Filter Design HDL Coder

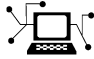
For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide

Version 1

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	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.
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 2004–2006 by The MathWorks, Inc.

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

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

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

ModelSim is a registered trademark of Mentor Graphics Corporation.

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

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

June 2004	Online only	New for Version 1.0 (Release 14)
October 2004	Online only	Updated for Version 1.1 (Release 14SP1)
March 2005	Online only	Updated for Version 1.2 (Release 14SP2)
September 2005	Online only	Updated for Version 1.3 (Release 14SP3)
March 2006	Online only	Updated for Version 1.4 (Release 2006a)
September 2006	Online only	Updated for Version 1.5 (Release 2006b)

Contents

1 Getting Started

What Is Filter Design HDL Coder? **1-2**
Expected Users **1-3**
Key Features and Components **1-3**
FDATool Plug-In — the GUI **1-4**
Command-Line Interface **1-6**
Quantized Filters — the Input **1-6**
Filter Properties — Input Parameters **1-8**
Generated HDL Files — the Output **1-9**

Installation **1-10**
Checking Product Requirements **1-10**
Installing the Software **1-10**

Getting Help with Filter Design HDL Coder **1-11**
Information Overview **1-11**
Online Help **1-12**
Using "What's This?" Context-Sensitive Help **1-12**
Demos and Tutorials **1-13**

Applying Filter Design HDL Coder to the Hardware Design Process **1-14**

2 Tutorials: Generating HDL Code for Filters

Creating a Directory for Your Tutorial Files **2-2**

Basic FIR Filter Tutorial **2-3**
Designing a Basic FIR Filter **2-3**
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 . . 2-17

Optimized FIR Filter Tutorial . . . **2-23**
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

IIR Filter Tutorial . . . **2-44**
Designing an IIR Filter . . . 2-44
Quantizing the IIR Filter . . . 2-46
Configuring and Generating the IIR Filter's VHDL Code . . 2-50
Getting Familiar with the IIR Filter's Generated VHDL Code . . . 2-57
Verifying the IIR Filter's Generated VHDL Code . . . 2-58

3 Generating HDL Code for a Filter Design

Overview of Generating HDL Code for a Filter Design . . . **3-3**

Opening the Generate HDL Dialog Box . . . **3-5**

What Is Generated by Default? . . . **3-10**
Default Settings for Generated Files . . . 3-10
Default Generation of Script Files . . . 3-11
Default Settings for Register Resets . . . 3-11
Default Settings for General HDL Code . . . 3-11
Default Settings for Code Optimizations . . . 3-13
Default Settings for Test Benches . . . 3-13

What Are Your HDL Requirements? **3-15**

Setting the Target Language **3-21**

Setting the Names and Location for Generated HDL Files **3-22**
Setting Filter Entity and General File Naming Strings ... **3-23**
Redirecting Filter Design HDL Coder Output **3-24**
Setting the Postfix String for VHDL Package Files **3-25**
Splitting Entity and Architecture Code into Separate Files **3-26**

Customizing Reset Specifications **3-29**
Setting the Reset Style for Registers **3-29**
Setting the Asserted Level for the Reset Input Signal **3-30**

Customizing the HDL Code **3-32**
Specifying a Header Comment **3-33**
Specifying a Prefix for Filter Coefficients **3-35**
Setting the Postfix String for Resolving Entity or Module Name Conflicts **3-36**
Setting the Postfix String for Resolving HDL Reserved Word Conflicts **3-37**
Setting the Postfix String for Process Block Labels **3-40**
Naming HDL Ports **3-42**
Specifying the HDL Data Type for Data Ports **3-43**
Suppressing Extra Input and Output Registers **3-45**
Minimizing Quantization Noise for Fixed-Point Filters ... **3-46**
Representing Constants with Aggregates **3-48**
Unrolling and Removing VHDL Loops **3-49**
Using the VHDL rising_edge Function **3-50**
Suppressing the Generation of VHDL Inline Configurations **3-52**
Specifying VHDL Syntax for Concatenated Zeros **3-53**
Suppressing Verilog Time Scale Directives **3-54**
Specifying Input Type Treatment for Addition and Subtraction Operations **3-55**

Setting Optimizations **3-57**
Optimizing Generated Code for HDL **3-58**
Optimizing Coefficient Multipliers **3-59**
Optimizing Final Summation for FIR Filters **3-60**

Speed vs. Area Optimizations for FIR Filters 3-61
Distributed Arithmetic for FIR Filters 3-71
Optimizing the Clock Rate with Pipeline Registers 3-81
Setting Optimizations for Synthesis 3-83

Generating Code for Multirate Filters **3-85**
Supported Multirate Filter Types 3-85
Generating Mutirate Filter Code 3-85
Code Generation Options for Multirate Filters 3-85

Generating Code for Cascade Filters **3-92**
Supported Cascade Filter Types 3-92
Generating Cascade Filter Code 3-92

Customizing the Test Bench **3-95**
Renaming the Test Bench 3-95
Specifying a Test Bench Type 3-97
Configuring the Clock 3-99
Configuring Resets 3-101
Setting a Hold Time for Data Input Signals 3-103
Setting an Error Margin for Optimized Filter Code 3-104
Setting Test Bench Stimuli 3-106

Generating the HDL Code **3-109**

Generating Scripts for EDA Tools **3-110**
Enabling and Disabling Script Generation 3-110
Default Script Generation 3-110
Customizing Script Names 3-111
Customizing Script Code 3-111
Mixed-Language Scripts 3-114

4 Testing a Filter Design

Overview of the Test Methods **4-2**

Testing with an HDL Test Bench **4-3**

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**

Testing with a ModelSim Tcl/Tk .do File **4-12**
Generating the Filter HDL Code and Test Bench .do File **4-12**
Starting ModelSim **4-16**
Compiling the Generated Filter File **4-16**
Execute the ModelSim .do File **4-17**

5 Properties — By Category

Language Selection Properties **5-2**

File Naming and Location Properties **5-2**

Reset Properties **5-2**

Header Comment and General Naming Properties **5-3**

Port Properties **5-3**

Advanced Coding Properties **5-4**

Optimization Properties **5-6**

Test Bench Properties **5-6**

Script Generation Properties **5-7**

6 Properties — Alphabetical List

7 Functions — Alphabetical List

A Examples

Tutorials **A-2**

Basic FIR Filter Tutorial **A-2**

Optimized FIR Filter Tutorial **A-2**

IIR Filter Tutorial **A-2**

Speed vs. Area Optimizations for FIR Filters **A-3**

Index

1

Getting Started

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

What Is Filter Design HDL Coder? (p. 1-2)	Describes key product features and components
Installation (p. 1-10)	Required products; required VHDL and Verilog versions; how to install and set up Filter Design HDL Coder
Getting Help with Filter Design HDL Coder (p. 1-11)	Discusses ways of applying Filter Design HDL Coder to the hardware design process, including signal analysis, algorithm verification, and reference design validation
Applying Filter Design HDL Coder to the Hardware Design Process (p. 1-14)	Identifies and explains how to gain access to available documentation and online help resources

What Is Filter Design HDL Coder?

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 description language (VHDL) and Verilog, to develop hardware designs. Although HDLs provide a proven method for hardware design, the task of coding filter designs, and hardware designs in general, is labor intensive and the use of these languages for algorithm and system-level design is not optimal.

Using 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 Filter Design HDL Coder to design a filter. Then, with the click of a button, 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 1-6
- “Quantized Filters — the Input” on page 1-6

- “Filter Properties — Input Parameters” on page 1-8
- “Generated HDL Files — the Output” on page 1-9

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 discrete-time filter structures:
  - Finite impulse response (FIR)
  - Antisymmetric FIR
  - Transposed FIR
  - Symmetric FIR

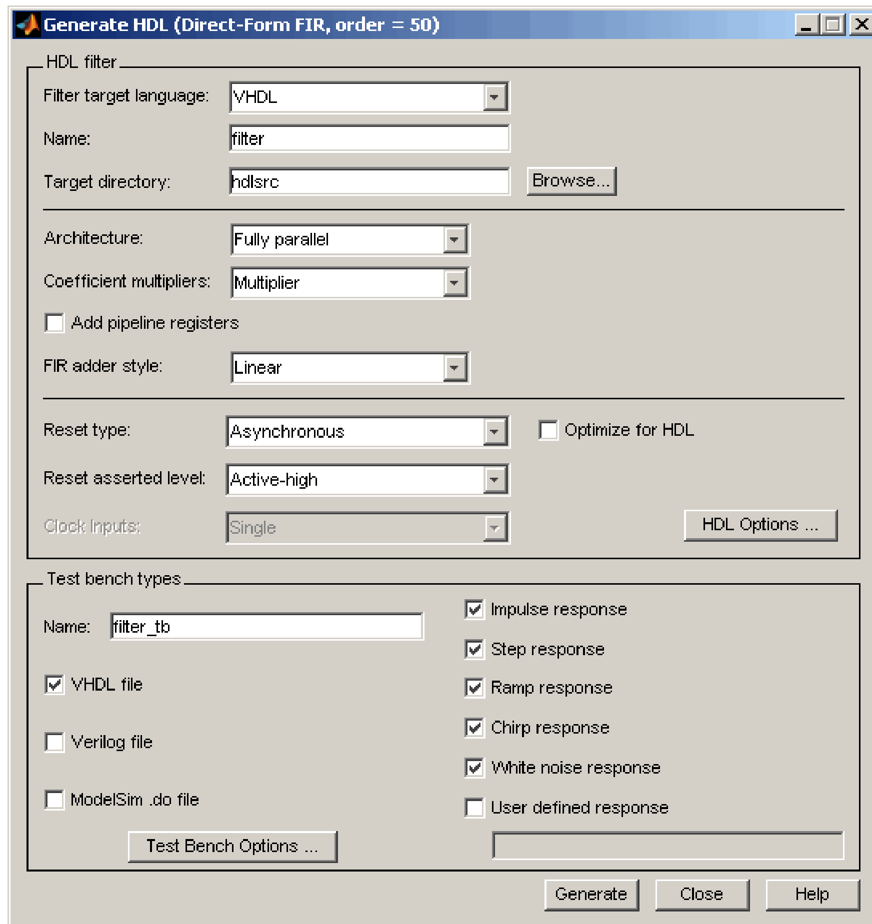
- Second-order section (SOS) infinite impulse response (IIR) Direct Form I
- SOS IIR Direct Form I transposed
- SOS IIR Direct Form II
- SOS IIR Direct Form II transposed
- Discrete-Time Scalar
- Delay filter

- Support for the following multirate filter structures:
  - Cascaded Integrator Comb (CIC) interpolation
  - Cascaded Integrator Comb (CIC) decimation
  - Direct-Form Transposed FIR Polyphase Decimator
  - Direct-Form FIR Polyphase Interpolator
  - Direct-Form FIR Polyphase Decimator
  - FIR Hold Interpolator
  - FIR Linear Interpolator
- Support for cascade filters (multirate and discrete-time)
- 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
- Automatic generation of scripts for third-party simulation and synthesis tools

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 box. To open this dialog box, click**Targets > Generate HDL** . The main dialog box displays the filter's structure and order in the title bar. The following figure 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".

Function	Purpose
`generatehdl`	Generate HDL code for quantized filter
`generatetb`	Generate test bench for quantized filter
`generatetbstimulus`	Generate and return test bench stimuli

Quantized Filters — the Input

The input to 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

Filter Design HDL Coder supports the following filter structures.

- Discrete-time:
  - 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
  - Discrete-Time Scalar

- Delay filter

- Multirate:
  - Cascaded Integrator Comb (CIC) interpolation
  - Cascaded Integrator Comb (CIC) decimation
  - Direct-Form Transposed FIR Polyphase Decimator
  - Direct-Form FIR Polyphase Interpolator
  - Direct-Form FIR Polyphase Decimator
  - FIR Hold Interpolator
  - FIR Linear Interpolator
- Cascade filters (multirate and discrete-time)

Each of these structures, (with the exception of the CIC filter structures), supports fixed-point, quantization type, and floating-point (double) realizations.

The CIC filter types support only fixed-point 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

When designing a filter for code generation with Filter Design HDL Coder, consider how filter coefficients are specified. If the coefficients for a filter are small in value and the word size and binary point are large, it is possible for Filter Design HDL Coder to compute integer coefficients that are numerically inaccurate. Double-precision coefficients support up to 53 bits of precision.

For information on how to design filter objects and specify filter coefficients, 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

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 — By Category” and Chapter 7, “Functions — Alphabetical List”, respectively.

Generated HDL Files — the Output

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

Language	File	Name
Verilog	Filter	*name*.v
	Filter test bench	*name*_tb.v
VHDL	Filter	*name*.vhd
	Filter test bench	*name*_tb.vhd
	Filter package (if required)	*name*_pkg.vhd

Installation

The following sections discuss installation:

- “Checking Product Requirements” on page 1-10
- “Installing the Software” on page 1-10

Checking Product Requirements

Filter Design HDL Coder requires the following MathWorks products:

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

VHDL and Verilog Language Support

Filter Design HDL Coder is compatible with HDL compilers, simulators and other tools that support

- VHDL versions 87, 93, and 02.

  Exception: VHDL test benches using double precision data types do not support VHDL version 87. (See also “Compiling the Generated Filter and Test Bench Files” on page 4-7)
- Verilog-2001 (IEEE 1364-2001) or later.

Installing the Software

For information on installing the required software listed above, and optional software, see the MATLAB installation documentation for your platform.

Getting Help with Filter Design HDL Coder

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

- "Information Overview" on page 1-11
- "Online Help" on page 1-12
- "Using "What's This?" Context-Sensitive Help" on page 1-12
- "Demos and Tutorials" on page 1-13

Information Overview

The following information is available with this product:

Chapter 1, "Getting Started"	Explains what the product is, how to install it, its applications in the hardware design process, and how to access 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 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 — By Category"	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 box 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 1-12.
- 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
  ```

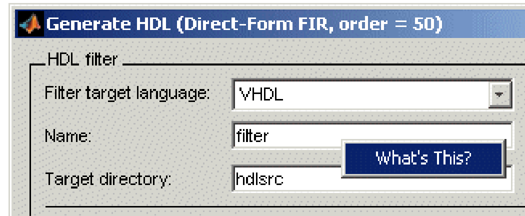
Using "What's This?" Context-Sensitive Help

"What's This?" context-sensitive help topic is available for each dialog box, 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 box.

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 box.



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

Demos and Tutorials

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:

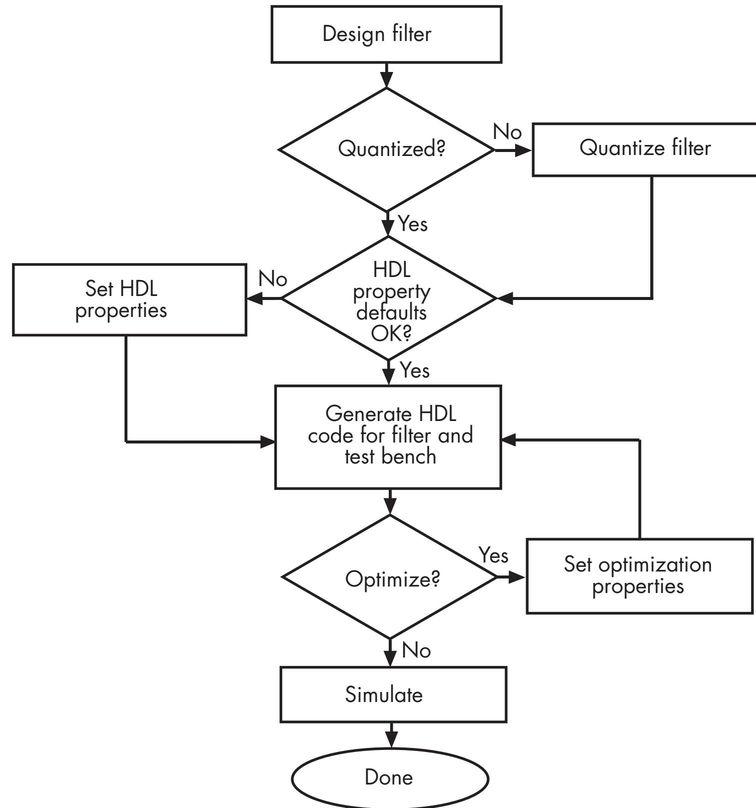
- "Basic FIR Filter Tutorial" on page 2-3
- "Optimized FIR Filter Tutorial" on page 2-23
- "IIR Filter Tutorial" on page 2-44

Applying Filter Design HDL Coder to the Hardware Design Process

The basic workflow for applying Filter Design HDL Coder to the hardware design process requires 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 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. 2-44)	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:

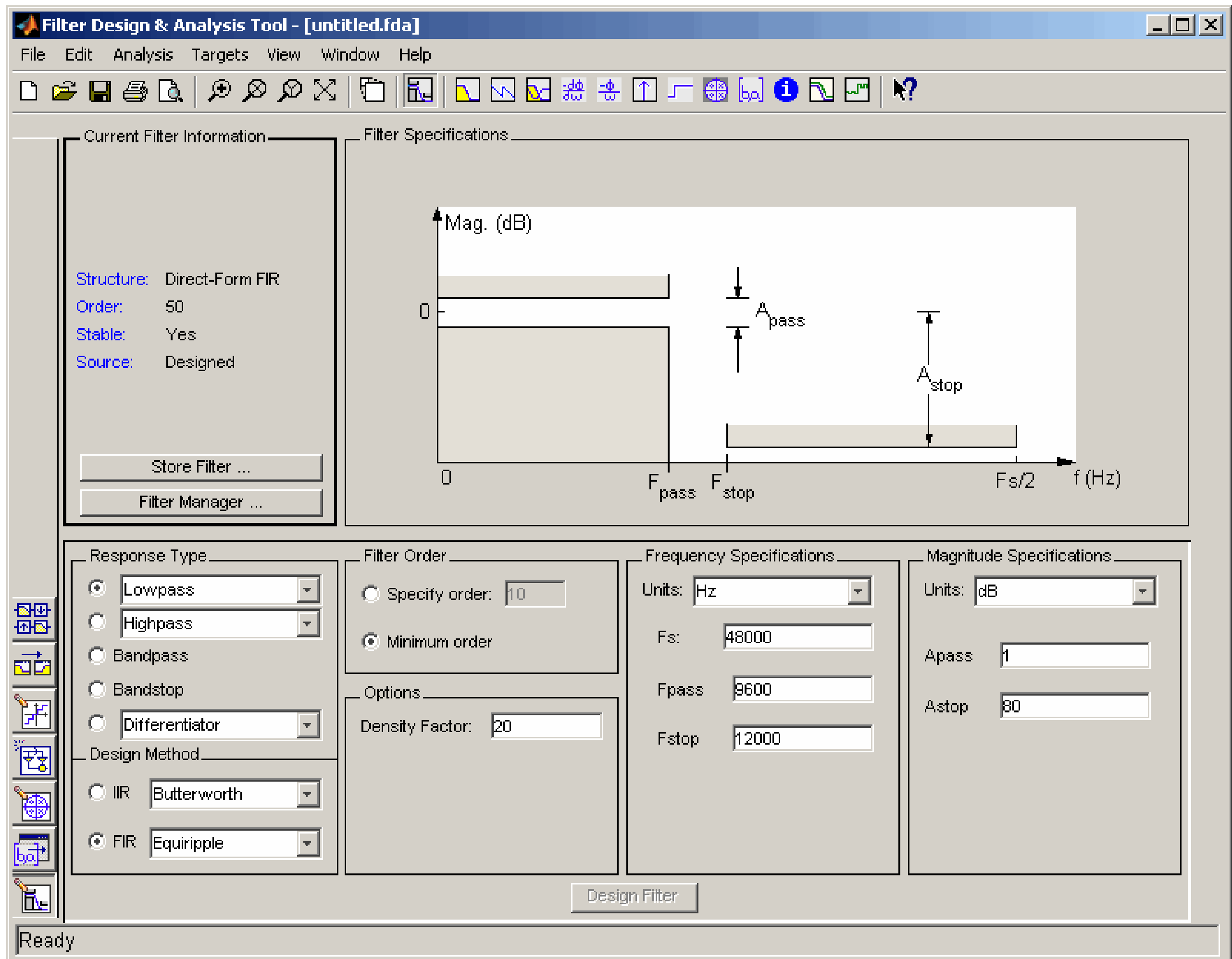
- "Designing a Basic FIR Filter" on page 2-3
- "Quantizing the Basic FIR Filter" on page 2-5
- "Configuring and Generating the Basic FIR Filter's VHDL Code" on page 2-8
- "Getting Familiar with the Basic FIR Filter's Generated VHDL Code" on page 2-15
- "Verifying the Basic FIR Filter's Generated VHDL Code" on page 2-17

Designing a Basic FIR Filter

One way of designing a filter in the MATLAB environment is to use the FDATool. This section assumes you are familiar with the MATLAB user interface and the FDATool. The following instructions guide you through the procedure of designing and creating a basic FIR filter:

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 box.



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

Option	**Value**
Response Type	Lowpass
Design Method	FIR Equiripple
Filter Order	**Minimum order**
Options	**Density Factor**: 20
Frequency Specifications	**Units**: Hz
	Fs: 48000
	Fpass: 9600
	Fstop: 12000
Magnitude Specifications	**Units**: dB
	Apass: 1
	Astop: 80

These settings are for the default filter design that the FDATool creates for you. If you do not need to make any changes and **Design Filter** is grayed 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 “FDATool: A Filter Design and Analysis GUI” in the Filter Design Toolbox documentation.

Quantizing the Basic FIR Filter

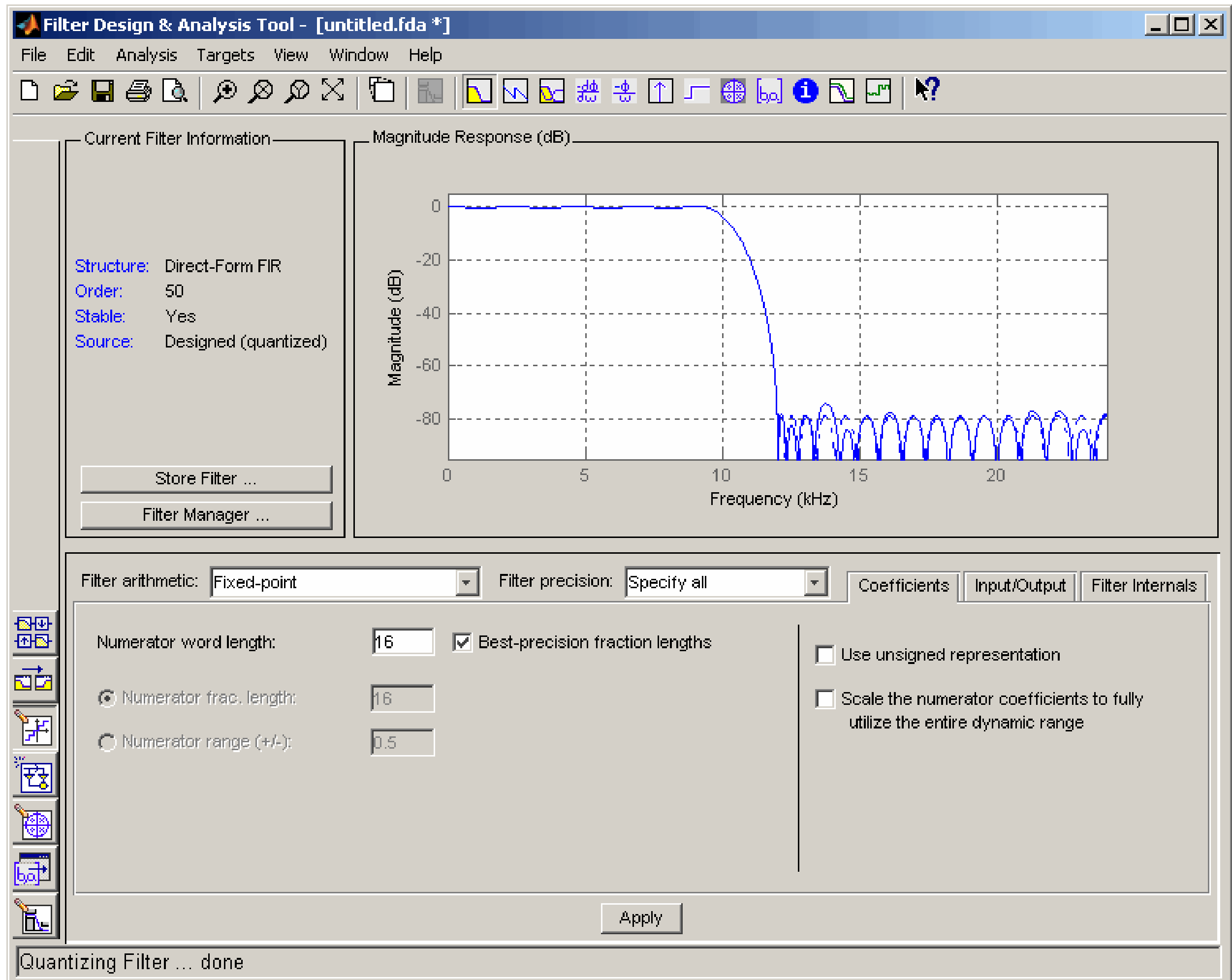
You should quantize filters for HDL code generation. To quantize your filter,

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 button in the left-side tool bar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog box.

3 Select `Fixed-point` from the **Filter arithmetic** list. Then select `Specify all` from the **Filter precision** list. The FDATool displays the first of

three tabbed panels of quantization parameters across the bottom half of its dialog box.



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

4 Set the quantization parameters as follows:

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

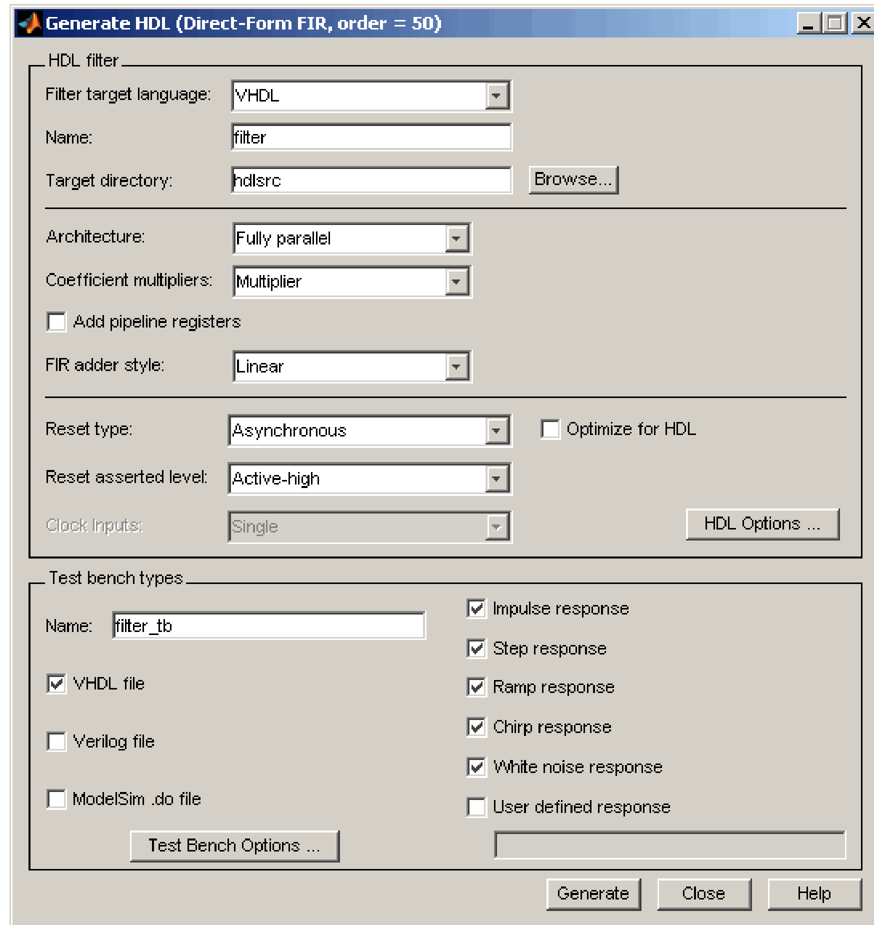
5 Click **Apply**.

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

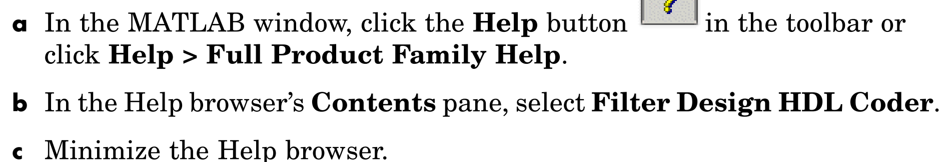
Configuring and Generating the Basic FIR Filter's VHDL Code

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 selecting **Targets > Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog.



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

a In the MATLAB window, click the **Help** button  in the toolbar or click **Help > Full Product Family Help**.

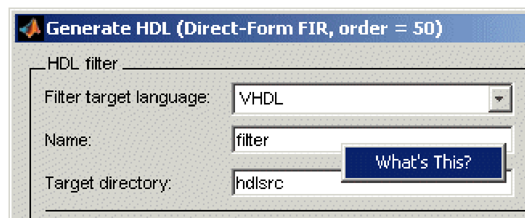
b In the Help browser's **Contents** pane, select **Filter Design HDL Coder**.

c Minimize the Help browser.

3 Click the Help button. The FDATool displays context-sensitive help for the dialog box. 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 box and right-click. A What's This? button appears.

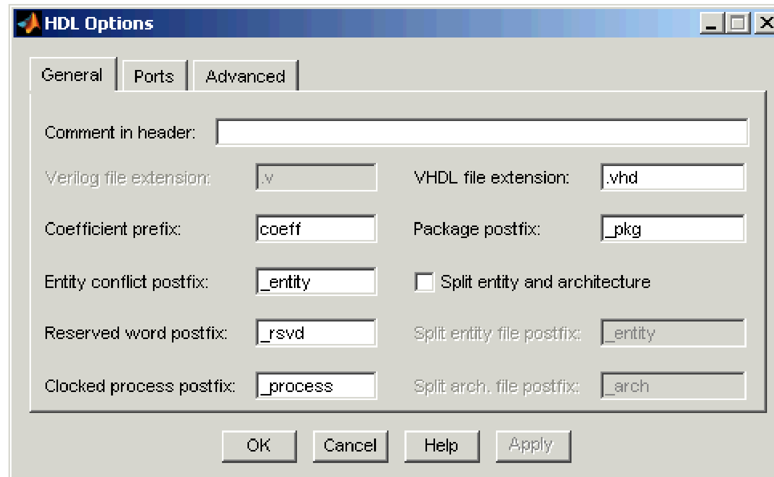


6 Click **What's This?** The Filter Design HDL Coder opens context-sensitive help that describes the **Name** option. Use the context-sensitive help as needed while using the GUI to configure options that control the contents and style of the generated HDL code and test bench. A help topic is available 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.

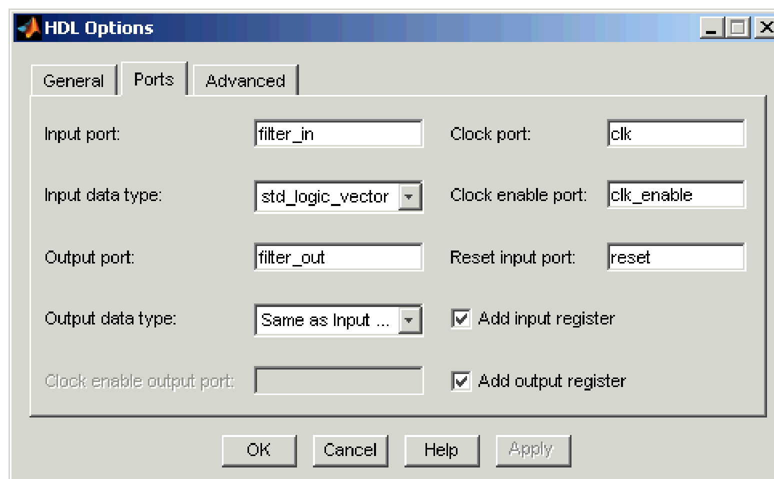
8 In the **Name** text box of the **Test bench types** pane, replace the default name with `basicfir_tb`. This option names the generated test bench file.

9 Click **HDL Options**. The Filter Design HDL Coder displays the HDL Options dialog box.



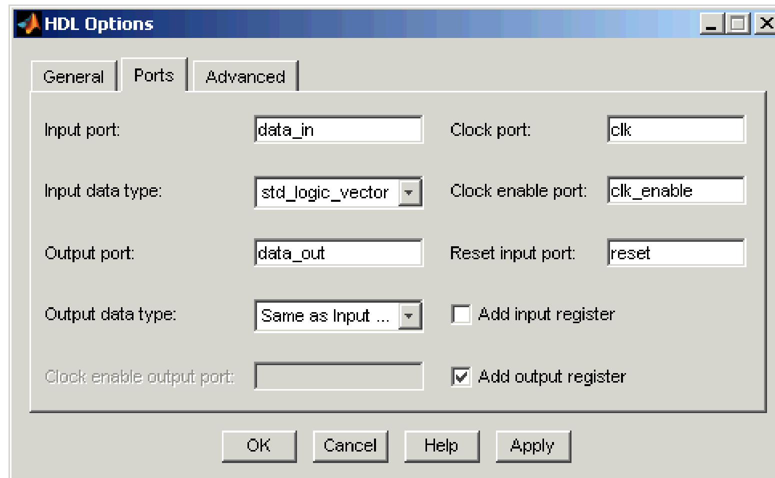
10 In the **Comment in header** text box, type `Tutorial - Basic FIR Filter` and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.

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



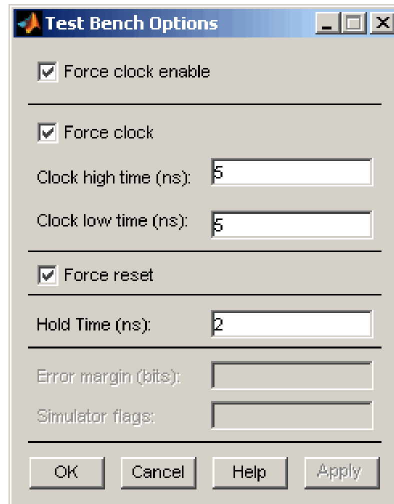
12 Change the names of the input and output ports. Replace `filter_in` with `data_in` and `filter_out` with `data_out`.

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



14 Click **Apply** and then **OK** to register your changes and close the HDL Options dialog box.

15 Click **Test Bench Options**. The Filter Design HDL Coder displays the Test Bench Options dialog box.



You use this dialog box to customize the generated test bench.

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

17 In the Generate HDL dialog box, click **Generate** to start the code generation process.

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: hdlsrc
### Starting generation of basicfir VHDL entity
### Starting generation of basicfir VHDL architecture
### HDL latency is 1 samples
### Successful completion of VHDL code generation process for filter: basicfir

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
```

```
### Generating VHDL file basicfir_tb.vhd in: hdlsrc
### Please wait ........
### 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:

VHDL Port	Name
Input	`data_in`
Output	`data_out`
Clock input	`clk`
Clock enable input	`clk_enable`
Reset input	`reset`

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

The generated test bench:

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

18. When you have finished generating code, click **Close** to close the Generate HDL dialog box.

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

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

1. Open the generated VHDL filter file `basicfir.vhd`.
2. Search for `basicfir`. This line identifies the VHDL module, using the string you specified for the **Name** option in the **HDL filter** pane. See step 5 in "Configuring and Generating the Basic FIR Filter's VHDL Code" on page 2-8.
3. Search for `Tutorial`. 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 box. 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,`coeff`*n*, 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`.

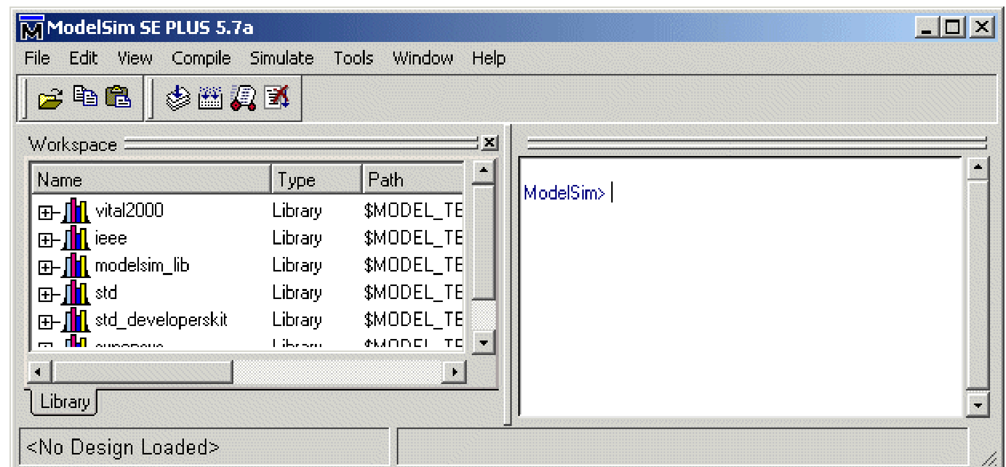
14 Search for `data_out`. This is where the filter writes its output data.

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
```

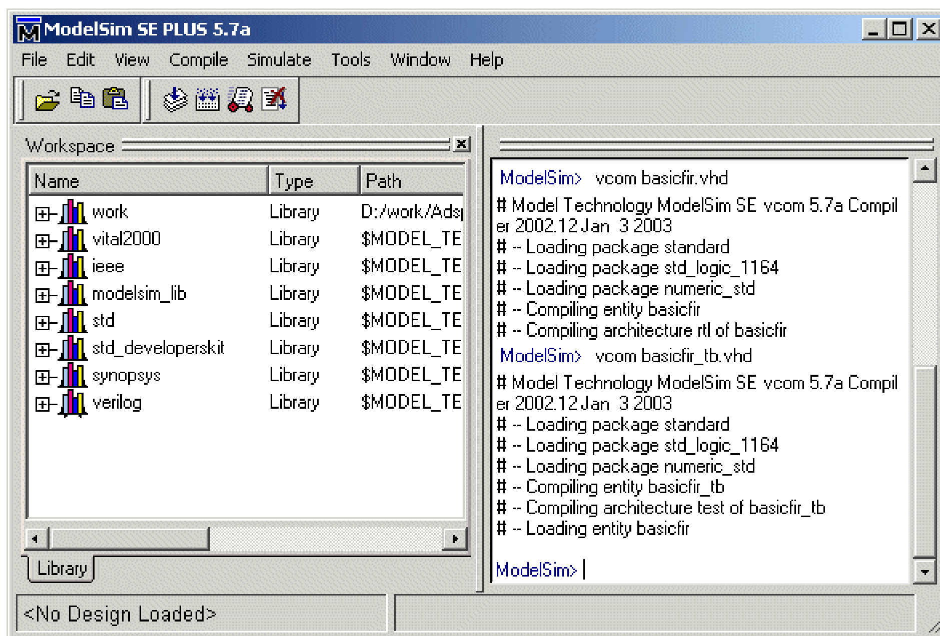
3 If necessary, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In ModelSim, you can create a design library with the `vlib` command.

```
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
```

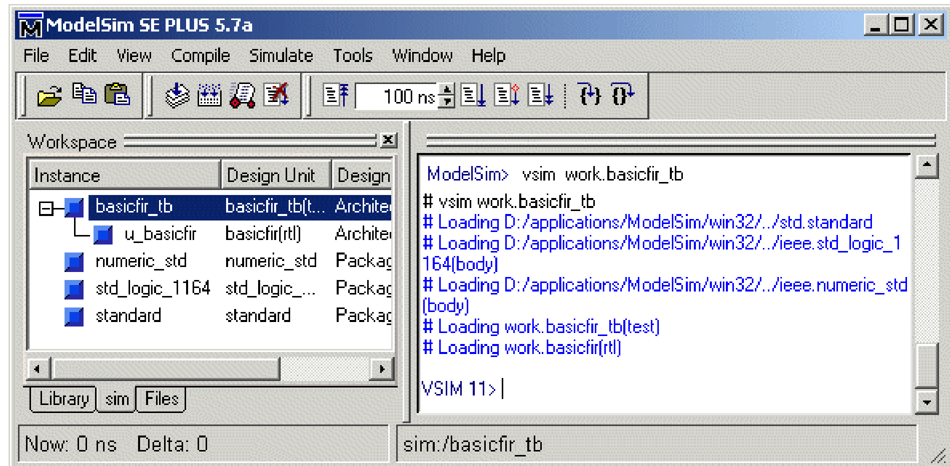
The following screen display shows this command sequence and informational messages displayed during compilation.



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.



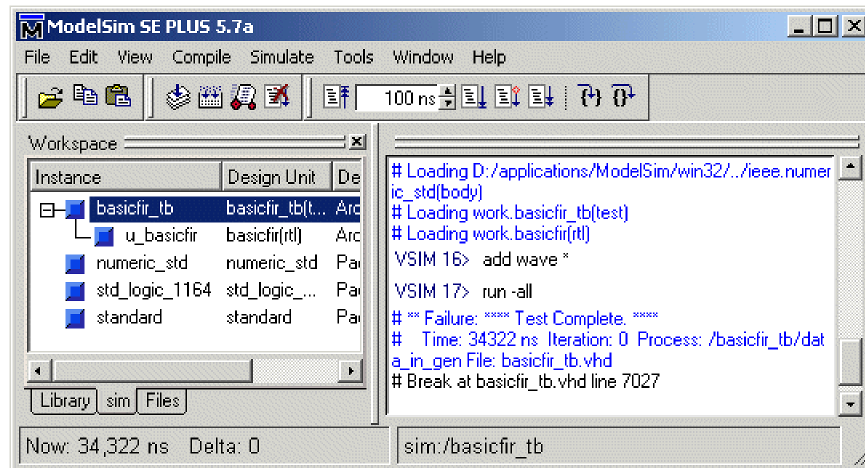
6 Open a display window for monitoring the simulation as the test bench runs. For example, in ModelSim, you can use the following command to open a **wave** window to view the results of the simulation as HDL waveforms:

```
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.

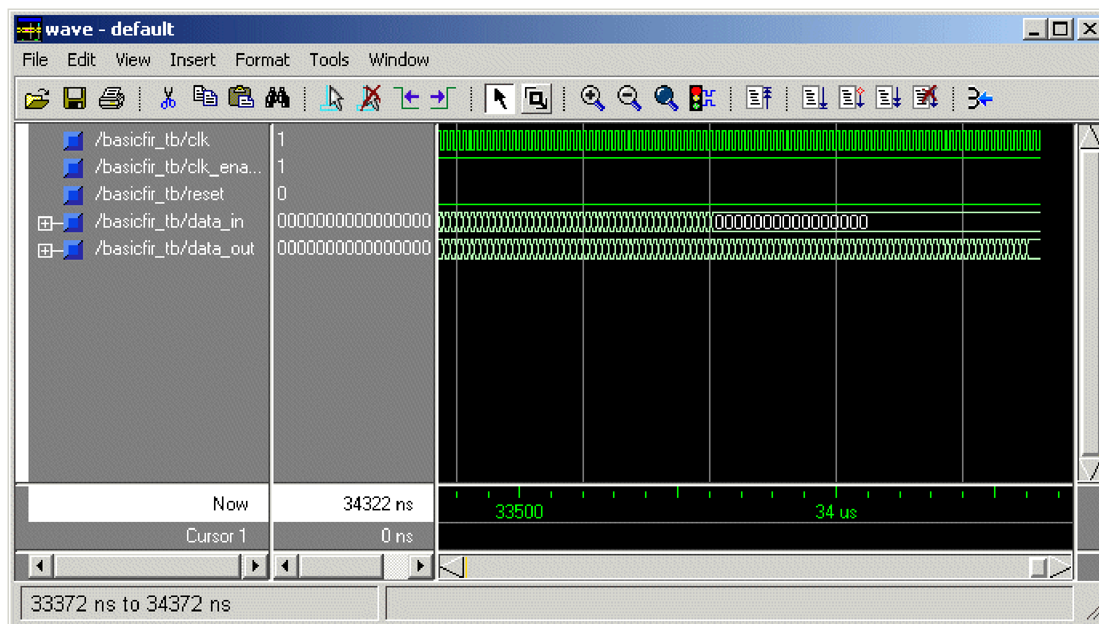
The following ModelSim display shows the `run -all` command being used to start a simulation.



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

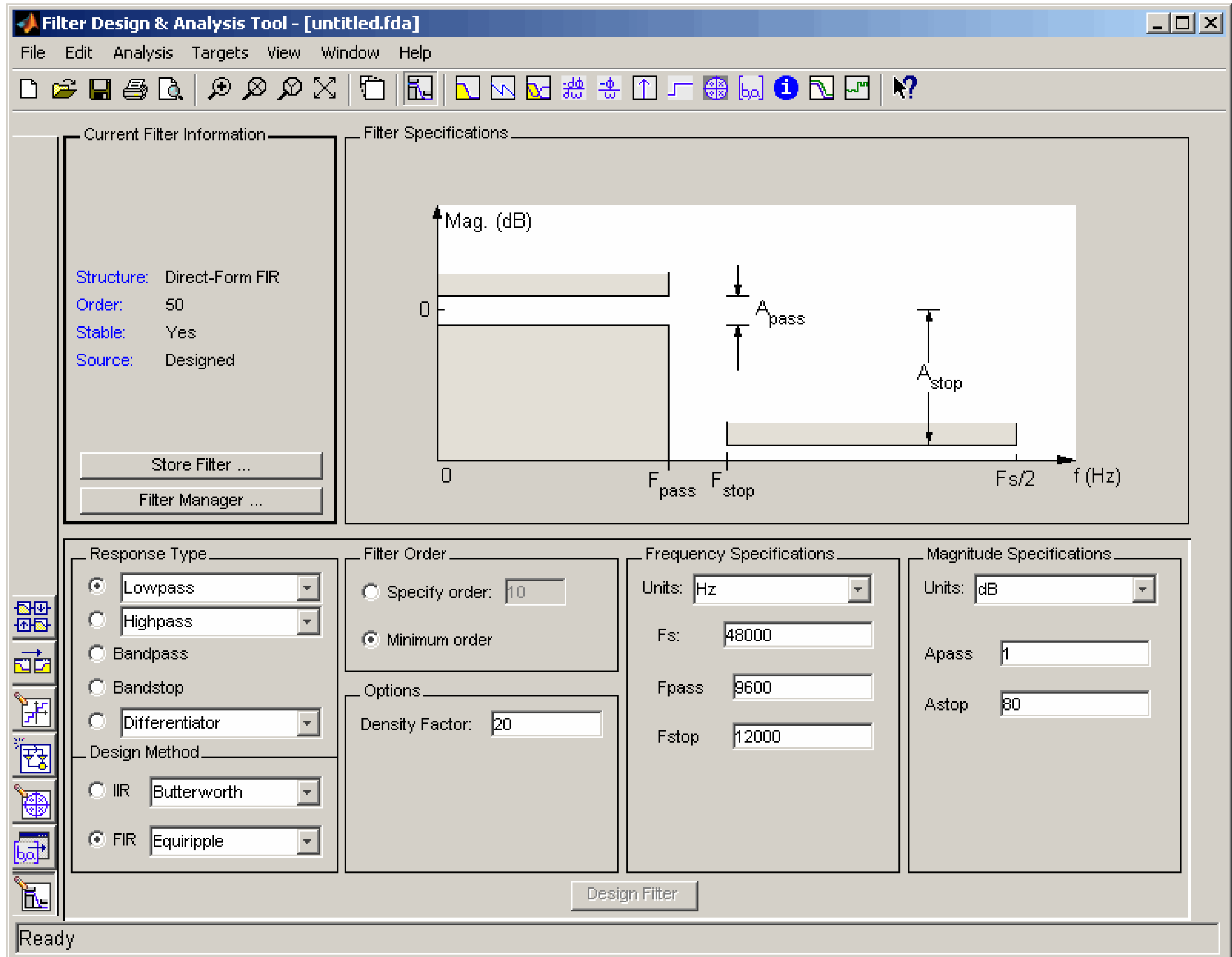
Designing the FIR Filter

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 box.



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

Option	Value
Response Type	`Lowpass`
Design Method	`FIR Equiripple`
Filter Order	**Minimum order**
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 grayed 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.

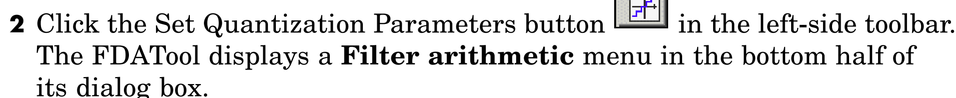
```
Designing Filter... Done
```

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

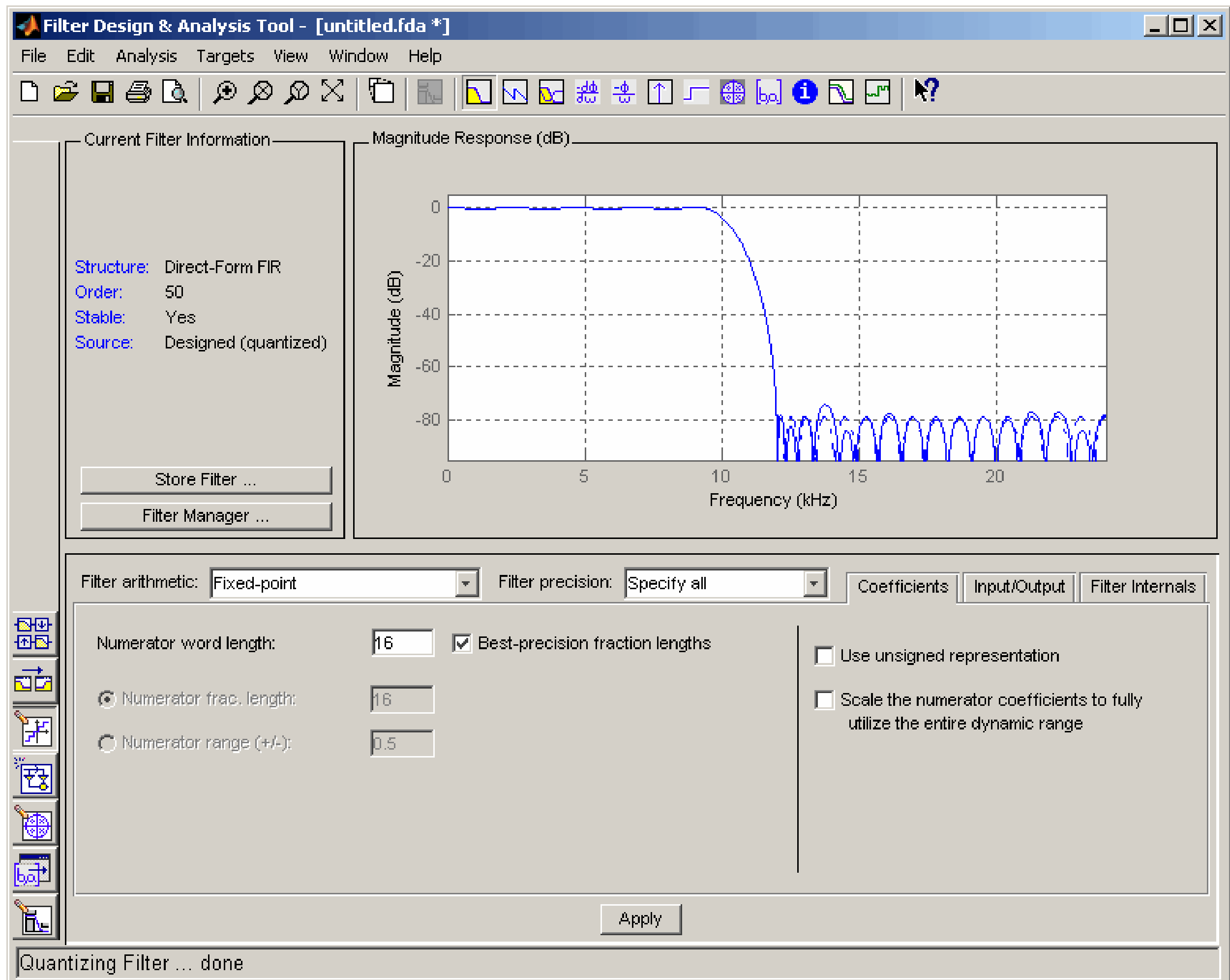
Quantizing the FIR Filter

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

1 Open the FIR filter design you created in "Optimized FIR Filter Tutorial" on page 2-23 if it is not already open.

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

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



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

4 Set the quantization parameters as follows:

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

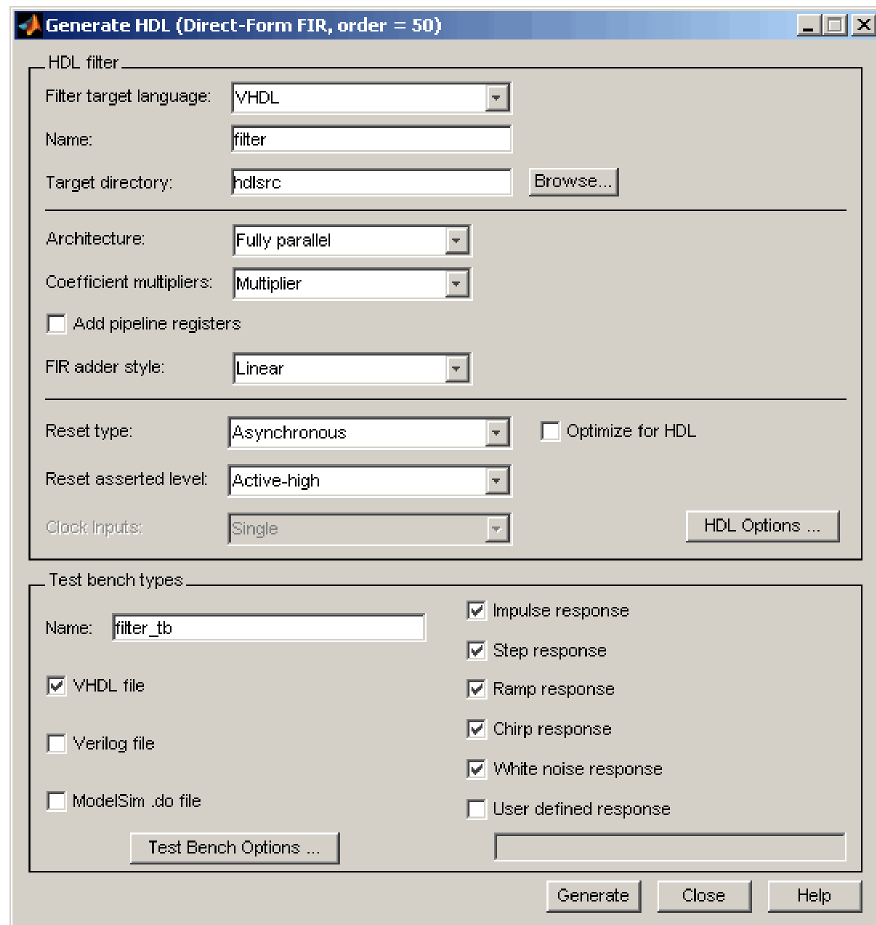
5 Click **Apply**.

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

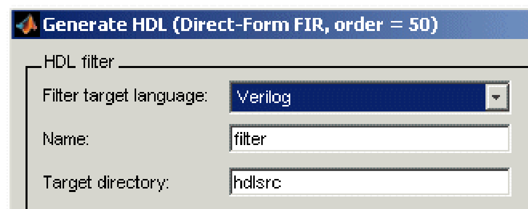
Configuring and Generating the FIR Filter's Optimized Verilog Code

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 selecting **Targets–>Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog box.



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



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.

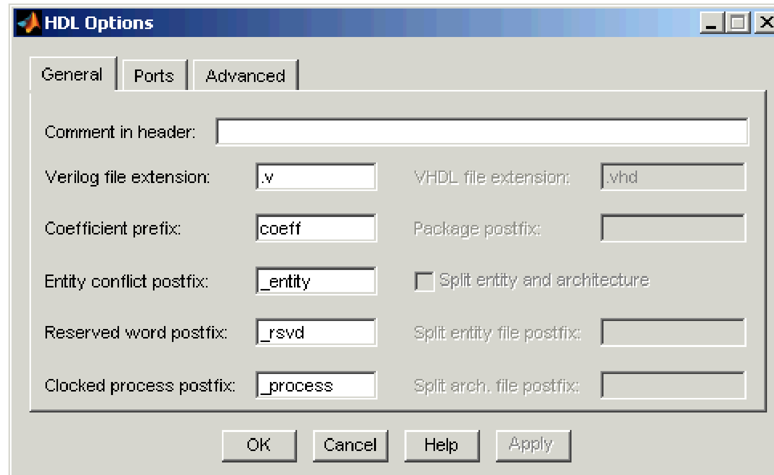
4. In the **Name** text box of the **Test bench types** pane, replace the default name with `optfir_tb`. This option names the generated test bench file.

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.

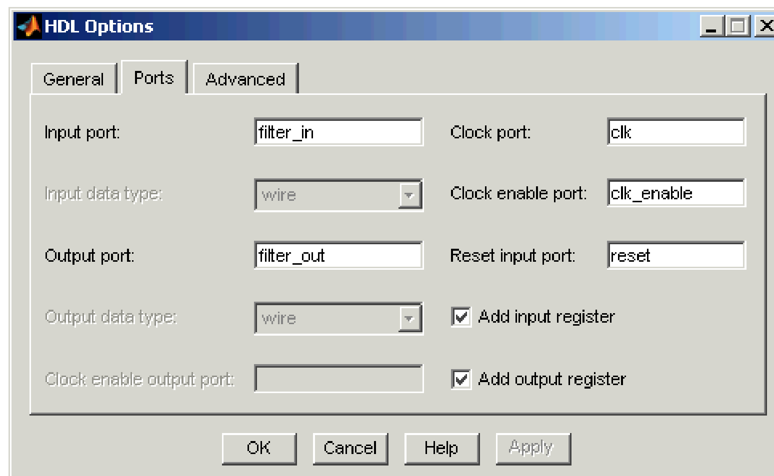
7. Select the **Add pipeline registers** option. For FIR filters, this option optimizes final summation. The coder creates a final adder that performs pair-wise addition on successive products and includes a stage of pipeline registers after each level of the tree. When used for FIR filters, this option also has the potential for producing numeric differences between MATLAB filter results and the simulated results for the optimized HDL code.

8. Click **HDL Options**. The Filter Design HDL Coder displays the HDL Options dialog box.



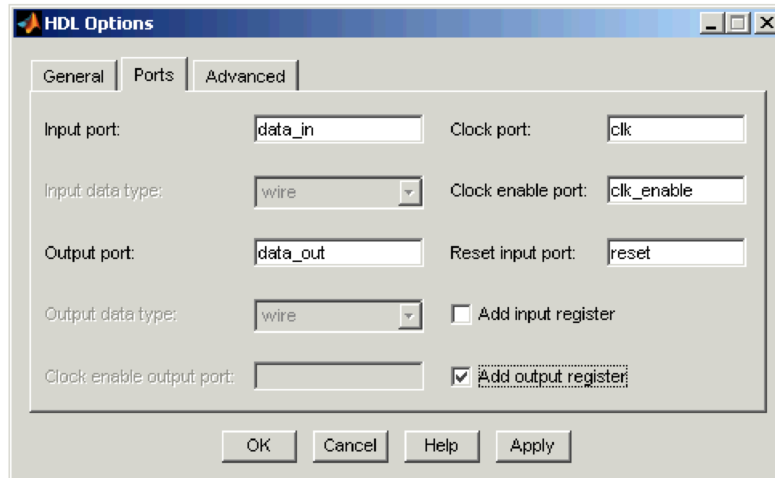
9 In the **Comment in header** text box, type `Tutorial - Optimized FIR Filter` and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.

10 Select the **Ports** tab. The **Ports** pane appears.



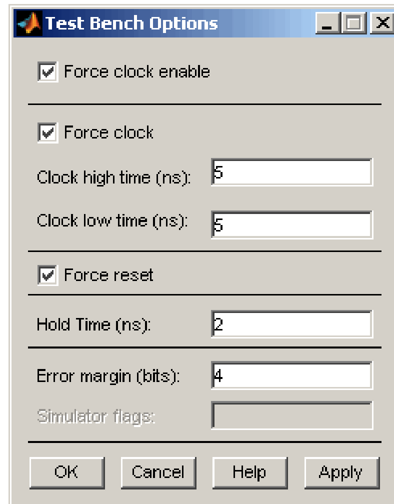
11 Change the names of the input and output ports. Replace `filter_in` with `data_in` and `filter_out` with `data_out`.

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



13 Click **Apply** and then **OK** to register your changes and close the HDL Options dialog box.

14 Click **Test Bench Options**. The Filter Design HDL Coder displays the Test Bench Options dialog box.



Use this dialog box to customize the generated test bench. Note that the **Error margin (bits)** option is enabled. This option is enabled because previously selected optimization options (such as **Add pipeline registers**) can potentially produce numeric results that differ from the results 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 box, click **Generate** to start the code generation process. When code generation completes, click **Close** to close the dialog box.

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:

Verilog Port	Name
Input	`data_in`
Output	`data_out`
Clock input	`clk`
Clock enable input	`clk_enable`
Reset input	`reset`

- 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 `Tutorial`. 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 box. The ports for data input and output are named with the strings you specified for the **Input port** and **Output port** options on the **Ports** tab of the HDL Options dialog box. 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, `coeff`*n*, 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`.

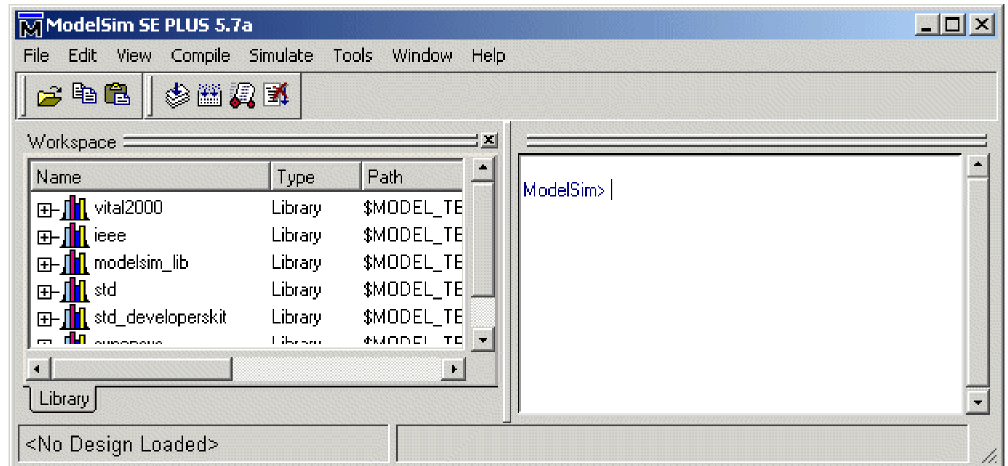
16 Search for `data_out`. This is where the filter writes its output data.

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
```

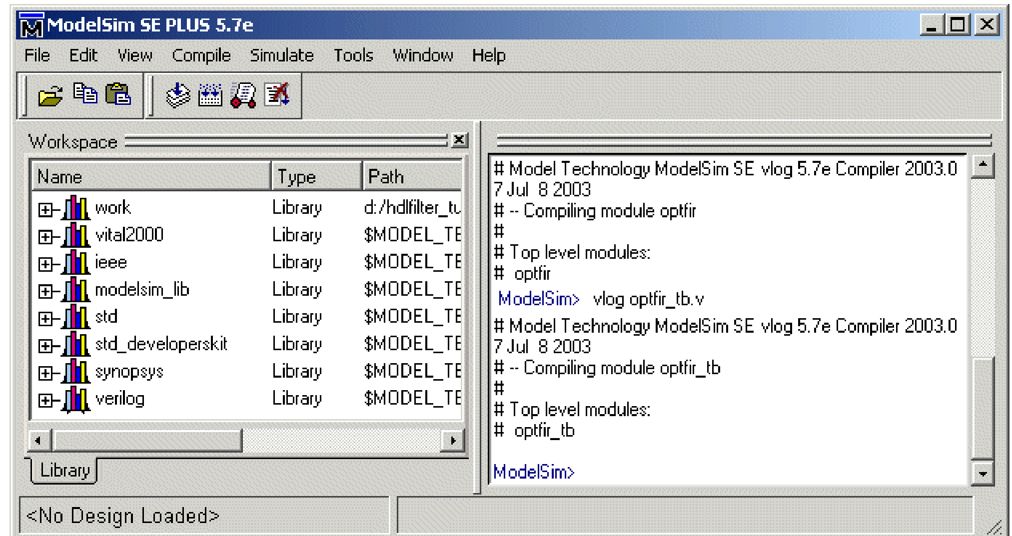
3 If necessary, create a design library to store the compiled Verilog modules. In ModelSim, you can create a design library with the `vlib` command.

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

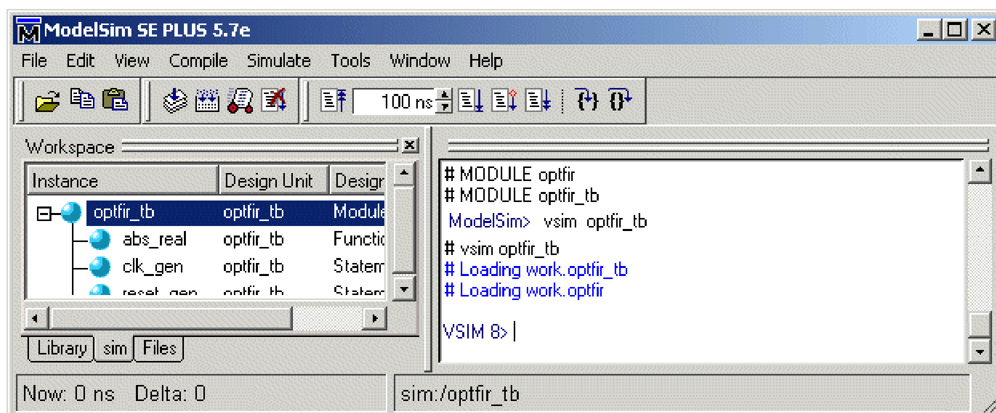
The following screen display shows this command sequence and informational messages displayed during compilation.



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

```
vsim optfir_tb
```

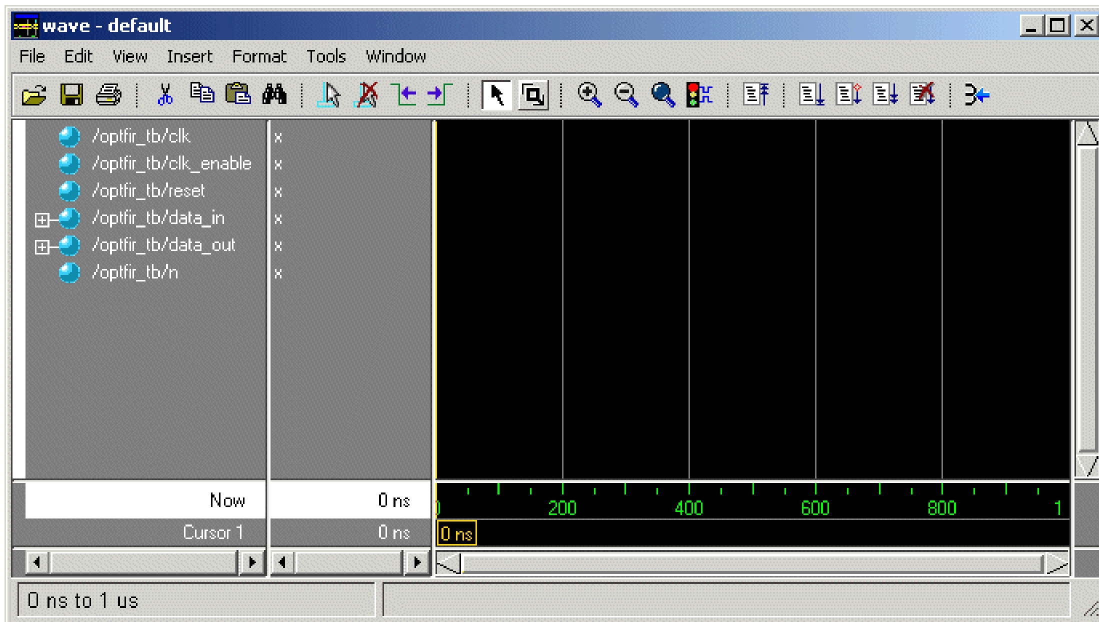
The following ModelSim display shows the results of loading `optfir_tb` with the `vsim` command.



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:

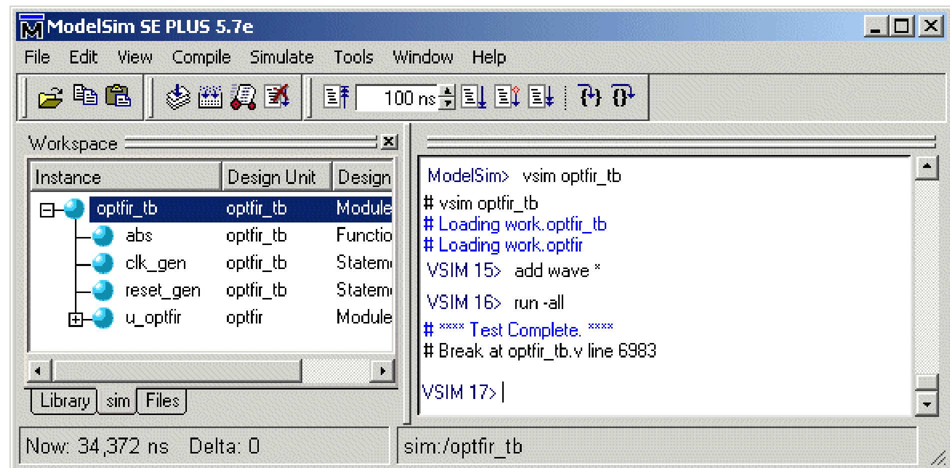
```
add wave *
```

The following **wave** window displays:



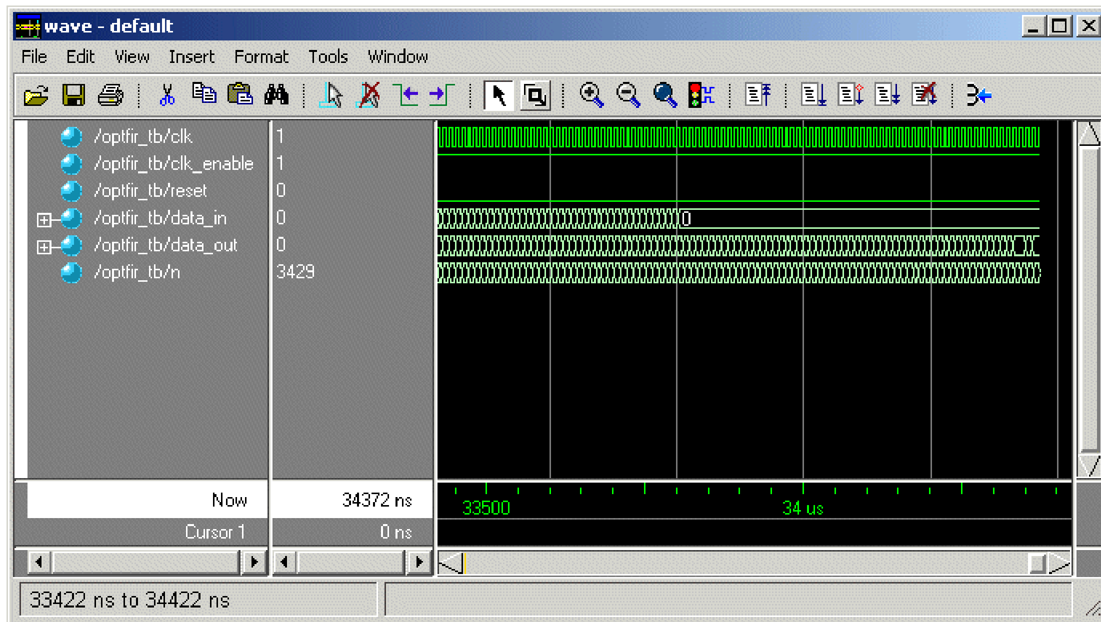
7 To start running the simulation, issue the appropriate command for your simulator. For example, in ModelSim, you can start a simulation with the `run` command.

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 2-44
- "Quantizing the IIR Filter" on page 2-46
- "Configuring and Generating the IIR Filter's VHDL Code" on page 2-50
- "Getting Familiar with the IIR Filter's Generated VHDL Code" on page 2-57
- "Verifying the IIR Filter's Generated VHDL Code" on page 2-58

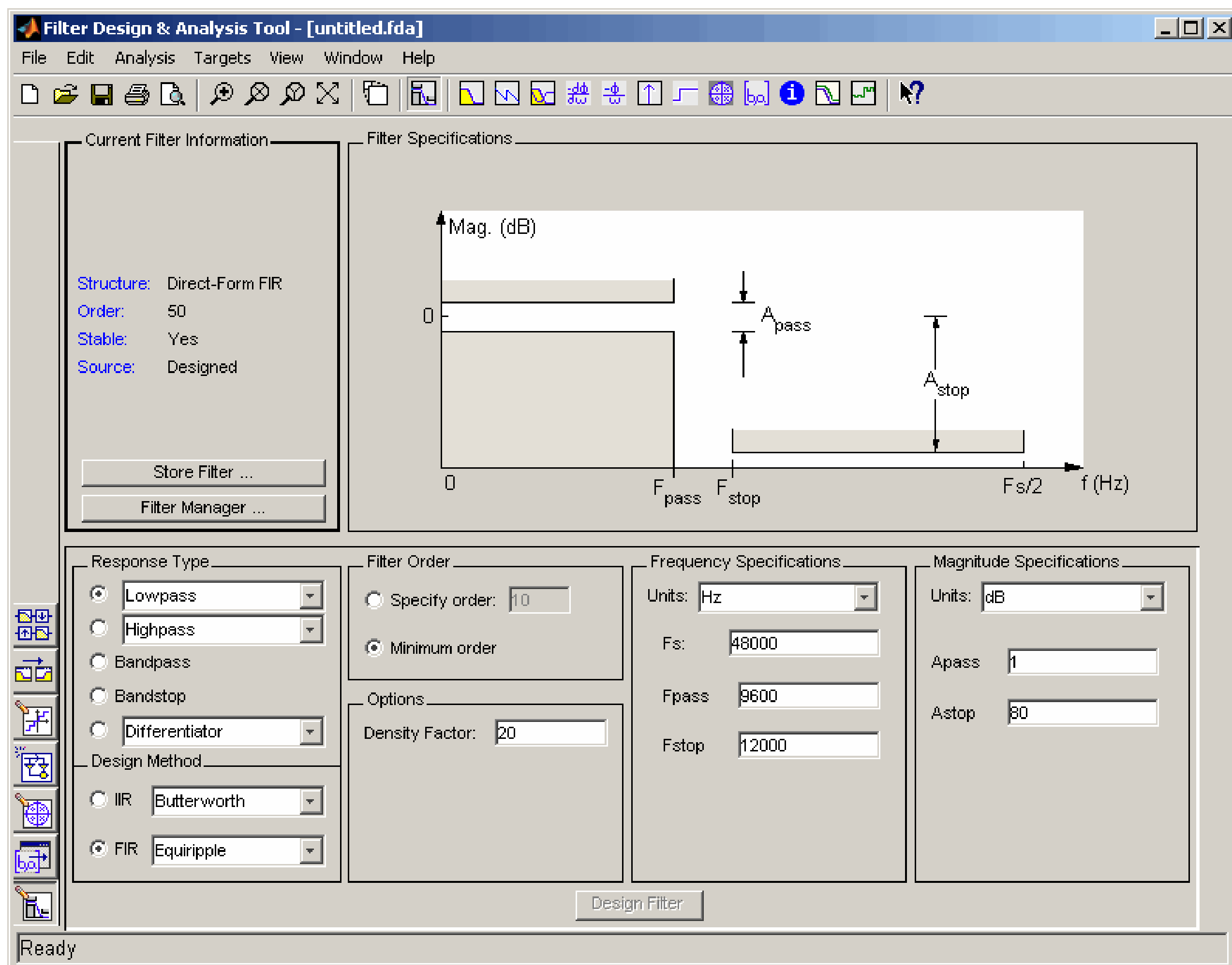
Designing an IIR Filter

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 box.



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

Option	Value
Response Type	`Highpass`
Design Method	`IIR Butterworth`
Filter Order	**Specify order:** `5`
Frequency Specifications	**Units**: `Hz`
	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.

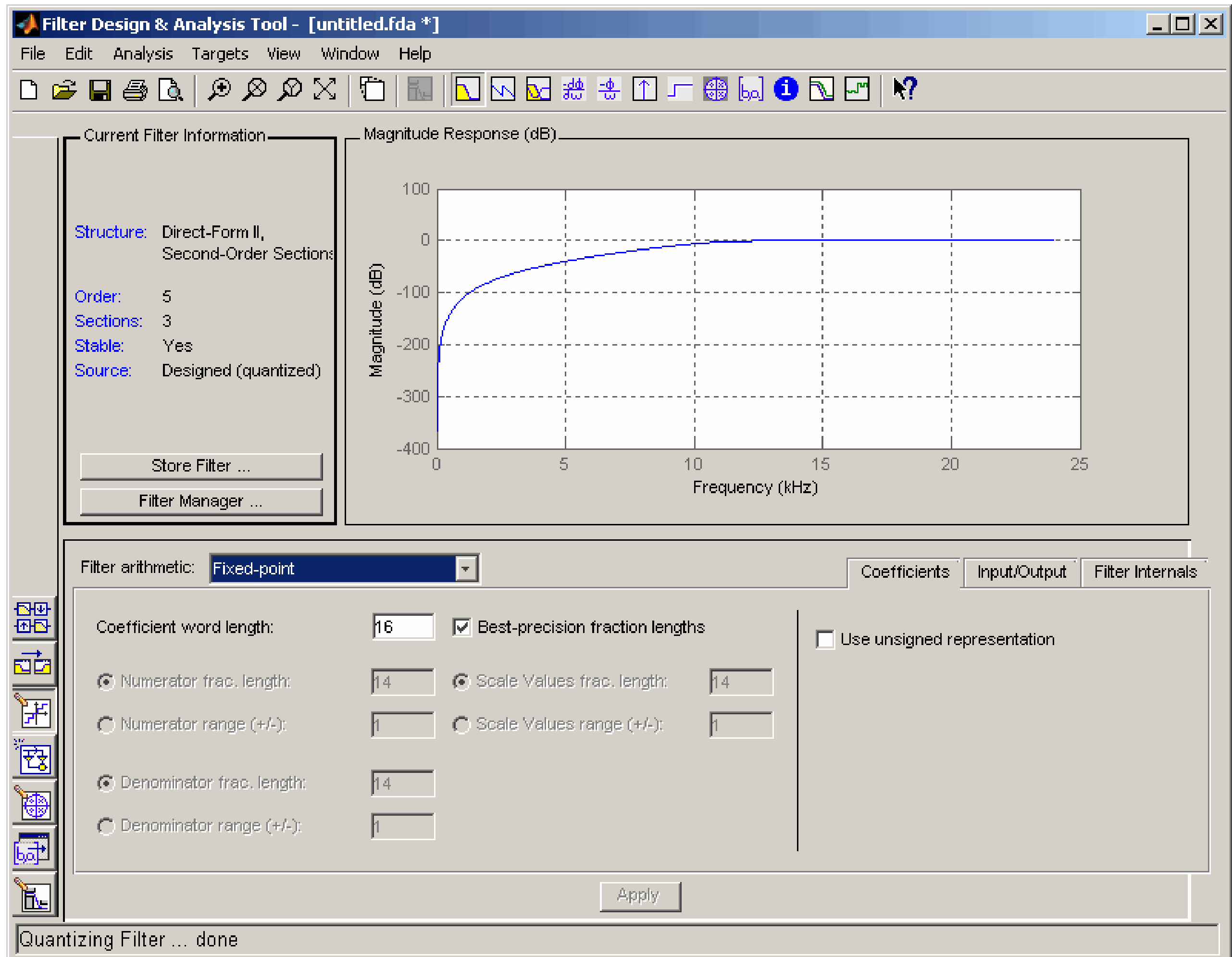
Quantizing the IIR Filter

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 2-44 if it is not already open.

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

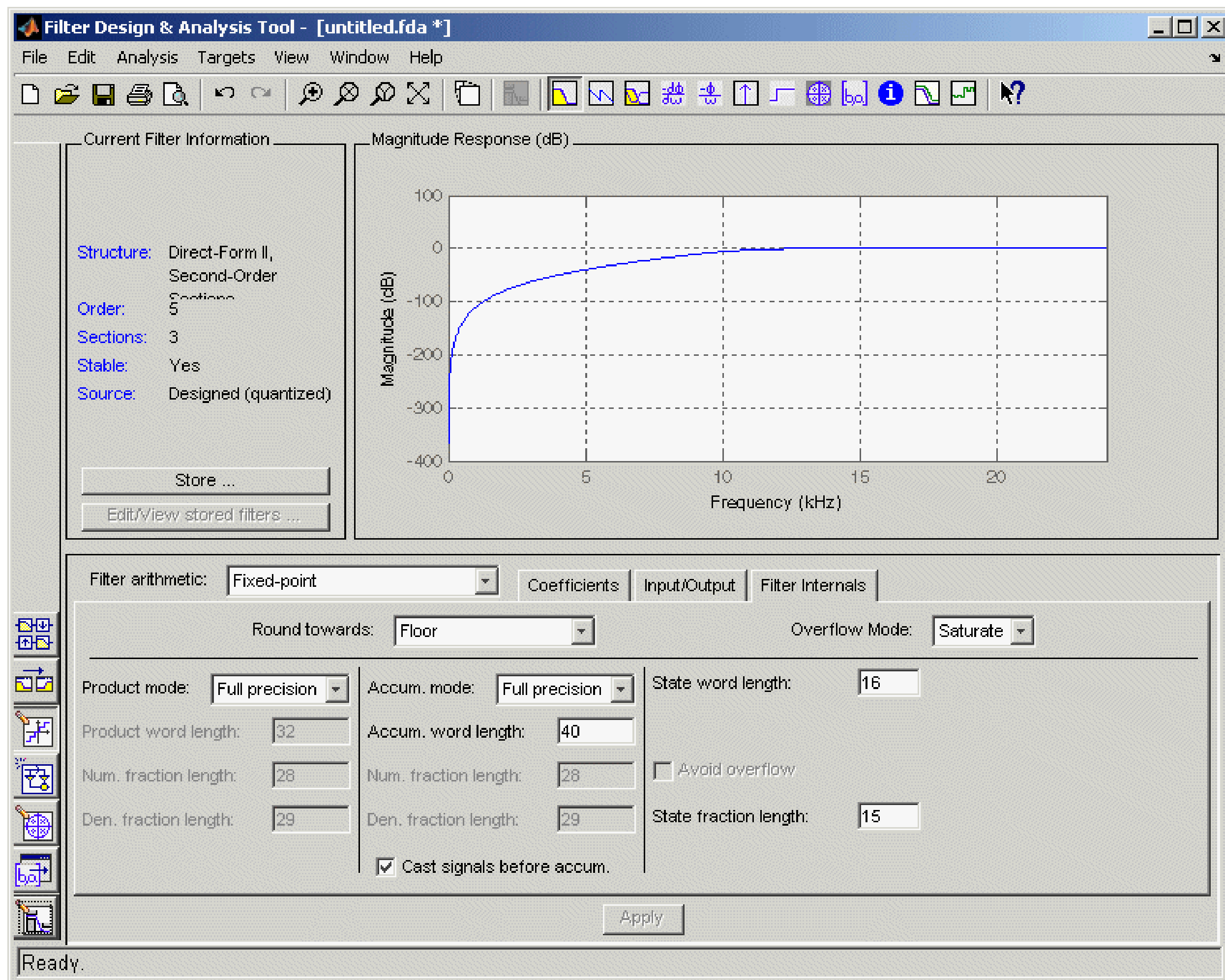
3 Select `Fixed-point` from the list. The FDATool displays the first of three tabbed panels of its dialog box.



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

4 Select the **Filter Internals** tab and set **Round towards** to `Floor` and **Overflow Mode** to `Saturate`.

5 Click **Apply**. The quantized filter appears as follows.

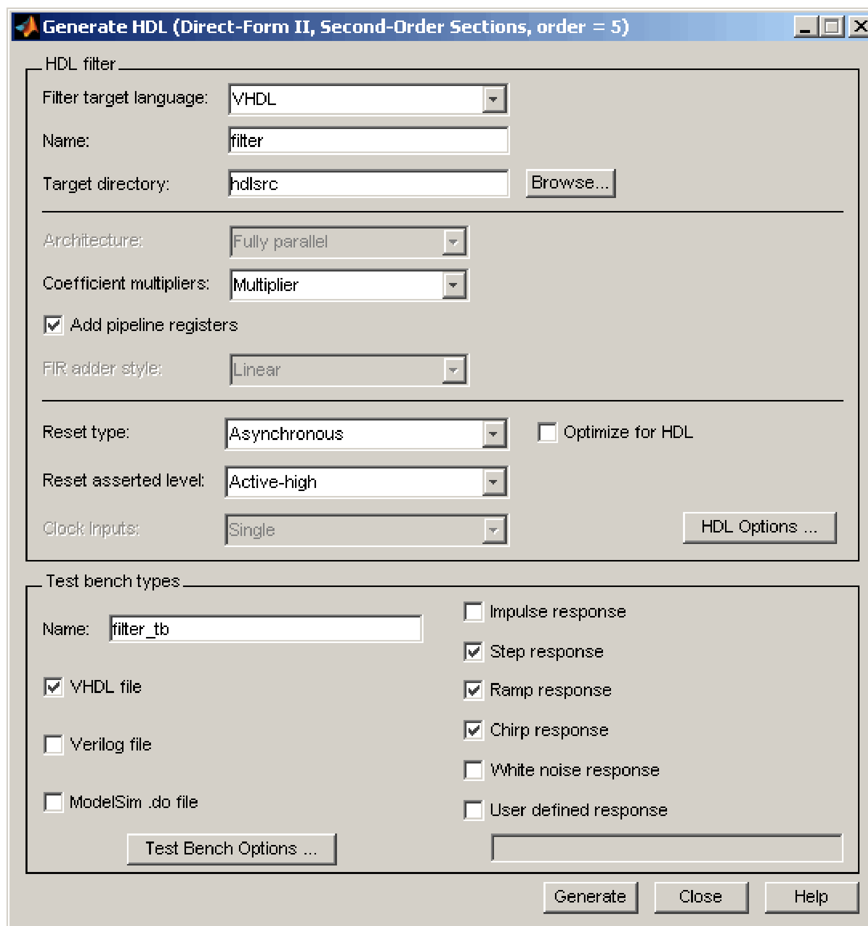


For more information on quantizing filters, see the FDATool and Filter Design Toolbox documentation.

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 2-44 and "Quantizing the IIR Filter" on page 2-46:

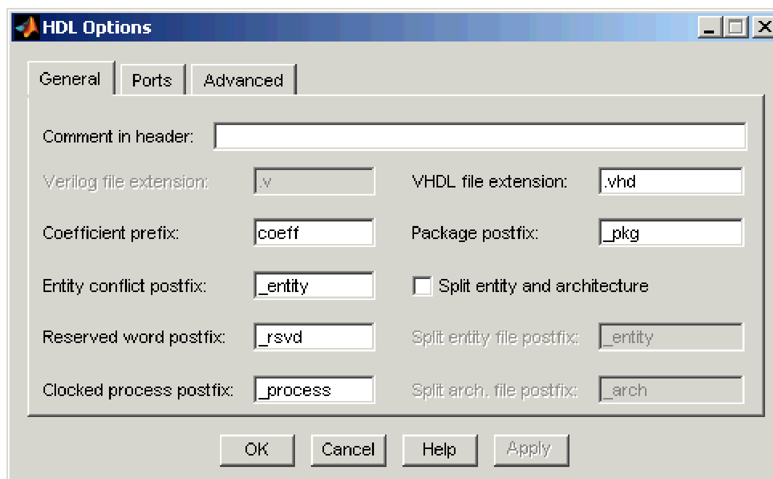
1 Start the Filter Design HDL Coder by selecting **Targets > Generate HDL** in the FDATool dialog box. The FDATool displays the Generate HDL dialog box.



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

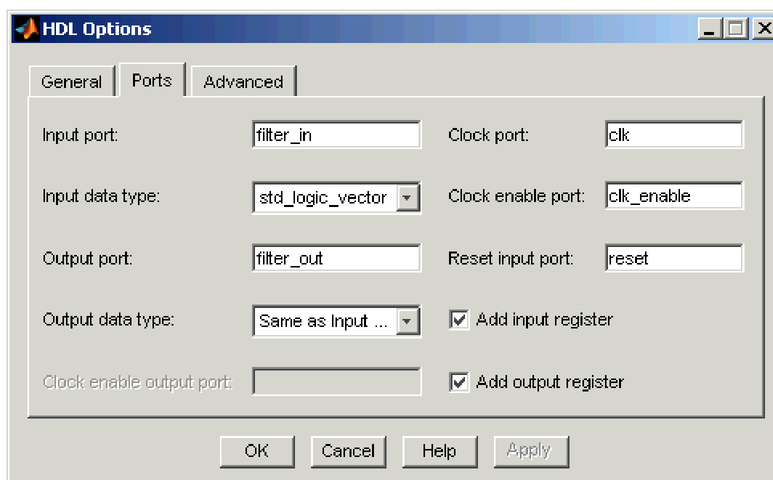
3 In the **Name** text box of the **Test bench types** pane, type `iir_tb`. This option names the generated test bench file.

4 Click **HDL Options**. The Filter Design HDL Coder displays the HDL Options dialog box.

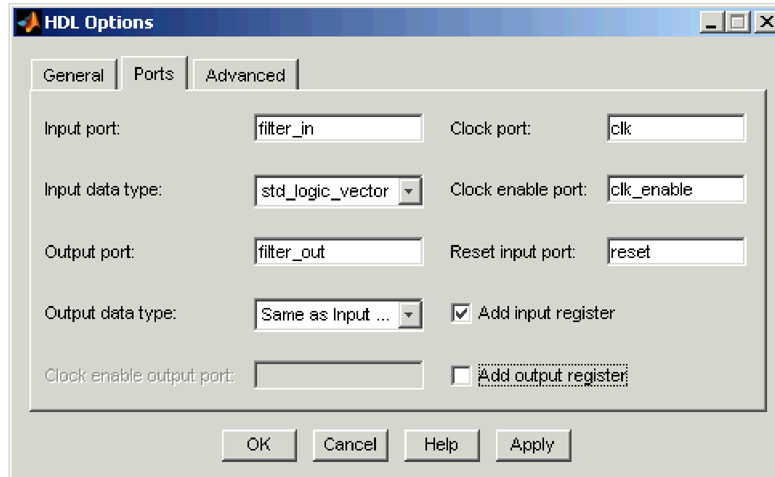


5 In the **Comment in header** text box, type `Tutorial - IIR Filter` and then click **Apply**. The Filter Design HDL Coder adds the comment to the end of the header comment block in each generated file.

6 Select the **Ports** tab. The **Ports** pane appears.

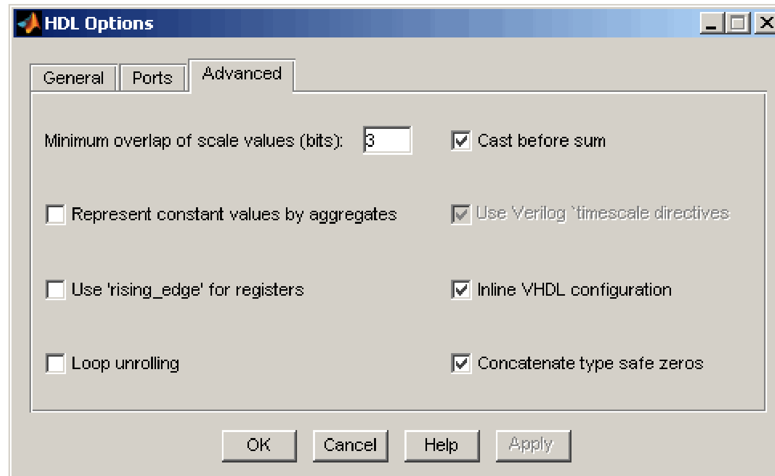


7 Clear the check box for the **Add output register** option. The **Ports** pane should now look like the following.

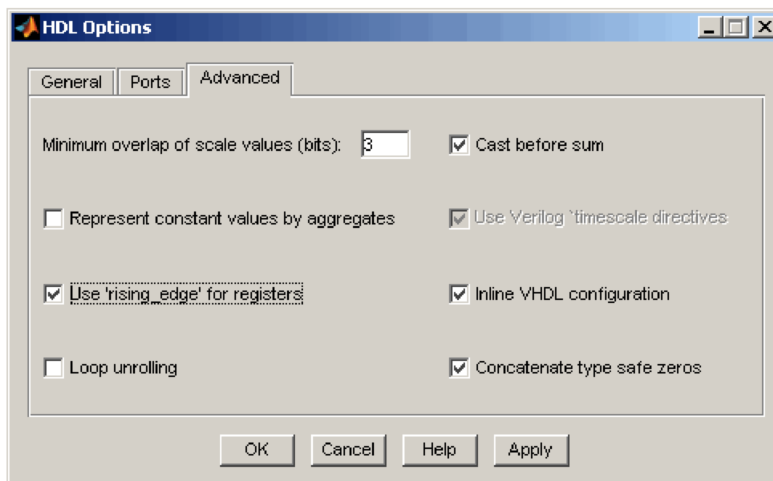


8 Click **Apply**.

9 Select the **Advanced** tab. The **Advanced** pane appears.

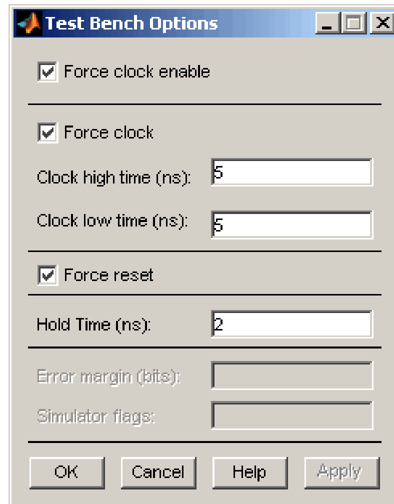


10 Select the **Use 'rising_edge' for registers** option. The **Advanced** pane should now look like the following.



11 Click **Apply** to register your changes and then **OK** to close the dialog box.

12 Click **Test Bench Options**. The Filter Design HDL Coder displays a Test Bench Options dialog box.



You use this dialog box to customize the generated test bench.

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

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

The 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
### HDL latency is 1 samples
### Successful completion of VHDL code generation process for filter: iir
```

```
### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 2172 samples.
### Generating 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:

VHDL Port	Name
Input	`filter_in`
Output	`filter_out`
Clock input	`clk`
Clock enable input	`clk_enable`
Reset input	`reset`

- 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 `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 `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 2-50.
3. Search for `Tutorial`. 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 2-50.
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 2-50.
5. Search for `Filter Settings`. This section of the VHDL code describes the filter design and quantization settings as you specified in "Designing an IIR Filter" on page 2-44 and "Quantizing the IIR Filter" on page 2-46.
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 2-50.

7 Search for `PORT`. This `PORT` declaration defines the filter's clock, clock enable, reset, and data input and output ports. The ports for clock, clock enable, reset, and data input and output signals are named with default strings.

8 Search for `CONSTANT`. This is where the coefficients are defined. They are named using the default naming scheme, `coeff_`*`xm`*`_section`*`n`*, 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** option selected.

11 Search for `IF reset`. This is where the reset signal is asserted. The default, active high (1), was specified. Also note that the `PROCESS` block applies the default asynchronous reset style when generating VHDL code for registers.

12 Search for `ELSIF`. This is where the VHDL code checks for rising edges when the filter operates on registers. The `rising_edge` function is used as you specified in the **Advanced** pane of the HDL Options dialog box. See step 10 in "Configuring and Generating the IIR Filter's VHDL Code" on page 2-50.

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.

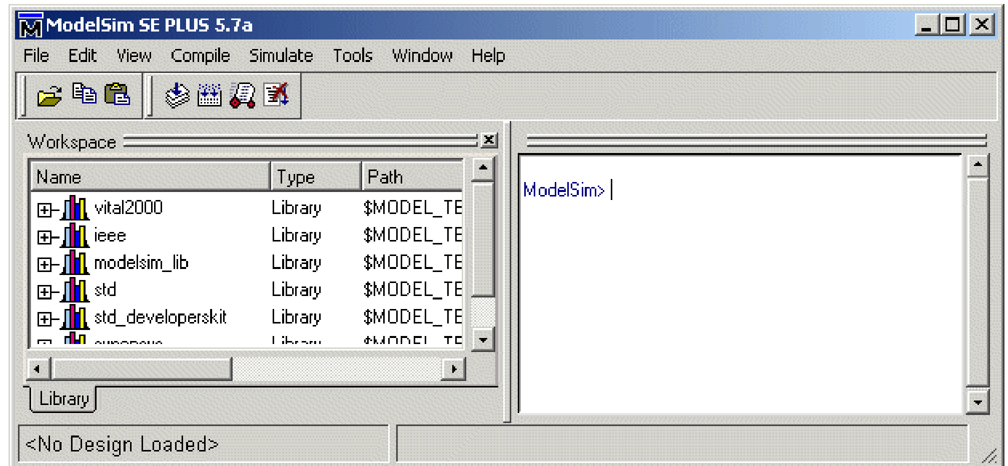
14 Search for `filter_out`. This is where the filter writes its output data.

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
```

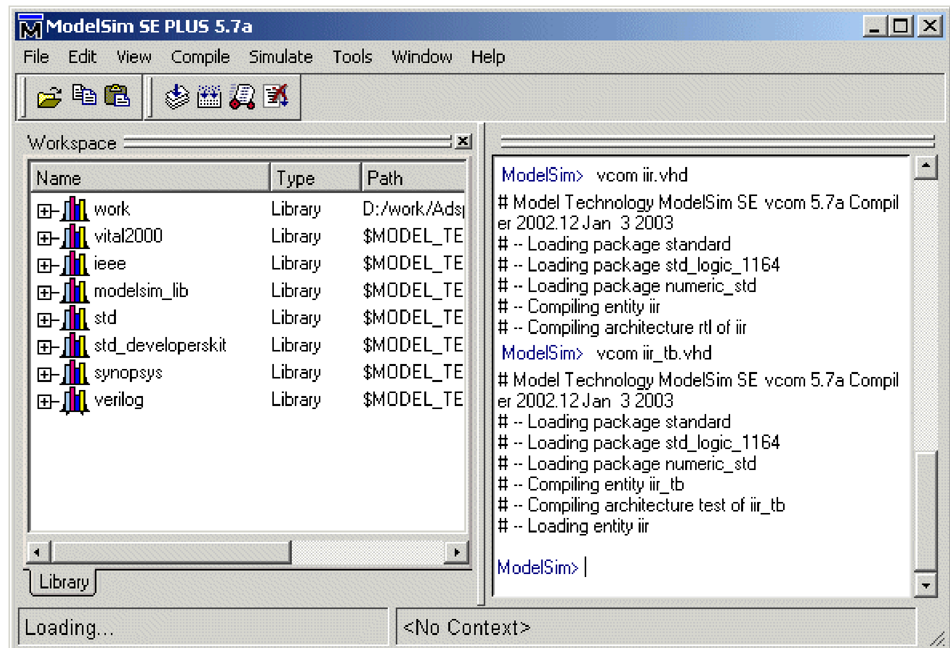
3 If necessary, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In ModelSim, you can create a design library with the `vlib` command.

```
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
```

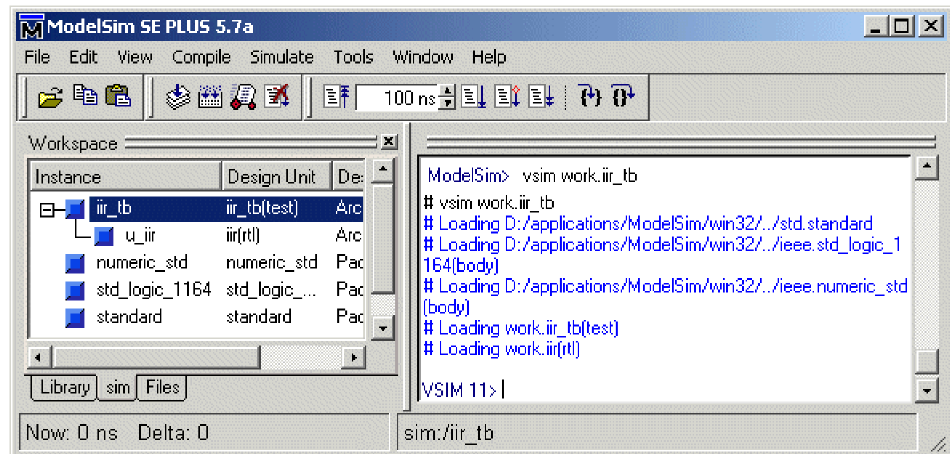
The following screen display shows this command sequence and informational messages displayed during compilation.



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
```

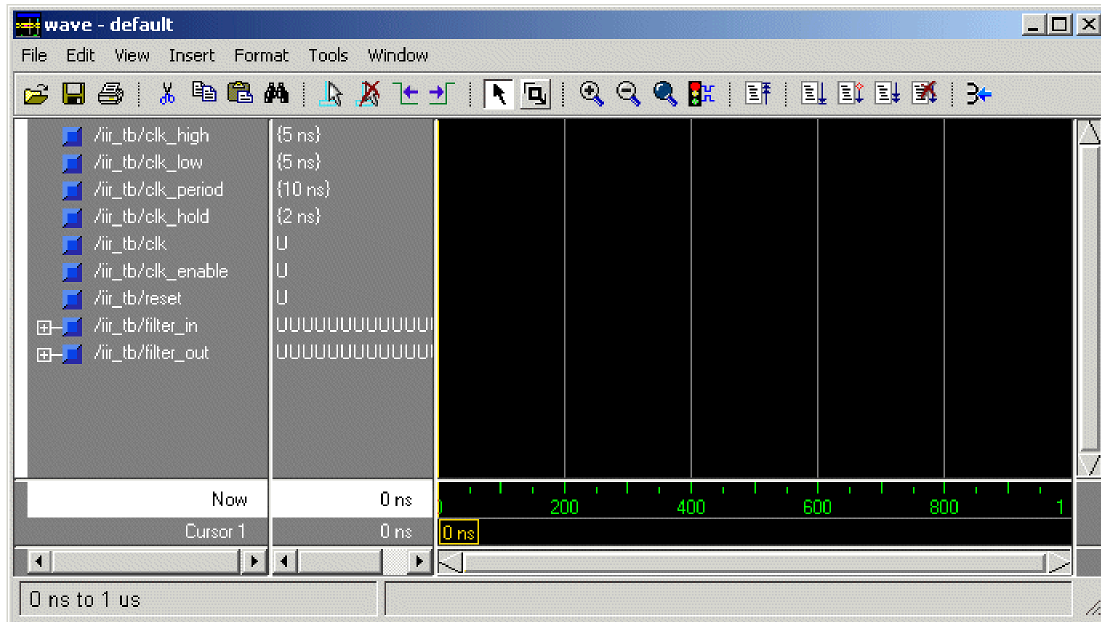
The following ModelSim display shows the results of loading `work.iir_tb` with the `vsim` command:



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.

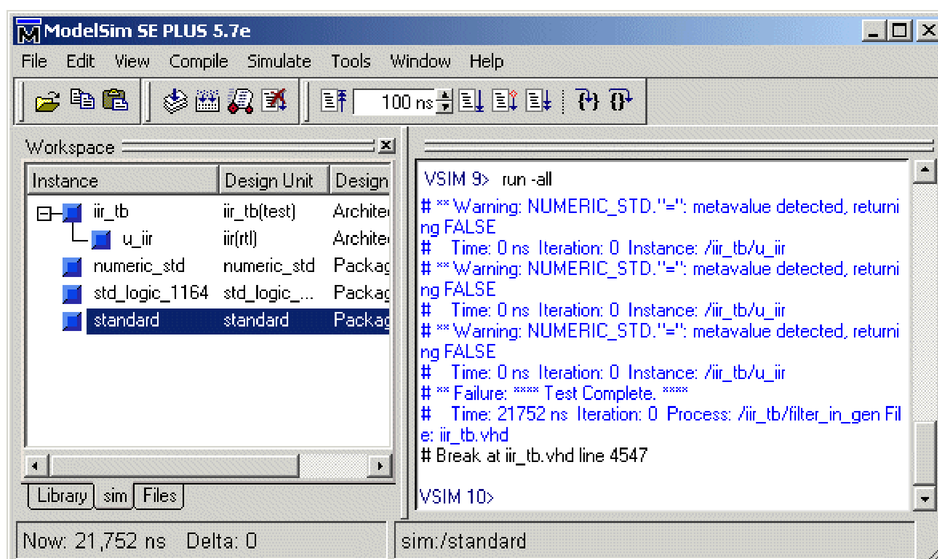
```
add wave *
```

The following **wave** window displays.



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.

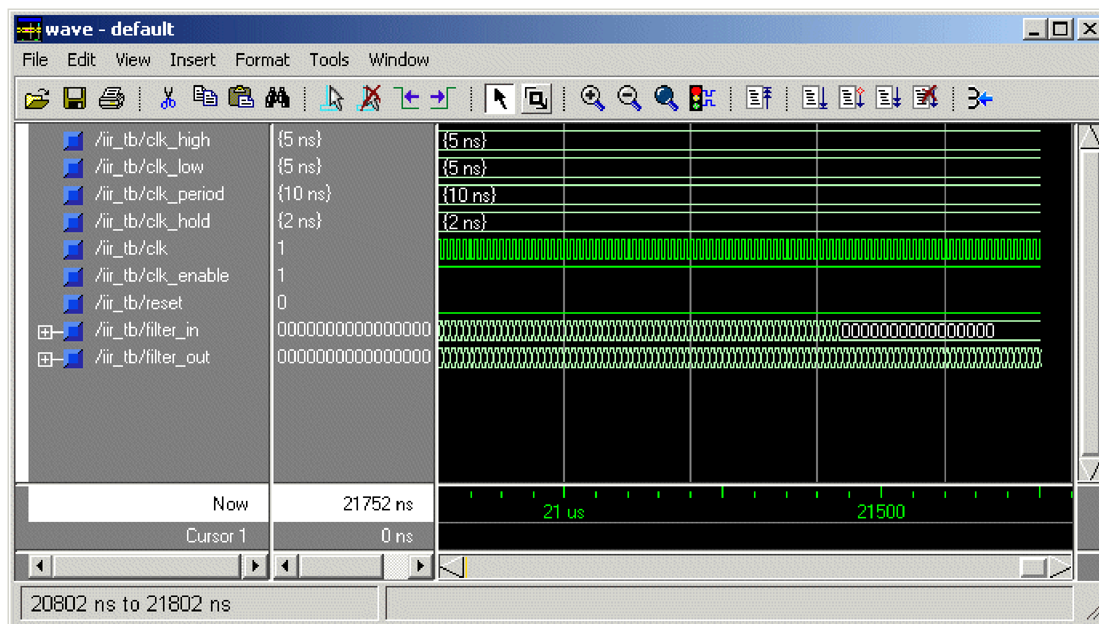


As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. You must determine whether the results are expected based on the customizations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

Note

- The 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.



3

Generating HDL Code for a Filter Design

The Generate HDL dialog box 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 box, 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 box to generate HDL code for a filter design
Opening the Generate HDL Dialog Box (p. 3-5)	Explains how to open the Generate HDL dialog box
What Is Generated by Default? (p. 3-10)	Describes what the Filter Design HDL Coder generates when you specify no customizations
What Are Your HDL Requirements? (p. 3-15)	Provides a checklist that helps you determine whether you need to specify generation customizations
Setting the Target Language (p. 3-21)	Explains how to specify whether VHDL or Verilog filter code is generated

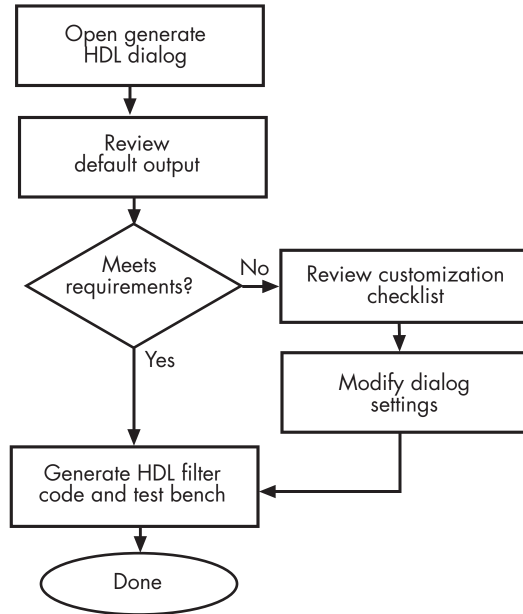
Setting the Names and Location for Generated HDL Files (p. 3-22)	Explains how to explicitly name and specify the location for generated HDL filter and test bench files
Customizing Reset Specifications (p. 3-29)	Explains how to customize the names and location of generated files and specifications for resets
Customizing the HDL Code (p. 3-32)	Explains how to customize various elements of generated HDL code
Setting Optimizations (p. 3-57)	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
Generating Code for Multirate Filters (p. 3-85)	Describes types of multirate filters supported for HDL code generation, and how to specify options for multirate filter code generation
Generating Code for Cascade Filters (p. 3-92)	Describes types of cascade filters supported for HDL code generation, and how to specify options for cascade filter code generation
Customizing the Test Bench (p. 3-95)	Explains how to specify a test bench type, customize clock and reset settings, and adjust the stimulus response
Generating the HDL Code (p. 3-109)	Explains how to initiate HDL code generation discusses the data type conversions that occur during the generation process
Generating Scripts for EDA Tools (p. 3-110)	Explains how to generate and customize scripts for third-party simulation and synthesis tools

Overview of Generating HDL Code for a Filter Design

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

1. Open the Generate HDL dialog box.
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 3-15 and identify required customizations.
5. Modify the Generate HDL dialog box options to address your application requirements, as described in the sections beginning with "Setting the Target Language" on page 3-21 .
6. Generate the filter's HDL code and test bench.

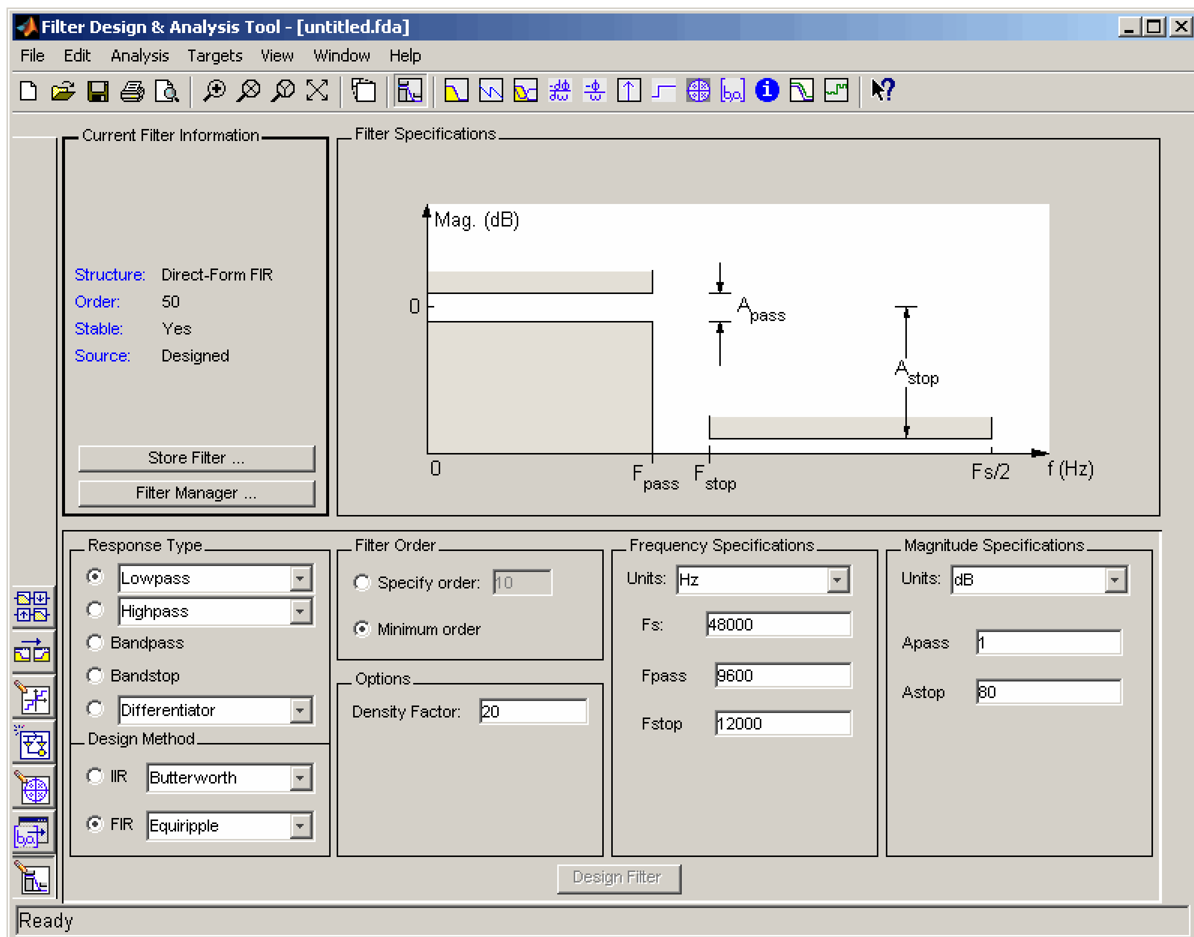
The following figure shows the steps in a flow diagram.

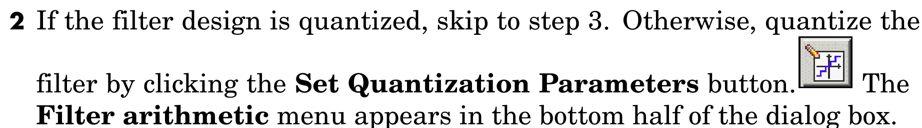


Opening the Generate HDL Dialog Box

The Generate HDL dialog box lets you customize HDL properties and initiate HDL code generation. The dialog box is accessible from the Filter Design and Analysis Tool (FDATool). To open the initial Generate HDL dialog box, do the following:

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

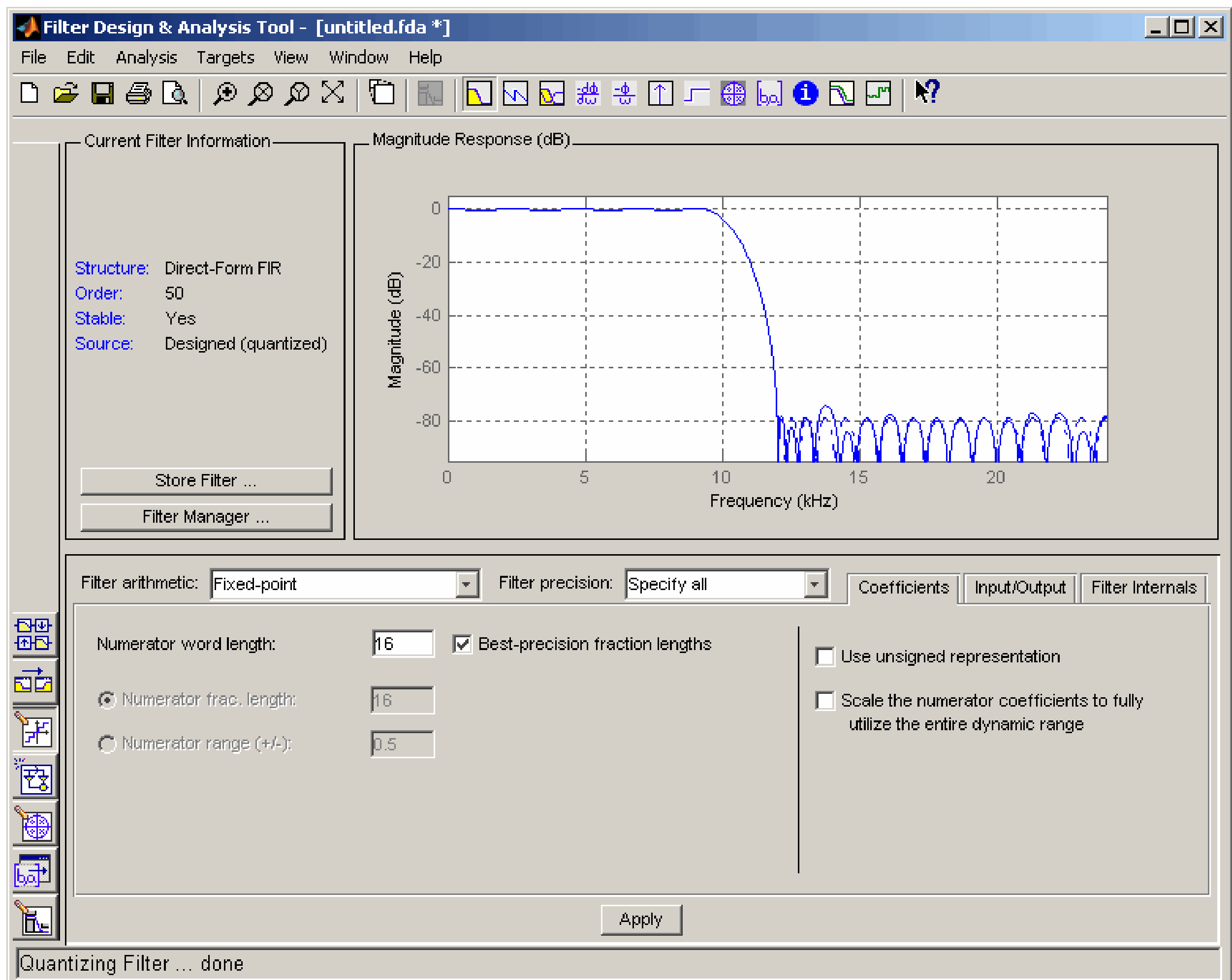


2 If the filter design is quantized, skip to step 3. Otherwise, quantize the filter by clicking the **Set Quantization Parameters** button.  The **Filter arithmetic** menu appears in the bottom half of the dialog box.

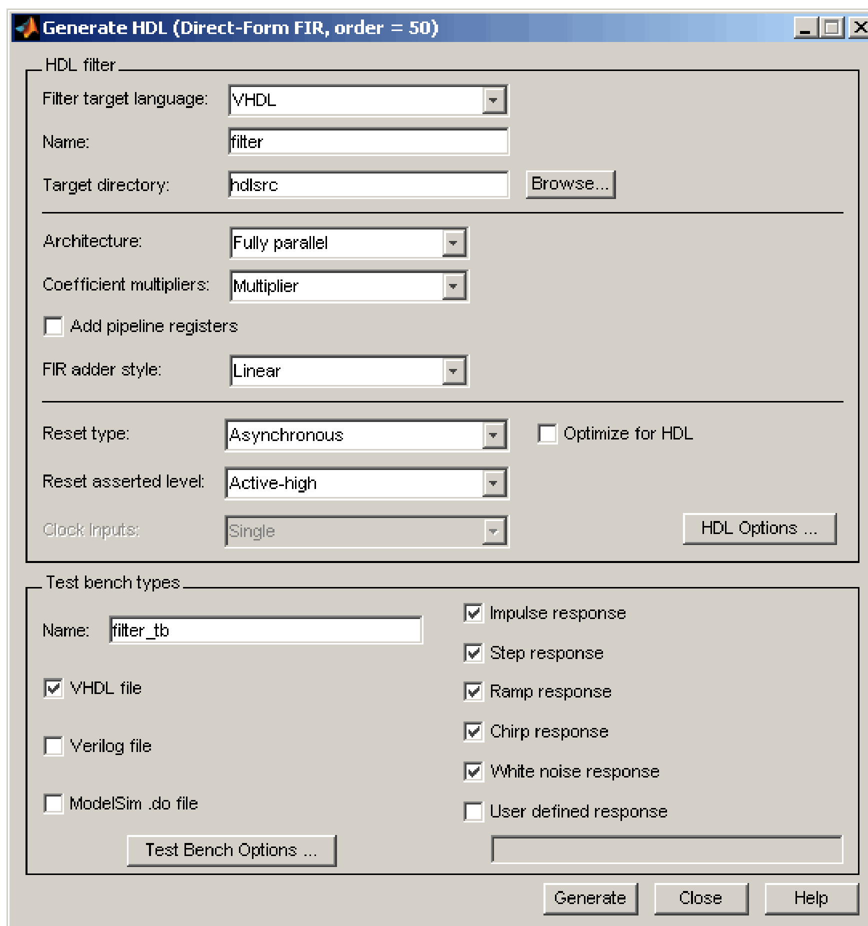
Note All supported filter structures support fixed-point, quantization type, and floating-point (double) realizations.

---

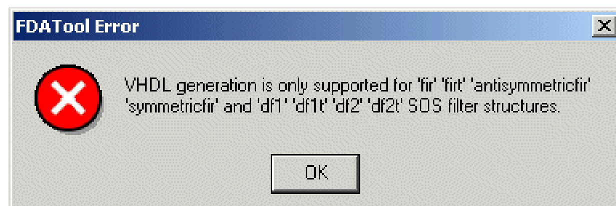
3 If necessary, adjust the setting of the **Filter arithmetic** option. The FDATool displays the first of three tabbed panels of its dialog.



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



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


FDATool Error
VHDL generation is only supported for 'fir' 'firt' 'antisymmetricfir' 'symmetricfir' and 'df1' 'df1t' 'df2' 'df2t' SOS filter structures.
OK

What Is Generated by Default?

The Generate HDL dialog box provides 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 3-10
- "Default Generation of Script Files" on page 3-11
- "Default Settings for Register Resets" on page 3-11
- "Default Settings for General HDL Code" on page 3-11
- "Default Settings for Code Optimizations" on page 3-13
- "Default Settings for Test Benches" on page 3-13

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	`Hd.v`
	Filter test bench	`Hd_tb.v`
VHDL	Filter source	`Hd.vhd`
	Package (if needed)	`Hd_pkg.vhd`
	Test bench	`Hd_tb.vhd`

- 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 3-15 and "Setting the Names and Location for Generated HDL Files" on page 3-22.

Default Generation of Script Files

Filter Design HDL Coder supports generation of script files for third-party Electronic Design Automation (EDA) tools.

Using the defaults, you can automatically generate scripts for the following tools:

- Mentor Graphics ModelSim® SE/PE HDL simulator
- The Synplify family of synthesis tools

See "Generating Scripts for EDA Tools" on page 3-110 for detailed information on generation and customization of scripts.

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 3-15 and "Customizing Reset Specifications" on page 3-29.

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:

HDL Port	Name
Input	`filter_in`
Output	`filter_out`
Clock input	`clk`

HDL Port	Name
Clock enable input	`clk_enable`
Reset input	`reset`

- Sets the data types for HDL ports as follows:

HDL Port	VHDL Type	Verilog Type
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:

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.
- 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 3-15 and "Customizing the HDL Code" on page 3-32.

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.
- For FIR filters, generates a fully parallel architecture (optimal for speed).
- 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 3-15 and "Setting Optimizations" on page 3-57.

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...	Applies Response Types...
FIR, FIRT, symmetric FIR, and antisymmetric FIR	Impulse, step, ramp, chirp, and white noise
All others	Step, ramp, and chirp

For information on modifying these settings, see "What Are Your HDL Requirements?" on page 3-15 and "Customizing the Test Bench" on page 3-95.

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...
Language Selection		
☐	Do you need to adjust the target language setting?	“Setting the Target Language” on page 3-21
File Naming and Location Specifications		
☐	Do you want to specify a unique **name**, which does *not* match the name of the quantized filter, for the VHDL **entity** or Verilog **module** that represents the filter?	“Setting the Names and Location for Generated HDL Files” on page 3-22
☐	Do you want the **file type extension** 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 3-22
Reset Specifications		
☐	Do you want to use **synchronous resets**?	“Setting the Reset Style for Registers” on page 3-29
☐	Do you need the **asserted level of the reset** signal to be low (0)?	“Setting the Asserted Level for the Reset Input Signal” on page 3-30
Header Comment and General Naming Specifications		
☐	Do you want to add a specific string, such as a revision control string, to the end of the **header comment block** in each generated file?	“Specifying a Header Comment” on page 3-33
☐	Do you want a string other than `coeff` to be used as the **base filter coefficient name**?	“Specifying a Prefix for Filter Coefficients” on page 3-35

HDL Requirements Checklist (Continued)

	Requirement	For More Information, See...
☐	If your filter design requires a **VHDL package file**, do you want the name of the generated file to include a string other than _pkg?	"Setting the Postfix String for VHDL Package Files" on page 3-25
☐	Do you want a string other than _entity to be appended to VHDL entity or Verilog module names if **duplicate names** are detected?	"Setting the Postfix String for Resolving Entity or Module Name Conflicts" on page 3-36
☐	Do you want a string other than _rsvd to be appended to specified names and labels that are **HDL reserved words**?	"Setting the Postfix String for Resolving HDL Reserved Word Conflicts" on page 3-37
☐	Do you want a string other than _process to be appended to **HDL process names**?	"Setting the Postfix String for Process Block Labels" on page 3-40
☐	Do you want the Filter Design HDL Coder to write the **entity** and **architecture** parts of generated VHDL code to **separate files**?	"Splitting Entity and Architecture Code into Separate Files" on page 3-26
☐	If the Filter Design HDL Coder writes the **entity and architecture** parts of VHDL code to separate files, do you want strings other than _entity and _arch included in the **filenames**?	"Splitting Entity and Architecture Code into Separate Files" on page 3-26
Port Specifications		
☐	Do you want the Filter Design HDL Coder to use strings other than filter_in and filter_out to name **HDL ports** for the filter's data input and output signals?	"Naming HDL Ports" on page 3-42
☐	Do you need the Filter Design HDL Coder to declare the filter's **data input and output ports** with a **VHDL type** other than STD_LOGIC_VECTOR?	"Specifying the HDL Data Type for Data Ports" on page 3-43
☐	Do you want the Filter Design HDL Coder to use strings other than clk and clk_enable to name HDL ports for the filter's **clock and clock enable** input signals?	"Naming HDL Ports" on page 3-42

HDL Requirements Checklist (Continued)

	Requirement	For More Information, See...
☐	Do you want the Filter Design HDL Coder to use a string other than `reset` to name an HDL port for the filter's **reset** input signals?	"Naming HDL Ports" on page 3-42
☐	Do you want the Filter Design HDL Coder to add an **extra input or output register** to support the filter's HDL input and output ports?	"Suppressing Extra Input and Output Registers" on page 3-45
	Advanced Coding Specifications	
☐	Do you expect the filter's **coefficient scale values** to be more than 3 bits smaller than the size of the filter's input?	" Minimizing Quantization Noise for Fixed-Point Filters" on page 3-46
☐	Do you want the Filter Design HDL Coder to represent all **constants as aggregates**?	"Representing Constants with Aggregates" on page 3-48
☐	Are you using an EDA tool that does not support **loops**? Do you need the Filter Design HDL Coder to unroll and remove VHDL `FOR` and `GENERATE` loops?	"Unrolling and Removing VHDL Loops" on page 3-49
☐	Do you want the Filter Design HDL Coder to use the VHDL `rising_edge` function to check for **rising edges** when the filter is operating on registers?	"Using the VHDL rising_edge Function" on page 3-50
☐	Do you want to suppress Verilog **time scale directives**?	"Suppressing Verilog Time Scale Directives" on page 3-54
☐	Do you want the Filter Design HDL Coder to omit **configurations** 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 3-52
☐	Do you want the Filter Design HDL Coder to use the VHDL syntax `"000000..."` to represent **concatenated zeros** instead of the type safe representation `'0' & '0'`?	"Specifying VHDL Syntax for Concatenated Zeros" on page 3-53

HDL Requirements Checklist (Continued)

	Requirement	For More Information, See...
☐	Do you want the Filter Design HDL Coder to apply typical DSP processor treatment of input data types when generating code for **addition and subtraction operations**?	"Specifying Input Type Treatment for Addition and Subtraction Operations" on page 3-55
Optimization Specifications		
☐	Do you need **numeric results** optimized, even if the results are not bit-true to the MATLAB filter function?	"Optimizing Generated Code for HDL" on page 3-58
☐	Do you want the Filter Design HDL Coder to replace **multiplier operations** by applying canonic signed digit (CSD) and factored CSD techniques?	"Optimizing Coefficient Multipliers" on page 3-59
☐	Do you need the Filter Design HDL Coder to optimize the **final summation for FIR filters**?	"Optimizing Final Summation for FIR Filters" on page 3-60
☐	Do you need to specify an **optimal FIR filter architecture** with respect to speed or chip area?	"Speed vs. Area Optimizations for FIR Filters" on page 3-61
☐	Do you need to use a **Distributed arithmetic** architecture for a fixed-point FIR filter?	"Distributed Arithmetic for FIR Filters" on page 3-71
☐	Do you want to optimize your filter's **clock rate**?	"Optimizing the Clock Rate with Pipeline Registers" on page 3-81
Multirate and Cascade Filter Specifications		
☐	Do you need to generate code for a **multirate filter** ?	"Generating Code for Multirate Filters" on page 3-85
☐	Do you need to generate code for a **cascade** of filter objects?	"Generating Code for Cascade Filters" on page 3-92
Test Bench Specifications		
☐	Do you want the **name** of the generated **test bench** file to include a string other than `_tb`?	"Setting the Names and Location for Generated HDL Files" on page 3-22
☐	Do you want to generate a **VHDL test bench**?	"Specifying a Test Bench Type" on page 3-97

HDL Requirements Checklist (Continued)

	Requirement	For More Information, See...
☐	Do you want to generate a **Verilog file test bench**?	"Specifying a Test Bench Type" on page 3-97
☐	Do you want to generate a **ModelSim .do file test bench**?	"Specifying a Test Bench Type" on page 3-97
☐	If the test bench type is a ModelSim `.do` file, does your application require you to specify any **simulation flags**?	"Specifying a Test Bench Type" on page 3-97
☐	Are you using a user-defined external source to **force clock enable** input signals to a constant value?	"Configuring the Clock" on page 3-99
☐	If the test bench is to force clock enable input signals, do you want it to force the signals to **active low** (0)?	"Configuring the Clock" on page 3-99
☐	Are you using a user-defined external source to **force clock** input signals?	"Configuring the Clock" on page 3-99
☐	If the test bench is to force clock input signals, do you want the signals to be **driven high or low** for a **duration** other than 5 nanoseconds?	"Configuring the Clock" on page 3-99
☐	Are you using a user-defined external source to **force reset** input signals?	"Configuring Resets" on page 3-101
☐	If the test bench is to force reset input signals, do you want it to force the signals to **active low** (0)?	"Configuring Resets" on page 3-101
☐	If the test bench is to force **reset** input signals, do you want it to apply a **hold time** other than two cycles plus a hold time of 2 nanoseconds?	"Configuring Resets" on page 3-101
☐	Do you want to apply a **hold time** other than 2 nanoseconds to **filter data** input signals?	"Setting a Hold Time for Data Input Signals" on page 3-103
☐	Do you want to customize the **stimulus** to be applied by the test bench?	"Setting Test Bench Stimuli" on page 3-106

HDL Requirements Checklist (Continued)

	Requirement	For More Information, See...
Script Generation Specifications		
☐	Do you want to customize **script code** that is auto-generated for third-party EDA tools?	"Generating Scripts for EDA Tools" on page 3-110
☐	Do you want to customize **script file names** for auto-generated EDA tool scripts??	"Generating Scripts for EDA Tools" on page 3-110

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 box options that are specific to Verilog are grayed 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 box. This setting causes the coder to enable options that are specific to Verilog and to gray out and disable options that are specific to VHDL.

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

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`.

Language	Generated File	Filename	Example
Verilog	Source file for the quantized filter	*dfilt_name*.v	Hq.v
	Source file for the filter's test bench	*dfilt_name*_tb.v	Hq_tb.v
VHDL	Source file for the quantized filter	*dfilt_name*.vhd	Hq.vhd
	Source file for the filter's test bench	*dfilt_name*_tb.vhd	Hq_tb.vhd
	Package file, if required by the filter design	*dfilt_name*_pkg.vhd	Hq_pkg.vhd

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 3-23

- "Redirecting Filter Design HDL Coder Output" on page 3-24
- "Setting the Postfix String for VHDL Package Files" on page 3-25
- "Splitting Entity and Architecture Code into Separate Files" on page 3-26

Setting Filter Entity and General File Naming Strings

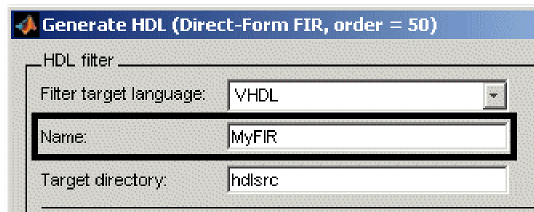
To set the string that the Filter Design HDL Coder uses to name the filter entity or module and generated files, specify a new value in the **Name** field of the **HDL filter** pane of the Generate HDL dialog box. The Filter Design HDL Coder uses the **Name** string to

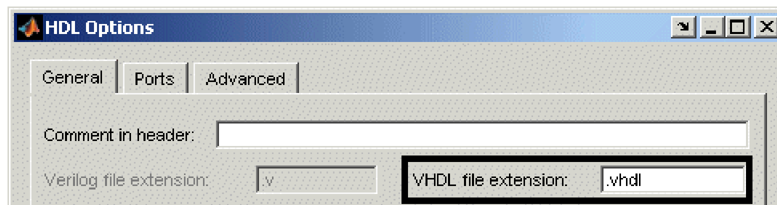
- Label the VHDL entity or Verilog module for your filter
- Name the file containing the HDL code for your filter
- Derive names for the filter's test bench and package files

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

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





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

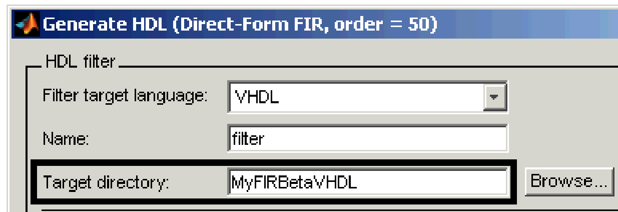
By default, the Filter Design HDL Coder places all generated HDL files in the subdirectory `hdlsrc` under your current working directory. To direct Filter Design HDL Coder output to a directory other than the default target directory, you can use either the **Target directory** field or the **Browse** button in the **HDL filter** pane of the Generate HDL dialog box.

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

Alternatively, you can enter the directory specification directly into the **Target directory** field. 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 setting instructs the Filter Design HDL Coder to create the subdirectory `MyFIRBetaVHDL` under the current working directory and write generated HDL files to that directory.

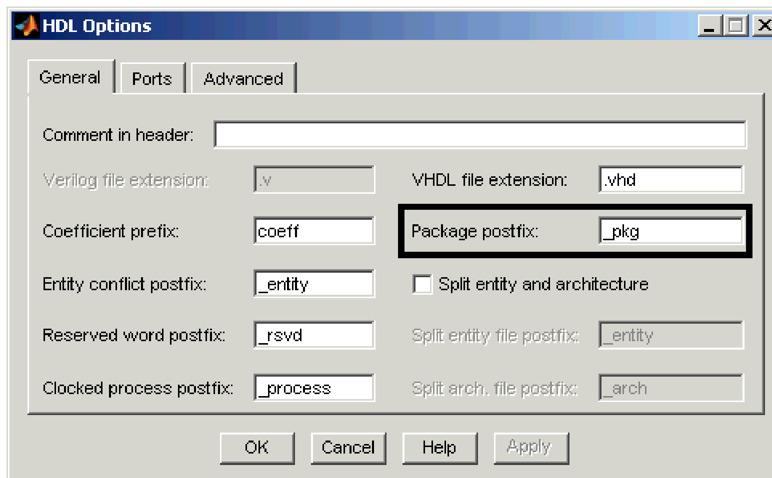
Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the`TargetDirectory` property to redirect Filter Design HDL Coder output.

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.
2. Select the **General** tab.

3 Specify a new value in the **Package postfix** 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 box.

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

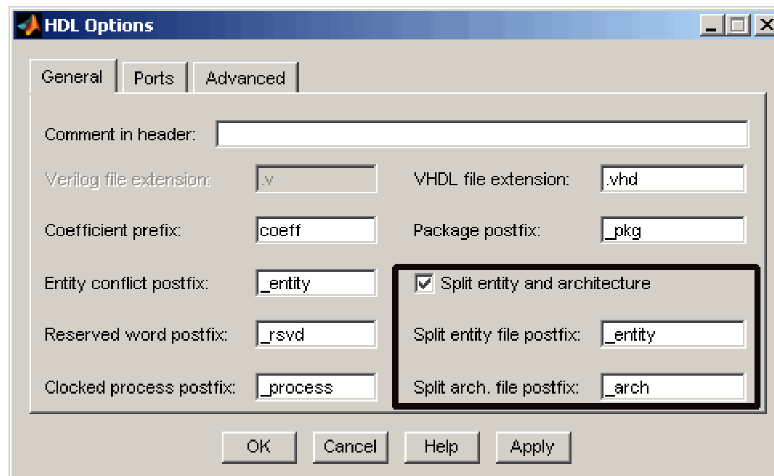
- The base filename, as specified by the **Name** field in the **HDL filter** pane of the Generate HDL dialog box
- Default postfix string values _entity and _arch
- The VHDL file type extension, as specified by the **VHDL file extension** field on the **General** pane of the HDL Options dialog box

To split the filter source file, do the following:

1. Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2. Select the **General** tab.

3. Select **Split entity and architecture**. The Filter Design HDL Coder enables the **Split entity file postfix** and **Split arch. file postfix** fields.



4. Specify new strings in the postfix 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 box.

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 portion of the **HDL filter** pane of the Generate HDL dialog box, as shown in the following figure.

Use the reset options for

- "Setting the Reset Style for Registers" on page 3-29
- "Setting the Asserted Level for the Reset Input Signal" on page 3-30

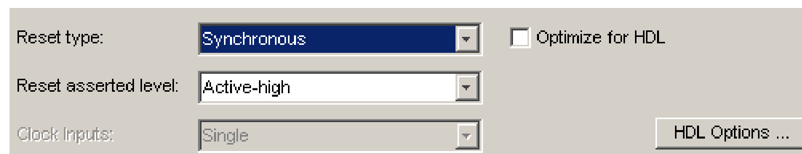
Setting the Reset Style for Registers

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 box.



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.

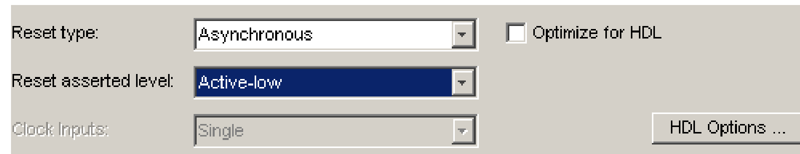
Setting the Asserted Level for the Reset Input Signal

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 box.



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

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

Note The **Reset asserted level** setting also determines the rest level for test bench reset input signals.

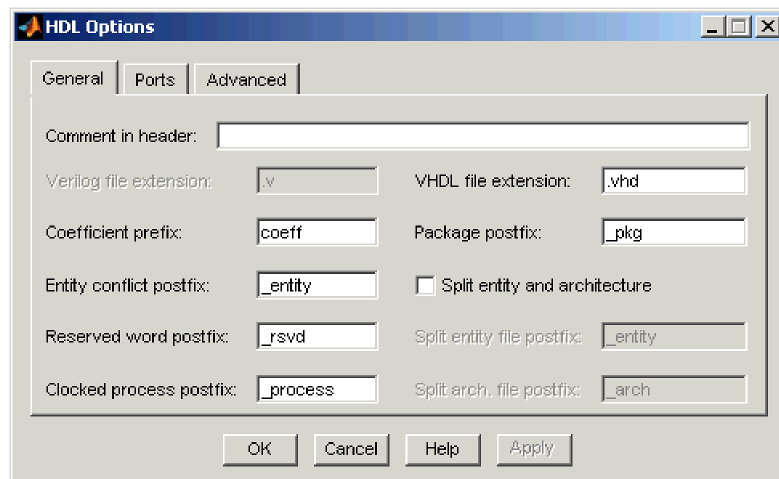
Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the property `ResetAssertedLevel` to set the asserted level for the filter's reset input signal.

Customizing the HDL Code

You select most HDL code customizations from options on the HDL Options dialog box. Options that are specific to VHDL or Verilog are active only if that language is selected. Inactive options appear gray 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 box 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 box to specify naming, port, and advanced coding customizations:

- "Specifying a Header Comment" on page 3-33
- "Specifying a Prefix for Filter Coefficients" on page 3-35

- “Setting the Postfix String for Resolving Entity or Module Name Conflicts” on page 3-36
- “Setting the Postfix String for Resolving HDL Reserved Word Conflicts” on page 3-37
- “Setting the Postfix String for Process Block Labels” on page 3-40
- “Naming HDL Ports” on page 3-42
- “Specifying the HDL Data Type for Data Ports” on page 3-43
- “Suppressing Extra Input and Output Registers” on page 3-45
- “ Minimizing Quantization Noise for Fixed-Point Filters” on page 3-46
- “Representing Constants with Aggregates” on page 3-48
- “Unrolling and Removing VHDL Loops” on page 3-49
- “Using the VHDL rising_edge Function” on page 3-50
- “Suppressing the Generation of VHDL Inline Configurations” on page 3-52
- “Specifying VHDL Syntax for Concatenated Zeros” on page 3-53
- “Suppressing Verilog Time Scale Directives” on page 3-54
- “Specifying Input Type Treatment for Addition and Subtraction Operations” on page 3-55

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: 1.1.4.92 $`. With this change, the preceding header comment block would appear as follows:

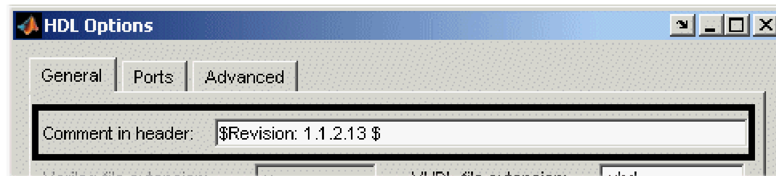
```
------------------------------------------------------------
--
-- Module:Hd
--
-- Generated by MATLAB(R) 7.0 and the Filter Designer HDL Coder 1.0.
--
-- Generated on: 2004-02-04 09:42:43
--
-- $Revision: 1.1.4.92 $
------------------------------------------------------------
```

To add a header comment,

1 Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **General** tab. General HDL coding options appear.

3 Type the comment string in the **Comment in header** field, as shown in the following display.



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

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 `_section`*n*, where *n* is the section number. Example: `coeff_b1_section3` (first numerator coefficient of the third section)

For example:

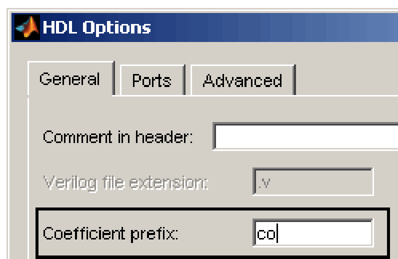
```
ARCHITECTURE rtl OF Hd IS
  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF signed(15 DOWNTO 0); -- sfix16_En15
  CONSTANT coeff1                : signed(15 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
  CONSTANT coeff2                : signed(15 DOWNTO 0) := to_signed(-89, 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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **General** tab.

3 Enter a new string in the **Coefficient prefix** field, 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 box.

Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the property `CoeffPrefix` 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,

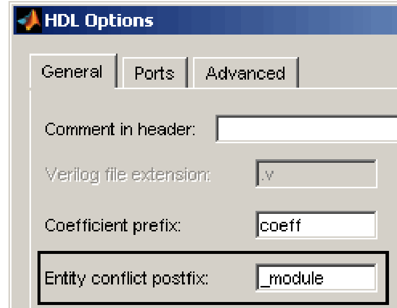
the Filter Design HDL Coder appends the postfix `_entity` to the second of the two matching strings.

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

1 Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **General** tab.

3 Enter a new string in the **Entity conflict postfix** field, as shown in the following display.



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

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. See the tables below for listings of all VHDL and Verilog reserved words.

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **General** tab.

3 Enter a new string in the **Reserved word postfix** field, as shown in the following display.

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

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

VHDL Reserved Words

`abs`	`access`	`after`	`alias`	`all`
`and`	`architecture`	`array`	`assert`	`attribute`
`begin`	`block`	`body`	`buffer`	`bus`
`case`	`component`	`configuration`	`constant`	`disconnect`

VHDL Reserved Words (Continued)

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

Verilog Reserved Words (Continued)

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_onevent	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
trireg	unsigned	use	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

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
```

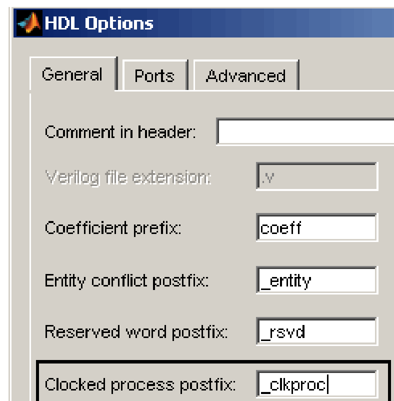
```
      IF clk_enable = '1' THEN
        delay_pipeline(0) <= signed(filter_in)
        delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
      END IF;
    END IF;
  END PROCESS delay_pipeline_process;
```

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **General** tab.

3 Enter a new string in the **Clocked process postfix** field, as shown in the following display.



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

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:

HDL Port	Default Port Name
Input port	filter_in
Output port	filter_out
Clock port	clk
Clock enable port	clk_enable
Reset port	reset

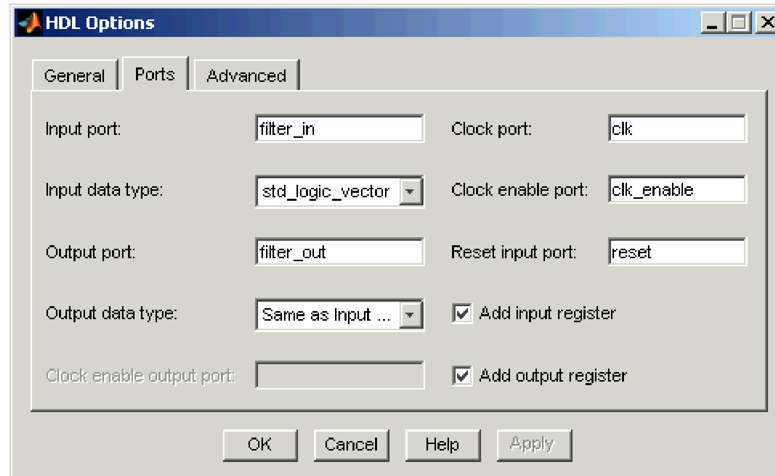
For example, the default VHDL declaration for entity Hd looks like the following.

```
ENTITYHd IS
   PORT( clk              :    IN    std_logic;
         clk_enable       :    IN    std_logic;
         reset            :    IN    std_logic;
         filter_in        :    IN    std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         filter_out       :    OUT   std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         );
ENDHd;
```

To change any of the port names,

1 Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **Ports** tab. Port options appear, as shown in the following display.



3 Enter new strings in the following fields, as necessary:

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

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

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

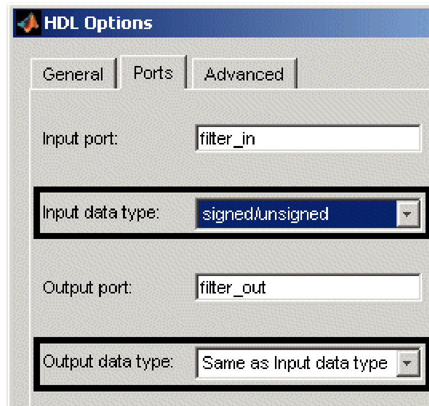
signed/unsigned, and for output data ports, Same as input data type. The Filter Design HDL Coder applies type SIGNED or UNSIGNED based on the data type specified in the filter design.

To change the VHDL data type setting for the input and output data ports,

1. Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2. Select 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 box.

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):

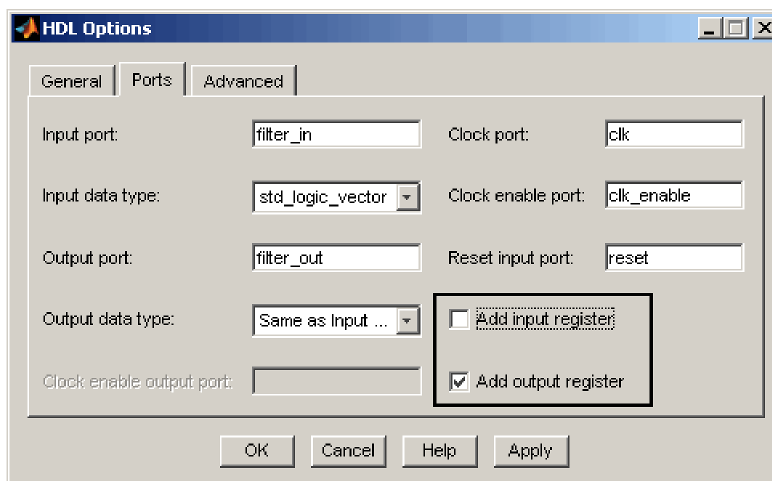
```
Output_Register_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    output_register <= (OTHERS => '0');
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      output_register <= output_typeconvert;
    END IF;
  END IF;
END PROCESS Output_Register_Process;
```

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2. Select 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 box.

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.

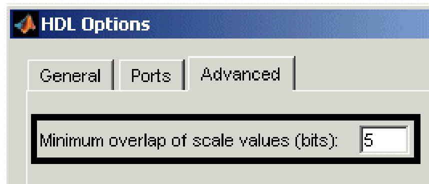
Example	Input Format	Fraction Length	Scale Value	Specified Minimum Overlap (bits)	Binary Representation of Converted Scale Value	Warning Generated?
1	16	15	0.625	3	0.101000000000000	No. <3 leading zeros
2	16	15	0.247	3	0.001111110011101	Yes
3	8	4	2.25	2	0010.0100	Yes
4	8	4	4.125	2	0100.0010	No. <2 leading 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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2. Select the **Advanced** tab. The **Advanced** pane appears.

3 Specify a positive integer in the **Minimum overlap of scale values (bits)** 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 box.

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:

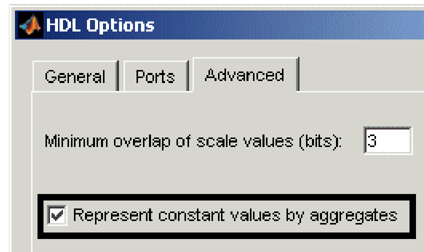
```
CONSTANT coeff1     :signed(15 DOWNTO 0) := to_signed(-30, 16);
CONSTANT coeff2     :signed(15 DOWNTO 0) := to_signed(-89, 16);
```

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **Advanced** tab. The **Advanced** pane appears.

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 box.

The preceding constant declarations would now appear as follows:

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

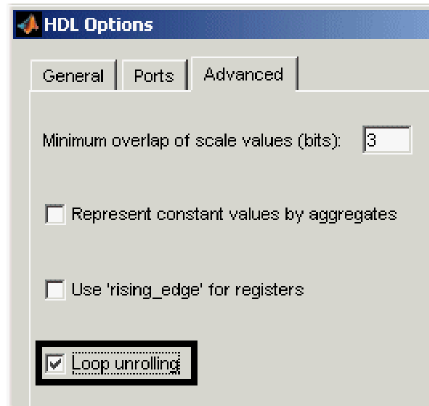
By default, the Filter Design HDL Coder supports VHDL loops. However, some EDA tools do not support them. If you are using such a tool along with VHDL, you might need to unroll and remove `FOR` and `GENERATE` loops from your filter's generated VHDL code. Verilog code is always unrolled.

To unroll and remove `FOR` and `GENERATE` loops,

1 Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **Advanced** tab. The **Advanced** pane appears.

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 box.

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) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
```

```
END PROCESS Delay_Pipeline_Process ;
```

If you prefer, the coder can produce VHDL code that applies the VHDL `rising_edge` function instead. For example, the `ELSIF` statement in the preceding process block would be replaced with the following statement:

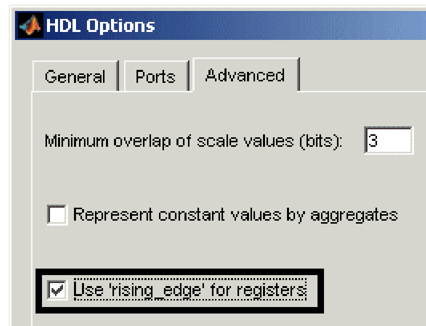
```
ELSIF rising_edge(clk) THEN
```

To use the `rising_edge` function,

1 Click **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2 Select the **Advanced** tab. The **Advanced** pane appears.

3 Select **Use 'rising_edge' for registers**, as shown in the following dialog box.



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

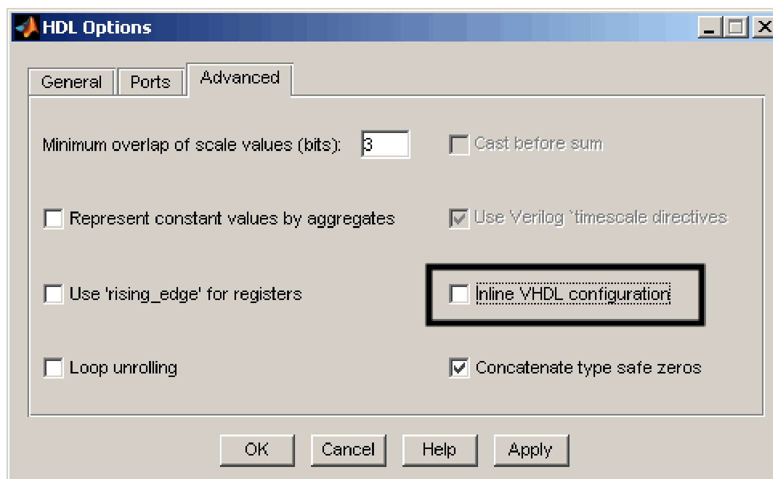
Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the property `UseRisingEdge` to use the VHDL `rising_edge` function to check for rising edges during register operations.

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.
2. Select the **Advanced** tab. The **Advanced** pane appears.
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 box.

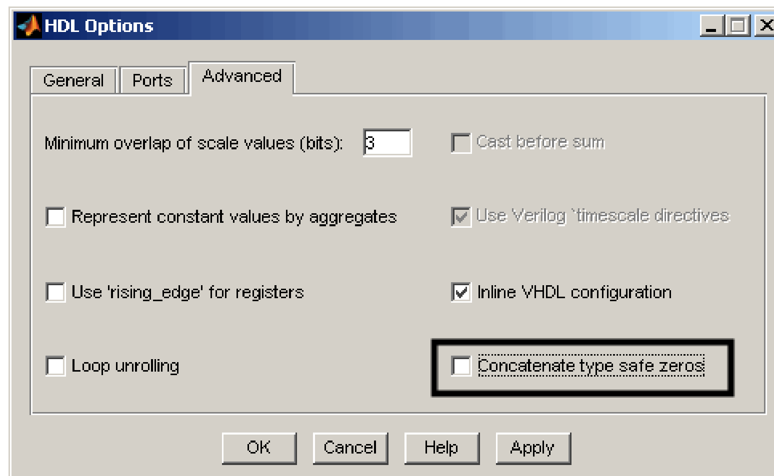
Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the property `InlineConfigurations` to suppress the generation of inline configurations.

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.
2. Select the **Advanced** tab. The **Advanced** pane appears.
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 box.

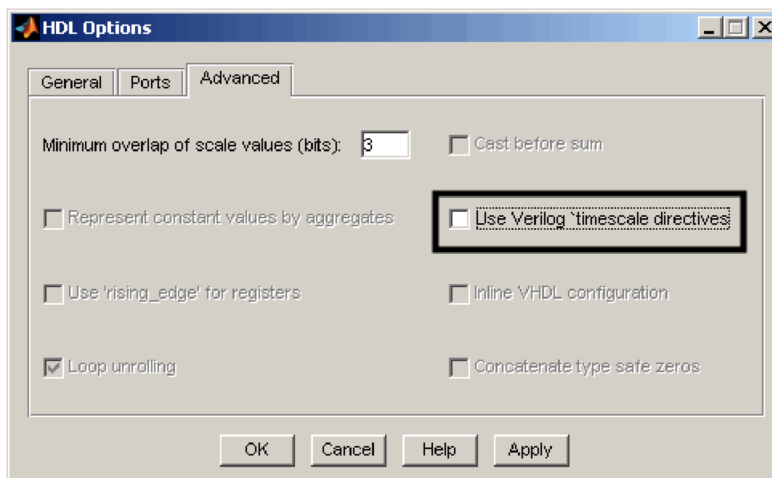
Command Line Alternative: Use the `generatehdl` and `generatetb` functions with the property `SafeZeroConcat` to use the syntax `"000000..."`, for concatenated zeros.

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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.
2. Select the **Advanced** tab. The **Advanced** pane appears.
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 box.

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

Specifying Input Type Treatment for Addition and Subtraction Operations

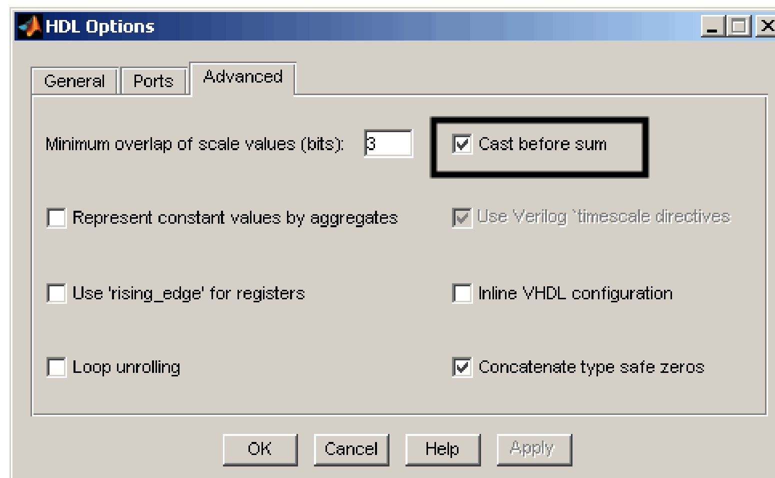
MATLAB and typical DSP processors handle the treatment of input data types for addition and subtraction operations differently. MATLAB operates on input data using the data types as specified and converts the result to the result type. Typical DSP processors, on the other hand, type cast input data to 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 **HDL Options** in the **HDL filter** pane of the Generate HDL dialog box. The HDL Options dialog box appears.

2. Select the **Advanced** tab. The **Advanced** pane appears.

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.**

---

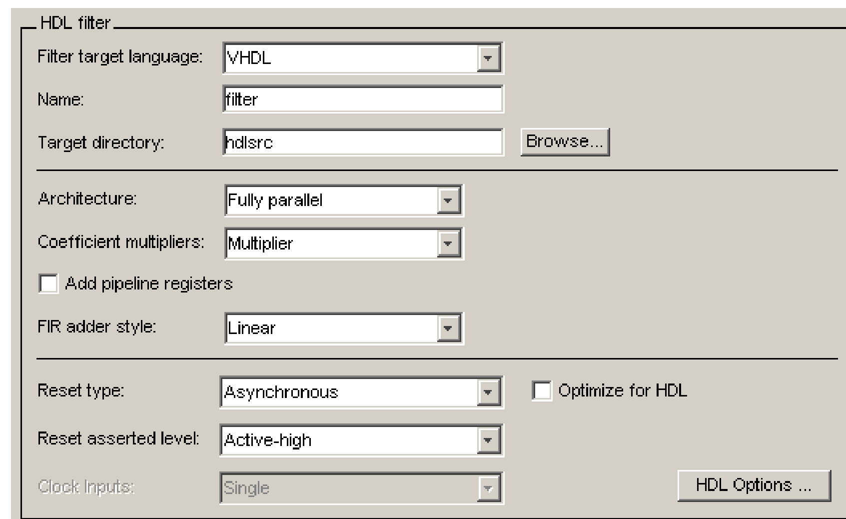
4. Click **Apply** to register the change or **OK** to register the change and close the dialog box.

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 multipliers and the final summation method used for FIR filters.

Code optimization options are listed in the **HDL filter** pane of the Generate HDL dialog box, as shown 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 3-58
- "Optimizing Coefficient Multipliers" on page 3-59
- "Optimizing Final Summation for FIR Filters" on page 3-60

- "Speed vs. Area Optimizations for FIR Filters" on page 3-61
- "Distributed Arithmetic for FIR Filters" on page 3-71
- "Optimizing the Clock Rate with Pipeline Registers" on page 3-81
- "Setting Optimizations for Synthesis" on page 3-83

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. You can choose to generate HDL code that is slightly optimized for clock speed or space requirements. However, note that this optimization causes the Filter Design HDL Coder to

- Make tradeoffs concerning data types
- Avoid extra quantization
- Generate 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 box.
2. Consider setting an error margin for the generated test bench. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,
   a. Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.
   b. Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.
3. Continue setting other options or click **Generate** to initiate code generation.

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

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 The Filter Design HDL Coder does not use coefficient multiplier operations for multirate filters. Therefore, the **Coeff multipliers** options described below are disabled for multirate filters.

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 (for nonmultirate filter types),

1. Select `CSD` or `Factored-CSD` from the **Coeff multipliers** menu in the **HDL filter** pane of the Generate HDL dialog box.

2. Consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,

   a. Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

b. Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.

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

3. Continue setting other options or click **Generate** 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 box:

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

2. If you specify tree or pipelined mode, consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin,

   a. Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

   b. Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.

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

3. Continue setting other options or click **Generate** 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.

Speed vs. Area Optimizations for FIR Filters

Filter Design HDL Coder provides options that extend your control over speed vs. area tradeoffs in the realization of FIR filter designs. To achieve the desired tradeoff, you can either specify a *fully parallel* architecture for generated HDL filter code, or choose one of several *serial* architectures. Supported architectures are described in “Parallel and Serial Architectures” on page 3-63.

The full range of parallel and serial architecture options is supported by properties passed in to the `generatehdl` command, as described in “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 3-64.

Alternatively, you can use the **Architecture** pop-up menu on the HDL Options dialog box to choose parallel and serial architecture options, as described in “Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box” on page 3-67.

The following table summarizes the filter types that are available for parallel and serial architecture choices in Filter Design HDL Coder 1.5.

Architecture	Available for Filter Types...
`Fully parallel` (default)	All filter types that are supported for HDL code generation
`Fully serial`	• `dfilt.dffir` • `dfilt.dfsymfir` • `dfilt.dfasymfir` • `mfilt.firdecim` • `mfilt.firinterp`
`Partly serial`	• `dfilt.dffir` • `dfilt.dfsymfir` • `dfilt.dfasymfir`
`Cascade serial`	• `dfilt.dffir` • `dfilt.dfsymfir` • `dfilt.dfasymfir`

Note Filter Design HDL Coder also supports distributed arithemetic (DA), another highly efficient architecture for realizing FIR filters. See “Distributed Arithmetic for FIR Filters” on page 3-71 for information about how to use this architecture.)

Parallel and Serial Architectures

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

Serial Architectures. Serial architectures reuse hardware resources in time, saving chip area. Filter Design HDL Coder provides a range of serial architecture options, summarized below. All of these architectures have a latency of one clock period (see "Latency in Serial Architectures" on page 3-64).

The available serial architecture options are

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

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

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

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

  When you select a partly serial architecture, you specify the number of partitions and the length (number of taps) of each partition. For example, you could specify a four-tap filter with two partitions, each having two taps. The system clock would run at twice the filter's sample rate.

- *Cascade-serial*: A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are

grouped into a number of serial partitions that execute in parallel with respect to one another. However, the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of all partitions is therefore computed at the accumulator of the first partition. This technique is termed *accumulator reuse*. No final adder is required, which saves area.

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

To generate a cascade-serial architecture, you specify a partly serial architecture with accumulator reuse enabled (see “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 3-64). If you do not specify the serial partitions, Filter Design HDL Coder automatically selects an optimal partitioning.

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

Specifying Speed vs. Area Tradeoffs via generatehdl Properties

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

Two properties are provided to specify serial architecture options when generating code via `generatehdl`:

`'SerialPartition'`: This property specifies the serial partitioning of the filter.

`'ReuseAccum'`: This property enables or disables accumulator reuse.

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

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Fully parallel	Omit this property	Omit this property
Fully serial	`N`, where `N` is the length of the filter	Not specified, or `'off'`
Partly serial	`[p1 p2 p3...pN]` : a vector of integers having `N` elements, where `N` is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter.	`'off'`
Cascade-serial with explicitly specified partitioning	`[p1 p2 p3...pN]`: a vector of integers having `N` elements, where `N` is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter.	`'on'`
Cascade-serial with automatically optimized partitioning	Omit this property	`'on'`

Specifying Parallel and Serial FIR Architectures in generatehdl. The following examples show the use of the `'SerialPartition'` and `'ResuseAccum'` properties in generating code with the `generatehdl` function. All examples assume that a direct-form FIR filter has been created in the MATLAB workspace as follows:

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

In this example, a fully parallel architecture is generated (by default).

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

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

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

```
generatehdl(Hd,'SerialPartition',9, 'Name','FullySerial')
### Starting VHDL code generation process for filter: FullySerial
### Generating: D:\Work\test\hdlsrc\FullySerial.vhd
### Starting generation of FullySerial VHDL entity
### Starting generation of FullySerial VHDL architecture
###   Clock rate will be 9 times the input sample rate for this arch.
###   There are 1 serial sections.
###   Serial section 1 - 9 inputs.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: FullySerial
```

In this example, a partly serial architecture with three partitions is generated.

```
generatehdl(Hd,'SerialPartition',[3 4 2], 'Name', 'PartlySerial')
### Starting VHDL code generation process for filter: PartlySerial
### Generating: D:\Work\test\hdlsrc\PartlySerial.vhd
### Starting generation of PartlySerial VHDL entity
### Starting generation of PartlySerial VHDL architecture
###   Clock rate will be 3 times the input sample rate for this arch.
###   There are 3 serial sections.
###   Serial section 1 - 4 inputs.
###   Serial section 2 - 3 inputs.
###   Serial section 3 - 2 inputs.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: PartlySerial
```

In this example, a cascade-serial architecture with three partitions is generated. Note that the clock rate is higher than that in the previous (partly serial without accumulator reuse) example.

```
generatehdl(Hd,'SerialPartition',[4 3 2], 'ReuseAccum', 'on','Name','CascadeSerial')
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: D:\Work\test\hdlsrc\CascadeSerial.vhd
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
###   Clock rate will be 5 times the input sample rate for this arch.
###   There are 3 serial sections.
###   Serial section 1 - 4 inputs.
###   Serial section 2 - 3 inputs.
###   Serial section 3 - 2 inputs.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: CascadeSerial
```

In this example, a cascade-serial architecture is generated, with the partitioning automatically determined by Filter Design HDL Coder.

```
generatehdl(Hd,'ReuseAccum','on', 'Name','CascadeSerial')
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: D:\Work\test\hdlsrc\CascadeSerial.vhd
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
###   Clock rate will be 5 times the input sample rate for this arch.
###   There are 3 serial sections.
###   Serial section 1 - 4 inputs.
###   Serial section 2 - 3 inputs.
###   Serial section 3 - 2 inputs.
### HDL latency is 3 samples
### Successful completion of VHDL code generation process for filter: CascadeSerial
```

Selecting Parallel and Serial Architectures in the Generate HDL Dialog Box

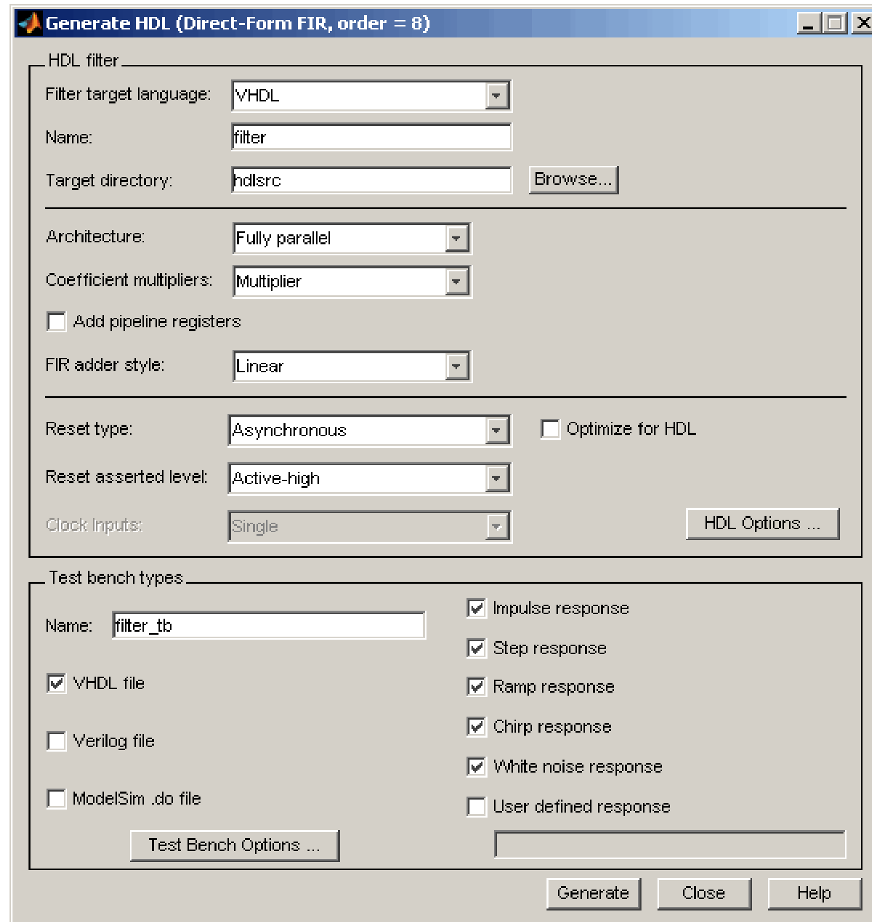
The **Architecture** pop-up menu, located on the **Generate HDL** dialog box, lets you select parallel and serial architecture options corresponding to those described in "Parallel and Serial Architectures" on page 3-63. These options are

- `Fully parallel` (default)

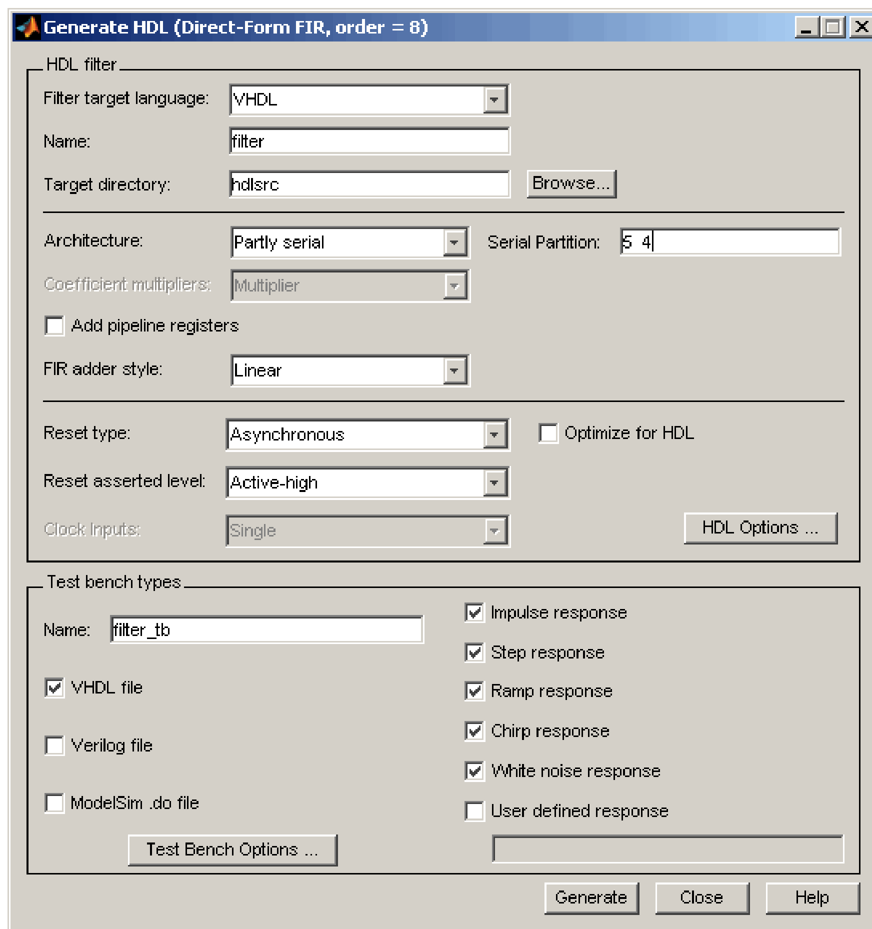
- `Fully serial`: Nonpartitioned serial architecture, without accumulator reuse
- `Partly serial`: Partitioned serial architecture, without accumulator resuse (See "Specifying Partitions for Partly Serial and Cascade Serial Architectures" on page 3-69.)
- `Cascade serial`: Partitioned serial architecture, with accumulator reuse (See "Specifying Partitions for Partly Serial and Cascade Serial Architectures" on page 3-69.)

Note The **Architecture** pop-up menu also includes the `Distributed arithmetic (DA)` option (see "Distributed Arithmetic for FIR Filters" on page 3-71).

The default (`Fully parallel`) setting is shown in the following figure.



Specifying Partitions for Partly Serial and Cascade Serial Architectures. When you select the `Partly serial` or `Cascade serial` option, the Generate HDL dialog box displays the **Serial Partition** field (shown in the following figure).



The **Serial Partition** field lets you enter a vector of integers specifying the number and size of the partitions, as described in "Specifying Speed vs. Area Tradeoffs via generatehdl Properties" on page 3-64.

By default, **Serial Partition** divides the filter into two partitions. For example, the preceding figure shows the default partition (5 4) for a filter with 9 taps.

Interactions Between Architecture Options and Other HDL Options. Selection of some **Architecture** menu options may change or disable other options, as described below.

- When the `Fully serial` option is selected, the following options are set to their default values and disabled:
  - **Coeff multipliers**
  - **Add pipeline registers**
  - **FIR adder style**
- When the `Partly serial` option is selected, the **Coeff multipliers** option is set to its default value and disabled.
- When the `Cascade serial` option is selected, the following options are set to their default values and disabled:
  - **Coeff multipliers**
  - **Add pipeline registers**
  - **FIR adder style**

Distributed Arithmetic for FIR Filters

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

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

Filter Design HDL Coder supports DA in HDL code generated for several single-rate and multirate FIR filter structures (see “Requirements and Considerations for Generating Distributed Arithmetic Code” on page 3-73). Only fixed-point filter designs are supported.

Distributed Arithmetic Overview

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

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

In a DA realization of a FIR filter structure, a sequence of input data words of width `W` is fed through a parallel to serial shift register, producing a serialized stream of bits. The serialized data is then fed to a bit-wide shift register. This shift register serves as a delay line, storing the bit serial data samples.

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

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

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

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

To compute more than one bit sum at a time, the LUT is replicated. For example, to perform DA on 2 bits at a time (radix 4), the odd bits are fed to one LUT and the even bits are simultaneously fed to an identical LUT. The LUT results corresponding to odd bits are left-shifted before they are added

to the LUT results corresponding to even bits. This result is then fed into a scaling accumulator that shifts its feedback value by 2 places.

Processing more than one bit at a time introduces a degree of parallelism into the operation, improving performance at the expense of area. The `DARadix` property lets you specify the number of bits processed simultaneously in DA (see “DARadix Property” on page 3-77).

Reducing LUT Size. The size of the LUT grows exponentially with the order of the filter. For a filter with `N` coefficients, the LUT must have `2^N` values. For higher order filters, LUT size must be reduced to reasonable levels. To reduce the size, you can subdivide the LUT into a number of LUTs, called *LUT partitions*. Each LUT partition operates on a different set of taps. The results obtained from the partitions are summed.

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

Although LUT partitioning reduces LUT size, more adders are required to sum the LUT data.

The `DALUTPartition` property lets you specify how the LUT is partitioned in DA (see “DALUTPartition Property” on page 3-74).

Requirements and Considerations for Generating Distributed Arithmetic Code

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

Supported Filter Types. Filter Design HDL Coder supports DA in HDL code generated for the following single-rate and multirate FIR filter structures:

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

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

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

- "DALUTPartition Property" on page 3-74
- "DARadix Property" on page 3-77

Fixed Point Quantization Required. Generation of DA code is supported only for fixed-point filter designs. If you are designing your filter in FDATool, select `Fixed-point` from the **Filter arithmetic** list in the Quantization Parameters pane. If you are creating a filter object in MATLAB, set the `arithmetic` property of your filter object to `'fixed'`.

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

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

DALUTPartition Property

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

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

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

- `N` is the number of partitions.

- Each vector element specifies the size of a partition. The maximum size for an individual partition is 12.
- The sum of all vector elements equals the filter length FL. FL is calculated differently depending on the filter type (see "Specifying DALUTPartition for Single-Rate Filters" on page 3-75 and "Specifying DALUTPartition for Multirate Filters" on page 3-76).

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

```
b = [0.0349 0.4302 0.4302 0.4302 0.0349];
Hd = dfilt.dffir(b);
Hd.arithmetic = 'fixed';
generatehdl (Hd, 'DALUTPartition', 5);
```

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

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

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

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

FL =
```

```
    31
generatehdl(Hd, 'DALUTPartition',[8 8 8 7]);
```

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

```
Hd = design(filtdes,'filterstructure','dfsymfir');
Hd.arithmetic = 'fixed';
FL = ceil(length(find(Hd.numerator~= 0))/2)

FL =

    16
generatehdl(Hd, 'DALUTPartition',[8 8]);
```

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

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

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

LUT Partition Specified As...	Filter Length (FL) Calculation
Vector: determine FL as shown in the **Filter Length (FL) Calculation** column to the right. Specify the LUT partition as a vector of integers whose elements sum to FL.	`FL = size(polyphase(Hm), 2)`
Matrix: determine the subfilter length FL*i* based on the polyphase decomposition of the filter, as shown in the **Filter Length (FL) Calculation** column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to FL*i*.	`p = polyphase(Hm);` `FLi = length(find(p(i,:)));` where *i* is the index to the *i*th row of the polyphase matrix of the multirate filter. The *i*th row of the matrix p represents the*i*th subfilter.

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

```
Hm = mfilt.firdecim(4);
Hm.arithmetic = 'fixed';
FL = size(polyphase(Hm),2)

FL =

    24

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

The following example shows the LUT partition specified as a maxtrix for the same direct-form FIR polyphase decimator:

```
Hm = mfilt.firdecim(4);
Hm.arithmetic = 'fixed';
generatehdl(Hm, 'DALUTPartition',[1 0 0 0; 7 7 7 3; 8 8 6 2; 8 8 8 0]);
```

DARadix Property

Syntax: 'DARadix', N

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

- A nonzero positive integer that is a power of two
- Such that `mod(W, log2(N)) = 0` where `W` is the input word size of the filter.

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

Note When setting a `DARadix` value for symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters, see "Considerations for Symmetrical and Asymmetrical Filters" on page 3-78.

Special Cases

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

Considerations for Symmetrical and Asymmetrical Filters. For symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters:

- A bit-level preadder or presubtractor is required to add tap data values that have coefficients of equal value and opposite sign. One extra clock cycle is required to compute the result because of the additional carry bit.
- Filter Design HDL Coder takes advantage of filter symmetry where possible. This reduces the DA LUT size substantially , because the effective filter length for these filter types is halved.
- If a `DARadix` value greater than 2 is passed in for these filter types, a warning is displayed and the symmetry or asymmetry in the filter structure is ignored in HDL code generation.

Distributed Arithmetic Options in the Generate HDL Dialog Box

This section describes Generate HDL dialog box options related to DA code generation. The following figure shows these options.

Architecture: Distributed arithmetic (DA)
LUT Partition: 8 8 8 8 8 8 3
Coefficient multipliers: Multiplier
DA Radix: 2
Add pipeline registers
FIR adder style: Linear

The DA related options are:

- The **Architecture** pop-up menu, which lets you enable DA code generation and displays related options
- The **LUT Partition** field, which displays and lets you change the value of the `DALUTPartition` property
- The **DA Radix** field, which displays and lets you change the value of the `DARadix` property

The Generate HDL dialog box initially displays default DA related option values that are appropriate for the current filter design. In other respects, the requirements for setting these options are identical to those described in "DALUTPartition Property" on page 3-74 and "DARadix Property" on page 3-77.

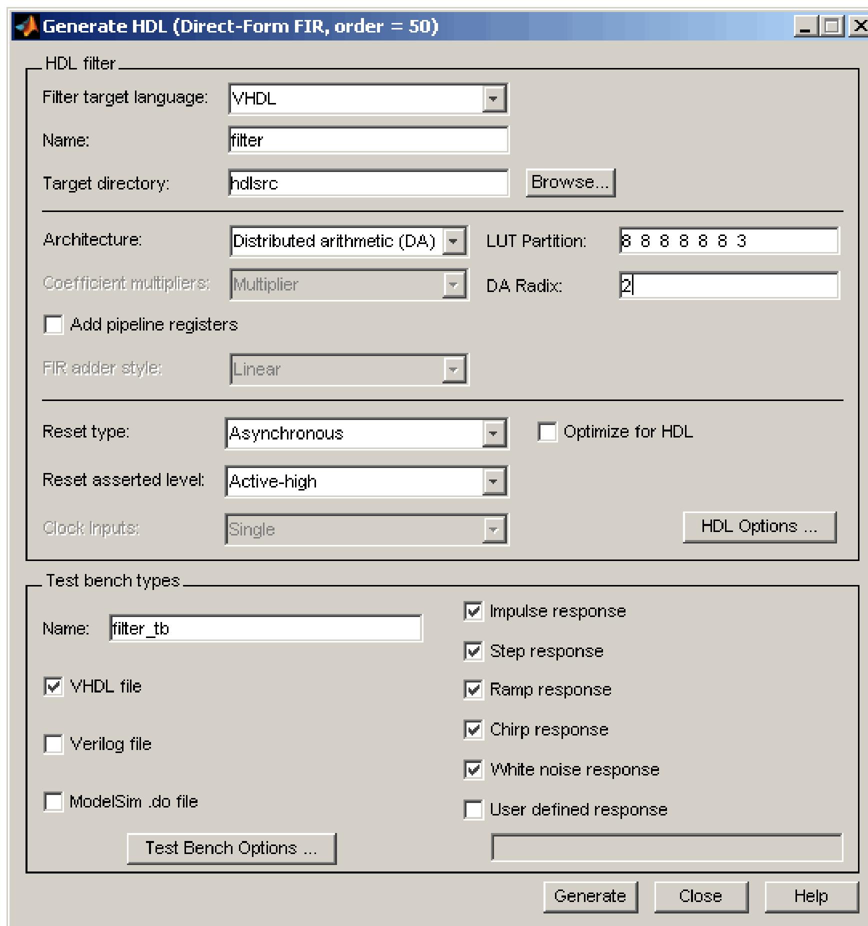
To specify DA code generation using the Generate HDL dialog box, follow these steps:

1. In FDATool, design a FIR filter that meets the requirements described in "Requirements and Considerations for Generating Distributed Arithmetic Code" on page 3-73.

2. Open the Generate HDL dialog box by selecting **Targets > Generate HDL** from the FDATool menu.

3 Select Distributed Arithmetic (DA) from the **Architecture** pop-up menu.

When you select this option, the related **LUT Partition** and **DA Radix** options are displayed to the right of the **Architecture** menu. The following figure shows the default DA options for a Direct Form FIR filter.



The default value for **LUT Partition** is a vector of integers such that each partition has a maximum of 8 inputs. The figure illustrates a 51-tap filter,

with 7 partitions. All partitions have 8 inputs except for the last, which has 3 inputs.

4 If desired, set the **LUT Partition** field to a nondefault value. See "DALUTPartition Property" on page 3-74 for detailed information.

5 The default **DA Radix** value is `2`, specifying processing of one bit at a time, or fully serial DA. If desired, set the **DA Radix** field to a nondefault value. See "DARadix Property" on page 3-77 for detailed information.

   If you are setting the **DA Radix** value for a `dfilt.dfsymfir` and `dfilt.dfasymfir` filter, see "Considerations for Symmetrical and Asymmetrical Filters" on page 3-78.

6 Set other HDL options as required, and generate code. Incorrect or illegal values for **LUT Partition** or **DA Radix** are reported at code generation time.

DA Interactions with Other HDL Options. When `Distributed Arithmetic (DA)` is selected in the **Architecture** menu, some other HDL options change automatically to settings that are appropriate for DA code generation:

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

Optimizing the Clock Rate with Pipeline Registers

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

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

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

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see "Optimizing Final Summation for FIR Filters" on page 3-60.

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 box.

2 For FIR, antisymmetric FIR, and symmetric FIR filters, consider setting an error margin for the generated test bench to account for numeric differences. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin:

   a Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

   b Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numerical results can differ before the coder issues a warning.

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

3. Continue setting other options or click **Generate** 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.

To...	Select...	Which...	At the Cost of...
Slightly increase the clock speed and slightly decrease the area used	**Optimize for HDL**	Removes extra quantization operations	Not remaining bit-true to MATLAB.
Increase the clock speed while maintaining the area used	`Tree` for **FIR adder style**	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	**Add pipeline registers**	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`.

To...	Select...	Which...	At the Cost of...
Decrease the area used while maintaining or increasing clock speed (depends on binary representation of coefficients)	`CSD` for **Coefficient multipliers**	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 **Coefficient multipliers**	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.

Generating Code for Multirate Filters

Supported Multirate Filter Types

The Filter Design HDL Coder supports code generation for several types of multirate filters:

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

Generating Mutirate Filter Code

To generate multirate filter code, you must first select and design one of the supported filter types in the multirate design panel of FDATool. (See "Designing Multirate Filters in FDATool" in the Filter Design Toolbox documentation for information about multirate filter design.)

After you have created the filter, open the Generate HDL dialog box, set the desired code generation properties, and generate code. GUI options that support multirate filter code generation are described in "Code Generation Options for Multirate Filters" on page 3-85.

If you prefer to generate code via the `generatehdl`function, the Filter Design HDL Coder also defines multirate filter code generation properties that are functionally equivalent to the GUI options. These properties are summarized in "generatehdl Properties for Multirate Filters" on page 3-91.

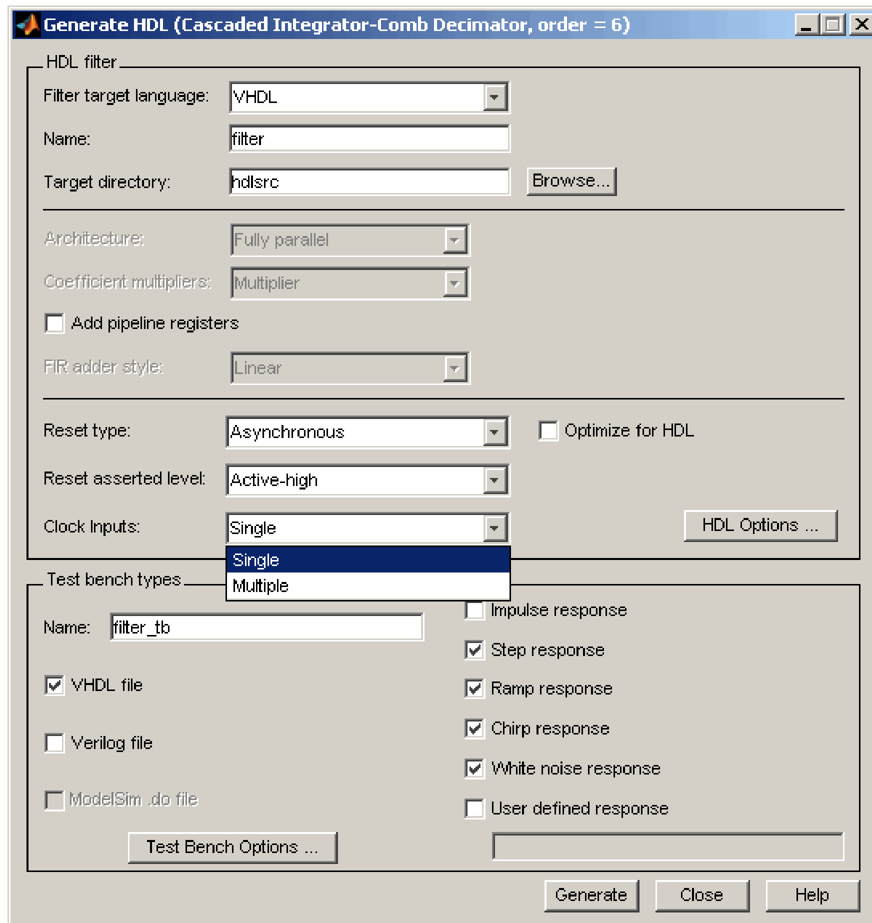
Code Generation Options for Multirate Filters

When a multirate filter of a supported type (see "Supported Multirate Filter Types" on page 3-85) is designed in `fdatool`, the enabled/disabled state of several options in the Generate HDL dialog box changes:

- The **Clock inputs** pull-down menu is enabled. This menu provides two alternatives for generating clock inputs for multirate filters, as discussed below.
- The **ModelSim .do file** option is disabled. Generation of ModelSim `.do` test bench files is not supported for multirate filters.
- For CIC filters, the **Coefficient multipliers** option is disabled. Coefficient multipliers are not used in CIC filters.

  However, the **Coefficient multipliers** option is enabled for Direct-Form Transposed FIR Polyphase Decimator (`mfilt.firtdecim`) filters.
- For CIC filters, the **FIR adder style** option is disabled, since CIC filters do not require a final adder.

The following figure shows the default settings of the Generate HDL dialog box options when a supported CIC filter has been designed in `fdatool`.



The **Clock inputs** options are

- `Single`: When `Single` is selected, the `ENTITY` declaration for the filter defines a single clock input with an associated clock enable input and clock enable output. The generated code maintains a counter that controls the timing of data transfers to the filter output (for decimation filters) or input

(for interpolation filters). The counter is, in effect, a secondary clock enable whose rate is determined by the filter's decimation or interpolation factor.

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

A clock enable output is also generated when Single is selected. If you want to customize the name of this output in generated code, see "Setting the Clock Enable Output Name" on page 3-90.

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

The ENTITY declaration is as follows.

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

END cic_decim_4_1_single;
```

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

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

```
  END PROCESS ce_output;

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

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

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

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

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

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

  Note that no synchronizers between multiple clock domains are provided.

  When `Multiple` is selected, clock enable outputs are not generated; therefore the **Clock enable output port** field of the HDL Options dialog box is disabled.

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

```
ENTITY cic_decim_4_1_multi IS
```

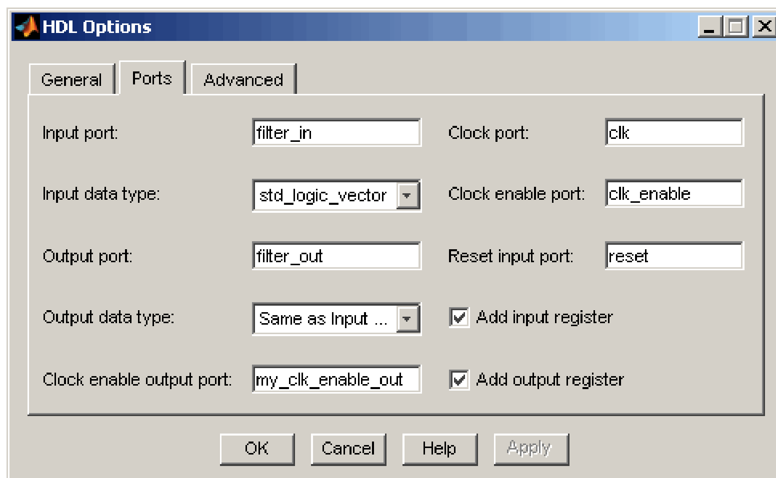
```
    PORT( clk             :   IN    std_logic;
          clk_enable      :   IN    std_logic;
          reset           :   IN    std_logic;
          filter_in       :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
          clk1            :   IN    std_logic;
          clk_enable1     :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

END cic_decim_4_1_multi;
```

Setting the Clock Enable Output Name

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

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



Note that the **Clock enable output port** field is disabled when multiple clocks are being generated.

Generating Test Bench Code for Multirate Filters

You can generate VHDL or Verilog test bench files for multirate filters. Generation of ModelSim `.do` test bench files is not supported for multirate filters, and the **ModelSim .do file** option of the Generate HDL dialog box is disabled.

generatehdl Properties for Multirate Filters

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

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

Generating Code for Cascade Filters

Supported Cascade Filter Types

The Filter Design HDL Coder supports code generation for the following types of cascade filters:

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

Generating Cascade Filter Code

To generate cascade filter code,

1. Instantiate the filter stages and cascade them in the MATLAB workspace (see the Filter Design Toolbox documentation for the `mfilt.cascade` and `mfilt.cascade` filter objects).

   The Filter Design HDL Coder currently imposes certain limitations on the filter types allowed in a cascade filter. See "Rules and Limitations for Code Generation with Cascade Filters" on page 3-92 before creating your filter stages and cascade filter object.

2. Import the cascade filter object into FDATool, as described in "Importing and Exporting Quantized Filters" in the Filter Design Toolbox documentation.

3. After you have imported the filter, open the Generate HDL dialog box, set the desired code generation properties, and generate code. See "Rules and Limitations for Code Generation with Cascade Filters" on page 3-92

4. Note that the Filter Design HDL Coder generates separate HDL code files for each stage of the cascade, in addition to the top-level code for the cascade filter itself. The filter stage code files are identified by appending the string `_stage1`, `_stage2`, ... `_stage`*N* to the filter name.

Rules and Limitations for Code Generation with Cascade Filters

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

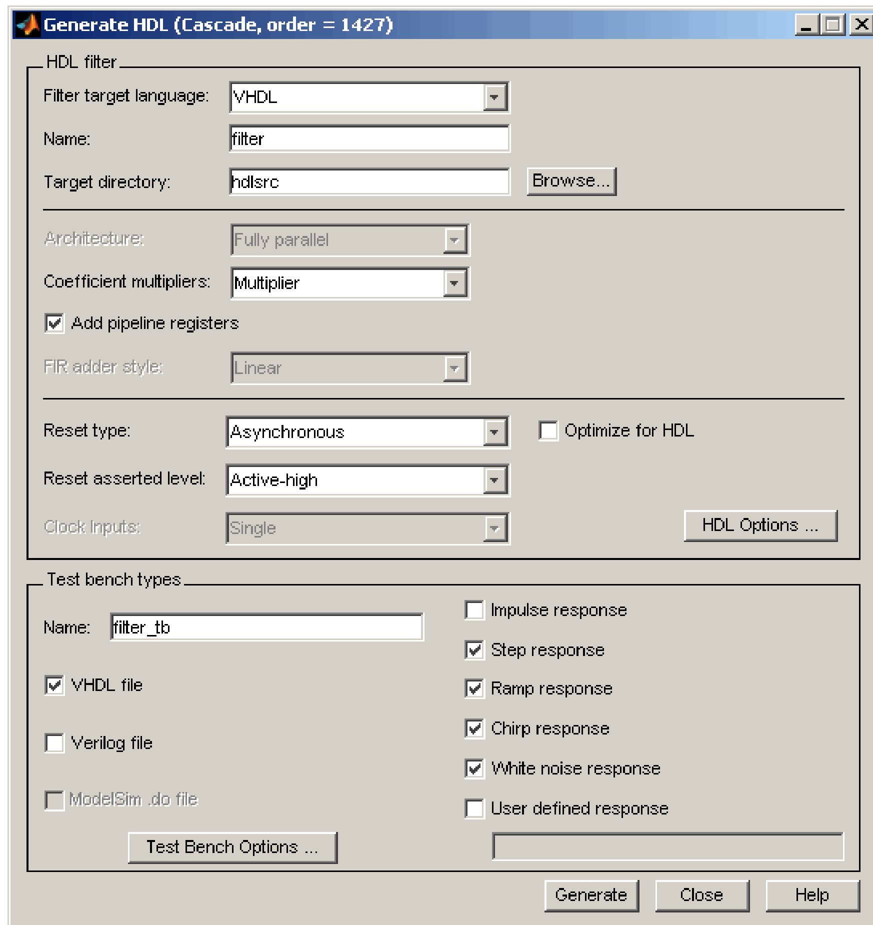
- You can generate code for cascades that combine the following filter types:
  - Decimators and/or single-rate filter structures
  - Interpolators and/or single-rate filter structures

  Code generation for cascades that include both decimators and interpolators is not currently supported, however. If unsupported filter structures or combinations of filter structures are included in the cascade, code generation is disallowed.
- For code generation, only a flat (single-level) cascade structure is allowed. Nesting of cascade filters is disallowed.
- By default, all input /output registers are removed from the stages of the cascade in generated code, except for the input of the first stage and the output of the final stage. However, if the **Add pipeline registers** option in Generate HDL dialog box is selected, the output registers for each stage are generated, and internal pipeline registers may be added, depending on the filter structures.

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

- When a cascade filter is created in `fdatool`, the enabled/disabled state of several options in the Generate HDL dialog box changes:
  - The **ModelSim .do file** option is disabled. Generation of ModelSim `.do` test bench files is not supported for multirate filters.
  - The **FIR adder style** option is disabled. If you require tree adders for FIR filters in a cascade, select the **Add pipeline registers** option (since pipelines require tree style FIR adders).

The following figure shows the default settings of the Generate HDL dialog box options when a cascade filter has been designed in `fdatool`.



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 3-95
- "Specifying a Test Bench Type" on page 3-97
- "Configuring the Clock" on page 3-99
- "Configuring Resets" on page 3-101
- "Setting a Hold Time for Data Input Signals" on page 3-103
- "Setting an Error Margin for Optimized Filter Code" on page 3-104
- "Setting Test Bench Stimuli" on page 3-106

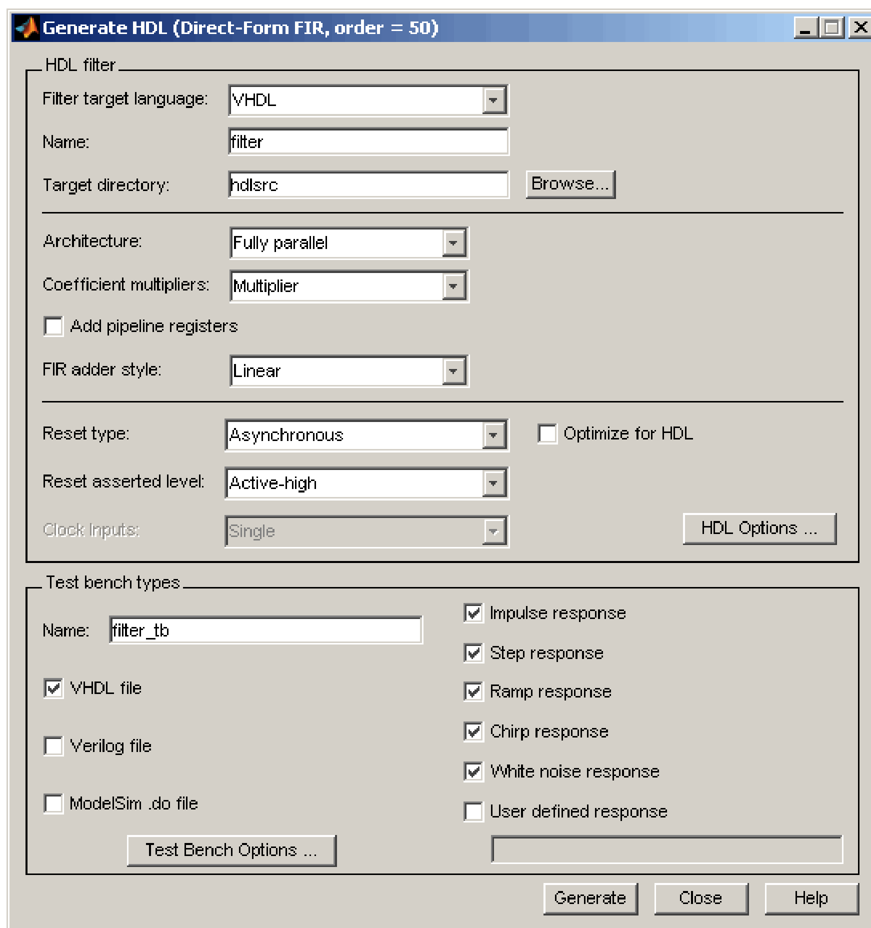
Renaming the Test Bench

As discussed in "Customizing Reset Specifications" on page 3-29, 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.

If the Test Bench Is a...	The Extension Is...
Verilog file	Defined by the **Verilog file extension** field in the **General** pane of the HDL Options dialog box
VHDL file	Defined by the **VHDL file extension** field in the **General** pane of the HDL Options dialog box
ModelSim `.do` file	`.do`

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

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



Note If you enter a string that is a VHDL or Verilog reserved word, the coder appends the reserved word postfix to the string to form a valid identifier.

Command Line Alternative: Use the `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
- A ModelSim `.do` file to be used for simulation in the ModelSim environment

---

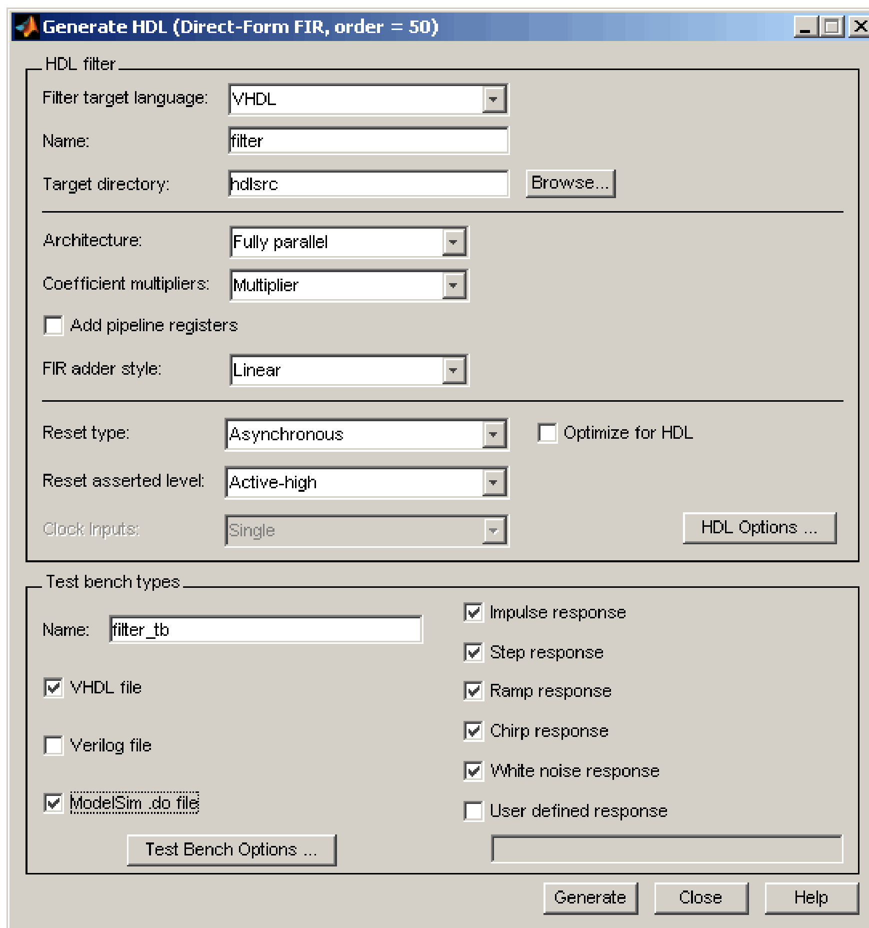
Note Due to differences in representation of double-precision data in VHDL and Verilog, restrictions apply to the types of test benches that are interoperable. The following table shows valid and invalid test bench type and HDL combinations when code is generated for a double-precision filter.

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

*Errors may be reported due to string comparisons.

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

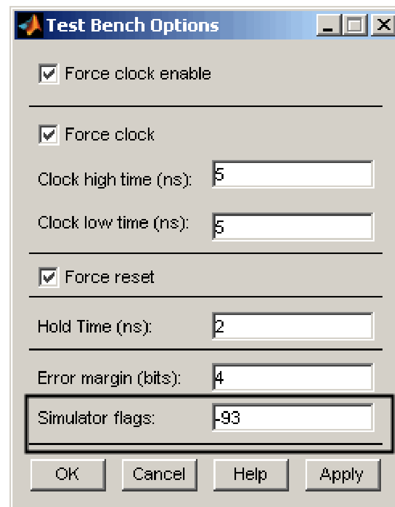
By default, the coder produces a VHDL or Verilog file only, depending on your language selection. If you want to generate additional test bench files, select the desired test bench types listed in the **Test bench types** pane of the Generate HDL dialog box. In the following figure, the dialog box specifies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

2 Type the flags of interest in the **Simulator flags** field. In the following figure, the dialog box specifies that ModelSim use the `-93` compiler option for compilation.



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

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.

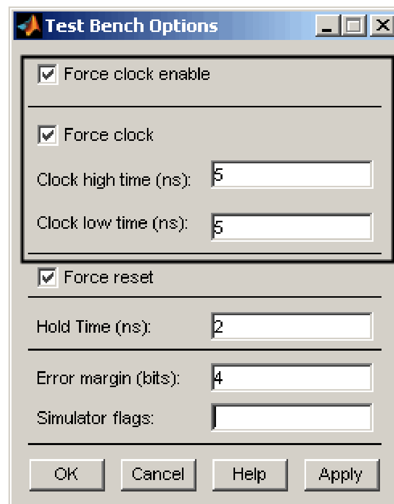
To change these clock configuration settings:

1 Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

2 Make the following configuration changes as needed:

If You Want to...	Then...
Disable the forcing of clock enable input signals	Clear **Force clock enable**.
Disable the forcing of clock input signals	Clear **Force clock**.
Reset the number of nanoseconds during which clock input signals are to be driven low (0)	Specify a positive integer in the **Clock low time** field.
Reset the number of nanoseconds during which clock input signals are to be driven high (1)	Specify a positive integer in the **Clock high time** field.

The following figure highlights the applicable options.



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

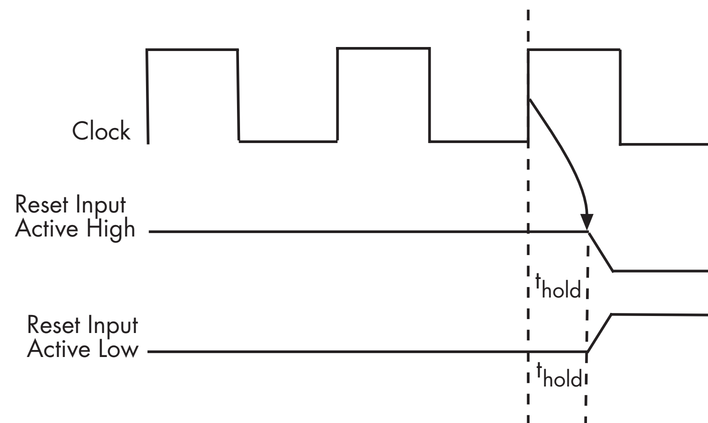
Command Line Alternative: Use the `generatetb` function with the properties `ForceClockEnable`, `ForceClock`, `ClockHighTime`, and `ClockLowTime` to reconfigure the test bench clock.

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). (Test bench reset input levels are set by the **Reset asserted level** option).
- 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_{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.

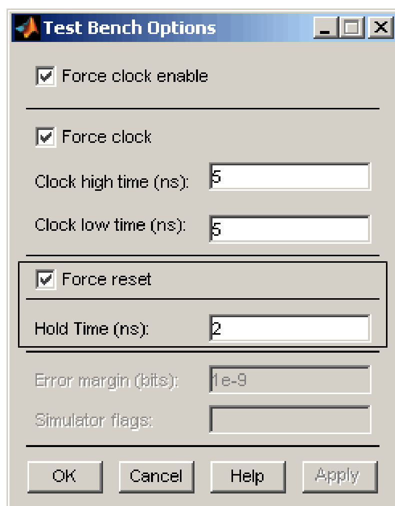
To change the default reset configuration settings,

1 Click **Test Bench Options** in the **Test bench types** pane in the Generate HDL dialog box. The Test Bench Options dialog box appears.

2 Make the following configuration changes as needed:

If You Want to...	Then...
Disable the forcing of reset input signals	Clear **Force reset**.
Change the reset value to active low (0)	Select `Active-low` from the **Reset asserted level** menu in the Generate HDL dialog box (see "Setting the Asserted Level for the Reset Input Signal" on page 3-30)
Reset the hold time	Specify a positive integer, representing nanoseconds, in the **Hold time** field.

The following figure highlights the applicable options.



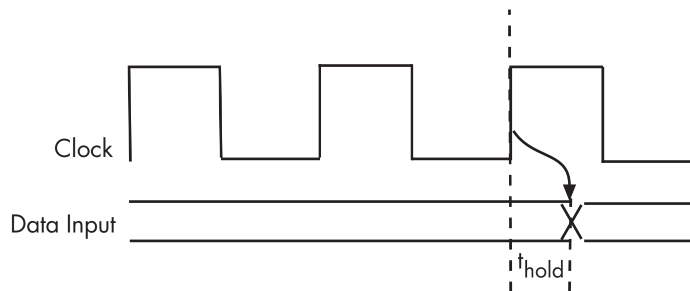
3 Click **Apply** to register the change or **OK** to register the change and close the dialog box.

Note The hold time setting also applies to data input signals.

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

Setting a Hold Time for Data Input Signals

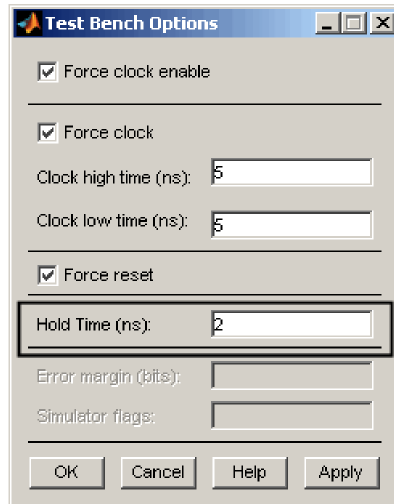
By default, the Filter Design HDL Coder applies a hold time of 2 nanoseconds for filter data input signals. The hold time is the amount of time that data input signals are to be held past the clock rising edge. The following figure shows the application of a hold time (t_{hold}) for data input signals.



To change the hold time setting,

1. Click **Test Bench Options** in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

2 Specify a positive integer, representing nanoseconds, in the **Hold time** field. In the following figure, the hold time is set to 3 nanoseconds.



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

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

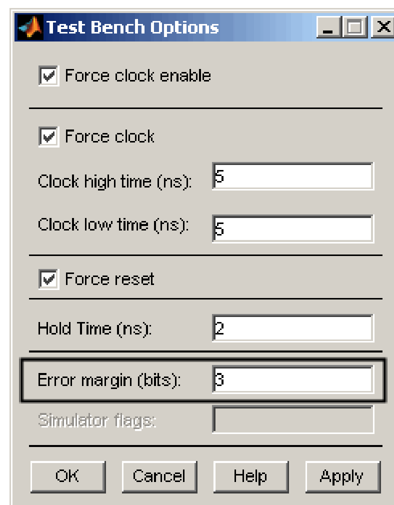
If you choose to use any of these options, consider setting an error margin for the generated test bench to account for differences in numeric results. The error margin is the number of least significant bits the test bench will ignore when comparing the results. To set an error margin:

1. Click Test Bench Options in the **Test bench types** pane of the Generate HDL dialog box. The Test Bench Options dialog box appears.

2. For fixed-point filters, the **Error margin (bits)** field is initialized to a default value of `4` when it is first enabled.

   For double-precision floating-point filters, the error margin value is fixed at `1e-9`. This value cannot be changed. The **Error margin (bits)** field displays a read-only (disabled) value.

3. Specify an integer in the **Error margin (bits)** field that indicates an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning. In the following figure, the error margin is set to `3` bits.



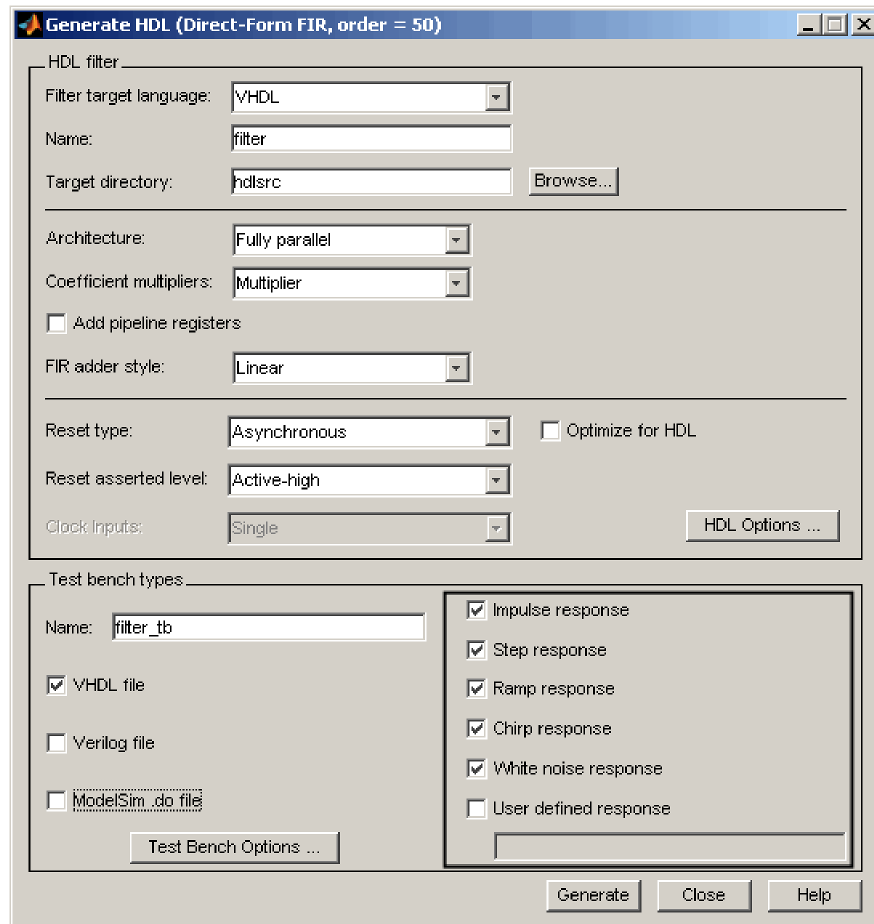
4. Click **Apply** to register the change or **OK** to register the change and close the dialog box.

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.

For Filters...	Default Response Types Include...
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, select one or more response types listed in the **Test bench types** pane of the Generate HDL dialog box. The following figure highlights this pane of the dialog box.



If you select **User defined response**, you must also specify 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 **Generate** on the Generate HDL dialog box. 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
## HDL latency is 2 samples
### 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.

Generating Scripts for EDA Tools

The Filter Design HDL Coder supports generation of script files for third-party Electronic Design Automation (EDA) tools. These scripts let you compile and simulate generated HDL code and/or synthesize generated HDL code.

Using the defaults, you can automatically generate scripts for the following tools:

- Mentor Graphics ModelSim SE/PE HDL simulator
- The Synplify family of synthesis tools

You can customize both the names and the content of generated script files. To do this, you must use the `generatehdl` or `generatetb` function, and pass in the appropriate property name/property value arguments as described in "Customizing Script Names" on page 3-111 and "Customizing Script Code" on page 3-111.

Enabling and Disabling Script Generation

By default, script generation takes place automatically, as part of the code and test bench generation process (whether initiated from the command line or from the Generate HDL dialog box).

The `EDAScriptGeneration` property controls the generation of script files. By default, `EDAScriptGeneration` is set `'on'`. To disable script generation, set `EDAScriptGeneration` to `'off'`, as in the following example.

```
generatehdl(Hd,'EDAScriptGeneration','off')
```

Default Script Generation

All script files are generated in the target directory.

When HDL code is generated for a filter *Hd*, the Filter Design HDL Coder writes the following script files:

- `Hd_compile.do`: ModelSim compilation script. This script contains commands to compile the generated filter code, but not to simulate it.

- *Hd*_synplify.tcl: Synplify synthesis script

When test bench code is generated for a filter *Hd*, the Filter Design HDL Coder writes the following script files:

- *Hd*_tb_compile.do: ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
- *Hd*_tb_sim.do: ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.

Customizing Script Names

When HDL code is generated, script names are generated by appending a postfix string to the filter name *Hd*.

When test bench code is generated, script names are generated by appending a postfix string to the test bench name *testbench_tb*.

The postfix string depends on the type of script (compilation, simulation, or synthesis) being generated. The default postfix strings are shown in the following table. For each type of script, you can define your own postfix using the associated property.

Script type	Property	Default Value
Compilation	'HDLCompileFilePostfix'	'_compile.do'
Simulation	'HDLSimFilePostfix'	'_sim.do'
Synthesis	'HDLSynthFilePostfix'	'_synplify.tcl'

In the following example, VHDL code is generated for the filter object myfilt. A custom postfix string is specified for the compilation script. The name of the generated compilation script will be myfilt_test_compilation.do.

```
generatehdl(myfilt, 'HDLCompileFilePostfix', '_test_compilation.do')
```

Customizing Script Code

A generated EDA script consists of three sections, which are generated and executed in the following order:

1. An initialization (`Init`) phase. The `Init` phase performs any required setup actions, such as creating a design library or a project file. Some arguments to the `Init` phase are implicit, for example, the top-level entity or module name.

   Properties that apply to the `Init` phase are identified by the substring `Init` in the property name.

2. A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file or once per signal. On each call, a different file or signal name is passed in.

   Properties that apply to the `Cmd` phase are identified by the substring `Cmd` in the property name.

3. A termination phase (`Term`). This is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase takes no arguments.

   Properties that apply to the `Term` phase are identified by the substring `Term` in the property name.

`generatehdl` and `generatetb` generate scripts by passing format strings to the MATLAB `fprintf` function. Using the property name/property value pairs summarized in the following table, you can pass in customized format strings to `generatehdl` or `generatetb`.

You can use any legal `fprintf` formatting characters. For example, `'\n'`inserts a newline into the script file.

Some of these format strings can take arguments, such as the top-level entity or module name, or the names of the VHDL or Verilog files in the design. The `'HDLSimViewWaveCommand'` format string takes the top-level signal names as its argument.

Property Name and Default	Description
Name: `'HDLCompileInit'` Default:`'vlib work\n'`	Format string passed to `fprintf` to write the `Init` section of the compilation script.

Property Name and Default	Description
Name: 'HDLCompileVHDLCmd' Default: 'vcom %s %s\n'	Format string passed to fprintf to write the Cmd section of the compilation script for VHDL files. The two arguments are the contents of the 'SimulatorFlags' property and the filename of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name: 'HDLCompileVerilogCmd' Default: 'vlog %s %s\n'	Format string passed to fprintf to write the Cmd section of the compilation script for Verilog files. The two arguments are the contents of the 'SimulatorFlags' property and the filename of the current entity or module. To omit the flags, set 'SimulatorFlags' to '' (the default).
Name:'HDLCompileTerm' Default:''	Format string passed to fprintf to write the termination portion of the compilation script.
Name: 'HDLSimInit' Default: ['onbreak resume\n',... 'onerror resume\n']	Format string passed to fprintf to write the initialization section of the simulation script.
Name: 'HDLSimCmd' Default: 'vsim work.%s\n'	Format string passed to fprintf to write the simulation command. The implicit argument is the top-level module or entity name.
Name: 'HDLSimViewWaveCmd' Default: 'add wave sim:%s\n'	Format string passed to fprintf to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.
Name: 'HDLSimTerm' Default: 'run -all\n'	Format string passed to fprintf to write the Term portion of the simulation script
Name: 'HDLSynthInit' Default: 'project -new %s.prj\n'	Format string passed to fprintf to write the the Init section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name.

Property Name and Default	Description
Name: 'HDLSynthCmd' Default: 'add_file %s\n'	Format string passed to fprintf to write the Cmd section of the synthesis script. The argument is the filename of the entity or module.
Name: 'HDLSynthTerm' Default: ['set_option -technology VIRTEX2\n',... 'set_option -part XC2V500\n',... 'set_option -synthesis_onoff_pragma 0\n',... 'set_option -frequency auto\n',... 'project -run synthesis\n']	Format string passed to fprintf to write the Term section of the synthesis script.

Example

The following example specifies a ModelSim command for the Init phase of a compilation script for VHDL code generated from the filter myfilt.

```
generatehdl(myfilt, 'HDLCompileInit', 'vlib mydesignlib\n')
```

The following code shows the resultant script, myfilt_compile.do.

```
vlib mydesignlib
vcom  myfilt.vhd
```

Mixed-Language Scripts

The Filter Design HDL Coder allows most combinations of filter and test bench languages. For example, it is possible to generate VHDL filter code and a Verilog test bench file, as in the following commands:

```
generatehdl(myfilt, 'TargetLanguage', 'VHDL')
generatetb(myfilt, 'Verilog')
```

The following listing shows the generated test bench compilation script for the above case (myfilt_tb_compile.do). The script contains the correct language-specific compile command for the generated filter and test bench code.

```
vlib work
vcom  myfilt.vhd
vlog  myfilt_tb.v
```

Note that there are two simulation compile `Cmd` properties (`'HDLCompileVHDLCmd'`, `'HDLCompileVerilogCmd'`) allowing you to customize the compilation command for each supported target language.

Note that you can specify generation of *both* VHDL and Verilog test bench code, via the Generate HDL dialog box. In this case, the test bench compilation script will default to the Verilog compilation command.

Note that the generation of ModelSim `.do` test bench files, which is controlled by the **ModelSim .do file** option, is independent from the generation of script files.

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 3-95, 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 when code is generated for a double-precision filter.

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

*Errors may be reported due to string comparisons.

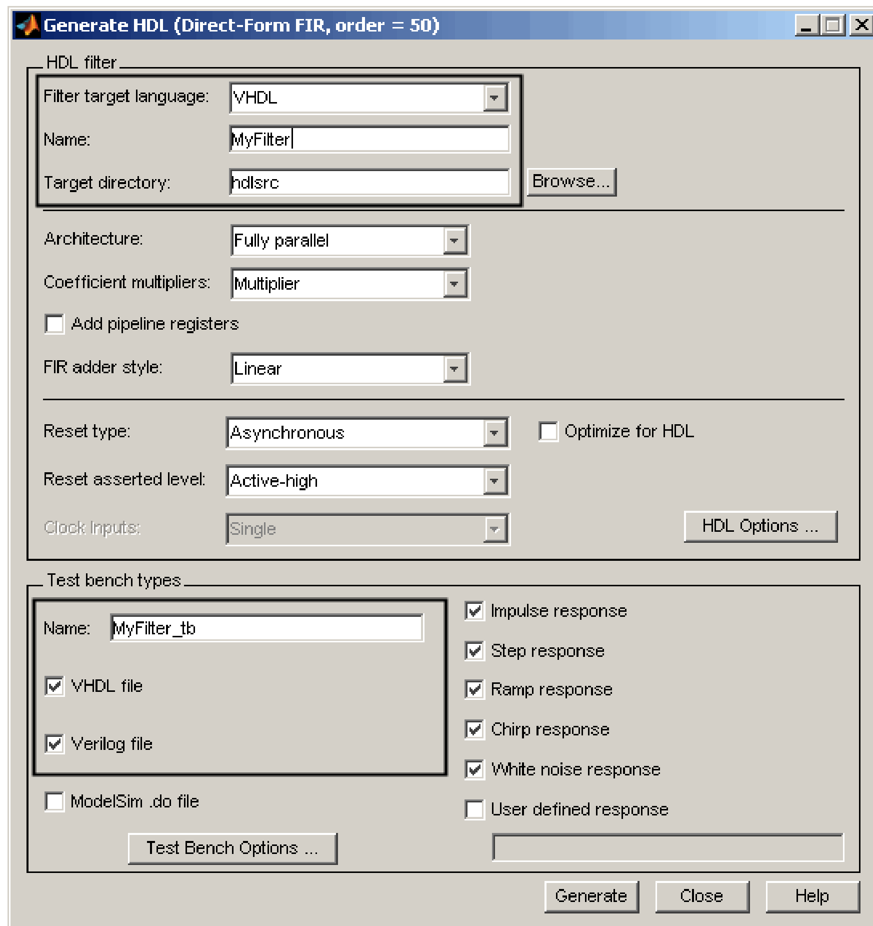
These restrictions *do not* apply for fixed-point filters.

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 3-97, 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 box.

You can specify a number of other test bench customizations, as described in "Customizing the Test Bench" on page 3-95.

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



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

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

### Starting generation of VHDL Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating VHDL testbench: D:\work\MyPlayArea\hdlsrc\MyFilter_tb.vhd
### Please wait ........
### Done generating VHDL test bench.

### Starting generation of Verilog Test Bench
### Generating input stimulus
### Done generating input stimulus; length 3429 samples.
### Generating Verilog testbench : D:\work\MyPlayArea\hdlsrc\MyFilter_tb.v
### Please wait ........
### Done generating Verilog test bench.
```

Note The length of the input stimulus samples varies from filter to filter. For example, the value 3429 in the preceding message sequence is not fixed; the value is dependent on the filter under test.

If you 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

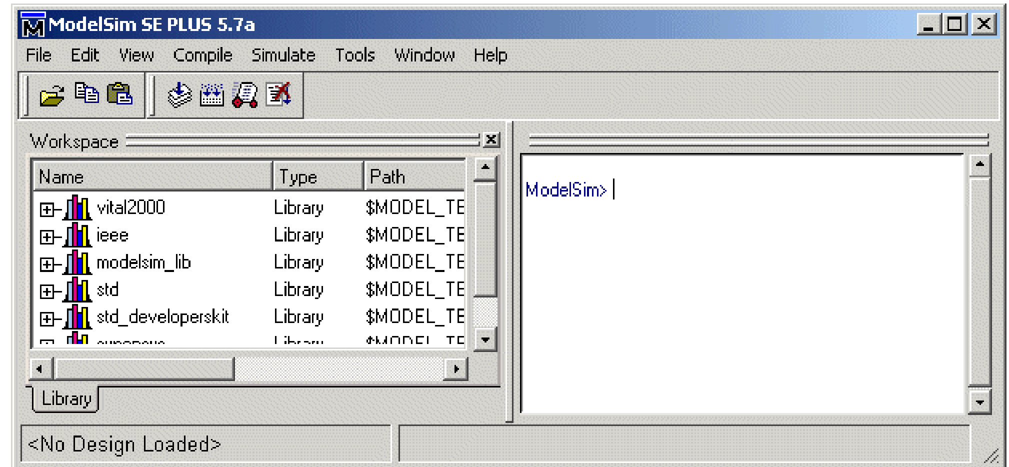
- Specify 'VHDL' or 'Verilog' for the TbType parameter. For double-precision filters, you must specify the type that matches the target language specified for your filter code.
- Make sure the property settings specified in the invocation of generatetb match those of the corresponding invocation of generatehdl. 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 — By Category”.

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

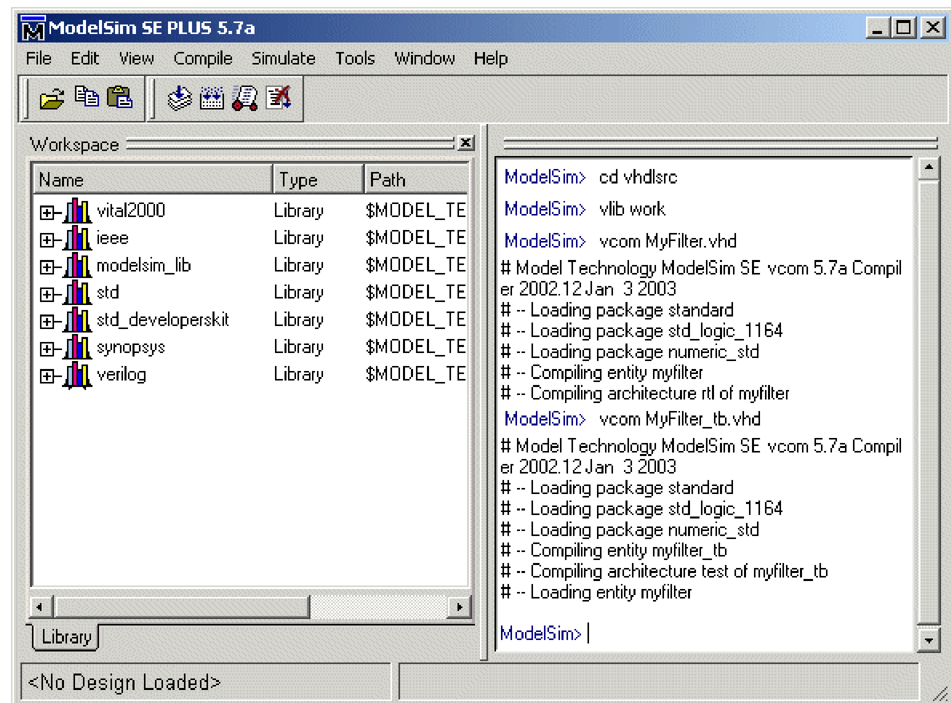
Using your choice HDL compiler, compile the generated filter and test bench HDL files. Depending on the language of the generated test bench and the simulator you are using, you might need to complete some precompilation setup. For example, in ModelSim, you might choose to create a design library to 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 VHDL test bench code that has floating-point (double) realizations, use a compiler that supports VHDL-93 or VHDL-02 (for example, in ModelSim, specify the `vcom` command with the `-93` option). Do not compile the generated test bench code with a VHDL-87 compiler. VHDL test benches using double- precision data types do not support VHDL-87, because test bench code uses the image attribute, which is available only in VHDL-93 or higher.

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



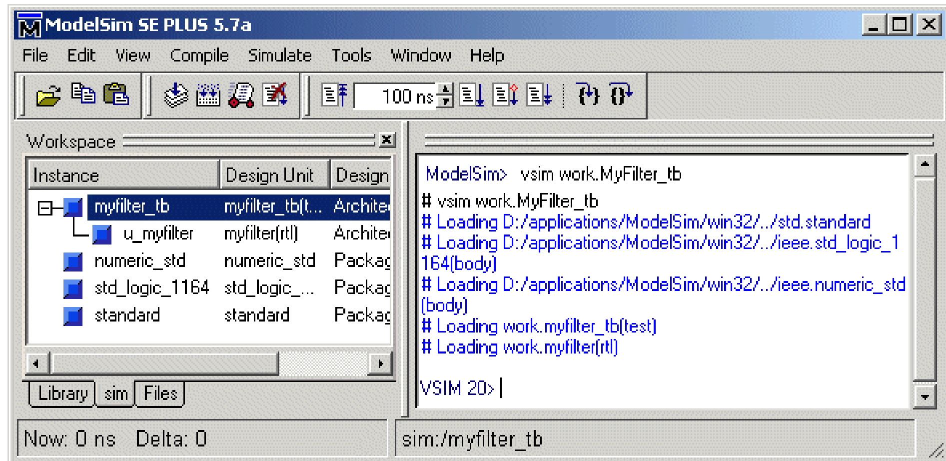
Running the Test Bench Simulation

Once your generated HDL files are compiled, load and run the test bench. The procedure for doing this varies depending on the simulator you are using. In

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

```
vsim work.MyFilter_tb
```

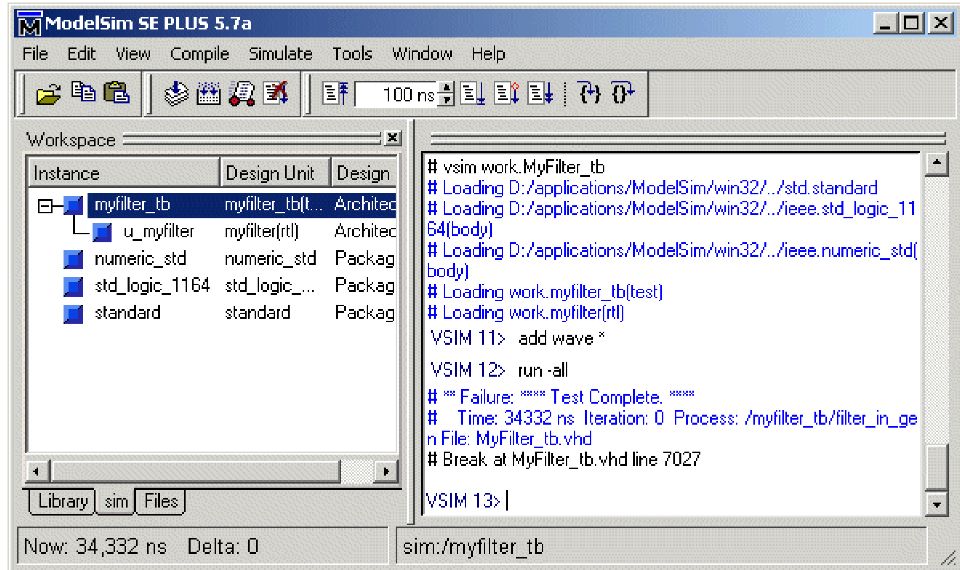
The following ModelSim display shows the results of loading `work.MyFilter_tb` with the `vsim` command.



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.



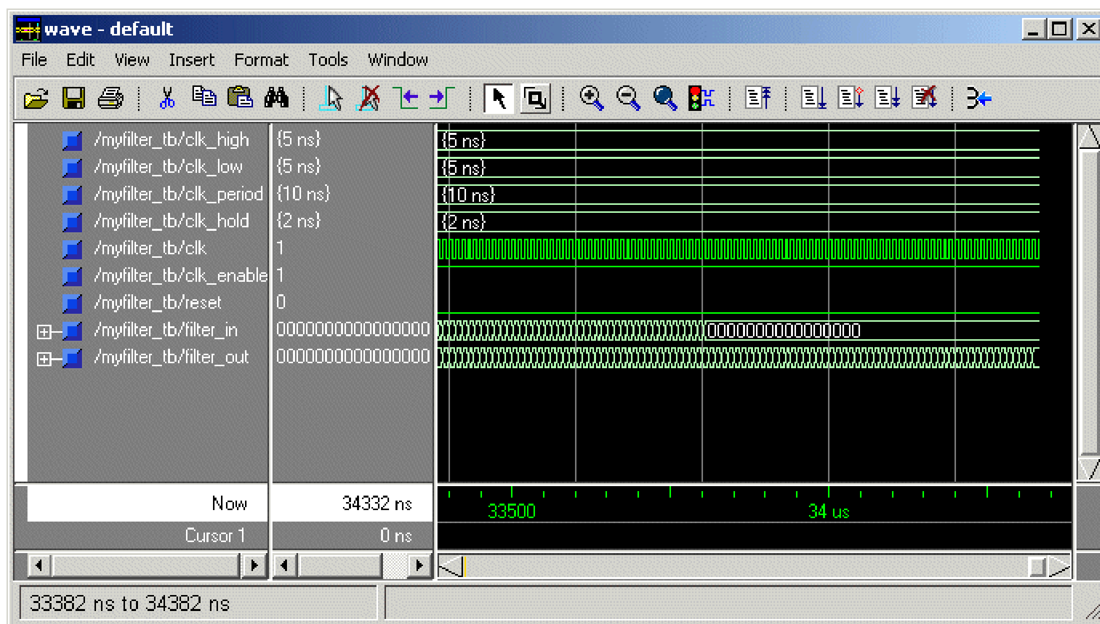
As your test bench simulation runs, watch for error messages. If any error messages appear, you must interpret them as they pertain to your filter design and the HDL customizations you applied with the Filter Design HDL Coder. For example, a number of HDL customization options allow you to specify settings 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.

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



Testing with a ModelSim Tcl/Tk .do File

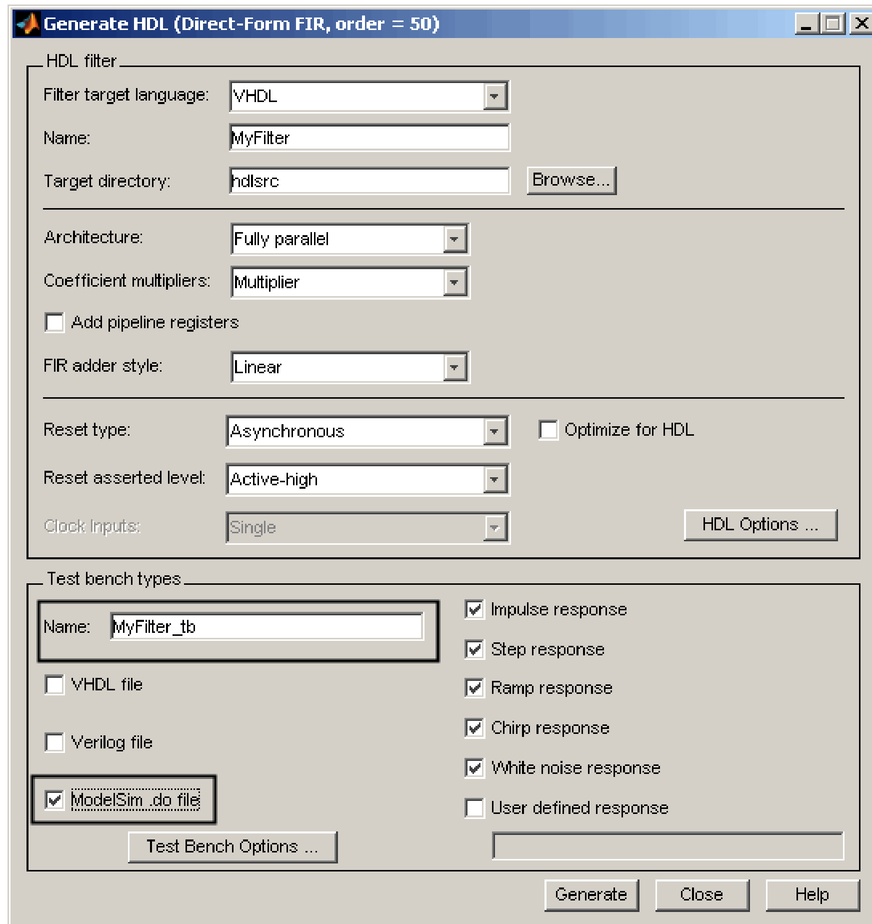
If you customize the Filter Design HDL Coder to generate a ModelSim Tcl/Tk `.do` file test bench, you must use ModelSim to test and verify your filter design. When you choose this test bench method, you need to

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 box. You can specify a number of other test bench customizations, as described in “Customizing the Test Bench” on page 3-95.

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



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

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: D:\work\MyPlayArea\hdlsrc\MyFilter.vhd
```

```
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation process for filter: MyFilter

### Starting generation of 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

Property...	Set to...	Can Affect...
'CastBeforeSum'	'on'	Numeric computations
'CoeffMultipliers'	'csd' or 'factored-csd'	Numeric computations

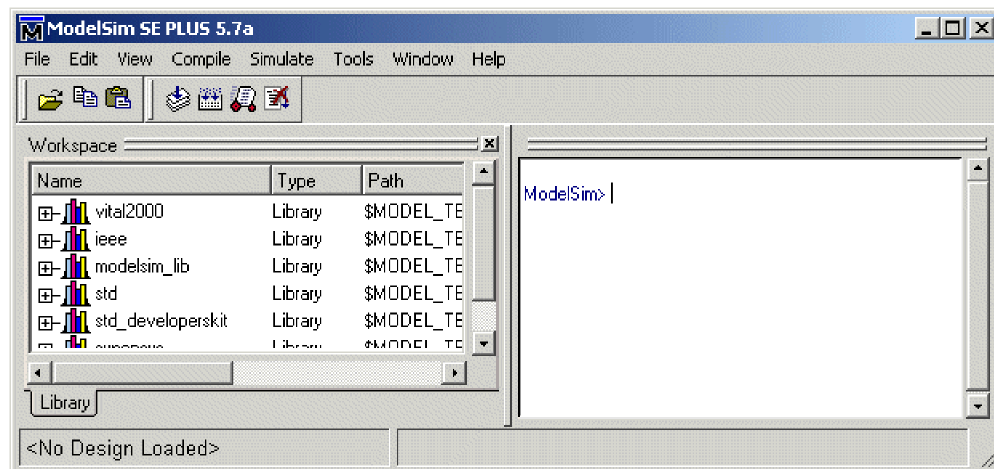
- Specify 'ModelSim' for the TbType parameter.
- Make sure the property settings specified in the invocation of generatetb match those of the corresponding invocation of generatehdl. You can do this in one of two ways:
  - Omit explicit property settings from the generatetb invocation. This function 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 — By Category”.

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

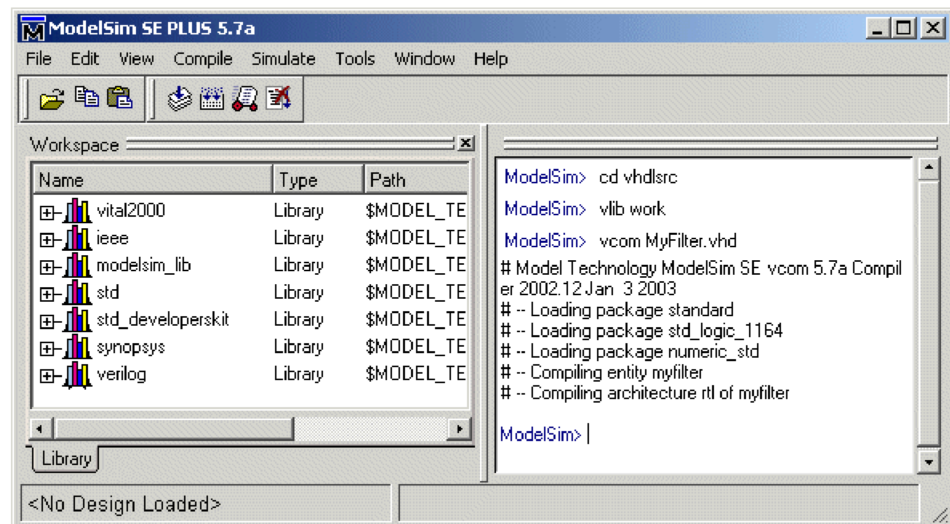
Using your choice HDL compiler, compile the generated filter HDL file. The test bench `.do` file looks for your compiled HDL elements in a design library named `work`. The design library stores the compiled HDL components. If the design library `work` does not exist, you can create it by setting the 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 test bench code that has floating-point (double) realizations, use a compiler that supports VHDL-93 or VHDL-02 (for example, in ModelSim, specify the `vcom` command with the `-93` option). Do not compile the generated test bench code with a VHDL-87 compiler. VHDL test benches using double- precision data types do not support VHDL-87, because test bench code uses the image attribute, which is available only in VHDL-93 or higher.

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



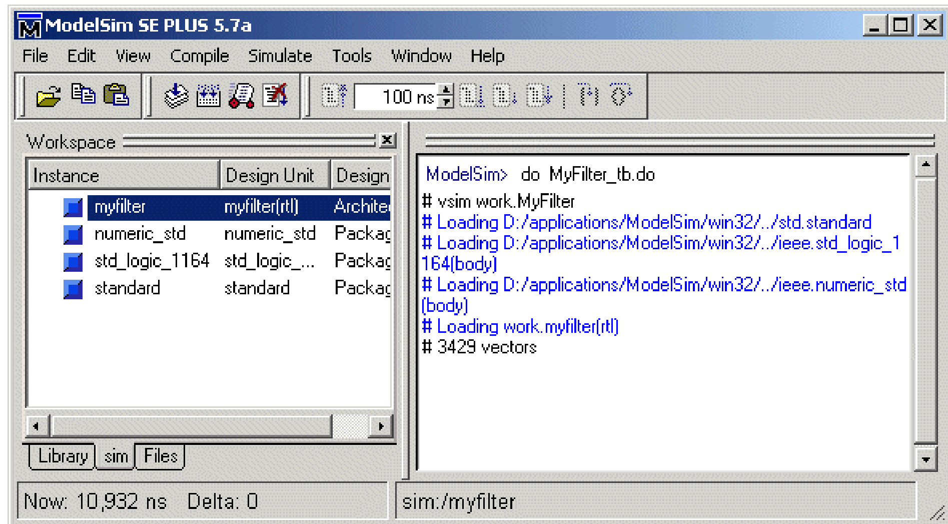
Execute the ModelSim .do File

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

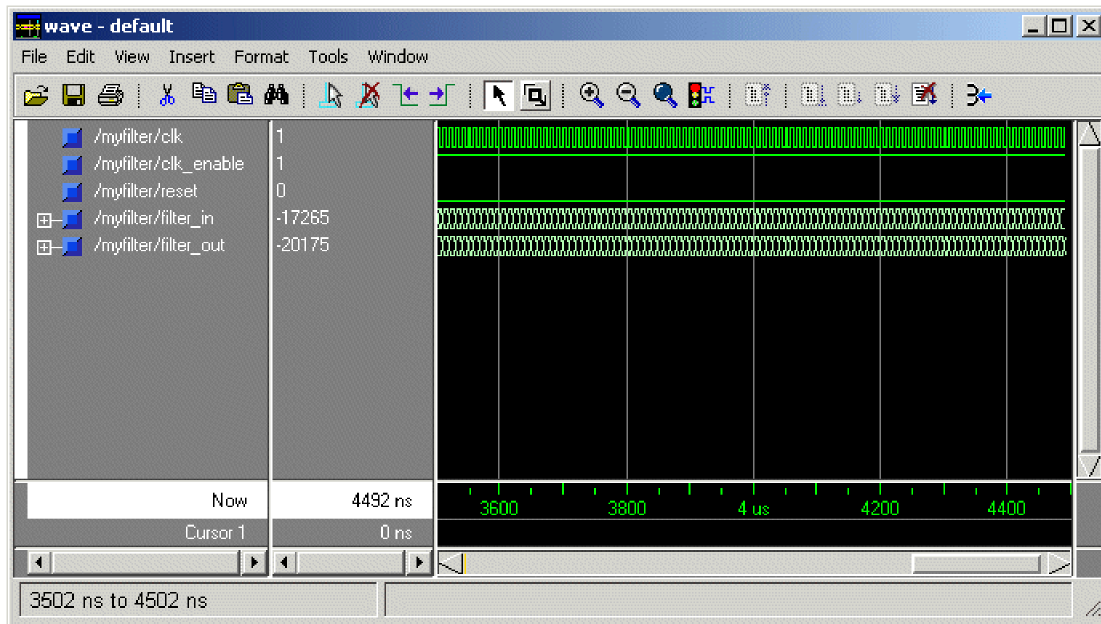
1. Loads the compiled filter for simulation.
2. Opens a **wave** window and populates it with filter signals.
3. Applies stimulus to filter signals with `force` commands.

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.



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.

5

Properties — By Category

Language Selection Properties (p. 5-2)	Lists properties for selecting language of generated HDL code
File Naming and Location Properties (p. 5-2)	Lists properties that name and specify 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. 5-3)	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
Script Generation Properties (p. 5-7)	Lists properties for customizing generated scripts for EDA tools

Language Selection Properties

`TargetLanguage`	Specify HDL language to use for generated filter code

File Naming and Location Properties

`Name`	Specify file name for generated HDL code and name for filter's VHDL entity or Verilog module
`TargetDirectory`	Identify directory into which generated output files are written
`VerilogFileExtension`	Specify file type extension for generated Verilog files
`VHDLFileExtension`	Specify file type extension for generated VHDL files

Reset Properties

`ResetAssertedLevel`	Specify asserted (active) level of reset input signal
`ResetType`	Specify whether to use asynchronous or synchronous reset style when generating HDL code for registers

Header Comment and General Naming Properties

ClockProcessPostfix	Specify string to append to HDL clock process names
CoeffPrefix	Specify prefix (string) for filter coefficient names
EntityConflictPostfix	Specify string to append to duplicate VHDL entity or Verilog module names
PackagePostfix	Specify a string to append to the specified filter name to form the name of a VHDL package file
ReservedWordPostfix	Specify string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words
SplitArchFilePostfix	Specify string to append to specified name to form name of file containing filter's VHDL architecture
SplitEntityArch	Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files
SplitEntityFilePostfix	Specify string to append to specified filter name to form name of file that contains filter's VHDL entity

Port Properties

AddInputRegister	Generate extra register in HDL code for filter input
AddOutputRegister	Generate extra register in HDL code for filter output

`ClockEnableInputPort`	Name HDL port for filter's clock enable input signals
`ClockEnableOutputPort`	For multirate filters (with single clock), specify name of clock enable output port
`ClockInputs`	For multirate filters, specify generation of single or multiple clock inputs
`InputPort`	Name HDL port for filter's input signals
`InputType`	Specify HDL data type for filter's input port
`OutputPort`	Name HDL port for filter's output signals
`OutputType`	Specify HDL data type for filter's output port
`ResetInputPort`	Name HDL port for filter's reset input signals

Advanced Coding Properties

`BlockGenerateLabel`	Specify string to append to block labels used for HDL `GENERATE` statements
`CastBeforeSum`	Enable or disable type casting of input values for addition and subtraction operations
`DALUTPartition`	Specify number and size of LUT partitions for distributed arithmetic architecture

`DARadix`	Specify number of bits processed simultaneously in distributed arithmetic architecture
`InlineConfigurations`	Specify whether generated VHDL code includes inline configurations
`InstanceGenerateLabel`	Specify string to append to instance section labels in VHDL `GENERATE` statements
`LoopUnrolling`	Specify whether VHDL `FOR` and `GENERATE` loops are unrolled and omitted from generated VHDL code
`OutputGenerateLabel`	Specify string that labels output assignment block for VHDL `GENERATE` statements
`SafeZeroConcat`	Specify syntax used in generated VHDL code for concatenated zeros
`ScaleWarnBits`	Specify threshold for generation of warning for scale values that may cause quantization noise
`UseAggregatesForConst`	Specify whether all constants are represented by aggregates, including constants that are less than 32 bits
`UserComment`	Specify string added as comment line in header of generated filter and test bench files
`UseRisingEdge`	Specify VHDL coding style used to check for rising edges when operating on registers
`UseVerilogTimescale`	Allow or exclude use of compiler `` `timescale `` directives in generated Verilog code

Optimization Properties

AddPipelineRegisters	Optimize clock rate used by filter code by adding pipeline registers
CoeffMultipliers	Specify technique used for processing coefficient multiplier operations
FIRAdderStyle	Specify final summation technique used for FIR filters
OptimizeForHDL	Specify whether generated HDL code is optimized for specific performance or space requirements
ReuseAccum	Enable accumulator reuse, generating cascade-serial architecture for FIR filters
SerialPartition	Specify number and size of partitions generated for serial FIR filter architectures

Test Bench Properties

ClockHighTime	Specify period, in nanoseconds, during which test bench drives clock input signals high (1)
ClockInputPort	Name HDL port for filter's clock input signals
ClockLowTime	Specify period, in nanoseconds, during which test bench drives clock input signals low (0)
ErrorMargin	Specify error margin for HDL language-based test benches
ForceClock	Specify whether test bench forces clock input signals

`ForceClockEnable`	Specify whether test bench forces clock enable input signals
`ForceReset`	Specify whether test bench forces reset input signals
`HoldTime`	Specify hold time for filter data input signals and forced reset input signals
`SimulatorFlags`	Specify simulator flags applied to generated test bench
`TestBenchName`	Name VHDL test bench entity or Verilog module and file that contains test bench code
`TestBenchStimulus`	Specify input stimuli that test bench applies to filter
`TestBenchUserStimulus`	Specify user-defined MATLAB function that returns vector of values that test bench applies to filter

Script Generation Properties

`EDAScriptGeneration`	Enable or disable generation of script files for third-party tools
`HDLCompileInit`	Specify string written to initialization section of compilation script
`HDLCompileTerm`	Specify string written to termination section of compilation script
`HDLCompileVerilogCmd`	Specify command string written to compilation script for Verilog files
`HDLCompileVHDLCmd`	Specify command string written to compilation script for VHDL files

`HDLSimCmd`	Specify simulation command written to simulation script
`HDLSimInit`	Specify string written to initialization section of simulation script
`HDLSimTerm`	Specify string written to termination section of simulation script
`HDLSimViewWaveCmd`	Specify waveform viewing command written to simulation script
`HDLSynthCmd`	Specify command written to synthesis script
`HDLSynthInit`	Specify string written to initialization section of synthesis script
`HDLSynthTerm`	Specify string written to termination section of synthesis script

Properties — Alphabetical List

AddInputRegister

Purpose

Generate extra register in HDL code for filter input

Settings

`'on'` (default)

Add an extra input register to the filter's generated HDL code.

The code declares a signal named `input_register` and includes a `PROCESS` block similar to the block below. Names and meanings of the timing parameters (clock, clock enable, and reset) and the coding style that checks for clock events may vary depending on other property settings.

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

`'off'`

Omit the extra input register from the filter's generated HDL code.

Consider omitting the extra register if you are incorporating the filter into HDL code that already has a source for driving the filter. You might also consider omitting the extra register if the latency it introduces to the filter is not tolerable.

See Also

`AddOutputRegister`

Purpose

Generate extra register in HDL code for filter output

Settings

'on' (default)

Add an extra output register to the filter's generated HDL code.

The code declares a signal named output_register and includes a PROCESS block similar to the block below. Names and meanings of the timing parameters (clock, clock enable, and reset) and the coding style that checks for clock events may vary depending on other property settings.

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

'off'

Omit the extra output register from the filter's generated HDL code.

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

See Also

AddInputRegister

AddPipelineRegisters

Purpose

Optimize clock rate used by filter code by adding pipeline registers

Settings

`'on'`

Add a pipeline register between stages of computation in a filter. For example, for a sixth-order IIR filter, the coder adds two pipeline registers, one between the first and second sections and one between the second and third sections. Although the registers add to the overall filter latency, they provide significant improvements to the clock rate.

For...	A Pipeline Register Is Added Between...
FIR Transposed filters	Coefficient multipliers and adders
FIR, Asymmetric FIR, and Symmetric FIR filters	Levels of a tree-based final adder
IIR filters	Sections

`'off'` (default)

Suppress the use of pipeline registers.

Usage Notes

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see "Optimizing Final Summation for FIR Filters" on page 3-60 .

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`

Purpose

Specify string to append to block labels used for HDL `GENERATE` statements

Settings

`'string'`

Specify a postfix string to append to block labels used for HDL `GENERATE` statements. The default string is `_gen`.

See Also

`InstanceGenerateLabel`, `OutputGenerateLabel`

CastBeforeSum

Purpose

Enable or disable type casting of input values for addition and subtraction operations

Settings

`'on'`(default)

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

`InlineConfigurations`, `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`, `UseVerilogTimescale`

Purpose

Name HDL port for filter's clock enable input signals

Settings

`'string'`

The default name for the filter's clock enable input port is `clk_enable`.

For example, if you specify the string `'filter_clock_enable'` for filter entity `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`

ClockEnableOutputPort

Purpose For multirate filters (with single clock), specify name of clock enable output port

Settings 'string'

The default name for the generated clock enable output port isce_out.

For multirate filters, a clock enable output is generated when you select Single from the **Clock inputs** menu in the Generate HDL dialog. In this case only, the **Clock enable output port** option is enabled.

Usage Notes For multirate filters, a clock enable output is generated when Single is selected from the **Clock inputs** menu in the Generate HDL dialog. In this case only, the **Clock enable output port** option is enabled.

See Also ClockInputs

Purpose

Specify period, in nanoseconds, during which test bench drives clock input signals high (1)

Settings

`ns`

The default is `5`.

Usage Notes

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

See Also

`ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ClockInputPort

Purpose

Name HDL port for filter's clock input signals

Settings

'string'

The default clock input port name is clk.

For example, if you specify the string 'filter_clock' for filter entity Hd, the generated entity declaration might look as follows:

```
ENTITY Hd IS
   PORT( filter_clock   :  IN  std_logic;
         clk_enable     :  IN  std_logic;
         reset          :  IN  std_logic;
         filter_in      :  IN  std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         filter_out     :  OUT std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         );
ENDHd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word signal, the resulting name string would be signal_rsvd. See ReservedWordPostfix in for more information.

See Also

ClockEnableInputPort, InputPort, InputType, OutputPort, OutputType, ResetInputPort

Purpose

For multirate filters, specify generation of single or multiple clock inputs

Settings

`'Single'` (default)

Generate a single clock input for a multirate filter. When this option is selected, the `ENTITY` declaration for the filter defines a single clock input with an associated clock enable input and clock enable output. The generated code maintains a counter that controls the timing of data transfers to the filter output (for decimation filters) or input (for interpolation filters). The counter is, in effect, a secondary clock whose rate is determined by the filter's decimation or interpolation factor.

`'Multiple'`

Generate multiple clock inputs for a multirate filter. When this option is selected, the `ENTITY` declaration for the filter defines separate clock inputs (each with an associated clock enable input) for each rate of a multirate filter. (For currently supported multirate filters, there are two such rates).

Usage Notes

The **Clock inputs** menu is enabled only when a multirate filter (of one of the types supported for code generation) has been designed in `fdatool`.

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

See Also

`ClockEnableOutputPort`

ClockLowTime

Purpose

Specify period, in nanoseconds, during which test bench drives clock input signals low (0)

Settings

`ns`

The default is `5`.

Usage Notes

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

See Also

`ClockHighTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`,

ClockProcessPostfix

Purpose

Specify string to append to HDL clock process names

Settings

`'string'`

The default postfix 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 technique used for processing coefficient multiplier operations

Settings `'multiplier'` (default)

Retain multiplier operations in the generated HDL code.

`'csd'`

This option uses canonic signed digit (CSD) techniques, which replace multiplier operations with shift and add operations. CSD techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This decreases the area used by the filter while maintaining or increasing clock speed.

`'factored-csd'`

This option uses factored CSD techniques, which replace multiplier operations with shift and add operations on prime factors of the coefficients. This option lets you achieve a greater filter area reduction than CSD, at the cost of decreasing clock speed.

Note If the CSD or Factored CSD optimizations are selected, 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 "Optimizing Coefficient Multipliers" on page 3-59 for more information.

See Also `AddPipelineRegisters`, `FIRAdderStyle`, `OptimizeForHDL`

Purpose

Specify prefix (string) for filter coefficient names

Settings

`'string'`

The default prefix for filter coefficient names is `coeff`.

For...	The Prefix Is Concatenated with...
FIR filters	Each coefficient number, starting with 1. For example, the default for the first coefficient would be `coeff1`.
IIR filters	An underscore (_) and an `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) followed by the string `_section`*n*, where *n* is the section number. For example, the default for the first numerator coefficient of the third section is `coeff_b1_section3`.

For example:

```
ARCHITECTURE rtl OF Hd IS
  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>)
       OF signed(15 DOWNTO 0); -- sfix16_En15
  CONSTANT coeff1 : signed(15 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
  CONSTANT coeff2 : signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
  CONSTANT coeff3 : signed(15 DOWNTO 0) := to_signed(-81, 16); -- sfix16_En15
  CONSTANT coeff4 : signed(15 DOWNTO 0) := to_signed(120, 16); -- sfix16_En15
.
.
.
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word `signal`, the resulting name string would be `signal_rsvd`. See `ReservedWordPostfix` for more information.

See Also

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

Purpose

Specify number and size of LUT partitions for distributed arithmetic architecture

Settings

`[p1 p2...pN]`

Where `[p1 p2 p3...pN]` is a vector of `N` integers, divides the LUT used in distributed arithmetic (DA) into `N` partitions. Each vector element specifies the size of a partition. The maximum size for an individual partition is 12. The sum of all vector elements must be equal to the filter length. The filter length is calculated differently depending on the filter type (see “Distributed Arithmetic for FIR Filters” on page 3-71).

Usage Notes

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

```
b = [0.0349 0.4302 0.4302 0.4302 0.0349];
Hd = dfilt.dffir(b);
Hd.arithmetic = 'fixed';
generatehdl (Hd, 'DALUTPartition', 5);
```

See “Distributed Arithmetic for FIR Filters” on page 3-71 for a complete description of DA.

See Also

`DARadix`

DARadix

Purpose

Specify number of bits processed simultaneously in distributed arithmetic architecture

Settings

`N`

`N` specifies the number of bits processed simultaneously in a distributed arithmetic (DA) architecture. `N` must be

- A nonzero positive integer that is a power of two
- Such that `mod(W, log2(N)) = 0` where `W` is the input word size of the filter.

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

Usage Notes

The `DARadix` property lets you introduce a degree of parallelism into the operation of DA, improving performance at the expense of area. See "Distributed Arithmetic for FIR Filters" on page 3-71 for a complete description of DA.

See Also

`DALUTPartition`

EDAScriptGeneration

Purpose Enable or disable generation of script files for third-party tools

Settings

`'on'` (default)

Enable generation of script files.

`'off'`

Disable generation of script files.

See Also "Generating Scripts for EDA Tools" on page 3-110

EntityConflictPostfix

Purpose

Specify string to append to duplicate VHDL entity or Verilog module names

Settings

`'string'`

The specified postfix resolves duplicate VHDL entity or Verilog module names. The default string is `_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`, `CoeffPrefix`, `PackagePostfix`, `ReservedWordPostfix`

Purpose

Specify error margin for HDL language-based test benches

Settings

`n`

Some HDL optimizations can generate test bench code that produces numeric results that differ from those produced by the original MATLAB filter function. By specifying an error margin, you can specify an acceptable minimum number of bits by which the numeric results can differ before the coder issues a warning.

Specify the error margin as an integer number of bits.

Usage Notes

Optimizations that can generate test bench code that produces numeric results that differ from those produced by the original MATLAB filter function include

- `CastBeforeSum` (`qfilts` only)
- `OptimizeForHDL`
- `CoeffMultipliers`
- `FIRAdderStyle` (`'Tree'`)
- `AddPipelineRegisters` (for FIR, Asymmetric FIR, and Symmetric FIR filters)

The error margin is the number of least significant bits a Verilog or VHDL language-based test bench can ignore when comparing the numeric results before generating a warning.

For fixed-point filters, the **Error margin (bits)** value is initialized to a default value of `4`.

For double precision floating-point filters, the **Error margin (bits)** value is fixed at `1e-9`. This value is displayed with the field disabled to indicate that the value cannot be changed.

See Also

`AddPipelineRegisters`, `CastBeforeSum`, `CoeffMultipliers`, `FIRAdderStyle`, `OptimizeForHDL`

FIRAdderStyle

Purpose

Specify final summation technique used for FIR filters

Settings

`'linear'` (default)

Apply linear adder summation. This technique is discussed in most DSP text books.

`'tree'`

Increase clock speed while maintaining the area used. This option creates a final adder that performs pair-wise addition on successive products that execute in parallel, rather than sequentially.

Usage Notes

If you are generating HDL code for a FIR filter, consider optimizing the final summation technique by applying tree or pipeline final summation techniques. Pipeline mode produces results similar to tree mode with the addition of a stage of pipeline registers after processing each level of the tree.

For information on applying pipeline mode, see `AddPipelineRegisters`.

Consider the following tradeoffs when selecting the final summation technique for your filter:

- The number of adder operations for linear and tree mode are the same, but the timing for tree mode might be significantly better due to summations occurring in parallel.
- Pipeline mode optimizes the clock rate, but increases the filter latency by the base 2 logarithm of the number of products to be added, rounded up to the nearest integer.
- Linear mode 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 MATLAB filter function.

See Also

`AddPipelineRegisters`, `CoeffMultipliers`, `OptimizeForHDL`

Purpose

Specify whether test bench forces clock input signals

Settings

`'on'` (default)

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

`'off'`

Specify that a user-defined external source forces the clock input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClockEnable`, `ForceReset`, `HoldTime`,

Purpose

Specify whether test bench forces clock enable input signals

Settings

`'on'` (default)

Specify that the test bench forces the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value.

`'off'`

Specify that a user-defined external source forces the clock enable input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceReset`, `HoldTime`,

ForceReset

Purpose

Specify whether test bench forces reset input signals

Settings

`'on'` (default)

Specify that the test bench forces the reset input signals. If you enable this option, you can also specify a hold time to control the timing of a reset.

`'off'`

Specify that a user-defined external source forces the reset input signals.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceClockEnable`, `HoldTime`,

HDLCompileInit

Purpose Specify string written to initialization section of compilation script

Settings `'string'`

The default string is `'vlib work\n'`.

See Also “Generating Scripts for EDA Tools” on page 3-110

Purpose Specify string written to termination section of compilation script

Settings `'string'`

The default is the null string (`''`).

See Also “Generating Scripts for EDA Tools” on page 3-110

HDLCompileVerilogCmd

Purpose Specify command string written to compilation script for Verilog files

Settings `'string'`

The default string is `'vlog %s %s\n'`.

The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).

See Also "Generating Scripts for EDA Tools" on page 3-110

Purpose

Specify command string written to compilation script for VHDL files

Settings

`'string'`

The default string is `'vcom %s %s\n'`.

The two arguments are the contents of the `'SimulatorFlags'` property and the file name of the current entity or module. To omit the flags, set `'SimulatorFlags'` to `''` (the default).

See Also

"Generating Scripts for EDA Tools" on page 3-110

Purpose Specify simulation command written to simulation script

Settings `'string'`

The default string is`'vsim work.%s\n'`.

The implicit argument is the top-level module or entity name.

See Also "Generating Scripts for EDA Tools" on page 3-110

Purpose Specify string written to initialization section of simulation script

Settings `'string'`

The default string is

```
['onbreak resume\n',...
'onerror resume\n']
```

See Also "Generating Scripts for EDA Tools" on page 3-110

Purpose Specify string written to termination section of simulation script

Settings `'string'`

The default string is `'run -all\n'`

See Also "Generating Scripts for EDA Tools" on page 3-110

HDLSynthCmd

Purpose

Specify command written to synthesis script

Settings

`'string'`

The default string is `'add_file %s\n'`

The implicit argument is the file name of the entity or module.

See Also

"Generating Scripts for EDA Tools" on page 3-110

HDLSynthInit

Purpose Specify string written to initialization section of synthesis script

Settings `'string'`

The default string is `'project -new %s.prj\n'`, which is a synthesis project creation command.

The implicit argument is the top-level module or entity name.

See Also "Generating Scripts for EDA Tools" on page 3-110

Purpose

Specify string written to termination section of synthesis script

Settings

'string'

The default string is

```
['set_option -technology VIRTEX2\n',...
'set_option -part XC2V500\n',...
'set_option -synthesis_onoff_pragma 0\n',...
'set_option -frequency auto\n',...
'project -run synthesis\n']
```

See Also

"Generating Scripts for EDA Tools" on page 3-110

HDLSimViewWaveCmd

Purpose Specify waveform viewing command written to simulation script

Settings `'string'`

The default string is `'add wave sim:%s\n'`

The implicit argument is the top-level module or entity name.

See Also "Generating Scripts for EDA Tools" on page 3-110

Purpose

Specify hold time for filter data input signals and forced reset input signals

Settings

ns

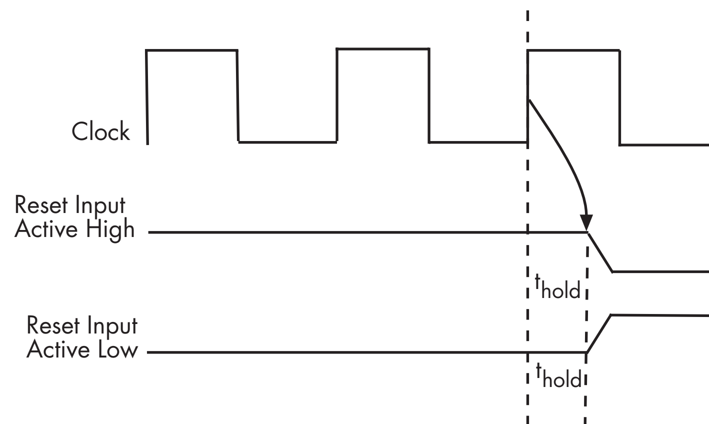
Specify the number of nanoseconds (a positive integer) during which filter data input signals and forced reset input signals are held past the clock rising edge. The default is 2.

This option applies to reset input signals only if forced resets are enabled.

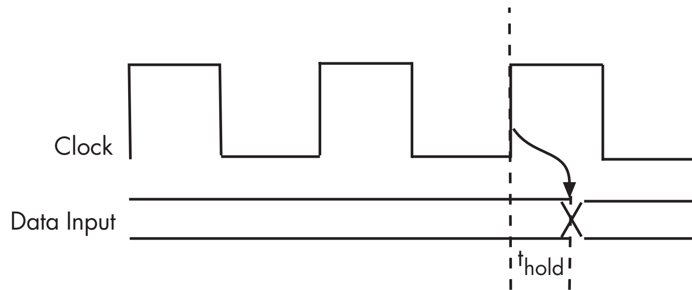
Usage Notes

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

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



Hold Time for Reset Input Signals



Hold Time for Data Input Signals

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`

InlineConfigurations

Purpose

Specify whether generated VHDL code includes inline configurations

Settings

`'on'` (default)

Include VHDL configurations in any file that instantiates a component.

`'off'`

Suppress the generation of configurations and require user-supplied external configurations. Use this setting if you are creating your own VHDL configuration files.

Usage Notes

VHDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, the 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`, , `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

Purpose Name HDL port for filter's input signals

Settings 'string'

The default string is filter_in.

For example, if you specify the string 'filter_data_in' for filter entity Hd, the generated entity declaration might look as follows:

```
ENTITY Hd IS
   PORT( clk              :  IN  std_logic;
         clk_enable       :  IN  std_logic;
         reset            :  IN  std_logic;
         filter_data_in   :  IN  std_logic_vector (15 DOWNTO 0);
         filter_out       :  OUT std_logic_vector (15 DOWNTO 0);
         );
END Hd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word signal, the resulting name string would be signal_rsvd. See ReservedWordPostfix for more information.

See Also ClockEnableInputPort, ClockInputPort, OutputPort, OutputType, ResetInputPort

InputType

Purpose

Specify HDL data type for filter's input port

Settings

`'std_logic_vector'`

Specifies VHDL type `STD_LOGIC_VECTOR` for the filter input port.

`'signed/unsigned'`

Specifies VHDL type `SIGNED` or `UNSIGNED` for the filter input port.

`'wire'` (Verilog)

If the target language is Verilog, the data type for all ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, `ClockInputPort`, `InputPort`, `OutputPort`, `OutputType`, `ResetInputPort`

InstanceGenerateLabel

Purpose

Specify string to append to instance section labels in VHDL `GENERATE` statements

Settings

`'string'`

Specify a postfix string to append to instance section labels in VHDL `GENERATE` statements. The default string is `_gen`.

See Also

`BlockGenerateLabel`, `OutputGenerateLabel`

Purpose

Specify whether VHDL `FOR` and `GENERATE` loops are unrolled and omitted from generated VHDL code

Settings

`'on'`

Unroll and omit `FOR` and `GENERATE` loops from the generated VHDL code. Verilog is always unrolled.

This option takes into account that some EDA tools do not support `GENERATE` loops. If you are using such a tool, enable this option to omit loops from your generated VHDL code.

`'off'` (default)

Include `FOR` and `GENERATE` loops in the generated VHDL code.

Usage Notes

The setting of this option does not affect generated VHDL code during simulation or synthesis.

See Also

`CastBeforeSum`, `InlineConfigurations`, , `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

Name

Purpose

Specify file name for generated HDL code and name for filter's VHDL entity or Verilog module

Settings

`'string'`

The defaults take the name of the filter as defined in the FDATool.

The file type extension for the generated file is the string specified for the file type extension option for the selected language.

The generated file is placed in the directory or path specified by `TargetDirectory`.

If you specify a string that is a reserved word in the selected language, the coder appends the string specified by `ReservedWordPostfix`. For a list of reserved words, see "Setting the Postfix String for Resolving HDL Reserved Word Conflicts" on page 3-37.

See Also

`VerilogFileExtension`, `VHDLFileExtension`, `TargetDirectory`

Purpose

Specify whether generated HDL code is optimized for specific performance or space requirements

Settings

`'on'`

Generate HDL code that is optimized for specific performance or space requirements. As a result of these optimizations, the Filter Design HDL Coder may

- Make tradeoffs concerning data types
- Avoid excessive quantization
- Generate code that produces numeric results that differ from results produced by the original MATLAB filter function

`'off'` (default)

Generate HDL code that maintains bit compatibility with the numeric results produced by the specified quantized filter in MATLAB.

See Also

`AddPipelineRegisters`, `CoeffMultipliers`, `FIRAdderStyle`

OutputGenerateLabel

Purpose Specify string that labels output assignment block for VHDL `GENERATE` statements

Settings `'string'`

Specify a postfix string to append to output assignment block labels in VHDL `GENERATE` statements. The default string is `outputgen`.

See Also `BlockGenerateLabel`, `InstanceGenerateLabel`

Purpose

Name HDL port for filter's output signals

Settings

'string'

The default is filter_out.

For example, if you specify 'filter_data_out' for filter entity Hd, the generated entity declaration might look as follows:

```
ENTITY Hd IS
   PORT( clk              :  IN  std_logic;
         clk_enable       :  IN  std_logic;
         reset            :  IN  std_logic;
         filter_in        :  IN  std_logic_vector (15 DOWNTO 0);
         filter_data_out  :  OUT std_logic_vector (15 DOWNTO 0);
         );
ENDHd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word signal, the resulting name string would be signal_rsvd. See ReservedWordPostfix for more information.

See Also

ClockEnableInputPort, ClockInputPort, InputPort, InputType, OutputType, ResetInputPort

OutputType

Purpose

Specify HDL data type for filter's output port

Settings

The filter's output port has the same type as the specified input port type.

`'std_logic_vector'` (VHDL default)

The filter's output port has VHDL type `STD_LOGIC_VECTOR`.

`'signed/unsigned'`

The filter's input port has type `SIGNED` or `UNSIGNED`.

`'wire'` (Verilog)

If the target language is Verilog, the data type for all ports is `wire`. This property is not modifiable in this case.

See Also

`ClockEnableInputPort`, `ClockInputPort`, `InputPort`, `InputType`, `OutputPort`, `ResetInputPort`

PackagePostfix

Purpose

Specify a string to append to the specified filter name to form the name of a VHDL package file

Settings

`'string'`

The coder applies this option only if a package file is required for the design. The default string is `_pkg`.

See Also

`ClockProcessPostfix`, `CoeffPrefix`, `EntityConflictPostfix`, `ReservedWordPostfix`

ReservedWordPostfix

Purpose

Specify string to append to value names, postfix values, or labels that are VHDL or Verilog reserved words

Settings

'string'

The default postfix is _rsvd.

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

See Also

ClockProcessPostfix, CoeffPrefix, EntityConflictPostfix, PackagePostfix

ResetAssertedLevel

Purpose

Specify asserted (active) level of reset input signal

Settings

`'active-high'` (default)

Specify that the reset input signal must be driven high (1) to reset registers in the filter design. For example, the following code fragment checks whether `reset` is active high before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
.
.
.
```

`'active-low'`

Specify that the reset input signal must be driven low (0) to reset registers in the filter design. For example, the following code fragment checks whether `reset` is active low before populating the `delay_pipeline` register:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '0' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
.
.
.
```

See Also

`ResetType`

ResetInputPort

Purpose

Name HDL port for filter's reset input signals

Settings

'string'

The default name for the filter's reset input port is reset.

For example, if you specify the string 'filter_reset' for filter entity Hd, the generated entity declaration might look as follows:

```
ENTITY Hd IS
   PORT( clk             :  IN  std_logic;
         clk_enable      :  IN  std_logic;
         filter_reset    :  IN  std_logic;
         filter_in       :  IN  std_logic_vector (15 DOWNTO 0);
         filter_out      :  OUT std_logic_vector (15 DOWNTO 0);
         );
END Hd;
```

If you specify a string that is a VHDL reserved word, a reserved word postfix string is appended to form a valid VHDL identifier. For example, if you specify the reserved word signal, the resulting name string would be signal_rsvd. See ReservedWordPostfix for more information.

Usage Notes

If the reset asserted level is set to active high, the reset input signal is asserted active high (1) and the input value must be high (1) for the entity's registers to be reset. If the reset asserted level is set to active low, the reset input signal is asserted active low (0) and the input value must be low (0) for the entity's registers to be reset.

See Also

ClockEnableInputPort, ClockInputPort, InputPort, InputType, OutputPort, OutputType

Purpose

Specify whether to use asynchronous or synchronous reset style when generating HDL code for registers

Settings

`'async'` (default)

Use an asynchronous reset style. The following generated code fragment illustrates the use of asynchronous resets. Note that the process block does not check for an active clock before performing a reset.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF Reset_Port = '1' THEN
    delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
  ELSIF Clock_Port'event AND Clock_Port = '1' THEN
    IF ClockEnable_Port = '1' THEN
      delay_pipeline(0) <= signed(Fin_Port)
      delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS delay_pipeline_process;
```

`'sync'`

Use a synchronous reset style. Code for a synchronous reset follows. This process block checks for a clock event, the rising edge, before performing a reset.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
  IF rising_edge(Clock_Port) THEN
    IF Reset_Port = '0' THEN
      delay_pipeline (0 To 50) <= (OTHERS => (OTHERS => '0'));
    ELSIF ClockEnable_Port = '1' THEN
      delay_pipeline(0) <= signed(Fin_Port)
      delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS delay_pipeline_process;
```

See Also `ResetAssertedLevel`

Purpose

Enable accumulator reuse, generating cascade-serial architecture for FIR filters

Settings

`'off'` (default)

Disable accumulator reuse.

`'on'`

Enable accumulator reuse when generating a partly serial architecture. (i.e., a cascade-serial architecture). If the number and size of serial partitions is not specified (see `SerialPartition`), Filter Design HDL Coder generates an optimal partition.

Usage Notes

In a cascade-serial architecture, filter taps are grouped into a number of serial partitions, and the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of all partitions is therefore computed at the accumulator of the first partition. This technique, termed *accumulator reuse*, saves chip area.

See "Speed vs. Area Optimizations for FIR Filters" on page 3-61 for a complete description of parallel and serial architectures and a list of filter types upported for each architecture.

See Also

`SerialPartition`

Purpose Specify syntax used in generated VHDL code for concatenated zeros

Settings `'on'` (default)

Use the type-safe syntax, `'0' & '0'`, for concatenated zeros. Typically, this syntax is preferred.

`'off'`

Use the syntax "`000000...`" for concatenated zeros. This syntax can be easier to read and is more compact, but can lead to ambiguous types.

See Also `CastBeforeSum`, `InlineConfigurations`, , `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

Purpose

Specify threshold for generation of warning for scale values that may cause quantization noise

Settings

`n`

Specify a numeric value the coder uses 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 3-46.

See Also

`CastBeforeSum`, `InlineConfigurations`, `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`, `UseVerilogTimescale`

SerialPartition

Purpose

Specify number and size of partitions generated for serial FIR filter architectures

Settings

`N`

Generate a fully serial architecture for a filter of length `N`.

`[p1 p2 p3...pN]`

Where `[p1 p2 p3...pN]` is a vector of `N` integers, generate a partly serial architecture with `N` partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter.

Usage Notes

To save chip area in a partly serial architecture, you can enable the `ReuseAccum` property.

See "Speed vs. Area Optimizations for FIR Filters" on page 3-61 for a complete description of parallel and serial architectures and a list of filter types upported for each architecture.

See Also

`ReuseAccum`

Purpose

Specify simulator flags applied to generated test bench

Settings

`'string'`

Specify options that are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag `-93`.

Usage Notes

The flags you specify with this option are added to the `vsim` command in generated ModelSim `.do` test bench files.

SplitArchFilePostfix

Purpose

Specify string to append to specified name to form name of file containing filter's VHDL architecture

Settings

`'string'`

The default is `_arch`. This option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.

Usage Notes

The option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.

See Also

`SplitEntityArch`, `SplitEntityFilePostfix`

Purpose

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

`'on'`

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

The names of the entity and architecture files derive from the base file name (as specified by the filter name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. You can override the default and specify your own postfix string. The file type extension is specified by the **VHDL file extension** option.

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

`'off'`(default)

Write the generated filter VHDL code to a single file.

See Also

`SplitArchFilePostfix`, `SplitEntityFilePostfix`

SplitEntityFilePostfix

Purpose

Specify string to append to specified filter name to form name of file that contains filter's VHDL entity

Settings

`'string'`

The default is `_entity`. This option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.

Usage Notes

This option applies only if you direct the Filter Design HDL Coder to place the filter's entity and architecture in separate files.

See Also

`SplitEntityArch`, `SplitArchFilePostfix`

Purpose

Identify directory into which generated output files are written

Settings

Specify the subdirectory under the current working directory into which generated files are written. The string can specify a complete pathname. The default string is `hdlsrc`.

See Also

`Name`, `VerilogFileExtension`, `VHDLFileExtension`

TargetLanguage

Purpose Specify HDL language to use for generated filter code

Settings `'VHDL'` (default)

Generate VHDL filter code.

`'verilog'`

Generate Verilog filter code.

Purpose

Name VHDL test bench entity or Verilog module and file that contains test bench code

Settings

`'string'`

The file type extension depends on the type of test bench that is being generated.

If the Test Bench Is a...	The Extension Is...
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`. To set the reserved word postfix string, see `ReservedWordPostfix`.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `ForceClockEnable`, `ForceReset`, `HoldTime`, `TestBenchName`

TestBenchStimulus

Purpose

Specify input stimuli that test bench applies to filter

Settings

`'impulse'`

Specify that the test bench acquire an impulse stimulus response. The impulse response is output arising from the unit impulse input sequence defined such that the value of `x(n)` is 1 when `n` equals 1 and `x(n)` equals 0 when `n` does not equal 1.

`'step'`

Specify that the test bench acquire a step stimulus response.

`'ramp'`

Specify that the test bench acquire a ramp stimulus response, which is a constantly increasing or constantly decreasing signal.

`'chirp'`

Specify that the test bench acquire a chirp stimulus response, which is a linear swept-frequency cosine signal.

`'noise'`

Specify that the test bench acquire a white noise stimulus response.

Default settings depend on the structure of the filter.

For Filters...	Default Responses Include...
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`

TestBenchUserStimulus

Purpose

Specify user-defined MATLAB function that returns vector of values that test bench applies to filter

Settings

`M-function`

For example, the following MATLAB function call generates a square wave with a sample frequency of 8 bits per second (Fs/8):

```
repmat([1 1 1 1 0 0 0 0], 1, 10)
```

See Also

`TestBenchStimulus`

Purpose

Specify whether all constants are 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 DOWNTO 0) := (4 DOWNTO 2 => '0', 0 =>'0',
OTHERS => ', '); -- sfix16_En15
CONSTANT coeff2 :signed(15 DOWNTO 0) := (6 => '0', 4 DOWNTO 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 DOWNTO 0) := to_signed(-30, 16); -- sfix16_En15
CONSTANT coeff2 :signed(15 DOWNTO 0) := to_signed(-89, 16); -- sfix16_En15
```

See Also

CastBeforeSum, InlineConfigurations, , LoopUnrolling, SafeZeroConcat, ScaleWarnBits, UseAggregatesForConst, UseRisingEdge, UseVerilogTimescale

UserComment

Purpose

Specify string added as comment line in header of generated filter and test bench files

Settings

'string'

For example, you might use this property to add the revision control tag `$Revision: 1.1.4.7 $` to the header. 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: 1.1.4.10 $
--------------------------------------------------------------
```

Purpose

Specify VHDL coding style used to check for rising edges when operating on registers

Settings

'on'

Use the VHDL rising_edge function to check for rising edges when operating on registers. The generated code applies rising_edge as shown in the following PROCESS block:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
  ELSIF rising_edge(clk) THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
  delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
  END IF;
END PROCESS Delay_Pipeline_Process ;
```

'off' (default)

Check for clock events when operating on registers. The generated code checks for a clock event as shown in the ELSIF statement of the following PROCESS block:

```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
  ELSIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
  delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);
    END IF;
```

```
    END IF;
  END PROCESS Delay_Pipeline_Process ;
```

Usage Notes

The two coding styles have different simulation behavior when the clock transitions from `'X'` to `'1'`.

See Also

`CastBeforeSum`, `InlineConfigurations`, , `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

Purpose

Allow or exclude use of compiler `` `timescale `` directives in generated Verilog code

Settings

`'on'` (default)

Use compiler `` `timescale `` directives in generated Verilog code.

`'off'`

Suppress the use of compiler `` `timescale `` directives in generated Verilog code.

Usage Notes

The `` `timescale `` directive provides a way of specifying different delay values for multiple modules in a Verilog file.

See Also

`CastBeforeSum`, `InlineConfigurations`, , `LoopUnrolling`, `SafeZeroConcat`, `ScaleWarnBits`, `UseAggregatesForConst`, `UseRisingEdge`

VerilogFileExtension

Purpose Specify file type extension for generated Verilog files

Settings `'string'`

The default file type extension for generated Verilog files is `.v`.

See Also `Name`, `TargetDirectory`

Purpose Specify file type extension for generated VHDL files

Settings `'string'`

The default file type extension for generated VHDL files is `.vhd`.

See Also `Name`, `TargetDirectory`

Functions — Alphabetical List

generatehdl

Purpose

Generate HDL code for 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. The defaults are summarized below.

Defaults for Naming, Location and Packaging of Generated Files

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

File	Name
Verilog source	*`Hd`*`.v`, where *`Hd`* is the name of the specified filter object
VHDL source	*`Hd`*`.vhd`, where *`Hd`* is the name of the specified filter object
VHDL package	*`Hd`*`_pkg.vhd`, where *`Hd`* 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.
- Generates script files for third-party EDA tools. Where *`Hd`* is the name of the specified filter object, the following script files are generated:
  - *`Hd`*`_compile.do` : ModelSim compilation script. This script contains commands to compile the generated filter code, but not to simulate it.

- *Hd*_synplify.tcl : Synplify synthesis script

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:

HDL Port	Name
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 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 — By Category" 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. **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
generatehdl(Hd, 'Name', 'MyFilter'); %Generate filter's VHDL code
```

See Also

`generatetb`, `generatetbstimulus`

generatetb

Purpose

Generate HDL test bench for quantized filter

Syntax

```
generatetb(Hd, 'TbType')
generatetb(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:

Specify...	To Generate a Test Bench Consisting of...
`'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.

If the Test Bench Is a...	The Extension Is...
Verilog file	Defined by the property `VerilogFileExtension`
VHDL file	Defined by the property `VHDLFileExtension`
ModelSim `.do` file	`.do`

- Generates script files for third-party EDA tools. Where *Hd* is the name of the specified filter object, the following script files are generated:
  - *Hd*`_tb_compile.do` :ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
  - *Hd*`_tb_sim.do`: ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.
- Forces clock, clock enable, and reset input signals.
- Forces clock enable and reset input to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces reset signals.
- Applies a hold time of 2 nanoseconds to filter reset and data input signals.
- For HDL test benches, applies an error margin of 4 bits.

Default Settings for Files

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

File	Name
Verilog source	*Hd*.v, where *Hd* is the name of the specified filter object
VHDL source	*Hd*.vhd, where *Hd* is the name of the specified filter object
VHDL package	*Hd*_pkg.vhd, where *Hd* 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:

HDL Port	Name
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.

`generatetb(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 — By Category" 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. **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.`

5 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
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

Use this list to find examples in the documentation.

Tutorials

"Basic FIR Filter Tutorial" on page 2-3
"Optimized FIR Filter Tutorial" on page 2-23
"IIR Filter Tutorial" on page 2-44

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 2-17

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 2-44
"Quantizing the IIR Filter" on page 2-46
"Configuring and Generating the IIR Filter's VHDL Code" on page 2-50
"Getting Familiar with the IIR Filter's Generated VHDL Code" on page 2-57
"Verifying the IIR Filter's Generated VHDL Code" on page 2-58

Speed vs. Area Optimizations for FIR Filters

"Specifying Parallel and Serial FIR Architectures in generatehdl" on page 3-65

Index

A

Add input register option 3-45
Add output register option 3-45
AddInputRegister property 6-2
addition operations
    specifying input type treatment for 3-55
    type casting 6-6
AddOutputRegister property 6-3
AddPipelineRegisters property 6-4
advanced coding properties 5-4
Advanced tab 3-46
antisymmetric FIR filters 1-6
application-specific integrated circuits
    (ASICs) 1-2
Architecture options
    for FIR filters 3-61
        cascade serial 3-61
        Distributed arithmetic (DA) 3-71
        fully parallel 3-61
        fully serial 3-61
        partly serial 3-61
architectures
    setting postfix from command line 6-60
    setting postfix from GUI 3-26
asserted level, reset 3-30
    setting 6-51
asynchronous resets
    setting from command line 6-53
    setting from GUI 3-29

B

block labels
    for GENERATE statements 6-5
    for output assignment blocks 6-46
    specifying postfix for 6-5
BlockGenerateLabel property 6-5

C

canonical signed digit (CSD) technique 3-59
cascade filters 3-92
Cascaded Integrator Comb (CIC) filters 3-85
Cast before sum option 3-55
CastBeforeSum property 6-6
checklist
    requirements 3-15
clock
    configuring for test benches 3-99
    specifying high time for 6-9
    specifying low time for 6-12
clock enable input port
    naming 3-42
    specifying forced signals for 6-24
Clock enable port options 3-42
Clock high time option 3-99
clock input port 6-10
    forcing 6-23
    naming 3-42
Clock low time 3-99
Clock port options 3-42
clock process names
    specifying postfix for 6-13
clock time
    configuring 3-99
    high 6-9
    low 6-12
clocked process block labels 3-40
Clocked process postfix option 3-40
ClockEnableInputPort property 6-7
ClockEnableOutputPort property 6-8
ClockHighTime property 6-9
ClockInputPort property 6-10
ClockInputs property 6-11
ClockLowTime property 6-12
ClockProcessPostfix property 6-13
code, generated 3-109
    advanced properties for customizing 5-4
    compiling 4-16

configuring for basic FIR filter tutorial 2-8
configuring for IIR filter tutorial 2-50
configuring for optimized FIR filter tutorial 2-28
customizing 3-32
defaults 3-10
for filter and test bench 4-3
general HDL defaults 3-11
optimizing 3-58
reviewing for basic FIR filter tutorial 2-15
reviewing for IIR filter tutorial 2-57
reviewing for optimized FIR filter tutorial 2-35
verifying for basic FIR filter tutorial 2-17
verifying for IIR filter tutorial 2-58
verifying for optimized FIR filter tutorial 2-37
coefficient multipliers 3-59
Coefficient prefix option 3-35
coefficients
naming 6-15
specifying a prefix for 3-35
CoeffMultipliers property 6-14
CoeffPrefix property 6-15
command line interface 1-6
generating filter and test bench code with 4-3
Comment in header option 3-33
comments, header
as property value 6-70
specifying 3-33
Concatenate type safe zeros 3-53
configurations, inline
suppressing from command line 6-39
suppressing from GUI 3-52
constants
setting representation from command line 6-69
setting representation from GUI 3-48
context-sensitive help 1-12

D

DALUTPartition property 6-17
DARadix property 6-18
data input port
naming from command line 6-40
naming from GUI 3-42
specifying hold time for from GUI 3-103
specifying hold time for with command line 6-37
data output port
specifying name from command line 6-47
specifying name from GUI 3-42
defaults
for general HDL code 3-11
for generated EDA tool scripts 3-11
for generated files 3-10
for optimizations 3-13
for resets 3-11
for test benches 3-13
demos 1-13
dialog box
HDL Options 3-32
dialogs
Generate HDL
description 1-4
opening 3-5
setting cascade filter options with 3-92
setting multirate filter options with 3-85
setting optimizations with 3-57
setting test bench options with 3-95
specifying test bench type with 3-97
Test Bench Options 3-95
Direct Form I filters 1-6
Direct Form II filters 1-6
directory, target 6-63

E

EDAScriptGeneration property 6-19

Electronic Design Automation (EDA) tool
scripts
defaults for generation of 3-11
Electronic Design Automation (EDA) tools
generation of scripts for 3-110
entities
name conflicts of 3-36
naming 6-44
setting names of 3-23
setting postfix from command line 6-62
setting postfix from GUI 3-26
Entity conflict postfix option 3-36
entity name conflicts 6-20
`EntityConflictPostfix` property 6-20
error margin
specifying from command line 6-21
specifying from GUI 3-104
Error margin (bits) option 3-104
`ErrorMargin` property 6-21

F

factored CSD technique 3-59
FDATool 1-4
`fdatool` command 3-5
field programmable gate arrays (FPGAs) 1-2
file extensions
setting 3-23
Verilog 6-74
VHDL 6-75
file location properties 5-2
file names
for architectures 6-60
for entities 6-62
file naming properties 5-2
filenames
defaults 3-10
for generated output 1-9
files, generated
default names 3-10
defaults 3-10
HDL output 1-9
setting architecture postfix for 3-26
setting entity postfix for 3-26
setting location of 3-24
setting names of 3-23
setting options for 3-22
setting package postfix for 3-25
splitting 6-61
test bench 6-65
filter arithmetic 3-5
Filter Design HDL Coder
applying to hardware design process 1-14
as FDATool plug-in 1-4
command line interface 1-6
features of 1-3
graphical user interface (GUI) 1-4
prerequisite knowledge for 1-3
user profiles for 1-3
what is 1-2
workflow 1-14
filter input 6-57
filter structures 1-6
Filter target language option 3-21
filters
designing in basic FIR tutorial 2-3
designing in IIR filter tutorial 2-44
designing in optimized FIR filter tutorial 2-23
generated HDL output for 1-9
naming generated file for 6-44
properties of 1-8
quantized 1-6
quantizing 3-5
realizations of 1-6
finite impulse response (FIR) filters 1-6
FIR adder style option 3-60
FIR filter architectures
serial 6-55
serial-cascade 6-55

FIR filter architectures property
partly serial 6-58
serial 6-58
FIR filter tutorial
basic 2-3
optimized 2-23
FIR filters 1-6
optimizing clock rate for 3-81
optimizing final summation for 3-60
specifying summation technique for 6-22
`FIRAdderStyle` property 6-22
Force clock enable option 3-99
Force clock option 3-99
force reset hold time 6-37
Force reset option 3-101
`ForceClock` property 6-23
`ForceClockEnable` property 6-24
`ForceReset` property 6-25
FPGAs (field programmable gate arrays) 1-2
functions
`generatehdl` 7-2
`generatetb` 7-6
`generatetbstimulus` 7-12
input parameters for 1-8

G

General tab 3-35
Generate HDL dialog box
defaults 3-10
description 1-4
opening 3-5
setting optimizations with 3-57
specifying test bench type with 3-97
`generatehdl` function 7-2
`generatetb` function 7-6
`generatetbstimulus` function 7-12
graphical user interface (GUI) 1-4

H

hardware description languages (HDLs) 1-2
See also Verilog; VHDL
HDL code 2-8
See also code, generated
HDL files 1-9
HDL language 3-21
HDL Options dialog box 3-32
HDL test benches 4-3
`HDLCompileInit` property 6-26
`HDLCompileTerm` property 6-27
`HDLCompileVerilogCmd` property 6-28
`HDLCompileVHDLCmd` property 6-29
HDLs (hardware description languages) 1-2
See also Verilog; VHDL
`HDLSimCmd` property 6-30
`HDLSimInit` property 6-31
`HDLSimTerm` property 6-32
`HDLSimViewWaveCmd` property 6-36
`HDLSynthCmd` property 6-33
`HDLSynthInit` property 6-34
`HDLSynthTerm` property 6-35
header comment properties 5-3
header comments 3-33
help
context-sensitive 1-12
getting 1-11
Help browser 1-12
hold time 6-37
for data input signals 3-103
for resets 3-101
`HoldTime` property 6-37

I

IIR filter tutorial 2-44
IIR filters 1-6
optimizing clock rate for 3-81
infinite impulse response (IIR) filters 1-6
inline configurations

specifying 6-39
suppressing the generation of 3-52
Inline VHDL configurations option 3-52
InlineConfigurations property 6-39
input data overlay with scale values 3-46
Input data type option 3-43
input parameters 1-8
Input port option 3-42
input ports
naming 3-42
specifying data type for 6-41
input registers
adding code for 6-2
suppressing generation of extra 3-45
InputPort property 6-40
InputType property 6-41
installation 1-10
instance sections 6-42
InstanceGenerateLabel property 6-42

L

labels
block 6-46
process block 3-40
language
setting target 3-21
target 6-64
language selection properties 5-2
linear FIR final summation 3-60
Loop unrolling option 3-49
loops
unrolling 6-43
unrolling and removing 3-49
LoopUnrolling property 6-43

M

M-help 1-12
Minimum overlap of scale values option 3-46
ModelSim 4-16
ModelSim .do file
executing 4-17
test benches 3-97
testing with 4-12
modules
name conflicts for 3-36
naming 6-44
setting names of 3-23
multipliers
optimizing coefficient 3-59
multirate filters
Cascaded Integrator Comb (CIC) 3-85
clock enable output for 6-8
clock inputs for 6-11
code generation for 3-85
decimating 3-85
Direct-Form Transposed FIR Polyphase Decimator 3-85
interpolating 3-85
types supported for code generation 3-85

N

name conflicts 6-20
Name option 3-23
Name property 6-44
names
clock process 6-13
coefficient 3-35
package file 6-49
naming properties 5-3

O

optimization properties 5-6
optimizations
defaults for 3-13
for synthesis 3-83
HDL code 3-58

setting 3-57
Optimize for HDL option 3-58
optimized FIR filter tutorial 2-23
`OptimizeForHDL` property 6-45
options
Add input register 3-45
Add output register 3-45
Add pipeline registers 3-81
Architecture 3-61
Cast before sum 3-55
Clock enable input port 3-42
Clock high time 3-99
Clock low time 3-99
Clock port 3-42
Clocked process postfix 3-40
Coeff multipliers 3-59
Coefficient prefix 3-35
Comment in header 3-33
Concatenate type safe zeros 3-53
DA radix 3-71
Entity conflict postfix 3-36
Error margin (bits) 3-104
Filter target language 3-21
FIR adder style 3-60
Force clock 3-99
Force clock enable 3-99
Force reset 3-101
Hold time 3-103
Inline VHDL configurations 3-52
Input data type 3-43
Input port 3-42
Loop unrolling 3-49
LUT partition 3-71
Minimum overlap of scale values 3-46
Optimize for HDL 3-58
Output data type 3-43
Output port 3-42
Package postfix 3-25
Represent constant values by aggregates 3-48
Reserved word postfix 3-37
Reset asserted level 3-30
Reset port 3-42
Reset type 3-29
Serial Partition 3-61
Split arch. file postfix 3-26
Split entity and architecture 3-26
Split entity file postfix 3-26
Target directory
for test bench output 3-95
redirecting output with 3-24
Use ’rising_edge’ for registers 3-50
Use Verilog `timescale directives 3-54
User defined response 3-106
Verilog file extension
setting file extension with 3-23
Verilog file extension option
renaming test bench file with 3-95
VHDL file extension
renaming test bench file with 3-95
setting file extension with 3-23
output
generated HDL 1-9
redirecting 3-24
Output data type option 3-43
Output port option 3-42
output ports
naming 3-42
specifying data type for 6-48
output registers
adding code for 6-3
suppressing generation of extra 3-45
`OutputGenerateLabel` property 6-46
`OutputPort` property 6-47
`OutputType` property 6-48

P

package files
default name for 3-10

specifying postfix for 6-49
Package postfix option 3-25
PackagePostfix property 6-49
packages
setting names of 3-23
setting postfix 3-25
parameters 1-8
pipeline registers
using from command line 6-4
using from GUI 3-81
pipelined FIR final summation 3-60
port data types 3-43
port properties 5-3
ports
clock enable input 6-7
clock input 6-10
data input 6-40
data output 6-47
input 6-41
naming 3-42
output 6-48
reset input 6-52
Ports tab 3-42
process block labels 3-40
properties
AddInputRegister 6-2
AddOutputRegister 6-3
AddPipelineRegisters 6-4
advanced coding 5-4
as input parameters 1-8
BlockGenerateLabel 6-5
CastBeforeSum 6-6
ClockEnableInputPort 6-7
ClockEnableOutputPort 6-8
ClockHighTime 6-9
ClockInputPort 6-10
ClockInputs 6-11
ClockLowTime 6-12
ClockProcessPostfix 6-13
coding 5-4
CoeffMultipliers 6-14
CoeffPrefix 6-15
DALUTPartition 6-17
DARadix 6-18
EDAScriptGeneration 6-19
EntityConflictPostfix 6-20
ErrorMargin 6-21
file location 5-2
file naming 5-2
FIRAdderStyle 6-22
ForceClock 6-23
ForceClockEnable 6-24
ForceReset 6-25
HDLCompileInit 6-26
HDLCompileTerm 6-27
HDLCompileVerilogCmd 6-28
HDLCompileVHDLCmd 6-29
HDLSimCmd 6-30
HDLSimInit 6-31
HDLSimTerm 6-32
HDLSimViewWaveCmd 6-36
HDLSynthCmd 6-33
HDLSynthInit 6-34
HDLSynthTerm 6-35
header comment 5-3
HoldTime 6-37
InlineConfigurations 6-39
InputPort 6-40
InputType 6-41
InstanceGenerateLabel 6-42
language selection 5-2
LoopUnrolling 6-43
Name 6-44
naming 5-3
optimization 5-6
OptimizeForHDL 6-45
OutputGenerateLabel 6-46
OutputPort 6-47
OutputType 6-48
PackagePostfix 6-49

port 5-3
ReservedWordPostfix 6-50
reset 5-2
ResetAssertedLevel 6-51
ResetInputPort 6-52
ResetType 6-53
ReuseAccum 6-55
SafeZeroConcat 6-56
ScaleWarnBits 6-57
script generation 5-7
SerialPartition 6-58
SimulatorFlags 6-59
SplitArchFilePostfix 6-60
SplitEntityArch 6-61
SplitEntityFilePostfix 6-62
TargetDirectory 6-63
TargetLanguage 6-64
test bench 5-6
TestBenchName 6-65
TestBenchStimulus 6-66
TestBenchUserStimulus 6-68
UseAggregatesForConst 6-69
UserComment 6-70
UseRisingEdge 6-71
UseVerilogTimescale 6-73
VerilogFileExtension 6-74
VHDLFileExtension 6-75

Q

quantization noise 3-46
quantized filters 1-6

R

registers
adding code for input 6-2
adding code for output 6-3
adding for optimization 6-4
pipeline 3-81
Represent constant values by aggregates option 3-48
requirements
identifying for HDL code and test benches 3-15
product 1-10
Reserved word postfix option 3-37
reserved words
setting postfix for resolution of 3-37
specifying postfix for 6-50
ReservedWordPostfix property 6-50
Reset asserted level option 3-30
reset input port 6-52
naming 3-42
Reset port options 3-42
reset properties 5-2
Reset type option 3-29
ResetAssertedLevel property 6-51
ResetInputPort property 6-52
resets
configuring for test benches 3-101
customizing 3-29
defaults for 3-11
setting asserted level for from command line 6-51
setting asserted level for from GUI 3-30
setting style of 3-29
specifying forced 6-25
types of 6-53
ResetType property 6-53
ReuseAccum property 6-55
rising_edge function 3-50

S

SafeZeroConcat property 6-56
scale values 3-46
ScaleWarnBits property 6-57
script generation properties 5-7
second-order section (SOS) filters 1-6

sections
  instance 6-42
SerialPartition property 6-58
simulator 4-7
SimulatorFlags property 6-59
SOS filters 1-6
Split arch. file postfix option 3-26
Split entity and architecture option 3-26
Split entity file postfix option 3-26
SplitArchFilePostfix property 6-60
SplitEntityArch property 6-61
SplitEntityFilePostfix property 6-62
stimulus
  setting for test benches 3-106
  specifying 6-66
  specifying user-defined 6-68
subtraction operations
  specifying input type treatment for 3-55
  type casting 6-6
summation technique 6-22
symmetric FIR filters 1-6
synchronous resets
  setting from command line 6-53
  setting from GUI 3-29
synthesis 3-83

T

Target directory option
  redirecting output with 3-24
  renaming test bench file with 3-95
target language 3-21
TargetDirectory property 6-63
TargetLanguage property 6-64
test bench
  generation of 7-6
test bench files 3-10
test bench properties 5-6
test benches
  compiling 4-7
  configuring clock for 3-99
  configuring resets for 3-101
  customizing 3-95
  defaults for 3-13
  error margin for 6-21
  generated HDL output for 1-9
  generating .do file 4-12
  HDL 4-3
  naming 6-65
  renaming 3-95
  running 4-8
  setting error margin for 3-104
  setting input data hold time 3-103
  setting names of 3-23
  setting stimuli for 3-106
  specifying clock enable input for 6-24
  specifying forced clock input for 6-23
  specifying forced resets for 6-25
  specifying stimulus for 6-66
  specifying type of 3-97
  specifying user-defined stimulus for 6-68
test methods 4-2
TestBenchName property 6-65
TestBenchStimulus property 6-66
TestBenchUserStimulus property 6-68
time
  clock high 6-9
  clock low 6-12
  hold 6-37
timescale directives
  specifying use of 6-73
  suppressing 3-54
transposed Direct Form I filters 1-6
transposed Direct Form II filters 1-6
transposed FIR filters 1-6
tree FIR final summation 3-60
tutorial files 2-2
tutorials 1-13
  basic FIR filter 2-3
  IIR filter 2-44

optimized FIR filter 2-23
type casting 6-6
for addition and subtraction operations 3-55

U

Use 'rising_edge' for registers option 3-50
Use Verilog `timescale directives option 3-54
UseAggregatesForConst property 6-69
User defined response option 3-106
UserComment property 6-70
UseRisingEdge property 6-71
UseVerilogTimescale property 6-73

V

Verilog 1-2
file extension 6-74
selecting 3-21
Verilog file extension option
naming filter file with 3-23
renaming test bench file with 3-95
Verilog reserved words 3-37
Verilog test benches 3-97
VerilogFileExtension property 6-74
VHDL 1-2
file extension 6-75
selecting 3-21
VHDL file extension option
naming filter file with 3-23
renaming test bench file with 3-95
VHDL reserved words 3-37
VHDL test benches 3-97
VHDLFileExtension property 6-75

Z

zeros
concatenated 3-53