings meet your application requirements. If they do, skip to step 6.
- 4 Review the customization checklist available in “What Are Your HDL Requirements?” on page 3-13 and identify required customizations.
- 5 Modify the Generate HDL dialogs “Setting the Target Language” on page 3-18 to address your application requirements.
- 6 Generate the filter’s HDL code and test bench.



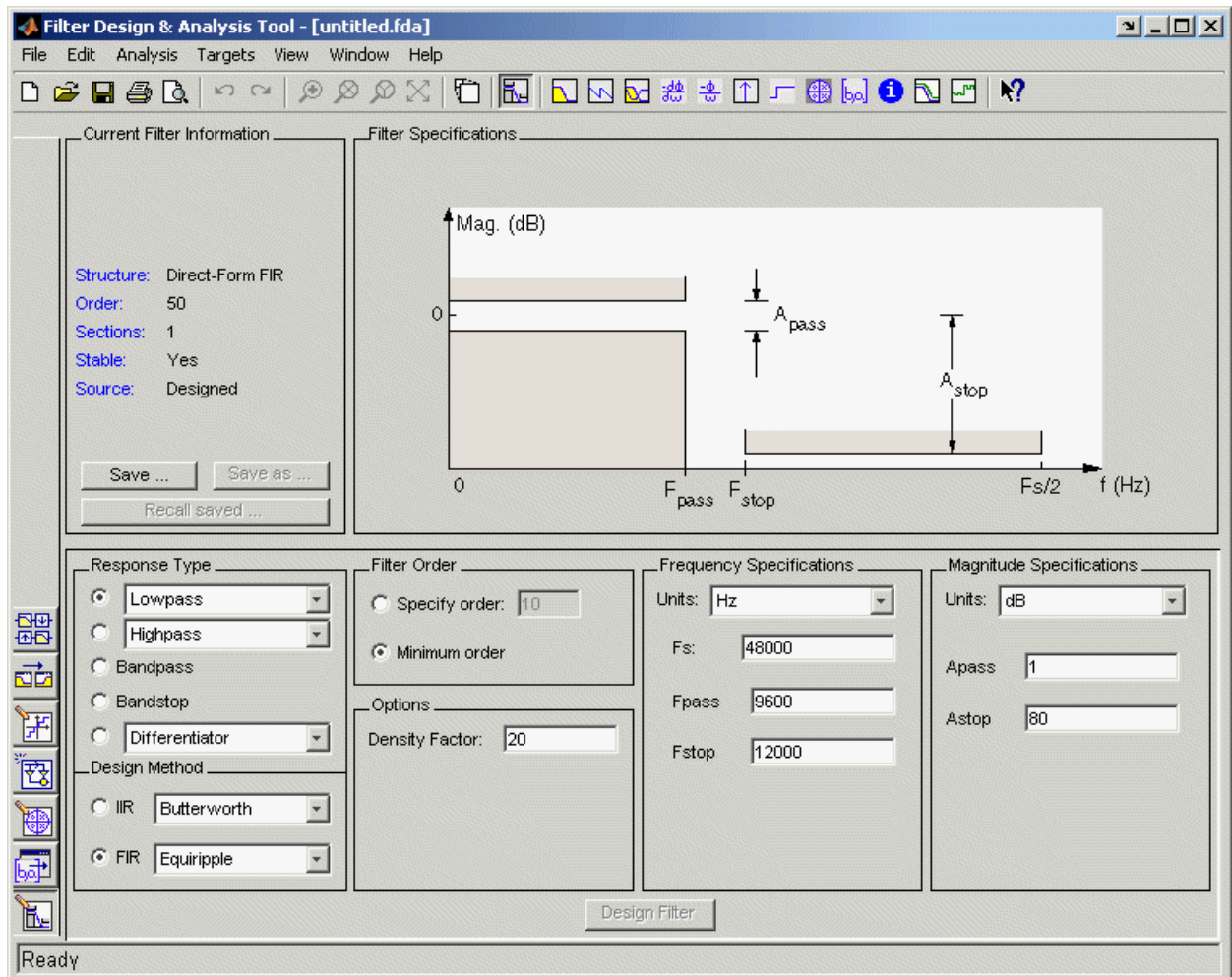
## Opening the Generate HDL Dialog

One way of customizing HDL properties and initiating the generation of HDL code is to use the **Generate HDL** dialog, which is accessible from the Filter Design and Analysis Tool (FDATool). To open the initial **Generate HDL** dialog, do the following:


- 1 Open the FDATool by entering the `fdatool` command at the MATLAB command prompt.

The FDATool displays its initial dialog.

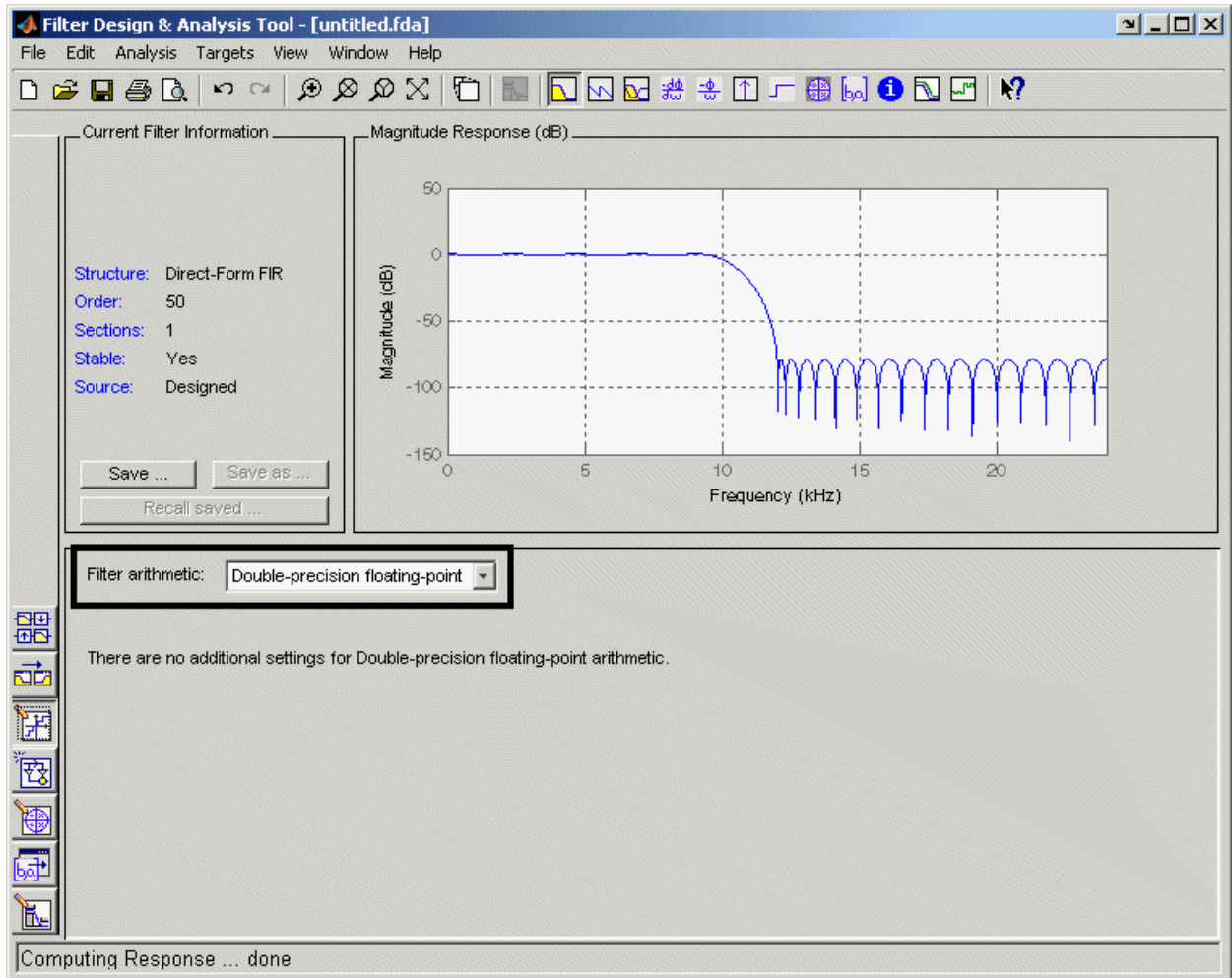




2 If the filter design is quantized, skip to step 3. Otherwise, quantize the

filter by clicking the **Set Quantization Parameters** icon . The **Filter arithmetic** menu appears in the bottom half of the dialog.

### 3 Generating HDL Code for a Filter Design

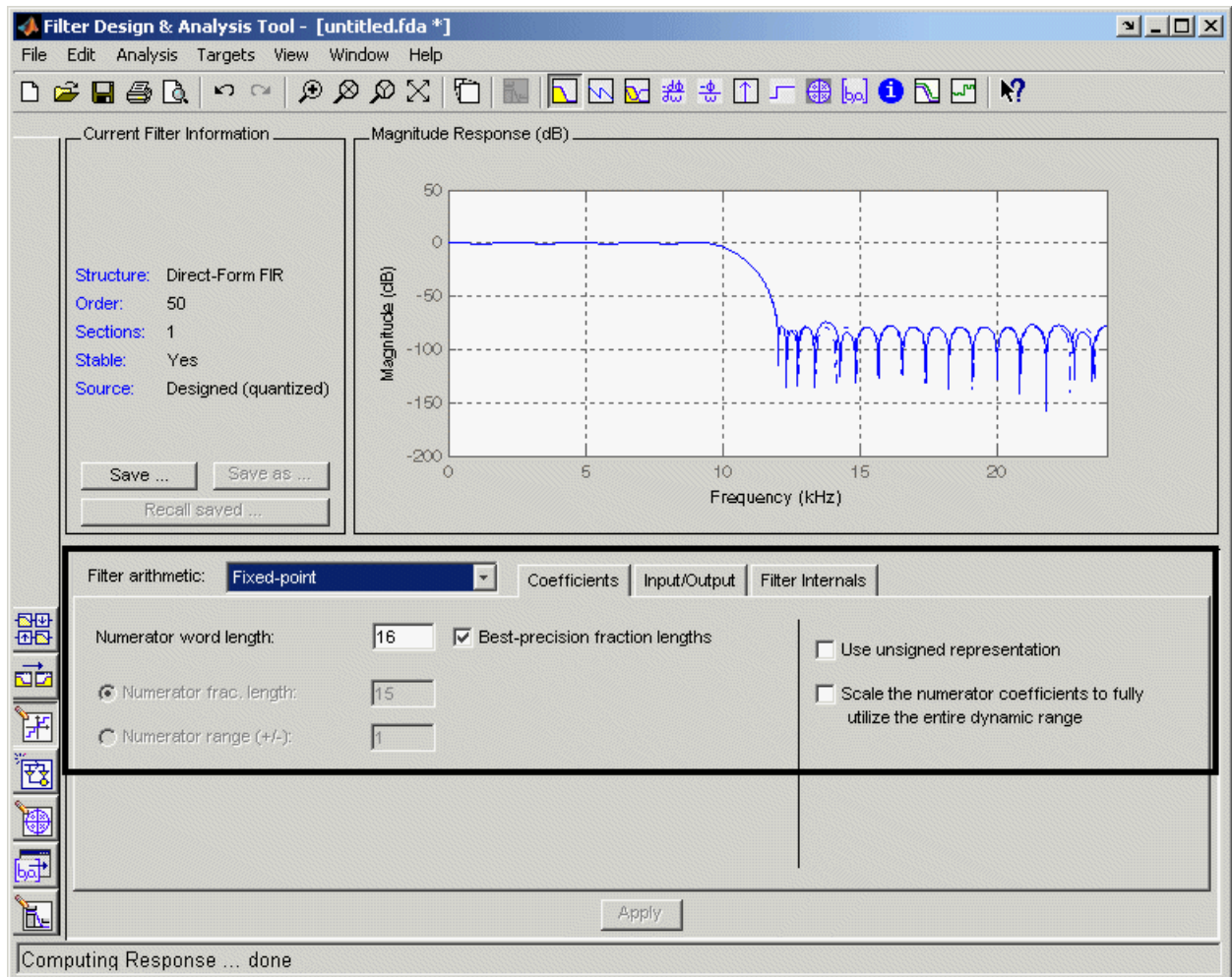


---

**Note** All supported filter structures support fixed-point, quantization type, and floating-point (double) realizations.

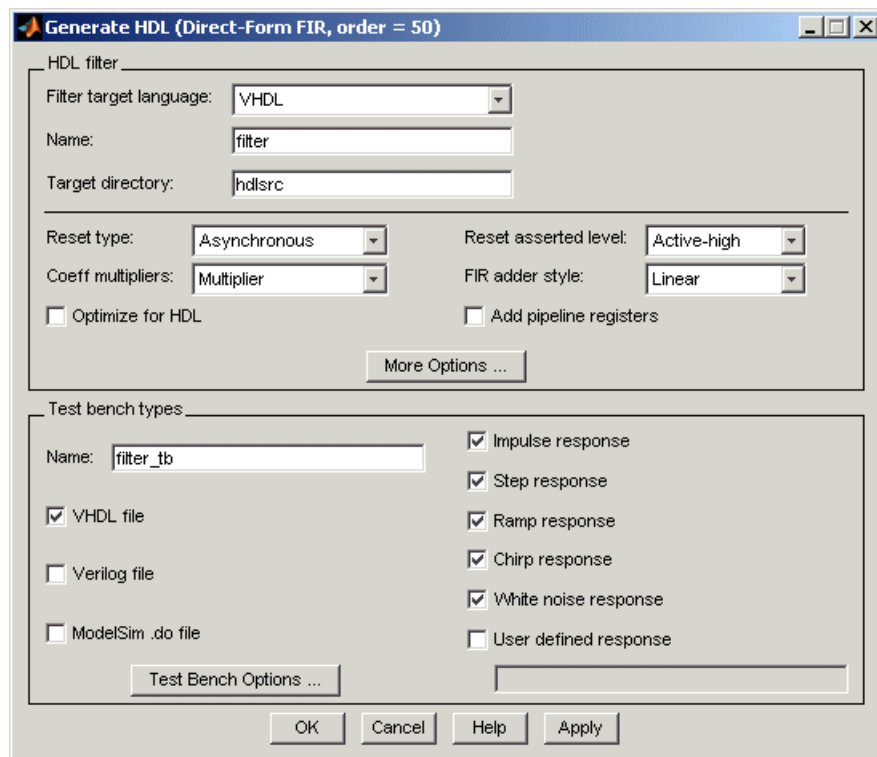
---

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

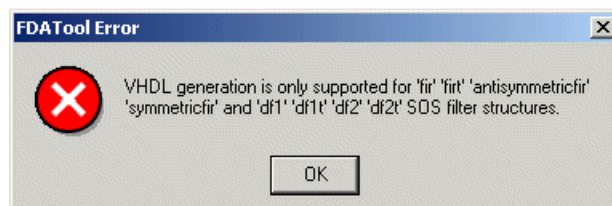


**4** Click **Targets**→**Generate HDL**. The FDATool displays the **Generate HDL** dialog.

### 3 Generating HDL Code for a Filter Design



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



## What Is Generated by Default?

The Generate HDL dialogs provide many options for you to customize the HDL code and test bench that the Filter Design HDL Coder generates. If you choose not to specify customizations, the Filter Design HDL Coder applies the default settings outlined in the following sections. Review these settings to determine whether you need to apply customizations.

- “Default Settings for Generated Files” on page 3-9
- “Default Settings for Register Resets” on page 3-10
- “Default Settings for General HDL Code” on page 3-10
- “Default Settings for Code Optimizations” on page 3-11
- “Default Settings for Test Benches” on page 3-12

### Default Settings for Generated Files

By default, the Filter Design HDL Coder

- Generates the following files, where *Hd* is the name of the quantized filter:

Language	File	Name
Verilog	Filter source	<i>Hd.v</i>
	Filter test bench	<i>Hd_tb.v</i>
VHDL	Filter source	<i>Hd.vhd</i>
	Package (if needed)	<i>Hd_pkg.vhd</i>
	Test bench	<i>Hd_tb.vhd</i>

- Places generated files in a subdirectory named `hdlsrc`, under your current working directory.
- Includes VHDL entity and architecture code in a single source file.

For information on modifying these settings, see “What Are Your HDL Requirements?” on page 3-13 and “Setting the Names and Location for Generated HDL Files” on page 3-19.

## Default Settings for Register Resets

By default, the Filter Design HDL Coder

- Uses an asynchronous reset when generating HDL code for registers.
- Uses an active-high (1) signal for register resets.

For information on modifying these settings, see “What Are Your HDL Requirements?” on page 3-13 and “Customizing Reset Specifications” on page 3-26.

## Default Settings for General HDL Code

By default, the Filter Design HDL Coder

- Names the generated VHDL entity or Verilog module with the name of the quantized filter.
- Names a filter’s HDL ports as follows:

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

- Sets the data types for HDL ports as follows:

<b>HDL Port</b>	<b>VHDL Type</b>	<b>Verilog Type</b>
Clock input	STD_LOGIC	wire
Clock enable input	STD_LOGIC	wire
Reset	STD_LOGIC	wire
Data input	STD_LOGIC_VECTOR	wire
Data output	STD_LOGIC_VECTOR	wire

- Names coefficients as follows:

<b>For...</b>	<b>Names Coefficients...</b>
FIR filters	<code>coeff<math>n</math></code> , where $n$ is the coefficient number, starting with 1
IIR filters	<code>coeff_<math>xm</math>_section<math>n</math></code> , where $x$ is a or b, $m$ 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 any file that instantiates a component.
- In VHDL, uses a type safe representation when concatenating zeros: '0' & '0'..
- In VHDL, applies the statement `ELSIF clk'event AND clk='1'` THEN to check for clock events.
- In Verilog, uses time scale directives.
- Allows a minimum of 3 bits of filter input and coefficient scale values to overlap before a warning is issued.
- Adds an extra input register and an extra output register to the filter code.
- Appends `_process` to process names.
- When creating labels for VHDL GENERATE statements:
  - Appends `_gen` to VHDL section and block names
  - Names VHDL output assignment blocks with the string `outputgen`

For information on modifying these settings, see “What Are Your HDL Requirements?” on page 3-13 and “Customizing the HDL Code” on page 3-29.

## Default Settings for Code Optimizations

By default, the Filter Design HDL Coder disables most optimizations. The coder

- Generates HDL code that is bit-true to the original MATLAB filter function and is *not* optimized for performance or space requirements.

- Applies a linear final summation to FIR filters. This is the form of summation explained in most DSP text books.
- Enables multiplier operations for a filter, as opposed to replacing them with additions of partial products.

For information on modifying these settings, see “What Are Your HDL Requirements?” on page 3-13 and “Setting Optimizations” on page 3-54.

## Default Settings for Test Benches

By default, the Filter Design HDL Coder generates a VHDL test bench that inherits all the HDL settings that are applied to the filter’s HDL code. In addition, the coder generates a test bench that

- Is named `filter_tb.vhd`.
- Forces clock, clock enable, and reset input signals.
- Forces clock enable and reset input signals to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces reset signals for two cycles plus the hold time.
- Applies a hold time of 2 nanoseconds to filter reset and data input signals.
- Applies the following stimulus response types:

### For Filters...

FIR, FIRT, Symmetric FIR, and Antisymmetric FIR

All others

### Applies Response Types...

Impulse, step, ramp, chirp, and white noise

Step, ramp, and chirp

For information on modifying these settings, see “What Are Your HDL Requirements?” on page 3-13 and “Customizing the Test Bench” on page 3-61.



## What Are Your HDL Requirements?

As part of the process of generating HDL code for a filter designed in the MATLAB environment, review the following checklist. The checklist will help you determine whether you need to adjust any of the HDL property settings. If your answer to any of the questions in the checklist is “yes,” go to the topic listed in the second column of the table for information on how to adjust the property setting to meet your project’s HDL requirements.

### HDL Requirements Checklist

Requirement	For More Information, See...
<b>Language Selection</b>	
<input type="checkbox"/> Do you need to adjust the target language setting?	“Setting the Target Language” on page 3-18
<b>File Naming and Location Specifications</b>	
<input type="checkbox"/> Do you want to specify a unique <b>name</b> , which does <i>not</i> match the name of the quantized filter, for the VHDL <b>entity</b> or Verilog <b>module</b> that represents the filter?	“Setting the Names and Location for Generated HDL Files” on page 3-19
<input type="checkbox"/> Do you want the <b>file type extension</b> for generated HDL files to be a string other than .vhd for VHDL or .v for Verilog?	“Setting the Names and Location for Generated HDL Files” on page 3-19
<b>Reset Specifications</b>	
<input type="checkbox"/> Do you want to use <b>synchronous resets</b> ?	“Setting the Reset Style for Registers” on page 3-26
<input type="checkbox"/> Do you need the <b>asserted level of the reset</b> signal to be low (0)?	“Setting the Asserted Level for the Reset Input Signal” on page 3-28
<b>Header Comment and General Naming Specifications</b>	
<input type="checkbox"/> Do you want to add a specific string, such as a revision control string, to the end of the <b>header comment block</b> in each generated file?	“Specifying a Header Comment” on page 3-30
<input type="checkbox"/> Do you want a string other than <code>coeff</code> to be used as the <b>base filter coefficient name</b> ?	“Specifying a Prefix for Filter Coefficients” on page 3-32

### HDL Requirements Checklist (Continued)

Requirement	For More Information, See...
<input type="checkbox"/> If your filter design requires a <b>VHDL package file</b> , do you want the name of the generated file to include a string other than <code>_pkg</code> ?	“Setting the Postfix String for VHDL Package Files” on page 3-22
<input type="checkbox"/> Do you want a string other than <code>_entity</code> to be appended to VHDL entity or Verilog module names if <b>duplicate names</b> are detected?	“Setting the Postfix String for Resolving Entity or Module Name Conflicts” on page 3-33
<input type="checkbox"/> Do you want a string other than <code>_rsvd</code> to be appended to specified names and labels that are <b>HDL reserved words</b> ?	“Setting the Postfix String for Resolving HDL Reserved Word Conflicts” on page 3-34
<input type="checkbox"/> Do you want a string other than <code>_process</code> to be appended to <b>HDL process names</b> ?	“Setting the Postfix String for Process Block Labels” on page 3-37
<input type="checkbox"/> Do you want the Filter Design HDL Coder to write the <b>entity</b> and <b>architecture</b> parts of generated VHDL code to <b>separate files</b> ?	“Splitting Entity and Architecture Code into Separate Files” on page 3-23
<input type="checkbox"/> If the Filter Design HDL Coder writes the <b>entity</b> and <b>architecture</b> parts of VHDL code to separate files, do you want strings other than <code>_entity</code> and <code>_arch</code> included in the <b>filenames</b> ?	“Splitting Entity and Architecture Code into Separate Files” on page 3-23
<b>Port Specifications</b>	
<input type="checkbox"/> Do you want the Filter Design HDL Coder to use strings other than <code>filter_in</code> and <code>filter_out</code> to name <b>HDL ports</b> for the filter’s data input and output signals?	“Naming HDL Ports” on page 3-38
<input type="checkbox"/> Do you need the Filter Design HDL Coder to declare the filter’s <b>data input and output ports</b> with a <b>VHDL type</b> other than <code>STD_LOGIC_VECTOR</code> ?	“Specifying the HDL Data Type for Data Ports” on page 3-40
<input type="checkbox"/> Do you want the Filter Design HDL Coder to use strings other than <code>clk</code> and <code>clk_enable</code> to name HDL ports for the filter’s <b>clock and clock enable</b> input signals?	“Naming HDL Ports” on page 3-38
<input type="checkbox"/> Do you want the Filter Design HDL Coder to use a string other than <code>reset</code> to name an HDL port for the filter’s <b>reset</b> input signals?	“Naming HDL Ports” on page 3-38

## HDL Requirements Checklist (Continued)

Requirement	For More Information, See...
<input type="checkbox"/> Do you want the Filter Design HDL Coder to add an <b>extra input or output register</b> to support the filter's HDL input and output ports?	"Suppressing Extra Input and Output Registers" on page 3-42
<b>Advanced Coding Specifications</b>	
<input type="checkbox"/> Do you expect the filter's <b>coefficient scale values</b> to be more than 3 bits smaller than the size of the filter's input?	"Minimizing Quantization Noise for Fixed-Point Filters" on page 3-43
<input type="checkbox"/> Do you want the Filter Design HDL Coder to represent all <b>constants as aggregates</b> ?	"Representing Constants with Aggregates" on page 3-45
<input type="checkbox"/> Are you using an EDA tool that does not support <b>loops</b> ? Do you need the Filter Design HDL Coder to unroll and remove VHDL FOR and GENERATE loops?	"Unrolling and Removing VHDL Loops" on page 3-46
<input type="checkbox"/> Do you want the Filter Design HDL Coder to use the VHDL <code>rising_edge</code> function to check for <b>rising edges</b> when the filter is operating on registers?	"Using the VHDL <code>rising_edge</code> Function" on page 3-47
<input type="checkbox"/> Do you want to suppress Verilog <b>time scale directives</b> ?	"Suppressing Verilog Time Scale Directives" on page 3-50
<input type="checkbox"/> Do you want the Filter Design HDL Coder to omit <b>configurations</b> from generated VHDL code? Are you going to create and store the filter's VHDL configurations in separate VHDL source files?	"Suppressing the Generation of VHDL Inline Configurations" on page 3-48
<input type="checkbox"/> Do you want the Filter Design HDL Coder to use the VHDL syntax "000000..." to represent <b>concatenated zeros</b> instead of the type safe representation '0' & '0'?	"Specifying VHDL Syntax for Concatenated Zeros" on page 3-49
<input type="checkbox"/> Do you want to suppress the initialization of <b>signals of type REAL</b> to 0.0?	"Suppressing the Initialization of Signals of Type REAL" on page 3-51
<input type="checkbox"/> Do you want the Filter Design HDL Coder to apply typical DSP processor treatment of input data types when generating code for <b>addition and subtraction operations</b> ?	"Specifying Input Type Treatment for Addition and Subtraction Operations" on page 3-52

### HDL Requirements Checklist (Continued)

Requirement	For More Information, See...
<b>Optimization Specifications</b>	
<input type="checkbox"/> Do you need <b>numeric results</b> optimized, even if the results are not bit-true to the MATLAB filter function?	“Optimizing Generated Code for HDL” on page 3-55
<input type="checkbox"/> Do you want the Filter Design HDL Coder to replace <b>multiplier operations</b> by applying canonic signed digit (CSD) and factored CSD techniques?	“Optimizing Coefficient Multipliers” on page 3-55
<input type="checkbox"/> Do you need the Filter Design HDL Coder to optimize the <b>final summation for FIR filters</b> ?	“Optimizing Final Summation for FIR Filters” on page 3-57
<input type="checkbox"/> Do you want to optimize your filters <b>clock rate</b> ?	“Optimizing the Clock Rate with Pipeline Registers” on page 3-58
<b>Test Bench Specifications</b>	
<input type="checkbox"/> Do you want the <b>name</b> of the generated <b>test bench</b> file to include a string other than <code>_tb</code> ?	“Setting the Names and Location for Generated HDL Files” on page 3-19
<input type="checkbox"/> Do you want to generate a <b>VHDL test bench</b> ?	“Specifying a Test Bench Type” on page 3-62
<input type="checkbox"/> Do you want to generate a <b>Verilog file test bench</b> ?	“Specifying a Test Bench Type” on page 3-62
<input type="checkbox"/> Do you want to generate a <b>ModelSim DO file test bench</b> ?	“Specifying a Test Bench Type” on page 3-62
<input type="checkbox"/> If the test bench type is a ModelSim DO file, does your application require you to specify any <b>simulation flags</b> ?	“Specifying a Test Bench Type” on page 3-62
<input type="checkbox"/> Are you using a user-defined external source to <b>force clock enable</b> input signals to a constant value?	“Configuring the Clock” on page 3-65
<input type="checkbox"/> If the test bench is to force clock enable input signals, do you want it to force the signals to <b>active low</b> (0)?	“Configuring the Clock” on page 3-65
<input type="checkbox"/> Are you using a user-defined external source to <b>force clock</b> input signals?	“Configuring the Clock” on page 3-65

### HDL Requirements Checklist (Continued)

Requirement	For More Information, See...
<input type="checkbox"/> If the test bench is to force clock input signals, do you want the signals to be <b>driven high or low</b> for a <b>duration</b> other than 5 nanoseconds?	“Configuring the Clock” on page 3-65
<input type="checkbox"/> Are you using a user-defined external source to <b>force reset</b> input signals?	“Configuring Resets” on page 3-67
<input type="checkbox"/> If the test bench is to force reset input signals, do you want it to force the signals to <b>active low</b> (0)?	“Configuring Resets” on page 3-67
<input type="checkbox"/> If the test bench is to force <b>reset</b> input signals, do you want it to apply a <b>hold time</b> other than two cycles plus a hold time of 2 nanoseconds?	“Configuring Resets” on page 3-67
<input type="checkbox"/> Do you want to apply a <b>hold time</b> other than 2 nanoseconds to <b>filter data</b> input signals?	“Setting a Hold Time for Data Input Signals” on page 3-69
<input type="checkbox"/> Do you want to customize the <b>stimulus</b> to be applied by the test bench?	“Setting Test Bench Stimuli” on page 3-72

### Setting the Target Language

By default, the Filter Design HDL Coder generates VHDL code for a filter. If you retain the VHDL setting, **Generate HDL** dialog options that are specific to Verilog are greyed out and are not selectable.

If you require or prefer to generate Verilog code, select Verilog for the **Filter target language** option in the **HDL filter** pane of the **Generate HDL** dialog. This setting causes the coder to enable options that are specific to Verilog and to grey 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.

## Setting the Names and Location for Generated HDL Files

By default, the Filter Design HDL Coder creates the HDL files listed in the following table and places them in subdirectory `hdlsrc` under your current working directory. The Filter Design HDL Coder derives HDL filenames 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 filenames based on filter name `Hq`.

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

The Filter Design HDL 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`.

By default, the Filter Design HDL Coder includes the code for a filter's VHDL entity and architectures in the same VHDL source file. Alternatively, you can specify that the Filter Design HDL Coder write the generated code for the entity and architectures to separate files. For example, if the filter name is `Hd`, the Filter Design HDL Coder writes the VHDL code for the filter to files `Hd_entity.vhd` and `Hd_arch.vhd`.

The following sections explain how to adjust the preceding default settings.

- “Setting Filter Entity and General File Naming Strings” on page 3-20

- “Redirecting Filter Design HDL Coder Output” on page 3-21
- “Setting the Postfix String for VHDL Package Files” on page 3-22
- “Splitting Entity and Architecture Code into Separate Files” on page 3-23

## Setting Filter Entity and General File Naming Strings

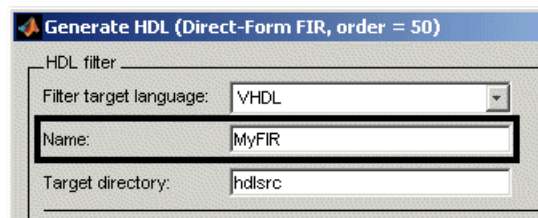
To set the string that the Filter Design HDL Coder uses to name the filter entity or module and generated files, specify a new value in the **Name** text field of the **HDL filter** pane of the **Generate HDL** dialog. The Filter Design HDL Coder uses the **Name** string to

- 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

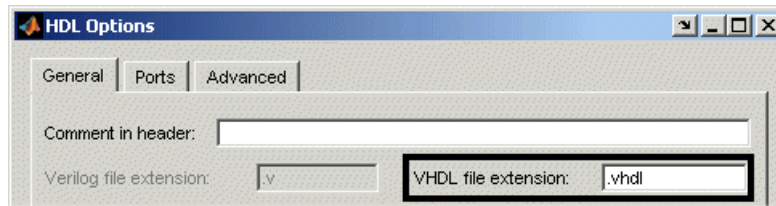
By default, the filter HDL files are generated with a .vhd or .v file extension, depending on the language selection. To change the file extension,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog.
- 2 Click the **General tab** on the **HDL Options** dialog.
- 3 Type the new file extension in the **Verilog file extension** or **VHDL file extension** text field.
- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

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








---

**Note** When specifying strings for filenames and file type extensions, consider platform-specific requirements and restrictions. Also consider postfix strings the Filter Design HDL Coder appends to the **Name** string, such as `_tb` and `_pkg`.

---

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

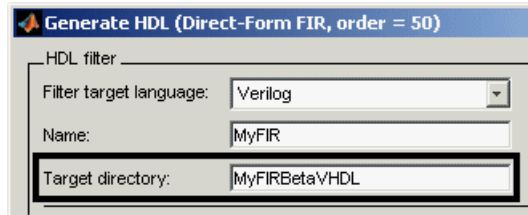
## Redirecting Filter Design HDL Coder Output

By default, the Filter Design HDL Coder places all generated HDL files in subdirectory `hdlsrc` under your current working directory. To direct Filter Design HDL Coder output to a directory other than the default target directory, specify a new directory in the **Target directory** text field in the **HDL filter** pane of the **Generate HDL** dialog. If you specify a directory that does not exist, the Filter Design HDL Coder creates the directory for you before depositing the generated files. Your directory specification can be one of the following:

- Directory name. In this case, the Filter Design HDL Coder looks for, and if necessary, creates a subdirectory under your current working directory.
- An absolute path to a directory under your current working directory. If necessary, the Filter Design HDL Coder creates the specified directory.
- A relative path to a higher level directory under your current working directory. For example, if you specify `../../../../myfiltvhd`, the Filter Design HDL Coder checks whether a directory named `myfiltvhd` exists

three levels up from your current working directory, creates the directory if it does not exist, and writes all generated HDL files to that directory.

The following dialog sets the directory to MyFIRBetaVHDL.



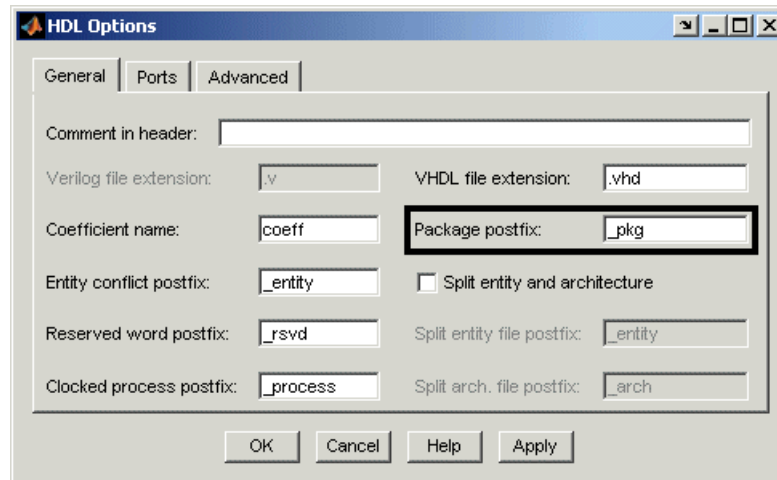
This change instructs the Filter Design HDL Coder to create the subdirectory MyFIRBetaVHDL under the current working directory and write generated HDL files to that directory.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the `TargetDirectory` property to redirect Filter Design HDL Coder output.

### Setting the Postfix String for VHDL Package Files

By default, the Filter Design HDL Coder appends the postfix `_pkg` to the base filename when generating a VHDL package file. To rename the postfix string for package files, do the following:

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Specify a new value in the **Package postfix** text field.




---

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

---

- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the `PackagePostfix` property to rename the filename postfix for VHDL package files.

## Splitting Entity and Architecture Code into Separate Files

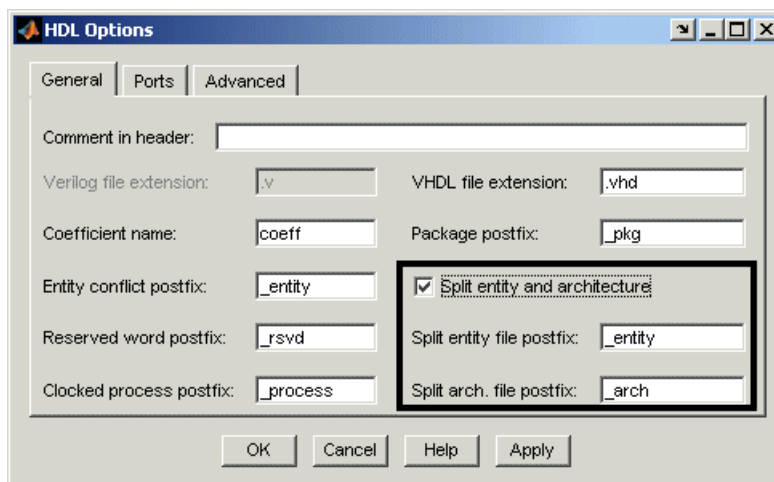
By default, the Filter Design HDL Coder includes a filter's VHDL entity and architecture code in the same generated VHDL file. Alternatively, you can instruct the Filter Design HDL Coder to place the entity and architecture code in separate files. For example, instead of all generated code residing in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

The Filter Design HDL Coder derives the names of the entity and architecture files from

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

To split the filter source file, do the following:

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Select **Split entity and architecture**. The Filter Design HDL Coder enables the text field options **Split entity file postfix** and **Split arch. file postfix**.



- 4 Specify new strings in the postfix text fields if you want the Filter Design HDL Coder 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 filenames, consider the size of the base name and platform-specific file naming requirements and restrictions.

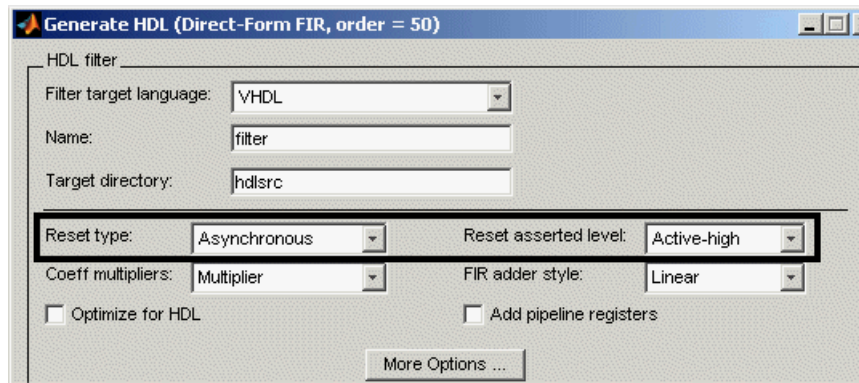
---

- 5 Click **Apply** to register the change or **OK** to register the change and close the dialog.

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

## Customizing Reset Specifications

Reset options appear in the lower half of the **HDL filter** pane of the **Generate HDL** dialog, as highlighted in the screen display below.



Use the reset options for

- “Setting the Reset Style for Registers” on page 3-26
- “Setting the Asserted Level for the Reset Input Signal” on page 3-28

## Setting the Reset Style for Registers

By default, the Filter Design HDL Coder uses an asynchronous reset style when generating HDL code for registers. Whether you should set the style 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.

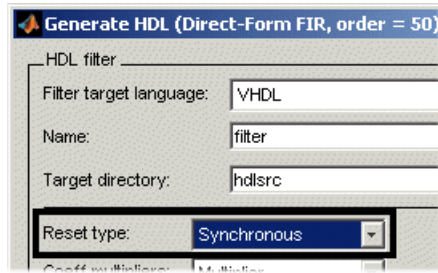
```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
  ELSIF clk'event AND clk = '1' THEN
```

```

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 style to synchronous, select Synchronous from the **Reset type** menu in the **HDL filter** pane of the **Generate HDL** dialog.



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;

```

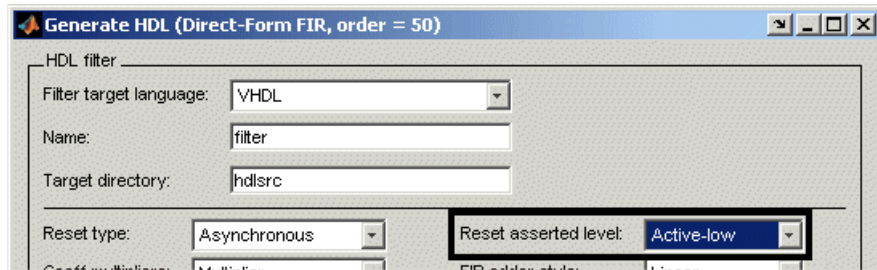
**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `ResetType` to set the reset style for your filter's registers.

## 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 Filter Design HDL 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 **HDL filter** pane of the **Generate HDL** dialog.



With this change, the IF statement in the preceding generated code changes to

```
IF reset = '0' THEN
```

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `ResetAssertedLevel` to set the asserted level for the filter's reset input signal.

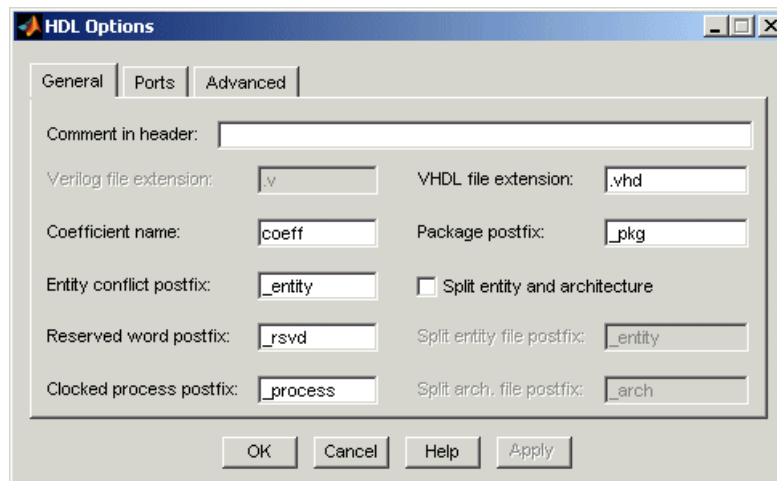


## Customizing the HDL Code

You select most HDL code customizations from options on the **HDL Options** dialog. Options that are specific to VHDL or Verilog are active only if that language is selected. Inactive options appear grey and are not selectable. An option may also appear inactive if it is dependent on the selection of another option.

Options provided by the **HDL Options** dialog are categorized into three tabs: **General**, **Ports**, and **Advanced**.

The following dialog shows general options that are active for VHDL.



Note that the **Verilog file extension** option is inactive due to the VHDL language selection. The **Split entity file postfix** and **Split arch. file postfix** options are inactive due to a dependency on the setting of **Split entity and architecture**.

The following sections explain how to use this dialog to specify naming, port, and advanced coding customizations:

- “Specifying a Header Comment” on page 3-30
- “Specifying a Prefix for Filter Coefficients” on page 3-32

- “Setting the Postfix String for Resolving Entity or Module Name Conflicts” on page 3-33
- “Setting the Postfix String for Resolving HDL Reserved Word Conflicts” on page 3-34
- “Setting the Postfix String for Process Block Labels” on page 3-37
- “Naming HDL Ports” on page 3-38
- “Specifying the HDL Data Type for Data Ports” on page 3-40
- “Suppressing Extra Input and Output Registers” on page 3-42
- “Minimizing Quantization Noise for Fixed-Point Filters” on page 3-43
- “Representing Constants with Aggregates” on page 3-45
- “Unrolling and Removing VHDL Loops” on page 3-46
- “Using the VHDL rising\_edge Function” on page 3-47
- “Suppressing the Generation of VHDL Inline Configurations” on page 3-48
- “Specifying VHDL Syntax for Concatenated Zeros” on page 3-49
- “Suppressing Verilog Time Scale Directives” on page 3-50
- “Suppressing the Initialization of Signals of Type REAL” on page 3-51
- “Specifying Input Type Treatment for Addition and Subtraction Operations” on page 3-52

## Specifying a Header Comment

The Filter Design HDL Coder includes a header comment block, such as the following, at the top of the files it generates:

```
-----  
--  
-- Module:Hd  
--  
-- Generated by MATLAB(R) 7.0 and the Filter Design HDL Coder 1.0.  
--  
-- Generated on: 2004-02-04 09:42:43  
--  
-----
```

You can use the **Comment in header** option to add a comment string, such as a revision control string, to the end of the header comment block in each generated file. For example, you might use this option to add the revision control tag `$Revision: 1.1.4.24.2.1 $`. With this change, the preceding header comment block would appear as follows:

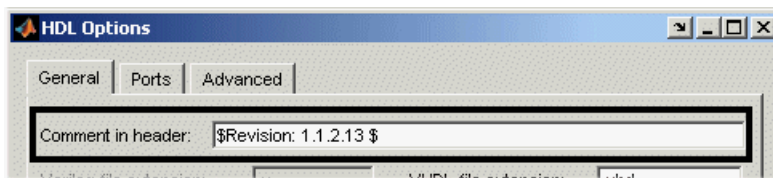
```

-----
--
-- Module:Hd
--
-- Generated by MATLAB(R) 7.0 and the Filter Designer HDL Coder 1.0.
--
-- Generated on: 2004-02-04 09:42:43
--
-- $Revision: 1.1.4.24.2.1 $
-----

```

To add a header comment,

- 1** Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2** Click the **General** tab. General HDL coding options appear.
- 3** Type the comment string in the **Comment in header** text box, as shown in the following display.



- 4** Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `UserComment` to add a comment string to the end of the header comment block in each generated HDL file.

## Specifying a Prefix for Filter Coefficients

The Filter Design HDL 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: `coeff1`, `coeff22`

IIR filters An underscore (`_`) and an `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) followed by the string `_sectionn`, where `n` is the section number.

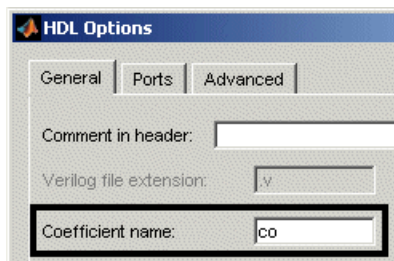
Example: `coeff_b1_section3` (first numerator coefficient of the third section)

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 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Enter a new string in the **Coefficient name** text box, as shown in the following display.



The string that you specify

- Must start with a letter.
- Cannot end with an underscore (\_)
- Cannot include a double underscore (\_\_)

---

**Note** If you specify a VHDL or Verilog reserved word, the Filter Design HDL 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`.

---

- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

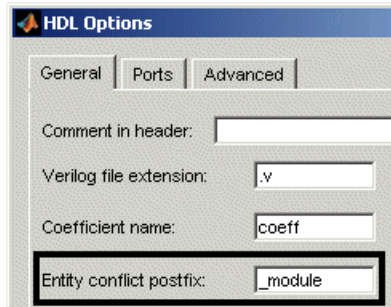
**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `CoeffName` to change the base name for filter coefficients.

## Setting the Postfix String for Resolving Entity or Module Name Conflicts

The Filter Design HDL Coder checks whether multiple entities in VHDL or multiple modules in Verilog share the same name. If a name conflict exists, the Filter Design HDL Coder appends the postfix `_entity` to the second of the two matching strings.

To change the postfix string that the Filter Design HDL Coder applies,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Enter a new string in the **Entity conflict postfix** text box, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `EntityConflictPostfix` to change the entity or module conflict postfix string.

## Setting the Postfix String for Resolving HDL Reserved Word Conflicts

The Filter Design HDL Coder checks whether any strings that you specify as names, postfix values, or labels are VHDL or 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

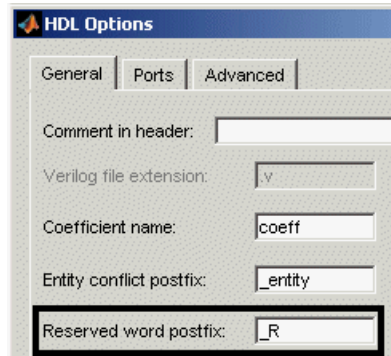
join	large	liblist	library	localparam
macromodule	medium	module	nand	negedge
nmos	nor	noshowcancelled	not	notif0
notif1	or	output	parameter	pmos
posedge	primitive	pull0	pull1	pulldown
pullup	pulsestyle_oneevent	pulsestyle_ondetect	rcmos	real
realtime	reg	release	repeat	rnmos
rpmos	rtran	rtranif0	rtranif1	scalared
showcancelled	signed	small	specify	specparam
strong0	strong1	supply0	supply1	table
task	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
triereg	unsigned	use	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

If you specify a reserved word, the Filter Design HDL Coder appends the postfix `_rsvd` to the string. For example, if you try to name your filter `mod`, for VHDL code, the Filter Design HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

To change the postfix string that the Filter Design HDL Coder applies,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Enter a new string in the **Reserved word postfix** text box, as shown in the following display.





- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `ReservedWordPostfix` to change the reserved word postfix string.

## Setting the Postfix String for Process Block Labels

The Filter Design HDL 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 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;

```

You have the option of setting the postfix string to a value other than `_process`. For example, you might change it to `_clkproc`. To change the string,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **General** tab.
- 3 Enter a new string in the **Clocked process postfix** text box, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `ClockProcessPostfix` to change the postfix string appended to process labels.

## Naming HDL Ports

By default, the Filter Design HDL Coder names a filter's HDL ports as follows:

<b>HDL Port</b>	<b>Default Port Name</b>
Input port	filter_in
Output port	filter_out
Clock port	clk
Clock enable port	clk_enable
Reset port	reset

For example, the default VHDL declaration for entity Hd looks like the following:

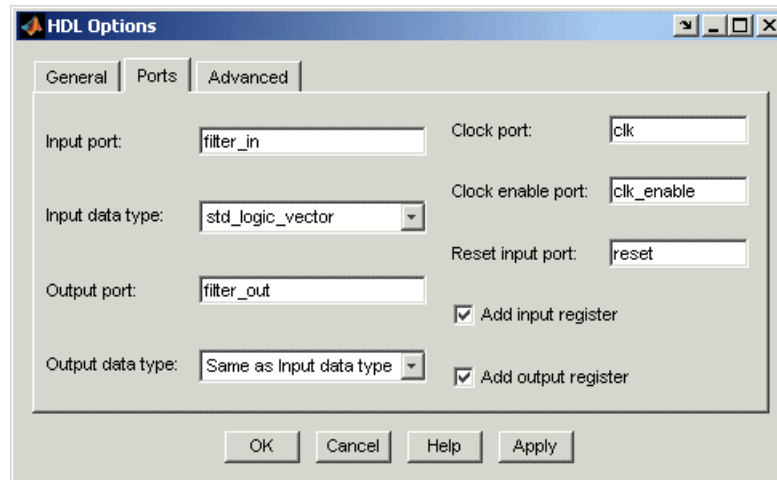
```

ENTITYHd 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
          filter_out    :    OUT  std_logic_vector (15 DOWNT0 0); -- sfix16_En15
        );
ENDHd;

```

To change any of the port names,

- 1** Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2** Click the **Ports** tab. Port options appear, as shown in the following display.



3 Enter new strings in the following text boxes, as necessary:

- **Input port**
- **Output port**
- **Clock port**
- **Clock enable port**
- **Reset input port**

4 Click **Apply** to register the changes or **OK** to register the changes and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the properties `InputPort`, `OutputPort`, `ClockInputPort`, `ClockEnableInputPort`, and `ResetInputPort` to change the names of a filter's VHDL ports.

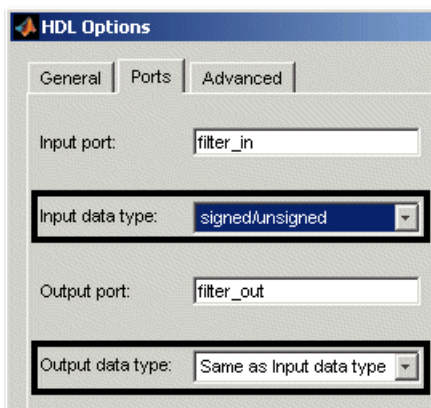
## Specifying the HDL Data Type for Data Ports

By default, the Filter Design HDL Coder declares a filter's input and output data ports to be of 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 Filter Design HDL 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 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Ports** tab. Port options appear.
- 3 Select a data type from the **Input data type** or **Output data type** menu identified in the following display. The type for Verilog ports is always wire.




---

**Note** The setting of **Input data type** does not affect double-precision input, which is always generated as type REAL for VHDL and wire[63:0] for Verilog.

---

- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the generatehdl and generatetb functions with the properties InputType and OutputType to change the VHDL data type for a filter's input and output ports.

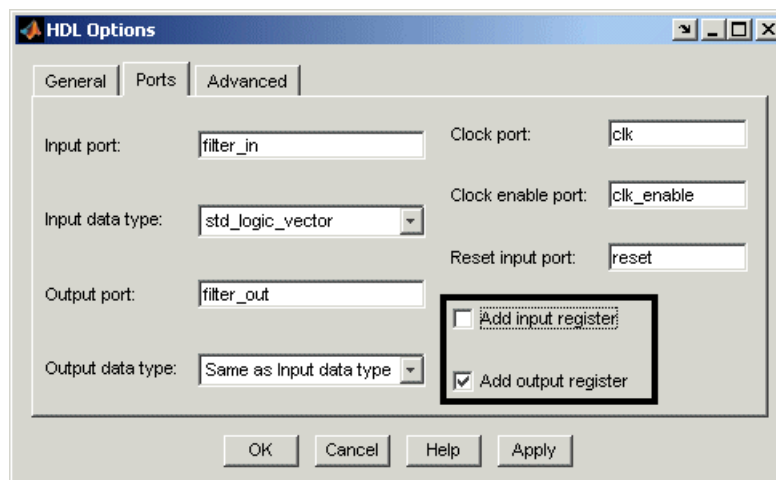
## Suppressing Extra Input and Output Registers

The Filter Design HDL 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 have no timing requirements, you can suppress generation of the extra registers as follows:

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Ports** tab. Port options appear.
- 3 Clear **Add input register** and **Add output register** per your requirements. The following display shows the setting for suppressing the generation of an extra input register.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

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

## Minimizing Quantization Noise for Fixed-Point Filters

For fixed-point filters, an option is available for controlling whether the coder generates a warning for scale values that are below a specified numeric threshold relative to the input data format. These warnings help identify scale values that cause the input range to be quantized to near zero, adding quantization noise.

You can control the warnings by specifying an overlap threshold. The coder temporarily converts a scale value to the data type of the filter input. Then, the coder checks whether the number of leading zeros in the converted value is greater than or equal to the specified overlap threshold. If this condition exists, the coder generates a warning.

You can prevent the coder from generating these warnings by setting the minimum overlap to the number of bits in the input format. However, if

the converted scale value equals zero, the coder reports an error because the input range is quantized away.

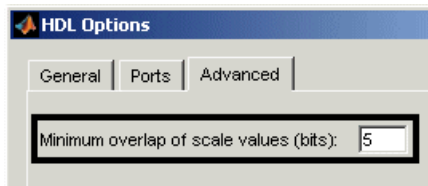
Consider the following examples. The second and third examples generate warnings because the number of leading zeros in the binary representation of the converted scale value is equal to or greater than the specified minimum scale value overlap. The first, fourth, and fifth examples do not generate a warning because the number of leading zeros is less than the specified minimum overlap. The last example generates an error because the input range is quantized away, causing the binary representation of the converted value to always be zero.

Example	Input Format	Fraction Length	Scale Value	Specified Minimum Overlap (bits)	Binary Representation of Converted Scale Value	Warning Generated?
1	16	15	0.625	3	0.1010000000000000	No. <3 leading zeros
2	16	15	0.247	3	0.001111110011101	Yes
3	8	4	2.25	2	0010.0100	Yes
4	8	4	4.125	2	0100.0010	No. <2 leading zeros
5	8	4	0.0625	8	0000.0001	No. <8 leading zeros
6	8	4	0.00625	8	0000.0000	No. Error.

By default, the minimum overlap is 3 bits. If this is not sufficient for your filter design, adjust the setting as follows:

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Specify a positive integer in the **Minimum overlap of scale values (bits)** text field, as shown in the following display. To suppress the warnings, specify the number of bits in the input format.





- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `ScaleWarnBits` to reset the minimum overlap of scale values between filter coefficients and filter input.

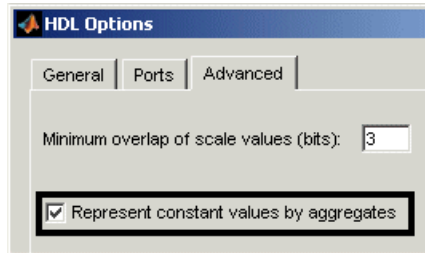
## Representing Constants with Aggregates

By default, the Filter Design HDL 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(-30, 16);
CONSTANT coeff2      :signed(15 DOWNT0 0) := to_signed(-89, 16);
```

If you prefer that all constant values be represented as aggregates, you can instruct the Filter Design HDL Coder to produce HDL code accordingly as follows:

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Select **Represent constant values by aggregates**, as shown the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

The preceding constant declarations would now appear as follows:

```
CONSTANT coeff1      :signed(15 DOWNT0 0) := (4 DOWNT0 2 => '0', 0 =>'0',  
OTHERS => ', '); -- sfix16_En15  
CONSTANT coeff2      :signed(15 DOWNT0 0) := (6 => '0', 4 DOWNT0 3 => '0',  
OTHERS => ', '); -- sfix16_En15
```

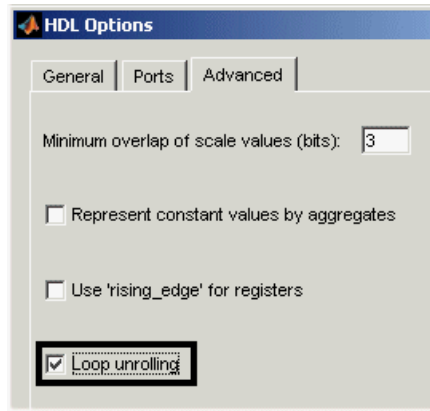
**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `UseAggregatesForConst` to represent all constants in the HDL code as aggregates.

## Unrolling and Removing VHDL Loops

By default, the Filter Design HDL Coder supports VHDL loops. However, some EDA tools do not support them. If you are using such a tool along with VHDL, you might need to unroll and remove FOR and GENERATE loops from your filter's generated VHDL code. Verilog code is always unrolled.

To unroll and remove FOR and GENERATE loops,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Select **Loop unrolling**, as shown in the following display.



#### 4

- Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `LoopUnrolling` to unroll and remove loops from generated VHDL code.

## Using the VHDL `rising_edge` Function

The Filter Design HDL 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'));
  ELSEIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
      delay_pipeline(1 TO 50) <= dleay_pipeline(0 TO 49);
    END IF;
  END IF;
```

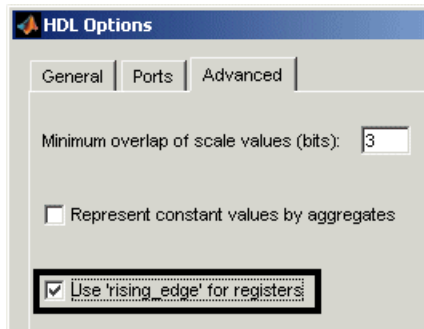
```
END PROCESS Delay_Pipeline_Process ;
```

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 **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Select **Use 'rising\_edge' for registers**, as shown in the following dialog.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `UseRisingEdge` to use the VHDL `rising_edge` function to check for rising edges during register operations.

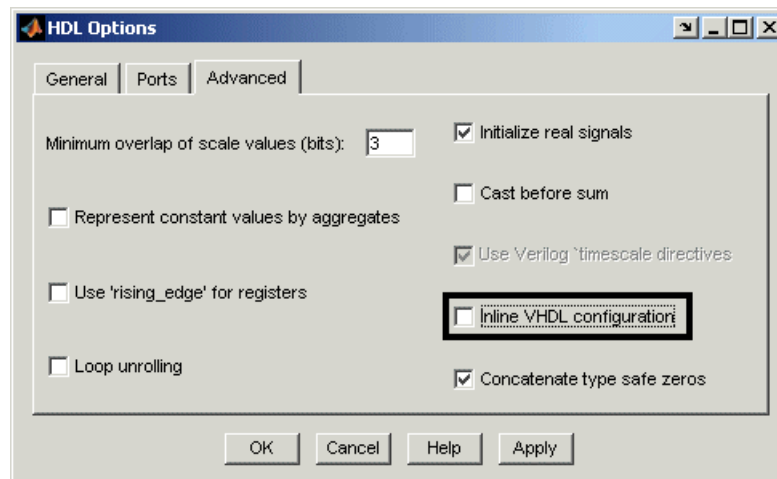
## Suppressing the Generation of VHDL Inline Configurations

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 Filter Design HDL 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 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Clear **Inline VHDL configuration**, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

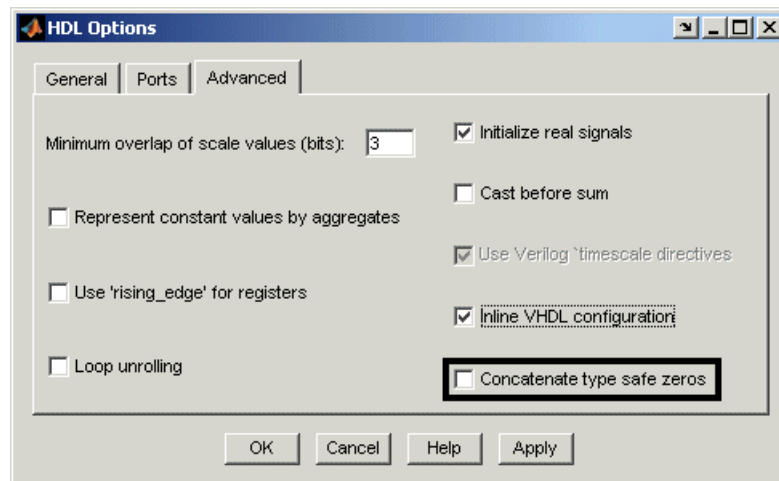
**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `InlineConfigurations` to suppress the generation of inline configurations.

## 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 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Clear **Concatenate type safe zeros**, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

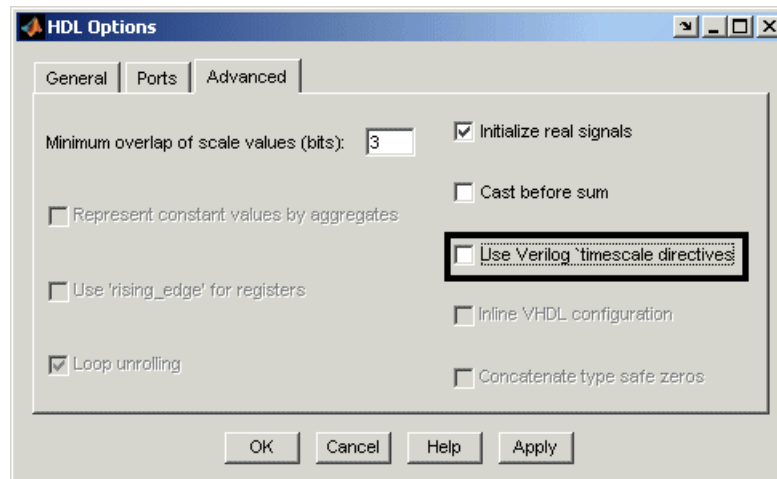
**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `SafeZeroConcat` to use the syntax "000000...", for concatenated zeros.

## Suppressing Verilog Time Scale Directives

In Verilog, the Filter Design HDL Coder generates time scale directives (``timescale`), as appropriate, 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 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Clear **Use Verilog `timescale directives**, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `UseVerilogTimescale` to suppress the use of time scale directives.

## Suppressing the Initialization of Signals of Type REAL

By default, the Filter Design HDL Coder initializes signals of type REAL with a value of 0.0. The coder assumes the signal is being used in a double-precision model. If it is possible that the filter's model might change, you can consider suppressing the initialization.

To suppress the initialization of type REAL signals to 0.0,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Clear **Initialize real signals**, as shown in the following display.



- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `InitializeRealSignals` to suppress the initialization of type REAL signals.

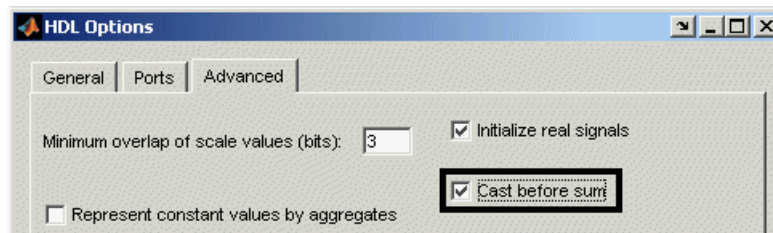
## Specifying Input Type Treatment for Addition and Subtraction Operations

MATLAB and typical DSP processors handle the treatment of input data types for addition and subtraction operations differently. MATLAB operates on input data using the data types as specified and converts the result to the result type. Typical DSP processors, on the other hand, type cast input data to the result type before operating on the data. Depending on the operation, the results can be very different.

By default, the Filter Design HDL Coder applies the MATLAB treatment of the input data. To specify the DSP processor treatment,

- 1 Click **More Options** in the **HDL filter** pane of the **Generate HDL** dialog. The **HDL Options** dialog appears.
- 2 Click the **Advanced** tab. Advanced coding options appear.
- 3 Select **Cast before sum**, as shown in the following display.





---

**Note** The setting of this option overrides the FDATool setting for the quantization parameter **Cast signals before accum.**

---

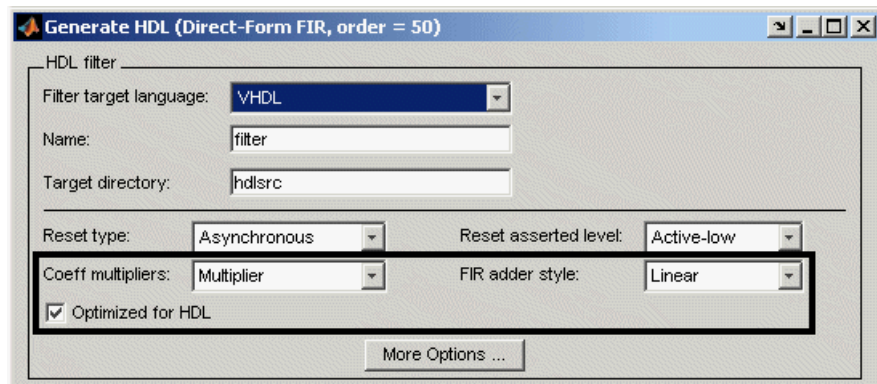
- 4 Click **Apply** to register the change or **OK** to register the change and close the dialog.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `CastBeforeSum` to cast input values to the result type for addition and subtraction operations. The setting of this property overrides the FDATool setting for the quantization parameter **Cast signals before accum.**

## Setting Optimizations

The Filter Design HDL Coder provides options for optimizing generated filter HDL code. You can optimize the code in a general sense by suppressing bit compatibility with MATLAB. Options are also available for optimizing coefficient multipliers and the final summation method used for FIR filters.

Code optimization options are listed in the lower half of the **HDL filter** pane of the **Generate HDL** dialog, as highlighted in the screen display below.



---

**Note** Some of the optimization settings generate HDL code that produces numeric results that differ from results produced by the quantized filter function.

---

The following sections discuss the various optimization options in more detail:

- “Optimizing Generated Code for HDL” on page 3-55
- “Optimizing Coefficient Multipliers” on page 3-55
- “Optimizing Final Summation for FIR Filters” on page 3-57
- “Optimizing the Clock Rate with Pipeline Registers” on page 3-58
- “Setting Optimizations for Synthesis” on page 3-59

## Optimizing Generated Code for HDL

By default, the Filter Design HDL Coder produces code that maintains bit compatibility with the numeric results produced by the specified quantized filter in MATLAB. If you need to generate HDL code that is slightly optimized for clock speed or space requirements, you can do so at the cost of the Filter Design HDL Coder:

- Making tradeoffs concerning data types
- Avoiding extra quantization
- Generating code that produces numeric results that are different than the filter results produced by MATLAB

To optimize generated code for clock speed or space requirements and suppress bit compatibility with MATLAB,

- 1** Select **Optimize for HDL** in the **HDL filter** pane of the **Generate HDL** dialog.
- 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** Click **Test Bench Options** in the **Test Bench Types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
  - b** Specify an integer in the **Error margin (bits)** text 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 **Apply** or **OK** to initiate code generation.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `OptimizeForHDL` to suppress bit compatibility with MATLAB.

## Optimizing Coefficient Multipliers

By default, the Filter Design HDL Coder produces code that includes coefficient multiplier operations. If necessary, 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** When you apply CSD or factored CSD techniques, the generated test bench can produce numeric results that differ from those produced by the original MATLAB filter function, unless no rounding or saturation occurs

---

To optimize coefficient multipliers,

- 1** Select CSD or Factored-CSD from the **Coeff multipliers** menu in the **HDL filter** pane of the **Generate HDL** dialog.
- 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** Click **Test Bench Options** in the **Test Bench Types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
  - b** Specify an integer in the **Error margin (bits)** text field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
  - c** Click **Apply** to register the change or **OK** to register the change and close the dialog.
- 3** Continue setting other options or click **Apply** or **OK** to initiate code generation.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `CoeffMultipliers` to optimize coefficient multipliers with CSD techniques.

## 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 Filter Design HDL 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 ensures numeric accuracy in comparison to the original MATLAB filter function. Tree and pipeline modes can produce numeric results that differ from those produced by the filter function.

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

- 1 Select one of the following options in the **HDL filter** pane of the **Generate HDL** dialog:

<b>For...</b>	<b>Select...</b>
Linear mode (the default)	Linear from the <b>FIR adder style</b> menu
Tree mode	Tree from the <b>FIR adder style</b> menu
Pipeline mode	The <b>Add pipeline registers</b> 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 Click **Test Bench Options** in the **Test Bench Types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
  - b Specify an integer in the **Error margin (bits)** text field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
  - c Click **Apply** to register the change or **OK** to register the change and close the dialog.
- 3 Continue setting other options or click **Apply** or **OK** to initiate code generation.

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

## 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.

<b>For...</b>	<b>Pipeline Registers Are Added Between...</b>
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 3-57.

---

**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 MATLAB filter function because they force the tree mode of final summation.

---

To use pipeline registers,

- 1** Select the **Add pipeline registers** option in the **HDL filter** pane of the **Generate HDL** dialog.
- 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** Click **Test Bench Options** in the **Test Bench Types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
  - b** Specify an integer in the **Error margin (bits)** text field that indicates an acceptable minimum number of bits by which the numerical results can differ before the coder issues a warning.
  - c** Click **Apply** to register the change or **OK** to register the change and close the dialog.
- 3** Continue setting other options or click **Apply** or **OK** to initiate code generation.

**Command Line Alternative:** Use the `generatehdl` and `generatetb` functions with the property `AddPipelineRegisters` to optimize the filters with pipeline registers.

## Setting Optimizations for Synthesis

The following table maps various synthesis goals with optimization settings that can help you achieve those goals. Use the table as a guide, while understanding that your results may vary depending on your synthesis target. For example, if you target FPGAs with built-in multipliers, the benefit of using CSD or factored CSD can be quite small until you utilize all the built-in multipliers. In an ASIC application, where the ability to route the

design largely controls the speed, the difference in speed between a linear and tree FIR adder style can be negligible. It may be necessary for you to combine various option settings to achieve your synthesis goals.

<b>To...</b>	<b>Select...</b>	<b>Which...</b>	<b>At the Cost of...</b>
Slightly increase the clock speed and slightly decrease the area used	<b>Optimize for HDL</b>	Removes extra quantization operations	Not remaining bit true to MATLAB.
To increase the clock speed while maintaining the area used	Tree for <b>FIR adder style</b>	Computes final summation for FIR, Asymmetric FIR, and Symmetric FIR pair-wise in parallel	Generally, not remaining bit true to MATLAB. Bit true to MATLAB only if no rounding or saturation occurs during final summation.
Significantly increase the clock speed while increasing overall latency and the area used	<b>Add pipeline registers</b>	Adds pipeline registers and forces use of the Tree FIR adder style, as necessary	Not remaining bit true to MATLAB when the FIR adder style is forced to Tree.
Decrease the area used while maintaining or increasing clock speed (depends on binary representation of coefficients)	CSD for <b>Coefficient multipliers</b>	Uses shift and add techniques instead of multipliers	Generally, not remaining bit true to MATLAB. Bit true to MATLAB only if no rounding or saturation occurs.
Decrease the area used (lower than what is achieved with CSD) while decreasing the clock speed	Factored CSD for <b>Coefficient multipliers</b>	Uses shift and add techniques on the prime factors of coefficients instead of multipliers	Generally, not remaining bit true to MATLAB. Bit true to MATLAB only if no rounding or saturation occurs.



## Customizing the Test Bench

In addition to generating HDL code for your quantized filter, the Filter Design HDL Coder generates a test bench you can use to verify filter results. The type of test bench, configurations for clock and reset signals, and the test stimuli will vary depending on your development environment and the filter you are testing. The following sections explain how to customize a test bench by

- “Renaming the Test Bench” on page 3-61
- “Specifying a Test Bench Type” on page 3-62
- “Configuring the Clock” on page 3-65
- “Configuring Resets” on page 3-67
- “Setting a Hold Time for Data Input Signals” on page 3-69
- “Setting an Error Margin for Optimized Filter Code” on page 3-70
- “Setting Test Bench Stimuli” on page 3-72

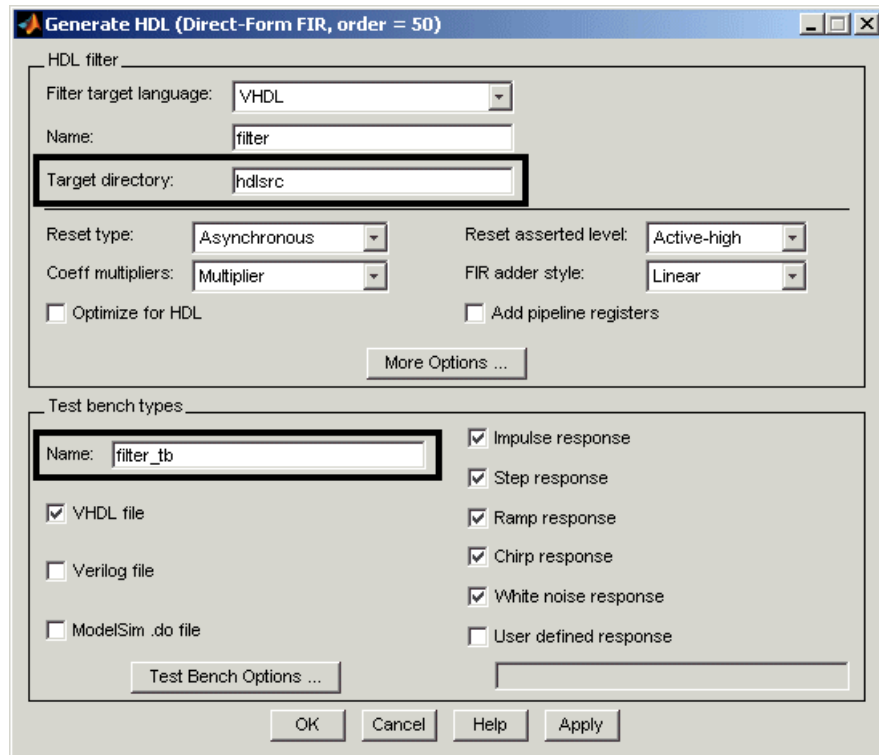
### Renaming the Test Bench

As discussed in “Customizing Reset Specifications” on page 3-26, the Filter Design HDL 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.

<b>If the Test Bench Is a...</b>	<b>The Extension Is...</b>
Verilog file	Defined by the <b>Verilog file extension</b> text field in the <b>HDL filter</b> pane of the <b>Generate HDL</b> dialog
VHDL file	Defined by the <b>VHDL file extension</b> text field in the <b>HDL filter</b> pane of the <b>Generate HDL</b> dialog
ModelSim DO file	<code>.do</code>

The file is placed in the directory defined by the **Target directory** option in the **HDL Filter** pane of the **Generate HDL** dialog.

To specify a test bench name, enter the name in the **Name** text field of the **Test bench types** pane, as shown in the following dialog.



---

**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 `generatetb` function with the property `TestBenchName` to specify a name for your filter's test bench.

### Specifying a Test Bench Type

The Filter Design HDL Coder can generate three 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
- ModelSim DO file to be used for simulation in the ModelSim environment

---

**Note** Due to differences in representation of double-precision data in VHDL and Verilog, restrictions apply to the types of test benches that are interoperable. The following table shows valid and invalid test bench type and HDL combinations.

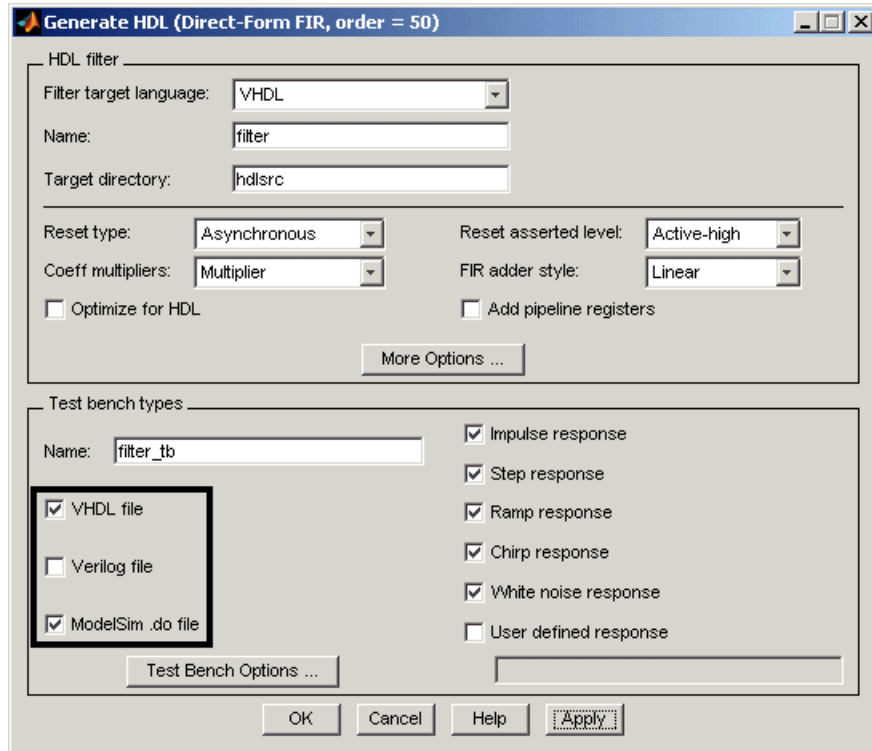
<b>Test Bench Type</b>	<b>VHDL</b>	<b>Verilog</b>
VHDL	Valid	Invalid
Verilog	Invalid	Valid
ModelSim .do	Not recommended*	Valid

\*Errors may be reported due to string comparisons.

These restrictions *do not* apply for fixed-point filters.

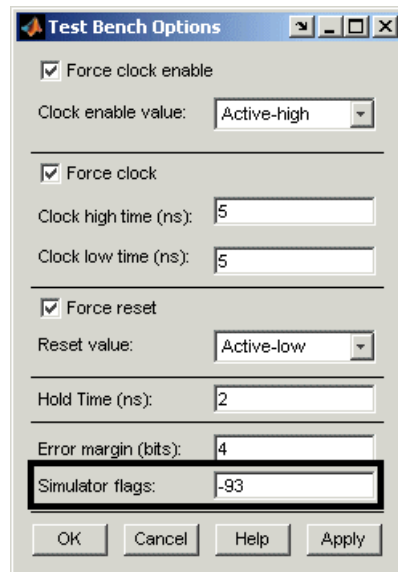
---

By default, the coder produces a VHDL or Verilog file only, depending on your language selection. If you want to generate additional test bench files, select the desired test bench types listed in the **Test bench types** pane of the **Generate HDL** dialog. The following dialog specifies that the coder generate VHDL and ModelSim DO test bench files.



If you choose to generate a ModelSim DO file, you have the option of specifying simulator flags. For example, you might need to specify a specific compiler version. To specify the flags,

- 1 Click **Test Bench Options** in the **Test bench types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
- 2 Type the flags of interest in the **Simulator flags** text box. The following dialog specifies that ModelSim use the `-93` compiler option for compilation.



- 3 Click **Apply** to register the change or **OK** register the change and close the dialog.

**Command Line Alternative:** Use the `generatetb` function's `TbType` parameter to specify the type of test bench files to be generated.

## Configuring the Clock

Based on default settings, the Filter Design HDL Coder configures the clock for a filter test bench such that it

- Forces clock enable input signals to active high (1)
- 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 **Test Bench Options** in the **Test bench types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
- 2 Make the following configuration changes as needed:

#### If You Want to...

Disable the forcing of clock enable input signals

Change the clock enable value to active low (0)

Disable the forcing of clock input signals

Reset the number of nanoseconds during which clock input signals are to be driven low (0)

Reset the number of nanoseconds during which clock input signals are to be driven high (1)

#### Then...

Clear **Force clock enable**.

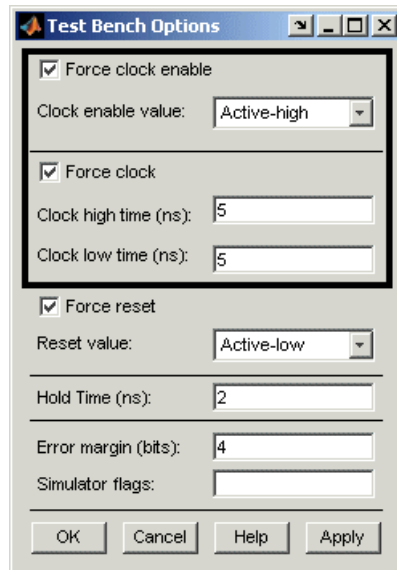
Select Active-low from the **Clock enable value** menu.

Clear **Force clock**.

Specify a positive integer in the **Clock low time** text field.

Specify a positive integer in the **Clock high time** text field.

The following dialog highlights the applicable options.



- 3 Click **Apply** to register the change or **OK** register the change and close the dialog.

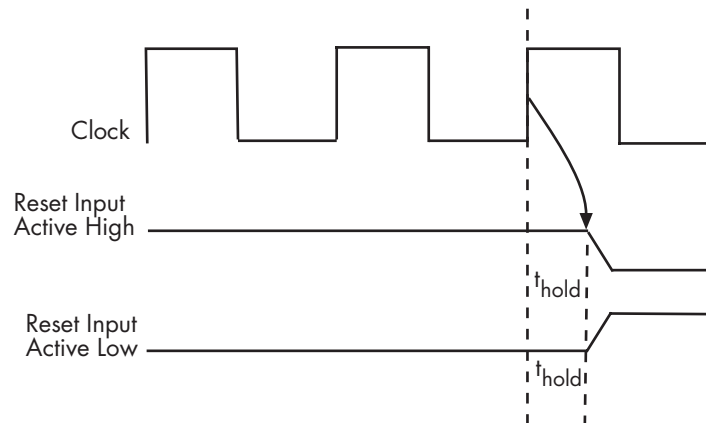
**Command Line Alternative:** Use the `generatetb` function with the properties `ForceClockEnable`, `ClockEnableValue`, `ForceClock`, `ClockHighTime`, and `ClockLowTime` to reconfigure the test bench clock.

## Configuring Resets

Based on default settings, the Filter Design HDL Coder configures the reset for a filter test bench such that it

- Forces reset input signals to active high (1).
- Applies a hold time of 2 nanoseconds for reset input signals.

The hold time is the amount of time, after two initial clock cycles, that reset input signals are to be held past the clock rising edge. The following figure shows the application of a hold time ( $t_{\text{hold}}$ ) for reset input signals when the signals are forced to active high and active low.




---

**Note** The hold time applies to reset input signals only if the forcing of reset input signals is enabled.

---

To change the default reset configuration settings,

- 1 Click **Test Bench Options** in the **Test bench types** pane in the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
- 2 Make the following configuration changes as needed:

**If You Want to...**

Disable the forcing of reset input signals

Change the reset value to active low (0)

Reset the hold time

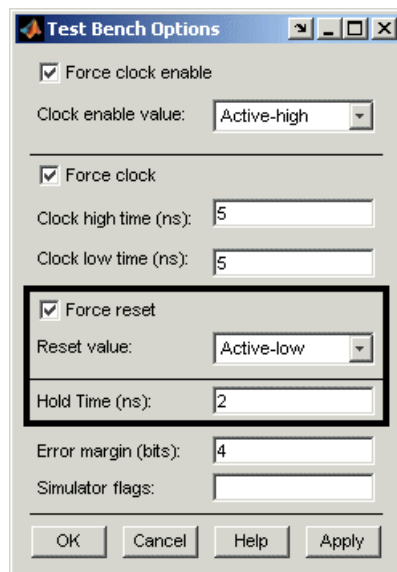
**Then...**

Clear **Force reset**.

Select Active-low from the **Reset value** menu.

Specify a positive integer, representing nanoseconds, in the **Hold time** text field.

The following dialog highlights the applicable options.



- 3 Click **Apply** to register the change or **OK** register the change and close the dialog.



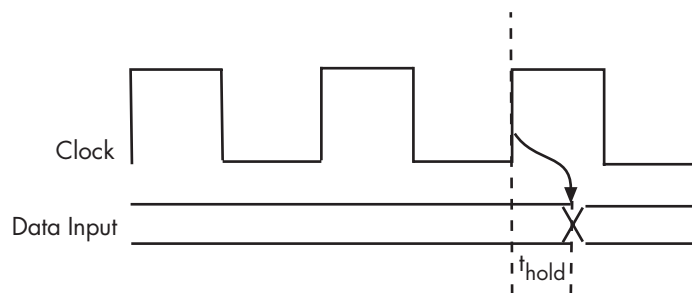
**Note**

- The reset value must match the setting of the reset asserted level specified for the filter.
- The hold time setting also applies to data input signals.

**Command Line Alternative:** Use the `generatetb` function with the properties `ForceReset`, `ResetValue`, and `HoldTime` to reconfigure test bench resets.

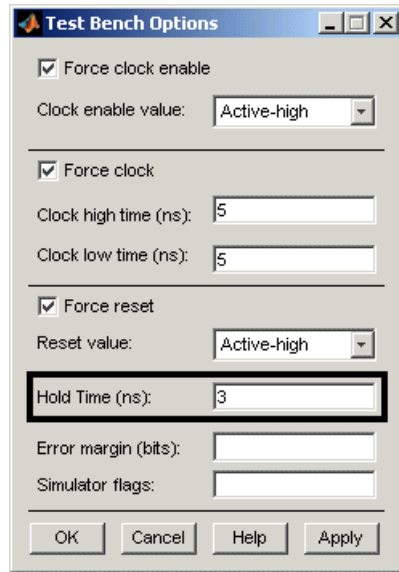
**Setting a Hold Time for Data Input Signals**

By default, the Filter Design HDL 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 ( $t_{\text{hold}}$ ) for data input signals.



To change the hold time setting,

- 1 Click **Test Bench Options** in the **Test bench types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
- 2 Specify a positive integer, representing nanoseconds, in the **Hold time** text field. The following dialog sets the hold time to 3 nanoseconds.



- 3** Click **Apply** to register the change or **OK** register the change and close the dialog.

---

**Note** The hold time setting also applies to reset input signals, if the forcing of such signals is enabled.

---

**Command Line Alternative:** Use the `generatetb` function with the property `HoldTime` to adjust the hold time setting.

### Setting an Error Margin for Optimized Filter Code

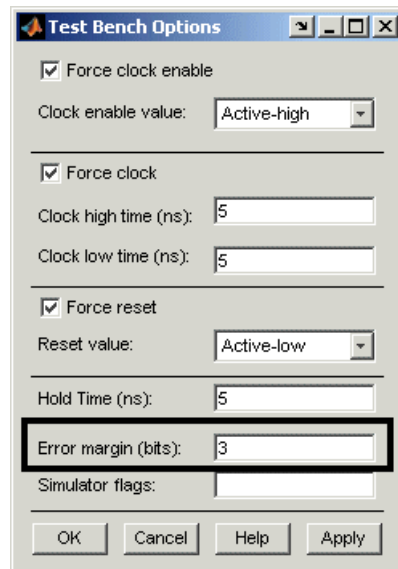
Customizations that provide optimizations can generate test bench code that produces numeric results that differ from those produced by the original MATLAB filter function. Specifically, these options include

- **Optimize for HDL**
- **Coeff multipliers**
- **FIR adder style** set to Tree

- **Add pipeline registers** for FIR, Asymmetric FIR, and Symmetric FIR filters

If you choose to use any of 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 Click **Test Bench Options** in the **Test Bench Types** pane of the **Generate HDL** dialog. The **Test Bench Options** dialog appears.
- 2 Specify an integer in the **Error margin** text field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning. The following dialog sets the error margin to 3 bits.



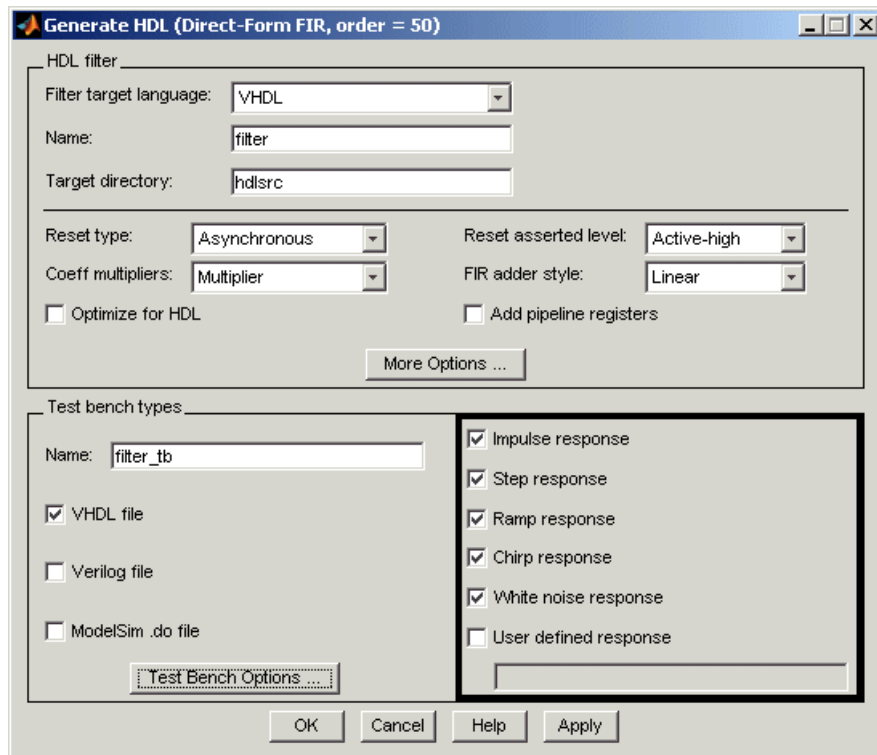
- 3 Click **Apply** to register the change or **OK** register the change and close the dialog.

## Setting Test Bench Stimuli

By default, the Filter Design HDL Coder generates a filter test bench that includes stimuli appropriate for the given filter. However, you can adjust the stimuli settings or specify user defined stimuli, if necessary. The following table lists the types of responses enabled by default.

<b>For Filters...</b>	<b>Default Response Types Include...</b>
FIR, FIRT, Symmetric FIR, and Antisymmetric FIR	Impulse, step, ramp, chirp, and white noise
All others	Step, ramp, and chirp

To modify the stimuli that the coder is to include in a test bench, enable the responses of interest listed in the **Test bench types** pane of the **Generate HDL** dialog. The following display highlights this pane of the dialog.



If you select **User defined response**, you must also specify a MATLAB 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 `generatetb` function with the properties `TestBenchStimulus` and `TestBenchUserStimulus` to adjust stimuli settings.

## Generating the HDL Code

To initiate HDL code generation for a filter and its test bench, click **Apply** or **OK** on the **Generate HDL** dialog. In addition to initiating code generation, **OK** closes the **Generate HDL** dialog. As the Filter Design HDL Coder processes the code, a sequence of messages similar to the following appears in your MATLAB Command Window:

```
### Starting VHDL code generation process for filter: MyFIR
### Generating filter.vhd file in: D:\work\FIRFilts
### Starting generation of MyFIR VHDL entity
### Starting generation of MyFIR VHDL architecture
### Successful completion of VHDL code generation process for
filter: MyFIR

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating VHDL file into D:\work\FIRFilts
### Done generating VHDL test bench.
```

---

**Note** The message text varies based on your customization settings (filenames and location, test bench type, and so on) and 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.

---

# Testing a Filter Design

---

This chapter explains how to apply supported test methods for verifying the HDL code that Filter Design HDL Coder generates for a filter design. Topics include the following:

- |  |   |
|--|---|
| “Overview of the Test Methods” (p. 4-2)            | Provides an overview of the available test methods                                  |
| “Testing with an HDL Test Bench” (p. 4-3)          | Explains how to test generated filter HDL code, using generated HDL test bench code |
| “Testing with a ModelSim Tcl/Tk DO File” (p. 4-12) | Explains how to test generated filter HDL code, using a generated ModelSim DO file  |

### Overview of the Test Methods

As explained in “Customizing the Test Bench” on page 3-61, the type of test bench, configurations for clock and reset signals, error margin, and the test stimuli will vary depending on your development environment and the customizations you apply when you generate your design. Depending on the types of test benches you generate, you can verify your filter design by

- “Testing with an HDL Test Bench” on page 4-3
- “Testing with a ModelSim Tcl/Tk DO File” on page 4-12



## Testing with an HDL Test Bench

If you customize the Filter Design HDL Coder to generate VHDL or Verilog test bench code, you can use a simulator of your choice to verify your filter design. For example purposes, the following sections explain how to apply generated HDL test bench code by using ModelSim. In summary, you need to

- 1 Generate the filter and test bench HDL code.
- 2 Start the simulator.
- 3 Compile the generated filter and test bench files.
- 4 Run the test bench simulation.

---

**Note** Due to differences in representation of double-precision data in VHDL and Verilog, restrictions apply to the types of test benches that are interoperable. The following table shows valid and invalid test bench type and HDL combinations.

Test Bench Type	VHDL	Verilog
VHDL	Valid	Invalid
Verilog	Invalid	Valid
ModelSim .do	Not recommended*	Valid

\*Errors may be reported due to string comparisons.

These restrictions *do not* apply for fixed-point filters.

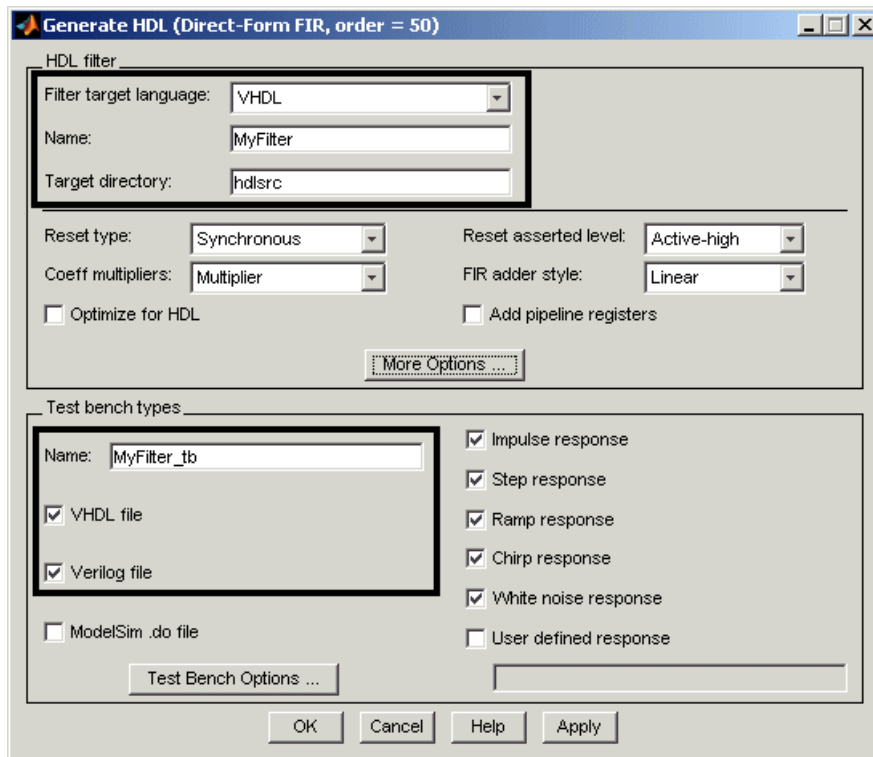
---

### 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 3-62, 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 **VHDL file** or **Verilog file** option in the **Test bench types** pane of the **Generate HDL** dialog. You

can specify a number of other test bench customizations, as described in “Customizing the Test Bench” on page 3-61.

The following dialog shows settings for generating the filter and test bench files `MyFilter.vhd`, `MyFilter_tb.vhd`, and `MyFilter_tb.v`. The dialog also specifies that the generated files are to be placed in the default target directory `hdlsrc` under the current working directory.



---

**Note** The settings for the **Reset asserted level** option in the **HDL filter** pane of the **Generate HDL** dialog and the **Reset value** option for **Force reset** in the **Test Bench Options** dialog must match. If you change one of these options, make sure you adjust the other option accordingly.

---

After you click **OK**, the Filter Design HDL Coder displays the following messages in the MATLAB Command Window:

```
### Starting VHDL code generation process for filter: MyFilter
### Generating MyFilter.vhd file in:hdlsrc
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for
filter: MyFilter

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating VHDL file MyFilter_tb.vhd in:hdlsrc
### Done generating VHDL test bench.

### Starting generation of Verilog Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating Verilog file MyFilter_tb.v in:hdlsrc
### 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 use the command line interface, you must

- Invoke the functions `generatehdl` and `generatetb`, in that order. The order is important because `generatetb` takes into account additional latency or numeric differences introduced into the filter's HDL code that results from the following property settings:

Property...	Set to...	Can Affect...
'AddInputRegister' or 'AddOutputRegister'	'on'	Latency
'FIRAdderStyle'	'pipeline'	Numeric computations and latency
'FIRAdderStyle'	'tree'	Numeric computations
'OptimizeForHDL'	'off'	Numeric computations
'CastBeforeSum'	'on'	Numeric computations
'CoeffMultipliers'	'csd' or 'factored-csd'	Numeric computations

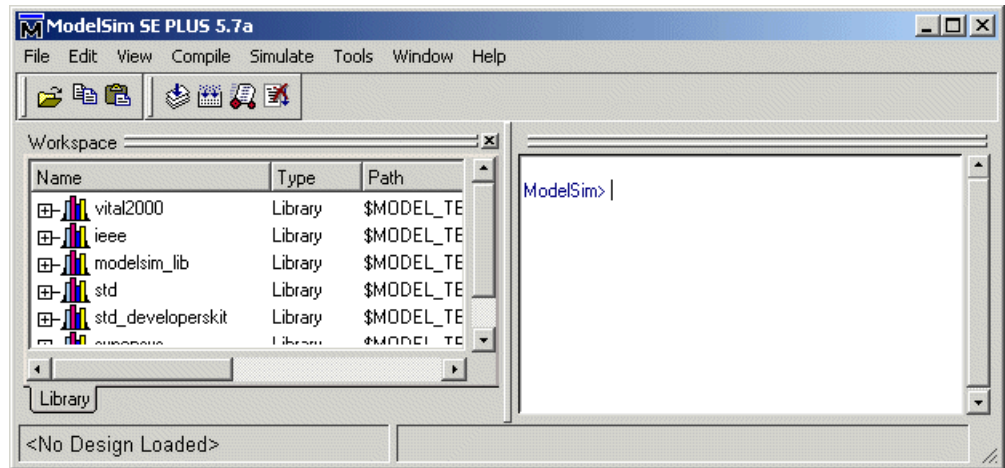
- Specify 'VHDL' or 'Verilog' for the TbType parameter. For double-precision filters, you must specify the type that matches the target language specified for your filter code.
- Make sure the property settings specified in the invocation of `generatetb` match those of the corresponding invocation of `generatehdl`. You can do this in one of two ways:
  - Omit explicit property settings from the `generatetb` invocation. This function automatically inherits the property settings established in the `generatehdl` invocation.
  - Take care to specify the same property settings specified in the `generatehdl` invocation.

You might also want to consider using the function `generatetbstimulus` to return the test bench stimulus to the MATLAB Command Window.

For details on the property name and property value pairs that you can specify with the `generatehdl` and `generatetb` functions for customizing the output, see Chapter 5, “Properties — Categorical List”.

## Starting the Simulator

After you generate your filter and test bench HDL files, start your simulator. When you start ModelSim, a screen display similar to the following appears:



After starting the simulator, set the current directory to the directory 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 might need to complete some precompilation setup. For example, in ModelSim, you might choose to create a design library to store compiled VHDL entities, packages, architectures, and configurations.

The following ModelSim command sequence changes the current directory 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.

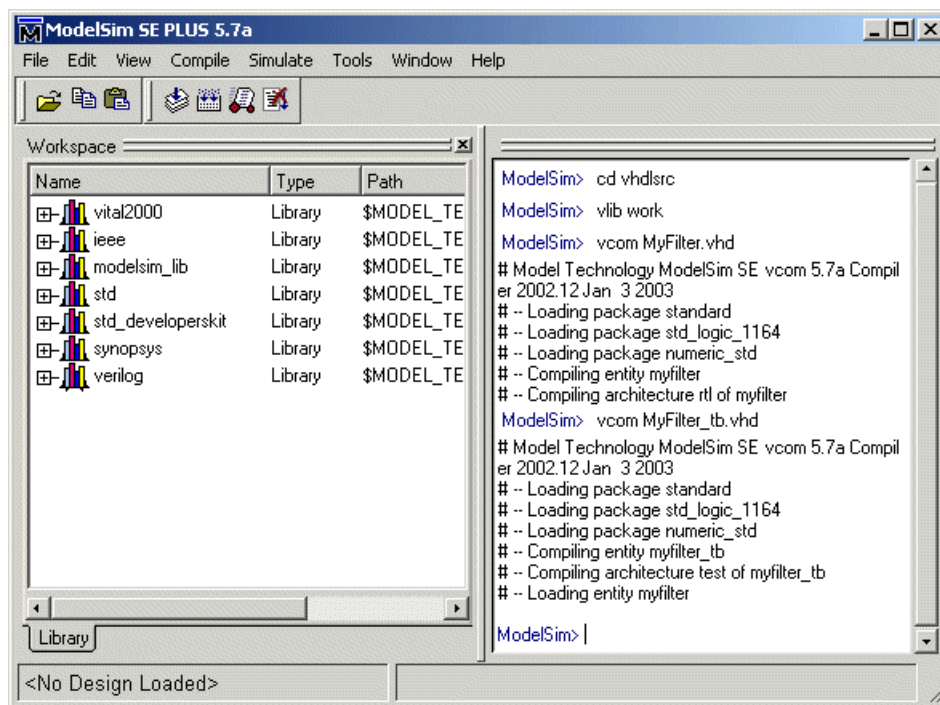
```
cd hdlsrc
vlib work
vcom MyFilter.vhd
vcom MyFilter_tb.vhd
```

---

**Note** For filters that have floating-point (double) realizations, you must specify the vcom command with the -93 option. This is because the test bench uses the image attribute, which is available only in VHDL-93.

---

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

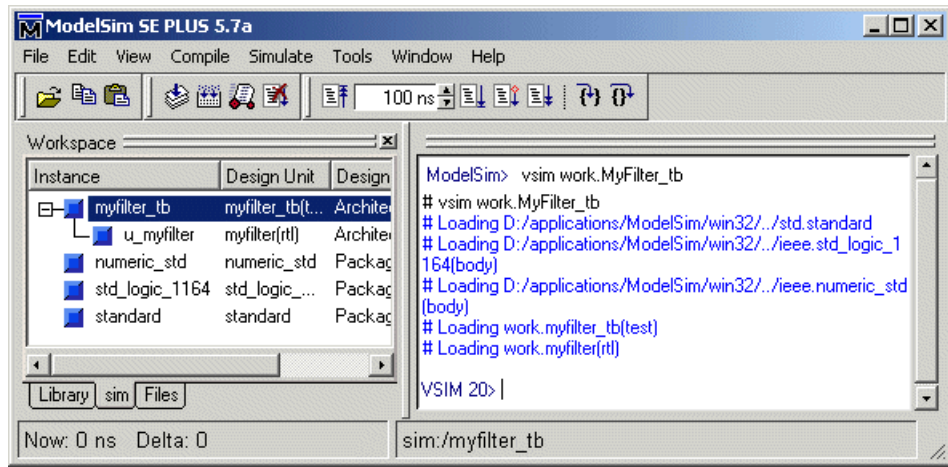


## Running the Test Bench Simulation

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 ModelSim, you load the test bench for simulation with the vsim command. For example:

```
vsim work.MyFilter_tb
```

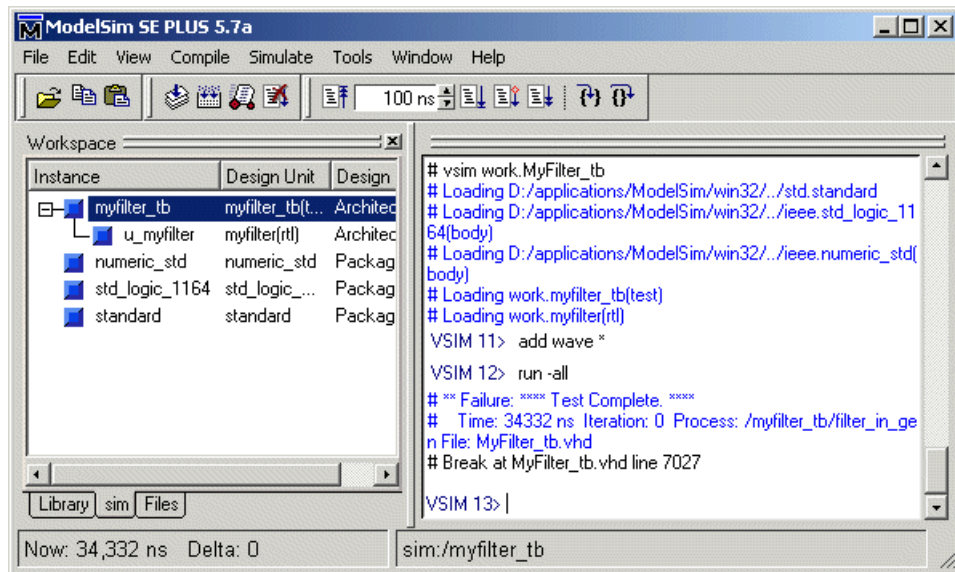
The following ModelSim display shows the results of loading `work.MyFilter_tb` with the `vsim` command:



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 ModelSim, 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 appropriate simulator command. For example, in ModelSim, you can start a simulation with the `run -all` command.

The following ModelSim 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 any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. 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 MATLAB filter function. For HDL test benches, the Filter Design HDL Coder compares the results and if they differ, excluding the specified error margin, returns an error message similar to the following:

```
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 successfully run to completion. The `Failure` part of the message is tied to the mechanism the Filter Design HDL Coder uses to end the simulation.

---





### Testing with a ModelSim Tcl/Tk DO File

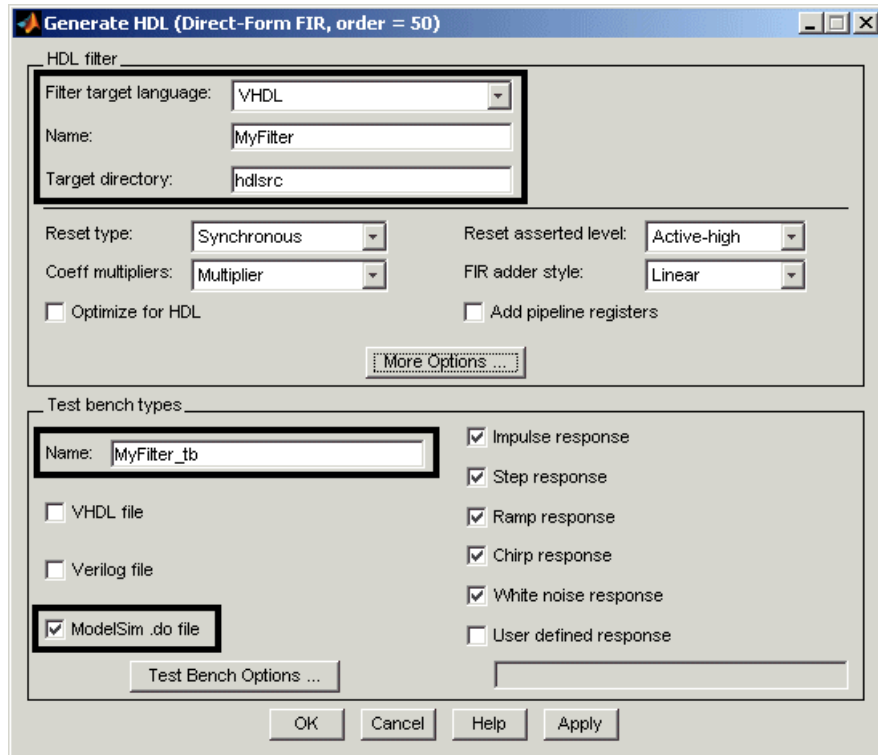
If you customize the Filter Design HDL Coder to generate a ModelSim Tcl/Tk DO file test bench, you must use ModelSim to test and verify your filter design. When you choose this test bench method, you need to

- 1 Generate the filter and test bench HDL code.
- 2 Start ModelSim.
- 3 Compile the generated filter file.
- 4 Execute the ModelSim DO file.

### Generating the Filter HDL Code and Test Bench DO File

Use the Filter Design HDL Coder GUI or command line interface to generate the HDL code for your filter design and test bench. The GUI generates a ModelSim DO file test bench if you select the **ModelSim .do file** option in the **Test bench types** pane of the **Generate HDL** dialog. You can specify a number of other test bench customizations, as described in “Customizing the Test Bench” on page 3-61.

The following dialog shows settings for generating the filter and test bench files `MyFilter.vhd` and `MyFilter_tb.do`. The dialog also specifies that the generated files are to be placed in the default target directory `hdlsrc` under the current working directory.



**Note** The settings for the **Reset asserted level** option in the **HDL filter** pane of the **Generate HDL** dialog and the **Reset value** option for **Force reset** in the **Test Bench Options** dialog must match. If you change one of these options, make sure you adjust the other option accordingly.

After you click **OK**, Filter Design HDL Coder displays the following messages in the MATLAB Command Window:

```
### Starting VHDL code generation process for filter: MyFilter
### Generating MyFilter.vhd file in: hdlsrc
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for
```

```
filter: MyFilter

### Starting generation of ModelSim .do file Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating ModelSim .do file MyFilter_tb in: hdlsrc
### Done generating ModelSim .do file 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 use the command line interface, you must

- Invoke the functions `generatehdl` and `generatetb`, in that order. The order is important because `generatetb` takes into account latency or numeric differences introduced into the filter's HDL code that results from the following property settings:

Property...	Set to...	Can Affect...
'AddInputRegister' or 'AddOutputRegister'	'on'	Latency
'FIRAdderStyle'	'pipeline'	Numeric computations and latency
'FIRAdderStyle'	'tree'	Numeric computations
'OptimizeForHDL'	'off'	Numeric computations
'CastBeforeSum'	'on'	Numeric computations
'CoeffMultipliers'	'csd' or 'factored-csd'	Numeric computations

- Specify 'ModelSim' for the `TbType` parameter.

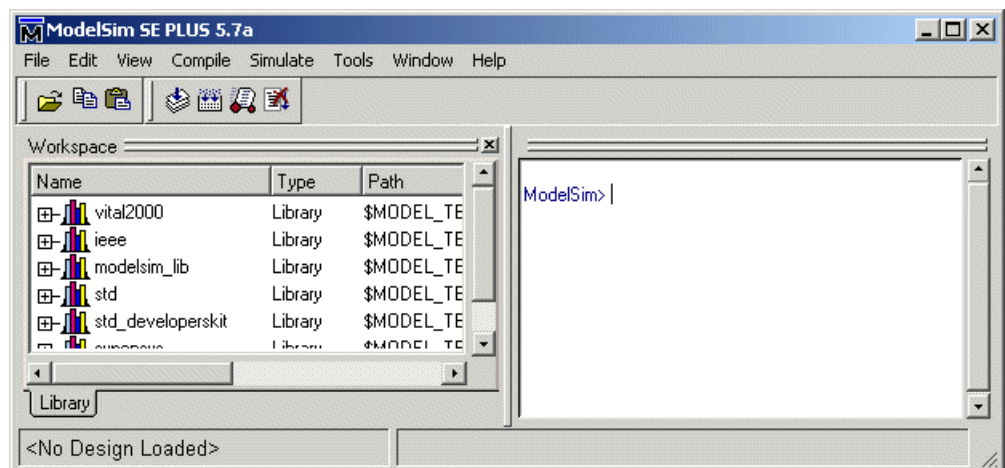
- Make sure the property settings specified in the invocation of `generatetb` match those of the corresponding invocation of `generatehdl`. You can do this in one of two ways:
  - Omit explicit property settings from the `generatetb` invocation. This function automatically inherits the property settings established in the `generatehdl` invocation.
  - Take care to specify the same property settings specified in the `generatehdl` invocation.

You might also want to consider using the function `generatetbstimulus` to return the test bench stimulus to the MATLAB Command Window.

For details on the property name and property value pairs that you can specify with the `generatehdl` and `generatetb` functions for customizing the output, see Chapter 5, “Properties — Categorical List”.

## Starting ModelSim

After you generate your filter and test bench HDL files, start ModelSim. A screen display similar to the following appears:



After starting the simulator, set the current directory to the directory that contains your generated filter and test bench files.

### Compiling the Generated Filter File

Using your choice HDL compiler, compile the generated filter HDL file. The test bench DO file looks for your compiled HDL elements in a design library named `work`. The design library stores the compiled HDL components. If the design library `work` does not exist, you can create it by setting the current directory to `hdlsrc` and then issuing the command `vlib work`. Once the library exists, you can use the ModelSim compiler to compile the filter's HDL file.

The following ModelSim command sequence changes the current directory to `hdlsrc`, creates the design library `work`, and compiles filter VHDL code:

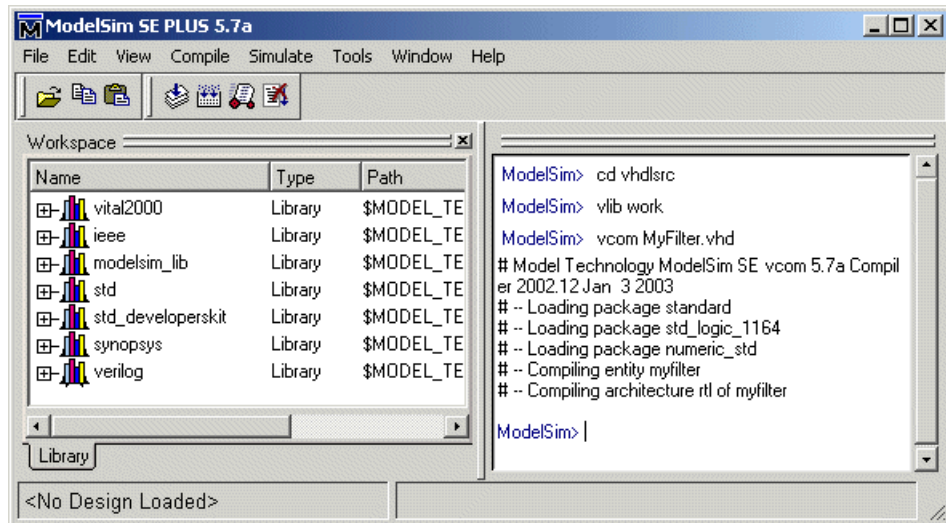
```
cd hdlsrc
vlib work
vcom MyFilter.vhd
```

---

**Note** For VHDL filter code that has floating-point (double) realizations, you must specify the `vcom` command with the `-93` option. This is because the test bench uses the `image` attribute, which is available only in VHDL-93.

---

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



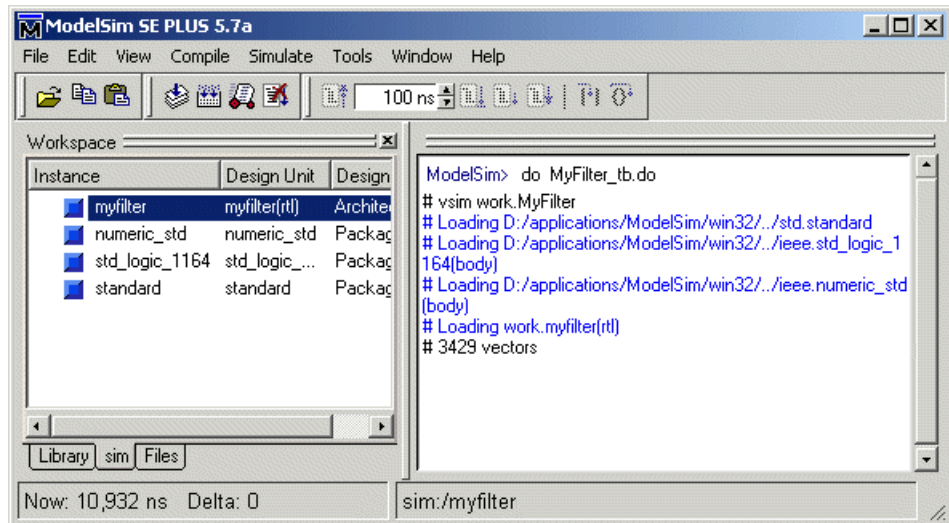
## Execute the ModelSim DO File

Once your filter's HDL file is compiled, execute the generated test bench DO file. The DO file

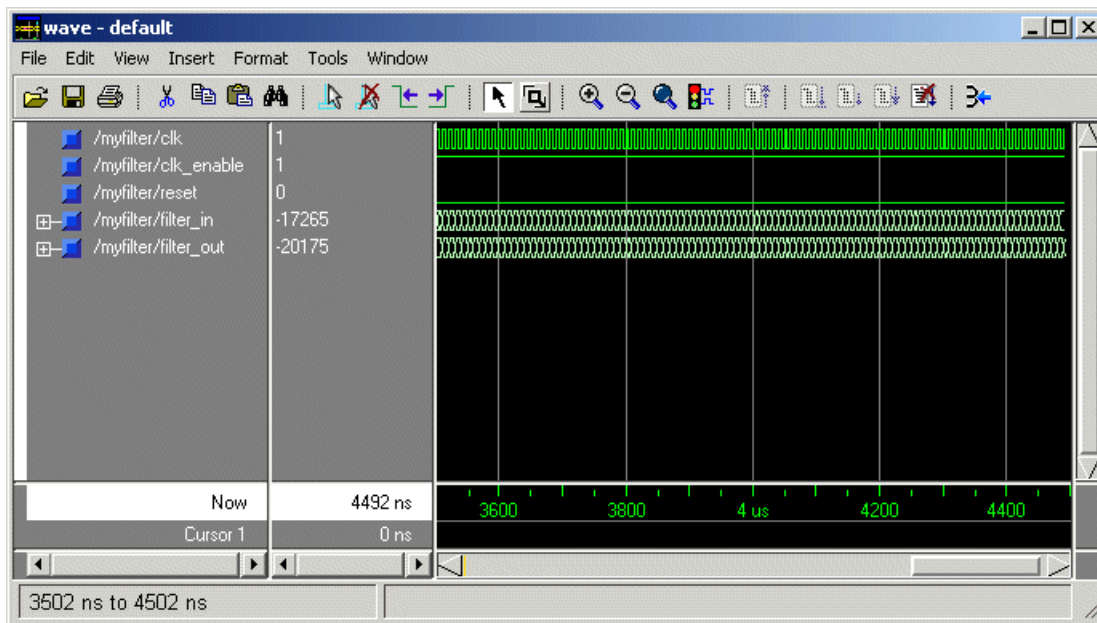
- 1 Loads the compiled filter for simulation.
- 2 Opens a **wave** window and populates it with filter signals.
- 3 Applies stimulus to filter signals with **force** commands.
- 4 Compares filter output to expected results.

You can execute the DO file by using the ModelSim **do** command or the Tcl **source** command. The following ModelSim display shows how to use the **do** command.

## 4 Testing a Filter Design



The test bench DO script displays the simulation results in a **wave** window that appears as follows:





---

**Note** The Filter Design HDL Coder adjusts the wave form such that it is appropriate for the specified filter output settings.

---

As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. 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 MATLAB filter function. The Filter Design HDL Coder compares the results and, if they differ, returns an error message similar to the following:

```
Error in filter test: Expected xxxxxxxx Actual xxxxxxxx
```

---

**Note** You cannot specify an error margin for ModelSim DO file test benches like you can for HDL test benches. The Filter Design HDL Coder returns an error if the expected and actual values do not match exactly.

---

You must determine whether the actual results are expected based on the customizations you specified when generating the filter HDL code.



# Properties — Categorical List

---

“Language Selection Properties” (p. 5-2)	Lists properties for selecting the language of the generated HDL code
“File Naming and Location Properties” (p. 5-2)	Lists properties that name and specify the location of generated files
“Reset Properties” (p. 5-2)	Lists reset properties
“Header Comment and General Naming Properties” (p. 5-3)	Lists header comment and general naming properties
“Port Properties” (p. 5-4)	Lists port properties
“Advanced Coding Properties” (p. 5-4)	Lists advanced HDL coding properties
“Optimization Properties” (p. 5-6)	Lists optimization properties
“Test Bench Properties” (p. 5-6)	Lists test bench properties

## Language Selection Properties

TargetLanguage	Specify the HDL language to use for the generated filter code
----------------	---

## File Naming and Location Properties

Name	Name a VHDL entity or Verilog module for the filter and the file that is to contain the generated HDL code
TargetDirectory	Identify the directory into which generated output files are to be written
VHDLFileExtension	Specify the string to be used as the file type extension for generated VHDL files
VerilogFileExtension	Specify the string to be used as the file type extension for generated Verilog files

## Reset Properties

ResetAssertedLevel	Specify the asserted (active) level of the reset input signal
ResetType	Specify whether an asynchronous or synchronous reset style is to be used when generating HDL code for registers

## Header Comment and General Naming Properties

ClockProcessPostfix	Specify a string to be appended to HDL clock process names
CoeffName	Specify a string to be used as the prefix for filter coefficient names
EntityConflictPostfix	Specify a string to be appended to duplicate VHDL entity or Verilog module names
PackagePostfix	Specify a string to be appended to the specified filter name to form the name of a VHDL package file
ReservedWordPostfix	Specify a string to be appended to strings that you specify with other options that are VHDL or Verilog reserved words
SplitArchFilePostfix	Specify a string to be appended to the specified name to form the name of the file that is to contain the filter's VHDL architecture
SplitEntityArch	Specify whether the generated VHDL entity and architecture code is to be included in a single VHDL file or separate files
SplitEntityFilePostfix	Specify a string to be appended to the specified filter name to form the name of the file that is to contain the filter's VHDL entity

## Port Properties

AddInputRegister	Specify whether an extra register is to be added to the HDL code for filter input
AddOutputRegister	Specify whether an extra register is to be added to the HDL code for filter output
ClockEnableInputPort	Name the HDL port for the filter's clock enable input signals
InputPort	Name the HDL port for a filter's input signals
InputType	Specify the HDL data type for the filter's input port
OutputPort	Name the HDL port for a filter's output signals
OutputType	Specify the HDL data type for the filter's output port
ResetInputPort	Name the HDL port for a filter's reset input signals

## Advanced Coding Properties

BlockGenerateLabel	Specify a string to be appended to block labels used for HDL GENERATE statements
CastBeforeSum	Specify whether input values for addition and subtraction operations are to be type cast before the operations occur

InitializeRealSignals	Specify whether signals of type REAL are to be declared with an initial value of 0.0
InlineConfigurations	Specify whether generated VHDL code is to include inline configurations
InstanceGenerateLabel	Specify a string to be appended to instance section labels in VHDL GENERATE statements
LoopUnrolling	Specify whether VHDL FOR and GENERATE loops are to be unrolled and excluded from the generated VHDL code
OutputGenerateLabel	Specify a string that labels an output assignment block for VHDL GENERATE statements
SafeZeroConcat	Specify the syntax to be used in the generated VHDL code for concatenated zeros
ScaleWarnBits	Control whether the coder generates a warning for scale values that are below a specified numeric threshold relative to the input data format
UseAggregatesForConst	Specify whether all constants should be represented by aggregates, including constants that are less than 32 bits
UseRisingEdge	Specify the VHDL coding style to be used to check for rising edges when operating on registers

UseVerilogTimescale	Specify whether generated Verilog code can include compiler <code>`timescale</code> directives
UserComment	Specify a string to be added as a comment line in the header of generated filter and test bench files

## Optimization Properties

AddPipelineRegisters	Optimize the clock rate used by filter code by adding pipeline registers
CoeffMultipliers	Specify the technique to be used for processing coefficient multiplier operations
FIRAdderStyle	Specify a final summation technique to be used for FIR filters
OptimizeForHDL	Specify whether generated HDL code is to be optimized for specific performance or space requirements

## Test Bench Properties

ClockEnableValue	Specify the constant value to which a test bench is to force clock enable input signals
ClockHighTime	Specify the number of nanoseconds during which a test bench is to drive the clock input signals high (1)
ClockInputPort	Name the HDL port for the filter's clock input signals



ClockLowTime	Specify the number of nanoseconds during which a test bench is to drive clock input signals low (0)
ErrorMargin	Specify an error margin for HDL language-based test benches
ForceClock	Specify whether the test bench is to force clock input signals
ForceClockEnable	Specify whether the test bench is to force clock enable input signals
ForceReset	Specify whether the test bench is to force reset input signals
HoldTime	Specify the hold time for filter data input signals and forced reset input signals
ResetValue	Specify the constant value to which the test bench is to force reset input signals
SimulatorFlags	Specify simulator flags that are to be applied to your generated test bench
TestBenchName	Name a VHDL test bench entity or Verilog module and the file that is to contain test bench code
TestBenchStimulus	Specify the input stimuli the test bench is to apply to the filter
TestBenchUserStimulus	Specify a user-defined MATLAB function that returns a vector of values that the test bench is to apply to the filter



# Properties — Alphabetical List

---

# AddInputRegister

---

**Purpose** Specify whether an extra register is to be added to the HDL code for filter input

**Settings** 'on' (default)  
Add an extra input register to the filter's generated HDL code. This is the default.

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.

You might 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 not using the extra register if the latency it introduces to the filter is not tolerable.

**See Also** AddOutputRegister

**Purpose** Specify whether an extra register is to be added to the HDL code for filter output

**Settings** 'on' (default)  
Add an extra output register to the filter's generated HDL code. This is the default.

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.

You might consider omitting the extra register if you are incorporating the filter into HDL code that has its own input register. You might also consider not using the extra register if the latency it introduces to the filter is not tolerable.

**See Also** AddInputRegister

# AddPipelineRegisters

---

**Purpose** Optimize the clock rate used by filter code by adding pipeline registers

**Settings** 'on'  
Add a pipeline register between stages of computation in a filter.

<b>For...</b>	<b>A Pipeline Register Is Added Between...</b>
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** Use this option to optimize the clock rate achievable by filter code by applying pipeline registers. Although the registers add to the overall filter latency, they provide significant improvements to the clock rate. These registers are disabled by default. When you enable them, the coder adds a register between stages of computation. 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.

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see “Optimizing Final Summation for FIR Filters” on page 3-57.

---

**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 MATLAB filter function because they force the tree mode of final summation. In such cases, consider adjusting the test bench error margin.

---

**See Also**

CoeffMultipliers, FIRAdderStyle, OptimizeForHDL

# BlockGenerateLabel

---

**Purpose** Specify a string to be appended to block labels used for HDL GENERATE statements

**Settings** 'string'  
Specify a postfix to be appended to block labels used for HDL GENERATE statements. The default string is `_gen`.

**See Also** InstanceGenerateLabel, OutputGenerateLabel



**Purpose** Specify whether input values for addition and subtraction operations are to be type cast before the operations occur

**Settings** 'on'(default) **Select the check box to**  
Type cast input values in addition and subtraction operations to the result type before operating on the values. This is the default. This setting produces numeric results that are typical of Simulink fixed-point results produced by DSP processors.

---

**Note** The FDATool sets this option by default. However, the Filter Design HDL Coder default behavior overrides the FDATool setting and disables type casting.

---

'off'  
Preserve the types of input values during addition and subtraction operations and then convert the result to the result type. This is the MATLAB mode of operation.

**See Also** InitializeRealSignals, InlineConfigurations, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge, UseVerilogTimescale

# ClockEnableInputPort

---

**Purpose** Name the HDL port for the filter's clock enable input signals

**Settings** 'string'  
Name the HDL port for the filter's clock enable input signals. The default string is `clk_enable`.

For example, if you specify the string `'filter_clock_enable'` for filter entity `Hq`, the generated entity declaration might look as follows:

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

<b>Purpose</b>	Specify the constant value to which a test bench is to force clock enable input signals
<b>Settings</b>	'active_high' (default) Specify that the test bench is to set the clock enable input to active high (1).  'active_low' Specify that the test bench is to set the clock enable input to active low (0).
<b>Usage Notes</b>	The Filter Design HDL Coder ignores this property if ForceClockEnable is set to 'off'.
<b>See Also</b>	ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, ForceReset, HoldTime, ResetValue

# ClockHighTime

---

**Purpose** Specify the number of nanoseconds during which a test bench is to drive the clock input signals high (1)

**Settings** ns  
Specify the number of nanoseconds during which a test bench is to drive clock input signals high (1). The default is 5.

**Usage Notes** The Filter Design HDL Coder ignores this property if ForceClock is set to 'off'.

**See Also** ClockEnableValue, ClockLowTime, ForceClock, ForceClockEnable, ForceReset, HoldTime, ResetValue

**Purpose** Name the HDL port for the filter's clock input signals

**Settings** 'string'  
Name the HDL port for the filter's clock input signals. The default 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 `ReservedWordPostfix` for more information.

**See Also** `ClockEnableInputPort`, `InputPort`, `InputType`, `OutputPort`, `OutputType`, `ResetInputPort`

# ClockLowTime

---

<b>Purpose</b>	Specify the number of nanoseconds during which a test bench is to drive clock input signals low (0)
<b>Settings</b>	ns Specify the number of nanoseconds during which a test bench is to drive clock input signals low (0). The default is 5.
<b>Usage Notes</b>	The Filter Design HDL Coder ignores this property if ForceClock is set to 'off'.
<b>See Also</b>	ClockEnableValue, ClockHighTime, ForceClock, ForceClockEnable, ForceReset, HoldTime, ResetValue

**Purpose** Specify a string to be appended to HDL clock process names

**Settings** 'string'  
Specify a string to be appended to HDL clock process names.  
The default is `_process`.

The Filter Design HDL 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`

# CoeffMultipliers

---

**Purpose** Specify the technique to be used for processing coefficient multiplier operations

**Settings**

'multiplier' (default)  
Retain multiplier operations in the generated HDL code. This is the default.

'csd'  
Decrease the area used by the filter while maintaining or increasing clock speed. This option uses canonic signed digit (CSD) techniques, which replace multiplier operations with shift and add operations.

'factored-csd'  
Decrease the area used by the filter by more than what you can achieve with CSD at the cost of decreasing clock speed. This option uses factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the coefficients.

**Usage Notes**

By default, the Filter Design HDL Coder produces code that includes coefficient multiplier operations. You can optimize these operations by instructing the coder to replace them with additions of partial products produced by 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.

---

**Note** When you specify one of the two options for optimizations, CSD or Factored CSD, the generated test bench can produce numeric results that differ from those produced by the original MATLAB filter function if rounding or saturation occurs.

---

**See Also** AddPipelineRegisters, FIRAdderStyle, OptimizeForHDL



**Purpose** Specify a string to be used as the prefix for filter coefficient names

**Settings** 'string'  
Customize the prefix used for filter coefficient names. The default string is `coeff`.

**For... The Prefix Is Concatenated with...**

**FIR** Each coefficient number, starting with 1. For example, filters the default for the first coefficient would be `coeff1`.

**IIR** An underscore (`_`) and an a or b coefficient name (for filters example, `_a2`, `_b1`, or `_b2`) followed by the string `_sectionn`, where *n* is the section number. For example, the default for the first numerator coefficient of the third section is `coeff_b1_section3`.

The string that you specify

- Must start with a letter.
- Cannot end with an underscore (`_`)
- Cannot include a double underscore (`__`)

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 coeff1 : signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
CONSTANT coeff1 : signed(15 DOWNTO 0) := to_signed(-81, 16); -- sfix16_En15
CONSTANT coeff1 : signed(15 DOWNTO 0) := to_signed(120, 16); -- sfix16_En15
.
.
.
```

## CoeffName

---

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 `constant`, the resulting base name string would be `constant_rsvd`. See `ReservedWordPostfix` for more information.

### See Also

`ClockProcessPostfix`, `EntityConflictPostfix`, `PackagePostfix`, `ReservedWordPostfix`

**Purpose** Specify a string to be appended to duplicate VHDL entity or Verilog module names

**Settings** 'string'  
Specify a string to be appended to duplicate VHDL entity or Verilog module names. The default string is `_entity`.

For example, if the Filter Design HDL Coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_entity`.

**See Also** `ClockProcessPostfix`, `CoeffName`, `PackagePostfix`, `ReservedWordPostfix`

# ErrorMargin

---

<b>Purpose</b>	Specify an error margin for HDL language-based test benches
<b>Settings</b>	<p>n</p> <p>Specify an integer indicating an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.</p>
<b>Usage Notes</b>	<p>Properties that provide optimizations can generate test bench code that produces numeric results that differ from those produced by the original MATLAB filter function. Specifically, these options include</p> <ul style="list-style-type: none"><li>• CastBeforeSum (qfilts only)</li><li>• OptimizeForHDL</li><li>• CoeffMultipliers</li><li>• FIRAdderStyle ('Tree')</li><li>• AddPipelineRegisters (for FIR, Asymmetric FIR, and Symmetric FIR filters)</li></ul> <p>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.</p>
<b>See Also</b>	AddPipelineRegisters, CastBeforeSum, CoeffMultipliers, FIRAdderStyle, OptimizeForHDL

**Purpose** Specify a final summation technique to be used for FIR filters

**Settings** 'linear' (default)  
Apply linear adder summation, which is discussed in most DSP text books. This is the default.

'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 to be applied to the filter. By default, the Filter Design HDL Coder applies linear adder summation. 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. For information on applying pipeline mode, see *Optimizing the Clock Rate with Pipeline Registers*.

### **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 ensures numeric accuracy in comparison to the original MATLAB filter function. Tree and pipeline modes can produce numeric results that differ from those produced by the filter function.

**See Also** AddPipelineRegisters, CoeffMultipliers, OptimizeForHDL

# ForceClock

---

## **Purpose**

Specify whether the test bench is to force clock input signals

## **Settings**

'on' (default)

Specify that the test bench is to force the clock input signals. This is the default. When this option is set, the clock high and low time settings control the clock waveform.

'off'

Specify that a user-defined external source is to force the clock input signals.

## **See Also**

ClockEnableValue, ClockHighTime, ClockLowTime, ForceClockEnable, ForceReset, HoldTime, ResetValue

<b>Purpose</b>	Specify whether the test bench is to force clock enable input signals
<b>Settings</b>	<p>'on' (default) Specify that the test bench is to force the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value. This is the default.</p> <p>'off' Specify that a user-defined external source is to force the clock enable input signals.</p>
<b>See Also</b>	ClockEnableValue, ClockHighTime, ClockLowTime, ForceClock, ForceReset, HoldTime, ResetValue

# ForceReset

---

## **Purpose**

Specify whether the test bench is to force reset input signals

## **Settings**

'on' (default)

Specify that the test bench is to force the reset input signals. This is the default. 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 is to force the reset input signals.

## **See Also**

ClockEnableValue, ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, HoldTime, ResetValue



**Purpose** Specify the 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 to be held past the clock rising edge. The default is 2.

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 to be held past the clock rising edge. The following figures show the application of a hold time ( $t_{\text{hold}}$ ) for reset and data input signals when the signals are forced to active high and active low.

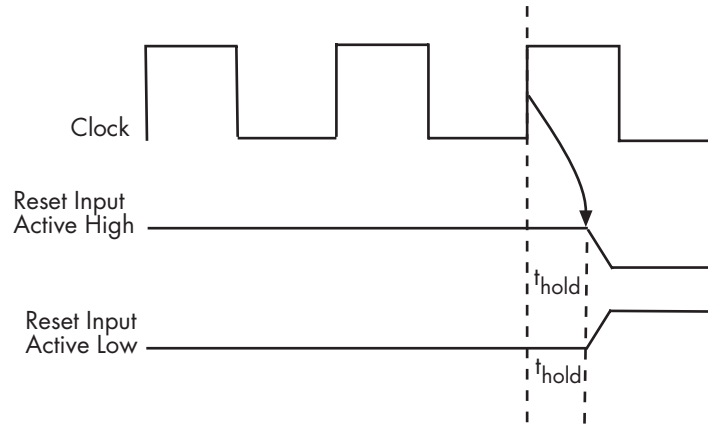
---

**Note** A reset signal is always asserted for two cycles plus  $t_{\text{hold}}$ .

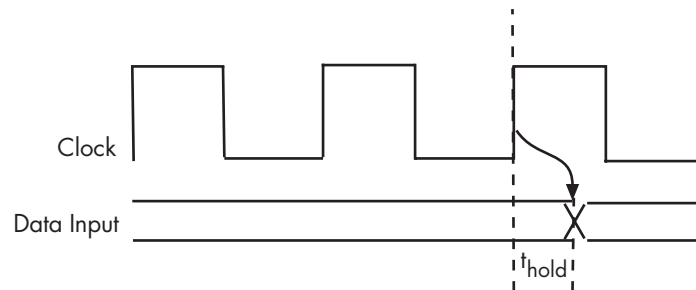
---

# HoldTime

---



## Hold Time for Reset Input Signals



## Hold Time for Data Input Signals

### See Also

ClockEnableValue, ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, ForceReset, ResetValue

**Purpose** Specify whether signals of type REAL are to be declared with an initial value of 0.0

**Settings** 'on' (default)  
Initialize signals of type REAL with a value of 0.0. This is the default.

The following line of VHDL code is an example of a declaration that results when this option is set:

```
SIGNAL a1sum1 : REAL := 0.0; -- double
```

'off'

Suppress the initialization of signals of type REAL.

**See Also** CastBeforeSum, InlineConfigurations, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge, UseVerilogTimescale

# InlineConfigurations

---

**Purpose** Specify whether generated VHDL code is to include inline configurations

**Settings** 'on' (default)  
Include VHDL configurations in any file that instantiates a component. This is the default.

'off'  
Suppress the generation of configurations and require user-supplied external configurations.

**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 Filter Design HDL 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, InitializeRealSignals, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge

**Purpose** Name the HDL port for a filter's input signals

**Settings** 'string'  
Name the HDL port for the filter's data input signals. 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

---

## **Purpose**

Specify the HDL data type for the filter's input port

## **Settings**

'std\_logic\_vector' (VHDL default)

Specify that the filter's input port be of VHDL type  
STD\_LOGIC\_VECTOR.

'signed\_unsigned'

Specify that the filter's input port be of VHDL type SIGNED or  
UNSIGNED.

'wire' (Verilog)

Cannot modify.

## **See Also**

ClockEnableInputPort, ClockInputPort, InputPort, OutputPort,  
OutputType, ResetInputPort

<b>Purpose</b>	Specify a string to be appended to instance section labels in VHDL GENERATE statements
<b>Settings</b>	'string' Specify a postfix to be appended to instance section labels for VHDL GENERATE statements. The default string is <code>_gen</code> .
<b>See Also</b>	BlockGenerateLabel, OutputGenerateLabel

# LoopUnrolling

---

<b>Purpose</b>	Specify whether VHDL FOR and GENERATE loops are to be unrolled and excluded from the generated VHDL code
<b>Settings</b>	<p>'on'</p> <p>Unroll and omit FOR and GENERATE loops from the generated VHDL code. Verilog is always unrolled.</p> <p>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.</p> <p>'off' (default)</p> <p>Include FOR and GENERATE loops in the generated VHDL code. This is the default.</p>
<b>Usage Notes</b>	The setting of this option does not affect generated VHDL code during simulation or synthesis.
<b>See Also</b>	CastBeforeSum, InlineConfigurations, InitializeRealSignals, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge



**Purpose**

Name a VHDL entity or Verilog module for the filter and the file that is to contain the generated HDL code

**Settings**

'string'

Name a VHDL entity or Verilog module and the file that is to contain the filter's HDL code. The default string is the name of the filter as defined in the context of the FDATool.

The file type extension is defined by the file extension option for the selected language and the file is placed in the directory defined by the specified target directory.

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`. See `ReservedWordPostfix` for more information.

**See Also**

`VerilogFileExtension`, `VHDLFileExtension`, `TargetDirectory`

# OptimizeForHDL

---

<b>Purpose</b>	Specify whether generated HDL code is to be optimized for specific performance or space requirements
<b>Settings</b>	<p>'on'</p> <p>Generate HDL code that is optimized for specific performance or space requirements at the cost of the Filter Design HDL Coder:</p> <ul style="list-style-type: none"><li>• Making tradeoffs concerning data types</li><li>• Avoiding excessive quantization</li><li>• Generating code that produces numeric results that differ from results produced by the original MATLAB filter function</li></ul> <p>'off' (default)</p> <p>Generate HDL code that maintains bit compatibility with the numeric results produced by the specified quantized filter in MATLAB. This is the default.</p>
<b>See Also</b>	AddPipelineRegisters, CoeffMultipliers, FIRAdderStyle

<b>Purpose</b>	Specify a string that labels an output assignment block for VHDL GENERATE statements
<b>Settings</b>	'string' Specify a postfix to be appended to output assignment block labels in VHDL GENERATE statements. The default string is outputgen.
<b>See Also</b>	BlockGenerateLabel, InstanceGenerateLabel

# OutputPort

---

**Purpose** Name the HDL port for a filter's output signals

**Settings** 'string'  
Name the HDL port for the filter's data output signals. 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`

<b>Purpose</b>	Specify the HDL data type for the filter's output port
<b>Settings</b>	'std_logic_vector' (VHDL default) Specify that the filter's output port be of VHDL type STD_LOGIC_VECTOR.  'signed_unsigned' Specify that the filter's input port be of type SIGNED or UNSIGNED.  'wire' (Verilog) Cannot modify.
<b>See Also</b>	ClockEnableInputPort, ClockInputPort, InputPort, InputType, OutputPort, ResetInputPort

# PackagePostfix

---

**Purpose** Specify a string to be appended to the specified filter name to form the name of a VHDL package file

**Settings** 'string'  
Specify a string to be appended to the value of the name option to form the name of a VHDL package file. The coder applies this option only if a package file is required for the design. The default string is `_pkg`.

**See Also** `ClockProcessPostfix`, `CoeffName`, `EntityConflictPostfix`, `ReservedWordPostfix`

**Purpose** Specify a string to be appended to strings that you specify with other options that are VHDL or Verilog reserved words

**Settings** 'string'  
Specify a string to be appended to the value names, postfix values, or labels that are VHDL or Verilog reserved words. The default is `_rsvd`.

For example, if you name your filter `mod`, the Filter Design HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

**See Also** `ClockProcessPostfix`, `CoeffName`, `EntityConflictPostfix`, `PackagePostfix`

# ResetAssertedLevel

---

<b>Purpose</b>	Specify the asserted (active) level of the reset input signal
<b>Settings</b>	'active_high' (default) Specify that the reset input signal must be driven high (1) to reset registers in the filter design.  'active_low' Specify that the reset input signal must be driven low (0) to reset registers in the filter design.
<b>Usage Notes</b>	<p>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. For example, the following code fragment checks whether reset is active high before populating the delay_pipeline register:</p> <pre>Delay_Pipeline_Process : PROCESS (clk, reset) BEGIN     IF reset = '1' THEN         delay_pipeline(0 TO 50) &lt;= (OTHERS =&gt; (OTHERS =&gt; '0'));     .     .     .</pre> <p>If you change the setting to active low, the IF statement in the preceding generated code changes to</p> <pre>IF reset = '0' THEN</pre>
<b>See Also</b>	ResetType



## Purpose

Name the HDL port for a filter's reset input signals

## Settings

'string'

Name the HDL port for the filter's reset signals. The default 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`

# ResetType

---

**Purpose** Specify whether an asynchronous or synchronous reset style is to be used when generating HDL code for registers

**Settings** 'sync'  
Use a synchronous reset style. When this style is active, clock process blocks check for a clock event before performing a reset.

'async' (default)  
Use an asynchronous reset style. This is the default.

**Usage Notes** Whether you should set the reset style to asynchronous or synchronous depends on the type of device you are designing (for example, FPGA or ASIC) and preference.

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 rising_edge(clk) THEN
    IF reset = '0' 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;
```

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 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;
```

## See Also

ResetAssertedLevel

# ResetValue

---

**Purpose** Specify the constant value to which the test bench is to force reset input signals

**Settings** 'active\_high' (default)  
Specify that the test bench set the reset input signal to active high (1). This is the default.

'active\_low'  
Specify that the test bench set the reset input signal to active low (0).

**Usage Notes** The setting for this option must match the setting of the reset asserted level specified for the filter's test bench. Also note that the Filter Design HDL Coder ignores the setting of this option if forced resets are disabled.

**See Also** ClockEnableValue, ClockHighTime, ClockLowTime, ForceClock, ForceClockEnable, ForceReset, HoldTime

**Purpose** Specify the syntax to be used in the generated VHDL code for concatenated zeros

**Settings**

'on' (default)  
Use the type safe syntax, '0' & '0', for concatenated zeros. Typically, this syntax is preferred. This is the default.

'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, InitializeRealSignals, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge

# ScaleWarnBits

---

**Purpose** Control whether the coder generates a warning for scale values that are below a specified numeric threshold relative to the input data format

**Settings** `n`  
Specify a numeric value the coder is to use as a minimum overlap threshold between input data and scale values converted to the input data format before issuing warnings that suggest quantization noise. The default minimum is 3 bits.

To suppress the warnings, specify a value that equals the number of bits in the input format.

**Usage** Use this option for fixed-point filters when you need to control whether the coder generates a warning for scale values that are below a specified numeric threshold relative to the input data format. These warnings help identify scale values that cause the input range to be quantized to near zero, adding quantization noise.

You can control the warnings by adjusting an overlap threshold. The coder temporarily converts a scale value to the data type of the filter input. Then, the coder checks whether the number of leading zeros in the converted value is greater than or equal to the specified overlap threshold. If this condition exists, the coder generates a warning.

You can prevent the coder from generating these warnings by setting the minimum overlap to the number of bits in the input format. However, if the converted scale value equals zero, the coder reports an error because the input range is quantized away.

For examples, see “Minimizing Quantization Noise for Fixed-Point Filters” on page 3-43.

**See Also** `CastBeforeSum`, `InlineConfigurations`, `InitializeRealSignals`, `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`, `UseVerilogTimescale`

<b>Purpose</b>	Specify simulator flags that are to be applied to your generated test bench
<b>Settings</b>	'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.
<b>Usage Notes</b>	The flags you specify with this option are added to the vcom command in generated ModelSim .do test bench files.

# SplitArchFilePostfix

---

<b>Purpose</b>	Specify a string to be appended to the specified name to form the name of the file that is to contain the filter's VHDL architecture
<b>Settings</b>	'string' Name the postfix to be appended to the name of the file containing the filter's VHDL architecture. The default is <code>_arch</code> .
<b>Usage Notes</b>	The option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.
<b>See Also</b>	<code>SplitEntityArch</code> , <code>SplitEntityFilePostfix</code>



**Purpose** Specify whether the generated VHDL entity and architecture code is to be included in a single VHDL file or separate files

## Settings

'on'

Write the generated filter VHDL code to a single file.

'off' (default)

Write the code for the filter VHDL entity and architecture to separate files.

If you separate the code, the Filter Design HDL Coder derives the names of the entity and architecture files from

- The base filename, as specified by the filter name option
- Default postfix strings `_entity` and `_arch` or user-specified postfix strings
- The VHDL file type extension, as specified by the VHDL file extension option

For example, instead of all generated code residing in `MyFIR.vhd`, you can specify that the code reside in `MyFIR_entity.vhd` and `MyFIR_arch.vhd`.

## See Also

`SplitArchFilePostfix`, `SplitEntityFilePostfix`

# SplitEntityFilePostfix

---

<b>Purpose</b>	Specify a string to be appended to the specified filter name to form the name of the file that is to contain the filter's VHDL entity
<b>Settings</b>	'string' Name the postfix to be appended to the name of the file containing the filter's VHDL entity. The default is <code>_entity</code> .
<b>Usage Notes</b>	This option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.
<b>See Also</b>	<code>SplitEntityArch</code> , <code>SplitArchFilePostfix</code>

<b>Purpose</b>	Identify the directory into which generated output files are to be written
<b>Settings</b>	'string' Name the subdirectory under the current working directory into which generated files are to be written. The string can specify a complete pathname. The default string is hdlsrc.
<b>See Also</b>	Name, VerilogFileExtension, VHDLFileExtension

# TargetLanguage

---

**Purpose** Specify the HDL language to use for the generated filter code

**Settings**

- 'VHDL' (default)  
Generate VHDL filter code.
- 'verilog'  
Generate Verilog filter code.

**Purpose** Name a VHDL test bench entity or Verilog module and the file that is to contain test bench code

**Settings** 'string'  
Name a VHDL test bench entity or Verilog module and the file that is to contain the test bench code. The file type extension depends on the type of test bench that is being generated.

<b>If the Test Bench Is a...</b>	<b>The Extension Is...</b>
Verilog file	Defined by the Verilog file extension option
VHDL file	Defined by the VHDL file extension option
ModelSim DO file	.do

The file is placed in the directory defined by the specified target directory.

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`. See `ReservedWordPostfix` for more information.

**See Also** `ClockEnableValue`, `ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`, `TestBenchName`

# TestBenchStimulus

---

**Purpose** Specify the input stimuli the test bench is to apply to the filter

**Settings**

'impulse'  
Specify that the test bench acquire an impulse stimulus response, which 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.

Default settings depend on the structure of the filter.

<b>For Filters...</b>	<b>Default Responses Include...</b>
FIR, FIRT, Symmetric FIR, and Antisymmetric FIR	Impulse, step, ramp, chirp, and white noise
All others	Step, ramp, and chirp

**Usage Notes** You can specify any combination of stimuli in any order. If you specify multiple stimuli, specify the appropriate strings in a cell array. For example:

```
{'impulse', 'ramp', 'noise'}
```

**See Also** TestBenchUserStimulus

**Purpose** Specify a user-defined MATLAB function that returns a vector of values that the test bench is to apply to the filter

**Settings** M-function  
Specify a MATLAB function that returns a vector of stimulus response values that the test bench is to apply to the filter. The filter's input quantizer settings are used to quantize and scale the function's input values.

For example, the following MATLAB 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

---

**Purpose** Specify whether all constants should be represented by aggregates, including constants that are less than 32 bits

**Settings** 'on'  
Specify that all 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 coeff1 :signed(15 DOWNT0 0) := (4 DOWNT0 2 => '0', 0 =>'0',  
OTHERS => ', '); -- sfix16_En15  
CONSTANT coeff2 :signed(15 DOWNT0 0) := (6 => '0', 4 DOWNT0 3 => '0',  
OTHERS => ', '); -- sfix16_En15
```

'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(-30, 16); -- sfix16_En15  
CONSTANT coeff2 :signed(15 DOWNT0 0) := to_signed(-89, 16); -- sfix16_En15
```

**See Also** CastBeforeSum, InlineConfigurations, InitializeRealSignals, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge, UseVerilogTimescale



## Purpose

Specify a string to be added as a comment line in the header of generated filter and test bench files

## Settings

'string'

Specify a string that is to be added to the end of the comment header of the generated filter and test bench files.

For example, you might use this option to add the revision control tag `$Revision: 1.1.2.2 $`. The resulting header comment block for filter Hd would appear as follows:

```
-----  
--  
-- Module:Hd  
--  
-- Generated by MATLAB(R) 7.0 and the Filter Design HDL Coder 1.0.  
--  
-- Generated on: 2004-02-04 09:42:43  
--  
-- $Revision: 1.1.4.7 $  
-----
```

# UseRisingEdge

---

**Purpose** Specify the VHDL coding style to be 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) <= dleay_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) <= dleay_pipeline(0 TO 49);
        END IF;
    END IF;
END PROCESS Delay_Pipeline_Process ;
```

```
        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, InitializeRealSignals, LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge

# UseVerilogTimescale

---

**Purpose** Specify whether generated Verilog code can include compiler ``timescale` directives

**Settings**

'on' (default)  
Use compiler ``timescale` directives in generated Verilog code.  
This is the default.

'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`, `InitializeRealSignals`, `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

<b>Purpose</b>	Specify the string to be used as the file type extension for generated Verilog files
<b>Settings</b>	'string' Specify a file type extension for generated Verilog files. The default string is <code>.v</code> .
<b>See Also</b>	Name, TargetDirectory

# VHDLFileExtension

---

**Purpose** Specify the string to be used as the file type extension for generated VHDL files

**Settings** 'string'  
Specify a file type extension for generated VHDL files. The default is .vhd

**See Also** Name, TargetDirectory

# Functions — Alphabetical List

---

# generatehdl

---

**Purpose** Generate HDL code for a quantized filter

**Syntax** generatehdl(Hd)  
generatehdl(Hd 'PropertyName', 'PropertyValue',...)

**Description** generatehdl(Hd) generates HDL code for the quantized filter identified by Hd. The function uses default settings for properties that determine file and HDL element naming, whether optimizations are applied, HDL coding styles, and test bench characteristics. By default the function does the following.

## Default Settings

- Places generated files in the target directory `hdlsrc` and names the files as follows:

<b>File</b>	<b>Name</b>
Verilog source	<i>Hd.v</i> , where <i>Hd</i> is the name of the specified filter object
VHDL source	<i>Hd.vhd</i> , where <i>Hd</i> is the name of the specified filter object
VHDL package	<i>Hd_pkg.vhd</i> , where <i>Hd</i> is the name of the specified filter object

- Places generated files in a subdirectory name `hdlsrc`, under your current working directory.
- Includes the 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.
- Uses an active-high (1) signal for register resets.



**Default Settings for General HDL Code**

- Names the generated VHDL entity or Verilog module with the name of the quantized filter.
- Names a filter’s HDL ports as follows:

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

- 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:

<b>For...</b>	<b>Names Coefficients...</b>
FIR filters	coeff $n$ , where $n$ is the coefficient number, starting with 1
IIR filters	coeff $_{xm\_sectionn}$ , where $x$ is a or b, $m$ 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 any file that instantiates 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 a minimum of 3 bits of filter input and coefficient scale values to overlap before a warning is issued.
- 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 MATLAB filter function and is *not* optimized for performance or space requirements.
- Applies a linear final summation to FIR filters. This is the form of summation explained in most DSP text books.
- Enables multiplier operations for a filter, as opposed to replacing them with additions of partial products.

generatehdl(Hd 'PropertyName', 'PropertyValue',...) generates HDL code for the 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 Chapter 5, “Properties — Categorical List” and Chapter 6, “Properties — Alphabetical List”.

## Example

- 1 **Design a filter.** The call to firceqrip in the following command sequence designs an equiripple lowpass finite impulse response (FIR) filter with linear phase, an order of 30, a cutoff frequency of 0.4,

and maximum passband and stopband errors set to 0.05 and 0.03, respectively. The design results are returned to the cell array `h`.

- 2 Construct a quantized filter.** The call to `dfilt` constructs a quantized FIR filter `Hd` with reference coefficients specified by the cell array `h`.
- 3 Set the filter arithmetic.** The arithmetic assignment statement sets the filter arithmetic to fixed-point arithmetic.
- 4 Override the FDATool typecasting setting.** By default, the FDATool enable typecasting of input values before addition and subtraction operations occur. For hardware efficiency, it is best to override this setting and disable typecasting.
- 5 Generate HDL code for the filter.** The call to `generatehdl` generates HDL code for the quantized filter `Hd`. The function names the file `MyFilter.vhd` and places it in the default target directory `hdlsrc`.

```
h=firceqrip(30,0.4,[0.05 0.03]); %Design a filter
Hd= dfilt.dffir(h); %Construct a quantized filter
Hd.arithmetic='fixed'; %Quantized filter with default settings
Hd.castbeforesum=false; %Improves hardware efficiency
generatehdl(Hd, 'Name', 'MyFilter'); %Generate filter's VHDL code
```

## See Also

`generatetb`, `generatetbstimulus`

# generatetb

---

**Purpose** Generate an HDL test bench for a quantized filter

**Syntax**  
`generatetb(Hd, 'TbType')`  
`generatehdl(Hd 'TbType', 'PropertyName', 'PropertyValue',...)`

**Description** `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 value that you specify for 'TbType' identifies the type of test bench to be generated and can be one of the following values or a cell array that contains one or more of the following values:

<b>Specify...</b>	<b>To Generate a Test Bench Consisting of...</b>
'Verilog'	Verilog code
'VHDL'	VHDL code
'ModelSim'	ModelSim script file

The generated test bench applies input stimuli based on the setting of the properties `TestBenchStimulus` and `TestBenchUserStimulus`. By default, `TestBenchStimulus` specifies impulse, step, ramp, chirp, and noise stimuli for FIR, FIRT, Symmetric FIR, and Antisymmetric FIR filters and step, ramp, and chirp stimuli for all other filters.

The function uses default settings for other properties that determine test bench characteristics. By default the function does the following.

### **Default Settings for the Test Bench**

- Places the generated test bench file in the target directory `hdlsrc` under your current working directory with the name `Hd_tb` and a file type extension that is based on the type of test bench you are generating.

<b>If the Test Bench Is a...</b>	<b>The Extension Is...</b>
Verilog file	Defined by the property VerilogFileExtension
VHDL file	Defined by the property VHDLFileExtension
ModelSim script file	.do

- 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 directory `hdlsrc` and names the files as follows:

<b>File</b>	<b>Name</b>
Verilog source	<i>Hd.v</i> , where <i>Hd</i> is the name of the specified filter object
VHDL source	<i>Hd.vhd</i> , where <i>Hd</i> is the name of the specified filter object
VHDL package	<i>Hd_pkg.vhd</i> , where <i>Hd</i> is the name of the specified filter object

- Places generated files in a subdirectory name `hdlsrc`, under your current working directory.

- 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:

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

- 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

`coeff $n$` , where  $n$  is the coefficient number, starting with 1

IIR filters

`coeff $_{xm\_section}$  $n$` , where  $x$  is a or b,  $m$  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 any file that instantiates 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 MATLAB filter function and is *not* optimized for performance or space requirements.

- Applies a linear final summation to FIR filters. This is the form of summation explained in most DSP text books.
- Enables multiplier operations for a filter, as opposed to replacing them with additions of partial products.

`generatehdl(Hd 'TbType', 'PropertyName', 'PropertyValue',...)` generates a 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 Chapter 5, “Properties — Categorical List” and Chapter 6, “Properties — Alphabetical List”.

## Example

- 1 Design a filter.** The call to `firceqrip` in the following command line sequence designs an equiripple lowpass finite impulse response (FIR) filter with linear phase, an order of 30, a cutoff frequency of 0.4, and maximum passband and stopband errors set to 0.05 and 0.03, respectively. The design results are returned to the cell array `h`.
- 2 Construct a quantized filter.** The call to `dfilt` constructs a quantized FIR filter `Hd` with reference coefficients specified by the cell array `h` returned by `firceqrip`.
- 3 Set the filter arithmetic.** The arithmetic assignment statement sets the filter arithmetic to fixed-point arithmetic.
- 4 Override the FDATATool typecasting setting.** By default, the FDATATool enable typecasting of input values before addition and subtraction operations occur. For hardware efficiency, it is best to override this setting and disable typecasting.
- 5 Generate VHDL code for the filter.** The call to `generatehdl` generates VHDL code for the quantized filter `Hd`. The function names the file `MyFilter.vhd` and places it in the default target directory `hdlsrc`.
- 6 Generate a test bench for the filter.** The call to `generatetb` generates a ModelSim VHDL test bench for the filter `Hd` named



Hd\_tb.do and places the generated test bench file in the default target directory hdlsrc.

```
h=firceqrip(30,0.4,[0.05 0.03]); %Design a filter
Hd= dfilt.dffir(h); %Construct a quantized filter
Hd.arithmetic='fixed'; %Quantized filter with default settings
Hd.castbeforesum=false; %Improves hardware efficiency
generatehdl(Hd, 'Name', 'MyFilter'); %Generate filter's VHDL code
generatetb(Hd, 'ModelSim', 'TestBenchName', 'MyFilterTB');
```

**See Also**

generatetbstimulus, generatehdl

# generatetbstimulus

---

**Purpose** 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` based on the setting of the properties `TestBenchStimulus` and `TestBenchUserStimulus`. By default, `TestBenchStimulus` specifies impulse, step, ramp, chirp, and noise stimuli for FIR, FIRT, Symmetric FIR, and Antisymmetric FIR filters, and step, ramp, and chirp stimuli for all other filters.

---

**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` based on specified settings for `TestBenchStimulus` and `TestBenchUserStimulus`.  
`x = generatetbstimulus(Hd, 'PropertyName', 'PropertyValue'...)` generates and returns filter input stimuli for the filter `Hd` based on specified settings for `TestBenchStimulus` and `TestBenchUserStimulus` and writes the output to `x` for future use or reference.

**Example**

- 1 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`.
- 2 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.

```
y = generatetbstimulus(Hd, 'TestBenchStimulus', {'ramp', 'chirp'});  
%Generate and return test bench stimuli  
plot(filter(Hd,y)); %Apply a quantized filter to the  
data and plot the results
```

## See Also

generatetb



# Examples

---

## **Tutorials**

“Basic FIR Filter Tutorial” on page 2-3

“Optimized FIR Filter Tutorial” on page 2-23

“IIR Filter Tutorial” on page 2-43

## **Basic FIR Filter Tutorial**

“Designing a Basic FIR Filter” on page 2-3

“Quantizing the Basic FIR Filter” on page 2-5

“Configuring and Generating the Basic FIR Filter’s VHDL Code” on page 2-8

“Getting Familiar with the Basic FIR Filter’s Generated VHDL Code” on page 2-15

“Verifying the Basic FIR Filter’s Generated VHDL Code” on page 2-16

## **Optimized FIR Filter Tutorial**

“Designing the FIR Filter” on page 2-23

“Quantizing the FIR Filter” on page 2-25

“Configuring and Generating the FIR Filter’s Optimized Verilog Code”  
on page 2-28

“Getting Familiar with the FIR Filter’s Optimized Generated Verilog  
Code” on page 2-35

“Verifying the FIR Filter’s Optimized Generated Verilog Code” on page 2-37



## **IIR Filter Tutorial**

“Designing an IIR Filter” on page 2-43

“Quantizing the IIR Filter” on page 2-45

“Configuring and Generating the IIR Filter’s VHDL Code” on page 2-49

“Getting Familiar with the IIR Filter’s Generated VHDL Code” on page 2-55

“Verifying the IIR Filter’s Generated VHDL Code” on page 2-56



## A

- Add input register option 3-42
- Add output register option 3-42
- Add pipeline registers option 3-58
- AddInputRegister property 6-2
- addition operations
  - specifying input type treatment for 3-52
  - type casting 6-7
- AddOutputRegister property 6-3
- AddPipelineRegisters property 6-4
- advanced coding properties 5-4
- Advanced tab 3-43
- antisymmetric FIR filters 1-6
- application-specific integrated circuits (ASICs) 1-2
- architectures
  - setting postfix for from command line 6-46
  - setting postfix for from GUI 3-23
- ASICs (application-specific integrated circuits ) 1-2
- asserted level, reset 3-28
  - setting 6-38
- asynchronous resets
  - setting from command line 6-40
  - setting from GUI 3-26

## B

- block labels
  - for GENERATE statements 6-6
  - for output assignment blocks 6-33
- BlockGenerateLabel property 6-6

## C

- canonical signed digit (CSD) technique 3-55
- Cast before sum option 3-52
- CastBeforeSum property 6-7
- checklist
  - requirements 3-13

- clock
  - configuring for test benches 3-65
    - specifying high time for 6-10
    - specifying low time for 6-12
  - clock enable input port
    - naming 3-38 6-8
    - specifying forced signals for 6-21
  - Clock enable port options 3-38
  - clock enable value 3-65 6-9
  - Clock enable value option 3-65
  - Clock high time option 3-65
  - clock input port
    - naming 3-38 6-11
    - specifying forced 6-20
  - Clock low time 3-65
  - Clock port options 3-38
  - clock process names
    - specifying postfix for 6-13
  - clock time
    - configuring 3-65
    - high 6-10
    - low 6-12
  - clocked process block labels 3-37
  - Clocked process postfix option 3-37
  - ClockEnableInputPort property 6-8
  - ClockEnableValue property 6-9
  - ClockHighTime property 6-10
  - ClockInputPort property 6-11
  - ClockLowTime property 6-12
  - ClockProcessPostfix property 6-13
- code, generated 3-74
  - advanced properties for customizing 5-4
  - compiling 4-7 4-16
  - configuring for basic FIR filter tutorial 2-8
  - configuring for IIR filter tutorial 2-49
  - configuring for optimized FIR filter tutorial 2-28
  - customizing 3-29
  - defaults for 3-9
  - for filter and test bench 4-3

- general HDL defaults 3-10
- optimizing 3-55
- reviewing for basic FIR filter tutorial 2-15
- reviewing for IIR filter tutorial 2-55
- reviewing for optimized FIR filter tutorial 2-35
- verifying for basic FIR filter tutorial 2-16
- verifying for IIR filter tutorial 2-56
- verifying for optimized FIR filter tutorial 2-37
- Coeff multipliers option 3-55
- coefficient multipliers 3-55
- Coefficient name option 3-32
- coefficients
  - naming 6-15
  - specifying a for 3-32
- CoeffMultipliers property 6-14
- CoeffName property 6-15
- command line interface 1-5
  - generating filter and test bench code with 4-3
- command, fdatool 3-4
- Comment in header option 3-30
- comments, header
  - as property value 6-55
  - specifying 3-30
- Concatenate type safe zeros 3-49
- configurations, inline
  - suppressing from command line 6-26
  - suppressing from GUI 3-48
- constants
  - setting representation from command line 6-54
  - setting representation from GUI 3-45
- context-sensitive help 1-11
- CSD technique 3-55

**D**

- data input port

- naming from command line 6-27
- naming from GUI 3-38
- specifying hold time for from GUI 3-69
- specifying hold time for with command line 6-23
- data output port
  - specifying name from command line 6-34
  - specifying name from GUI 3-38
- defaults
  - for general HDL code 3-10
  - for generated files 3-9
  - for optimizations 3-11
  - for resets 3-10
  - for test benches 3-12
- demos 1-12
- dialogs
  - Generate HDL
    - description 1-4
    - opening 3-4
    - setting optimizations with 3-54
    - setting test bench options with 3-61
    - specifying test bench type with 3-62
  - HDL Options 3-29
  - Test Bench Options 3-61
- Direct Form I filters 1-6
- Direct Form II filters 1-6
- directory, target 6-49

**E**

- entities
  - name conflicts of 3-33
  - naming 6-31
  - setting names of 3-20
  - setting postfix for from command line 6-48
  - setting postfix for from GUI 3-23
- Entity conflict postfix option 3-33
- entity name conflicts 6-17
- EntityConflictPostfix property 6-17
- error margin

- specifying from command line 6-18
- specifying from GUI 3-70
- Error margin option 3-70
- ErrorMargin property 6-18

## F

- factored CSD technique 3-55
- FDATool 1-4
- fdatool command 3-4
- features 1-3
- field programmable gate arrays (FPGAs) 1-2
- file extensions
  - setting 3-20
  - Verilog 6-59
  - VHDL 6-60
- file location properties 5-2
- file naming properties 5-2
- filenames
  - defaults for 3-9
  - for architectures 6-46
  - for entities 6-48
  - for generated output 1-8
- files, generated
  - default names for 3-9
  - defaults for 3-9
  - HDL output 1-8
  - setting architecture postfix for 3-23
  - setting entity postfix for 3-23
  - setting location of 3-21
  - setting names of 3-20
  - setting options for 3-19
  - setting package postfix for 3-22
  - splitting 6-47
  - test bench 6-51
- filter arithmetic 3-4
- Filter Design HDL Coder
  - applying to hardware design process 1-13
  - as FDATool plug-in 1-4
  - command line interface 1-5
  - features of 1-3
  - graphical user interface 1-4
  - prerequisite knowledge for 1-3
  - user profiles for 1-3
  - what is 1-2
  - workflow 1-13
- filter input 6-44
- filter structures 1-6
- Filter target language option 3-18
- filters
  - designing in basic FIR tutorial 2-3
  - designing in IIR filter tutorial 2-43
  - designing in optimized FIR filter tutorial 2-23
  - generated HDL output for 1-8
  - naming generated file for 6-31
  - properties of 1-7
  - quantized 1-6
  - quantizing 3-4
  - quantizing in basic FIR filter tutorial 2-5
  - quantizing in IIR filter tutorial 2-45
  - quantizing in optimized FIR filter tutorial 2-25
  - realizations of 1-6
- finite impulse response (FIR) filters 1-6
- FIR adder style option 3-57
- FIR filter tutorial
  - basic 2-3
  - optimized 2-23
- FIR filters 1-6
  - optimizing clock rate for 3-58
  - optimizing final summation for 3-57
  - specifying summation technique for 6-19
- FIRAdderStyle property 6-19
- Force clock enable option 3-65
- Force clock option 3-65
- force reset hold time 6-23
- Force reset option 3-67
- ForceClock property 6-20
- ForceClockEnable property 6-21

- ForceReset property 6-22
- FPGAs (field programmable gate arrays) 1-2
- functions
  - generatehdl 7-2
  - generatetb 7-6
  - generatetbstimulus 7-12
  - input parameters for 1-7

## **G**

- General tab 3-32
- Generate HDL dialog
  - defaults 3-9
  - description 1-4
  - opening 3-4
  - setting optimizations with 3-54
  - specifying test bench type with 3-62
- generatehdl function 7-2
- generatetb function 7-6
- generatetbstimulus function 7-12
- graphical user interface 1-4

## **H**

- hardware description languages (HDLs) 1-2
  - See also* Verilog; VHDL
- HDL code 2-8
  - See also* code, generated
- HDL files 1-8
- HDL language 3-18
- HDL Options dialog 3-29
- HDL test benches 4-3
- HDLs (hardware description languages) 1-2
  - See also* Verilog; VHDL
- header comment properties 5-3
- header comments 3-30
- help
  - context-sensitive 1-11
  - getting 1-10
- Help browser 1-11

- hold time 6-23
  - for data input signals 3-69
  - for resets 3-67
- Hold time option
  - for data input signals 3-69
  - for resets 3-67
- HoldTime property 6-23

## **I**

- IIR filter tutorial 2-43
- IIR filters 1-6
  - optimizing clock rate for 3-58
- infinite impulse response (IIR) filters 1-6
- Initialize real signals to 0.0 option 3-51
- InitializeRealSignals property 6-25
- inline configurations
  - specifying 6-26
  - suppressing the generation of 3-48
- Inline VHDL configurations option 3-48
- InlineConfigurations property 6-26
- input data overlay with scale values 3-43
- Input data type option 3-40
- input parameters 1-7
- Input port option 3-38
- input ports
  - naming 3-38
  - specifying data type for 6-28
- input registers
  - adding code for 6-2
  - suppressing generation of extra 3-42
- InputPort property 6-27
- InputType property 6-28
- installation 1-9
- instance sections 6-29
- InstanceGenerateLabel property 6-29

## **L**

- labels

- block 6-33
  - specifying postfix for 6-6
  - process block 3-37
- language
  - setting target 3-18
  - target 6-50
- language selection properties 5-2
- linear FIR final summation 3-57
- Loop unrolling option 3-46
- loops
  - unrolling 6-30
  - unrolling and removing 3-46
- LoopUnrolling property 6-30

## M

- M-help 1-11
- Minimum overlap of scale values option 3-43
- ModelSim 4-15
- ModelSim DO file
  - executing 4-17
  - testing with 4-12
- ModelSim DO file test benches 3-62
- module name conflicts 6-17
- modules
  - name conflicts for 3-33
  - naming 6-31
  - setting names of 3-20
- multipliers
  - optimizing coefficient 3-55

## N

- name conflicts 6-17
- Name option 3-20
- Name property 6-31
- names
  - clock process 6-13
  - coefficient 3-32
  - package file 6-36

- naming properties 5-3

## O

- optimization properties 5-6
- optimizations
  - defaults for 3-11
  - for synthesis 3-59
  - HDL code 3-55 6-32
  - setting 3-54
- Optimize for HDL option 3-55
- optimized FIR filter tutorial 2-23
- OptimizeForHDL property 6-32
- options
  - Add input register 3-42
  - Add output register 3-42
  - Add pipeline registers 3-58
  - Cast before sum 3-52
  - Clock enable input port 3-38
  - Clock enable value 3-65
  - Clock high time 3-65
  - Clock low time 3-65
  - Clock port 3-38
  - Clocked process postfix 3-37
  - Coeff multipliers 3-55
  - Coefficient name 3-32
  - Comment in header 3-30
  - Concatenate type safe zeros 3-49
  - Entity conflict postfix 3-33
  - Error margin 3-70
  - Filter target language 3-18
  - FIR adder style 3-57
  - Force clock 3-65
  - Force clock enable 3-65
  - Force reset 3-67
  - Hold time 3-67 3-69
  - Initialize real signals to 0.0 3-51
  - Inline VHDL configurations 3-48
  - Input data type 3-40
  - Input port 3-38

- Loop unrolling 3-46
- Minimum overlap of scale values 3-43
- Optimize for HDL 3-55
- Output data type 3-40
- Output port 3-38
- Package postfix 3-22
- Represent constant values by
  - aggregates 3-45
- Reserved word postfix 3-34
- Reset asserted level 3-28
- Reset port 3-38
- Reset type 3-26
- Reset value 3-67
- Split arch. file postfix 3-23
- Split entity and architecture 3-23
- Split entity file postfix 3-23
- Target directory
  - for test bench output 3-61
  - redirecting output with 3-21
- Use 'rising\_edge' for registers 3-47
- Use Verilog `timescale directives 3-50
- User defined response 3-72
- Verilog file extension
  - setting file extension with 3-20
- Verilog file extension option
  - renaming test bench file with 3-61
- VHDL file extension
  - renaming test bench file with 3-61
  - setting file extension with 3-20
- output
  - generated HDL 1-8
  - redirecting 3-21
- Output data type option 3-40
- Output port option 3-38
- output ports
  - naming 3-38
  - specifying data type for 6-35
- output registers
  - adding code for 6-3
  - suppressing generation of extra 3-42

- OutputGenerateLabel property 6-33
- OutputPort property 6-34
- OutputType property 6-35

## **P**

- package files
  - default name for 3-9
  - specifying postfix for 6-36
- Package postfix option 3-22
- PackagePostfix property 6-36
- packages
  - setting names of 3-20
  - setting postfix for 3-22
- parameters 1-7
- pipeline registers
  - using from command line 6-4
  - using from GUI 3-58
- pipelined FIR final summation 3-57
- port data types 3-40
- port properties 5-4
- ports
  - clock enable input 6-8
  - clock input 6-11
  - data input 6-27
  - data output 6-34
  - input 6-28
  - naming 3-38
  - output 6-35
  - reset input 6-39
- Ports tab 3-38
- process block labels 3-37
- properties
  - AddInputRegister 6-2
  - AddOutputRegister 6-3
  - AddPipelineRegisters 6-4
  - advanced coding 5-4
  - as input parameters 1-7
  - BlockGenerateLabel 6-6
  - CastBeforeSum 6-7



- ClockEnableInputPort 6-8
  - ClockEnableValue 6-9
  - ClockHighTime 6-10
  - ClockInputPort 6-11
  - ClockLowTime 6-12
  - ClockProcessPostfix 6-13
    - coding 5-4
  - CoeffMultipliers 6-14
  - CoeffName 6-15
  - EntityConflictPostfix 6-17
  - ErrorMargin 6-18
    - file location 5-2
    - file naming 5-2
  - FIRAdderStyle 6-19
  - ForceClock 6-20
  - ForceClockEnable 6-21
  - ForceReset 6-22
  - header comment 5-3
  - HoldTime 6-23
  - InitializeRealSignals 6-25
  - InlineConfigurations 6-26
  - InputPort 6-27
  - InputType 6-28
  - InstanceGenerateLabel 6-29
    - language selection 5-2
  - LoopUnrolling 6-30
  - Name 6-31
    - naming 5-3
  - optimization 5-6
  - OptimizeForHDL 6-32
  - OutputGenerateLabel 6-33
  - OutputPort 6-34
  - OutputType 6-35
  - PackagePostfix 6-36
    - port 5-4
  - ReservedWordPostfix 6-37
  - reset 5-2
  - ResetAssertedLevel 6-38
  - ResetInputPort 6-39
  - ResetType 6-40
    - ResetValue 6-42
    - SafeZeroConcat 6-43
    - ScaleWarnBits 6-44
    - SimulatorFlags 6-45
    - SplitArchFilePostfix 6-46
    - SplitEntityArch 6-47
    - SplitEntityFilePostfix 6-48
    - TargetDirectory 6-49
    - TargetLanguage 6-50
    - test bench 5-6
    - TestBenchName 6-51
    - TestBenchStimulus 6-52
    - TestBenchUserStimulus 6-53
    - UseAggregatesForConst 6-54
    - UserComment 6-55
    - UseRisingEdge 6-56
    - UseVerilogTimescale 6-58
    - VerilogFileExtension 6-59
    - VHDLFileExtension 6-60
- Q**
- quantization noise 3-43
  - quantized filters 1-6
- R**
- real signals
    - specifying initialization of 6-25
    - suppressing initialization of 3-51
  - registers
    - adding code for input 6-2
    - adding code for output 6-3
    - adding for optimization 6-4
    - pipeline 3-58
  - Represent constant values by aggregates
    - option 3-45
  - requirements
    - identifying for HDL code and test benches 3-13

- product 1-9
- Reserved word postfix option 3-34
- reserved words
  - setting postfix for resolving 3-34
  - specifying postfix for 6-37
- ReservedWordPostfix property 6-37
- Reset asserted level option 3-28
- reset input port 6-39
  - naming 3-38
- Reset port options 3-38
- reset properties 5-2
- Reset type option 3-26
- Reset value option 3-67
- ResetAssertedLevel property 6-38
- ResetInputPort property 6-39
- resets
  - configuring for test benches 3-67
  - customizing 3-26
  - defaults for 3-10
  - setting asserted level for from command line 6-38
  - setting asserted level for from GUI 3-28
  - setting style of 3-26
  - specifying forced 6-22
  - specifying test bench 6-42
  - types of 6-40
- ResetType property 6-40
- ResetValue property 6-42
- rising\_edge function 3-47 6-56

## **S**

- SafeZeroConcat property 6-43
- scale values 3-43 6-44
- ScaleWarnBits property 6-44
- second-order section (SOS) filters 1-6
- sections
  - instance 6-29
- simulator 4-7
- SimulatorFlags property 6-45

- SOS filters 1-6
- Split arch. file postfix option 3-23
- Split entity and architecture option 3-23
- Split entity file postfix option 3-23
- SplitArchFilePostfix property 6-46
- SplitEntityArch property 6-47
- SplitEntityFilePostfix property 6-48
- stimulus
  - setting for test benches 3-72
  - specifying 6-52
  - specifying user-defined 6-53
- subtraction operations
  - specifying input type treatment for 3-52
  - type casting 6-7
- summation technique 6-19
- symmetric FIR filters 1-6
- synchronous resets
  - setting from command line 6-40
  - setting from GUI 3-26
- synthesis 3-59

## **T**

- Target directory option
  - redirecting output with 3-21
  - renaming test bench file with 3-61
- target language 3-18
- TargetDirectory property 6-49
- TargetLanguage property 6-50
- test bench files 3-9
- test bench properties 5-6
- test benches
  - compiling 4-7
  - configuring clock for 3-65
  - configuring resets for 3-67
  - customizing 3-61
  - defaults for 3-12
  - error margin for 6-18
  - generated HDL output for 1-8
  - generating DO file 4-12

- HDL 4-3
  - naming 6-51
  - renaming 3-61
  - running 4-8
  - setting error margin for 3-70
  - setting input data hold time 3-69
  - setting names of 3-20
  - setting stimuli for 3-72
  - specifying clock enable input for 6-21
  - specifying forced clock input for 6-20
  - specifying forced resets for 6-22
  - specifying reset value for 6-42
  - specifying stimulus for 6-52
  - specifying type of 3-62
  - specifying user-defined stimulus for 6-53
- test methods 4-2
- TestBenchName property 6-51
- TestBenchStimulus property 6-52
- TestBenchUserStimulus property 6-53
- time
  - clock high 6-10
  - clock low 6-12
  - hold 6-23
- timescale directives
  - specifying use of 6-58
  - suppressing 3-50
- transposed Direct Form I filters 1-6
- transposed Direct Form II filters 1-6
- transposed FIR filters 1-6
- tree FIR final summation 3-57
- tutorial files 2-2
- tutorials 1-12
  - basic FIR filter 2-3
  - IIR filter 2-43
  - optimized FIR filter 2-23
- type casting 6-7
  - for addition and subtraction operations 3-52

**U**

- Use 'rising\_edge' for registers option 3-47
- Use Verilog `timescale directives option 3-50
- UseAggregatesForConst property 6-54
- User defined response option 3-72
- UserComment property 6-55
- UseRisingEdge property 6-56
- UseVerilogTimescale property 6-58

**V**

- Verilog 1-2
  - file extension 6-59
  - selecting 3-18
- Verilog file extension option
  - naming filter file with 3-20
  - renaming test bench file with 3-61
- Verilog reserved words 3-34
- Verilog test benches 3-62
- VerilogFileExtension property 6-59
- VHDL 1-2
  - file extension 6-60
  - selecting 3-18
- VHDL file extension option
  - naming filter file with 3-20
  - renaming test bench file with 3-61
- VHDL reserved words 3-34
- VHDL test benches 3-62
- VHDLFileExtension property 6-60

**Z**

- zeros
  - concatenated 3-49
- zeros, concatenated 6-43