Filter Design HDL Coder™

User's Guide

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Filter Design HDL Coder™ User's Guide

Revision History

June 2004	Online only	New for Version 1.0 (Release 14)
October 2004	Online only	Revised for Version 1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.2 (Release 14SP2)
September 2005	Online only	Revised for Version 1.3 (Release 14SP3)
March 2006	Online only	Revised for Version 1.4 (Release 2006a)
September 2006	Online only	Revised for Version 1.5 (Release 2006b)
March 2007	Online only	Revised for Version 2.0 (Release 2007a)
September 2007	Online only	Revised for Version 2.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.2 (Release 2008a)
October 2008	Online only	Revised for Version 2.3 (Release 2008b)
March 2009	Online only	Revised for Version 2.4 (Release 2009a)
September 2009	Online only	Revised for Version 2.5 (Release 2009b)
March 2010	Online only	Revised for Version 2.6 (Release 2010a)
September 2010	Online only	Revised for Version 2.7 (Release 2010b)
April 2011	Online only	Revised for Version 2.8 (Release 2011a)
September 2011	Online only	Revised for Version 2.9 (Release 2011b)
March 2012	Online only	Revised for Version 2.9.1 (Release 2012a)
September 2012	Online only	Revised for Version 2.9.2 (Release 2012b)
March 2013	Online only	Revised for Version 2.9.3 (Release 2013a)
September 2013	Online only	Revised for Version 2.9.4 (Release 2013b)
March 2014	Online only	Revised for Version 2.9.5 (Release 2014a)
October 2014	Online only	Revised for Version 2.9.6 (Release 2014b)
March 2015	Online only	Revised for Version 2.9.7 (Release 2015a)
September 2015	Online only	Revised for Version 2.10 (Release 2015b)
March 2016	Online only	Revised for Version 3.0 (Release 2016a)
September 2016	Online only	Revised for Version 3.1 (Release 2016b)
March 2017	Online only	Revised for Version 3.1.1 (Release 2017a)
September 2017	Online only	Revised for Version 3.1.2 (Release 2017b)
March 2018	Online only	Revised for Version 3.1.3 (Release 2018a)
September 2018	Online only	Revised for Version 3.1.4 (Release 2018b)
March 2019	Online only	Revised for Version 3.1.5 (Release 2019a)
September 2019	Online only	Revised for Version 3.1.6 (Release 2019b)
March 2020	Online only	Revised for Version 3.1.7 (Release 2020a)
September 2020	Online only	Revised for Version 3.1.8 (Release 2020b)
March 2021	Online only	Revised for Version 3.1.9 (Release 2021a)
September 2021	Online only	Revised for Version 3.1.10 (Release 2021b)
March 2022	Online only	Revised for Version 3.1.11 (Release 2022a)
September 2022	Online only	Revised for Version 3.1.12 (Release 2022b)
March 2023	Online only	Revised for Version 3.1.13 (Release 2023a)

Contents

1 Getting Started

2 HDL Filter Code Generation Fundamentals

3 HDL Code for Supported Filter Structures

4 Optimization of HDL Filter Code

5 Customization of HDL Filter Code

6 Verification of Generated HDL Filter Code

7 Synthesis and Workflow Automation

8 Filter Design HDL Coder Featured Examples

9 Properties

10 Functions

Getting Started

Filter Design HDL Coder Product Description

Generate HDL code for fixed-point filters

Filter Design HDL Coder generates synthesizable, portable VHDL® and Verilog® code for implementing fixed-point filters designed with MATLAB® on FPGAs or ASICs. It automatically creates VHDL and Verilog test benches for simulating, testing, and verifying the generated code.

Key Features

- Generation of synthesizable IEEE® 1076 compliant VHDL code and IEEE 1364-2001 compliant Verilog code
- Control over generated code content, optimization, and style
- Distributed arithmetic and other options for speed vs. area tradeoff and architecture exploration
- VHDL and Verilog test-bench generation for quick verification and validation of generated HDL filter code
- Simulation and synthesis script generation

Automated HDL Code Generation

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

Traditionally, system designers and hardware developers use hardware description languages (HDLs), such as VHDL and Verilog, to develop hardware filter designs. HDLs provide a proven method for hardware design, but coding filter designs is labor-intensive. Also, algorithms and system-level designs created using HDLs are difficult to analyze, explore, and share.

The Filter Design HDL Coder workflow automates the implementation of designs in HDL. First, using DSP System Toolbox™ features (apps, filter System objects), an architect or designer develops a filter algorithm targeted for the hardware. Then, using the Generate HDL dialog box (`fdhdltool`) or command-line tool (`generatehdl`) of Filter Design HDL Coder, a designer configures code generation options and generates a VHDL or Verilog implementation of the design. Designers can easily modify these designs and share them between teams, in HDL or MATLAB formats.

The generated HDL code adheres to a clean, readable coding style. The optional generated HDL test bench confirms that the generated code behaves as expected, and can accelerate system-level test bench implementation. Designers can also use Filter Design HDL Coder software to generate test signals automatically and validate models against standard reference designs.

This workflow enables designers to fine-tune algorithms and models through rapid prototyping and experimentation, while spending less time on HDL implementation.

See Also

`generatehdl` | `fdhdltool` | `filterBuilder` | `filterDesigner`

Related Examples

- “Starting Filter Design HDL Coder” on page 2-2
- “Generating HDL Code” on page 2-12
- “Generate HDL Code for Filter System Objects” on page 2-16

Supported Filter System Objects

Filter Design HDL Coder supports these System objects from DSP System Toolbox.

Single Rate Filters

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`
- `dsp.FilterCascade`
- `dsp.VariableFractionalDelay`

Multirate Filters

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FarrowRateConverter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FilterCascade`
- `dsp.DigitalDownConverter`
- `dsp.DigitalUpConverter`

You can also model hardware behavior, and generate HDL code by using System objects or Simulink® blocks from DSP HDL Toolbox™. These objects and blocks include hardware-friendly control signals and architecture options. To generate HDL code from “DSP HDL Toolbox” objects and blocks, you must also have the HDL Coder™ product. See “HDL-Optimized Filters and Transforms” (DSP HDL Toolbox).

See Also

`generatehdl` | `fdhdltool`

Related Examples

- “Generate HDL Code for Filter System Objects” on page 2-16

Basic FIR Filter

In this section...

Note The Filter Design HDL Coder product will be discontinued in a future release. Instead, you can model hardware behavior, and generate HDL code by using System objects or Simulink blocks from DSP HDL Toolbox. These objects and blocks include hardware-friendly control signals and architecture options. To generate HDL code from "DSP HDL Toolbox" objects and blocks, you must also have the HDL Coder product.

For examples of modeling an FIR filter for hardware, see "Fully Parallel Systolic FIR Filter Implementation" (DSP HDL Toolbox) and "Partly Serial Systolic FIR Filter Implementation" (DSP HDL Toolbox). These examples use the Discrete FIR Filter block. Equivalent functionality is also available in the `dsphdl.FIRFilter` System object™.

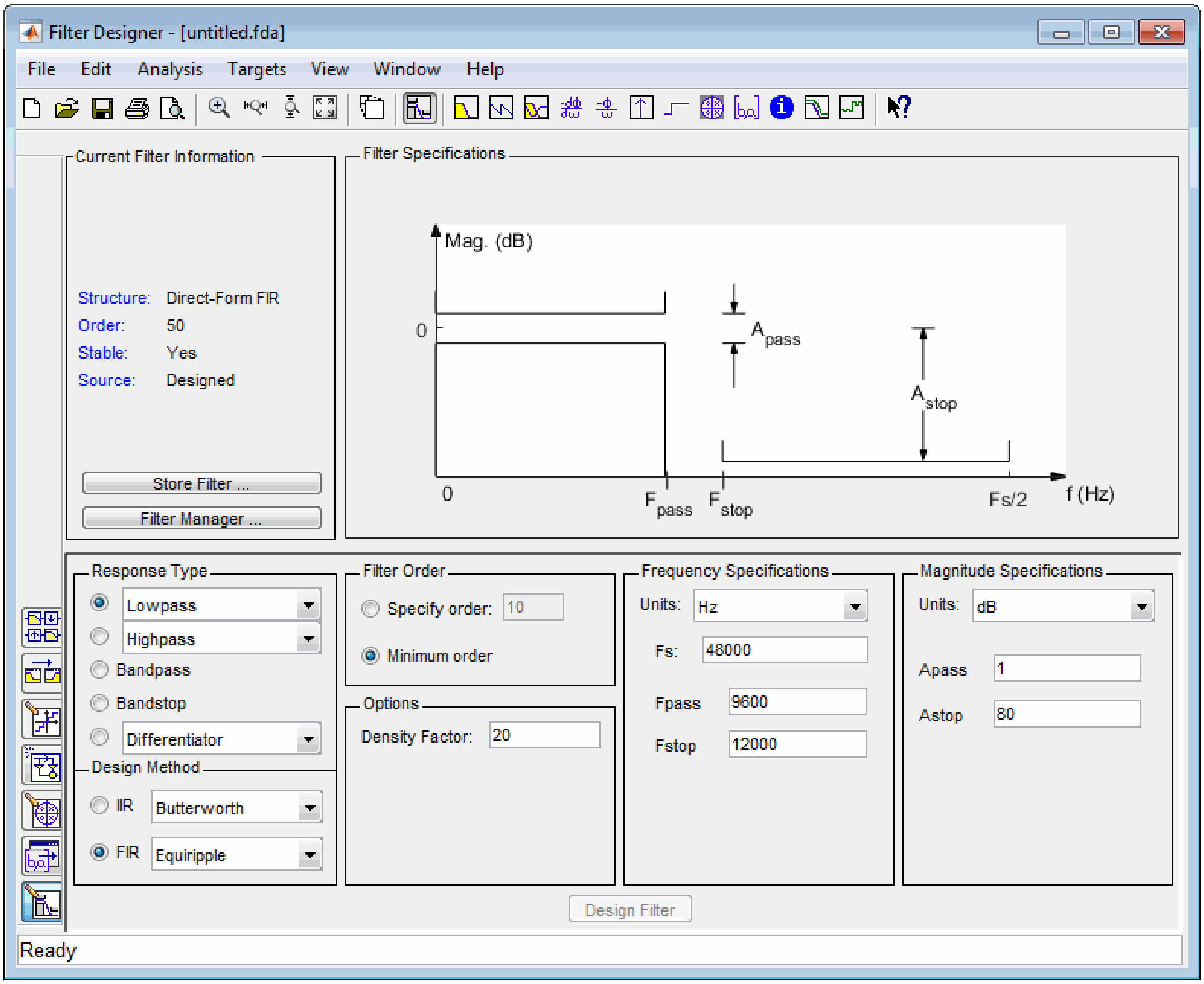
Create a Folder for Your Tutorial Files

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

Design a FIR Filter in Filter Designer

This section assumes that you are familiar with the MATLAB user interface and the Filter Designer. These instructions guide you through designing and creating a basic FIR filter using Filter Designer.

1. Start the MATLAB software.
2. Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-5.
3. Start Filter Designer by entering the `filterDesigner` command in the MATLAB Command Window. The Filter Design & Analysis Tool appears.



4 In the Filter Design & Analysis Tool, check that these 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`

Option	Value
Magnitude Specifications	**Units**: `dB` **Apass**: `1` **Astop**: `80`

These settings are for the default filter design that the Filter Designer creates for you. If you do not have to change the filter, and **Design Filter** is grayed out, you are done and can skip to “Quantize the Filter” on page 1-7.

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

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

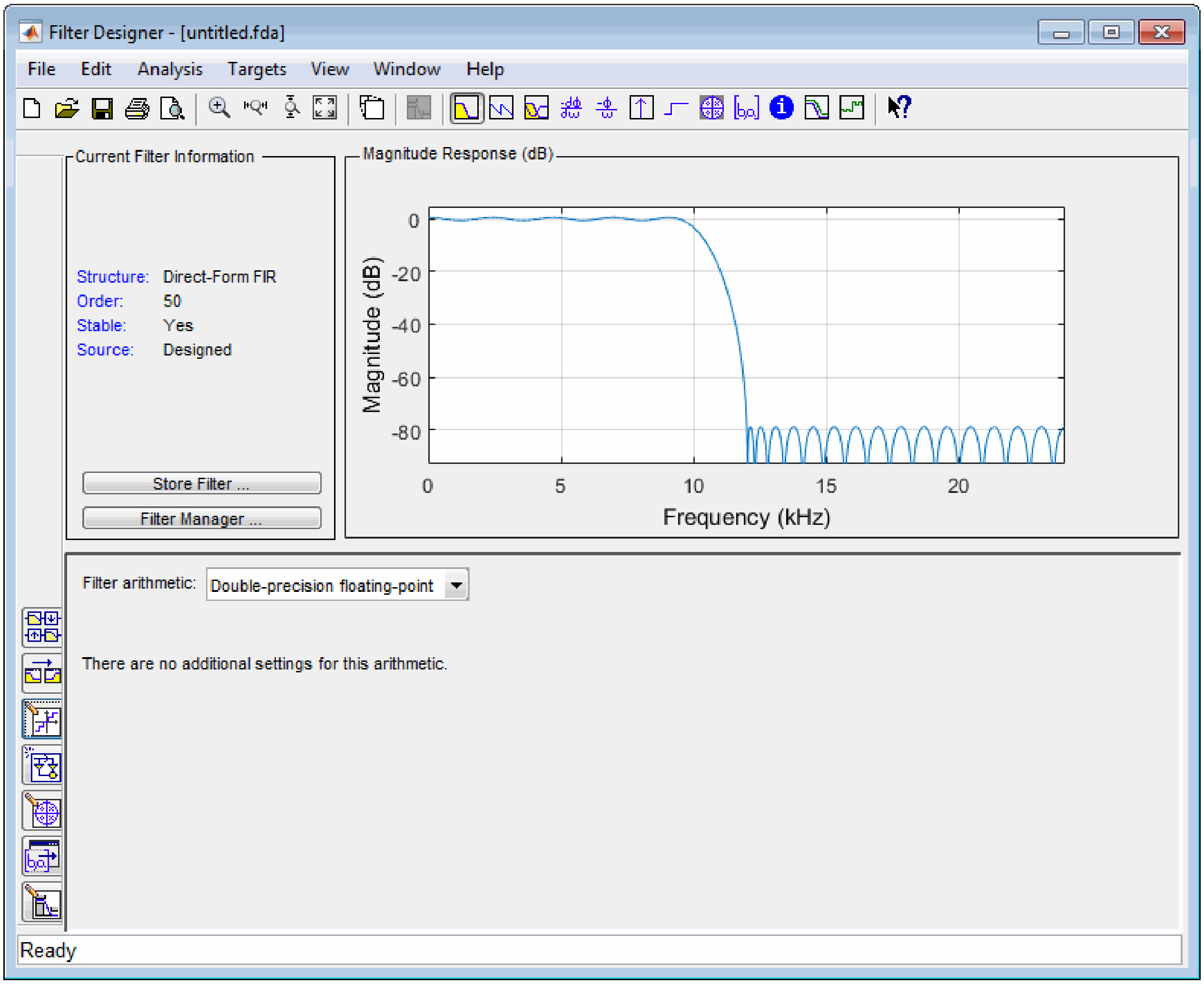
For more information on designing filters with the Filter Designer, see the DSP System Toolbox documentation.

Quantize the Filter

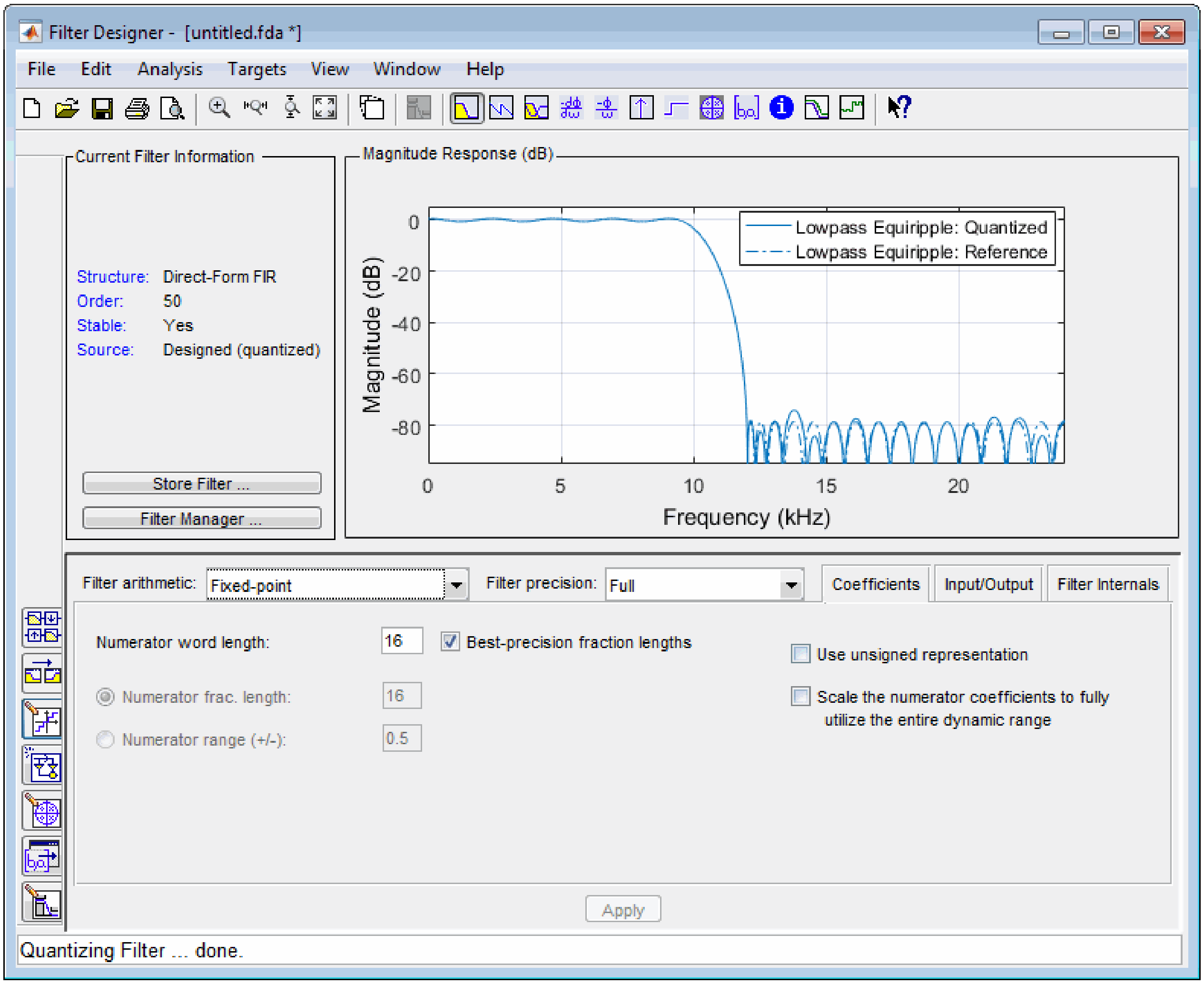
You must quantize filters for HDL code generation. To quantize your filter,

1 Open the basic FIR filter design you created in “Design a FIR Filter in Filter Designer” on page 1-5.

2 Click the Set Quantization Parameters button  in the left-side toolbar. The Filter Designer displays a **Filter arithmetic** menu in the bottom half of the window.



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



Use the quantization options to test the effects of various settings on the performance and accuracy of the quantized filter.

Set the quantization parameters as follows:

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

Tab	Parameter	Setting
	Accum. word length	40

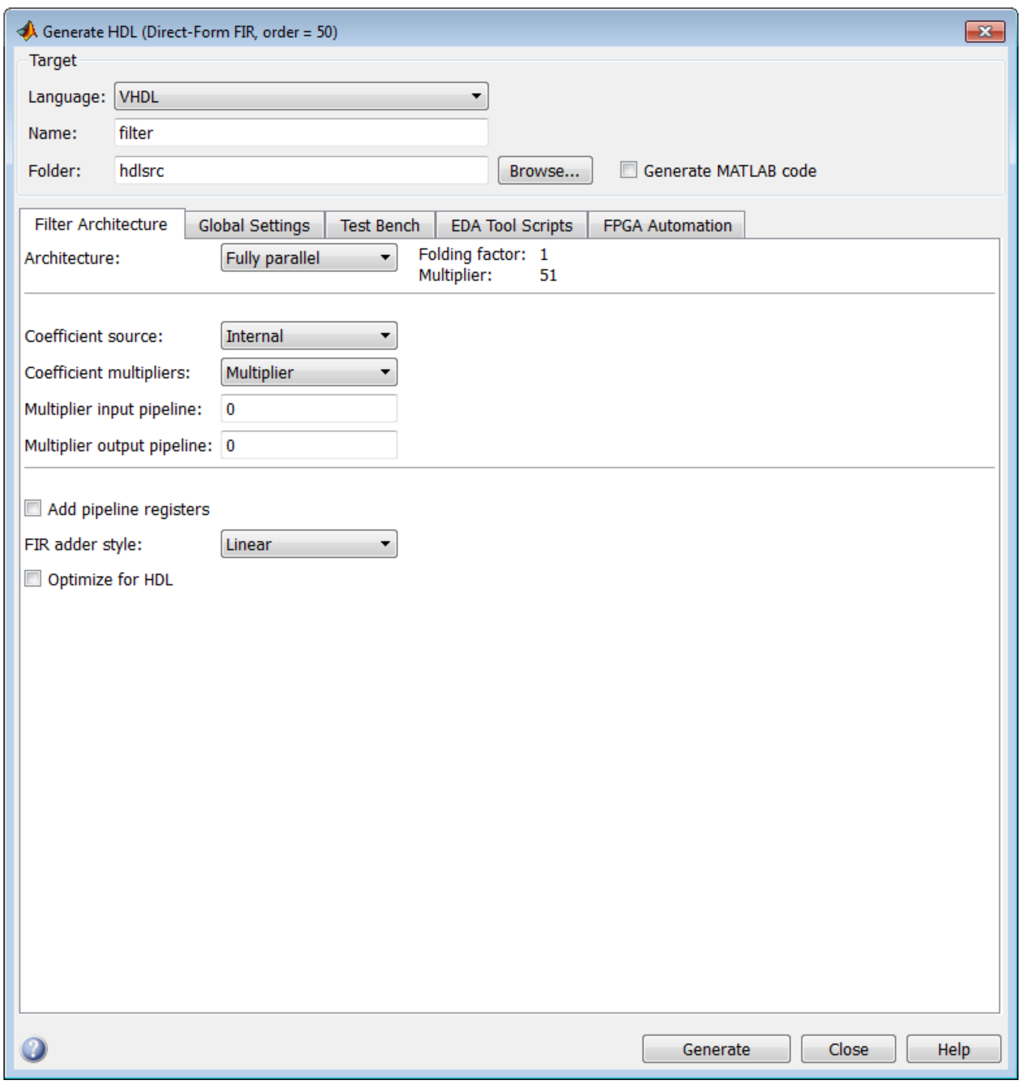
4 Click **Apply**.

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

Configure and Generate VHDL Code

After you quantize your filter, you are ready to configure coder options and generate VHDL code for the filter. This section guides you through starting the Filter Design HDL Coder UI, setting options, and generating the VHDL code and test bench for the basic FIR filter you designed and quantized in “Design a FIR Filter in Filter Designer” on page 1-5 and “Quantize the Filter” on page 1-7.

1 Start the Filter Design HDL Coder UI by selecting **Targets** > **Generate HDL** in the Filter Designer tool. The Filter Designer displays the Generate HDL tool.



2 Find the Filter Design HDL Coder online help.

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

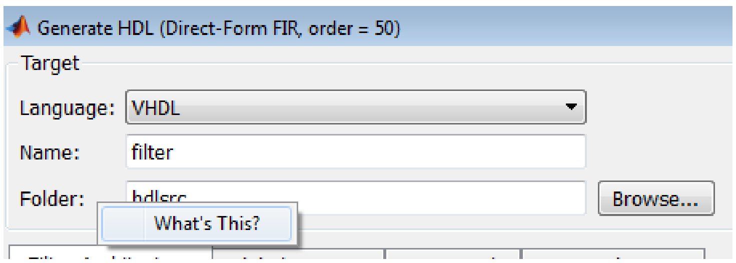
   b In the **Contents** pane of the **Help** browser, select the **Filter Design HDL Coder** entry.

   c Minimize the **Help** browser.

3 In the Generate HDL tool, click the **Help** button. A small context-sensitive help window opens. The window displays information about the tool.

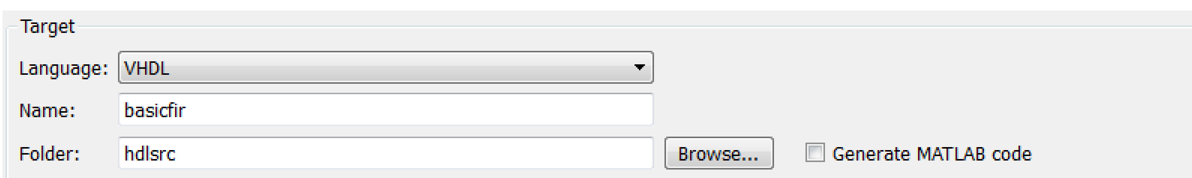
4 Close the **Help** window.

5 Place your cursor over the **Folder** label or text box in the **Target** pane of the Generate HDL tool, and right-click. A **What's This?** button appears.

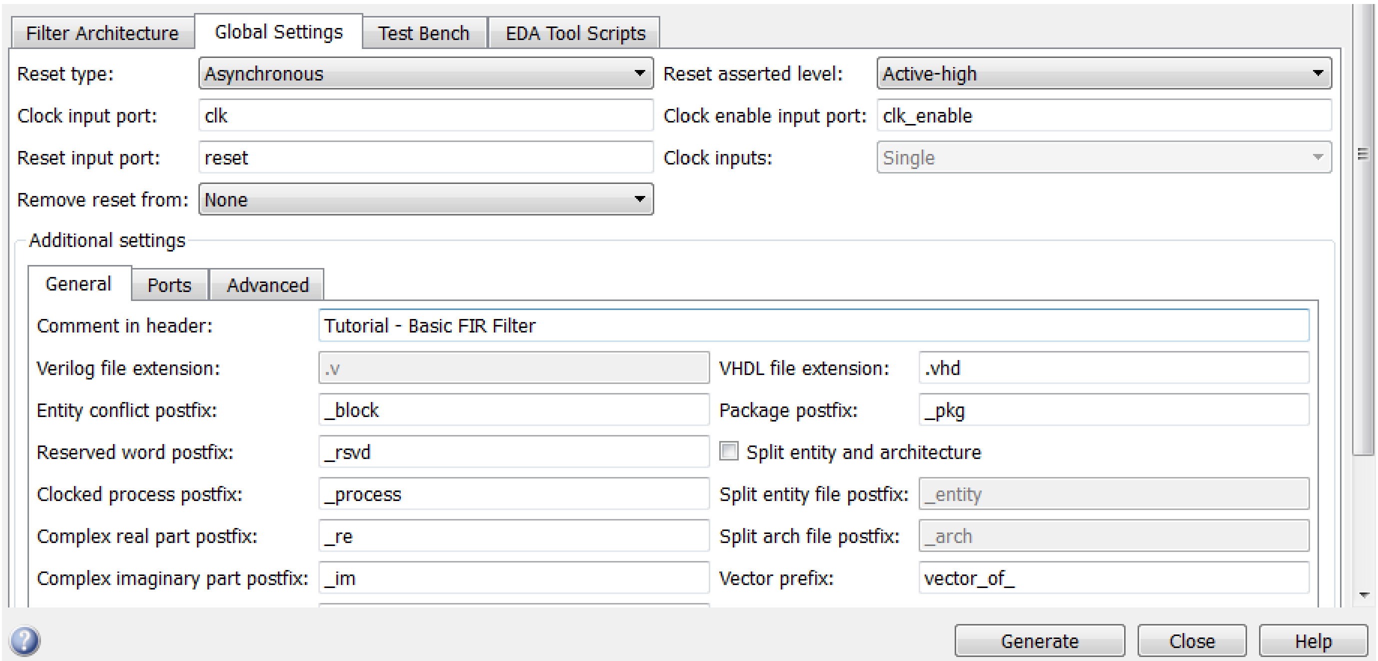


6 Click **What's This?** The context-sensitive help window displays information describing the **Folder** option. Configure the contents and style of the generated HDL code, using the context-sensitive help to get more information as you work. A help topic is available for each option.

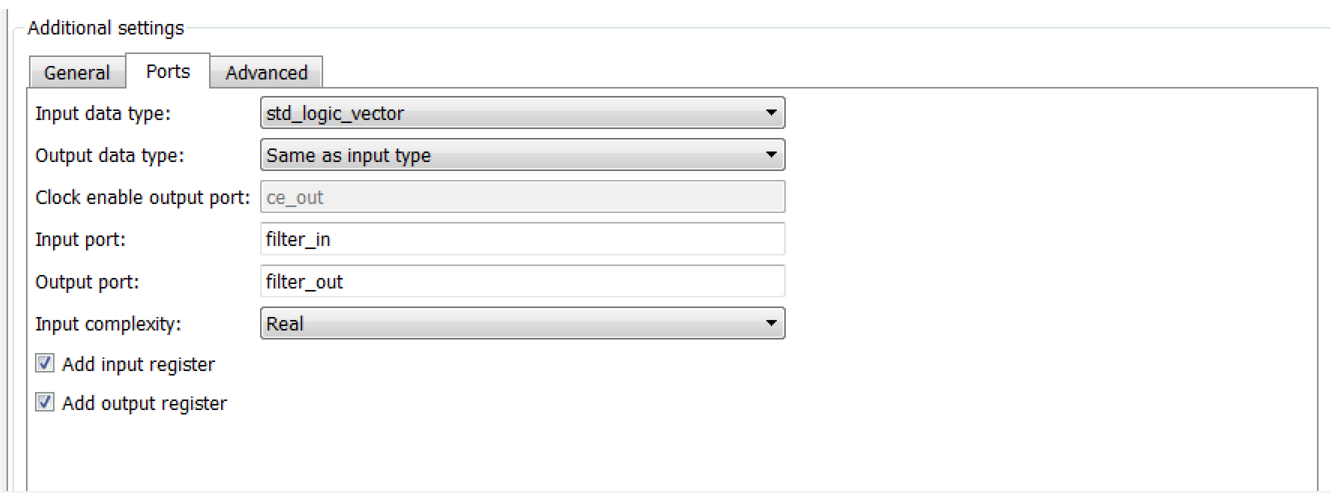
7 In the **Name** text box of the **Target** pane, replace the default name with `basicfir`. This option names the VHDL entity and the file that contains the VHDL code for the filter.



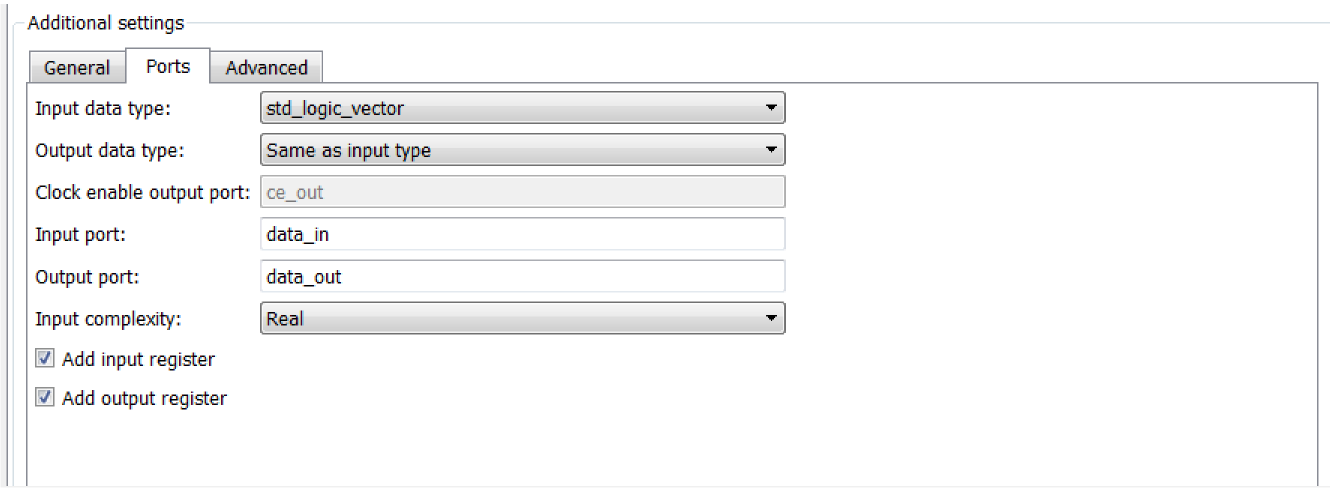
8 Select the **Global settings** tab of the UI. Then select the **General** tab of the **Additional settings** section of the UI. Type `Tutorial - Basic FIR Filter` in the **Comment in header** text box. The coder adds the comment to the end of the header comment block in each generated file.



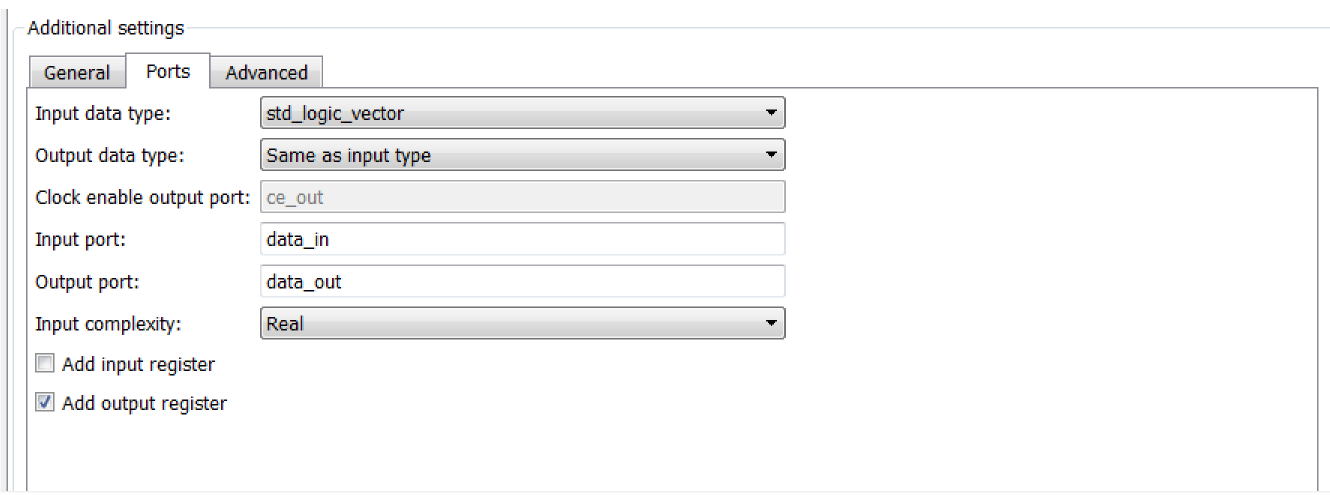
9 Select the **Ports** tab of the **Additional settings** section of the UI.



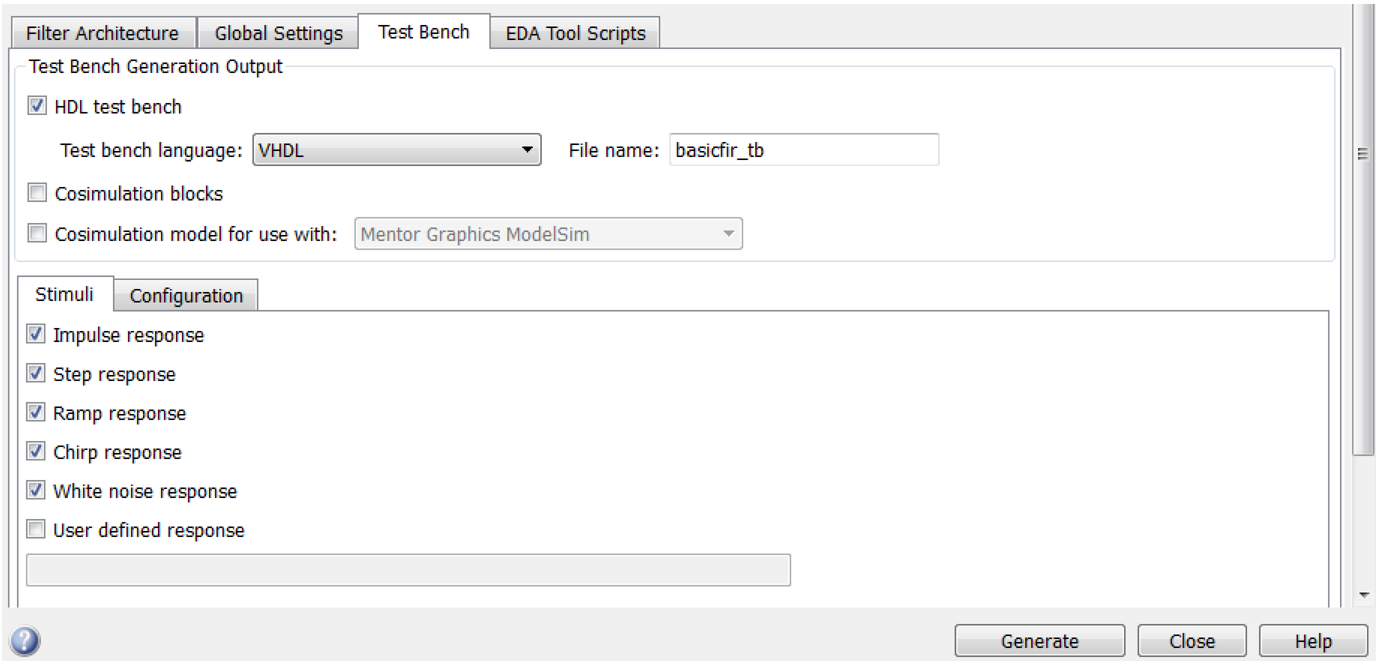
10 Change the names of the input and output ports. In the **Input port** text box, replace `filter_in` with `data_in`. In the **Output port** text box, replace `filter_out` with `data_out`.



11 Clear the check box for the **Add input register** option. The **Ports** pane now looks like this figure.



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



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

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

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

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

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

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

The generated VHDL code has these characteristics:

- VHDL entity named `basicfir`.
- Registers that use asynchronous resets when the reset signal is active high (1).
- The table shows the names of the ports.

VHDL Port	Name
Input	`data_in`
Output	`data_out`
Clock input	`clk`

VHDL Port	Name
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 `coeff`*n*, where *n* is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'`...
- Registers are generated with the statement `ELSIF clk'event AND clk='1' THEN` rather than with the `rising_edge` function.
- The postfix `'_process'` is appended to process names.

The generated test bench:

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

14. When you have finished generating code, click **Close** to close the Generate HDL tool.

Explore the Generated VHDL Code

Get familiar with the 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 value you specified for the **Name** option in the **Target** pane. See step 5 in “Configure and Generate VHDL Code” on page 1-10.
3. Search for `Tutorial`. This section is where the coder places the text you entered for the **Comment in header** option. See step 10 in “Configure and Generate VHDL Code” on page 1-10.
4. Search for `HDL Code`. This section lists coder options you modified in “Configure and Generate VHDL Code” on page 1-10.
5. Search for `Filter Settings`. This section describes the filter design and quantization settings as you specified in “Design a FIR Filter in Filter Designer” on page 1-5 and “Quantize the Filter” on page 1-7.
6. Search for `ENTITY`. This line names the VHDL entity, using the value you specified for the **Name** option in the **Target** pane. See step 5 in “Configure and Generate VHDL Code” on page 1-10.
7. Search for `PORT`. This `PORT` declaration defines the clock, clock enable, reset, and data input and output ports. The ports for clock, clock enable, and reset signals are named with default character vectors. The ports for data input and output are named as you specified on the **Input port** and **Output port** options on the **Ports** tab of the Generate HDL tool. See step 12 in “Configure and Generate VHDL Code” on page 1-10.

8 Search for `Constants`. This section defines the coefficients. They are named using the default naming scheme, `coeffn`, where *n* is the coefficient number, starting with 1.

9 Search for `Signals`. This section of code defines the signals for the filter.

10 Search for `process`. The `PROCESS` block name `Delay_Pipeline_process` includes the default `PROCESS` block postfix `'_process'`.

11 Search for `IF reset`. This code asserts the reset signal. 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 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 section of code writes the filter data to an output register. Code for this register is generated by default. In step 13 in "Configure and Generate VHDL Code" on page 1-10, 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 `'_process'`.

14 Search for `data_out`. This section of code drives the output data of the filter.

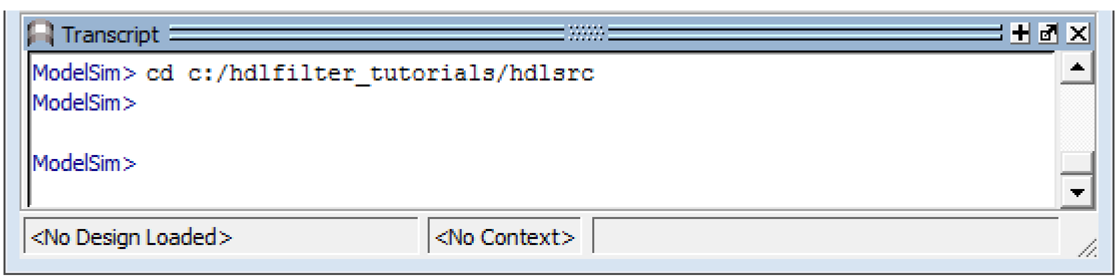
Verify the Generated VHDL Code

This section explains how to verify the generated VHDL code for the basic FIR filter with the generated VHDL test bench. This tutorial uses the Mentor Graphics® ModelSim® software as the tool for compiling and simulating the VHDL code. You can also use other VHDL simulation tool packages.

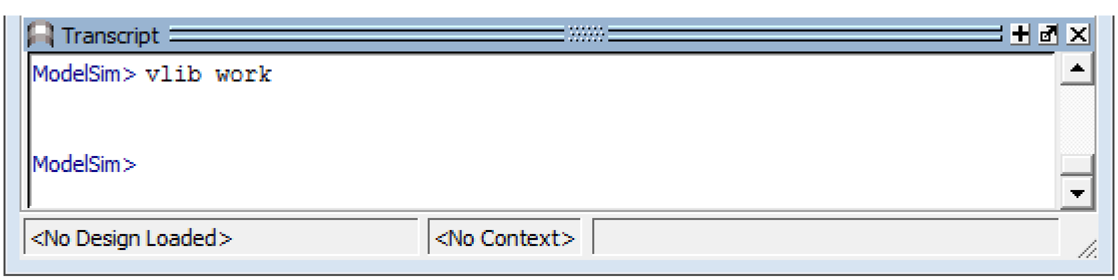
To verify the filter code, complete these steps:

1 Start your Mentor Graphics ModelSim simulator.

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



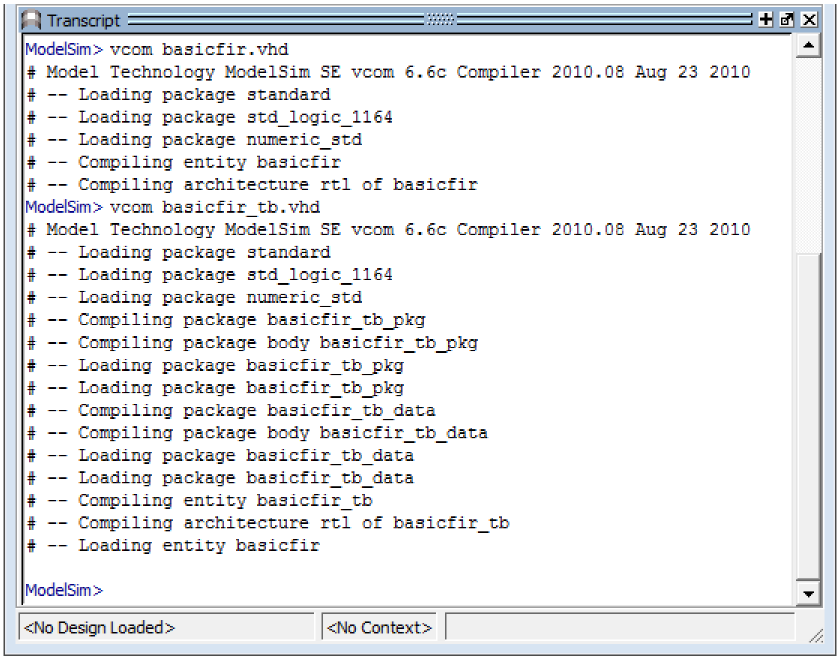
3 If desired, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.



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

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

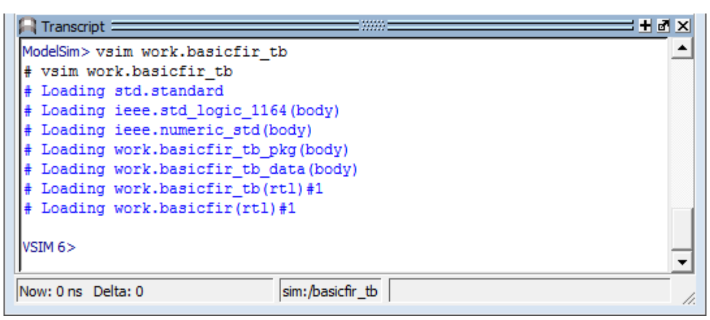
This screen display shows the command sequence and informational messages displayed during compilation.



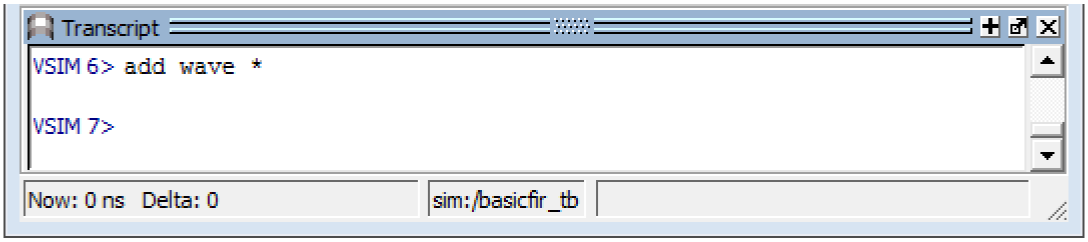
5 Load the test bench for simulation. The procedure for loading the test bench varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.basicfir_tb
```

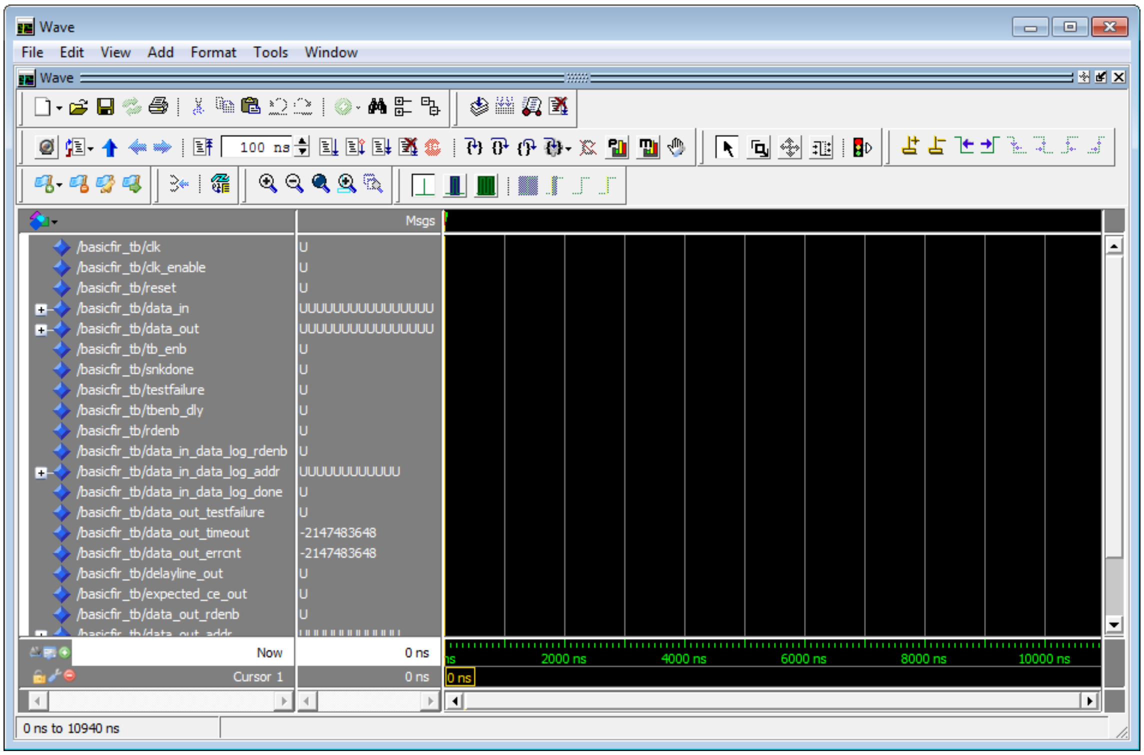
This figure shows the results of loading `work.basicfir_tb` with the `vsim` command.



6 Open a display window for monitoring the simulation as the test bench runs. In the Mentor Graphics ModelSim simulator, use this command to open a **wave** window and view the results of the simulation as HDL waveforms.

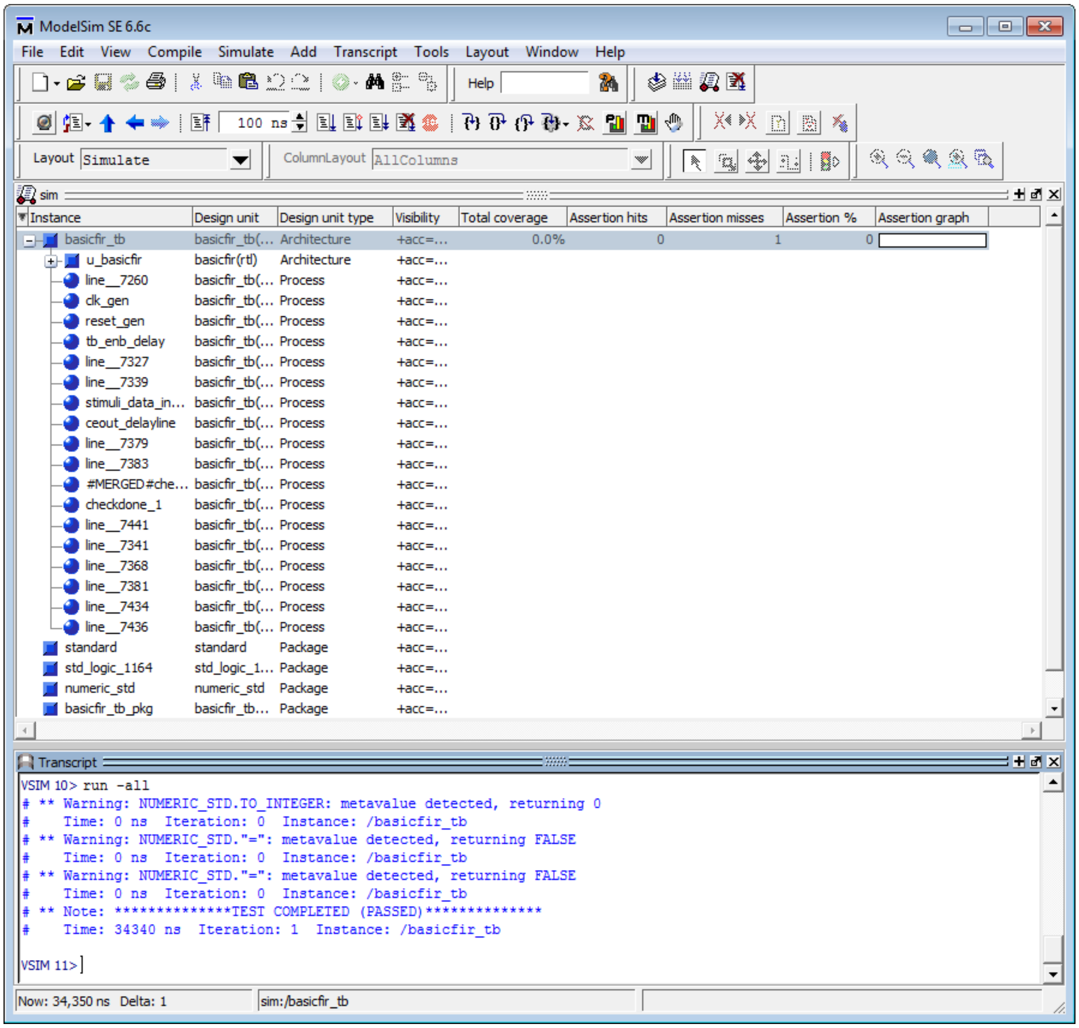


This **wave** window opens.



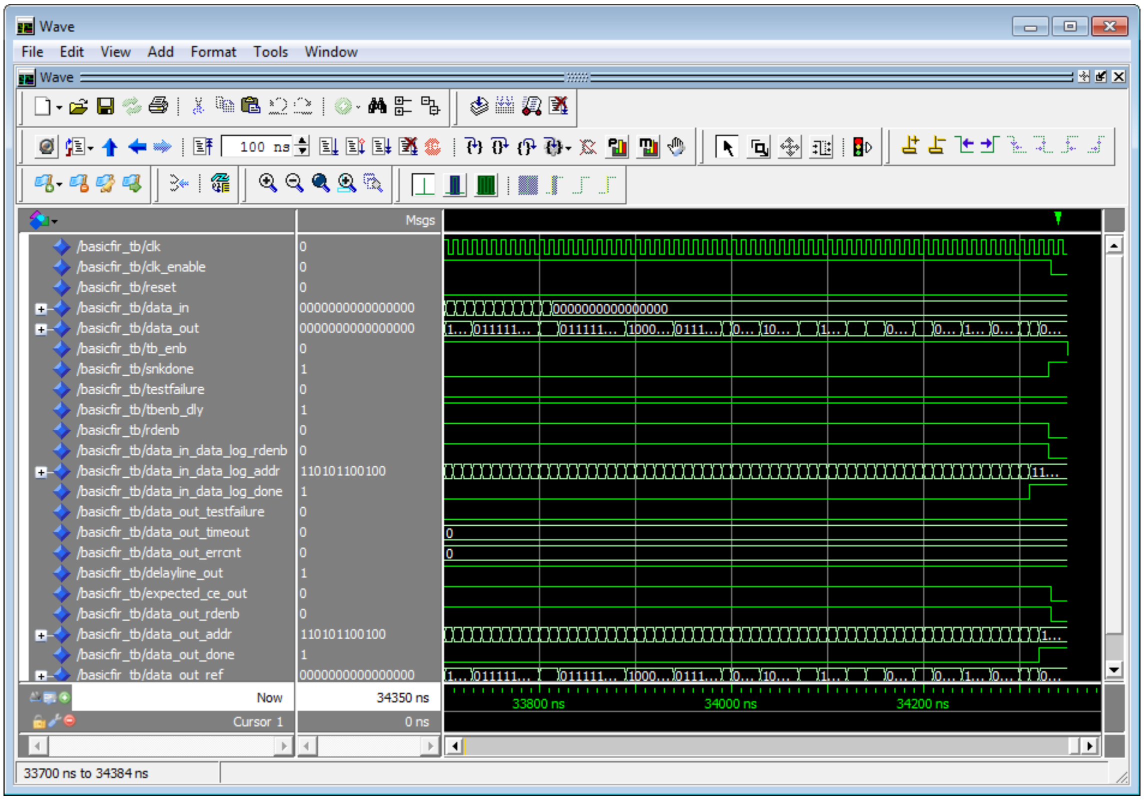
7 To start running the simulation, issue the start simulation command for your simulator. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the `run` command.

This figure shows starting a simulation with the `run -all` command.



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

This **wave** window shows the simulation results as HDL waveforms.


Wave
File Edit View Add Format Tools Window
Msgs
/basicfir_tb/clk 0
/basicfir_tb/clk_enable 0
/basicfir_tb/reset 0
/basicfir_tb/data_in 0000000000000000
/basicfir_tb/data_out 0000000000000000
/basicfir_tb/tb_enb 0
/basicfir_tb/snkdone 1
/basicfir_tb/testfailure 0
/basicfir_tb/tbenb_dly 1
/basicfir_tb/rdenb 0
/basicfir_tb/data_in_data_log_rdenb 0
/basicfir_tb/data_in_data_log_addr 110101100100
/basicfir_tb/data_in_data_log_done 1
/basicfir_tb/data_out_testfailure 0
/basicfir_tb/data_out_timeout 0
/basicfir_tb/data_out_errcnt 0
/basicfir_tb/delayline_out 1
/basicfir_tb/expected_ce_out 0
/basicfir_tb/data_out_rdenb 0
/basicfir_tb/data_out_addr 110101100100
/basicfir_tb/data_out_done 1
/basicfir_tb/data_out_ref 0000000000000000
Now 34350 ns
Cursor 1 0 ns
33800 ns
34000 ns
34200 ns
33700 ns to 34384 ns

Optimized FIR Filter

In this section...

Note The Filter Design HDL Coder product will be discontinued in a future release. Instead, you can model hardware behavior, and generate HDL code by using System objects or Simulink blocks from DSP HDL Toolbox. These objects and blocks include hardware-friendly control signals and architecture options. To generate HDL code from "DSP HDL Toolbox" objects and blocks, you must also have the HDL Coder product.

For examples of modeling an FIR filter for hardware, see "Fully Parallel Systolic FIR Filter Implementation" (DSP HDL Toolbox) and "Partly Serial Systolic FIR Filter Implementation" (DSP HDL Toolbox). These examples use the Discrete FIR Filter block. Equivalent functionality is also available in the `dsphdl.FIRFilter` System object.

Create a Folder for Your Tutorial Files

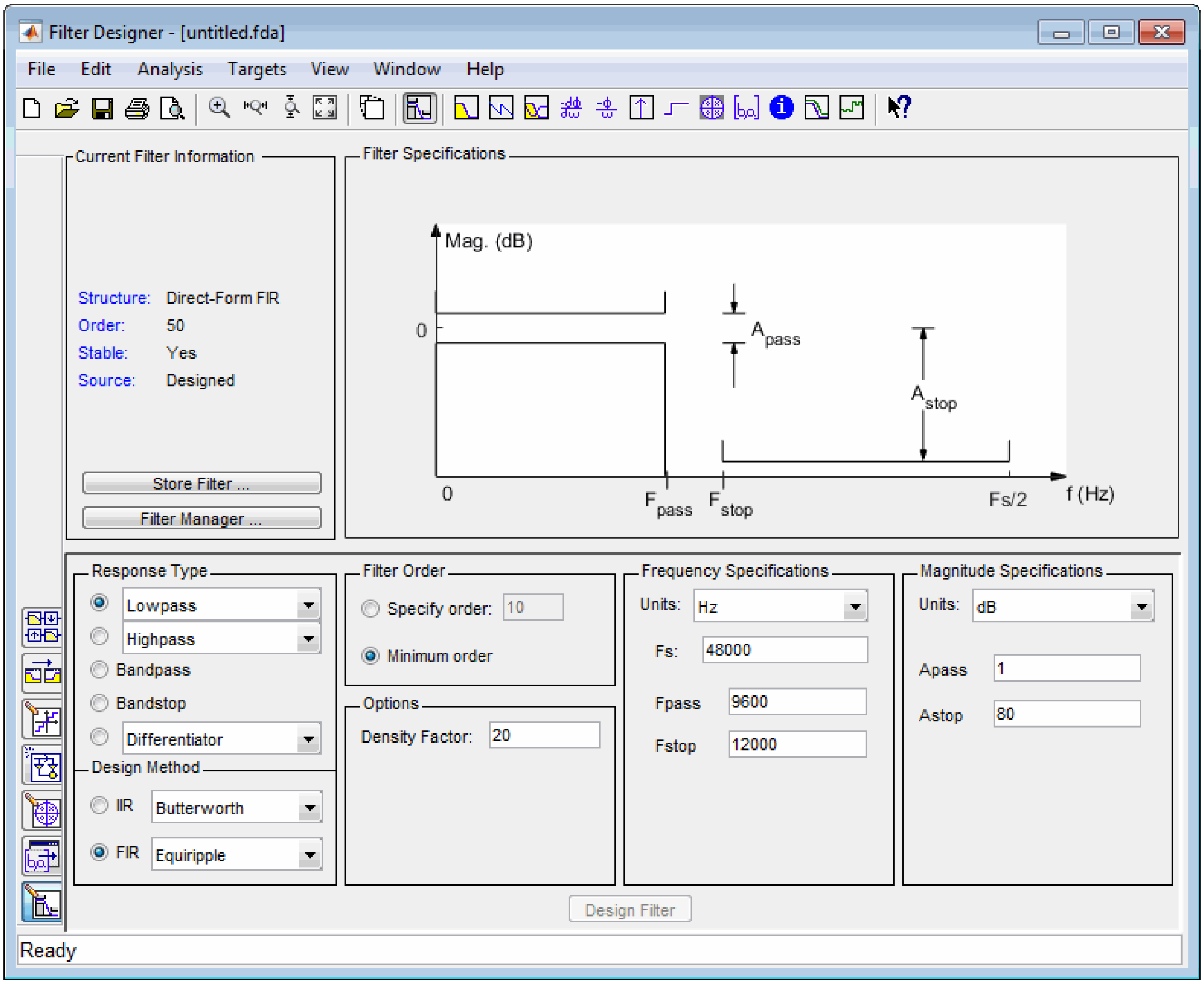
Set up a writable working folder outside your MATLAB installation folder to store files that will be generated as you complete your tutorial work. The tutorial instructions assume that you create the folder `hdlfilter_tutorials` on drive C.

Design the FIR Filter in Filter Designer

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

This section assumes that you are familiar with the MATLAB user interface and the Filter Designer.

1. Start the MATLAB software.
2. Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-21.
3. Start the Filter Designer by entering the `filterDesigner` command in the MATLAB Command Window. The Filter Design & Analysis Tool appears.



4 In the Filter Design & Analysis Tool, set these 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`

Option	Value
Magnitude Specifications	**Units**: dB **Apass**: 1 **Astop**: 80

These settings are for the default filter design that the Filter Designer creates for you. If you do not have to change the filter, and **Design Filter** is grayed out, you are done and can skip to “Quantize the FIR Filter” on page 1-23.

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

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

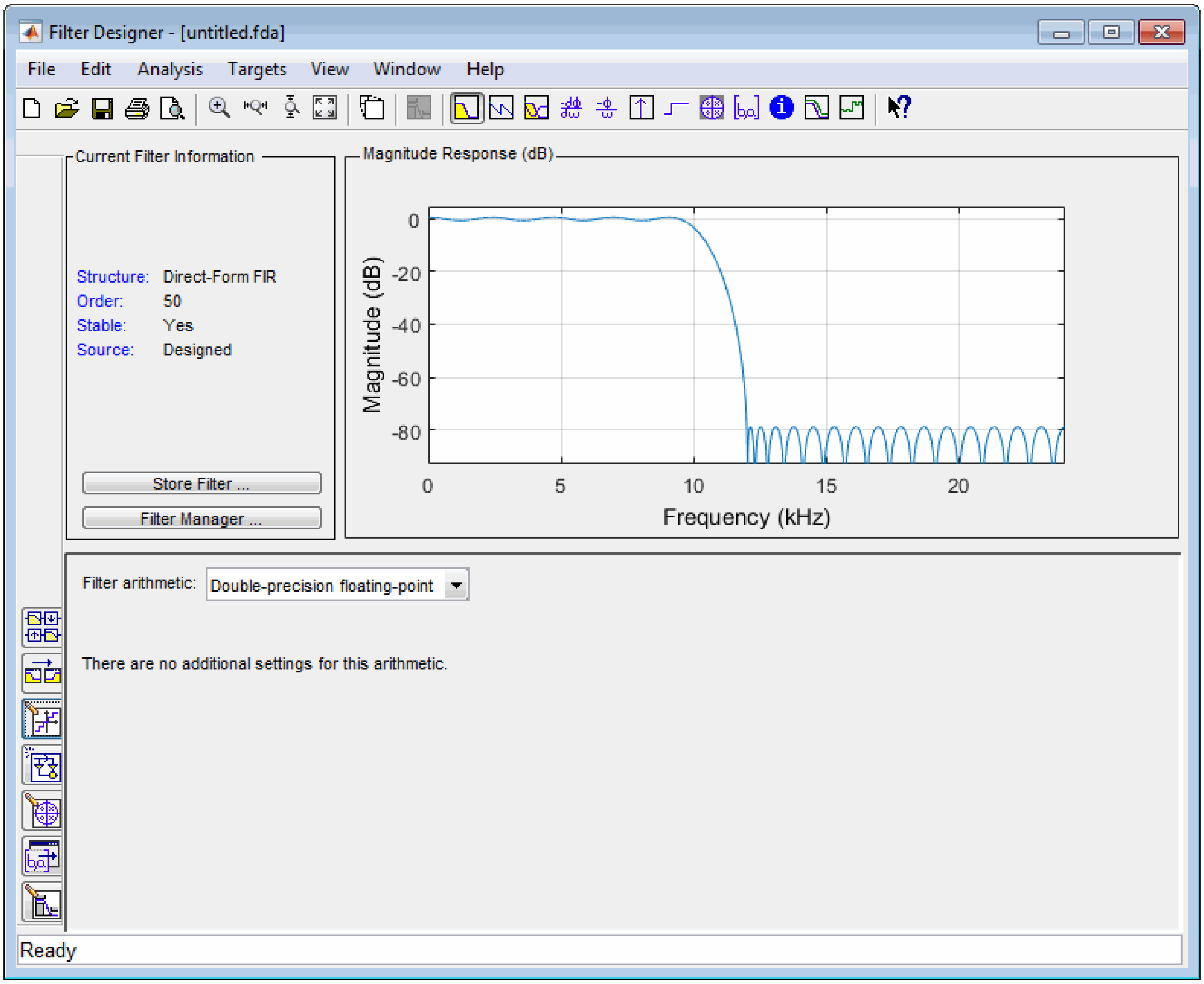
For more information on designing filters with the Filter Designer, see the DSP System Toolbox documentation.

Quantize the FIR Filter

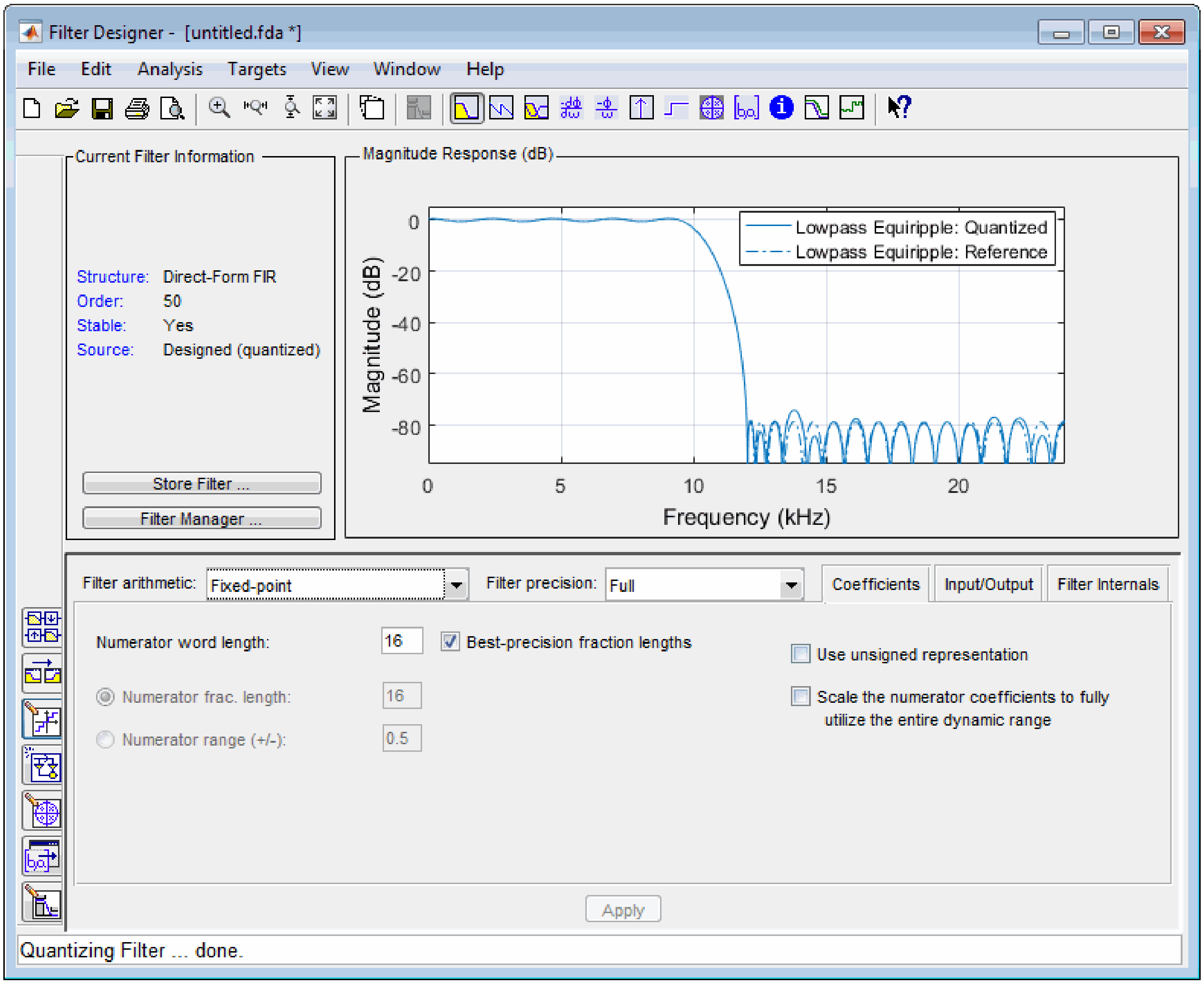
You must quantize filters for HDL code generation. To quantize your filter,

1 Open the FIR filter design you created in “Design the FIR Filter in Filter Designer” on page 1-21 if it is not already open.

2 Click the Set Quantization Parameters button  in the left-side toolbar. The Filter Designer displays a **Filter arithmetic** menu in the bottom half of the window.



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



Use the quantization options to test the effects of various settings on the performance and accuracy of the quantized filter.

4 Set the quantization parameters as follows:

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

Tab	Parameter	Setting
	Accum. word length	40

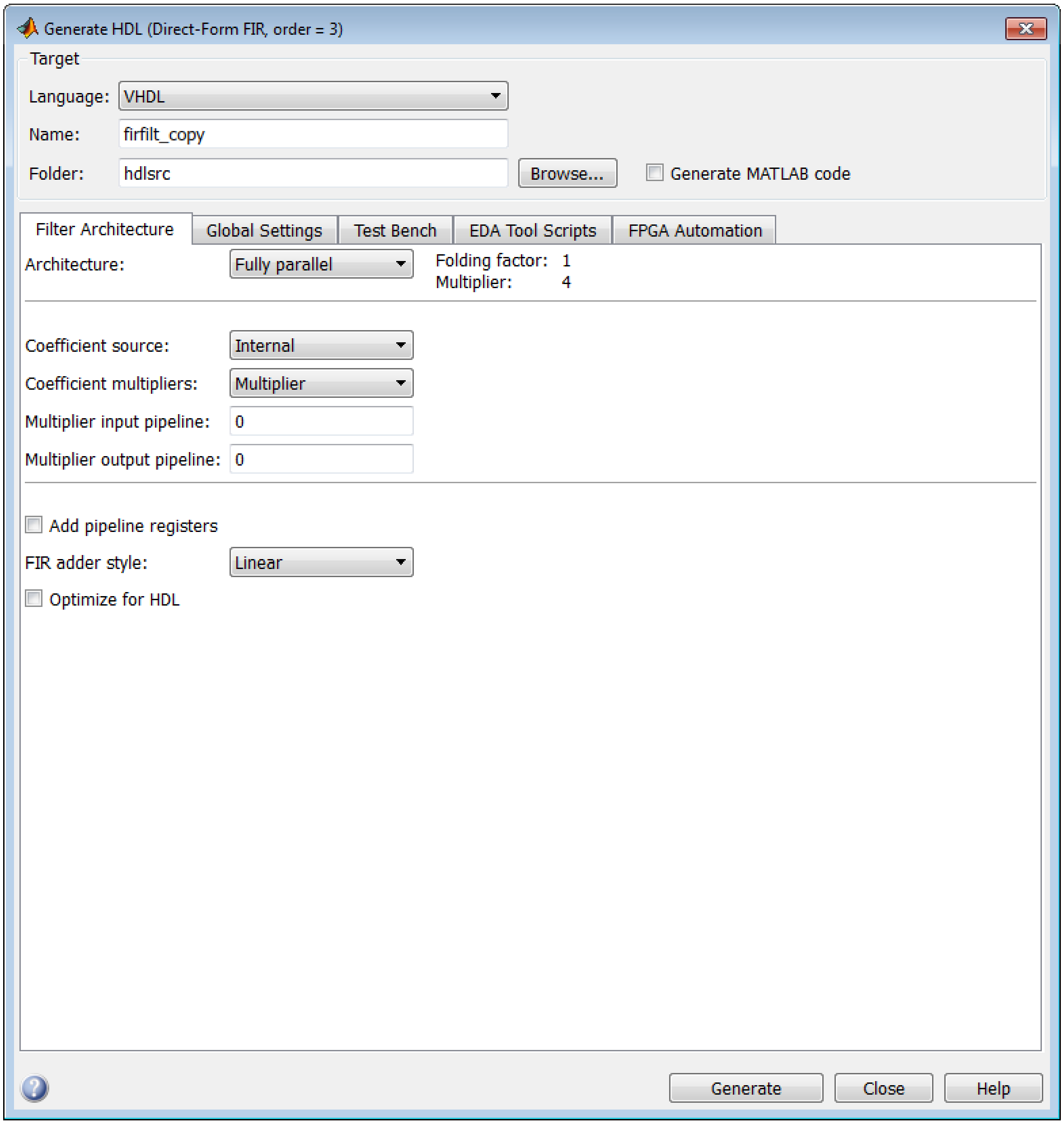
5 Click **Apply**.

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

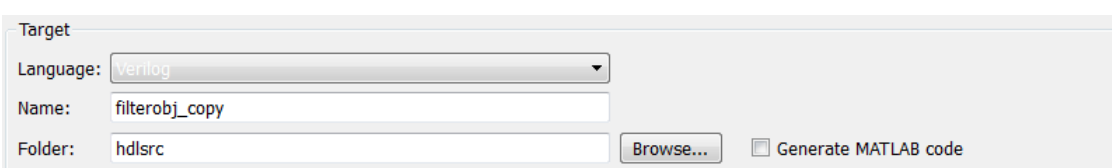
Configure and Generate Optimized Verilog Code

After you quantize your filter, you are ready to configure coder options and generate Verilog code for the filter. This section guides you through starting the UI, setting options, and generating the Verilog code and a test bench for the FIR filter you designed and quantized in “Design the FIR Filter in Filter Designer” on page 1-21 and “Quantize the FIR Filter” on page 1-23.

1 Start the Filter Design HDL Coder UI by selecting **Targets** > **Generate HDL** in the Filter Designer tool. The Filter Designer displays the Generate HDL tool.



2 Select `Verilog` for the **Language** option, as shown in this figure.



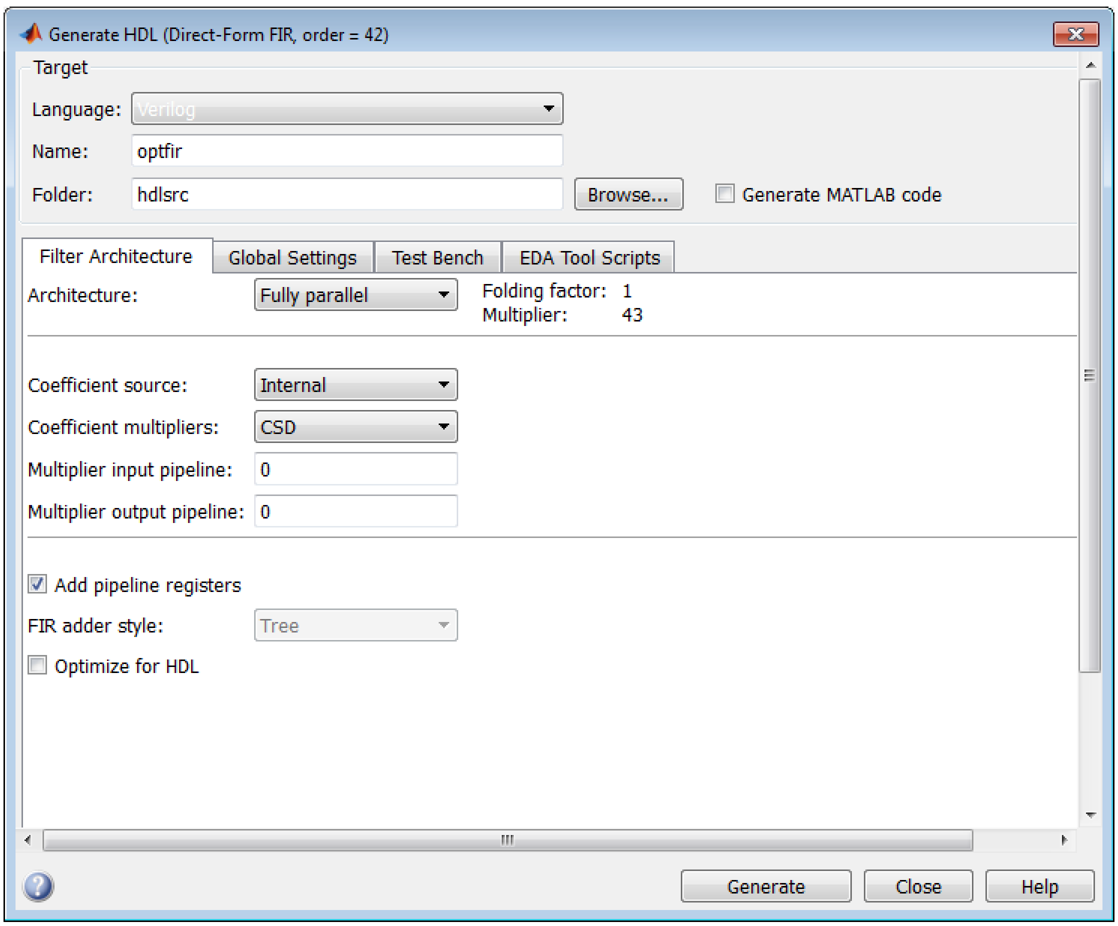
3 In the **Name** text box of the **Target** pane, replace the default name with `optfir`. This option names the Verilog module and the file that contains the Verilog code for the filter.

4 In the **Filter architecture** pane, select the **Optimize for HDL** option. This option is for generating HDL code that is optimized for performance or space requirements. When this option is enabled, the coder makes tradeoffs concerning data types and might ignore your quantization settings to achieve optimizations. When you use the option, keep in mind that you do so at the cost of potential numeric differences between filter results produced by the original filter object and the simulated results for the optimized HDL code.

5 Select CSD for the **Coefficient multipliers** option. This option optimizes coefficient multiplier operations by instructing the coder to replace them with additions of partial products produced by a canonical signed digit (CSD) technique. This technique minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.

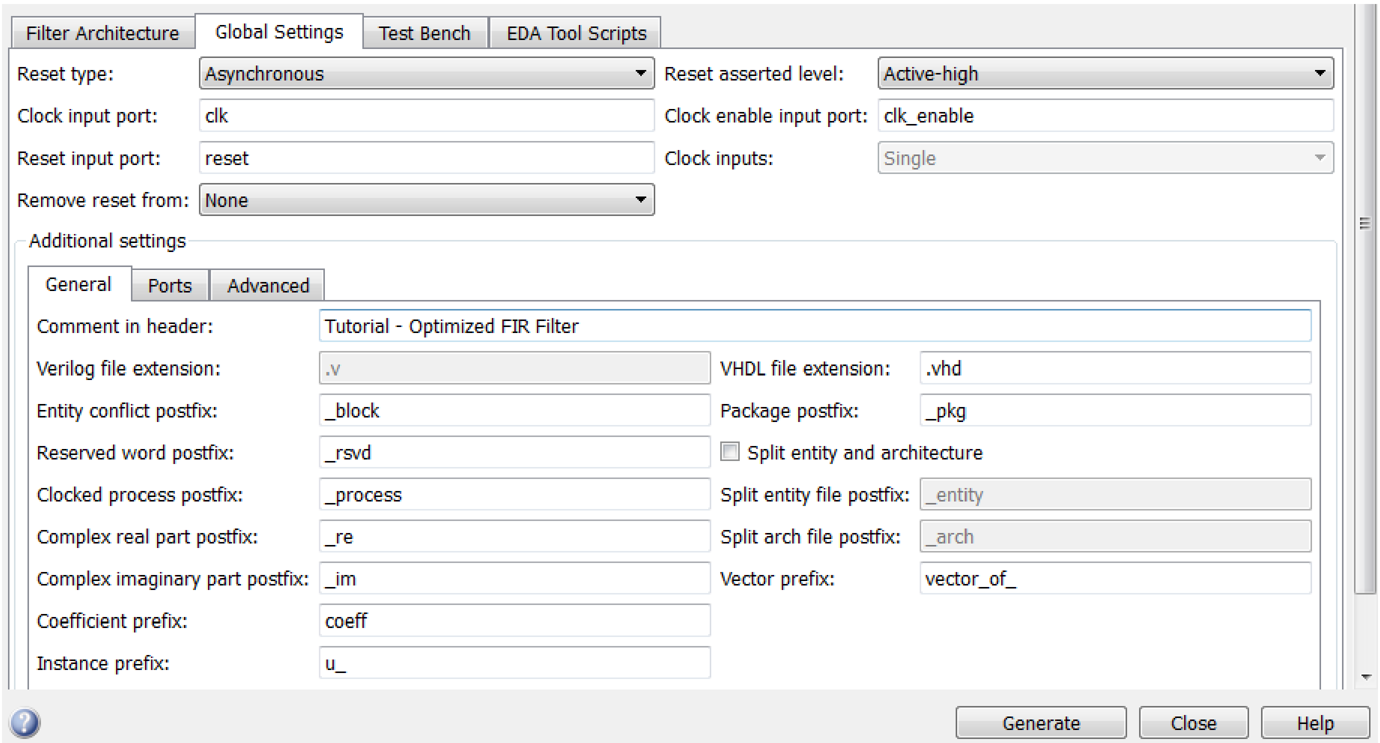
6 Select the **Add pipeline registers** option. For FIR filters, this option optimizes final summation. The coder creates a final adder that performs pairwise addition on successive products and includes a stage of pipeline registers after each level of the tree. When used for FIR filters, this option can produce numeric differences between results produced by the original filter object and the simulated results for the optimized HDL code.

7 The Generate HDL tool now appears.

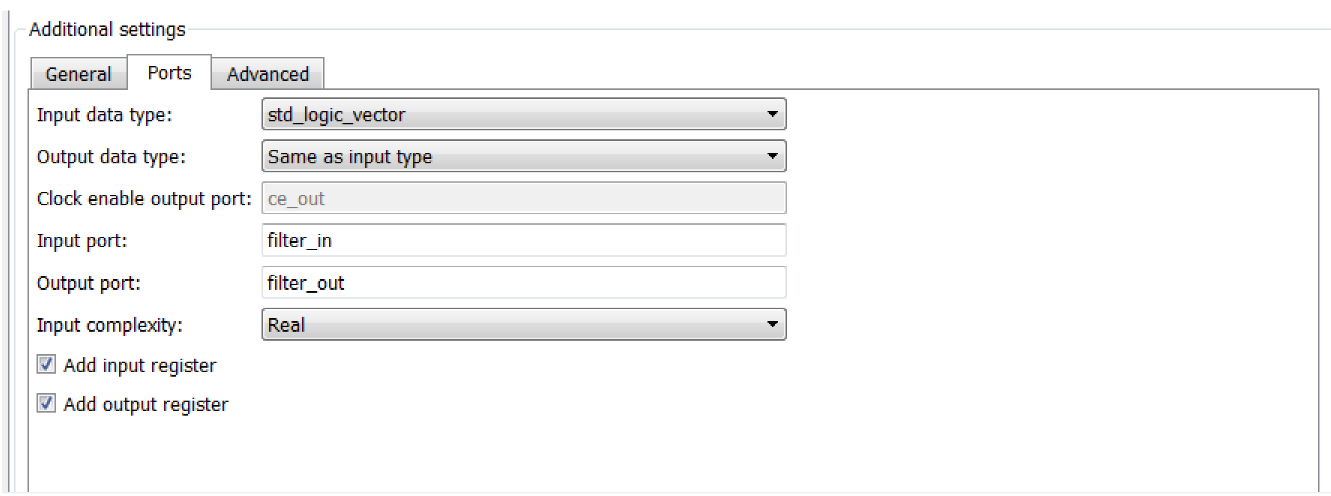


8 Select the **Global settings** tab of the UI. Then select the **General** tab of the **Additional settings** section.

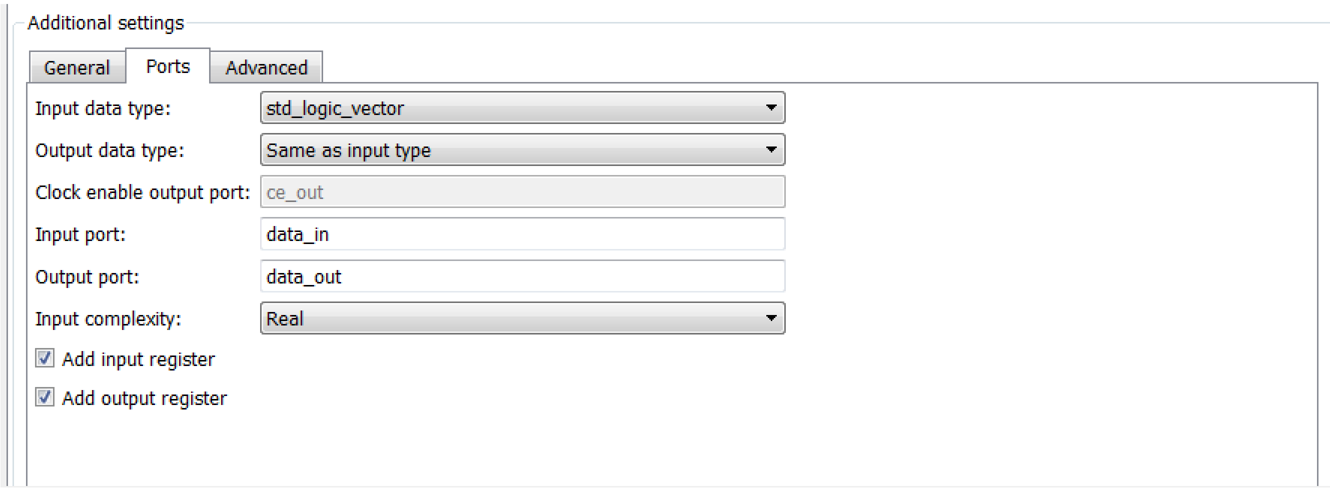
In the **Comment in header** text box, type `Tutorial - Optimized FIR Filter`. The coder adds the comment to the end of the header comment block in each generated file.



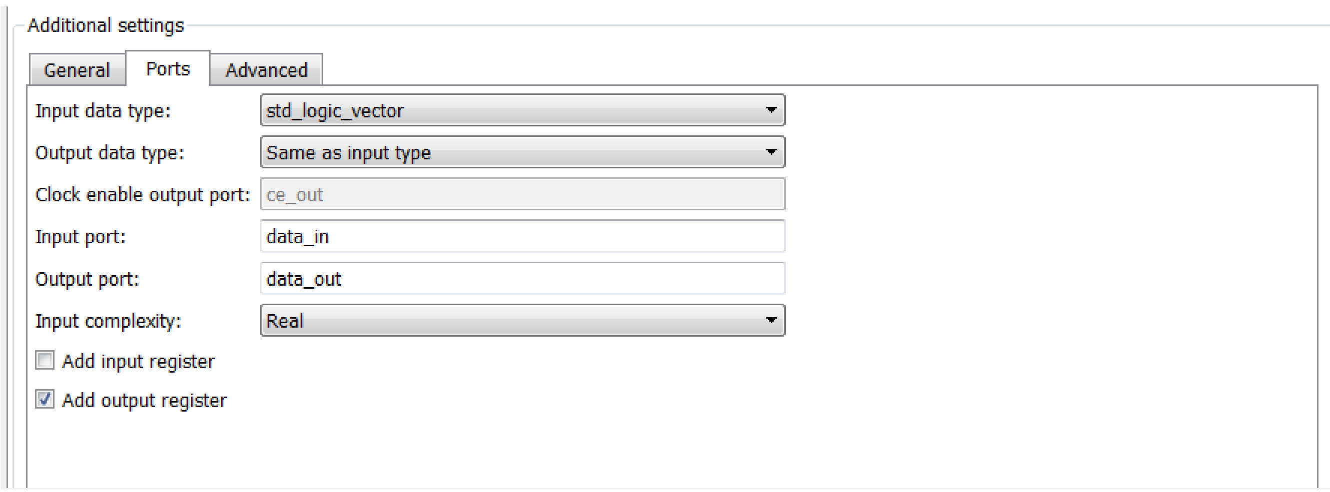
9 Select the **Ports** tab of the **Additional settings** section of the UI.



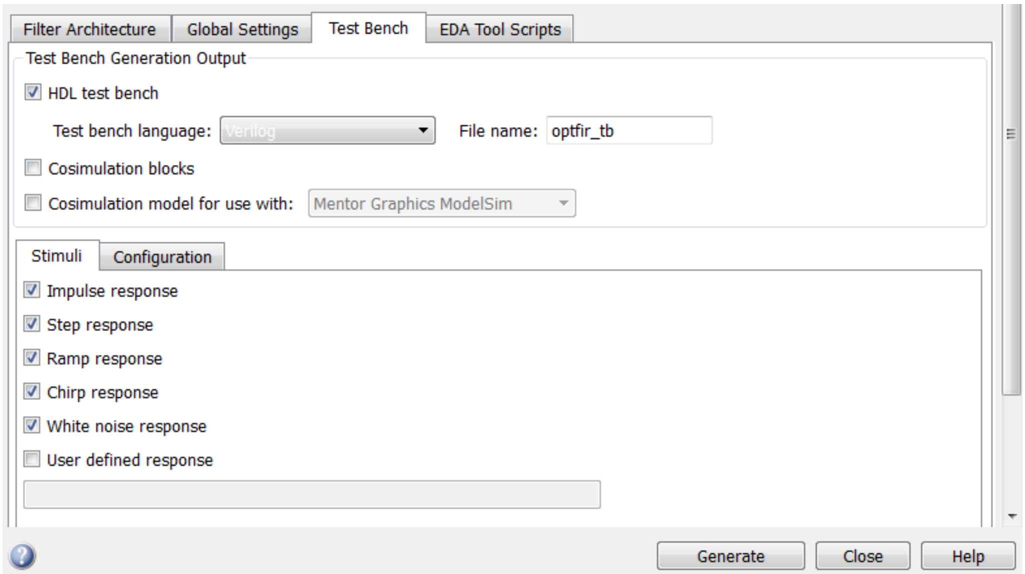
10 Change the names of the input and output ports. In the **Input port** text box, replace `filter_in` with `data_in`. In the **Output port** text box, replace `filter_out` with `data_out`.



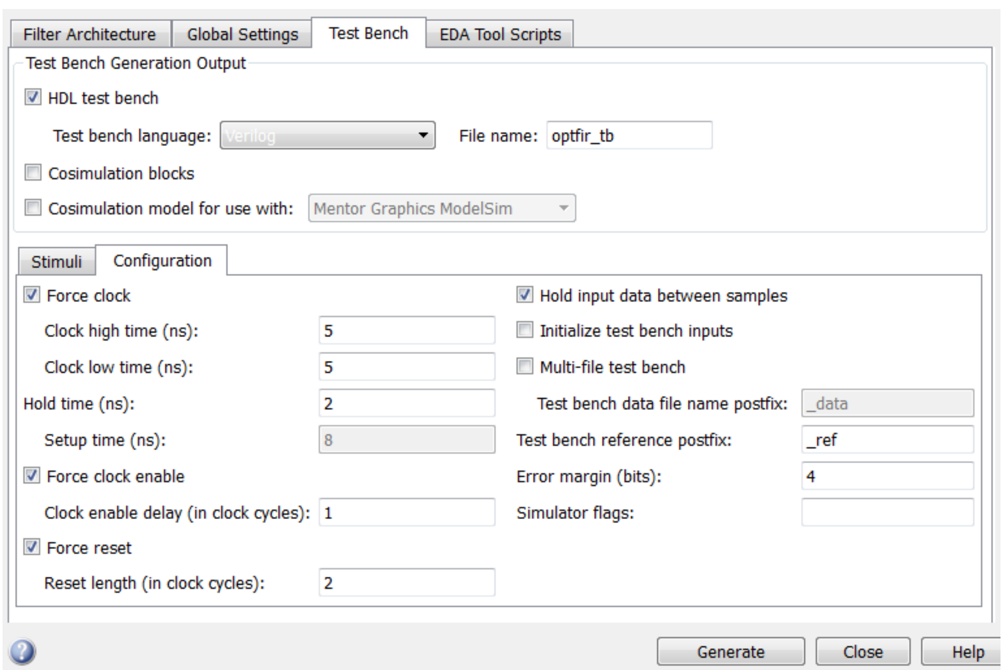
11 Clear the check box for the **Add input register** option. The **Ports** pane now looks as shown.



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



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



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

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

```
### Starting Verilog code generation process for filter: optfir
### Generating: C:\hdlfilter_tutorials\hdlsrc\optfir.v
```

```
### Starting generation of optfir Verilog module
### Starting generation of optfir Verilog module body
### HDL latency is 8 samples
### Successful completion of Verilog code generation process for filter: optfir

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

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

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

The generated Verilog code has these 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.
- The table shows the names of the ports.

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 `coeff`*n*, where *n* is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'`...
- The postfix `'_process'` is appended to sequential (`begin`) block names.

The generated test bench:

- Is a portable Verilog file.
- Forces clock, clock enable, and reset input signals.
- Forces the clock enable input signal to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.
- Applies a hold time of 2 nanoseconds to data input signals.
- Applies an error margin of 4 bits.
- For a FIR filter, applies impulse, step, ramp, chirp, and white noise stimulus types.

Explore the Optimized Generated Verilog Code

Get familiar with the 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 value you specified for the **Name** option in the **Target** pane. See step 3 in "Configure and Generate Optimized Verilog Code" on page 1-26.
3. Search for `Tutorial`. This section of code is where the coder places the text you entered for the **Comment in header** option. See step 9 in "Configure and Generate Optimized Verilog Code" on page 1-26.
4. Search for `HDL Code`. This section lists the coder options you modified in "Configure and Generate Optimized Verilog Code" on page 1-26.
5. Search for `Filter Settings`. This section of the VHDL code describes the filter design and quantization settings as you specified in "Design the FIR Filter in Filter Designer" on page 1-21 and "Quantize the FIR Filter" on page 1-23.
6. Search for `module`. This line names the Verilog module, using the value you specified for the **Name** option in the **Target** pane. This line also declares the list of ports, as defined by options on the **Ports** pane of the Generate HDL tool. The ports for data input and output are named with the values you specified for the **Input port** and **Output port** options on the **Ports** tab of the Generate HDL tool. See steps 3 and 11 in "Configure and Generate Optimized Verilog Code" on page 1-26.
7. Search for `input`. This line and the four lines that follow, declare the direction mode of each port.
8. Search for `Constants`. This code defines the coefficients. They are named using the default naming scheme, `coeff`*n*, where *n* is the coefficient number, starting with 1.
9. Search for `Signals`. This code defines the signals of the filter.
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 "Configure and Generate Optimized Verilog Code" on page 1-26.
11. Search for `process`. The `block` name `Delay_Pipeline_process` includes the default `block` postfix `'_process'`.
12. Search for `reset`. This code asserts the reset signal. 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 "Configure and Generate Optimized Verilog Code" on page 1-26.
15. Search for `output_register`. This code writes the filter output to an output register. The code for this register is generated by default. In step 12 in "Configure and Generate Optimized Verilog Code" on page 1-26 , 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 `'_process'`.

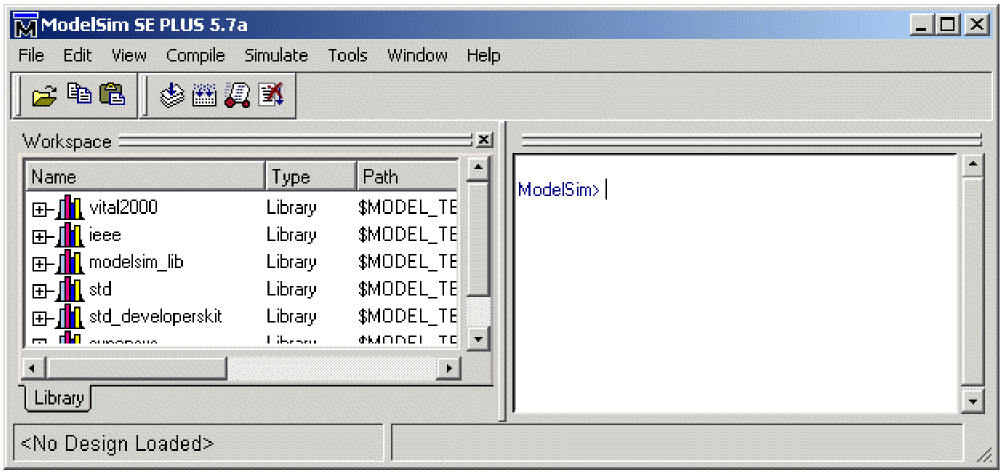
16 Search for `data_out`. This code drives the output data of the filter.

Verify the Generated Verilog Code

This section explains how to verify the optimized generated Verilog code for the FIR filter with the generated Verilog test bench. This tutorial uses the Mentor Graphics ModelSim simulator as the tool for compiling and simulating the Verilog code. You can use other HDL simulation tool packages.

To verify the filter code, complete these steps.

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



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

```
cd hdlsrc
```

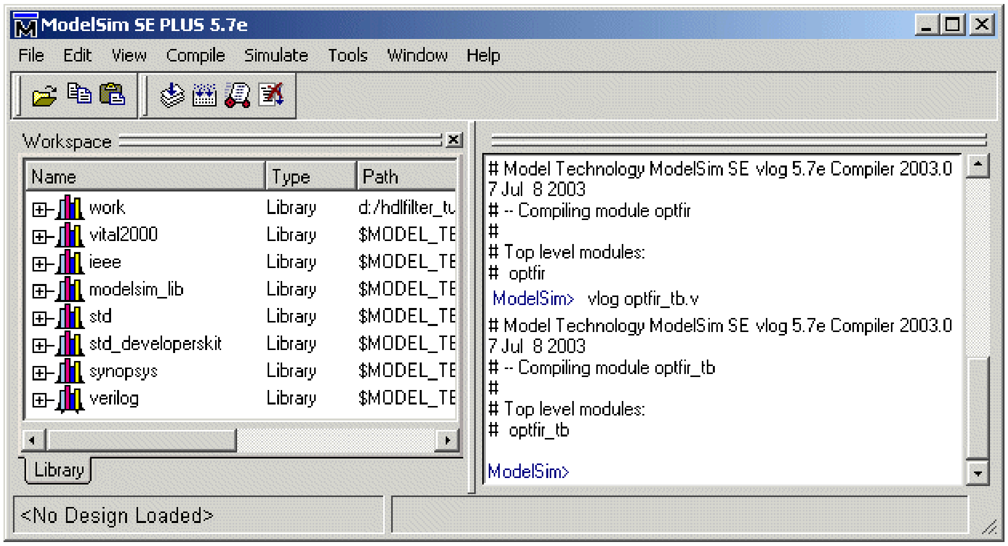
3 If desired, create a design library to store the compiled Verilog modules. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

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

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

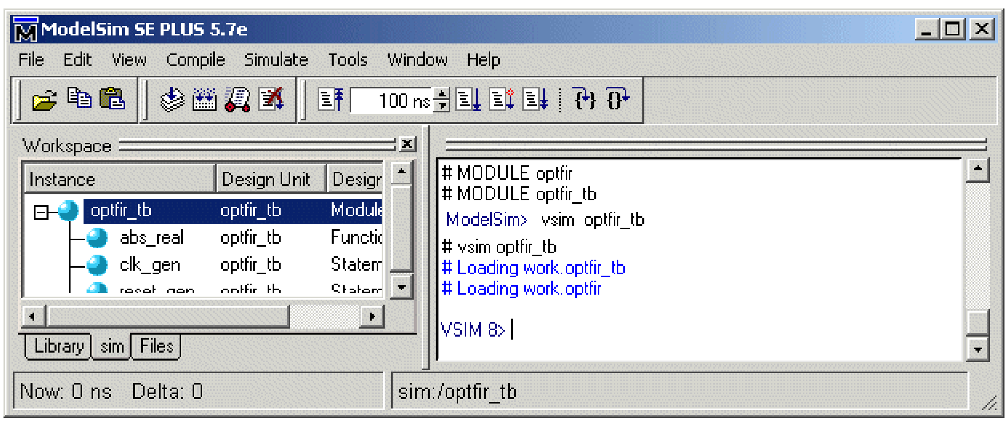
This figure shows the command sequence and informational messages displayed during compilation.



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

```
vsim optfir_tb
```

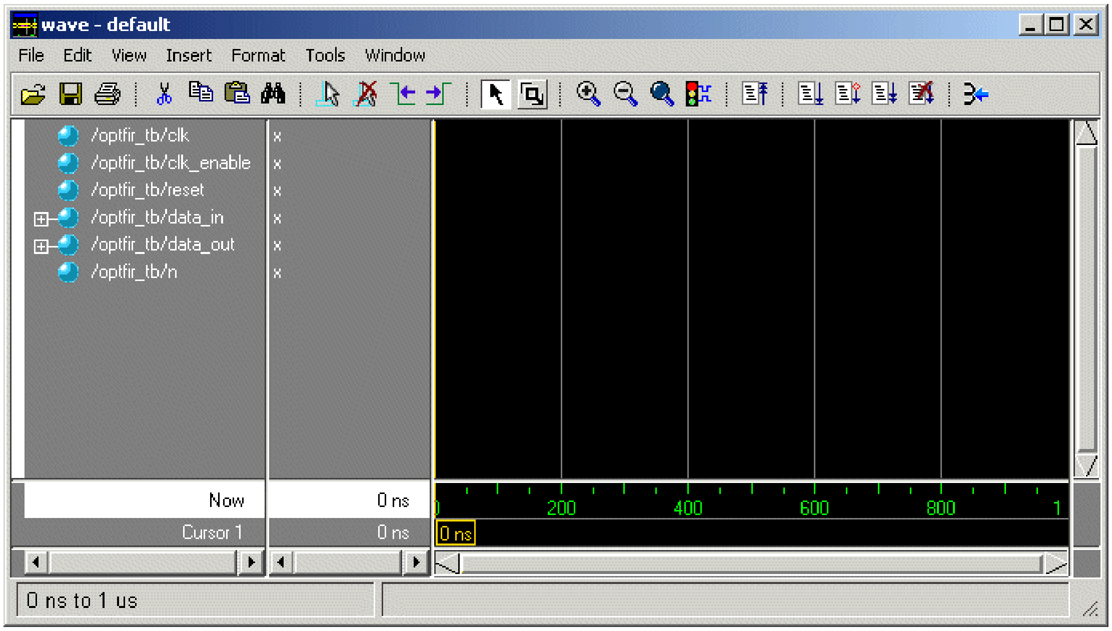
This figure shows the results of loading `optfir_tb` with the `vsim` command.



6 Open a display window for monitoring the simulation as the test bench runs. In the Mentor Graphics ModelSim simulator, can use this command to open a **wave** window and view the results of the simulation as HDL waveforms.

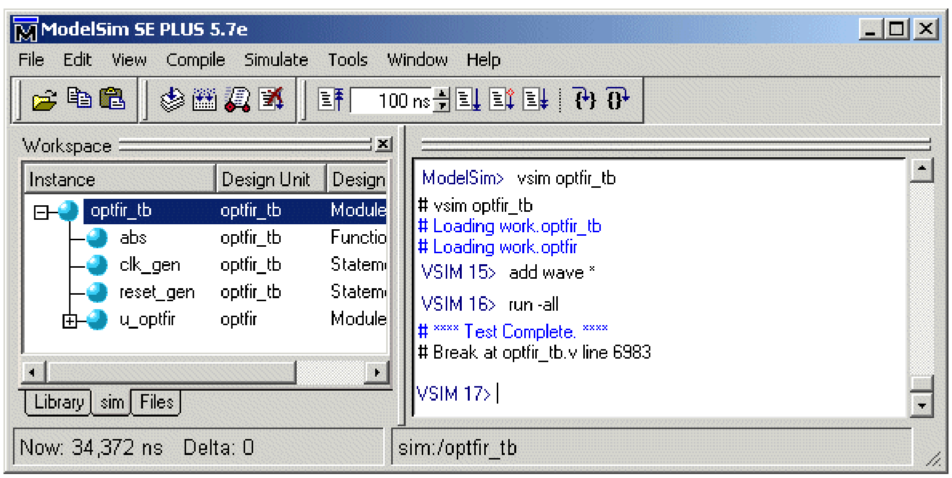
```
add wave *
```

This **wave** window opens.



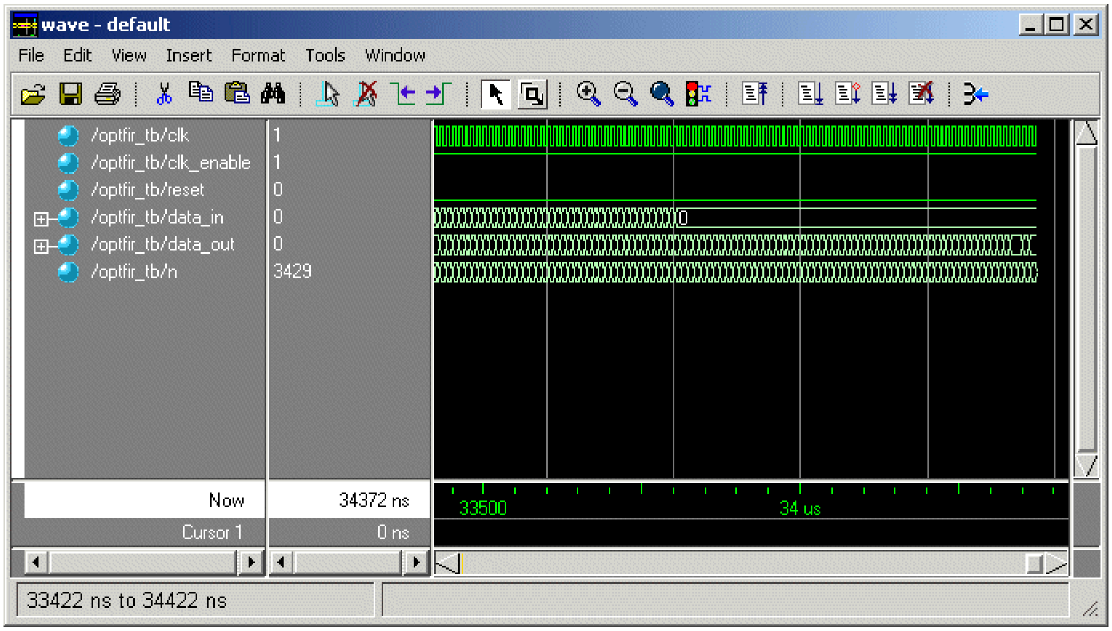
7 To start running the simulation, issue the start simulation command for your simulator. For example, in the Mentor Graphics ModelSim simulator, you can start a simulation with the `run` command.

This figure shows the `run -all` command being used to start a simulation.



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

This **wave** window shows the simulation results as HDL waveforms.


wave - default
File Edit View Insert Format Tools Window
/optfir_tb/clk 1
/optfir_tb/clk_enable 1
/optfir_tb/reset 0
/optfir_tb/data_in 0
/optfir_tb/data_out 0
/optfir_tb/n 3429
Now 34372 ns
Cursor 1 0 ns
33500
34 us
33422 ns to 34422 ns

IIR Filter

In this section...

Note The Filter Design HDL Coder product will be discontinued in a future release. Instead, you can model hardware behavior, and generate HDL code by using System objects or Simulink blocks from DSP HDL Toolbox. These objects and blocks include hardware-friendly control signals and architecture options. To generate HDL code from "DSP HDL Toolbox" objects and blocks, you must also have the HDL Coder product.

For an example of modeling an IIR filter for hardware, see "High Performance DC Blocker for FPGA" (DSP HDL Toolbox). This example uses the Biquad Filter block. Equivalent functionality is also available in the `dsphdl.BiquadFilter` System object.

Create a Folder for Your Tutorial Files

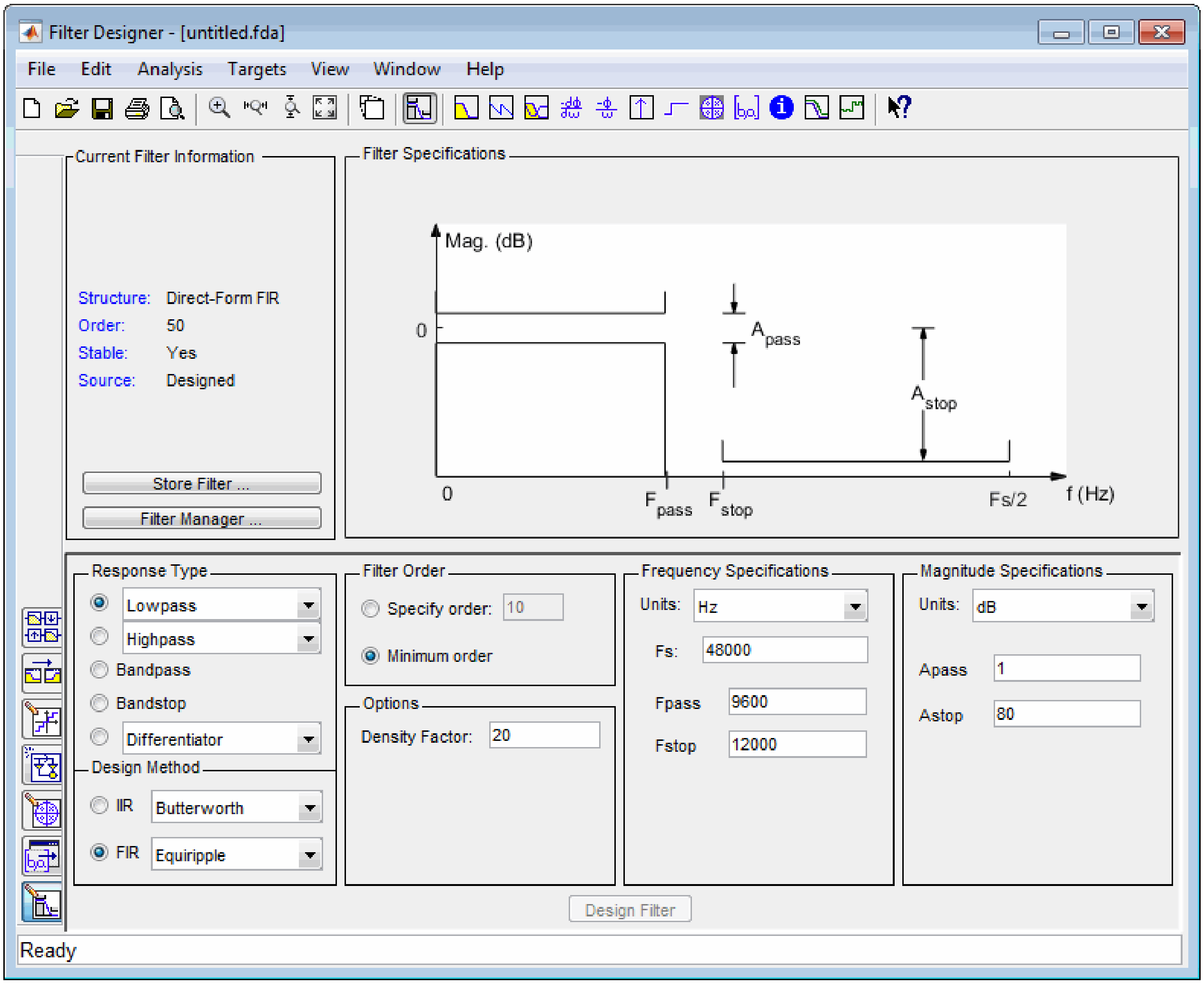
Set up a writable working folder outside your MATLAB installation folder to store files that will be generated as you complete your tutorial work. The tutorial instructions assume that you create the folder `hdlfilter_tutorials` on drive C.

Design an IIR Filter in Filter Designer

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

This section guides you through the procedure of designing and creating a filter for an IIR filter. This section assumes that you are familiar with the MATLAB user interface and the Filter Designer.

1. Start the MATLAB software.
2. Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-38.
3. Start the Filter Designer by entering the `filterDesigner` command in the MATLAB Command Window. The Filter Design & Analysis Tool appears.



4 In the Filter Design & Analysis Tool, set these 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 Filter Designer creates a filter for the specified design. This message appears in the Filter Designer status bar when the task is complete.

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

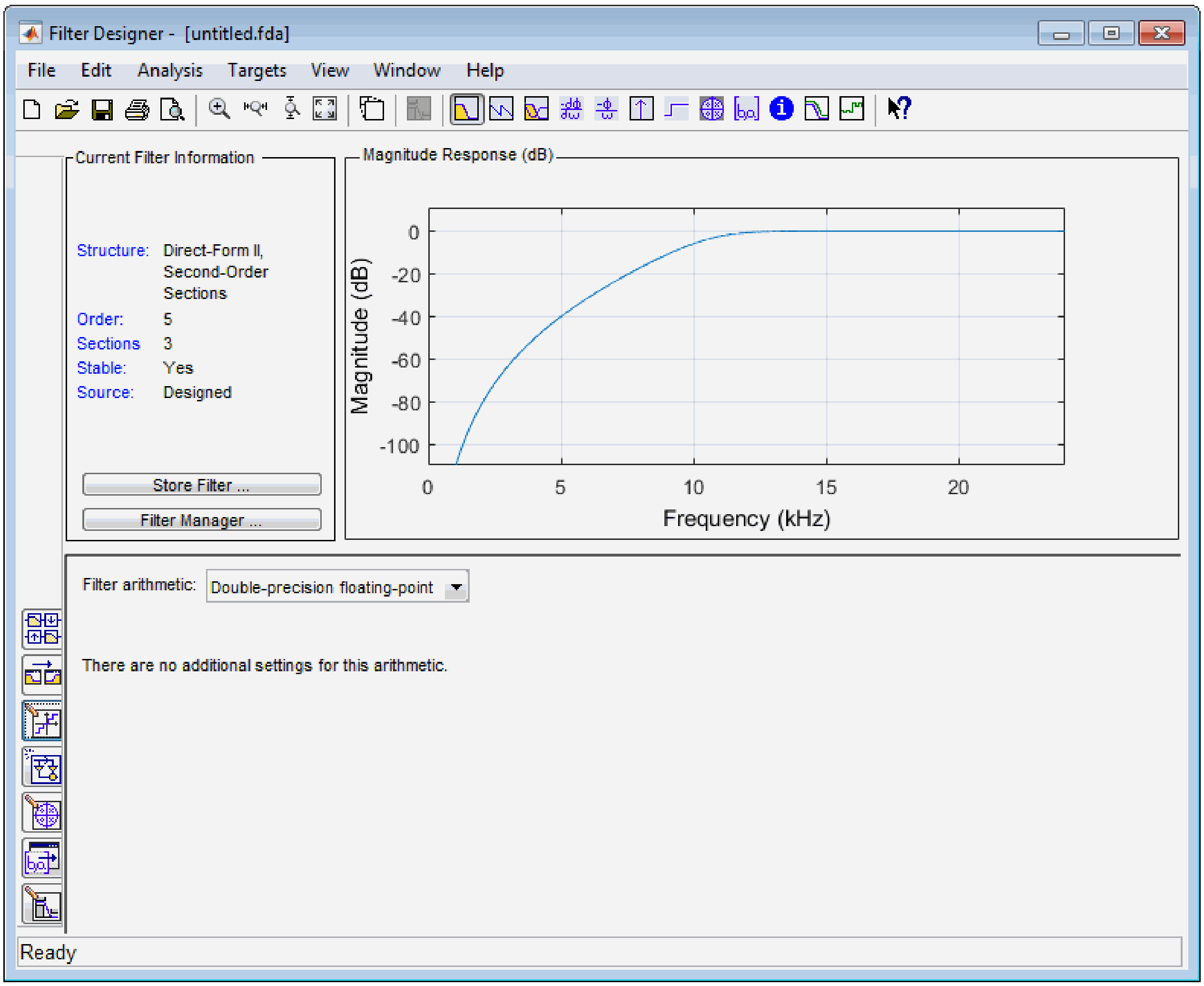
For more information on designing filters with the Filter Designer, see “Use Filter Designer with DSP System Toolbox Software”.

Quantize the IIR Filter

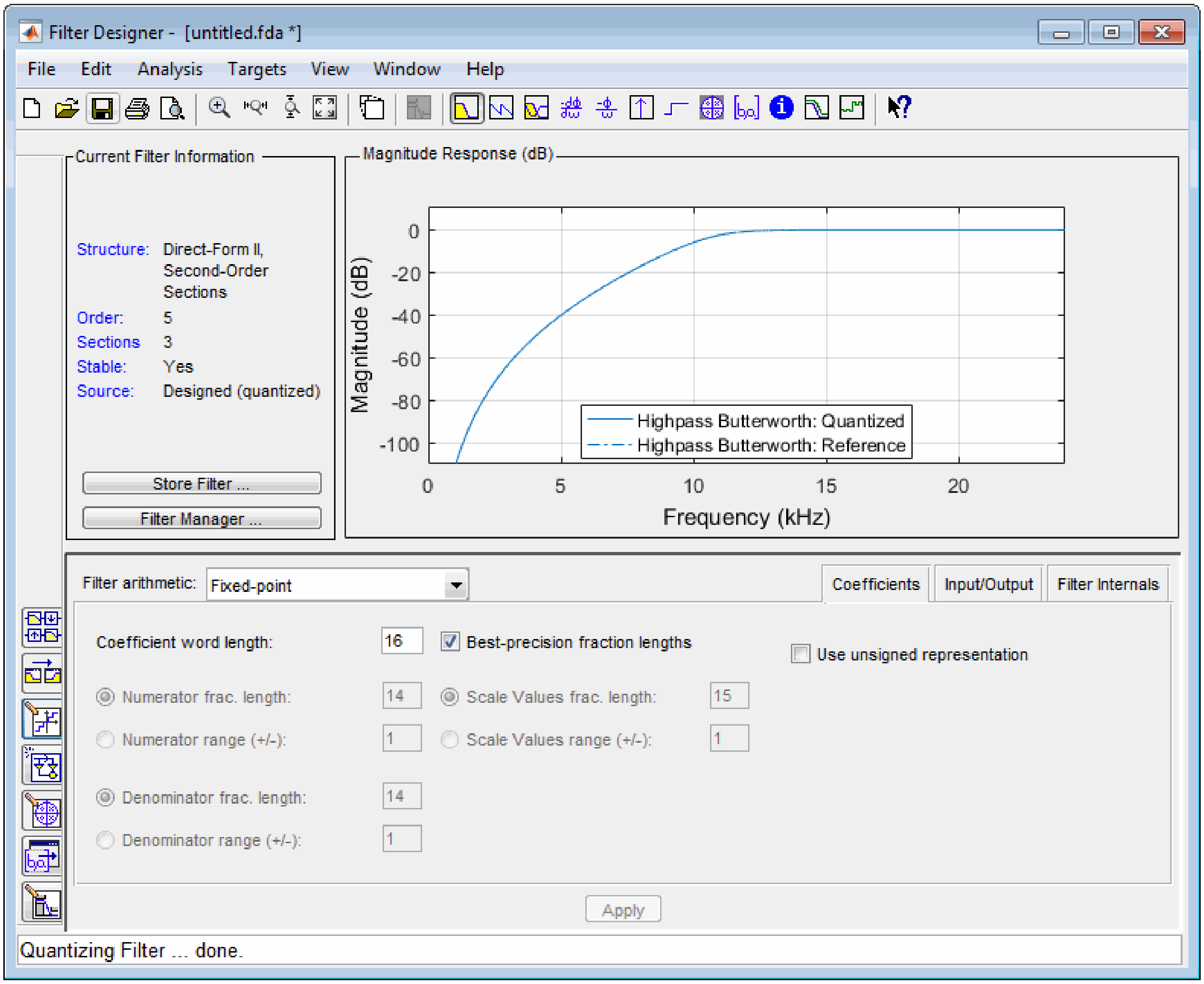
You should quantize filters for HDL code generation. To quantize your filter,

1 Open the IIR filter design you created in “Design an IIR Filter in Filter Designer” on page 1-38 if it is not already open.

2 Click the Set Quantization Parameters button  in the left-side toolbar. The Filter Designer displays the **Filter arithmetic** list in the bottom half of the window.



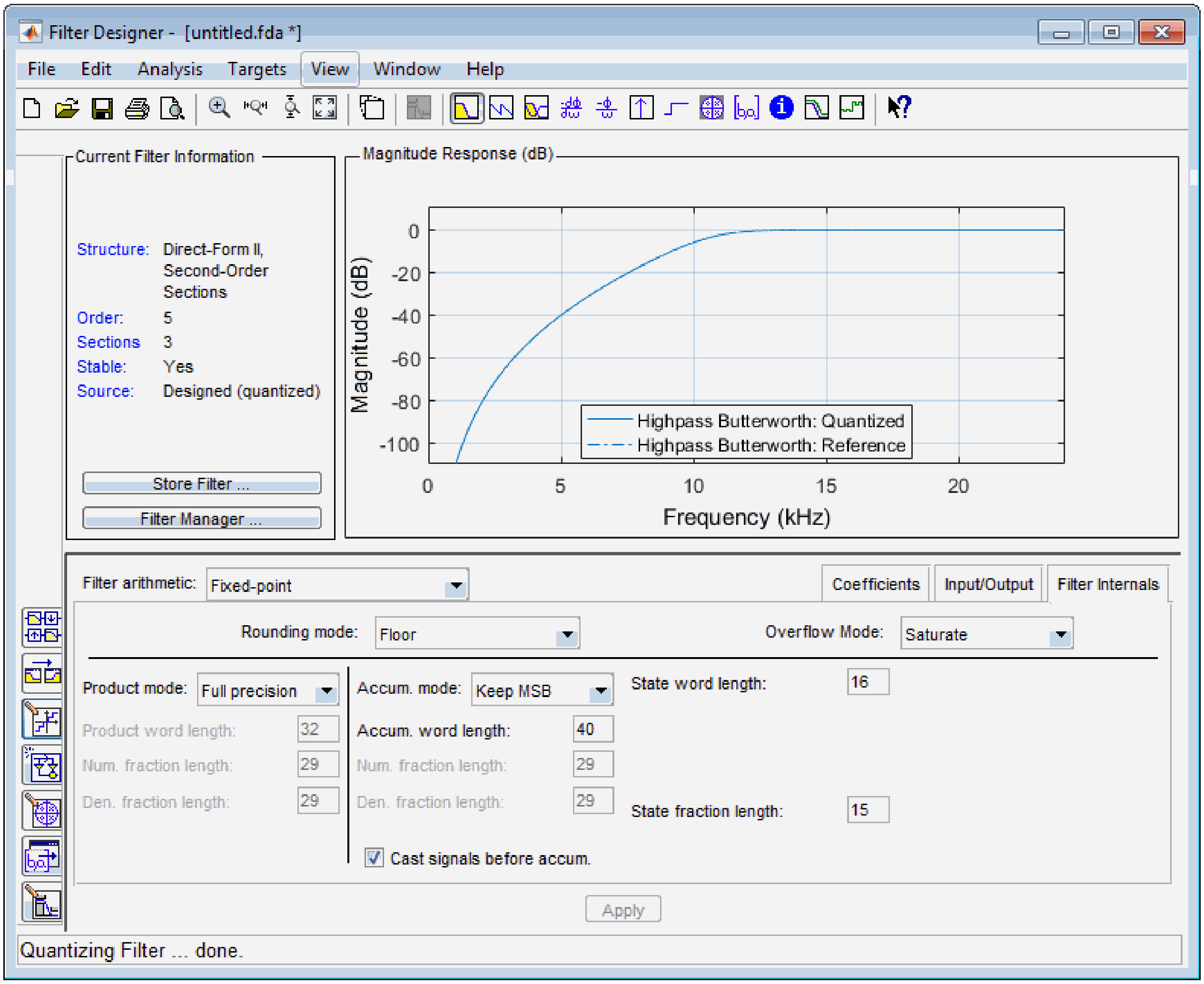
3 Select `Fixed-point` from the list. The Filter Designer displays the first of three tabbed panels of the window.



Use the quantization options to test the effects of various settings on the performance and accuracy of the quantized filter.

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

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

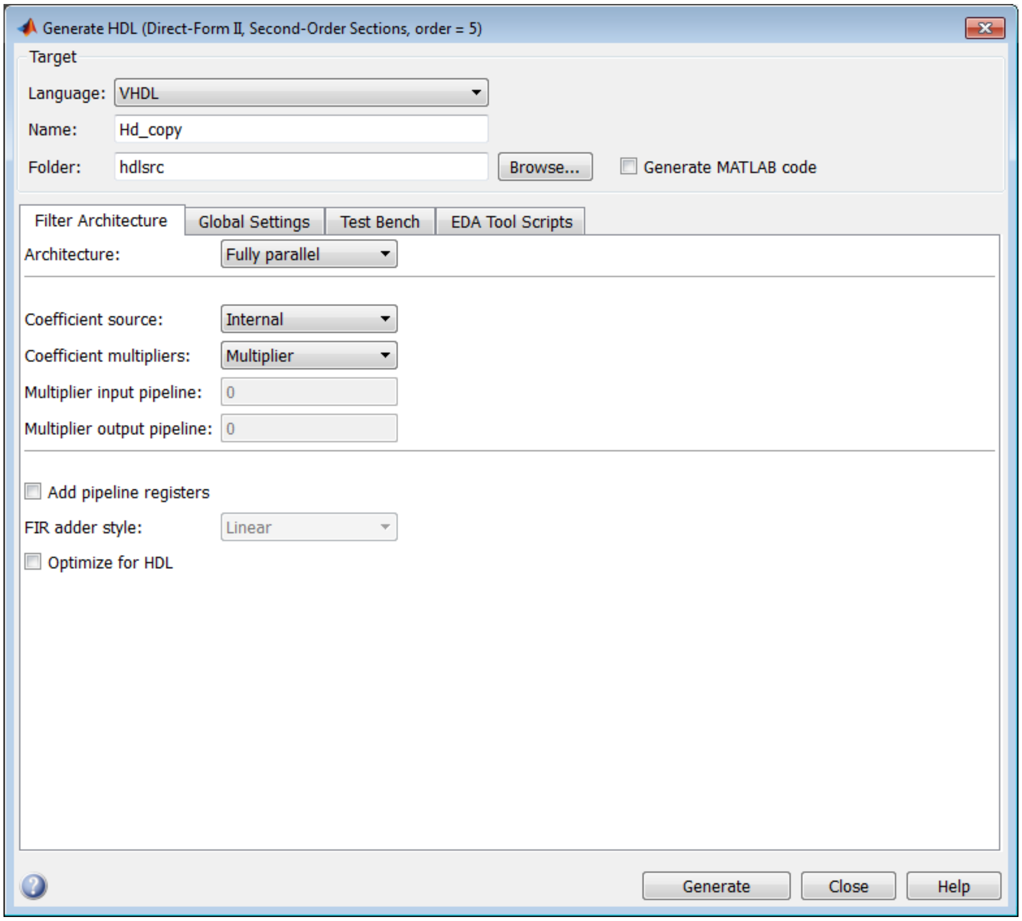


For more information on quantizing filters with the Filter Designer, see "Use Filter Designer with DSP System Toolbox Software".

Configure and Generate VHDL Code

After you quantize your filter, you are ready to configure coder options and generate VHDL code. This section guides you through starting the Filter Design HDL Coder UI, setting options, and generating the VHDL code and a test bench for the IIR filter you designed and quantized in "Design an IIR Filter in Filter Designer" on page 1-38 and "Quantize the IIR Filter" on page 1-40.

1 Start the Filter Design HDL Coder UI by selecting **Targets > Generate HDL** in the Filter Designer tool. The Filter Designer displays the Generate HDL tool.

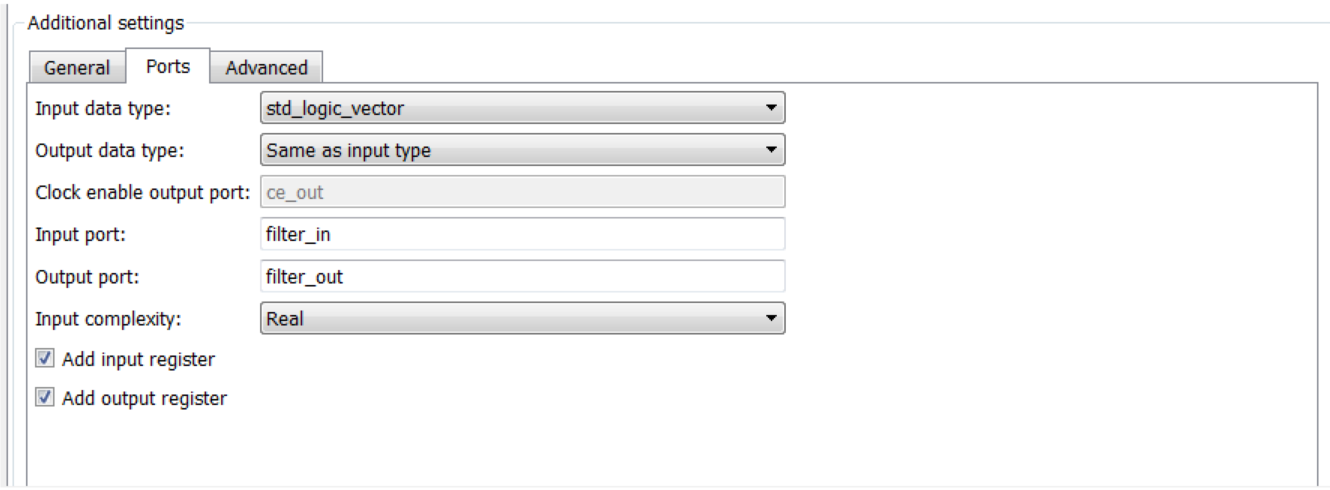


2 In the **Name** text box of the **Target** pane, type `iir`. This option names the VHDL entity and the file that contains the VHDL code for the filter.

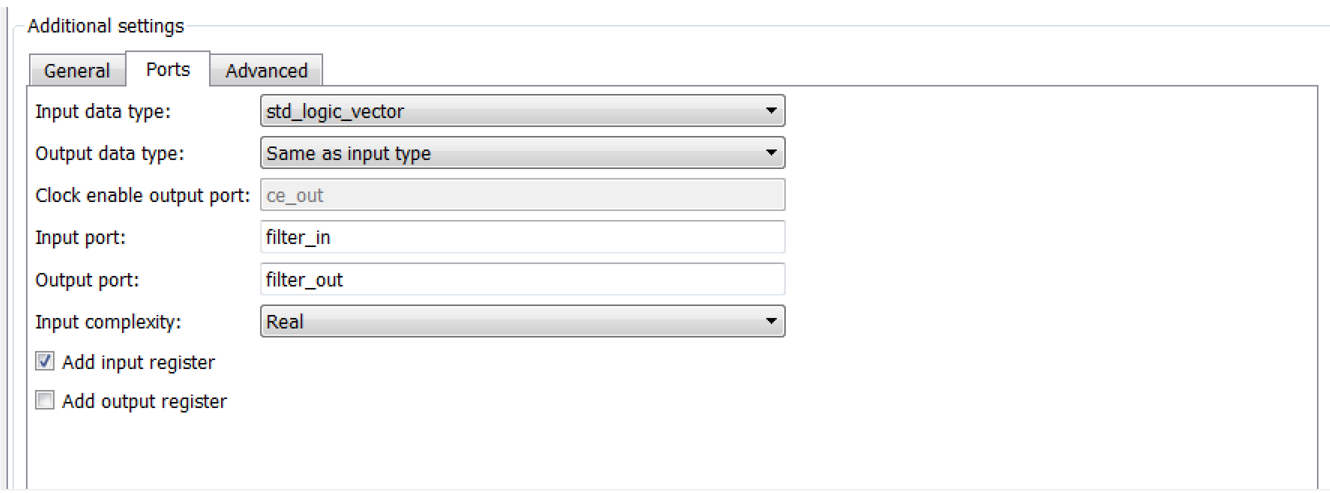
3 Select the **Global settings** tab of the UI. Then select the **General** tab of the **Additional settings** section.

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

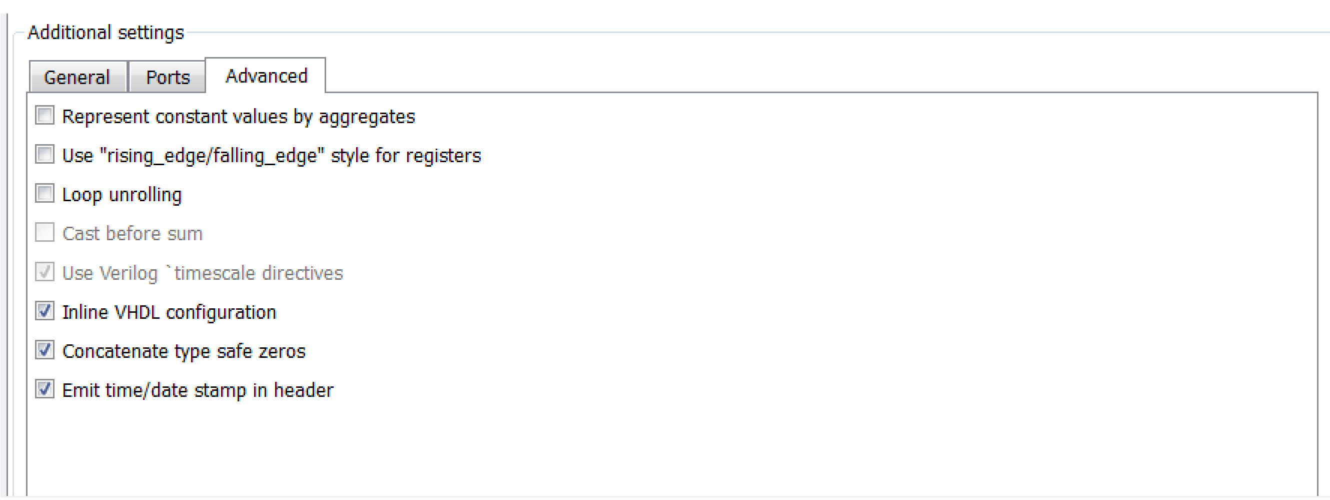
4 Select the **Ports** tab. The **Ports** pane appears.



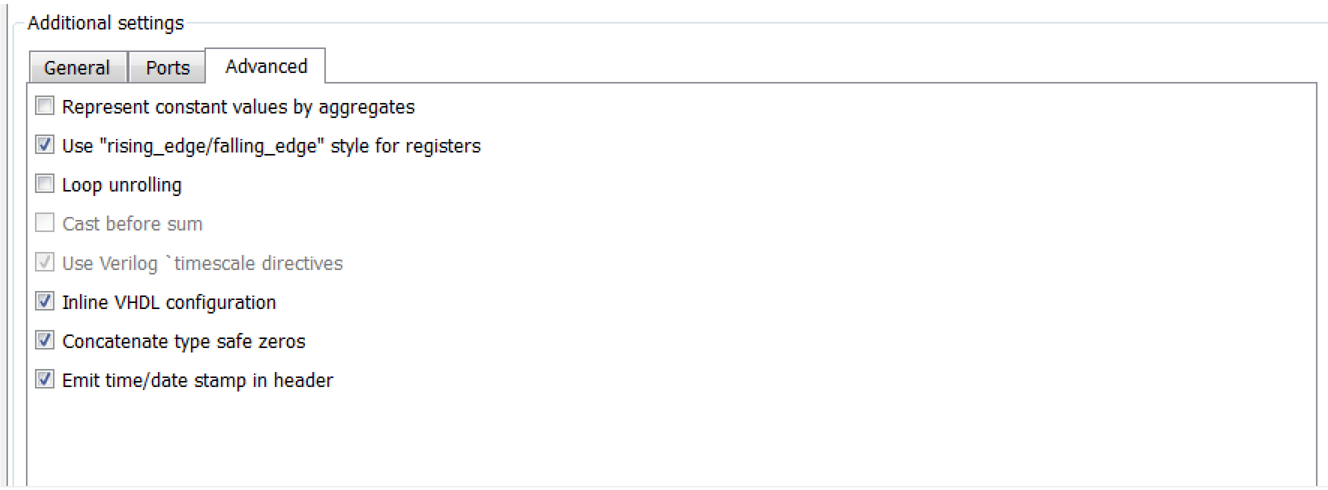
5 Clear the check box for the **Add output register** option. The **Ports** pane now appears as in this figure.



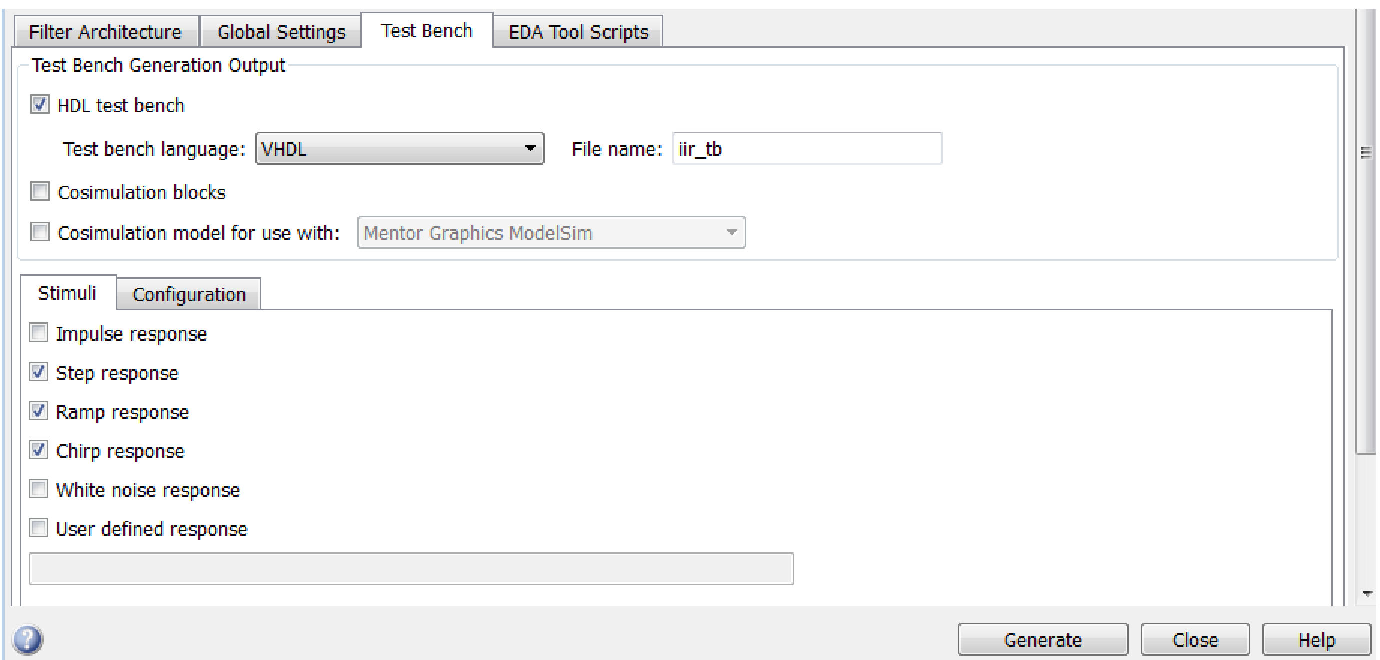
6 Select the **Advanced** tab. The **Advanced** pane appears.



7 Select the **Use 'rising_edge' for registers** option. The **Advanced** pane now appears as in this figure.



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



9 In the Generate HDL tool, click **Generate** to start the code generation process. When code generation completes, click **OK** to close the window.

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

```
### Starting VHDL code generation process for filter: iir
### Starting VHDL code generation process for filter: iir
### Generating: H:\hdlsrc\iir.vhd
### Starting generation of iir VHDL entity
### Starting generation of iir VHDL architecture
### Second-order section, # 1
### Second-order section, # 2
```

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

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

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

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

The generated VHDL code has these characteristics:

- VHDL entity named `iir`.
- Registers that use asynchronous resets when the reset signal is active high (1).
- The table shows the default names of the ports.

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 `coeffn`, where *n* is the coefficient number, starting with 1.
- Type-safe representation is used when zeros are concatenated: `'0' & '0'`...
- Registers are generated with the `rising_edge` function rather than the statement `ELSIF clk'event AND clk='1' THEN`.
- The postfix `'_process'` is appended to process names.

The generated test bench:

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

- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.
- Applies a hold time of 2 nanoseconds to data input signals.
- For an IIR filter, applies impulse, step, ramp, chirp, and white noise stimulus types.

Explore the Generated VHDL Code

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

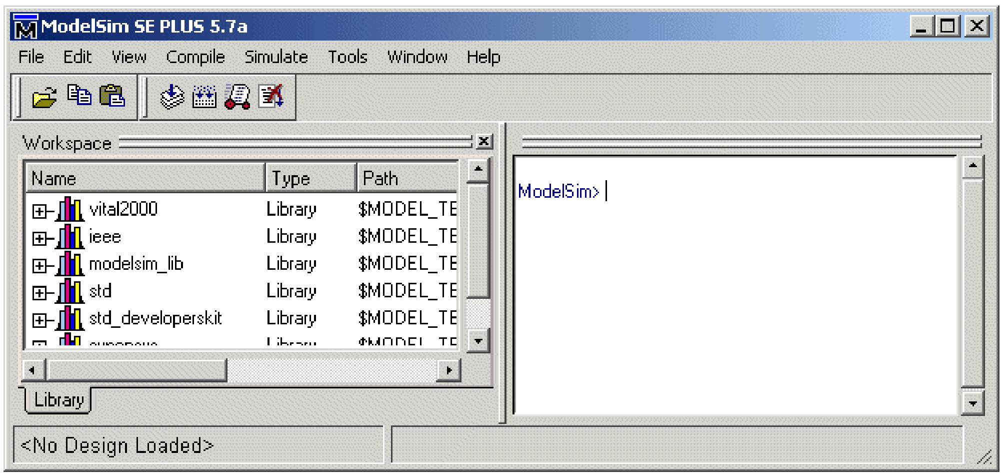
1. Open the generated VHDL filter file `iir.vhd`.
2. Search for `iir`. This line identifies the VHDL module, using the value you specified for the **Name** option in the **Target** pane. See step 2 in “Configure and Generate VHDL Code” on page 1-42.
3. Search for `Tutorial`. This section is where the coder places the text you entered for the **Comment in header** option. See step 5 in “Configure and Generate VHDL Code” on page 1-42.
4. Search for `HDL Code`. This section lists coder options you modified in “Configure and Generate VHDL Code” on page 1-42.
5. Search for `Filter Settings`. This section of the VHDL code describes the filter design and quantization settings as you specified in “Design an IIR Filter in Filter Designer” on page 1-38 and “Quantize the IIR Filter” on page 1-40.
6. Search for `ENTITY`. This line names the VHDL entity, using the value you specified for the **Name** option in the **Target** pane. See step 2 in “Configure and Generate VHDL Code” on page 1-42.
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 character vectors.
8. Search for `CONSTANT`. This code defines the coefficients. 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 code defines the signals of the filter.
10. Search for `input_reg_process`. The `PROCESS` block name `input_reg_process` includes the default `PROCESS` block postfix `'_process'`. This code reads the filter input from an input register. Code for this register is generated by default. In step 7 in “Configure and Generate VHDL Code” on page 1-42, you cleared the **Add output register** option, but left the **Add input register** option selected.
11. Search for `IF reset`. This code asserts the reset signal. 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 code checks for rising edges when the filter operates on registers. The `rising_edge` function is used as you specified in the **Advanced** pane of the Generate HDL tool. See step 10 in “Configure and Generate VHDL Code” on page 1-42.
13. Search for `Section 1`. This section 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 code drive the filter output data.

Verify the Generated VHDL Code

This section explains how to verify the generated VHDL code for the IIR filter with the generated VHDL test bench. This tutorial uses the Mentor Graphics ModelSim simulator as the tool for compiling and simulating the VHDL code. You can use other HDL simulation tool packages.

To verify the filter code, complete these steps.

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



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

```
cd hdlsrc
```

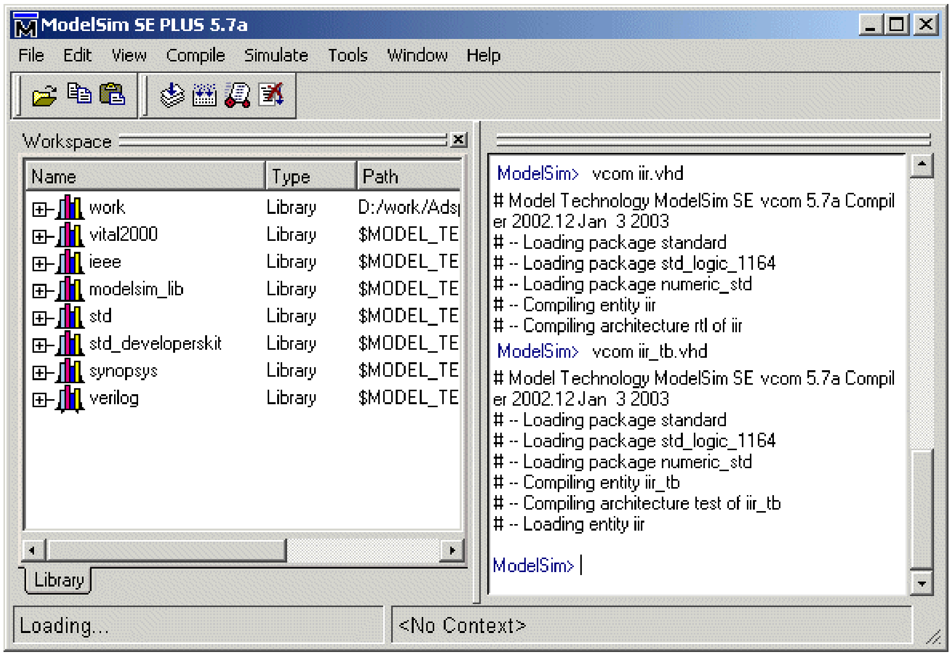
3 If desired, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

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

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

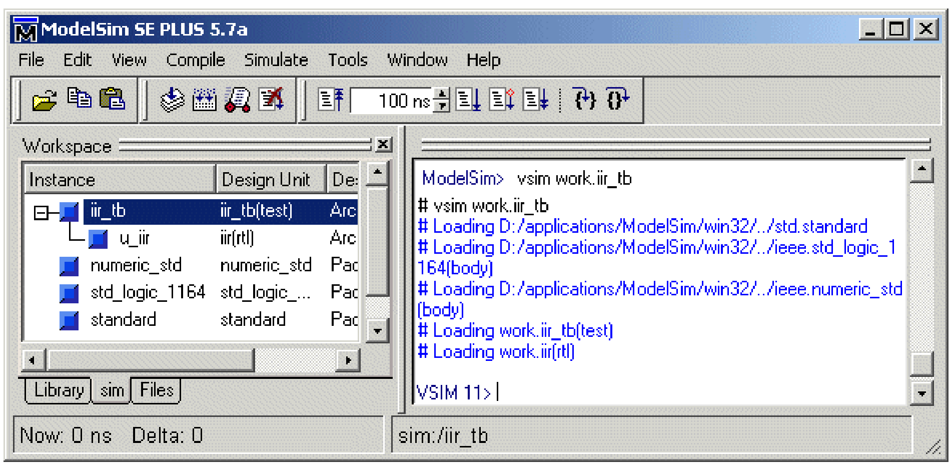
This figure shows the command sequence and informational messages displayed during compilation.



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

```
vsim work.iir_tb
```

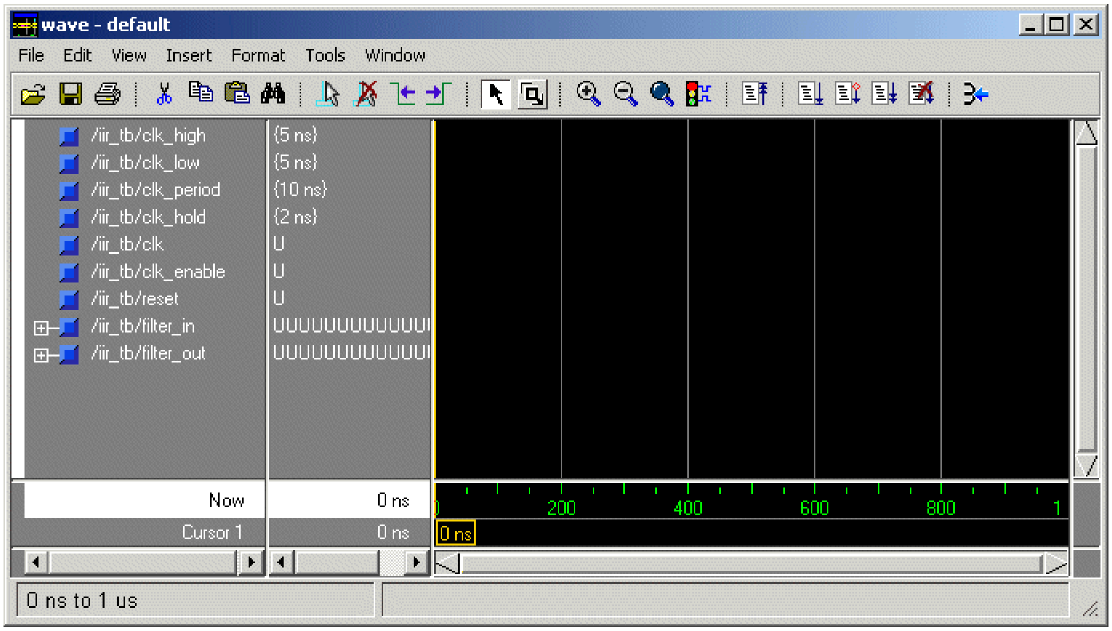
This figure shows the results of loading `work.iir_tb` with the `vsim` command.



6 Open a display window for monitoring the simulation as the test bench runs. In the Mentor Graphics ModelSim simulator, use this command to open a **wave** window and view the results of the simulation as HDL waveforms.

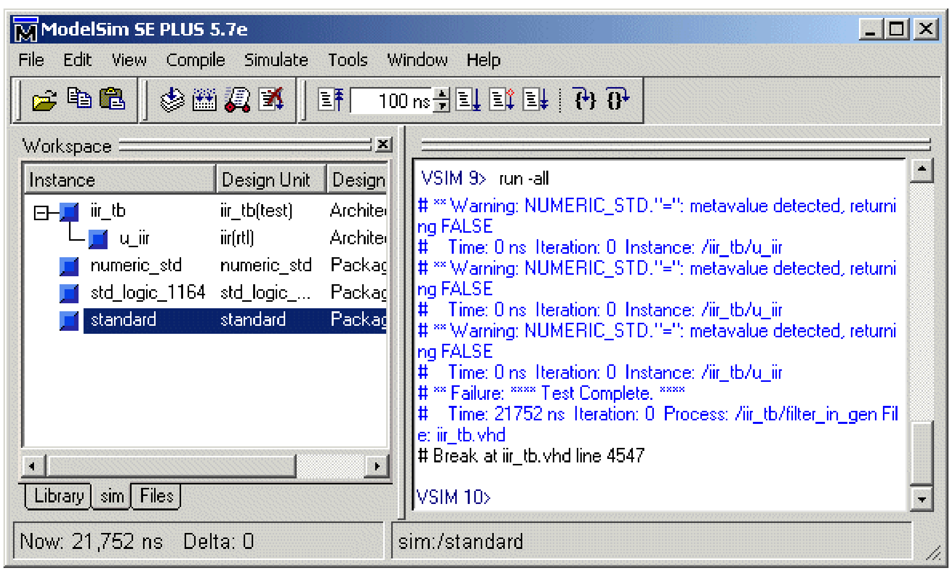
```
add wave *
```

This **wave** window appears.



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

This figure shows the `run -all` command being used to start a simulation.

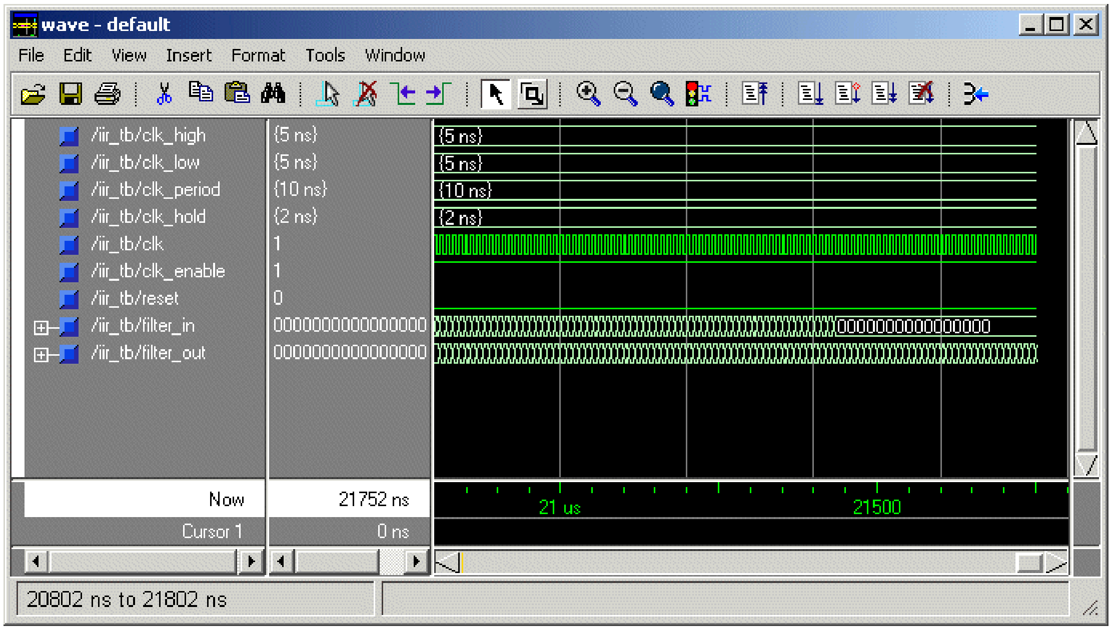


As your test bench simulation runs, watch for error messages. If error messages appear, interpret them as they pertain to your filter design and the HDL code generation options you selected. Determine whether the results are expected based on the customizations you specified when generating the filter 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 text`Test Complete`, the test bench has run to completion without encountering an error. The `Failure` part of the message is tied to the mechanism that the coder uses to end the simulation.

This **wave** window shows the simulation results as HDL waveforms.



2

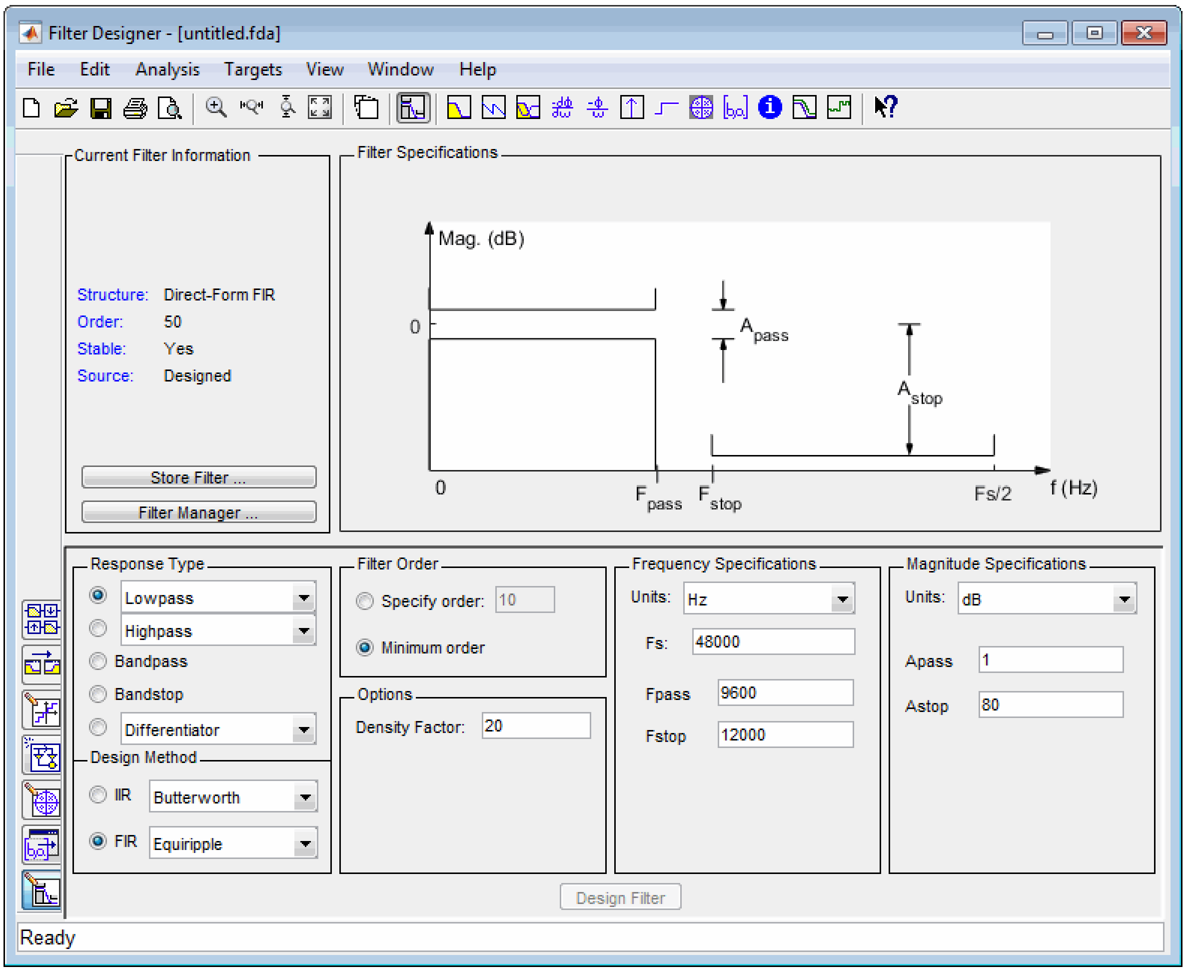
HDL Filter Code Generation Fundamentals

Starting Filter Design HDL Coder

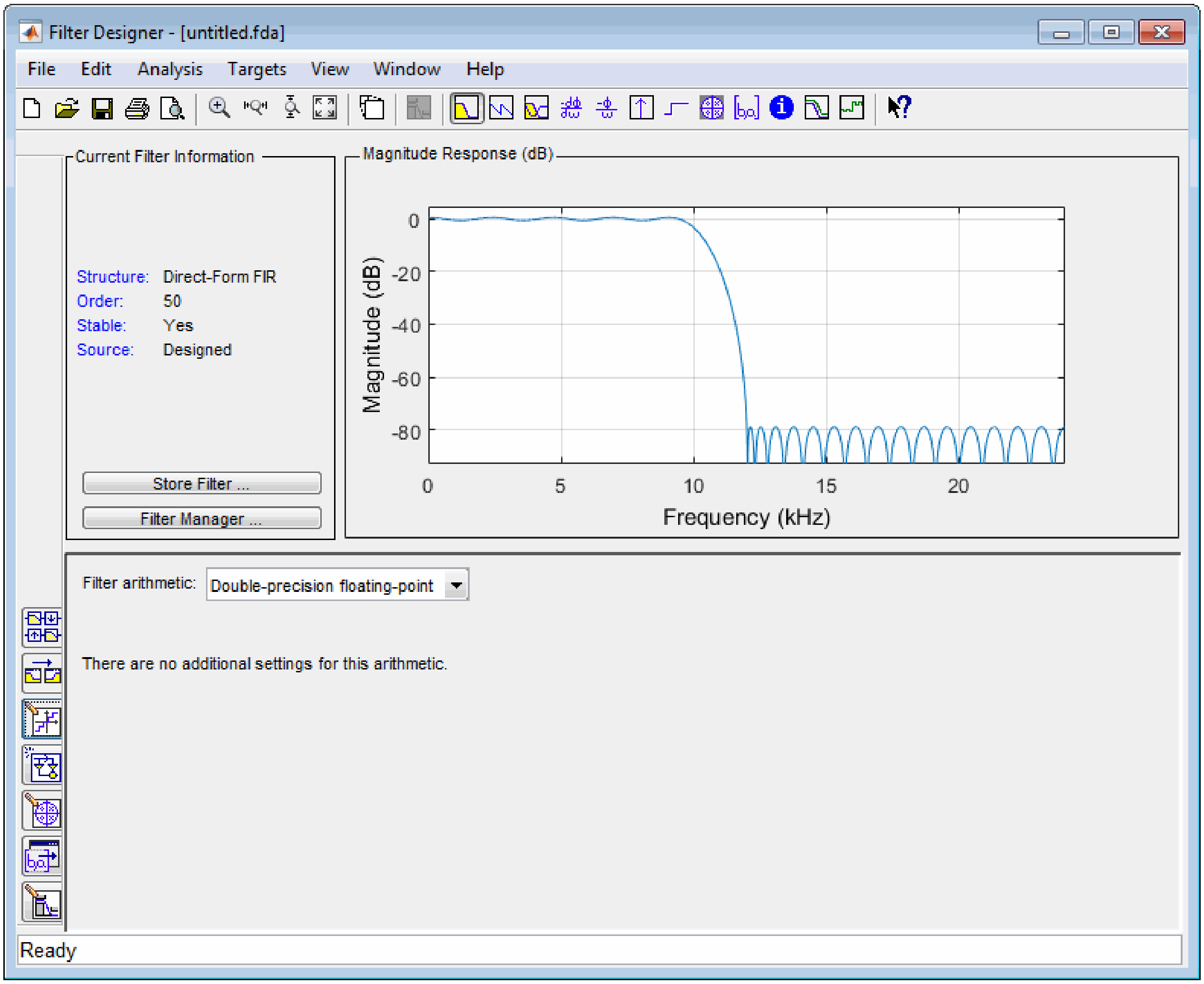
Opening the Filter Design HDL Coder UI from Filter Designer

To open the initial Generate HDL tool from Filter Designer, do this:

1. Enter the `filterDesigner` command at the MATLAB command prompt. The Filter Designer tool opens.

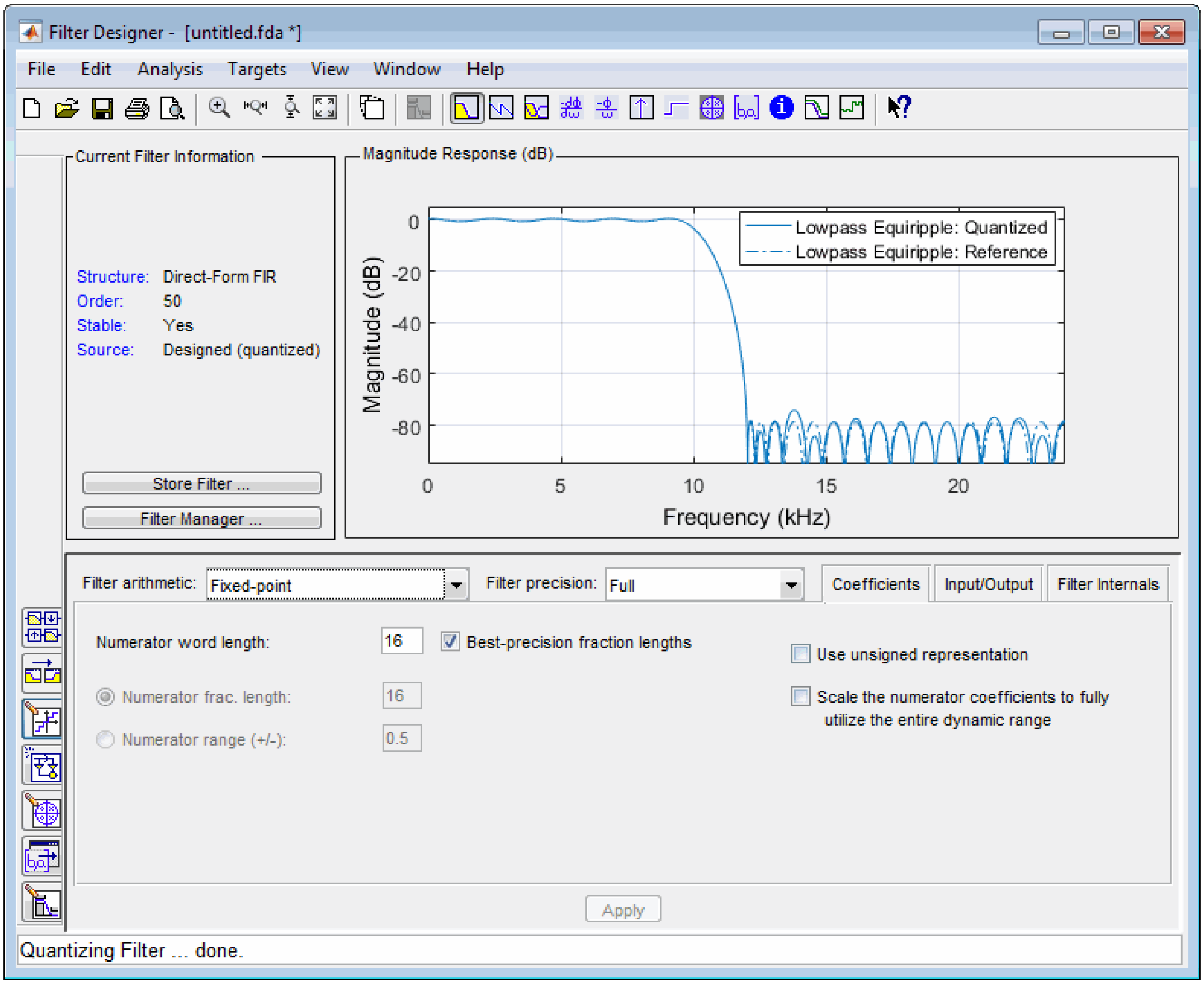


2. If the filter design is quantized, skip to step 3. Otherwise, quantize the filter by clicking the **Set Quantization Parameters** button . The **Filter arithmetic** menu appears in the bottom half of the window.

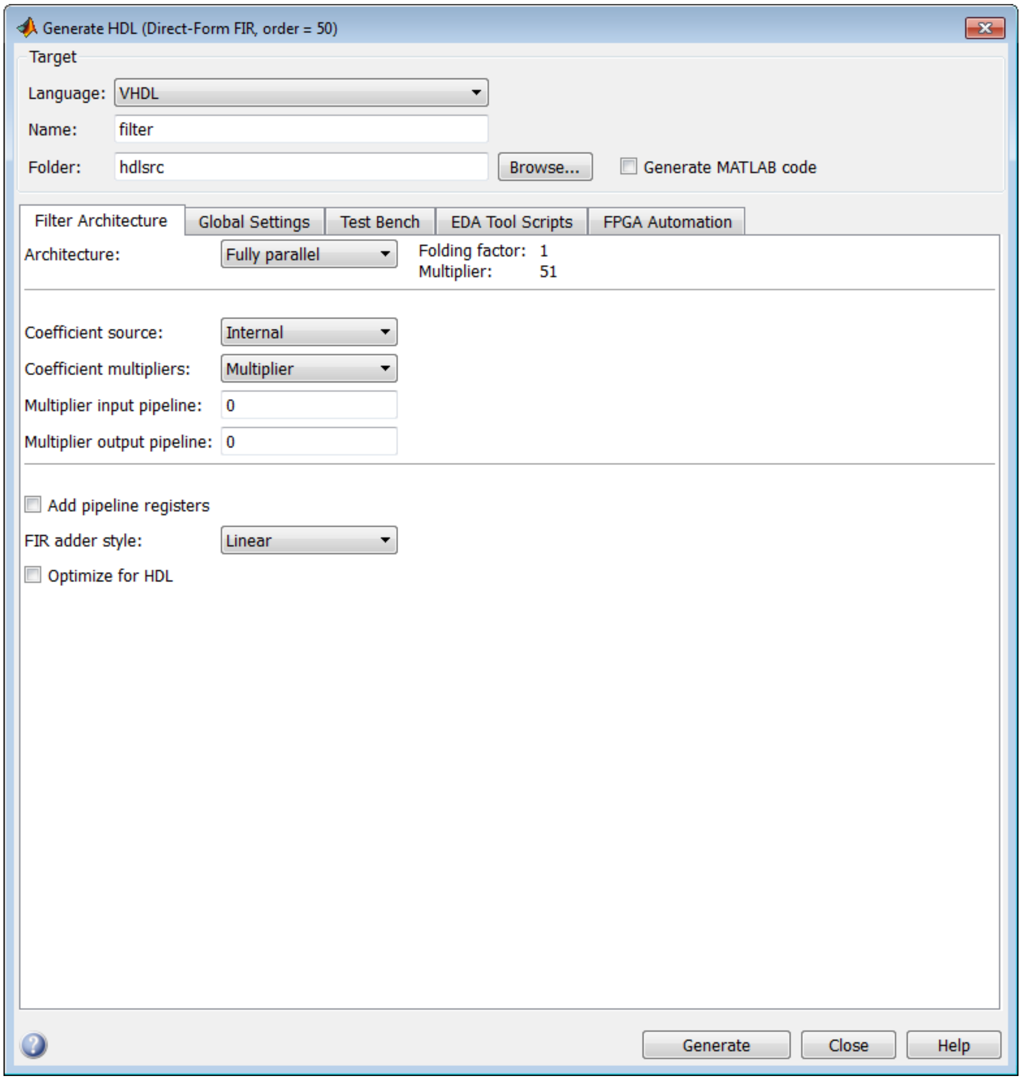


Note Supported filter structures allow both fixed-point and floating-point (double) realizations.

3 If desired, adjust the setting of the **Filter arithmetic** option. The Filter Designer displays the first of three tabbed panes.



4 Select **Targets** > **Generate HDL**. The Filter Designer displays the Generate HDL tool.



If the coder does not support the structure of the current filter in the Filter Designer, an error message appears.

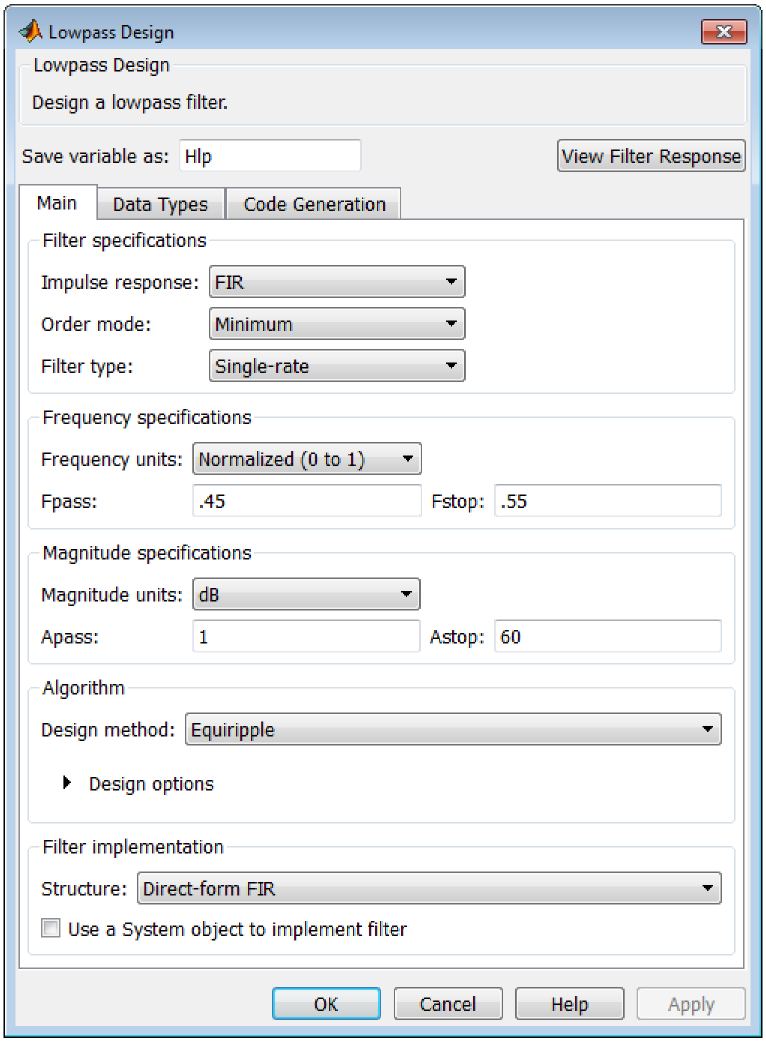
Opening the Filter Design HDL Coder UI from the Filter Builder

If you are not familiar with the Filter Builder UI, see the DSP System Toolbox documentation.

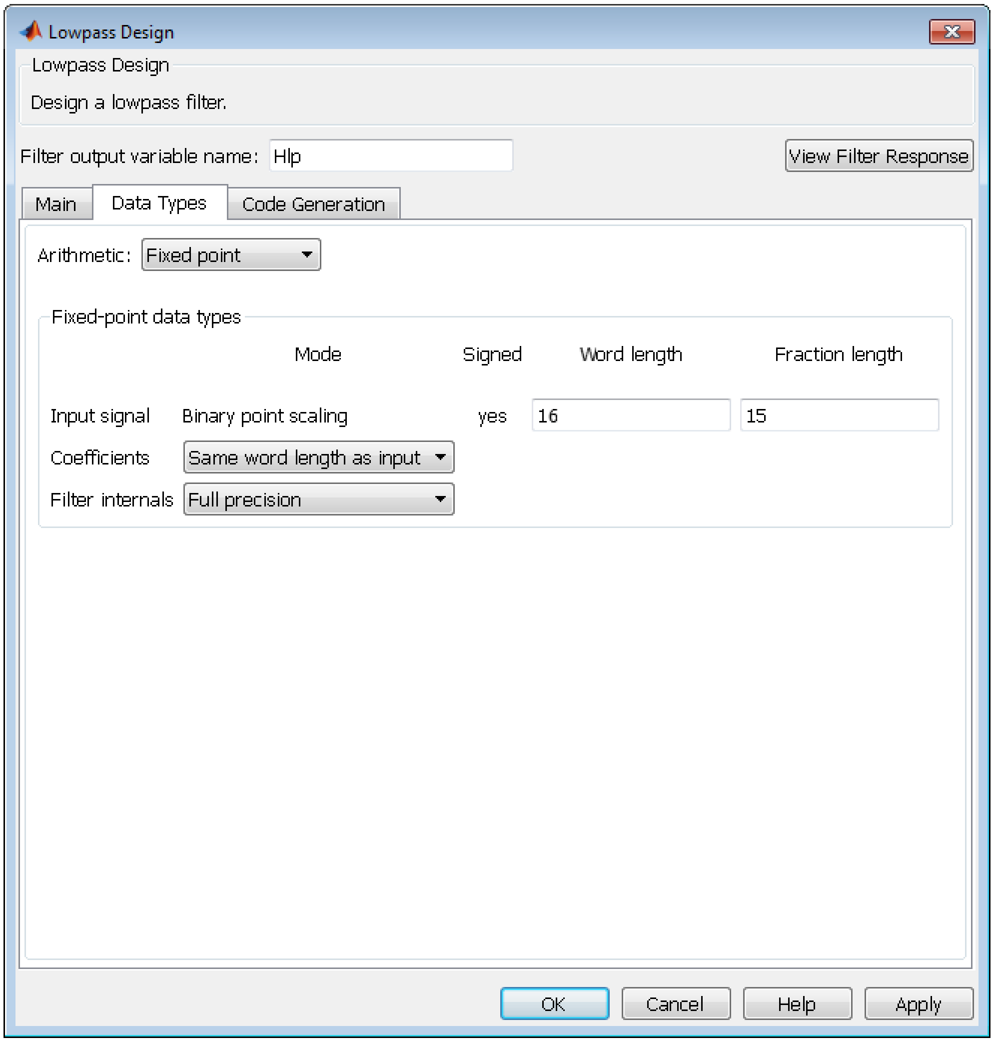
To open the Generate HDL tool from Filter Builder, do this:

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

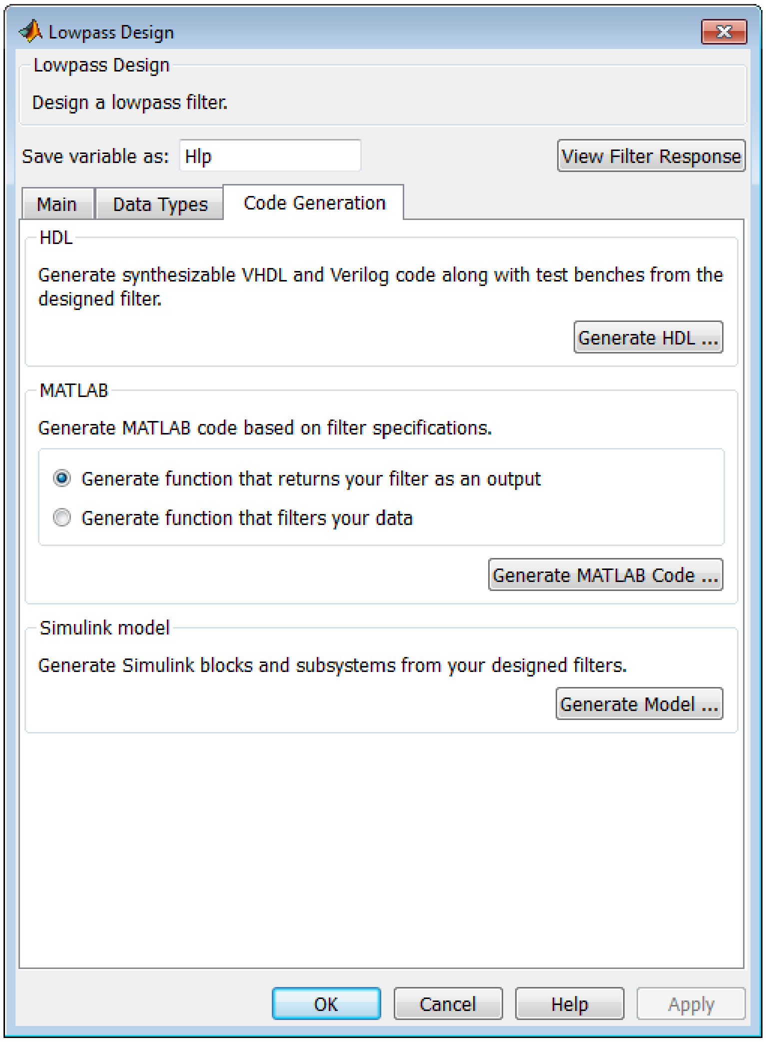
This figure shows the default settings of the main pane of the Filter Builder **Lowpass Design** dialog box.



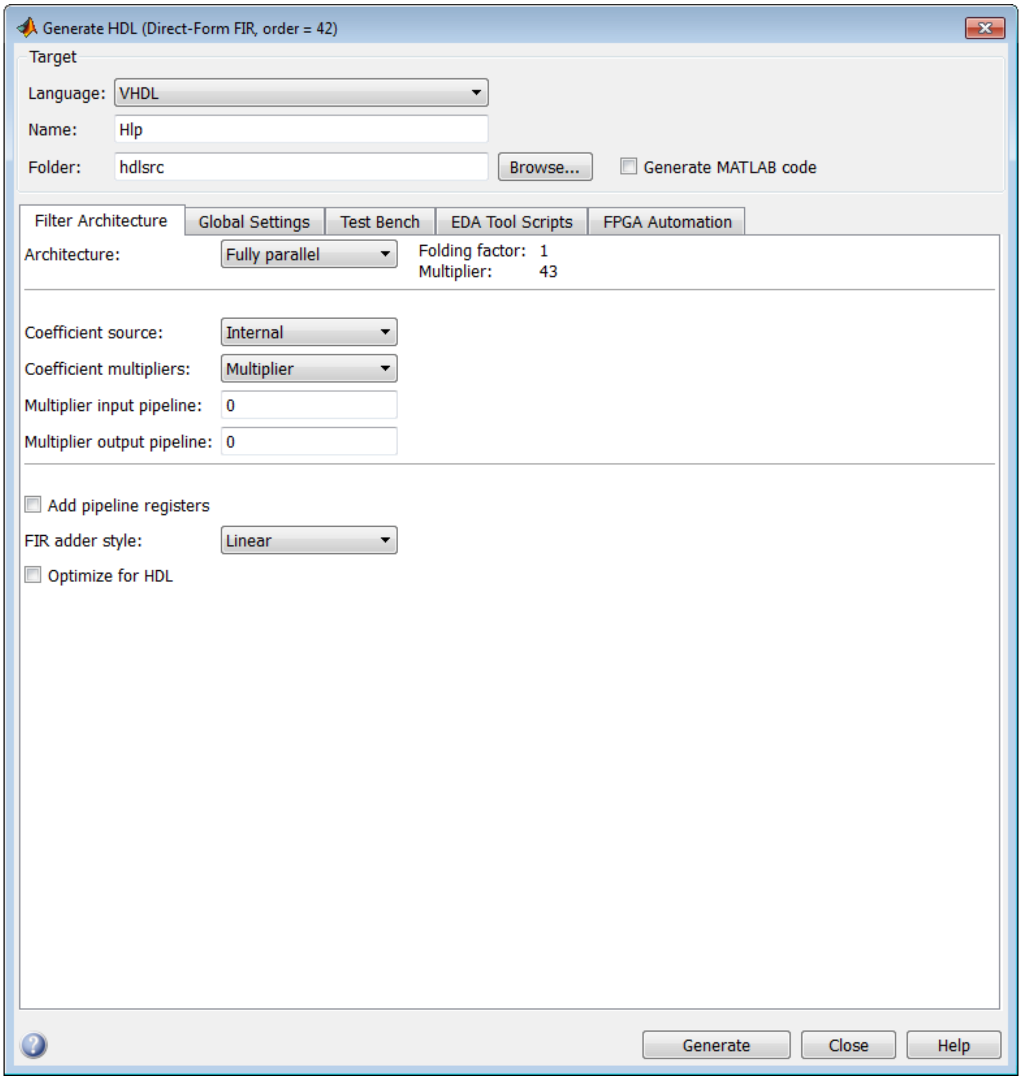
2 Set the filter design parameters as required.
3 Optionally, select the check box **Use a System object to implement filter**.
4 Click the **Data Types** tab. Set **Arithmetic** to `Fixed point` and select data types for internal calculations.



5 Click the **Code Generation** tab.



6 In the **Code Generation** pane, click the **Generate HDL** button. This button opens the Generate HDL tool, passing in the current filter object from Filter Builder.



7 Set the desired code generation and test bench options and generate code in the Generate HDL tool.

Opening the Filter Design HDL Coder UI Using the fdhdltool Command

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

```
fdhdltool(Hd)
```

where `Hd` is a type of filter object that is supported for HDL code generation. If the filter is a System object, you must specify the input data type.

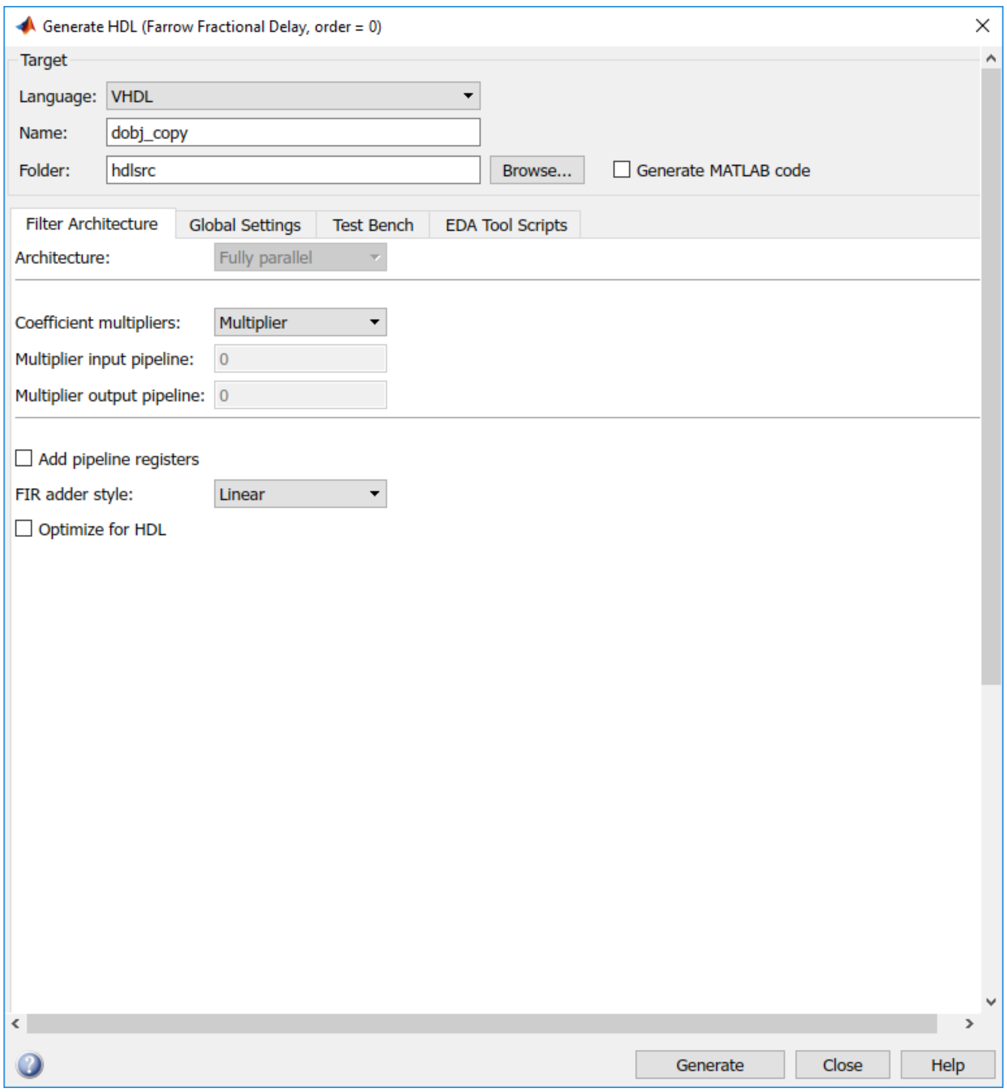
```
fdhdltool(FIRLowpass,numerictype(1,16,15))
```

The `fdhdltool` function is particularly useful when you must use the Filter Design HDL Coder UI to generate HDL code for filter structures that are not supported by Filter Designer or Filter Builder. For example, this code creates a Farrow fractional delay filter object `farrowfilt`, which is passed in to the `fdhdltool` function.

```
farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
inputDataType = numerictype(1,18,17);
fdDataType = numerictype(1,8,7);
fdhdltool(farrowfilt,inputDataType,fdDataType);
```

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

The name of the copied filter object by default is `dobj_copy`. This is reflected in the filter **Name** field. Likewise, the test bench file name is `dobj_tb_copy`. This is reflected in the **File name** field on the **Test Bench** pane. Update these default values to user-defined names if required.



Selecting Target Language

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

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

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

Generating HDL Code

Once your filter design and HDL settings are ready, generate HDL code for your design.

Applying Your Settings

When you generate HDL, either from the UI or the command line, the coder

- Applies code generation option settings that you have edited
- Generates HDL code and other requested files, such as a test bench.

Tip To preserve your coder settings, use the **Generate MATLAB code** option, as described in "Capturing Code Generation Settings" on page 2-14. **Generate MATLAB code** is available only in the UI. The function `generatehdl` does not have an equivalent property.

Generating HDL Code from the UI

This section assumes that you have opened the Generate HDL tool. See "Starting Filter Design HDL Coder" on page 2-2.

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

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

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

The messages include hyperlinks to the generated code and test bench files. Click these hyperlinks to open the code files in the MATLAB Editor.

Generate HDL From the Command Prompt

Design a filter.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60)
```

```
d = 
  lowpass with properties:

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4x1 cell}
```

```
    NormalizedFrequency: 1
                  Fpass: 0.2000
                  Fstop: 0.2200
                  Apass: 1
                  Astop: 60
```

```
Hd = design(d,'equiripple','filterstructure','dfsymfir','Systemobject',true)
```

```
Hd = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [-0.0011 -0.0017 -0.0025 -0.0031 -0.0031 -0.0024 -9.7703e-04 0.0010 0.0031
    InitialConditions: 0

  Show all properties
```

To generate HDL code for the filter and its test bench from the command line, use the `generatehdl` function. When you call the `generatehdl` function, specify the filter name and (optionally) desired property name and property value pairs. When the filter is a System object™, you must specify the input data type property.

As code generation proceeds, a sequence of messages appears in the MATLAB Command Window. The messages include hyperlinks to the generated code and test bench files. Click these hyperlinks to open the code files in the MATLAB Editor.

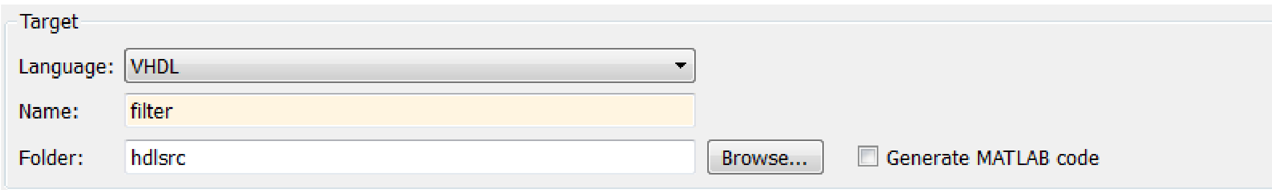
```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'Name','MyFilter',...
               'TargetLanguage','Verilog','GenerateHDLTestbench', 'on')
```

```
### Starting Verilog code generation process for filter: MyFilter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex97122369\hdlsrc\M
### Starting generation of MyFilter Verilog module
### Starting generation of MyFilter Verilog module body
### Successful completion of Verilog code generation process for filter: MyFilter
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 4486 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex971223
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
```

Capturing Code Generation Settings

To save your code generation settings, you can generate a script that includes the options you selected.

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



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

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

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

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

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

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

Note

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

See Also

More About

- "Generating HDL Code" on page 2-12

Closing Code Generation Session

The filter object in the workspace does not save the code generation settings. To preserve your coder settings, the best practice is to select the **Generate MATLAB code** option, as described in “Capturing Code Generation Settings” on page 2-14.

Click the **Close** button to close the Generate HDL tool and end a session with the coder.

See Also

More About

- “Starting Filter Design HDL Coder” on page 2-2
- “Generating HDL Code from the UI” on page 2-12

Generate HDL Code for Filter System Objects

You can generate HDL code for a supported filter System object by using the **Filter Builder** app, the Generate HDL tool, or by calling the `generatehdl` function. You can also explore filter architectures and generate test bench stimulus for a filter System object by using the `hdlfilterserialinfo`, `hdlfilterdainfo`, and `generatetbstimulus` functions. In either cases, you must specify a fixed-point data type for the System object. The HDL code generation tool quantizes the input signal to this data type.

Using Filter Builder

Open the **Filter Builder** app by calling the `filterBuilder` function, then set these options.

- On the **Main** tab, select **Use a System object to implement filter**.
- On the **Data Types** tab, set **Arithmetic** to `Fixed point` and select the internal fixed-point data types.
- On the **Code Generation** tab, click **Generate HDL** to set HDL code generation options and generate code.

Using Generate HDL

Open the Generate HDL tool by calling the `fdhdltool` function. When calling the function with a System object, specify the input data type as a `numerictype` object. Create this object by calling `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits. In this example, the call to `numerictype(1,8,7)` specifies a signed 8-bit number with 7 fractional bits.

```
filt =  dsp.BiquadFilter;
fdhdltool(filt,numerictype(1,8,7));
```

When the tool opens, you can set HDL code generation options and generate code for the System object.

At the Command Line

When calling the `generatehdl` function with a System object, specify the input data type as a `Name,Value` pair argument using the `InputDataType` property. Specify the property value as a `numerictype` object. For example:

```
filt =  dsp.BiquadFilter;
generatehdl(filt,'Name','HDLButter', ...
    'InputDataType',numerictype(1,8,7));
```

When calling `generatehdl`, you can set additional HDL code generation properties using `Name,Value` pair arguments. For example:

```
coeffs = fir1(22,0.45);
firfilt = dsp.FIRFilter('Numerator',coeffs, ...
    'Structure','Direct form antisymmetric');
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'SerialPartition',[7 4],'CoefficientMemory','DualPortRAMs', ...
    'CoefficientSource','ProcessorInterface');
```

See Also

`generatehdl` | `fdhdltool` | `numerictype` | `filterBuilder`

Related Examples

- “Supported Filter System Objects” on page 1-4
- “HDL Butterworth Filter” on page 8-2
- “HDL Inverse Sinc Filter” on page 8-8
- “HDL Tone Control Filter Bank” on page 8-20
- “HDL Sample Rate Conversion Using Farrow Filters” on page 8-53

HDL Code for Supported Filter Structures

Multirate Filters

Supported Multirate Filter Types

HDL code generation is supported for these types of multirate filters:

- Cascaded Integrator Comb (CIC) Interpolator (`dsp.CICInterpolator`)
- Cascaded Integrator Comb (CIC) Decimator (`dsp.CICDecimator`)
- FIR Polyphase Decimator (`dsp.FIRDecimator`)
- FIR Polyphase Interpolator (`dsp.FIRInterpolator`)
- FIR Polyphase Sample Rate Converter (`dsp.FIRRateConverter`)
- CIC Compensation Interpolator (`dsp.CICCompensationInterpolator`)
- CIC Compensation Decimator (`dsp.CICCompensationDecimator`)

Generating Multirate Filter Code

To generate multirate filter code, first select and design one of the supported filter types using Filter Designer, Filter Builder, or the MATLAB command line.

After you have created the filter, open the Generate HDL tool, set the desired code generation properties, and generate code. See "Code Generation Options for Multirate Filters" on page 3-2.

To generate code using the `generatehdl` function, specify multirate filter code generation properties that are functionally equivalent to the UI options. See "generatehdl Properties for Multirate Filters" on page 3-5.

Code Generation Options for Multirate Filters

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

- On the **Global settings** tab, the **Clock inputs** pull-down menu is enabled. This menu provides two alternatives for generating clock inputs for multirate filters.

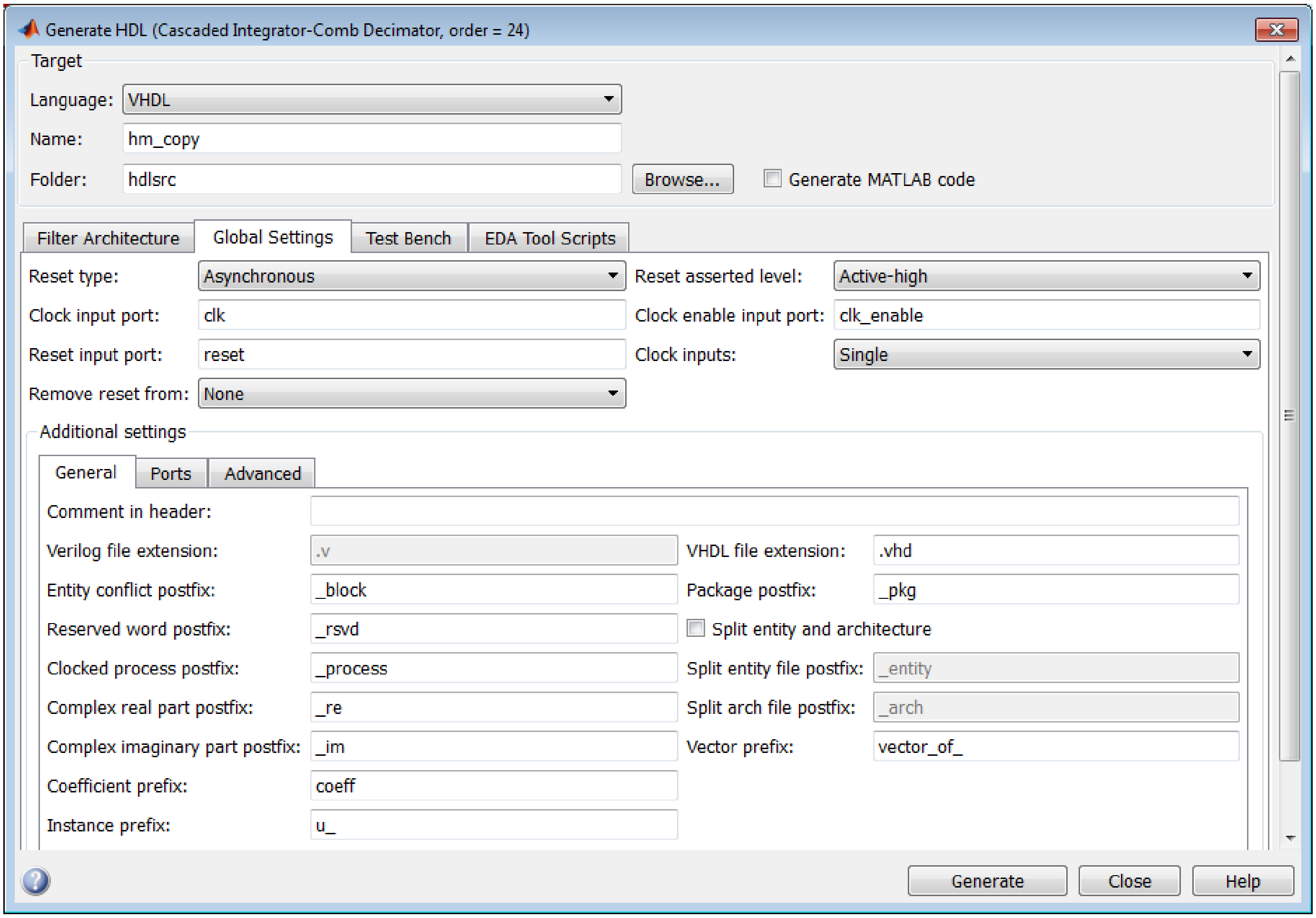
  Note The **Clock inputs** menu is not supported for:

  - Filters with a `Partly serial` architecture
  - Multistage sample rate converters: `dsp.FIRRateConverter`, or `dsp.FilterCascade` containing multiple rates

  ---

- For CIC filters, on the **Filter Architecture** tab, the **Coefficient multipliers** option is disabled. Coefficient multipliers are not used in CIC filters.
- For CIC filters, on the **Filter Architecture** tab, the **FIR adder style** option is disabled, since CIC filters do not require a final adder.

This figure shows the default settings of the Generate HDL tool options for a supported CIC filter.



The **Clock inputs** options are:

- `Single`: When you select `Single`, the coder generates a single clock input for a multirate filter. The module or entity declaration for the filter has a single clock input with an associated clock enable input, and a clock enable output. The generated code includes a counter that controls the timing of data transfers to the filter output (for decimation filters) or input (for interpolation filters). The counter behaves as a secondary clock whose rate is determined by the decimation or interpolation factor. This option provides a self-contained clocking solution for FPGA designs.

  To customize the name of the clock enable output, see "Setting the Clock Enable Output Name" on page 3-5. Interpolators also pass through the clock enable input signal to an output port named `ce_in`. This signal indicates when the object accepted an input sample. You can use this signal to control the upstream data flow. You cannot customize this port name.

  This code was generated from a CIC decimation filter having a decimation factor of 4, with **Clock inputs** set to `Single`.

  The coder generates an input clock, input clock enable, and an output clock enable.

```
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 clock enable output process, `ce_output`, maintains the signal `counter`. Every 4th clock cycle, `counter` toggles 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';
```

This code illustrates a typical use of the `counter` signal 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 you select `Multiple`, the coder generates multiple clock inputs for a multirate filter. The module or entity declaration for the filter has separate clock inputs (each with an associated clock enable input) for each rate of a multirate filter. You are responsible for providing input clock signals that correspond to the desired 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.

  The `Multiple` option is intended for ASICs and FPGAs. It provides more flexibility than the `Single` option, but assumes that you provide higher-level HDL code to drive the input clocks of your filter.

  Synchronizers between multiple clock domains are not provided.

  When you select `Multiple`, the coder does not generate clock enable outputs; therefore the **Clock enable output port** field of the **Global Settings** pane is disabled.

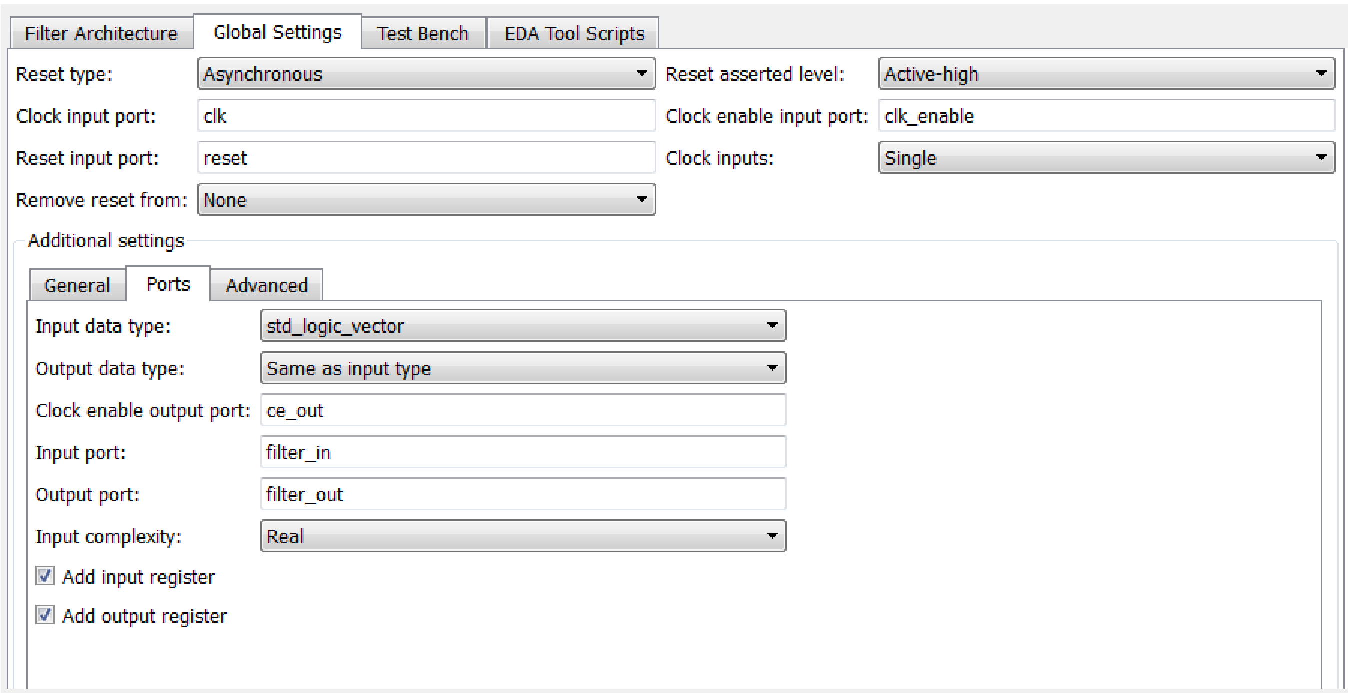
  This `ENTITY` declaration was generated from a CIC decimation filter with **Clock inputs** set to `Multiple`.

```
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

The coder generates a clock enable output when you set **Clock inputs** to `Single` in the Generate HDL tool. The default name for the clock enable output is `ce_out`.

To change the name of the clock enable output, modify the **Clock enable output port** field of the **Ports** pane of the Generate HDL tool.



The coder enables the **Clock enable output port** field only when generating code for a multirate filter with a single input clock.

generatehdl Properties for Multirate Filters

If you are using `generatehdl` to generate code for a multirate filter, you can set these 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.
- `ClockEnableInputPort` corresponds to the **Clock enable input port** field; specifies the name of the clock enable input port.

Variable Rate CIC Filters

Supported Variable Rate CIC Filter Types

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

- CIC Decimator (`dsp.CICDecimator`)
- CIC Interpolator (`dsp.CICInterpolator`)
- Multirate cascade with one CIC stage (`dsp.FilterCascade`)

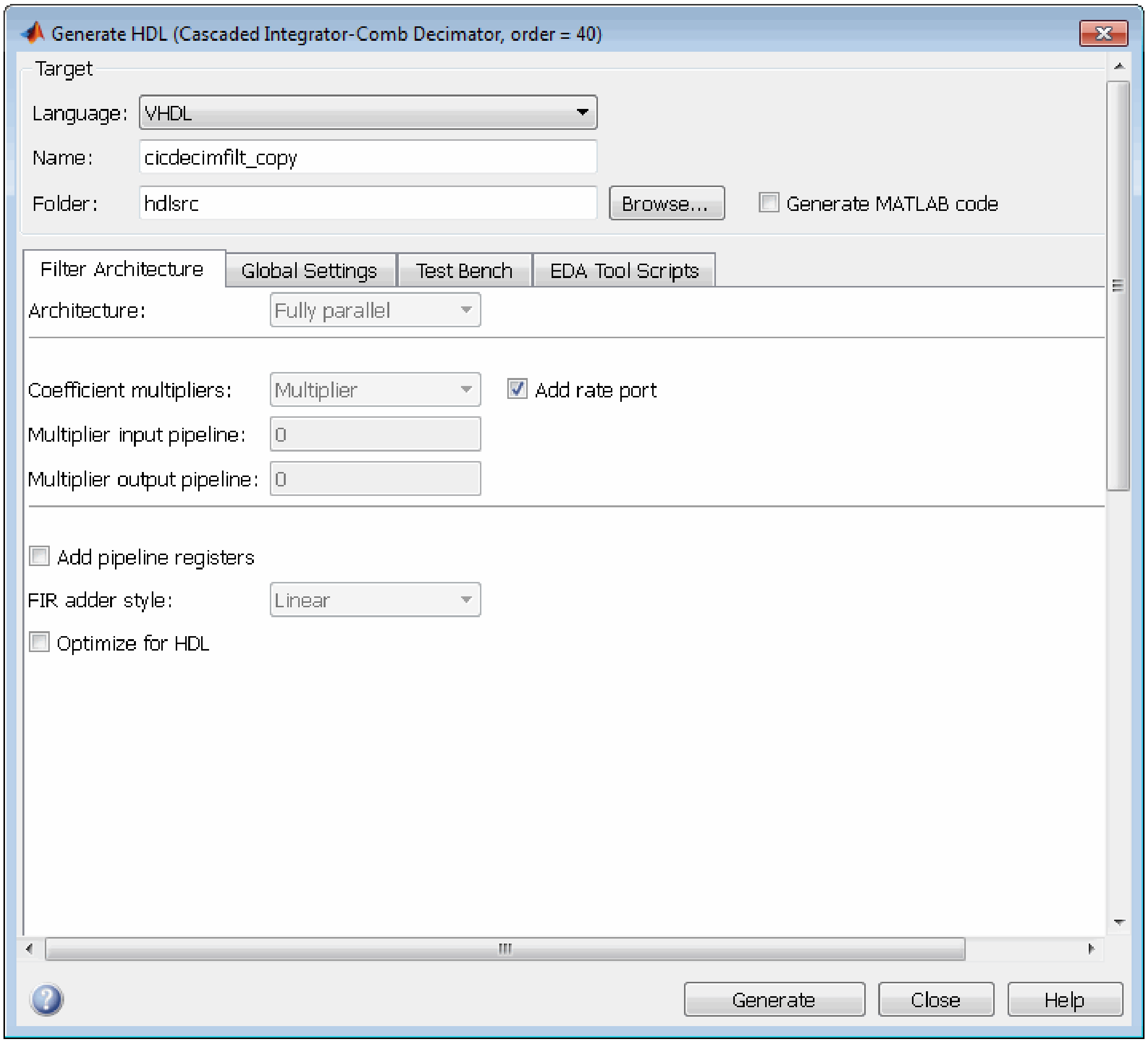
Code Generation Options for Variable Rate CIC Filters

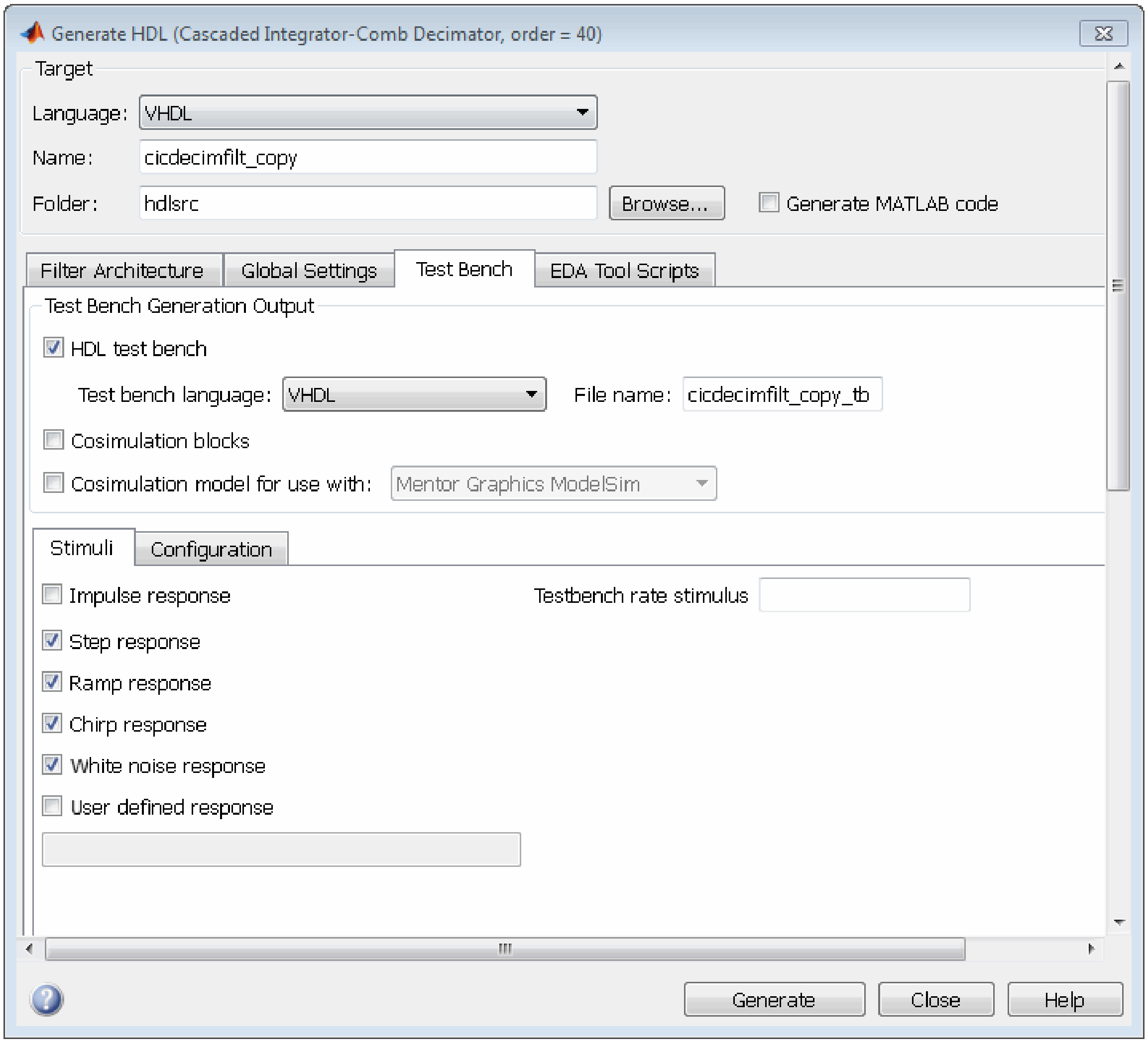
A variable rate CIC filter has a programmable rate change factor. The coder assumes that the filter is designed with the maximum rate expected, and that the Decimation Factor (for CIC Decimators) or Interpolation Factor (for CIC Interpolators) is set to this maximum ratio.

Two properties support variable rate CIC filters:

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

You can also specify these properties in the UI using the **Add rate port** check box and the **Testbench rate stimulus** edit box.


Generate HDL (Cascaded Integrator-Comb Decimator, order = 40)
Target
Language: VHDL
Name: cicdecimfilt_copy
Folder: hdlsrc
Browse...
Generate MATLAB code
Filter Architecture
Global Settings
Test Bench
EDA Tool Scripts
Architecture: Fully parallel
Coefficient multipliers: Multiplier
Add rate port
Multiplier input pipeline: 0
Multiplier output pipeline: 0
Add pipeline registers
FIR adder style: Linear
Optimize for HDL
Generate
Close
Help


Generate HDL (Cascaded Integrator-Comb Decimator, order = 40)
Target
Language: VHDL
Name: cicdecimfilt_copy
Folder: hdlsrc
Browse...
Generate MATLAB code
Filter Architecture
Global Settings
Test Bench
EDA Tool Scripts
Test Bench Generation Output
HDL test bench
Test bench language: VHDL
File name: cicdecimfilt_copy_tb
Cosimulation blocks
Cosimulation model for use with:
Mentor Graphics ModelSim
Stimuli
Configuration
Impulse response
Testbench rate stimulus
Step response
Ramp response
Chirp response
White noise response
User defined response
Generate
Close
Help

Cascade Filters

In this section...

Supported Cascade Filter Types

The coder supports code generation for a multirate cascade of filter objects (`dsp.FilterCascade`).

Generating Cascade Filter Code

Instantiate the filter stages and cascade them in the MATLAB workspace.

```
hm1 = dsp.FIRDecimator('DecimationFactor',12);
hm2 = dsp.FIRDecimator('DecimationFactor',4);
my_cascade = dsp.FilterCascade(hm1,hm2);
```

For usage details, see `dsp.FilterCascade` in the DSP System Toolbox documentation.

The coder currently imposes certain limitations on the filter types allowed in a cascade filter. See "Limitations for Code Generation with Cascade Filters" on page 3-9 before creating your filter stages and cascade filter object.

Generating Cascade Filter Code with the fdhdltool Function

Call `fdhdltool` to open the Generate HDL tool, passing in the cascade filter System object and the fixed-point input data type.

```
fdhdltool(my_cascade,numerictype(1,16,15))
```

Set the desired code generation properties and click the **Generate** button to generate code.

Generating Cascade Filter Code with the generatehdl Function

Call `generatehdl` to generate HDL code for your filter, passing in the cascade filter System object, the fixed-point input data type, and code generation properties as desired.

```
generatehdl(my_cascade,'InputDataType',numerictype(1,16,15), ...
    'Name','MyFilter','TargetLanguage','Verilog', ...
    'GenerateHDLTestbench','on')
```

Limitations for Code Generation with Cascade Filters

These rules and limitations apply to cascade filters when used for code generation.

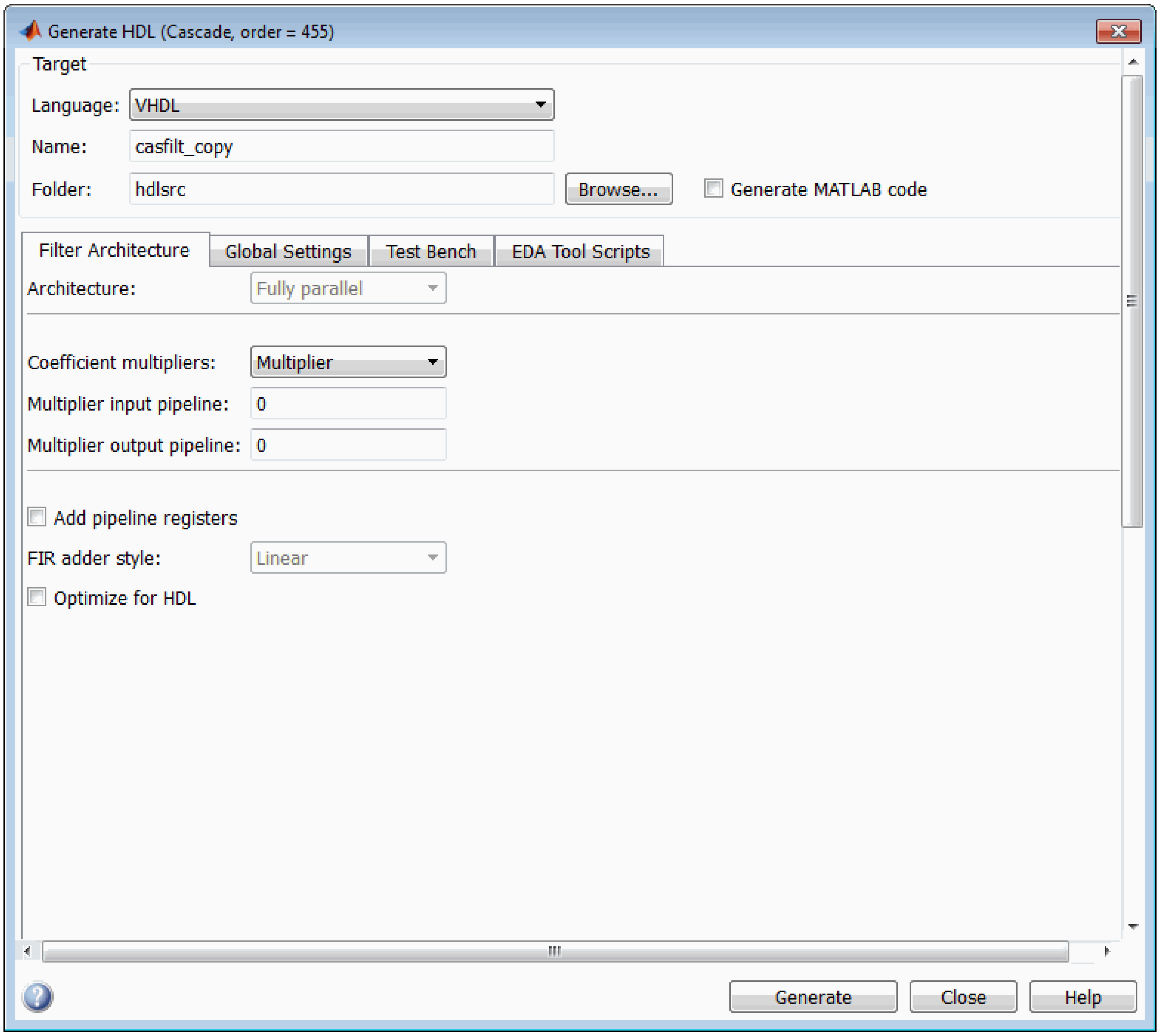
- You can generate code for cascades that combine theses filter types:
  - Decimators and/or single-rate filter structures
  - Interpolators and/or single-rate filter structures

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

- For code generation, only a flat (single-level) cascade structure is allowed. Nesting of cascade filters is disallowed.
- By default, generated HDL code excludes the input and output registers from the stages of the cascade, except for:
  - The input of the first stage and the output of the final stage.
  - The input registers of interpolator stages.

  To generate output registers for each stage, select the **Add pipeline registers** option in the Generate HDL tool. When using this option, internal pipeline registers might also be added, depending on the filter structures.
- When a cascade filter is passed to `fdhdltool`, the **FIR adder style** option is disabled. If you require tree adders for FIR filters in a cascade, select the **Add pipeline registers** option (since pipelines require tree style FIR adders).
- The coder generates separate HDL code files for each stage of the cascade, in addition to the top-level code for the cascade filter itself. The filter stage code files are identified by appending the character vector `'_stage1'`, `'_stage2'`, ... `'_stageN'` to the filter name.

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


Generate HDL (Cascade, order = 455)
Target
Language: VHDL
Name: casfilt_copy
Folder: hdlsrc
Browse...
Generate MATLAB code
Filter Architecture
Global Settings
Test Bench
EDA Tool Scripts
Architecture: Fully parallel
Coefficient multipliers: Multiplier
Multiplier input pipeline: 0
Multiplier output pipeline: 0
Add pipeline registers
FIR adder style: Linear
Optimize for HDL
Generate
Close
Help

Polyphase Sample Rate Converters

Code Generation for Polyphase Sample Rate Converter

The coder supports HDL code generation for direct form FIR polyphase sample rate converters. `dsp.FIRRateConverter` is a multirate filter structure that combines an interpolation factor and a decimation factor. This combination enables you to perform fractional interpolation or decimation on an input signal.

The interpolation factor (`l`) and decimation factor (`m`) for a polyphase sample rate converter are specified as integers in the `InterpolationFactor` and `DecimationFactor` properties of a `dsp.FIRRateConverter` System object. This code constructs an object with a resampling ratio of 5/3:

```
frac_cvrter = dsp.FIRRateConverter('InterpolationFactor',5, ...
    'DecimationFactor',3)
```

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

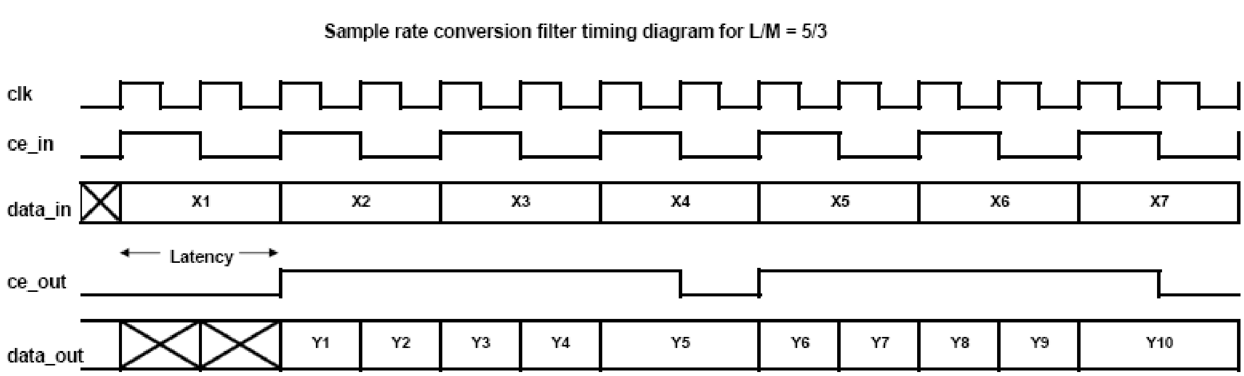
For general information on this filter structure, see the `dsp.FIRRateConverter` reference page in the DSP System Toolbox documentation.

HDL Implementation for Polyphase Sample Rate Converter

Signal Flow, Latency, and Timing

The signal flow for the `dsp.FIRRateConverter` filter is similar to the polyphase FIR interpolator (`dsp.FIRInterpolator`). The delay line is advanced to deliver each input after the required set of polyphase coefficients are processed.

The diagram illustrates the timing of the HDL implementation for `dsp.FIRRateConverter`. A clock enable input (`ce_in`) indicates valid input samples. The output data, and a clock enable output (`ce_out`), are produced and delivered simultaneously, which results in a nonperiodic output.



Clock Rate

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

```
ceil(InterpolationFactor/DecimationFactor)
```

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

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

Clock Enable Ports

The HDL entity or module generated from the `dsp.FIRRateConverter` filter has one input and two output clock enable ports:

- Clock enable outputs: The default clock enable output port name is `ce_out`. This signal indicates when the output data sample is valid. As with other multirate filters, you can use the **Clock enable output port** field on the **Global Settings** > **Ports** tab of the Generate HDL tool to specify the port name. Alternatively, you can use the `ClockEnableOutputPort` property to set the port name in the `generatehdl` command.

  The filter also passes through the clock enable input to an output port named `ce_in`. This signal indicates when the object accepted an input sample. You can use this signal to control the upstream data flow. You cannot customize this port name.
- Clock enable input: The default clock enable input port name is `clk_enable`. This signal indicates when the input data sample is valid. You can use the **Clock enable input port** field on the **Global Settings** tab of the Generate HDL tool to specify the port name. Alternatively, you can use the `ClockEnableInputPort` property to set the port name in the `generatehdl` command.

Test Bench Generation

Generated test benches apply the test vectors at the correct rate, then observe and verify the output as it is available. The test benches control the data flow using the input and output clock enables.

Code Generation

This example constructs a fixed-point `dsp.FIRRateConverter` object with a resampling ratio of 5/3, and generates VHDL filter code. When you generate HDL code for a System object, specify the input fixed-point data type. The object determines internal data types based on the input type and property settings.

```
frac_cvrter = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3)
generatehdl(frac_cvrter,'InputDataType',numerictype(1,16,15))
```

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

These code generation options are not supported for `dsp.FIRRateConverter` filters:

- Use of pipeline registers (AddPipelineRegisters)
- Distributed Arithmetic architecture (DARadix and (DALUTPartition))
- Fully or partially serial architectures (SerialPartition and ReuseAccum)
- Multiple clock inputs (ClockInputs)

Multirate Farrow Sample Rate Converters

In this section...

Code Generation for Multirate Farrow Sample Rate Converters

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

Unlike a single-rate Farrow filter (see "Single-Rate Farrow Filters" on page 3-18), a multirate Farrow sample rate converter does not have a fractional delay input. For general information on this filter structure, see the `dsp.FarrowRateConverter` reference page in the DSP System Toolbox documentation.

Generating Code for dsp.FarrowRateConverter Filters at the Command Line

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

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

```
Hm = dsp.FarrowRateConverter(48,44.1);
generatehdl(Hm,'InputDataType',numerictype(1,16,15),...
               'GenerateHDLTestbench','on')
```

This example generates HDL code for a cascade that includes a `dsp.FarrowRateConverter` filter. The coder requires that the `dsp.FarrowRateConverter` filter is in the last position of the cascade.

First, interpolate the original 8-kHz signal by four, using a cascade of FIR halfband filters.

```
Astop = 50;         % Minimum stopband attenuation
TW = .125;          % Transition Width
f2 = fdesign.interpolator(4,'Nyquist',4,'TW,Ast',TW,Astop);
hfir = design(f2,'multistage','HalfbandDesignMethod','equiripple','Systemobject',true);
```

Then, interpolate the intermediate 32-kHz signal to get the designer 44.1-kHz sampling frequency. The `dsp.FarrowRateConverter` System object calculates a piecewise polynomial fit using Lagrange interpolation coefficients.

```
N = 3;  % Polynomial Order
hfar = dsp.FarrowRateConverter(32,44.1,'PolynomialOrder',N)
```

Obtain the overall filter by cascading the FIR phases with the Farrow stage. The `dsp.FarrowRateConverter` filter is at the end of the cascade.

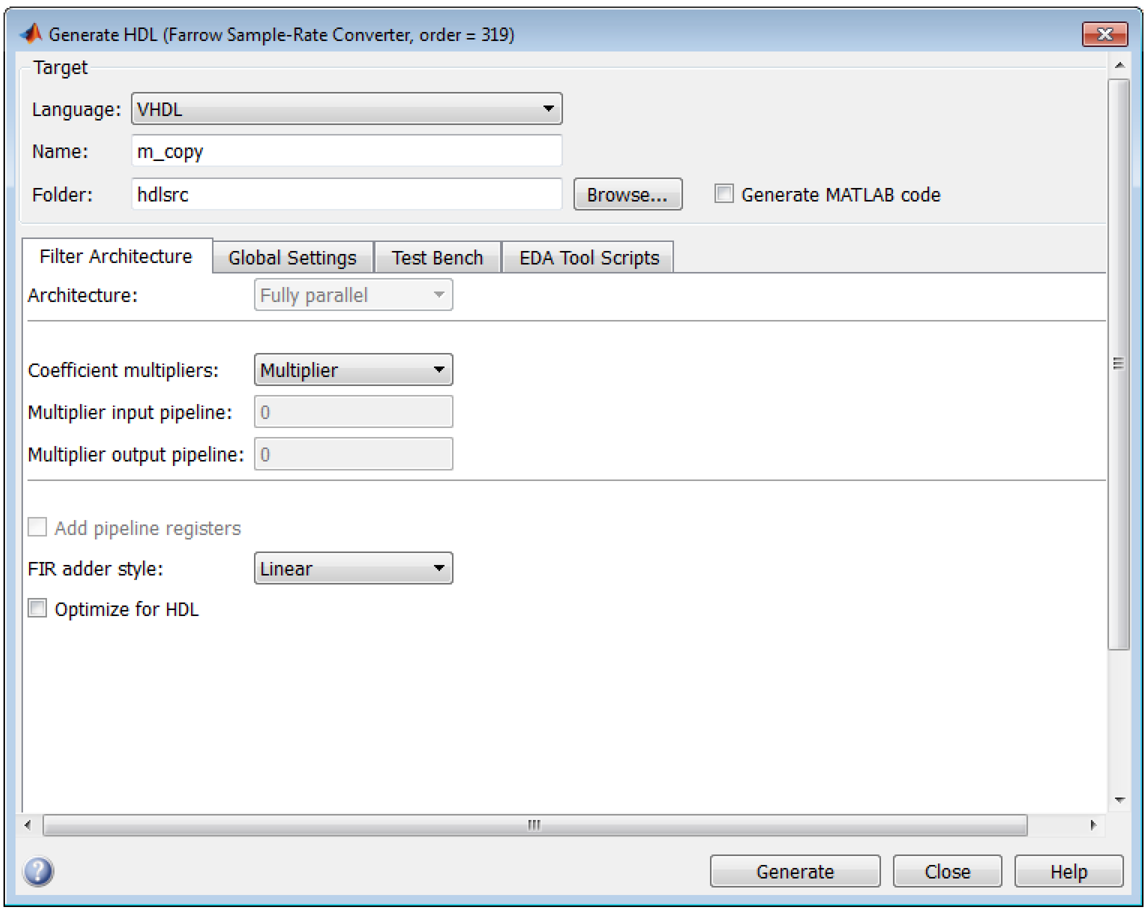
```
interp_cascade.addStage(hfar);
generatehdl(interp_cascade,'InputDataType',numerictype(1,16,15),...
                           'GenerateHDLTestbench','on');
```

Generating Code for dsp.FarrowRateConverter Filters in the UI

`filterDesigner` and `filterBuilder` do not currently support `dsp.FarrowRateConverter` filters. To generate code for a `dsp.FarrowRateConverter` filter in the HDL code generation UI, use the `fdhdltool` command, as in this example:

```
m = dsp.FarrowRateConverter(48,44.1);
fdhdltool(m,numerictype(1,16,15));
```

`fdhdltool` opens the Generate HDL tool for the `dsp.FarrowRateConverter` filter, as shown in this figure.



These code generation options are not supported for `dsp.FarrowRateConverter` filters and are disabled in the UI:

- Use of pipeline registers (`AddPipelineRegisters`)
- Distributed Arithmetic architecture (`DARadix` and (`DALUTPartition`))
- Fully or partially serial architectures (`SerialPartition` and `ReuseAccum`)

- Multiple clock inputs (`ClockInputs`)

See Also
`fdhdltool` | `generatehdl`

More About
- "Single-Rate Farrow Filters" on page 3-18
- "Multirate Filters" on page 3-2

Single-Rate Farrow Filters

In this section...

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

Supported Single-Rate Farrow Filters

You can generate HDL code for single-rate Farrow filters from these objects and structures:

- `dsp.VariableFractionalDelay` System object
- `dfilt.farrowfd` filter structure
- `dfilt.farrowlinearfd` filter structure

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

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

Code Generation Mechanics for Farrow Filters

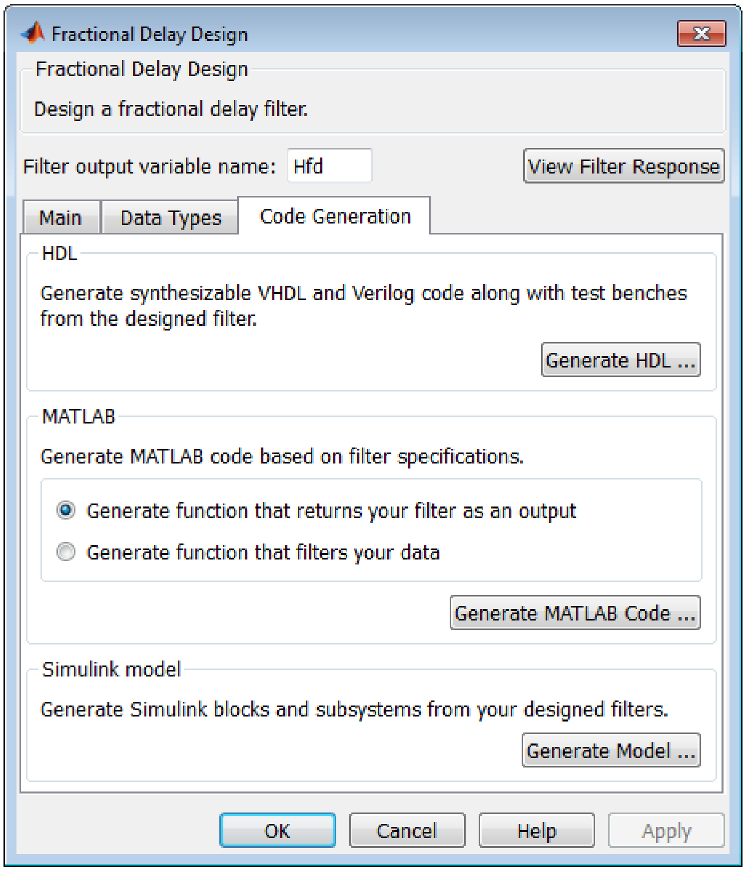
Filter Designer does not support the design or import of Farrow filters. To generate HDL code for a Farrow filter, use one of these methods:

- Use the MATLAB command line to create a Farrow filter object. Initiate code generation, and set Farrow-related properties. See "Code Generation Properties for Farrow Filters" on page 3-19.
- Use the MATLAB command line to create a Farrow filter object. Then pass this object in to `fdhdltool`.

  For example, these commands create a Farrow fractional delay System object, `farrowfilt`, and pass it to `fdhdltool`, together with its input and fractional delay data types.

  ```
  farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
  inputDataType = numerictype(1,18,17);
  fdDataType = numerictype(1,8,7);
  fdhdltool(farrowfilt,inputDataType,fdDataType);
  ```

  The `fdhdltool` tool opens the Generate HDL tool. See "UI Options for Farrow Filters" on page 3-20 for more information how to set properties in the tool.
- Use `filterBuilder` to design a Farrow (fractional delay) filter object. In the Filter Builder tool, select the **Code Generation** tab. To open the Generate HDL tool, click **Generate HDL**.



See “UI Options for Farrow Filters” on page 3-20 for more information how to set properties in the Generate HDL tool.

Options Disabled for Farrow Filters

When the Generate HDL tool opens with a Farrow filter, the coder disables some options or sets them to fixed default values. The options affected are:

- **Architecture**. The coder sets this option to its default (`Fully parallel`) and disables it.
- **Clock inputs**. The coder sets this option to its default (`Single`) and disables it.

Code Generation Properties for Farrow Filters

These properties are supported for Farrow filter code generation:

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

```
farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17),...
            'FractionalDelayDataType',numerictype(1,8,7),...
            'FracDelayPort','FractionalDelay');
```

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

By default, an internal constant value is applied to the fractional delay port. To use the default, leave the `TestBenchFracDelayStimulus` property unspecified, or pass in the empty character vector (`''`):

```
farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17),...
            'FractionalDelayDataType',numerictype(1,8,7),...
            'GenerateHDLTestbench','on');
```

Alternatively, you can specify generation of these types of stimulus vectors:

- `'RandSweep'`: A vector of random values between 0 and 1. This stimulus signal has the same duration as the input to the filter, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal. For example:

  ```
  farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
  generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17),...
              'FractionalDelayDataType',numerictype(1,8,7),...
              'GenerateHDLTestbench','on',...
              'TestbenchFracDelayStimulus','RandSweep');
  ```
- `'RampSweep'`: A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the input to the filter, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.
- A user-defined stimulus vector. You can pass in a vector to define your own stimulus. For example:

  ```
  farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
  mytestv = [0.5*ones(1, 512), 0.2*ones(1, 513)];
  generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17),...
             'FractionalDelayDataType',numerictype(1,8,7),...
             'GenerateHDLTestbench','on',...
             'TestBenchStimulus',{'chirp'},...
             'TestbenchFracDelayStimulus',mytestv);
  ```

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

  ```
  farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
  mytestv = [0.5*ones(1, 512), 0.2*ones(1, 513)];
  generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17),...
             'FractionalDelayDataType',numerictype(1,8,7),...
             'GenerateHDLTestbench','on',...
             'TestBenchStimulus',{'chirp' 'noise'},...
             'TestbenchFracDelayStimulus',mytestv);

  Error using hdlfilter.abstractfarrow/genVecDataforFarrow
  The lengths of specified vectors for FracDelay (1025) and Input (2050) do not match.
  ```

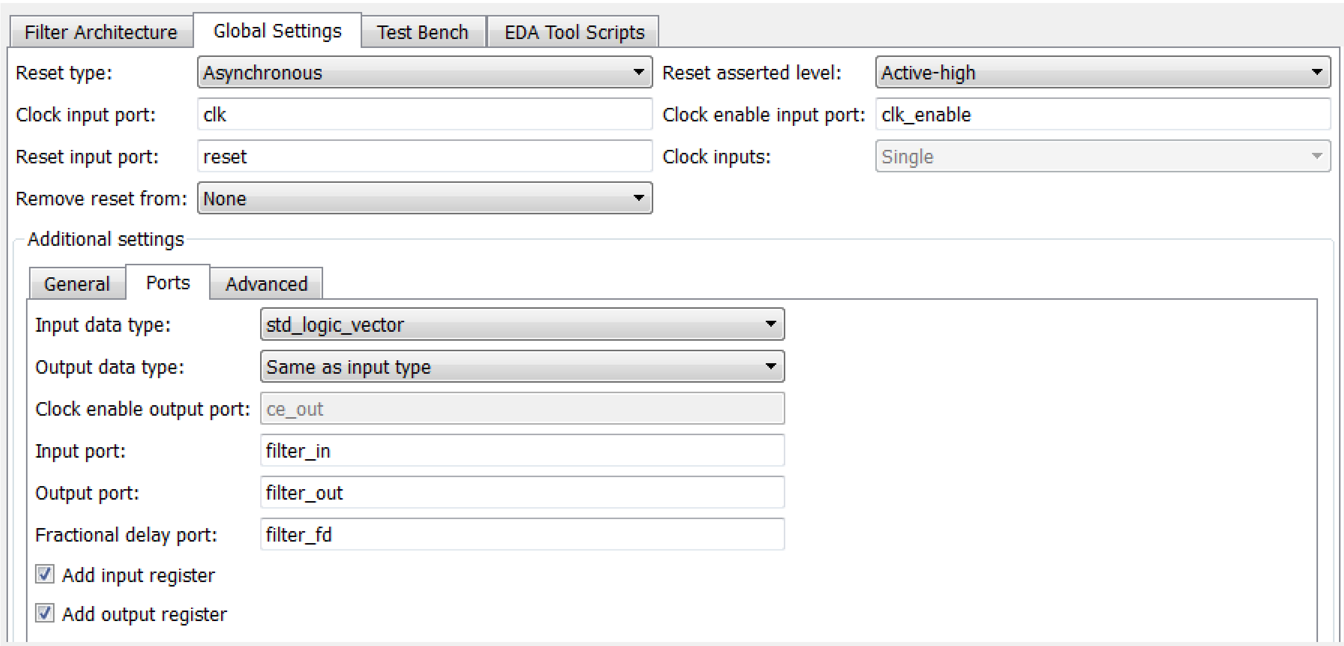
UI Options for Farrow Filters

The Filter Design HDL Coder UI provides options for generating Farrow filter code. These options correspond to the properties described in “Code Generation Properties for Farrow Filters” on page 3-19.

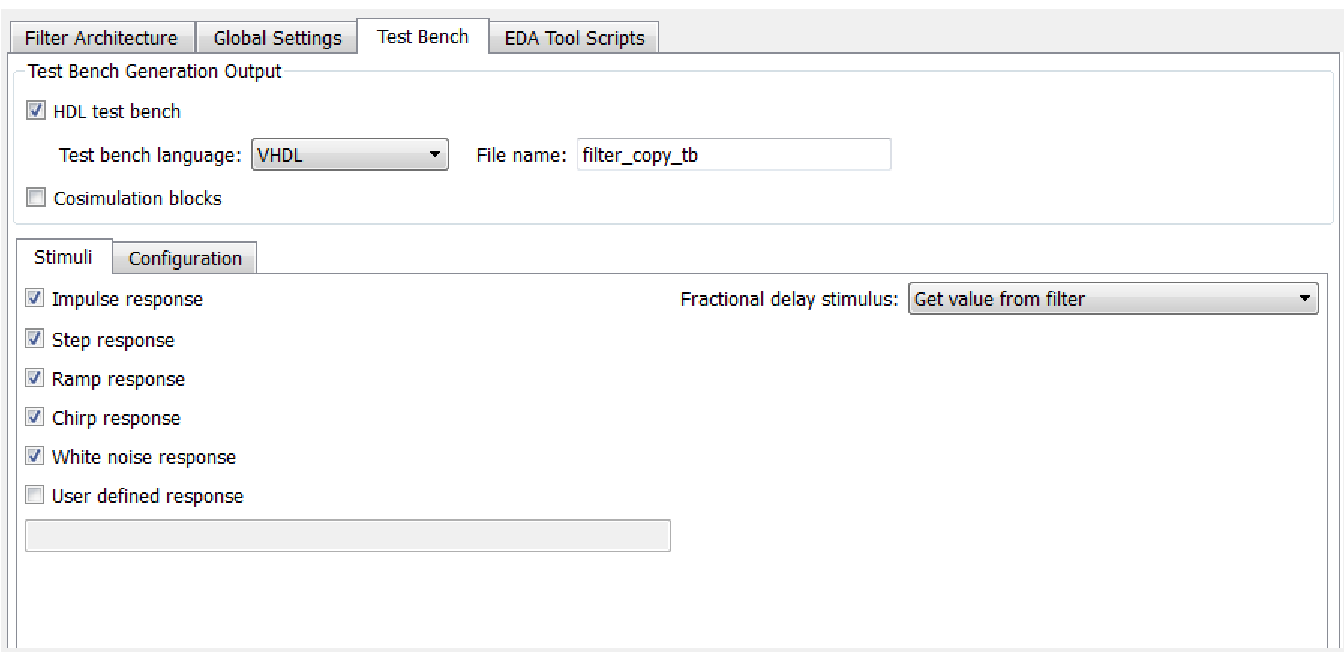
Note The Farrow filter options are displayed only when a Farrow filter is selected for HDL code generation.

These properties are supported for Farrow filter code generation:

- In the Generate HDL tool, on the **Global Settings > Ports** tab, **Fractional delay port** specifies the name of the fractional delay port in the generated code. The default name is `filter_fd`.



- In the Generate HDL tool, on the **Test Bench > Stimuli** tab, use the **Fractional delay stimulus** to select a stimulus signal. This signal is applied to the fractional delay port in the generated test bench.



Use the **Fractional delay stimulus** to select the type of stimulus signal in the generated code:

- **Get value from filter** (default). An internal constant value is applied to the fractional delay port.
- **Ramp sweep**. A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the input to the filter, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.
- **Random sweep**. A vector of random values between 0 and 1. This stimulus signal has the same duration as the input to the filter, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.
- **User defined**. When you select this option, the **User defined stimulus** box is enabled. You can enter a vector to define your own stimulus as shown in this figure:

See Also

Related Examples

- "HDL Fractional Delay (Farrow) Filter" on page 8-47

Programmable Filter Coefficients for FIR Filters

Note The Filter Design HDL Coder product will be discontinued in a future release. Instead, you can model hardware behavior, and generate HDL code by using System objects or Simulink blocks from DSP HDL Toolbox. These objects and blocks include hardware-friendly control signals and architecture options. To generate HDL code from DSP HDL Toolbox objects and blocks, you must also have the HDL Coder product.

For examples of modeling a programmable FIR filter for hardware, see “Programmable FIR Filter for FPGA” and “Optimize Programmable FIR Filter Resources” (DSP HDL Toolbox). These examples use the Discrete FIR Filter block. Equivalent functionality is also available in the `dsphdl.FIRFilter` System object.

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

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

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

Programmable filter coefficients are supported for these direct form FIR filter types:

- Direct form
- Direct form symmetric
- Direct form antisymmetric

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

Programmable filter coefficients are supported for these filter architectures:

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

When you choose a serial FIR filter architecture, you can also specify how the coefficients are stored. You can select a dual-port or single-port RAM, or a register file. See “Programmable Filter Coefficients for FIR Filters” on page 3-23.

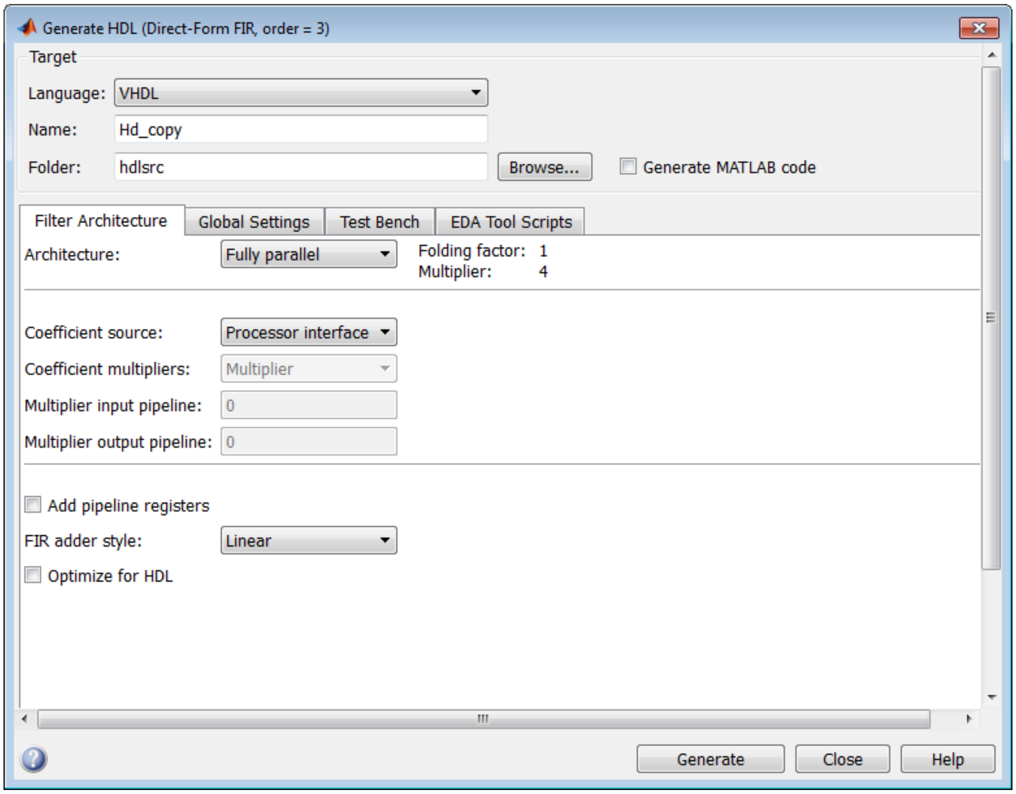
You can also generate a processor interface for loading IIR filter coefficients. See “Programmable Filter Coefficients for IIR Filters” on page 3-30.

UI Options for Programmable Coefficients

These UI options let you specify programmable coefficients.

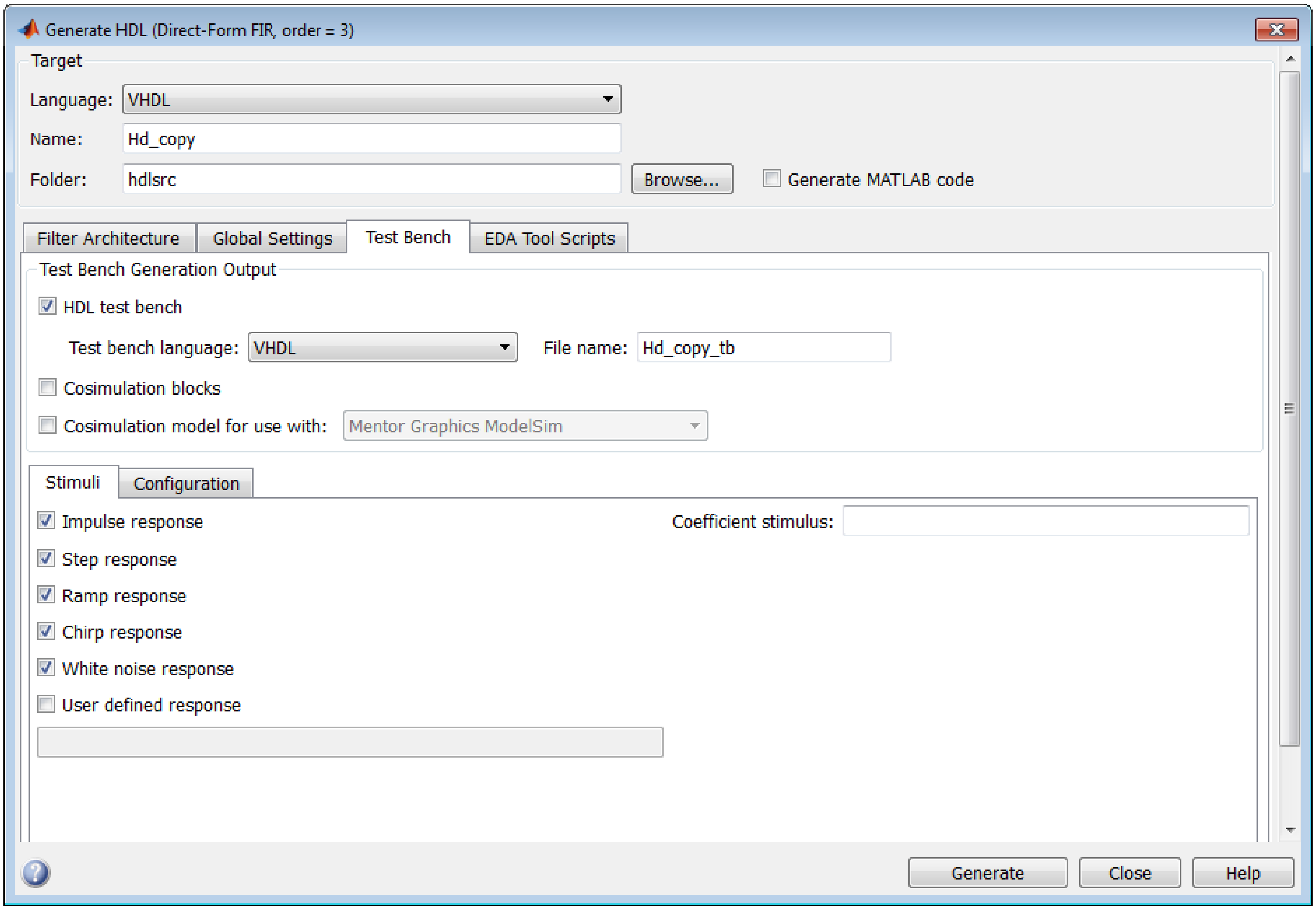
- The **Coefficient source** list on the Generate HDL tool lets you select whether coefficients are obtained from the filter object and hard-coded (`Internal`), or from memory (`Processor interface`). The default is `Internal`.

  The corresponding command-line property is `CoefficientSource`.



- The **Coefficient stimulus** option on the **Test Bench** pane of the Generate HDL tool specifies how the test bench tests the generated memory interface.

  The corresponding command-line property is `TestBenchCoeffStimulus`.



Generating a Test Bench for Programmable FIR Coefficients

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

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

The `TestBenchStimulus` property determines the filter input stimuli.

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

- `[]`: Empty vector. (default)

  When the value of `TestbenchCoeffStimulus` is an empty vector, the test bench loads the coefficients from the filter object, and then forces the input stimuli. This test verifies that the interface writes one set of coefficients into the memory without encountering an error.
- `[coeff1,coeff2,...coeffN]`: Vector of `N` coefficients, where `N` is determined as follows:

- For symmetric filters, `N` must equal `ceil(length(filterObj.Numerator)/2)`.
- For antisymmetric filters, `N` must equal `floor(length(filterObj.Numerator)/2)`.
- For other filters, `N` must equal the length of the filter object.

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

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

For an example, see “Test Bench for FIR Filter with Programmable Coefficients” on page 10-21.

Using Programmable Coefficients with Serial FIR Filter Architectures

This section discusses special considerations for using programmable filter coefficients with FIR filters that have one of these serial architectures.

- `Fully serial`
- `Partly serial`
- `Cascade serial`

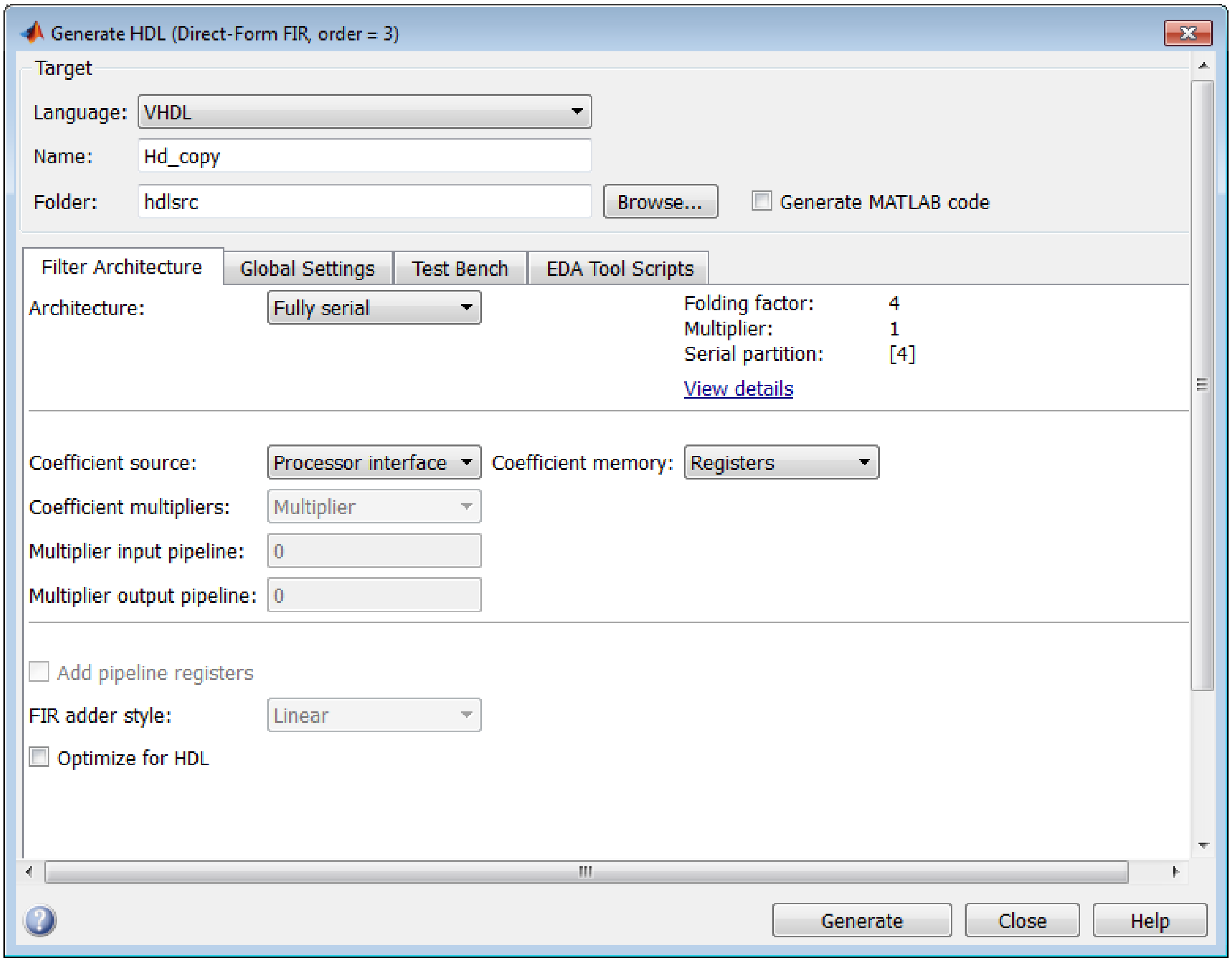
Specifying Memory for Programmable Coefficients

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

You can set **Coefficient memory** when:

- The filter is a FIR filter.
- You set **Coefficient source** to `Processor interface`.
- You set **Architecture** to `Fully serial`, `Partly serial`, or `Cascade serial`.

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



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

Coefficient memory Selection	Description
`Registers`	*default*: Store programmable coefficients in a register file.
`Single Port RAMs`	Store programmable coefficients in single-port RAM. The coder writes each RAM and its interface to a separate file. The number of generated RAMs depends on the filter partitioning.
`Dual Port RAMs`	Store programmable coefficients in dual-port RAM. The coder writes each RAM and its interface to a separate file. The number of generated RAMs depends on the filter partitioning.

Timing Considerations

In a serial implementation of a FIR filter, the rate of the system clock (`clk`) is generally a multiple of the input data rate (the sample rate of the filter). The exact relationship between the clock rate and the data rate depends on your choice of serial architecture and partitioning options.

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

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

These two examples illustrate how serial partitioning affects the timing of coefficient loading.

Create a direct form filter with 11 coefficients.

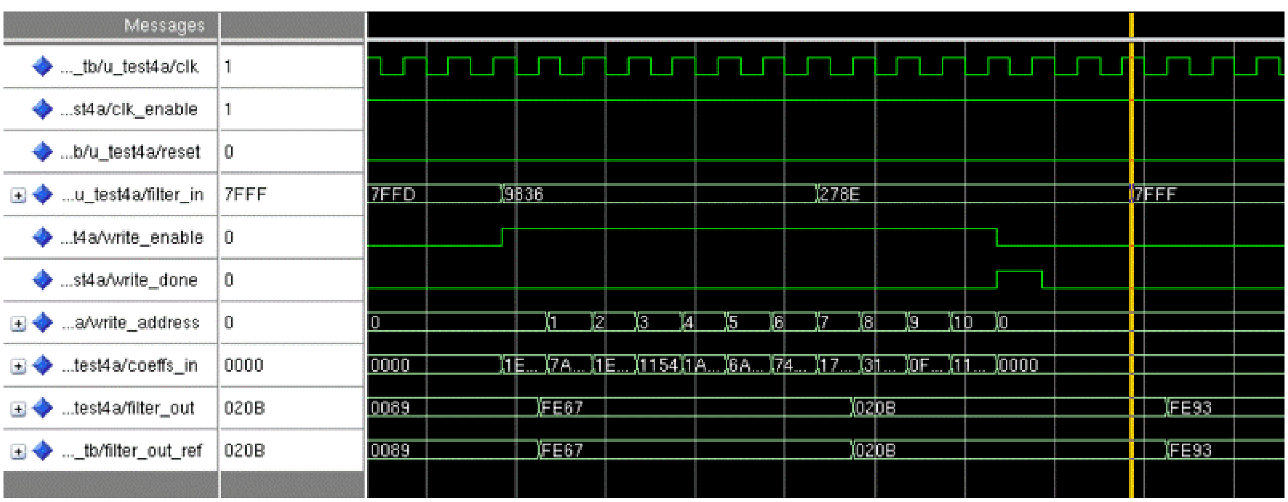
```
rng(13893,'v5uniform');
b = rand(1,11);
filt = dsp.FIRFilter('Numerator',b,'Structure','Direct form');
```

Generate VHDL code for `filt`, using a partly serial architecture with the serial partition `[7 4]`. Set `CoefficientSource` to generate a processor interface.

```
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'SerialPartition',[7 4],'CoefficientSource','ProcessorInterface');

### Clock rate is 7 times the input sample rate for this architecture.
### HDL latency is 3 samples
```

This partitioning results in a clock rate that is seven times the input sample rate.

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

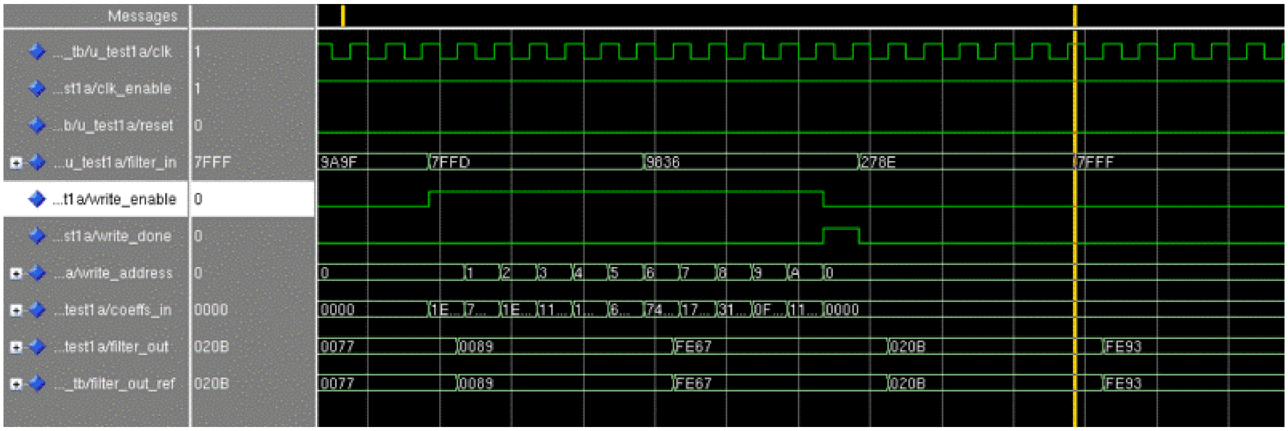


Now define a serial partition of `[6 5]` for the same filter. This partition results in a slower clock rate, six times the input sample rate.

```
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'SerialPartition',[6 5],'CoefficientSource','ProcessorInterface');
```

```
### Clock rate is 6 times the input sample rate for this architecture.
### HDL latency is 3 samples
```

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



Programmable Filter Coefficients for IIR Filters

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

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

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

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

These IIR filter types support programmable filter coefficients.

- 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

Limitations

- Programmable filter coefficients are supported for IIR filters with fully parallel architectures only.
- The generated interface assumes that the coefficients are stored in a register file.
- When you generate a processor interface for an IIR filter, the `OptimizeScaleValues` property must be between `1` and `0`. For example:

  ```
  filt.OptimizeScaleValues = 0
  ```

  Check that the filter still has the desired response, using the `fvtool` and `filter`, commands. Disabling `filt.OptimizeScaleValues` may add quantization at section inputs and outputs.

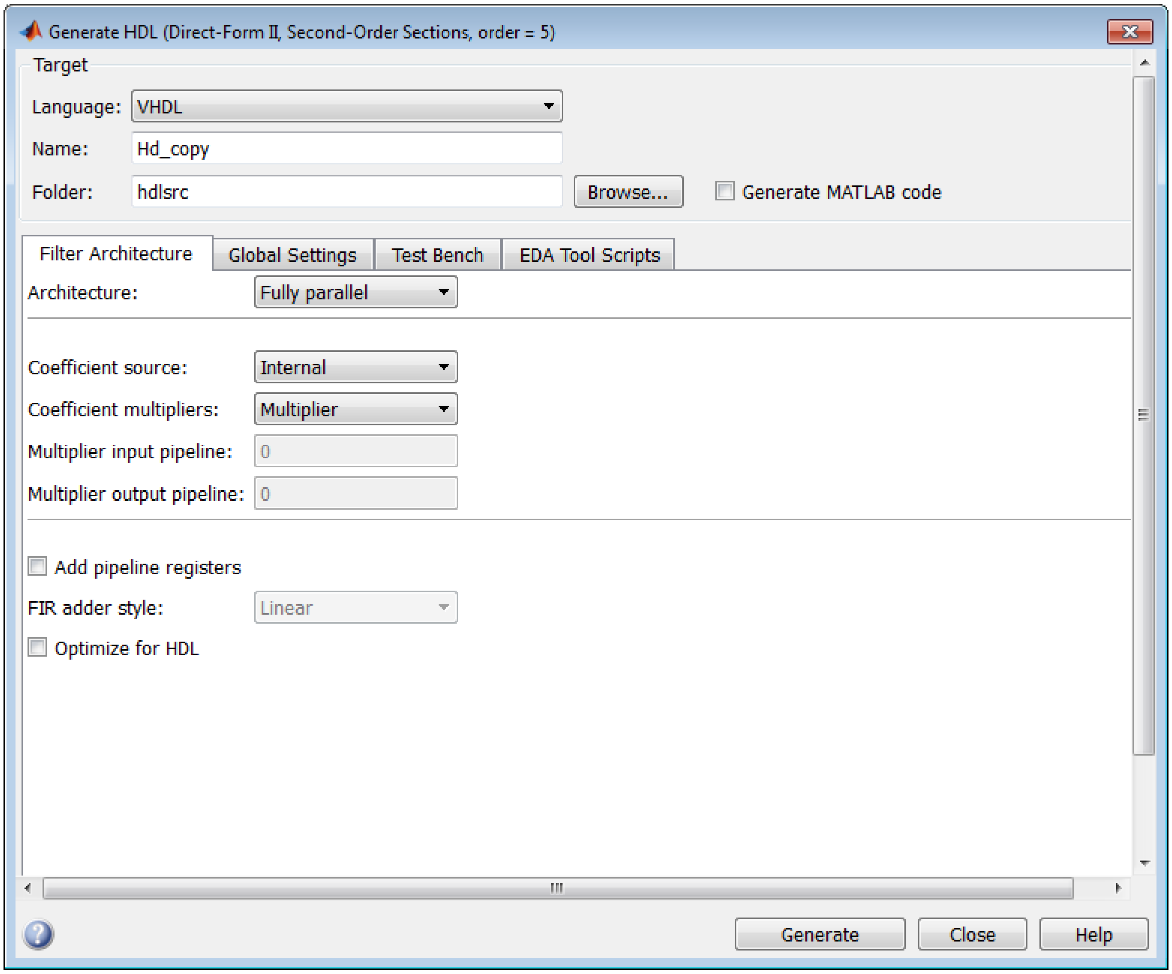
You can also generate a processor interface for loading FIR filter coefficients."Specifying Memory for Programmable Coefficients" on page 3-26 for further information.

Generate a Processor Interface for a Programmable IIR Filter

You can specify a processor interface using the **Coefficient source** menu on the Generate HDL tool.

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

  The corresponding command-line property is `CoefficientSource`.



- The **Coefficient stimulus** option on the **Test Bench** pane of the Generate HDL tool specifies how the test bench tests the generated memory interface.

  The corresponding command-line property is `TestBenchCoeffStimulus`.

Generating a Test Bench for Programmable IIR Coefficients

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

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

The `TestBenchStimulus` property determines the filter input stimuli.

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

- `[]`: Empty vector. (default)

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

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

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

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

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

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

Addressing Scheme for Loading IIR Coefficients

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

Each type of coefficient has the same word length, but can have different fractional lengths.

The address for each coefficient is divided into two fields:

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

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

Section Address	Coefficient Address	Description
S S ... S	000	Section scale value
S S ... S	001	Numerator coefficient: b1
S S ... S	010	Numerator coefficient: b2

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

DUC and DDC System Objects

You can generate HDL code for Digital Up Converter (DUC) and Digital Down Converter (DDC) System objects. This capability is limited to code generation at the command line only.

When calling `generatehdl` for a System object, you must specify the data type of the input signal. Set the `InputDataType` property to a `numerictype` object.

```
hDDC = dsp.DigitalDownConverter('Oscillator','NCO')
generatehdl(hDDC,'InputDataType',numerictype(1,8,7))
```

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

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

Limitations

You cannot set the input and output port names. These ports have the default names of `ddc_in` and `ddc_out`. The coder inserts registers on input and output signals. If you attempt to turn them off, the coder returns a warning.

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

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

Optimization of HDL Filter Code

Speed vs. Area Tradeoffs

In this section...

Overview of Speed or Area Optimizations

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

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

Architecture	Available for Filter Types...
`Fully parallel` (default)	Filter types that are supported for HDL code generation
`Fully serial`	• direct form • direct form symmetric • direct form asymmetric • direct form I SOS • direct form II SOS
`Partly serial`	• direct form • direct form symmetric • direct form asymmetric • direct form I SOS • direct form II SOS
`Cascade serial`	• direct form • direct form symmetric • direct form asymmetric

The coder supports the full range of parallel and serial architecture options via properties passed in to the `generatehdl` function, as described in “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-5.

Alternatively, you can use the **Architecture** pop-up menu on the Generate HDL tool to choose parallel and serial architecture options, as described in “Select Architectures in the Generate HDL Tool” on page 4-7.

Note The coder also supports distributed arithmetic (DA), another highly efficient architecture for realizing filters. See “Distributed Arithmetic for FIR Filters” on page 4-16.

Parallel and Serial Architectures

Fully Parallel Architecture

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

Serial Architectures

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

You can select from these serial architecture options:

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

  In a fully serial architecture, the system clock runs at a much higher rate than the sample rate of the filter. Thus, for a given filter design, the maximum speed achievable by a fully serial architecture is less than the maximum speed 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 serial partitions. The taps within each partition execute serially, but the partitions execute together in parallel. The outputs of the partitions are summed at the final output.

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

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

  The Generate HDL tool lets you specify a partly serial architecture in terms of these three parameters. You can then view how a change in one parameter interacts with the other two. The coder also provides `hdlfilterserialinfo`, an informational function that helps you define an optimal serial partition for a filter.
- *Cascade-serial*: A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are grouped into several serial partitions that execute together in parallel. However, the accumulated output of each partition cascades to the accumulator of the previous partition. The output of the partitions is therefore computed at the accumulator of the first partition. This technique is termed *accumulator reuse*. You do not require a final adder, which saves area.

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

To generate a cascade-serial architecture, you specify a partly serial architecture with accumulator reuse enabled. If you do not specify the serial partitions, the coder automatically selects an optimal partitioning.

Latency in Serial Architectures

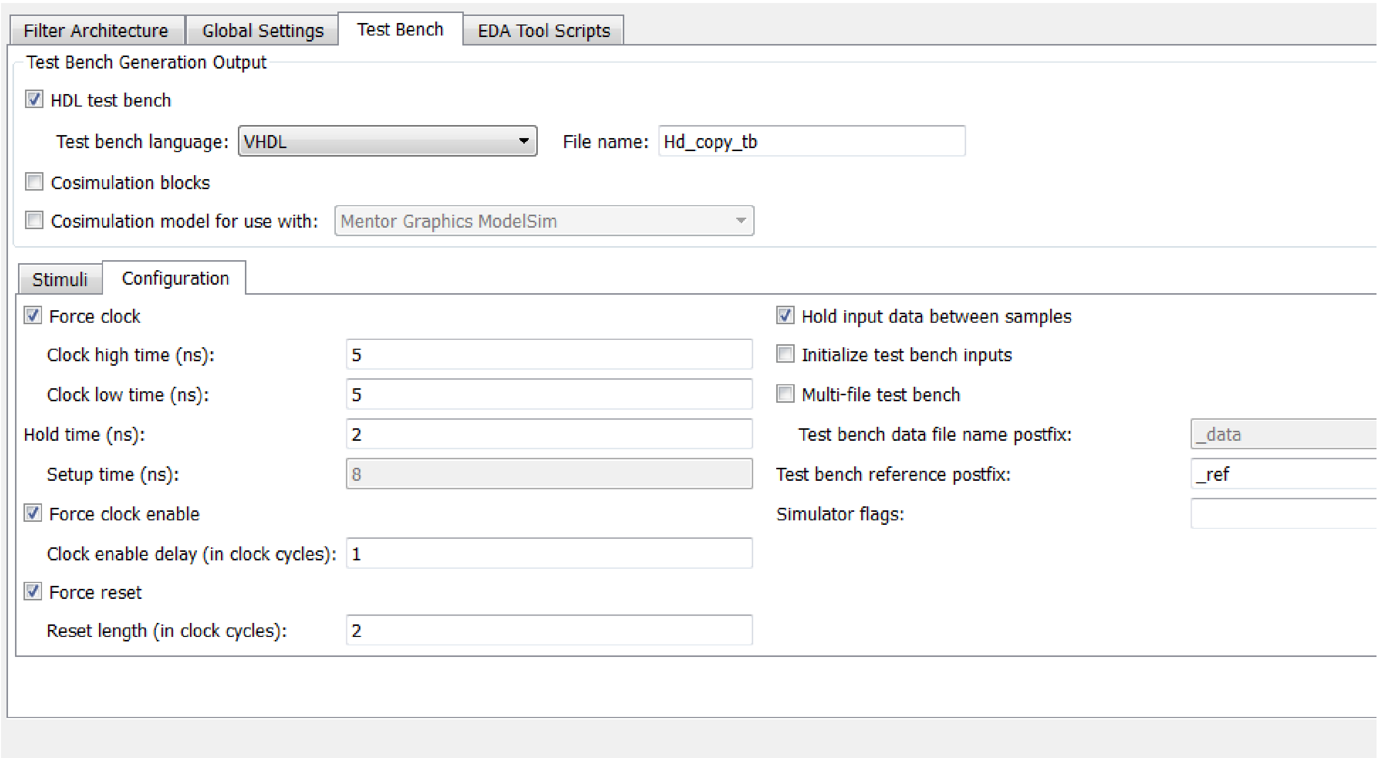
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the sequential products. An additional final register is used to store the summed result of each of the serial partitions. The operation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

Holding Input Data in a Valid State

Serial architectures implement internal clock rates higher than the input rate. In such filter implementations, there are `N` cycles (`N >= 2`) of the base clock for each input sample. You can specify how many clock cycles the test bench holds the input data values in a valid state.

- When you select **Hold input data between samples** (the default), the test bench holds the input data values in a valid state for `N` clock cycles.
- When you clear **Hold input data between samples**, the test bench holds input data values in a valid state for only one clock cycle. For the next `N-1` cycles, the test bench drives the data to an unknown state (expressed as `'X'`) until the next input sample is clocked in. Forcing the input data to an unknown state verifies that the generated filter code registers the input data only on the first cycle.

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



Use the equivalent `HoldInputDataBetweenSamples` property when you call the `generatehdl` function.

Specifying Speed vs. Area Tradeoffs via generatehdl Properties

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

Two properties specify serial architecture options to the `generatehdl` function:

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

The table summarizes how to set these properties to generate the desired architecture.

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'`

To Generate This Architecture...	Set SerialPartition to...	Set ReuseAccum to...
Partly serial	[p1 p2 p3...pN]: a vector of Ninteger elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. When you define the partitioning for a partly serial architecture, consider these points. • The filter length should be divided as uniformly as you can into a vector of length equal to the number of multipliers intended. For example, if your design requires a filter of length 9 with 2 multipliers, the recommended partition is [5 4]. If your design requires 3 multipliers, the recommended partition is[3 3 3] rather than some less uniform division such as [1 4 4] or [3 4 2]. • If your design is constrained by having to compute each output value (corresponding to each input value) in an exact number N of clock cycles, use N as the largest partition size and partition the other elements as uniformly as you can. For example, if the filter length is 9 and your design requires exactly 4 cycles to compute the output, define the partition as [4 3 2]. This partition executes in 4 clock cycles, at the cost of 3 multipliers. You can also specify a serial architecture in terms of a desired hardware folding factor, or in terms of the optimal number of multipliers. See hdlfilterserialinfo for detailed information.	'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 equal the length of the filter. The values of the vector elements must appear in descending order, except that the last two elements must be equal. For example, for a filter of length 9, partitions such as[5 4] or [4 3 2] would be legal, but the partitions [3 3 3] or [3 2 4] raise an error at code generation time.	'on'
Cascade-serial with automatically optimized partitioning	Omit this property	'on'

You can use the helper function hdlfilterserialinfo to explore possible partitions for your filter.

For an example, see "Generate Serial Partitions for FIR Filter" on page 10-9.

Serial Architectures for IIR SOS Filters

To specify a partly or fully serial architecture for an IIR SOS filter structure (`df1sos` or `dsp.BiquadFilter`), specify either one of these parameters:

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

To obtain information about the folding factor options and the corresponding number of multipliers for a filter, call the `hdlfilterserialinfo` function.

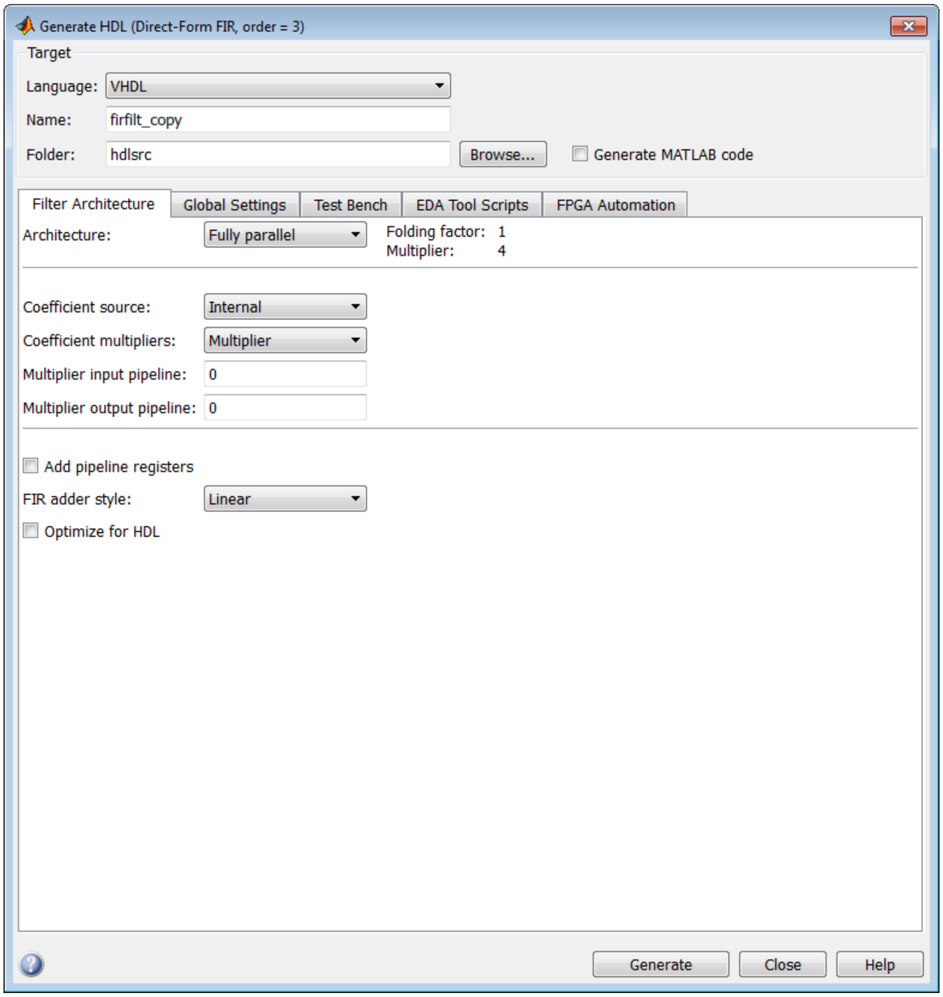
For an example, see “Generate Serial Architectures for IIR Filter” on page 10-13.

Select Architectures in the Generate HDL Tool

The **Architecture** pop-up menu, in the Generate HDL tool, lets you select parallel and serial architecture. These topics describe the UI options you must set for each **Architecture** choice.

Specifying a Fully Parallel Architecture

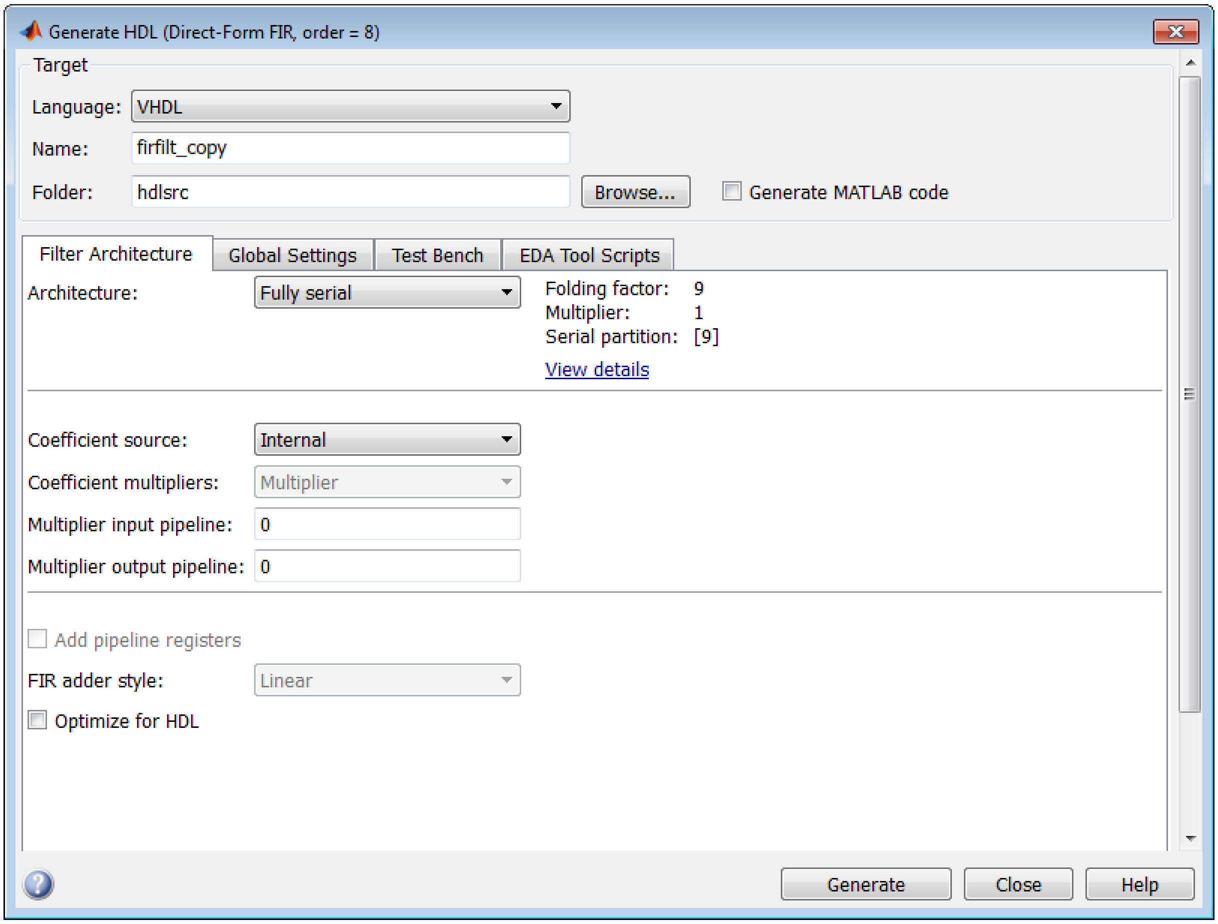
The default **Architecture** setting is `Fully parallel`, as shown.



Specifying a Fully Serial Architecture

When you select the `Fully serial`, **Architecture** options, the Generate HDL tool displays additional information about the folding factor, number of multipliers, and serial partitioning. Because these parameters depend on the length of the filter, they display in a read-only format, as shown in this figure.

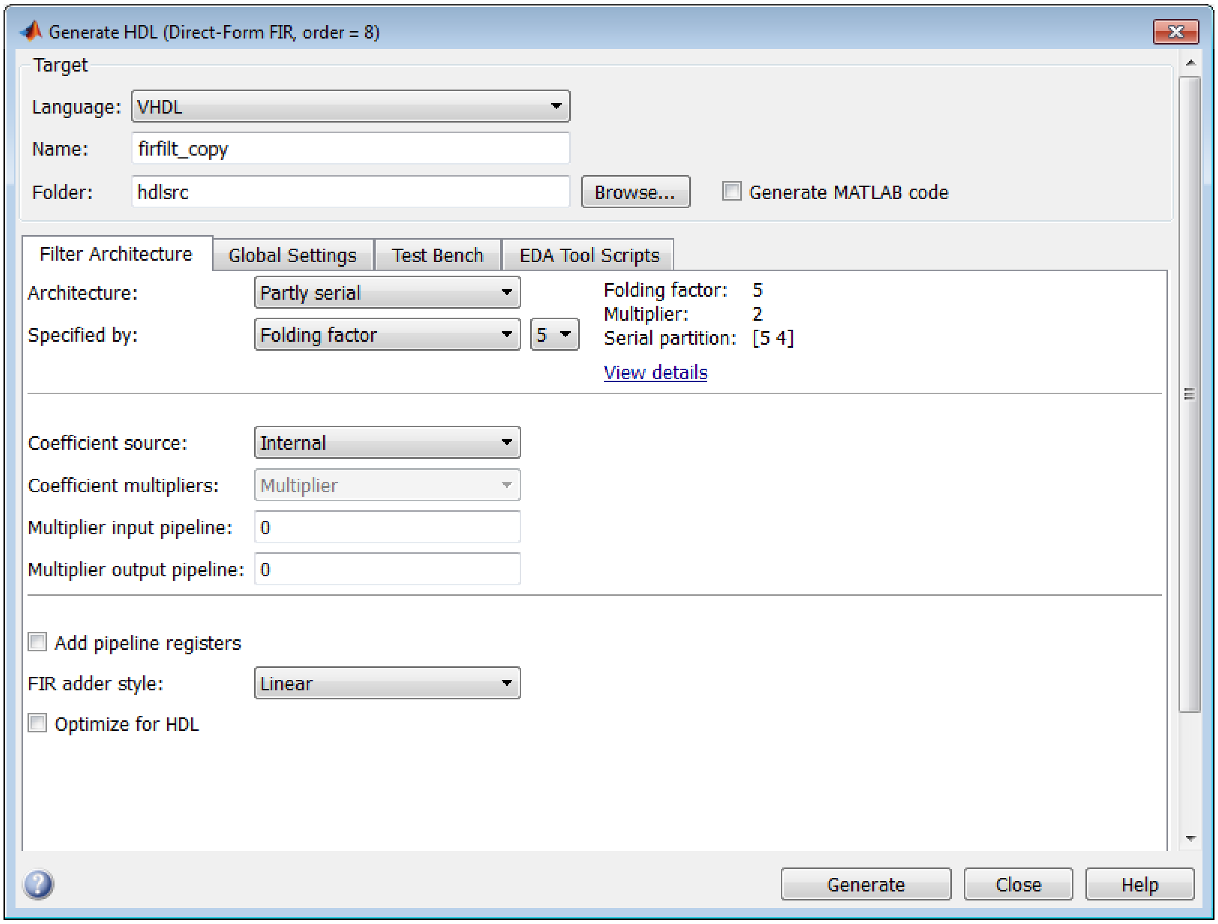
The Generate HDL tool also displays a **View details** link. When you click this link, the coder displays an HTML report in a separate window. The report displays an exhaustive table of folding factor, multiplier, and serial partition settings for the current filter. You can use the table to help you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



Specify Partitions for a Partly Serial Architecture

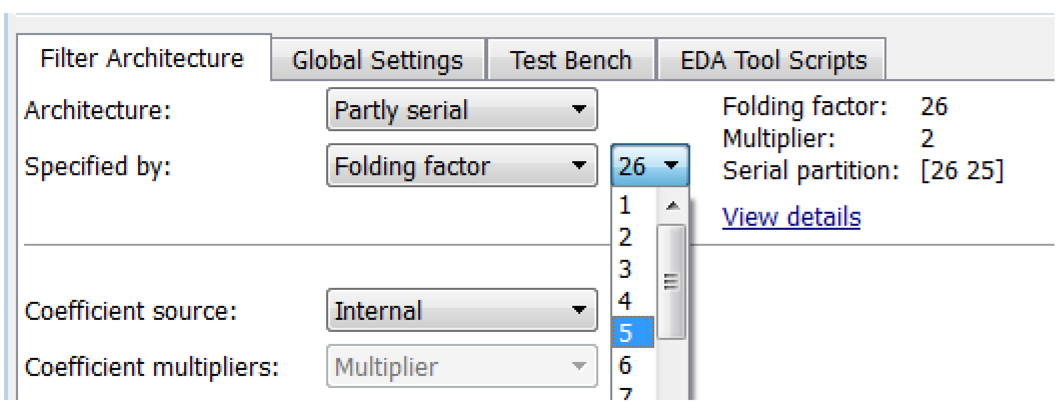
When you select the `Partly serial` **Architecture** option, the Generate HDL tool displays additional information and data entry fields related to serial partitioning.

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

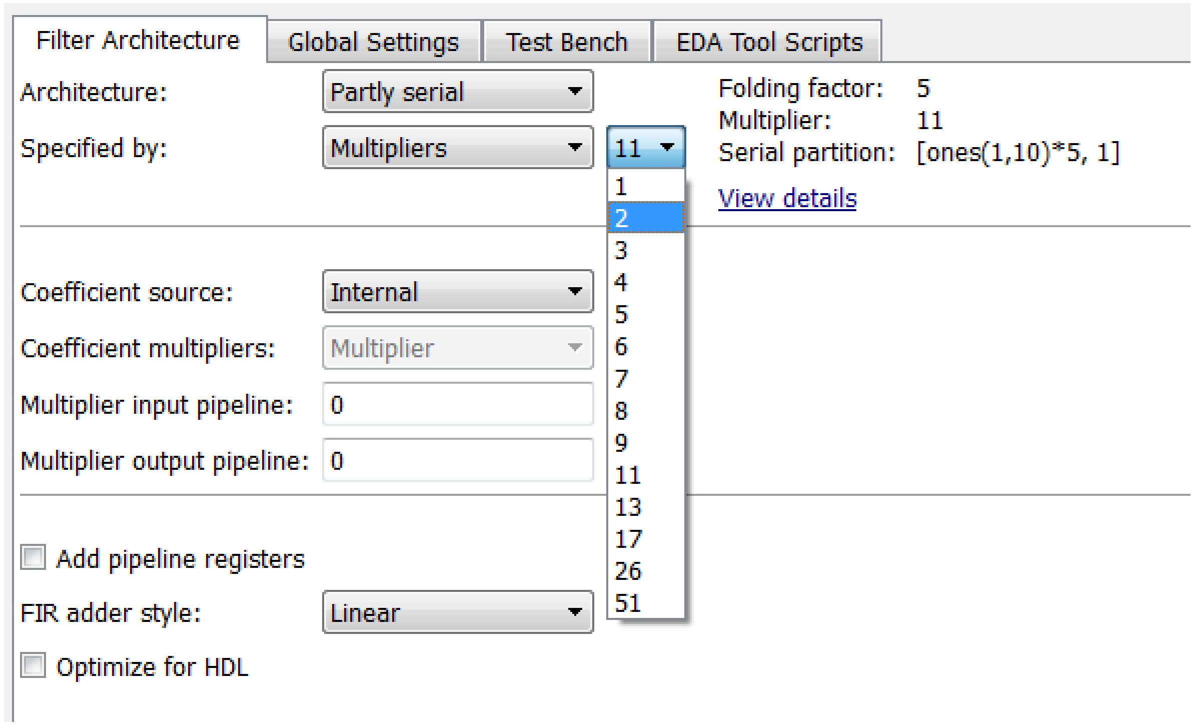


The **Specified by** drop-down menu lets you decide how you define the partly serial architecture. Select one of these options:

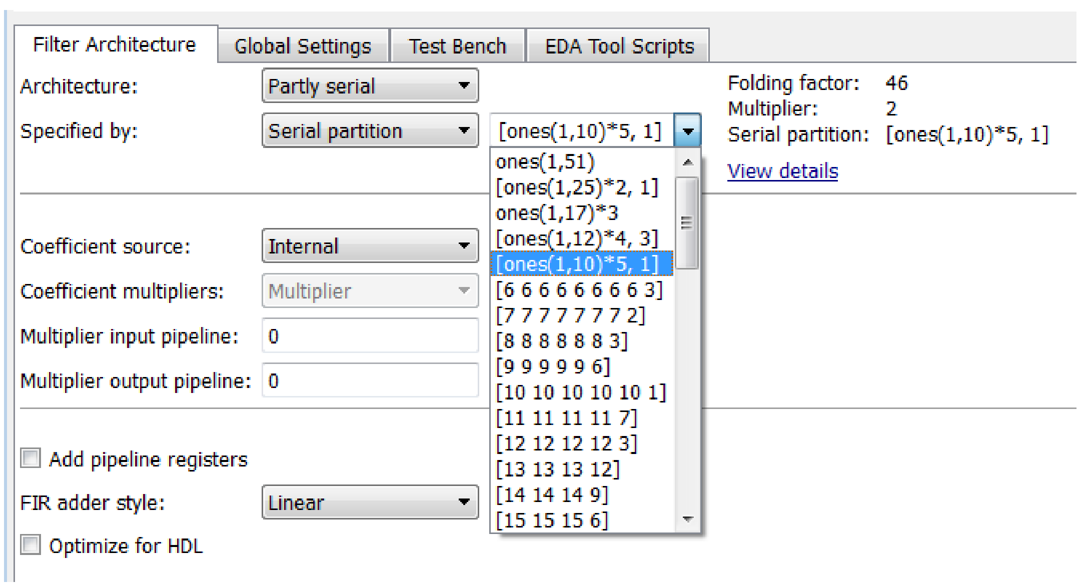
- `Folding factor`: The drop-down menu to the right of `Folding factor` contains an exhaustive list of folding factors for the filter. When you select a value, the display of the current folding factor, multiplier, and serial partition settings updates.



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

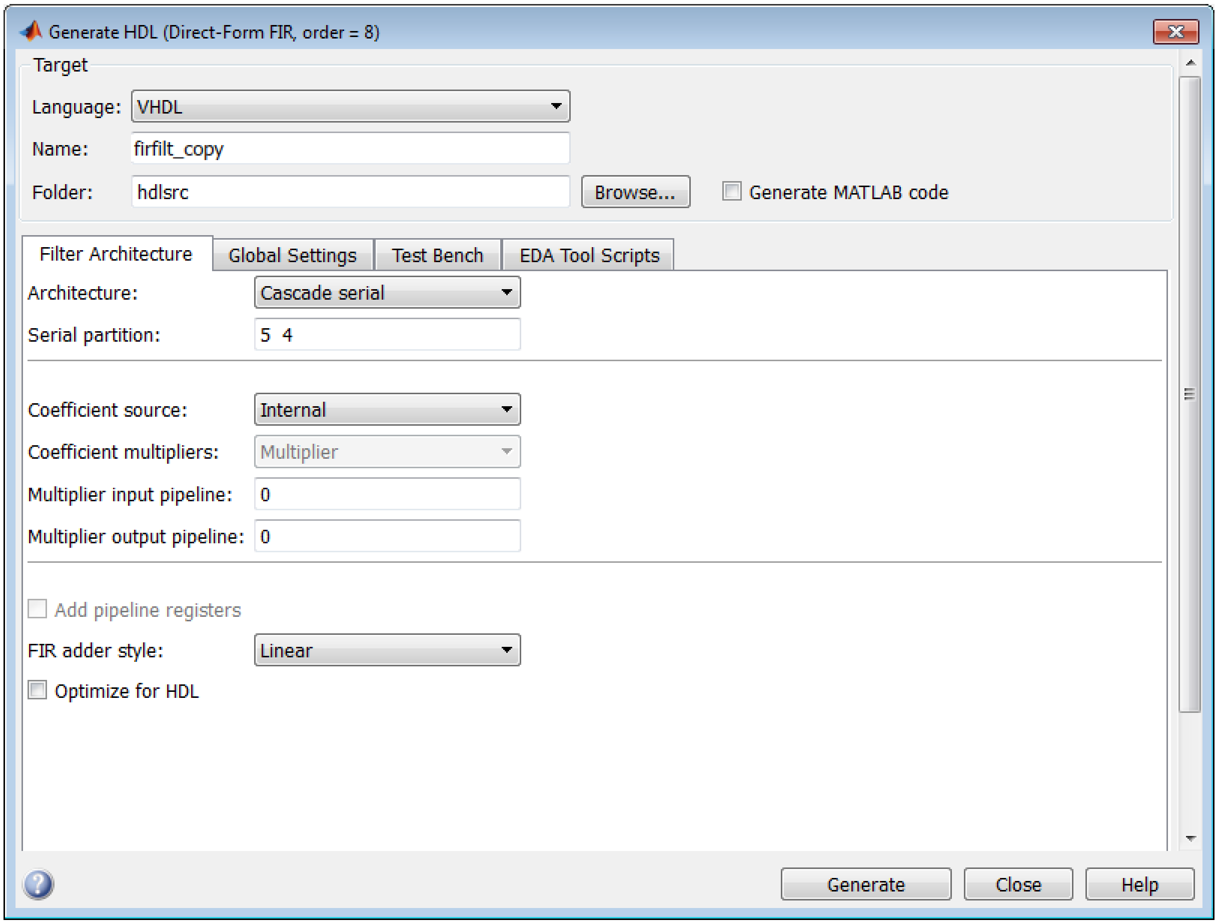


- `Serial partition`: The drop-down menu to the right of `Serial partition` contains an exhaustive list of serial partition options for the filter. When you select a value, the display of the current folding factor, multiplier, and serial partition settings updates.



Specifying a Cascade Serial Architecture

When you select the `Cascade serial` **Architecture** option, the Generate HDL tool displays the **Serial partition** field, as shown in this figure.



The **Specified by** menu lets you define the number and size of the serial partitions according to different criteria, as described in "Specifying Speed vs. Area Tradeoffs via generatehdl Properties" on page 4-5.

Specifying Serial Architectures for IIR SOS Filters

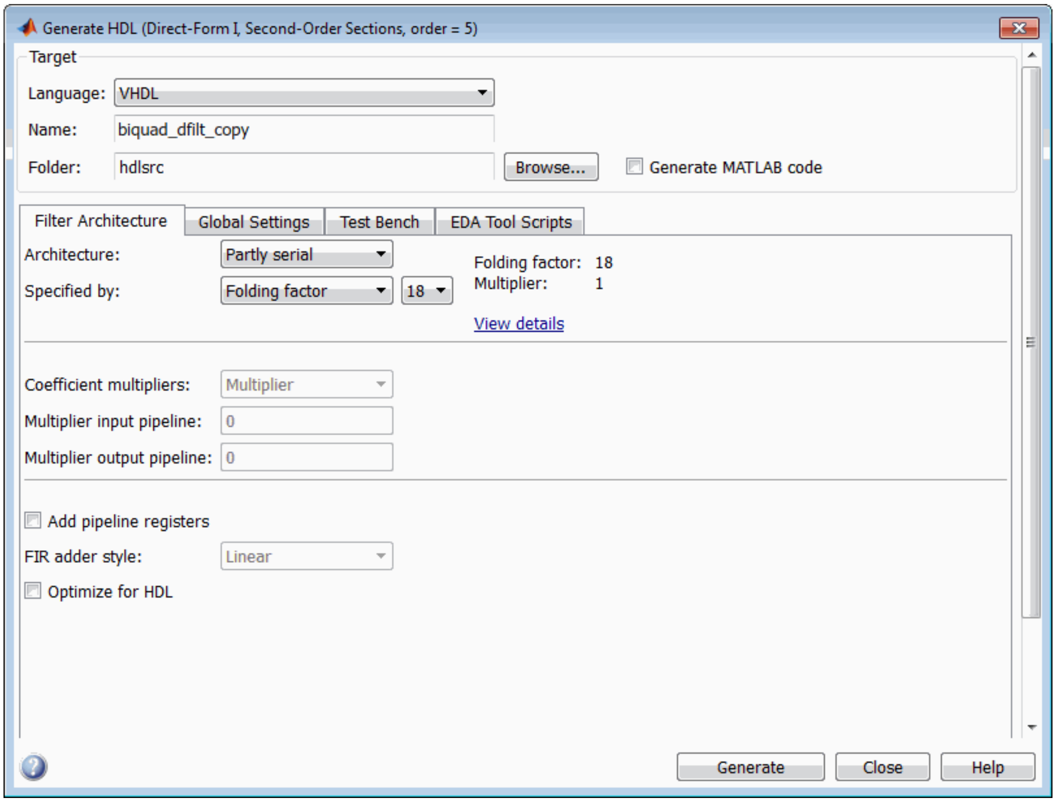
To specify a partly or fully serial architecture for an IIR SOS filter structure in the UI, set these options:

- **Architecture** — Select `Fully parallel` (the default), `Fully serial`, or `Partly serial`. If you select `Partly serial`, the UI displays the **Specified by** drop-down menu.
- **Specified by** — Select one of these methods:
  - `Folding factor` — Specify the desired hardware folding factor, *`ff`*, an integer greater than 1. Given the folding factor, the coder computes the number of multipliers.
  - `Multipliers` — Specify the desired number of multipliers, *`nmults`*, an integer greater than 1. Given the number of multipliers, the coder computes the folding factor.

Example: Direct Form I SOS Filter

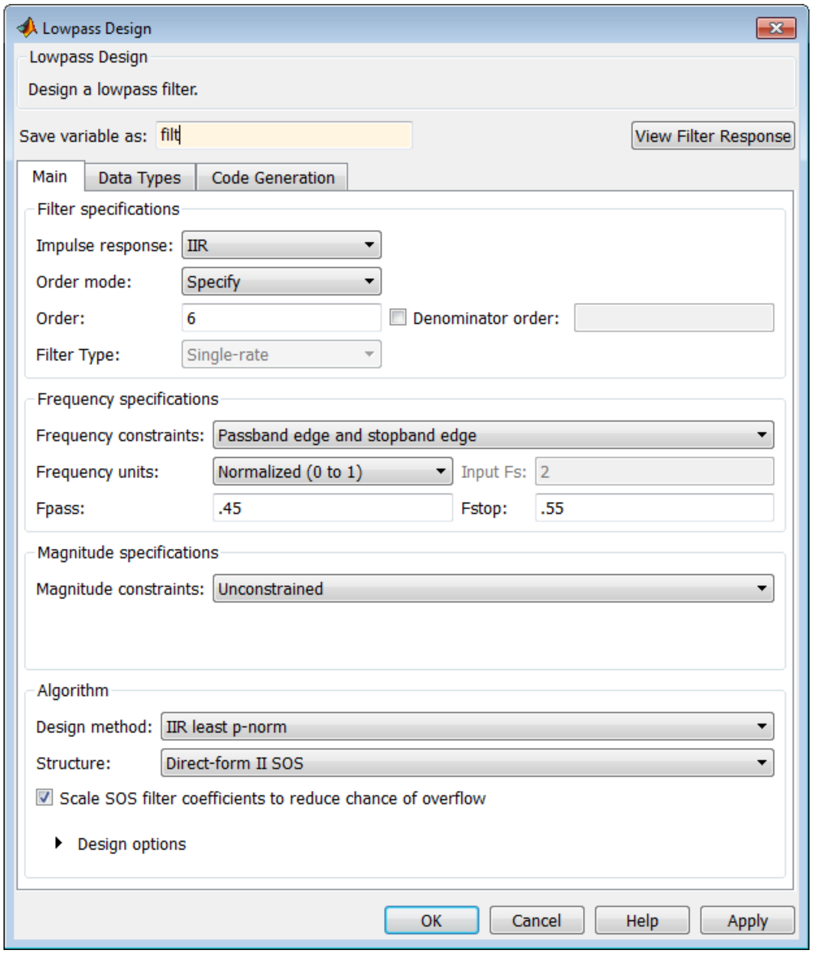
This example creates a Direct Form I SOS (`df1sos`) filter design and opens the UI. The figure following the code example shows the coder options configured for a partly serial architecture specified with a `Folding factor` of 18.

```
Fs = 48e3             % Sampling frequency
Fc = 10.8e3           % Cut-off frequency
N = 5                 % Filter Order
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs)
filt = design(f_lp,'butter','FilterStructure','df1sos','SystemObject',true)
fdhdltool(filt,numerictype(1,16,15))
```



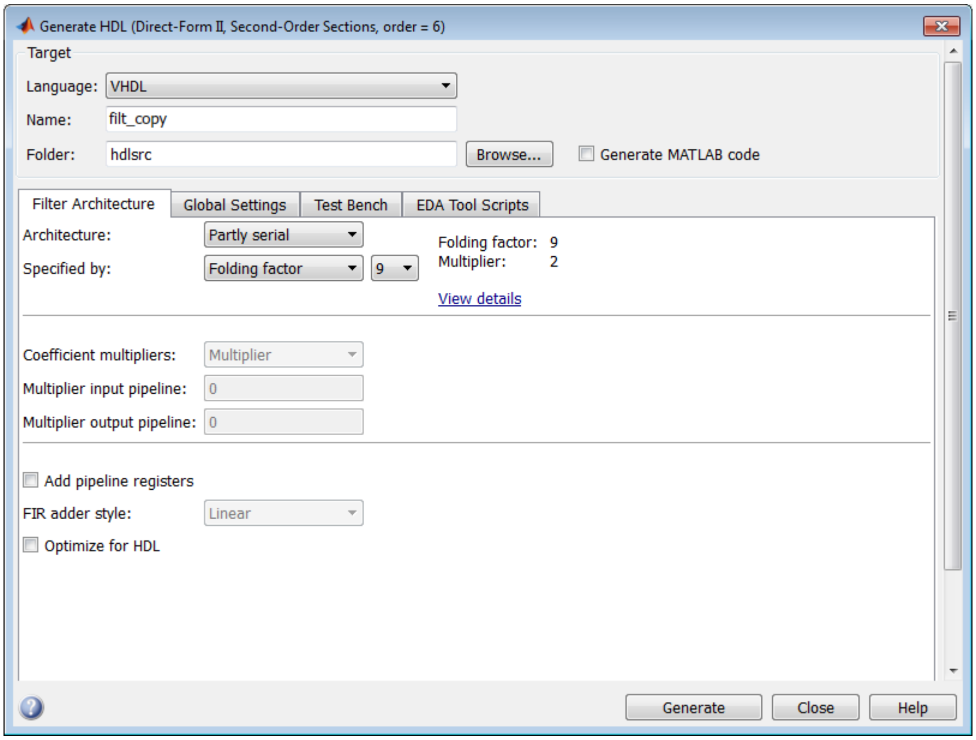
Example: Direct Form II SOS Filter

This example creates a Direct Form II SOS (`df2sos`) filter design using Filter Builder.



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

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



Specifying a Distributed Arithmetic Architecture

The **Architecture** pop-up menu also includes the `Distributed arithmetic (DA)` option. See “Distributed Arithmetic for FIR Filters” on page 4-16) for information about this architecture.

Interactions Between Architecture Options and Other HDL Options

Selecting certain **Architecture** menu options can change or disable other options.

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

Distributed Arithmetic for FIR Filters

In this section...

Distributed Arithmetic Overview

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

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

The coder supports DA in HDL code generated for several single-rate and multirate FIR filter structures for fixed-point filter designs. (See "Requirements and Considerations for Generating Distributed Arithmetic Code" on page 4-17.)

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

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

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

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

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

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

Improving Performance with Parallelism

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

To compute more than one bit-sum at a time, the coder replicates the LUT. For example, to perform DA on two 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 two places.

Processing more than one bit at a time introduces a degree of parallelism into the operation, which can improve performance at the expense of area. The `DARadix` property lets you specify the number of bits processed simultaneously in DA.

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 several 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. You can achieve a significant reduction in LUT size by dividing the LUT into 16 LUT partitions, each taking 10 inputs (taps). This division reduces the total LUT size to `16*(2^10)*W` bits.

Although LUT partitioning reduces LUT size, the architecture uses more adders to sum the LUT data.

The `DALUTPartition` property lets you specify how the LUT is partitioned in DA.

Requirements and Considerations for Generating Distributed Arithmetic Code

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

Supported Filter Types

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

- direct form (`dfilt.dffir` or `dsp.FIRFilter`)
- direct form symmetric (`dfilt.dfsymfir` or `dsp.FIRFilter`)
- direct form asymmetric (`dfilt.dfasymfir` or `dsp.FIRFilter`)
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`

Fixed-Point Quantization Required

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

Specifying Filter Precision

The data path in HDL code generated for the DA architecture is optimized for full precision computations. The filter casts the result to the output data size at the final stage. If your filter object is set to use full precision data types, numeric results from simulating the generated HDL code are bit-true to the output of the original filter object.

If your filter object has customized word or fraction lengths, the generated DA code may produce numeric results that are different than the output of the original filter object.

Coefficients with Zero Values

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

Considerations for Symmetric and Asymmetric Filters

For symmetric and asymmetric FIR filters:

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

Holding Input Data in a Valid State

Partitioned distributed arithmetic architectures implement internal clock rates higher than the input rate. In such filter implementations, there are `N` cycles (`N >= 2`) of the base clock for each input sample. You can specify how many clock cycles the test bench holds the input data values in a valid state.

- When you select **Hold input data between samples** (the default), the test bench holds the input data values in a valid state for `N` clock cycles.
- When you clear **Hold input data between samples**, the test bench holds input data values in a valid state for only one clock cycle. For the next `N-1` cycles, the test bench drives the data to an unknown state (expressed as `'X'`) until the next input sample is clocked in. Forcing the input data to an unknown state verifies that the generated filter code registers the input data only on the first cycle.

Distributed Arithmetic via generatehdl Properties

Two properties specify distributed arithmetic options to the `generatehdl` function:

- `DALUTPartition` — Number and size of lookup table (LUT) partitions.
- `DARadix` — Number of bits processed in parallel.

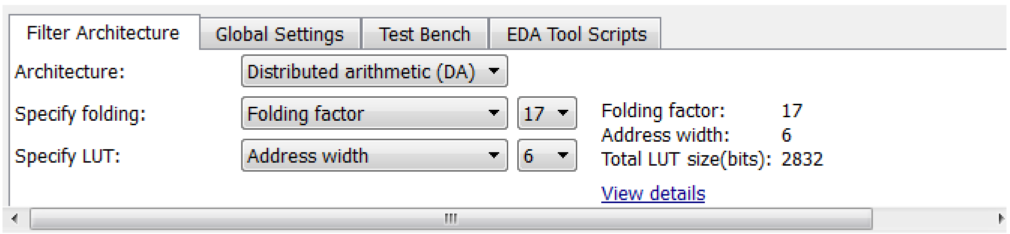
You can use the helper function `hdlfilterdainfo` to explore possible partitions and radix settings for your filter.

For examples, see

- “Distributed Arithmetic for Single Rate Filters” on page 10-15
- “Distributed Arithmetic for Multirate Filters” on page 10-15
- “Distributed Arithmetic for Cascaded Filters” on page 10-16

Distributed Arithmetic Options in the Generate HDL Tool

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



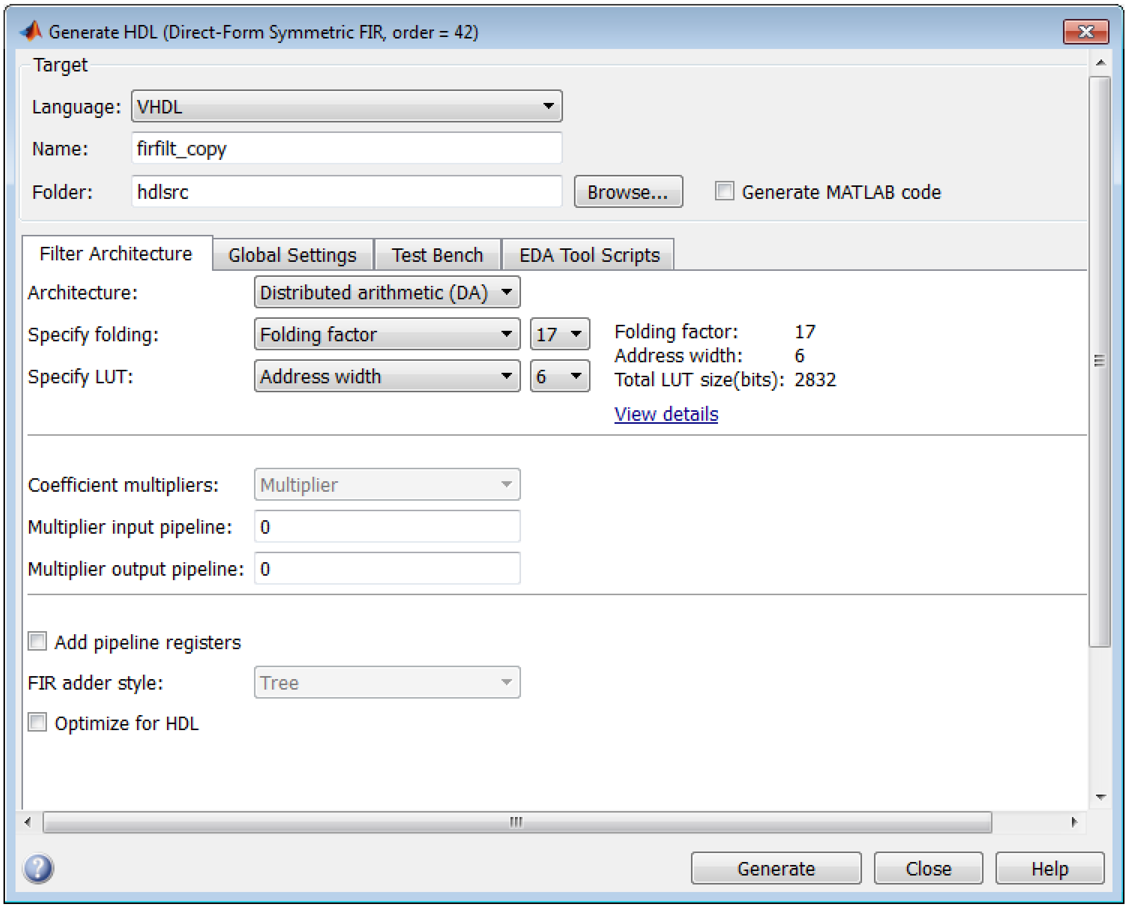
- The **Architecture** pop-up menu, which lets you enable DA code generation and displays related options.
- The **Specify folding** drop-down menu, which lets you directly specify the folding factor, or set a value for the `DARadix` property.
- The **Specify LUT** drop-down menu, which lets you directly set a value for the `DALUTPartition` property. You can also select an address width for the LUT. If you specify an address width, the coder uses input LUTs as required.

The Generate HDL tool initially displays default DA-related option values that correspond to the current filter design. For the requirements for setting these options, see `DALUTPartition` and `DARadix`.

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

1. Design a FIR filter (using Filter Designer, Filter Builder, or MATLAB commands) that meets the requirements described in “Requirements and Considerations for Generating Distributed Arithmetic Code” on page 4-17.
2. Open the Generate HDL tool.
3. Select `Distributed Arithmetic (DA)` from the **Architecture** pop-up menu.

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



4 Select one of these options from the **Specify folding** drop-down menu.

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

5 Select one of these options from the **Specify LUT** drop-down menu.

- `Address width` (default): Select from the drop-down menu to the right of **Specify LUT**. The menu contains an exhaustive list of LUT address widths for the filter.
- `Partition`: Select, or enter, a vector specifying the number and size of LUT partitions.

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

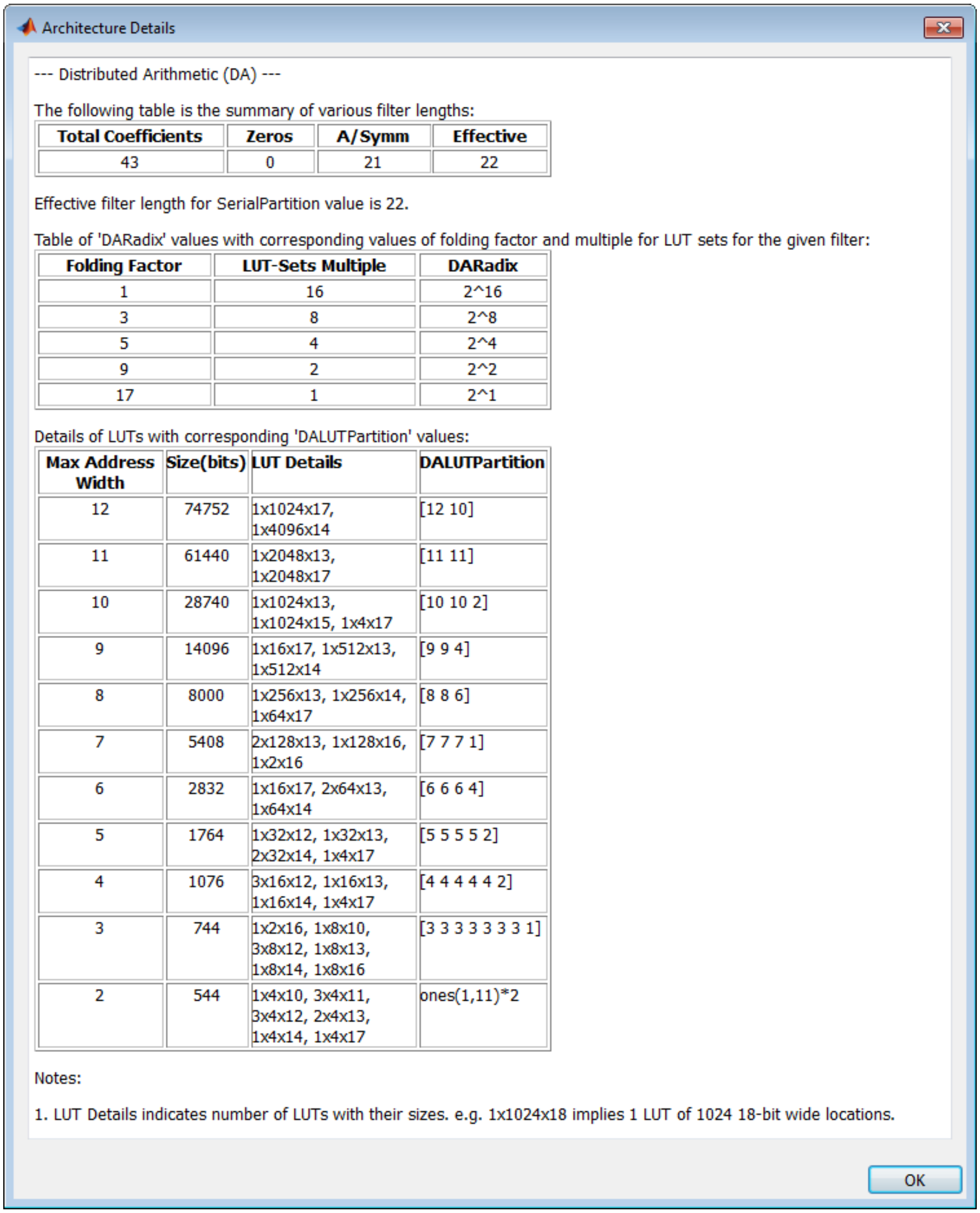
Viewing Detailed DA Options

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

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

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

This figure shows a typical report.


Architecture Details

--- Distributed Arithmetic (DA) ---

The following table is the summary of various filter lengths:

Total Coefficients	Zeros	A/Symm	Effective
43	0	21	22

Effective filter length for SerialPartition value is 22.

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

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

Details of LUTs with corresponding 'DALUTPartition' values:

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

Notes:

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

OK

DA Interactions with Other HDL Options

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

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

Architecture Options for Cascaded Filters

You can specify unique serial, distributed arithmetic, or parallel architectures for each stage of cascade filters. These options lead to area efficient implementations of cascade filters, including Digital Down Converter (DDC), and Digital Up Converter (DUC) objects. You can use this feature only with the command-line interface (`generatehdl`). When you use the Generate HDL tool, each stage of a cascade uses the same architecture options.

You can pass a cell array of values to the `SerialPartition`, `DALUTPartition`, and `DARadix` properties, with each element corresponding to its respective stage. To skip the corresponding specification for a stage, specify the default value of that property. When you set a partition to a size of `-1`, the coder implements a parallel architecture for that stage.

Property	Default Value
`SerialPartition`	–1
`DALUTPartition`	–1
`DARadix`	2

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

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

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

For examples, see

- “Distributed Arithmetic for Cascaded Filters” on page 10-16
- “Generate Serial Partitions of Cascaded Filter” on page 10-11
- “Cascaded Filter with Multiple Architectures” on page 10-19

CSD Optimizations for Coefficient Multipliers

By default, the coder produces code that includes coefficient multipliers. You can optimize these operations to decrease the area and maintain or increase clock speed. You can 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 optimization you can achieve depends on the binary representation of the coefficients used.

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

To optimize coefficient multipliers (for nonmultirate filter types):

1. Select `CSD` or `Factored-CSD` from the **Coefficient multipliers** menu in the **Filter architecture** pane of the Generate HDL tool.
2. To account for numeric differences, consider setting an error margin for the generated test bench. When comparing the results, the test bench ignores the number of least significant bits specified in the error margin. To set an error margin,
   a. Select the **Test Bench** pane in the Generate HDL tool. Then click the **Configuration** tab.
   b. Set the **Error margin (bits)** field to an integer that indicates the maximum acceptable number of bits of difference in the numeric results.
3. Continue setting other options or click **Generate** to initiate code generation.

If you are generating code for an FIR filter, see “Multiplier Input and Output Pipelining for FIR Filters” on page 4-26 for information on a related optimization.

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

Improving Filter Performance with Pipelining

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
FIR, antisymmetric FIR, and symmetric FIR filters	Between levels of the final summation tree
Transposed FIR filters	Between coefficient multipliers and adders
IIR filters	Between sections
CIC	Between comb sections

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

For FIR filters, the use of pipeline registers optimizes filter final summation. For details, see "Optimizing Final Summation for FIR Filters" on page 4-26.

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

To use pipeline registers,

1. Select the **Add pipeline registers** option in the **Filter architecture** pane of the Generate HDL tool.
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 ignores when comparing the results. To set an error margin:
   - a. Select the **Test Bench** pane in the Generate HDL tool. Then click the **Configuration** tab.
   - b. Set the **Error margin (bits)** field to an integer that indicates the maximum acceptable number of bits of difference in the numeric results.
3. Continue setting other options or click **Generate** to initiate code generation.

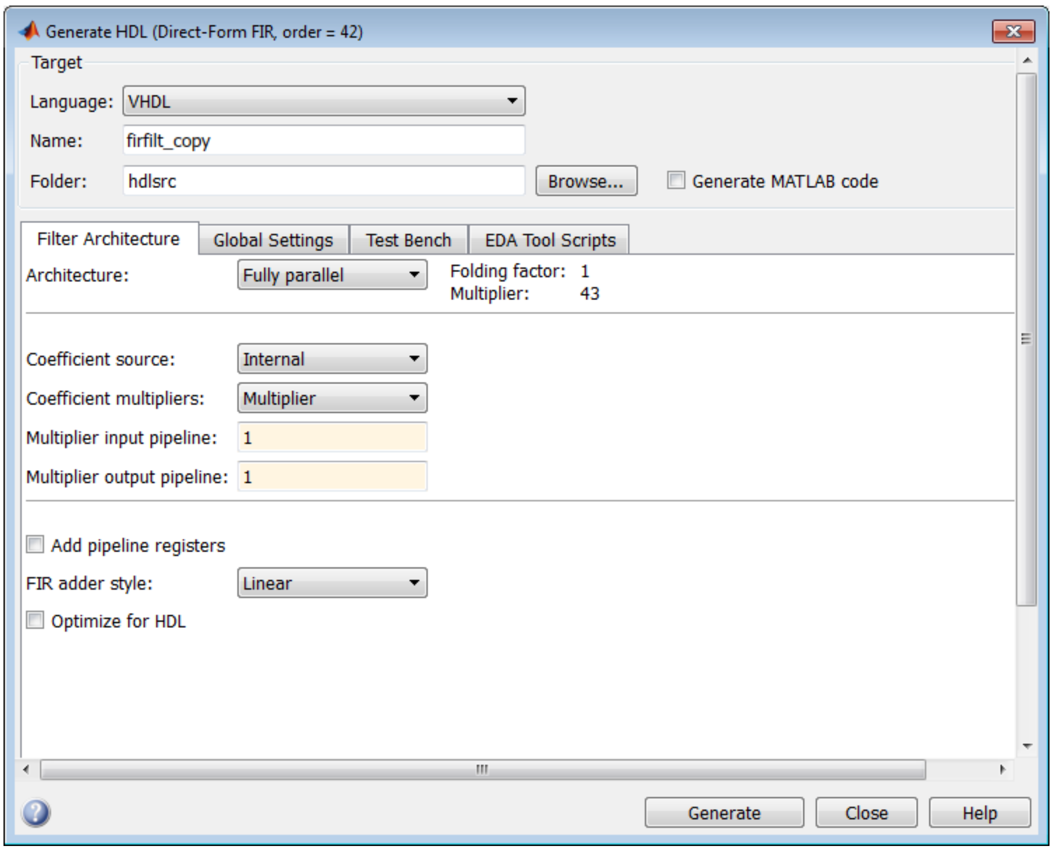
Command-Line Alternative: Use the `generatehdl` function with the property `AddPipelineRegisters` to optimize the filters with pipeline registers.

Multiplier Input and Output Pipelining for FIR Filters

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

This figure shows the UI options for multiplier pipelining options. To enable these options, **Coefficient multipliers** to `Multiplier`.

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



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

Optimizing Final Summation for FIR Filters

If you are generating HDL code for an FIR filter, consider optimizing the final summation technique to be applied to the filter. By default, the coder applies linear adder summation, which is the final summation technique discussed in most DSP text books. Alternatively, you can instruct the coder to apply tree or pipeline final summation. When set to tree mode, the coder creates a final adder that performs pairwise 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. The timing for tree mode can be better due to parallel additions.
- Pipeline mode optimizes the clock rate, but increases the filter latency. The latency increases by `log`$_2$`(number of products)`, rounded up to the nearest integer.
- Linear mode helps attain numeric accuracy in comparison to the original filter object. Tree and pipeline modes can produce numeric results that differ from the results produced by the filter object.

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

1. Select one of these options in the **Filter architecture** pane of the Generate HDL tool.

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 ignores when comparing the results. To set an error margin,
   a. Select the **Test Bench** pane in the Generate HDL tool. Then click the **Configuration** tab.
   b. Set the **Error margin (bits)** field to an integer that indicates the maximum acceptable number of bits of difference in the numeric results.
3. Continue setting other options or click **Generate** to initiate code generation.

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

Specifying or Suppressing Registered Input and Output

The coder adds an extra input register (`input_register`) and an extra output register (`output_register`) during HDL code generation. These extra registers can be useful for timing purposes, but they add to the overall latency.

This process block writes to extra input register `input_register` when a clock event occurs and `clk` is active high (1):

```
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;
```

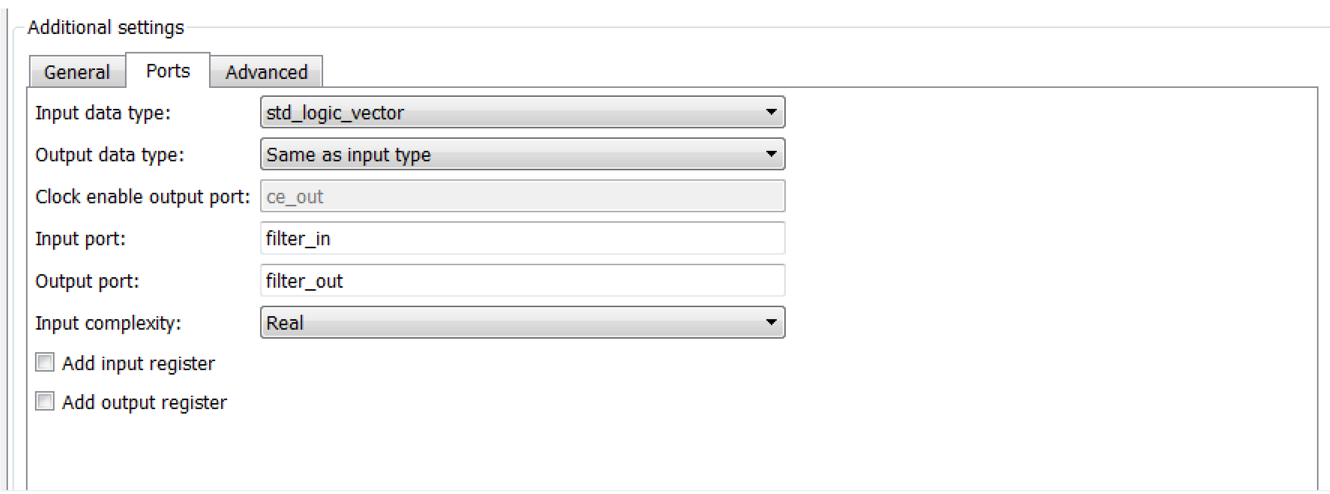
```
  END IF;
END PROCESS Input_Register_Process ;
```

This process block writes to extra output register `output_register` when a clock event occurs and `clk` is active high (1):

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

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

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



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

Overall HDL Filter Code Optimization

In this section...

Optimize for HDL

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

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

To optimize generated code for clock speed or space requirements:

1. Select **Optimize for HDL** in the **Filter architecture** pane of the Generate HDL tool.
2. Consider setting an error margin for the generated test bench. The error margin is the number of least significant bits the test bench ignores when comparing the results. To set an error margin,
   a. Select the **Test Bench** pane in the Generate HDL tool. Then click the **Configuration** tab.
   b. Set the **Error margin (bits)** field to an integer that indicates the maximum acceptable number of bits of difference in the numeric results.
3. Continue setting other options or click **Generate** to initiate code generation.

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

Set Error Margin for Test Bench

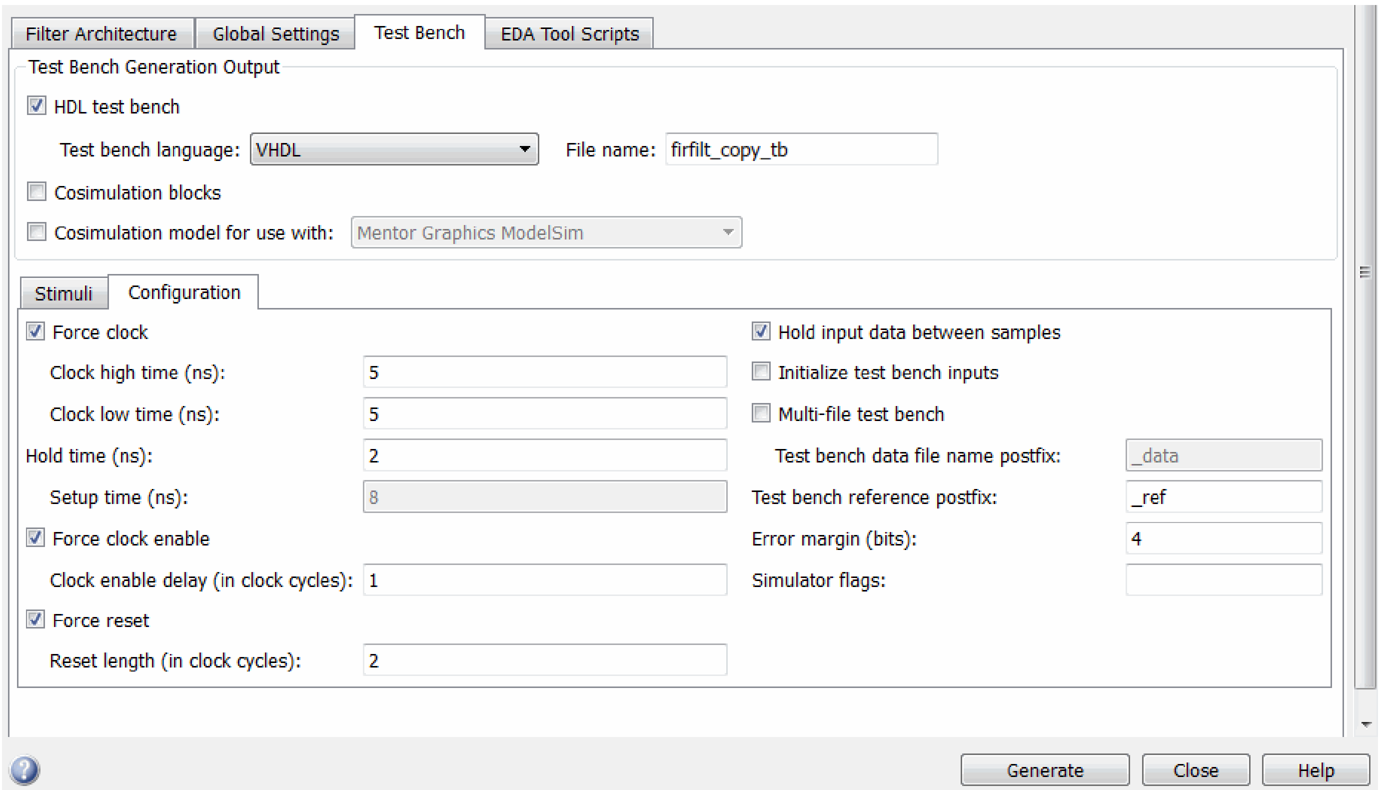
Customizations that provide optimizations can generate test bench code that produces numeric results that differ from results produced by the original filter object. These options include:

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

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

1. Select the **Test Bench** pane in the Generate HDL tool.
2. Within the **Test Bench** pane, select the **Configuration** subpane.

3 For fixed-point filters, the initial **Error margin (bits)** field has a default value of 4. To change the error margin, enter an integer in the **Error margin (bits)** field. In the figure, the error margin is set to 4 bits.



Command-Line Alternative: Use the `generatehdl` function with the property `ErrorMargin` to set the comparison tolerance.

Customization of HDL Filter Code

HDL File Names and Locations

In this section...

Setting the Location of Generated Files

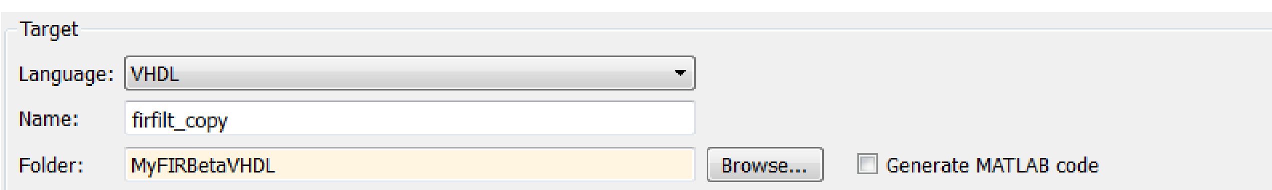
By default, the coder places generated HDL files in the subfolder `hdlsrc` under your current working folder. To direct the coder output to a folder other than the default target folder, use either the **Folder** field or the **Browse** button in the **Target** pane of the Generate HDL tool.

Clicking the **Browse** button opens a browser window that lets you select (or create) the folder where the coder puts generated files. When the folder is selected, the full path and folder name are automatically entered into the **Folder** field.

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

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

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



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

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

Naming the Generated Files and Filter Entity

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

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

Derivation of File Names

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

Language	Generated File	File Name	Example
Verilog	Source file for the quantized filter	*filt_name*.v	firfilt.v
	Source file for the test bench	*filt_name*_tb.v	firfilt_tb.v
VHDL	Source file for the quantized filter	*filt_name*.vhd	firfilt.vhd
	Source file for the test bench	*filt_name*_tb.vhd	firfilt_tb.vhd
	Package file, if required by the filter design	*filt_name*_pkg.vhd	firfilt_pkg.vhd

By default, the coder generates a single test bench file, containing test bench helper functions, data, and test bench code. You can split these elements into separate files, as described in “Splitting Test Bench Code and Data into Separate Files” on page 6-10.

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

Derivation of Entity Names

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

Set HDL File Name Extensions

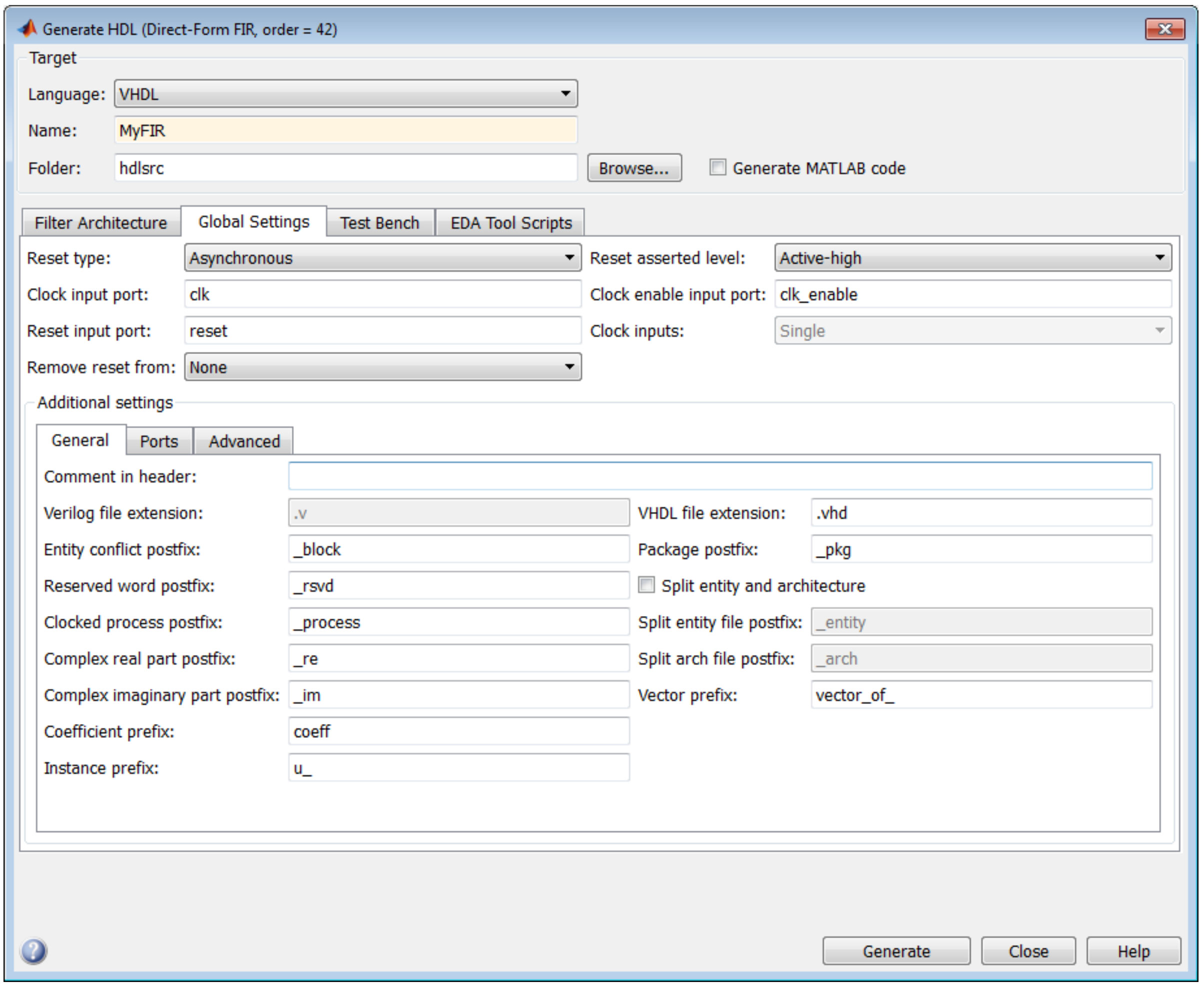
- “Set File Name Extension Via the Generate HDL Tool” on page 5-4
- “Set HDL File Name Extensions Via the Command-Line” on page 5-5

Set File Name Extension Via the Generate HDL Tool

When you select VHDL code generation, by default the filter HDL files are generated with a `.vhd` file extension. When you select Verilog, the default file extension is `.v`. To change the file extension,

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Type the new file extension in either the **VHDL file extension** or **Verilog file extension** field. The field for the language you have not selected is disabled.

This figure shows how to specify an alternate file extension for VHDL files. The coder generates the filter file `MyFIR.vhdl`.



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

Set HDL File Name Extensions Via the Command-Line

Command-Line Alternative: Use the `generatehdl` function with the `Name` property to set the name of your filter entity and the base character vector for generated HDL file names. To specify an alternative file type extension for generated files, call the function with the `VerilogFileExtension` or `VHDLFileExtension` property.

Splitting Entity and Architecture Code Into Separate Files

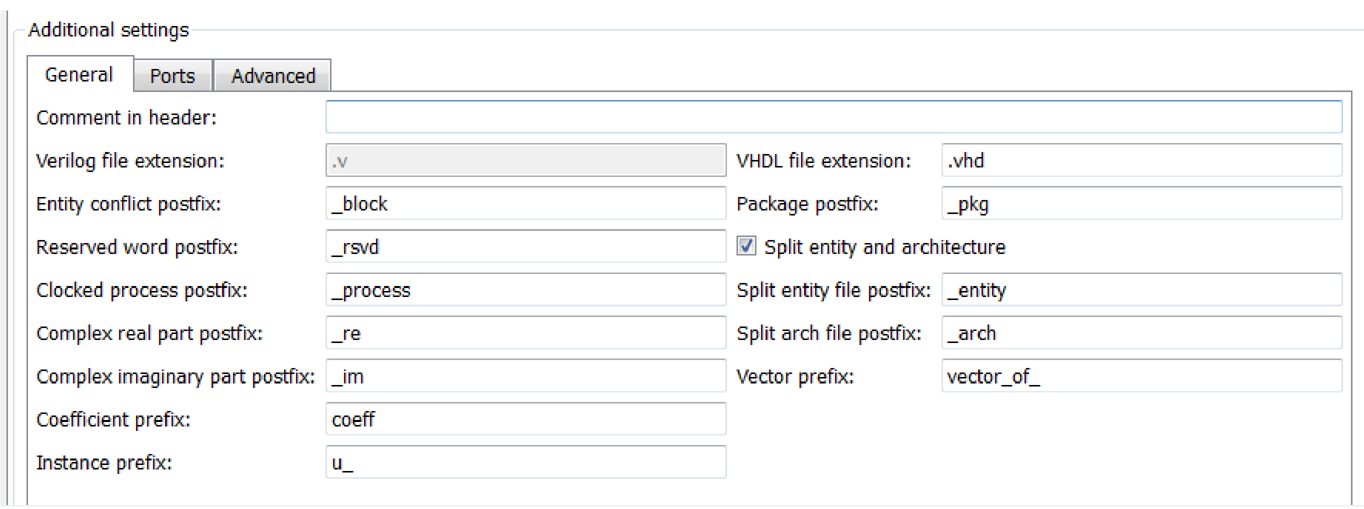
By default, the coder includes a VHDL entity and architecture code in the same generated VHDL file. Alternatively, you can instruct the coder to place the entity and architecture code in separate files. For example, instead of generated code residing in `MyFIR.vhd`, you can specify that the code reside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

The names of the entity and architecture files derive from:

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

To split the filter source file, do these steps.

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



4. Specify new character vectors in the postfix fields if you want to use postfixes other than `'_entity'` and `'_arch'` to identify the generated VHDL files.

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

Command-Line Alternative: Use the `generatehdl` function with the property `SplitEntityArch` to split the VHDL code into separate files. To modify the file name postfix for the separate entity and architecture files, use the `SplitEntityFilePostfix` and `SplitArchFilePostfix` properties.

HDL Identifiers and Comments

In this section...

Specifying a Header Comment

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

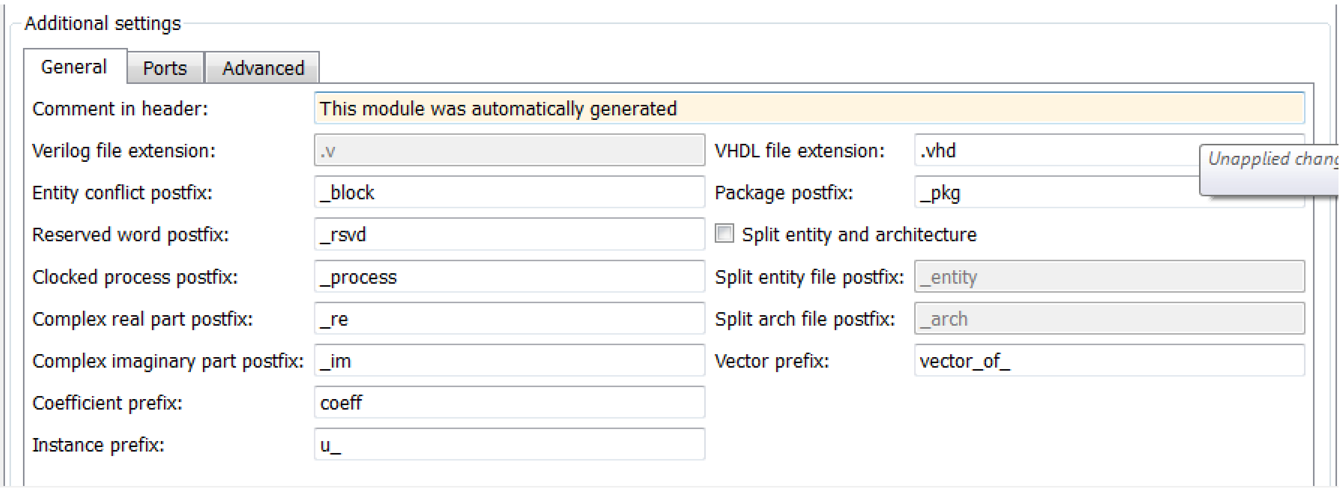
You can use the **Comment in header** option to add a comment to the end of the header comment block in each generated file. For example, use this option to add "`This module was automatically generated`". With this change, the preceding header comment block would appear as follows:

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

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

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

-- -------------------------------------------------------------
-- HDL Implementation    : Fully parallel
-- Multipliers           : 43
-- Folding Factor        : 1
-- -------------------------------------------------------------
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- ------------------------------
-- Filter Structure  : Direct-Form FIR
```

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

To add a header comment,

1 Select the **Global Settings** tab on the Generate HDL tool.
2 Select the **General** tab in the **Additional settings** pane.
3 Type the comment text in the **Comment in header** field, as shown in this figure.



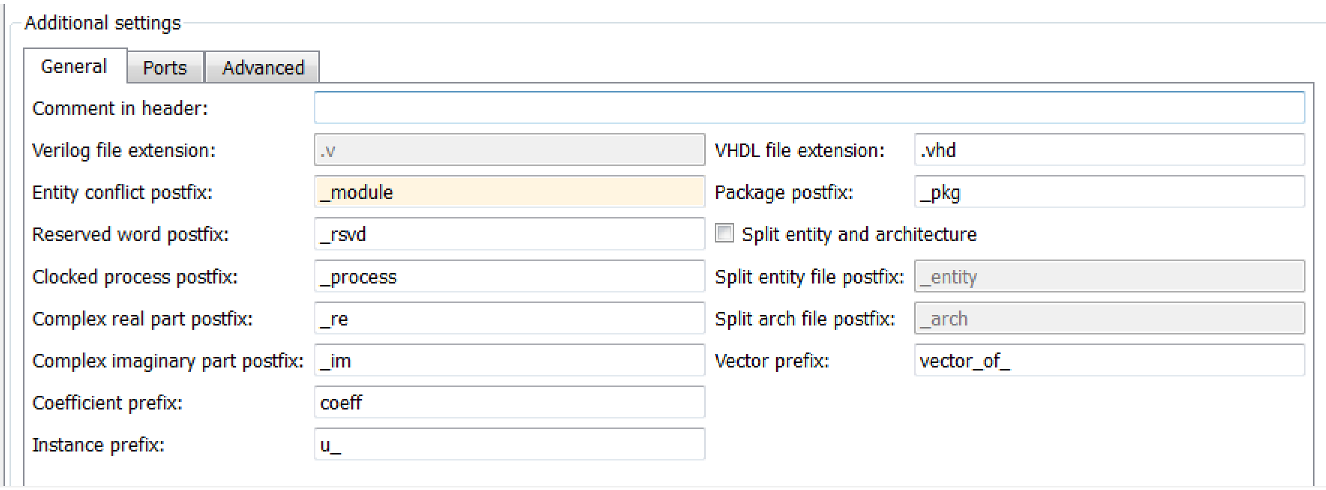
Command-Line Alternative: Use the `generatehdl` function with the property `UserComment` to add a comment to the end of the header comment block in each generated HDL file.

Resolving Entity or Module Name Conflicts

The coder checks whether multiple entities in VHDL or multiple modules in Verilog share the same name. If a name conflict exists, the coder appends the postfix `'_block'` to the second of the two matching character vectors.

To change the postfix:

1 Select the **Global Settings** tab on the Generate HDL tool.
2 Select the **General** tab in the **Additional settings** pane.
3 Enter a new character vector in the **Entity conflict postfix** field, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `EntityConflictPostfix` to change the entity or module conflict postfix.

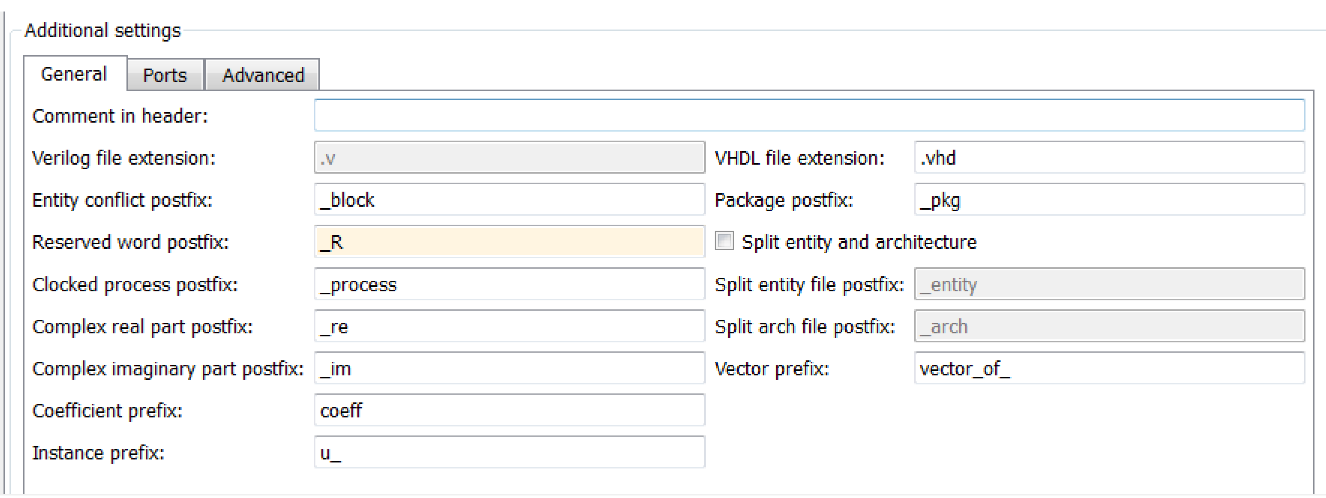
Resolving HDL Reserved Word Conflicts

The coder checks whether character vectors that you specify as names, postfix values, or labels are VHDL or Verilog reserved words. See “Reserved Word Tables” on page 5-10 for listings of VHDL and Verilog reserved words.

If you specify a reserved word, the coder appends the postfix `_rsvd` to the character vector. For example, if you try to name your filter `mod`, for VHDL code, the coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

To change the postfix:

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new character vector in the **Reserved word postfix** field, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `ReservedWordPostfix` to change the reserved word postfix.

Reserved Word Tables

The tables list VHDL and Verilog reserved words.

VHDL Reserved Words

abs	access	after	alias	all
and	architecture	array	assert	attribute
begin	block	body	buffer	bus
case	component	configuration	constant	disconnect
downto	else	elsif	end	entity
exit	file	for	function	generate
generic	group	guarded	if	impure
in	inertial	inout	is	label
library	linkage	literal	loop	map
mod	nand	new	next	nor
not	null	of	on	open
or	others	out	package	port
postponed	procedure	process	pure	range
record	register	reject	rem	report
return	rol	ror	select	severity
signal	shared	sla	sll	sra
srl	subtype	then	to	transport
type	unaffected	units	until	use
variable	wait	when	while	with
xnor	xor			

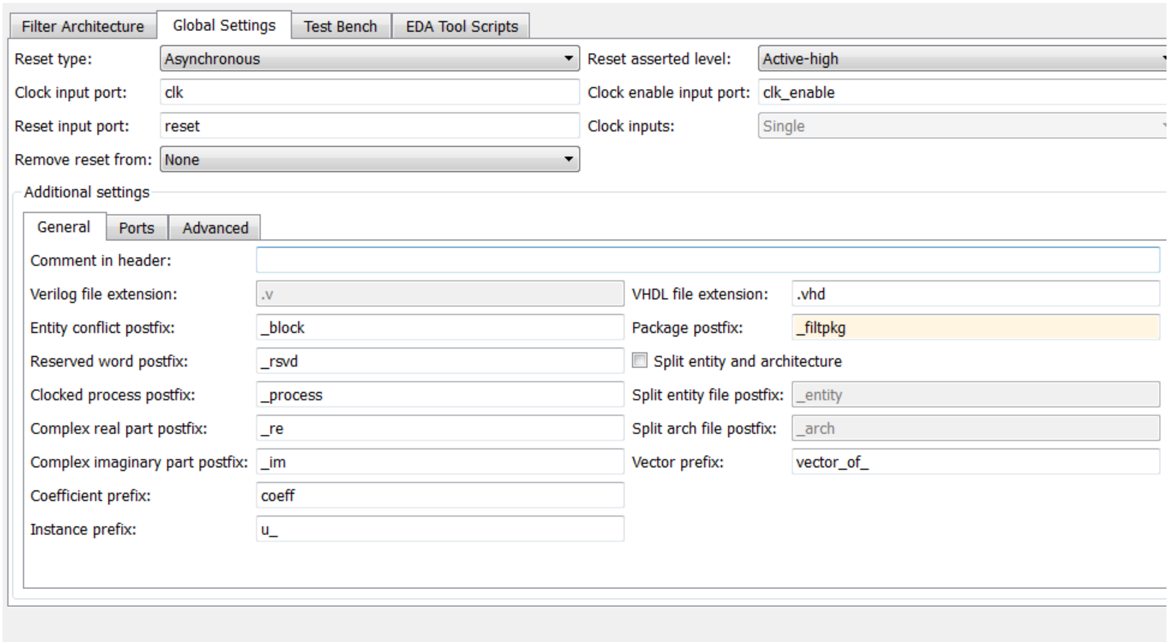
Verilog Reserved Words

`always`	`and`	`assign`	`automatic`	`begin`
`buf`	`bufif0`	`bufif1`	`case`	`casex`
`casez`	`cell`	`cmos`	`config`	`deassign`
`default`	`defparam`	`design`	`disable`	`edge`
`else`	`end`	`endcase`	`endconfig`	`endfunction`
`endgenerate`	`endmodule`	`endprimitive`	`endspecify`	`endtable`
`endtask`	`event`	`for`	`force`	`forever`
`fork`	`function`	`generate`	`genvar`	`highz0`
`highz1`	`if`	`ifnone`	`incdir`	`include`
`initial`	`inout`	`input`	`instance`	`integer`
`join`	`large`	`liblist`	`library`	`localparam`
`macromodule`	`medium`	`module`	`nand`	`negedge`
`nmos`	`nor`	`noshowcancelled`	`not`	`notif0`
`notif1`	`or`	`output`	`parameter`	`pmos`
`posedge`	`primitive`	`pull0`	`pull1`	`pulldown`
`pullup`	`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 for VHDL Package Files

By default, the coder appends the postfix `_pkg` to the base file name when generating a VHDL package file. To rename the postfix for package files, do these steps.

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Specify a new value in the **Package postfix** field.



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

Command-Line Alternative: Use the `generatehdl` function with the `PackagePostfix` property to rename the file name postfix for VHDL package files.

Specifying a Prefix for Filter Coefficients

The coder declares the coefficients for the filter as constants within a `rtl` architecture. The coder derives the constant names adding the prefix `coeff`. The coefficient names depend on the type of filter.

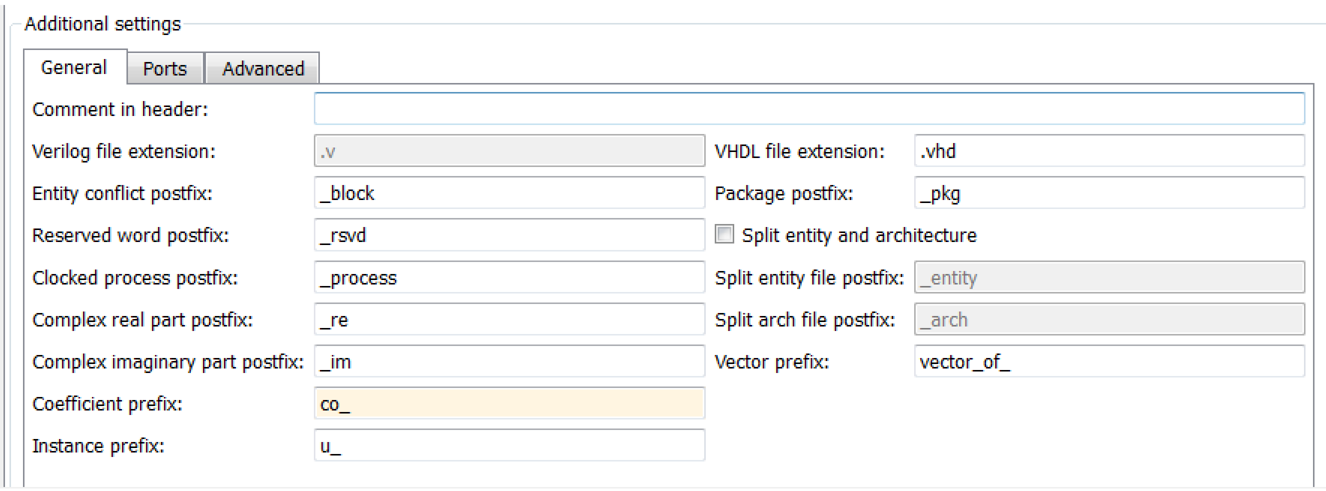
For...	The Prefix Is Concatenated with...
FIR filters	Each coefficient number, starting with 1. Examples: `coeff1`, `coeff22`
IIR filters	An underscore (_) and an `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) followed by `_sectionn`, where *n* is the section number. Example: `coeff_b1_section3` (first numerator coefficient of the third section)

For example:

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

To use a prefix other than `coeff`,

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new character vector in the **Coefficient prefix** field, as shown in this figure.



The character vector that you specify

- Must start with a letter.
- Cannot include a double underscore (__).

Note If you specify a VHDL or Verilog reserved word, the coder appends a reserved word postfix to the character vector to form a valid identifier. If you specify a prefix that ends with an underscore, the coder replaces the underscore character with `under`. For example, if you specify `coef_`, the coder generates coefficient names such as `coefunder1`.

Command-Line Alternative: Use the `generatehdl` function with the property `CoeffPrefix` to change the base name for filter coefficients.

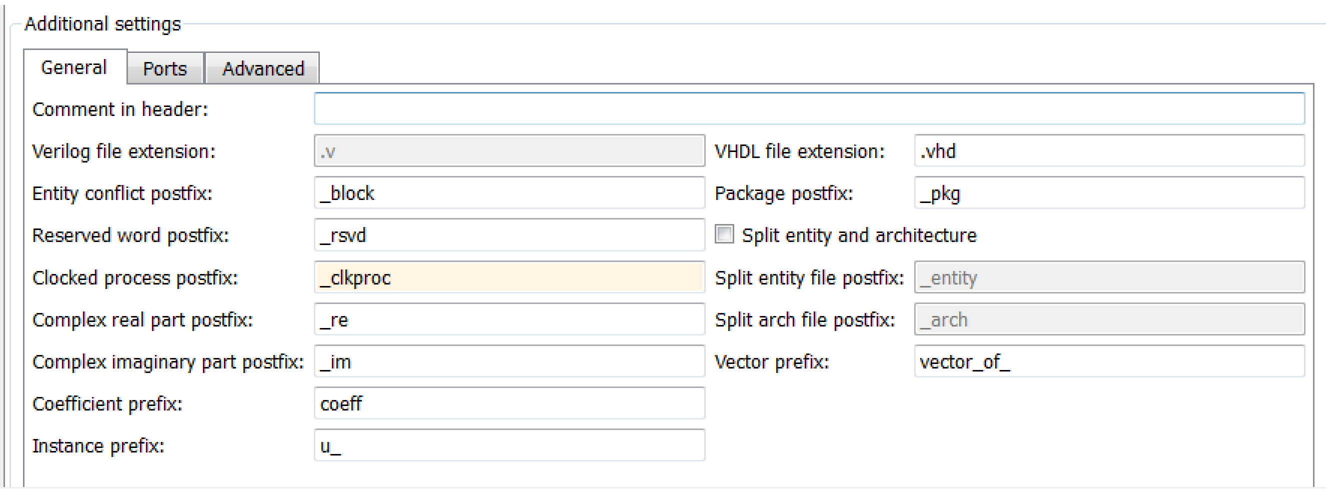
Specifying a Postfix for Process Block Labels

The coder generates process blocks to modify the content of the 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 `'_process'`.

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

The **Clocked process postfix** property lets you change the postfix to a value other than `'_process'`. For example, to change the postfix to `'_clkproc'`, do these steps.

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new character vector in the **Clocked process postfix** field, as shown in this figure.



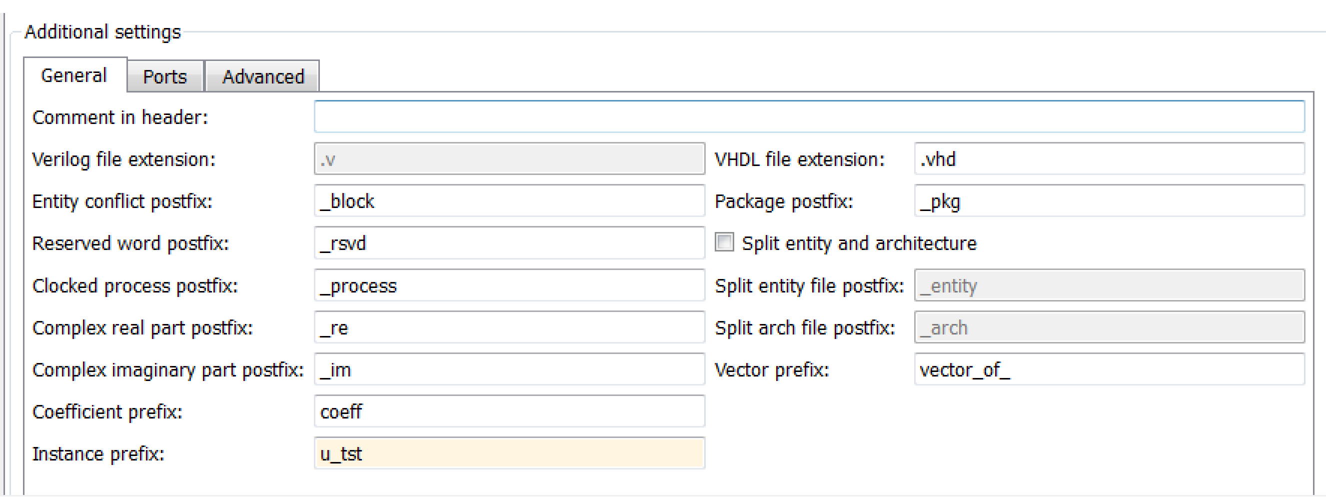
Command-Line Alternative: Use the `generatehdl` function with the property `ClockProcessPostfix` to change the postfix appended to process labels.

Setting a Prefix for Component Instance Names

Instance prefix specifies a character vector to be prefixed to component instance names in generated code. The default is `'u_'`.

You can set the prefix to a value other than `'u_'`. To change the prefix:

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new character vector in the **Instance prefix** field, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `InstancePrefix` to change the instance prefix.

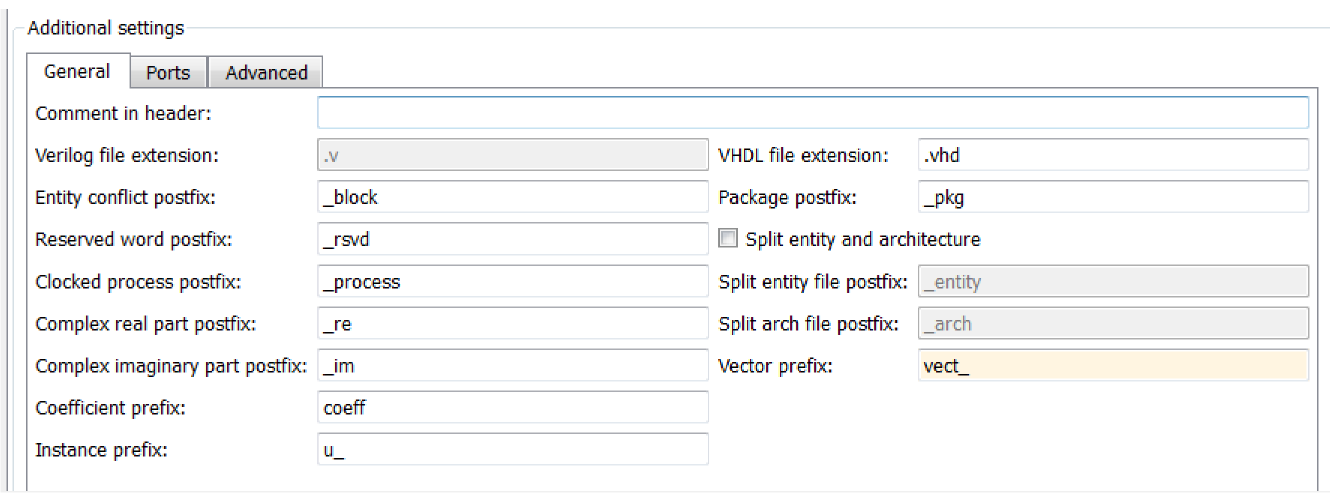
Setting a Prefix for Vector Names

Vector prefix specifies a character vector to be prefixed to vector names in generated VHDL code. The default is `'vector_of_'`.

Note **Vector prefix** is not supported for Verilog code generation.

You can set the prefix to a value other than `'vector_of_'`. To change the prefix:

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new character vector in the **Vector prefix** field, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `VectorPrefix` to change the instance prefix.

Ports and Resets

In this section...

Naming HDL Ports

The default names for filter HDL ports are as follows:

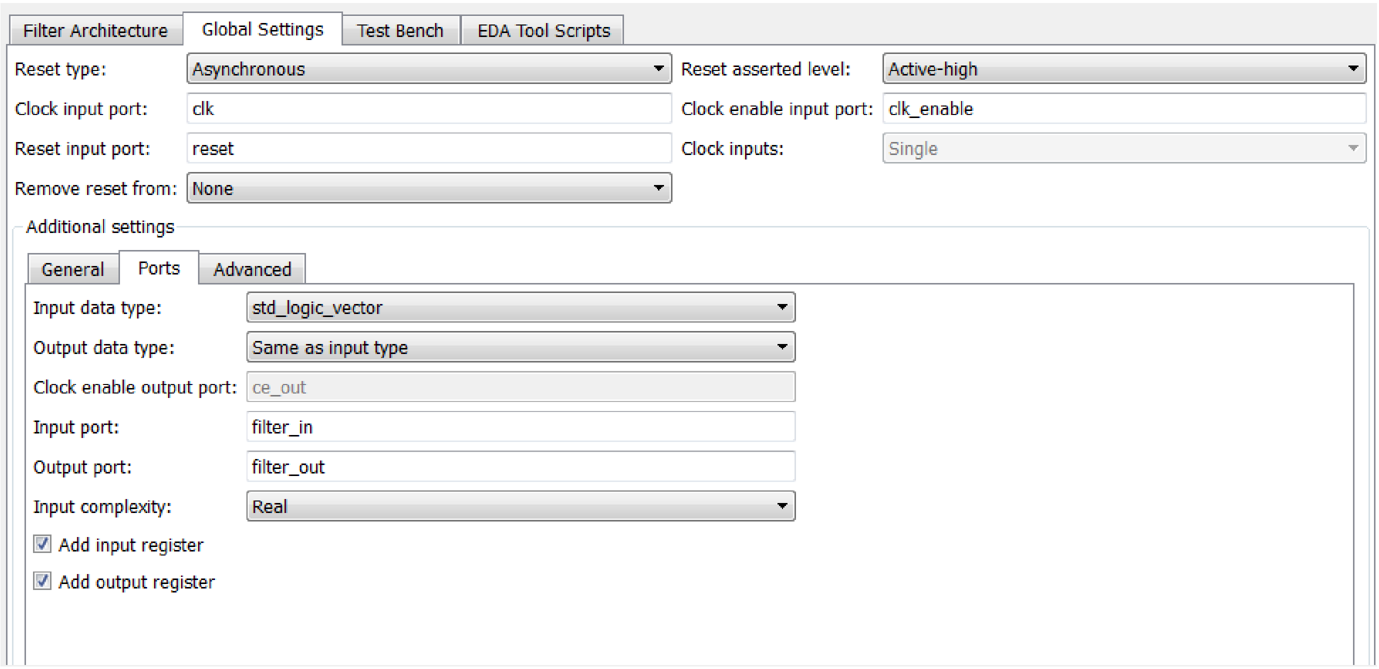
HDL Port	Default Port Name
Input port	`filter_in`
Output port	`filter_out`
Clock port	`clk`
Clock enable port	`clk_enable`
Reset port	`reset`
Fractional delay port (Farrow filters only)	`filter_fd`

For example, this code shows the default VHDL declaration for entity `filt`.

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

To change port names,

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **Ports** tab in the **Additional settings** pane. This figure highlights the port name fields for **Input port**, **Output port**, **Clock input port**, **Reset input port**, and **Clock enable output port**.



3 Enter new character vectors in the port name fields.

Command-Line Alternative: Use the `generatehdl` function with the properties `InputPort`, `OutputPort`, `ClockInputPort`, `ClockEnableInputPort`, and `ResetInputPort` to change the names of the filter ports in the generated HDL code.

Specifying the HDL Data Type for Data Ports

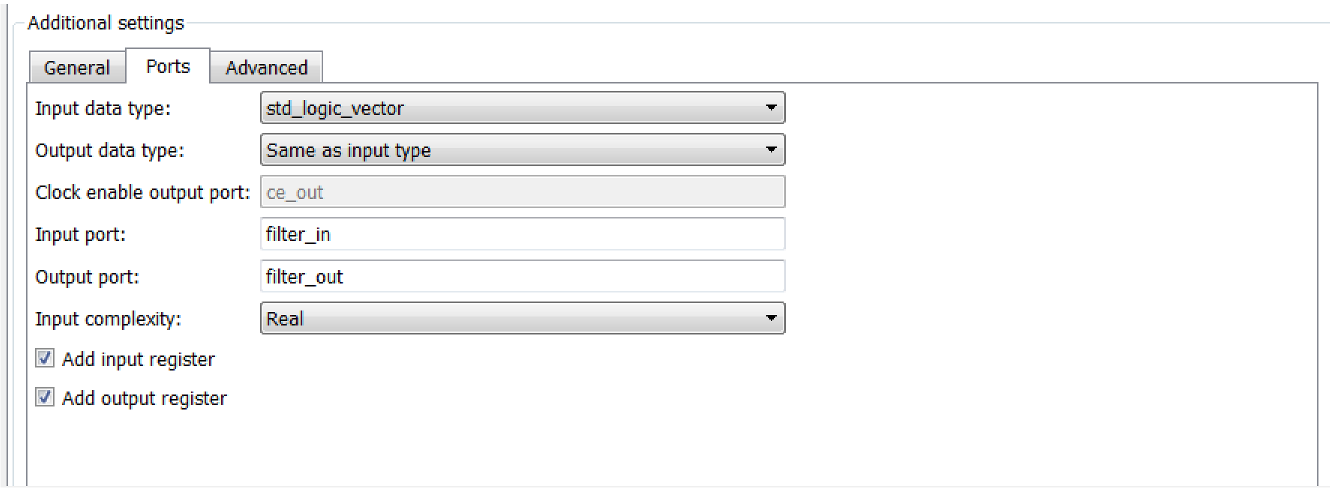
By default, filter input and output data ports have data type `std_logic_vector` in VHDL and type `wire` in Verilog. If you are generating VHDL code, alternatively, you can specify `signed/unsigned`, and for output data ports, `Same as input data type`. The coder applies type `SIGNED` or `UNSIGNED` based on the data type specified in the filter design.

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

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **Ports** tab in the **Additional settings** pane.
3. Select a data type from the **Input data type** or **Output data type** menu identified in this figure.

   By default, the output data type is the same as the input data type.

   The type for Verilog ports is `wire`, and cannot be changed.



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

Command-Line Alternative: Use the generatehdl function with the properties InputType and OutputType to change the VHDL data type for the input and output ports.

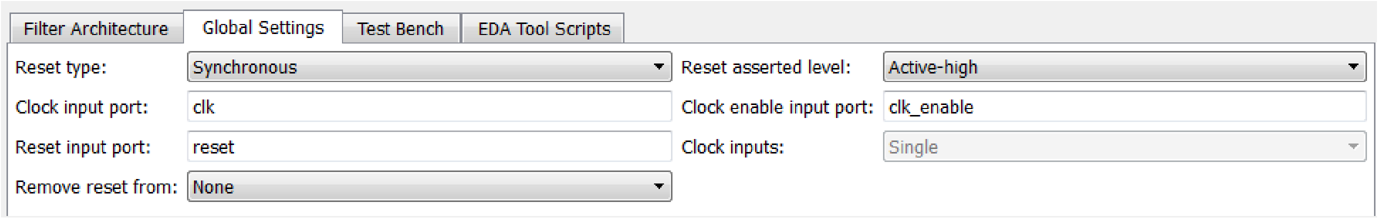
Selecting Asynchronous or Synchronous Reset Logic

By default, generated HDL code for registers uses asynchronous reset logic. Select asynchronous or synchronous reset logic depending on the type of device you are designing (for example, FPGA or ASIC) and preference.

This code fragment illustrates the use of asynchronous resets. The process block does not check for an active clock before performing a reset.

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

To change the reset type to synchronous, select Synchronous from the **Reset type** menu in the **Global settings** pane of the Generate HDL tool.



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

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

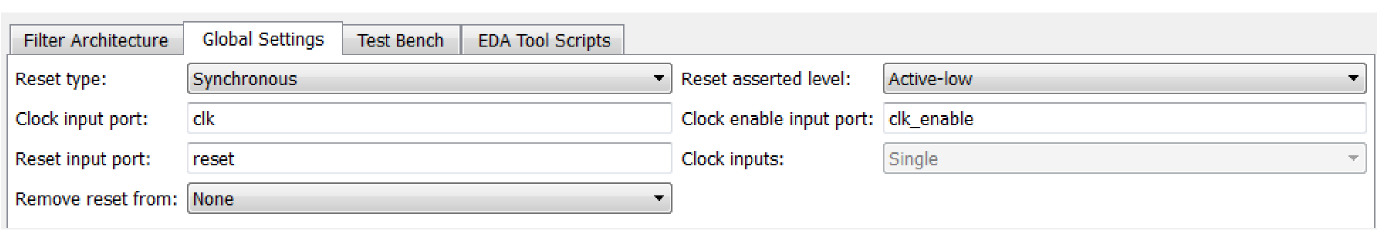
Command-Line Alternative: Use the `generatehdl` function with the property `ResetType` to set the reset style for the registers in the generated HDL code.

Setting the Asserted Level for the Reset Input Signal

The asserted level for the reset input signal determines whether that signal must be driven to active high (1) or active low (0) for registers to be reset in the filter design. By default, the coder sets the asserted level to active high. For example, this code fragment checks whether `reset` is active high before populating the `delay_pipeline` register:

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

To change the setting to active low, select `Active-low` from the **Reset asserted level** menu in the **Global settings** pane of the Generate HDL tool.



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

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

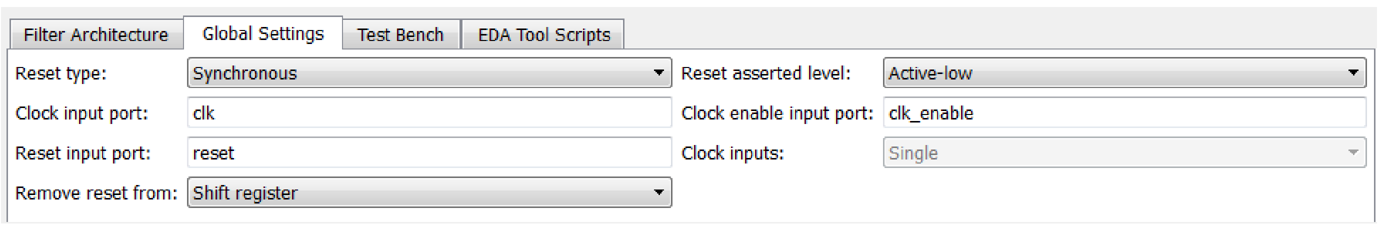
Note The **Reset asserted level** setting also determines the reset level for test bench reset input signals.

Command-Line Alternative: Use the `generatehdl` function with the property `ResetAssertedLevel` to set the asserted level for the reset input signal.

Suppressing Generation of Reset Logic

For some FPGA applications, it is desirable to avoid generation of resets. The **Remove reset from** option in the **Global settings** pane of the Generate HDL tool lets you suppress generation of resets from shift registers.

To suppress generation of resets from shift registers, select `Shift register` from the **Remove reset from** pull-down menu in the **Global settings** pane of the Generate HDL tool.



If you do not want to suppress generation of resets from shift registers, leave **Remove reset from** set to its default, which is `None`.

Command-Line Alternative: Use the `generatehdl` function with the property `RemoveResetFrom` to suppress generation of resets from shift registers.

HDL Constructs

In this section...

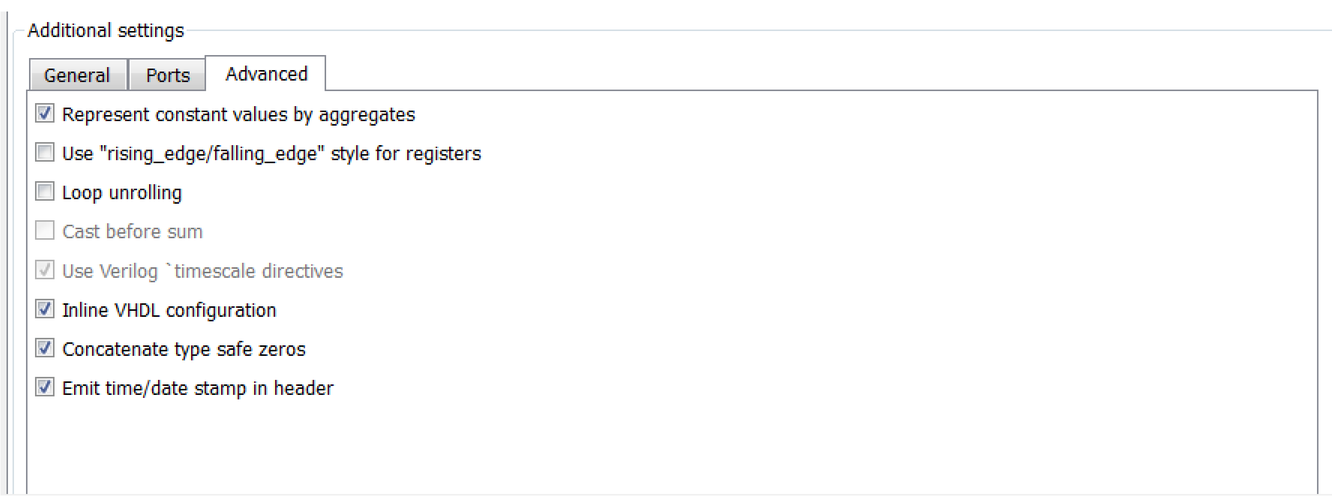
Representing VHDL Constants with Aggregates

By default, the coder represents constants as scalars or aggregates depending on the size and type of the data. The coder represents values that are less than $2^{32} – 1$ as integers and values greater than or equal to $2^{32} – 1$ as aggregates. These VHDL constant declarations are examples of declarations generated by default for values less than 32 bits:

```
CONSTANT coeff1: signed(15 DOWNTO 0) := to_signed(-60, 16); -- sfix16_En16
CONSTANT coeff2: signed(15 DOWNTO 0) := to_signed(-178, 16); -- sfix16_En16
```

If you prefer that constant values be represented as aggregates, set the **Represent constant values by aggregates** as follows:

1 Select the **Global Settings** tab on the Generate HDL tool.
2 Select the **Advanced** tab.
3 Select **Represent constant values by aggregates**, as shown this figure.



The preceding constant declarations would now appear as follows:

```
CONSTANT coeff1: signed(15 DOWNTO 0) := (5 DOWNTO 3 => '0',1 DOWNTO 0 => '0,OTHERS =>'1');
CONSTANT coeff2: signed(15 DOWNTO 0) := (7 => '0',5 DOWNTO 4 => '0',0 => '0',OTHERS =>'1');
```

Command-Line Alternative: Use the `generatehdl` function with the property `UseAggregatesForConst` to represent constants in the HDL code as aggregates.

Unrolling and Removing VHDL Loops

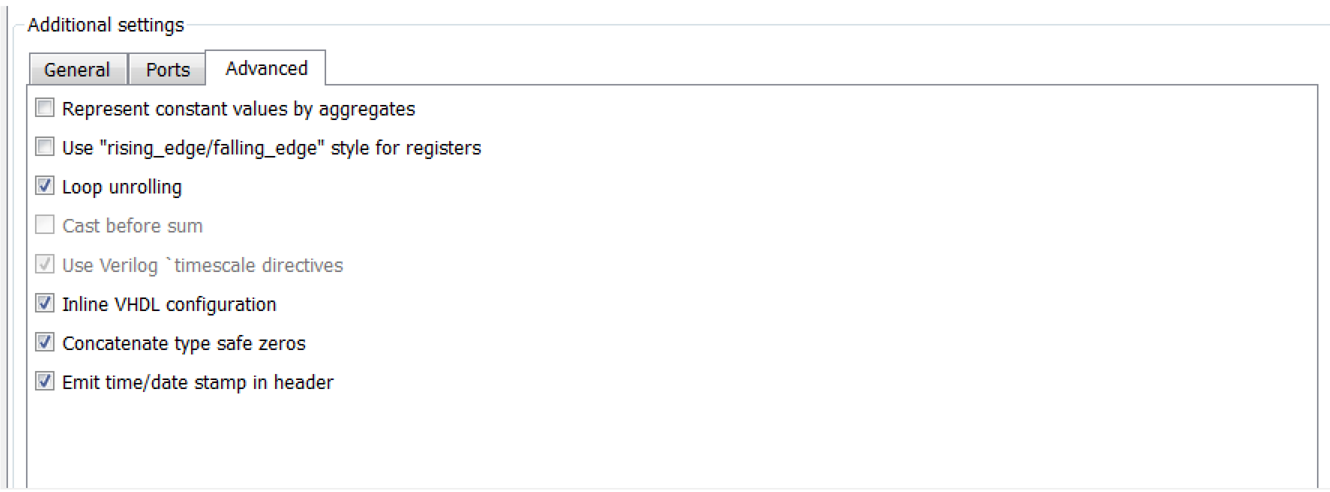
By default, the coder supports VHDL loops. However, some EDA tools do not support them. If you are using such a tool along with VHDL, you can unroll and remove `FOR` and `GENERATE` loops from the generated VHDL code. Verilog code is already unrolled.

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

1 Select the **Global Settings** tab on the Generate HDL tool.

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

3 Select **Loop unrolling**, as shown in this figure.



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

Using the VHDL rising_edge Function

The coder can generate two styles of VHDL code for checking for rising edges when the filter operates on registers. By default, the generated code checks for a clock event, as shown in the `ELSIF` statement of this VHDL process block.

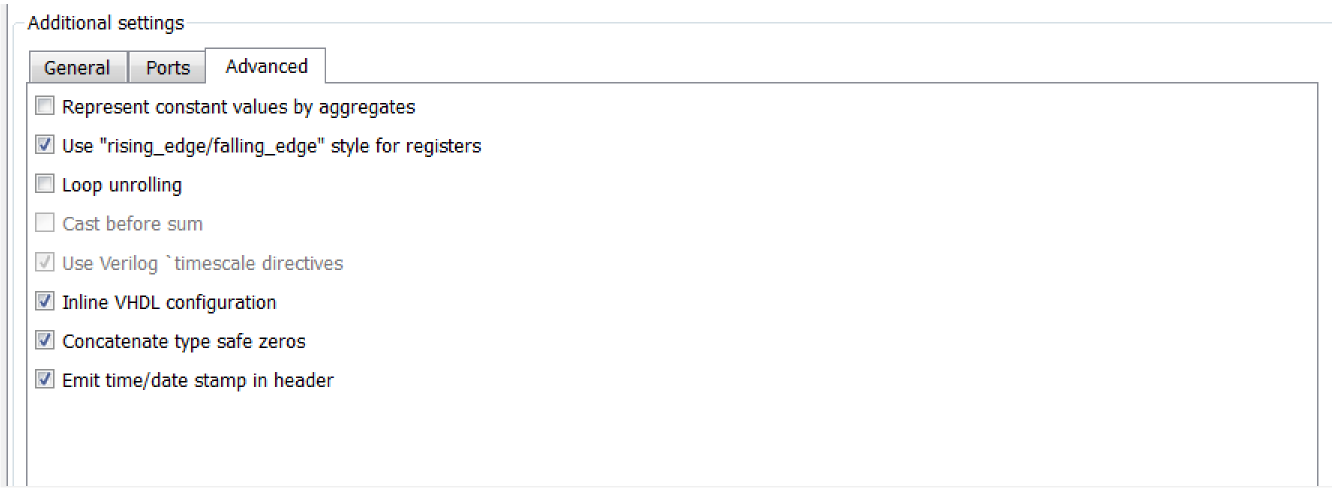
```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
  ELSEIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
      delay_pipeline(1 TO 50) <= delay_pipeline(0 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 this statement:

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

To use the `rising_edge` function,

1 Click **Global Settings** in the Generate HDL tool.
2 Select the **Advanced** tab. The **Advanced** pane appears.
3 Select **Use 'rising_edge' for registers**, as shown in this figure.



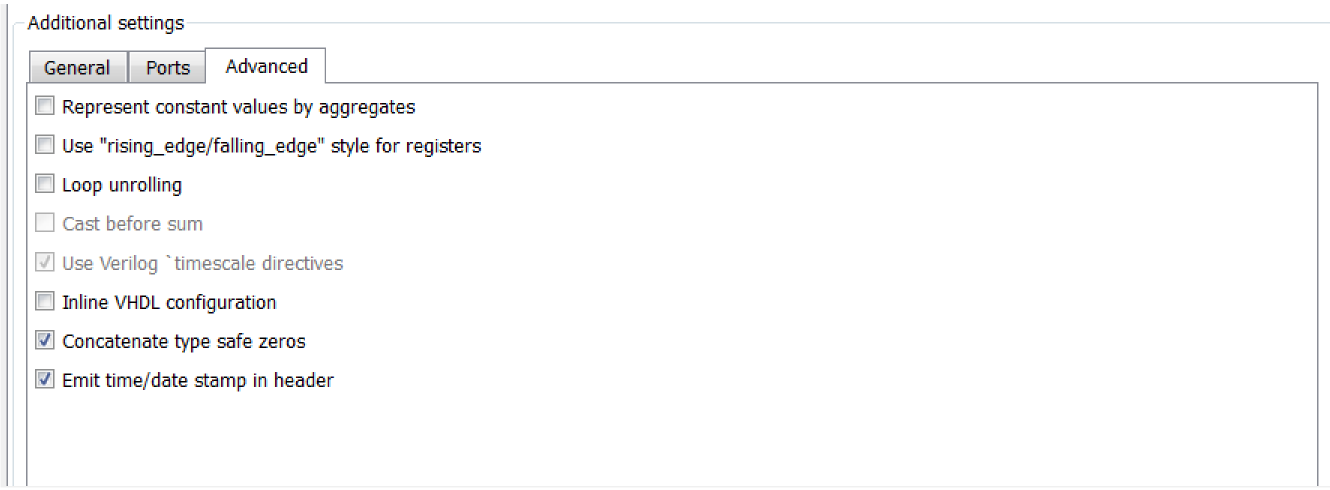
Command-Line Alternative: Use the `generatehdl` function with the property `UseRisingEdge` to use the VHDL `rising_edge` function to check for rising edges during register operations.

Suppressing the Generation of VHDL Inline Configurations

VHDL configurations can be either inline with the rest of the VHDL code for an entity or external in separate VHDL source files. By default, the coder includes configurations for a filter within the generated VHDL code. If you are creating your own VHDL configuration files, suppress the generation of inline configurations.

To suppress the generation of inline configurations,

1 Select the **Global Settings** tab on the Generate HDL tool.
2 Select the **Advanced** tab. The **Advanced** pane appears.
3 Clear **Inline VHDL configuration**, as shown in this figure.



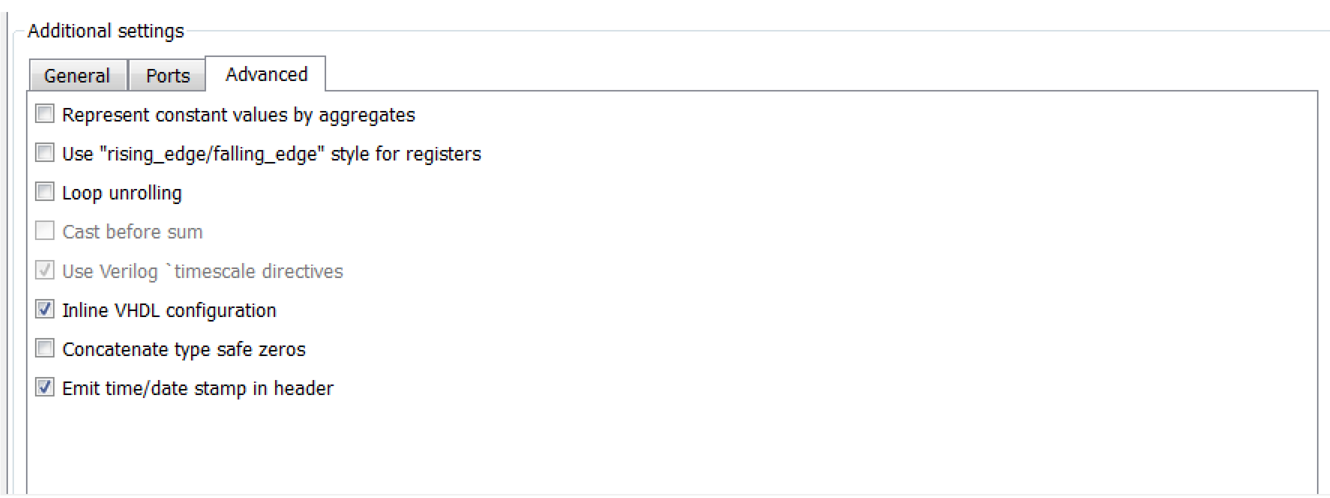
Command-Line Alternative: Use the `generatehdl` function with the property `InlineConfigurations` to suppress the generation of inline configurations.

Specifying VHDL Syntax for Concatenated Zeros

In VHDL, the concatenation of zeros can be represented in two syntax forms. One form, `'0' & '0'`, is type-safe. This syntax is the default. The alternative syntax, `"000000..."`, can be easier to read and is more compact, but can lead to ambiguous types.

To use the syntax `"000000..."` for concatenated zeros,

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **Advanced** tab. The **Advanced** pane appears.
3. Clear **Concatenate type safe zeros**, as shown in this figure.



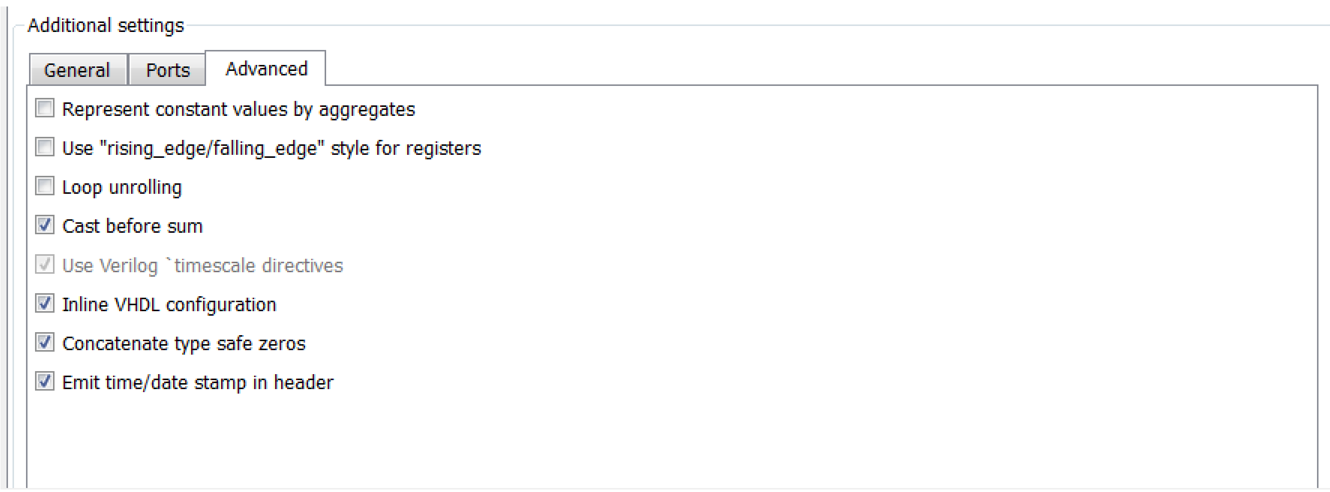
Command-Line Alternative: Use the `generatehdl` function with the property `SafeZeroConcat` to use the syntax `"000000..."`, for concatenated zeros.

Specifying Input Type Treatment for Addition and Subtraction Operations

By default, generated HDL code operates on input data using data types as specified by the filter design, and then converts the result to the specified result type.

Typical DSP processors type cast input data to the result type *before* operating on the data. Depending on the operation, the results can be different. If you want generated HDL code to handle result typing in this way, use the **Cast before sum** option as follows:

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **Advanced** tab. The **Advanced** pane appears.
3. Select **Cast before sum**, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `CastBeforeSum` to cast input values to the result type for addition and subtraction operations.

Relationship With Cast Before Sum in Filter Designer

The **Cast before sum** option is related to the Filter Designer setting for the quantization option **Cast signals before sum** as follows:

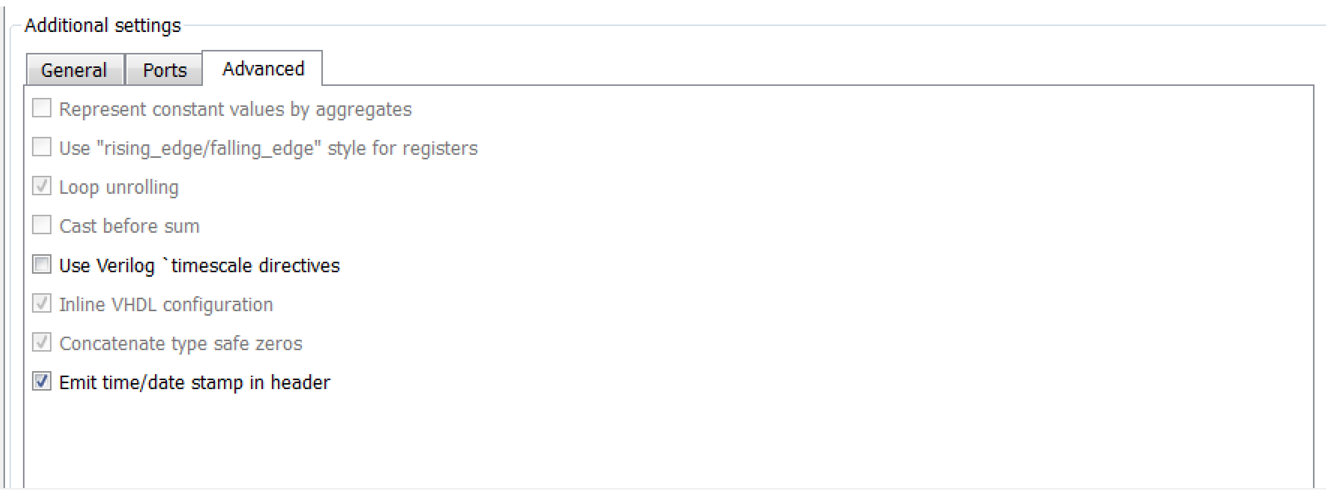
- Some filter object types do not have the **Cast signals before sum** property. For such filter objects, **Cast before sum** is effectively off when HDL code is generated; it is not relevant to the filter.
- Where the filter object does have the **Cast signals before sum** property, the coder by default follows the setting of **Cast signals before sum** in the filter object. This setting is visible in the UI. If you change the setting of **Cast signals before sum**, the coder updates the setting of **Cast before sum**.
- However, by explicitly setting **Cast before sum**, you can override the **Cast signals before sum** setting passed in from Filter Designer.

Suppressing Verilog Time Scale Directives

In Verilog, the coder generates time scale directives (`` `timescale ``) by default. This compiler directive provides a way of specifying different delay values for multiple modules in a Verilog file.

To suppress the use of `` `timescale `` directives,

1. Select the **Global Settings** tab on the Generate HDL tool.
2. Select the **Advanced** tab. The **Advanced** pane appears.
3. Clear **Use Verilog `timescale directives**, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` function with the property `UseVerilogTimescale` to suppress the use of time scale directives.

Using Complex Data and Coefficients

The coder supports complex coefficients and complex input signals.

Enabling Code Generation for Complex Data

To generate ports and signal paths for the real and imaginary components of a complex input signal, set **Input complexity** to `Complex`. The default setting for **Input complexity** is `Real`, disabling generation of ports for complex input data.

The corresponding command-line property is `InputComplex`. By default, `InputComplex` is set to `'off'`, disabling generation of ports for complex input data. To enable generation of ports for complex input data, set `InputComplex` to `'on'`, as in this code example:

```
filt = design(fdesign.lowpass,'equiripple','Filterstructure','dffir','SystemObject',true);
generatehdl(filt,numerictype(1,16,15),'InputComplex','on')
```

This VHDL code excerpt shows the entity definition generated by the preceding commands:

```
ENTITY firfilt IS
   PORT( clk               :   IN    std_logic;
         clk_enable        :   IN    std_logic;
         reset             :   IN    std_logic;
         filter_in_re      :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
         filter_in_im      :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
         filter_out_re     :   OUT   std_logic_vector(37 DOWNTO 0); -- sfix38_En31
```

```
        filter_out_im      :   OUT   std_logic_vector(37 DOWNTO 0)  -- sfix38_En31
        );

END firfilt;
```

In the code excerpt, the port names generated for the real components of complex signals have the default postfix `'_re'`, and port names generated for the imaginary components of complex signals have the default postfix `'_im'`.

Setting the Port Name Postfix for Complex Ports

Two code generation properties let you customize naming conventions for the real and imaginary components of complex signals in generated HDL code. These properties are:

- The **Complex real part postfix** option (corresponding to the `ComplexRealPostfix` command-line property) specifies a character vector to be appended to the names generated for the real part of complex signals. The default postfix is `'_re'`.
- The **Complex imaginary part postfix** option (corresponding to the `ComplexImagPostfix` command-line property) specifies a character vector to be appended to the names generated for the imaginary part of complex signals. The default postfix is `'_im'`.

Verification of Generated HDL Filter Code

Testing with an HDL Test Bench

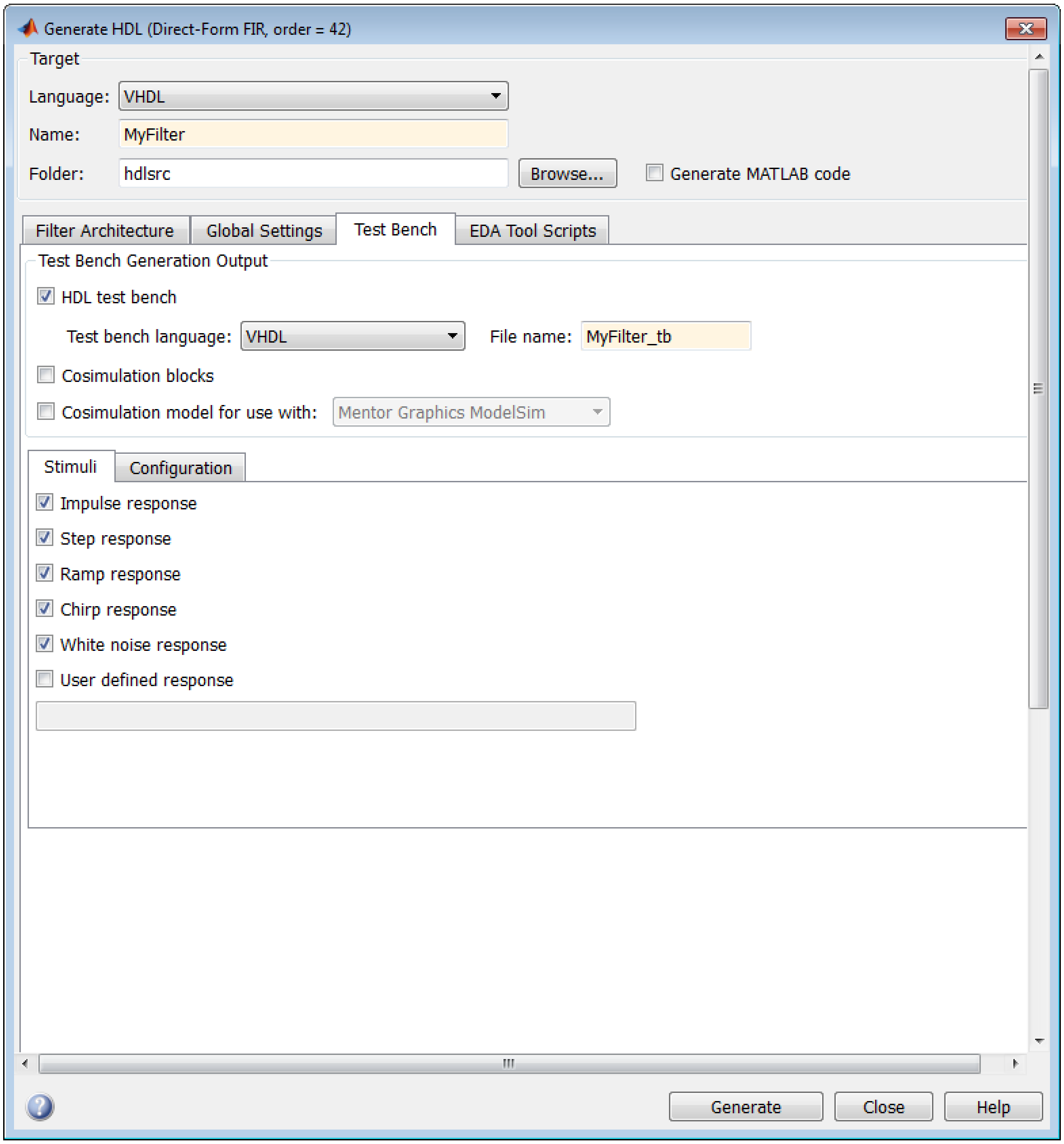
In this section...

Workflow for Testing with an HDL Test Bench

Generating the Filter and Test Bench HDL Code

Use the Filter Design HDL Coder UI or command-line interface to generate the HDL code for your filter design and test bench. The UI generates a VHDL or Verilog test bench file, depending on your language selection for the generated HDL code. You can specify a different test bench language by selecting the **Test bench language** option in the **Test Bench** pane of the Generate HDL tool. You cannot specify a different test bench language when using the command-line interface.

This figure shows settings for generating the filter (VHDL) and test bench (Verilog) files `MyFilter.vhd`, and `MyFilter_tb.v`. The tool also specifies the location for the generated files, in this case, the folder `hdlsrc` under the current working folder.



After you click **Generate**, the coder displays progress information similar to the following in the MATLAB Command Window:

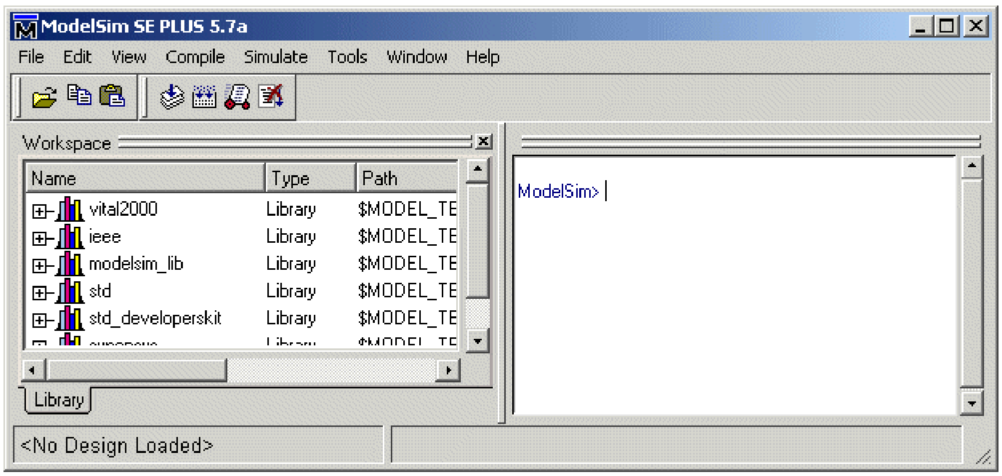
```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\Work\sl_hdlcoder_work\hdlsrc\MyFilter.vhd
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation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

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

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

If you call the generatehdl function from the command-line interface, set code and test bench generation options with property name and value pairs. You can also use the function generatetbstimulus to return the test bench stimulus to a workspace variable.

Starting the Simulator

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



After starting the simulator, set the current folder to the folder that contains your generated HDL files.

Compiling the Generated Filter and Test Bench Files

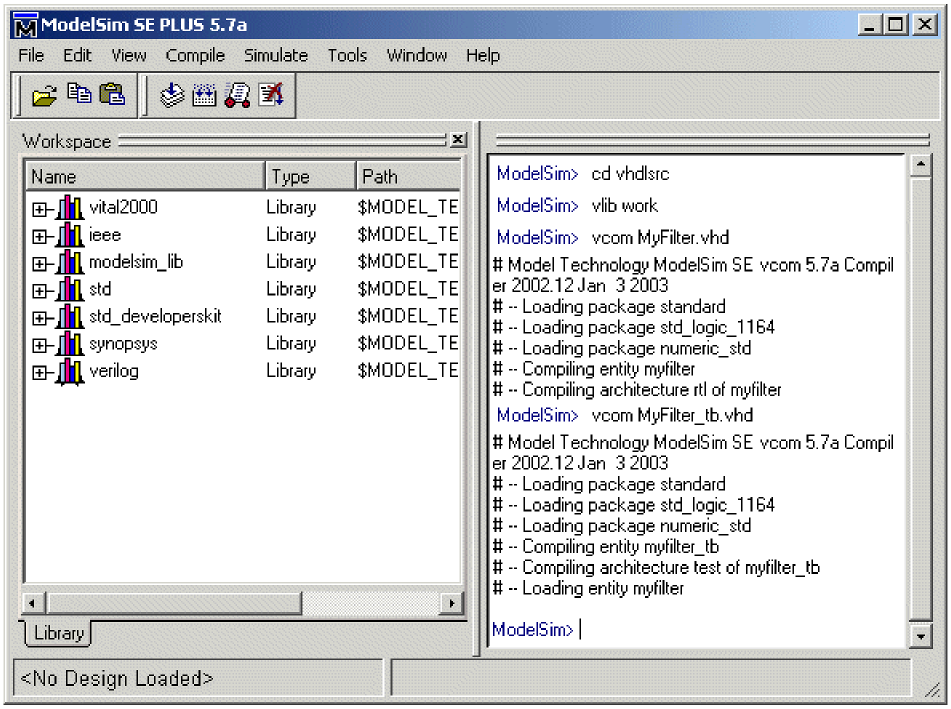
Using your choice of HDL compiler, compile the generated filter and test bench HDL files. Depending on the language of the generated test bench and the simulator you are using, you may have to complete some precompilation setup. For example, in the Mentor Graphics ModelSim simulator, you might choose to create a design library to store compiled VHDL entities, packages, architectures, and configurations.

This Mentor Graphics ModelSim command sequence changes the current folder to hdlsrc, creates the design library work, and compiles VHDL filter and filter test bench code. The vlib command creates the design library work and the vcom commands initiate the compilations.

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

Note For VHDL test bench code that has floating-point (double) realizations, use a compiler that supports VHDL-93 or VHDL-02. For example, in the Mentor Graphics ModelSim simulator, specify the vcom command with the -93 option. Do not compile the generated test bench code with a VHDL-87 compiler. VHDL test benches using double-precision data types do not support VHDL-87. The test bench code uses the image attribute, which is available only in VHDL-93 or higher.

This figure shows this command sequence and informational messages displayed during compilation.

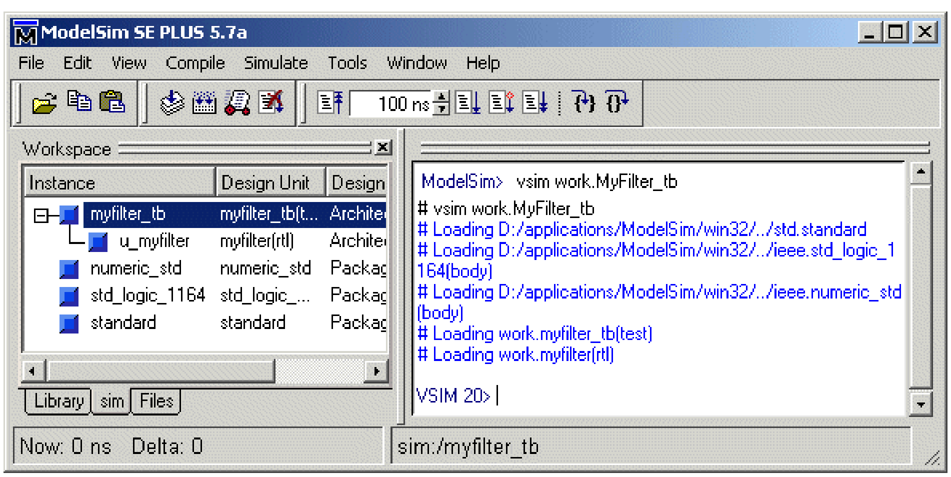


Running the Test Bench Simulation

Once your generated HDL files are compiled, load and run the test bench. The procedure varies depending on the simulator you are using. In the Mentor Graphics ModelSim simulator, you load the test bench for simulation with the `vsim` command. For example:

```
vsim work.MyFilter_tb
```

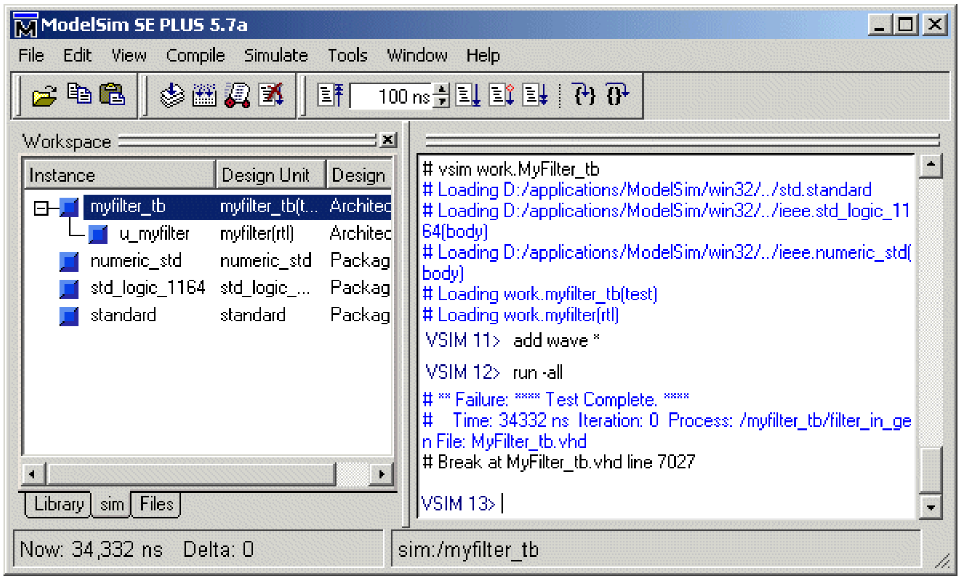
This figure shows the results of loading `work.MyFilter_tb` with the `vsim` command.



Once the design is loaded into the simulator, consider opening a display window for monitoring the simulation as the test bench runs. For example, in the Mentor Graphics ModelSim simulator, you can use the `add wave *` command to open a **wave** window to view the results of the simulation as HDL waveforms.

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

This figure shows the `add wave *` command being used to open a **wave** window and the `-run all` command being used to start a simulation.



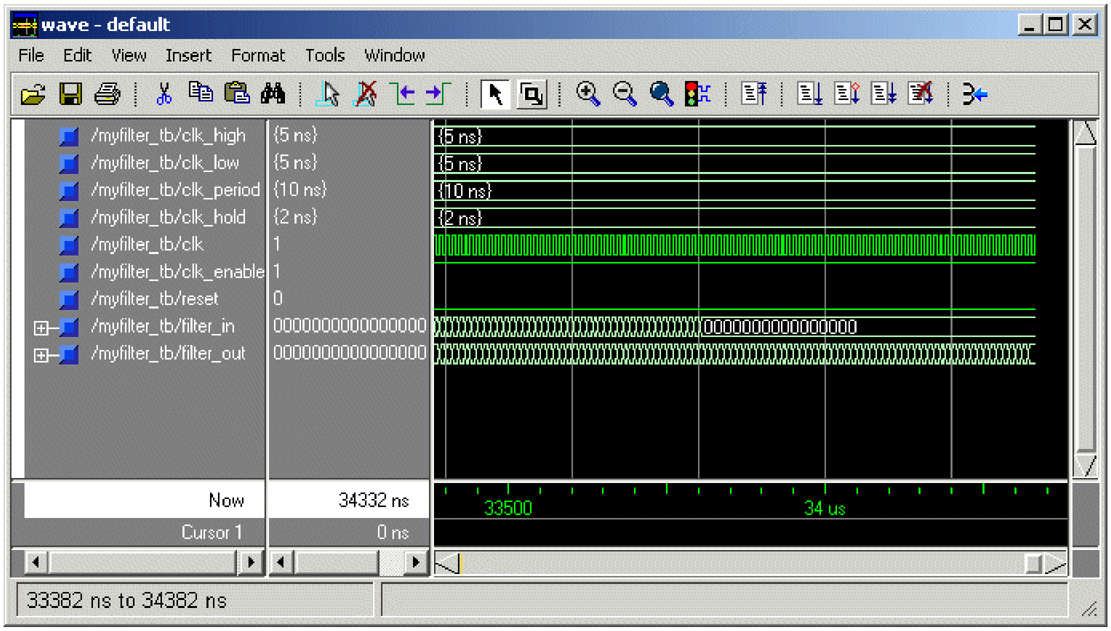
As your test bench simulation runs, watch for error messages. If error messages appear, interpret them as they pertain to your filter design and the code generation options you applied. For example, some HDL optimization options can produce numeric results that differ from the results produced by the original filter object. For HDL test benches, expected and actual results are compared. If they differ (excluding the specified error margin), an error message similar to this is returned:

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

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

Note The failure message that appears in the preceding display is not flagging an error. If the message includes the text `Test Complete`, the test bench has run to completion without encountering an error. The `Failure` part of the message is tied to the mechanism the coder uses to end the simulation.

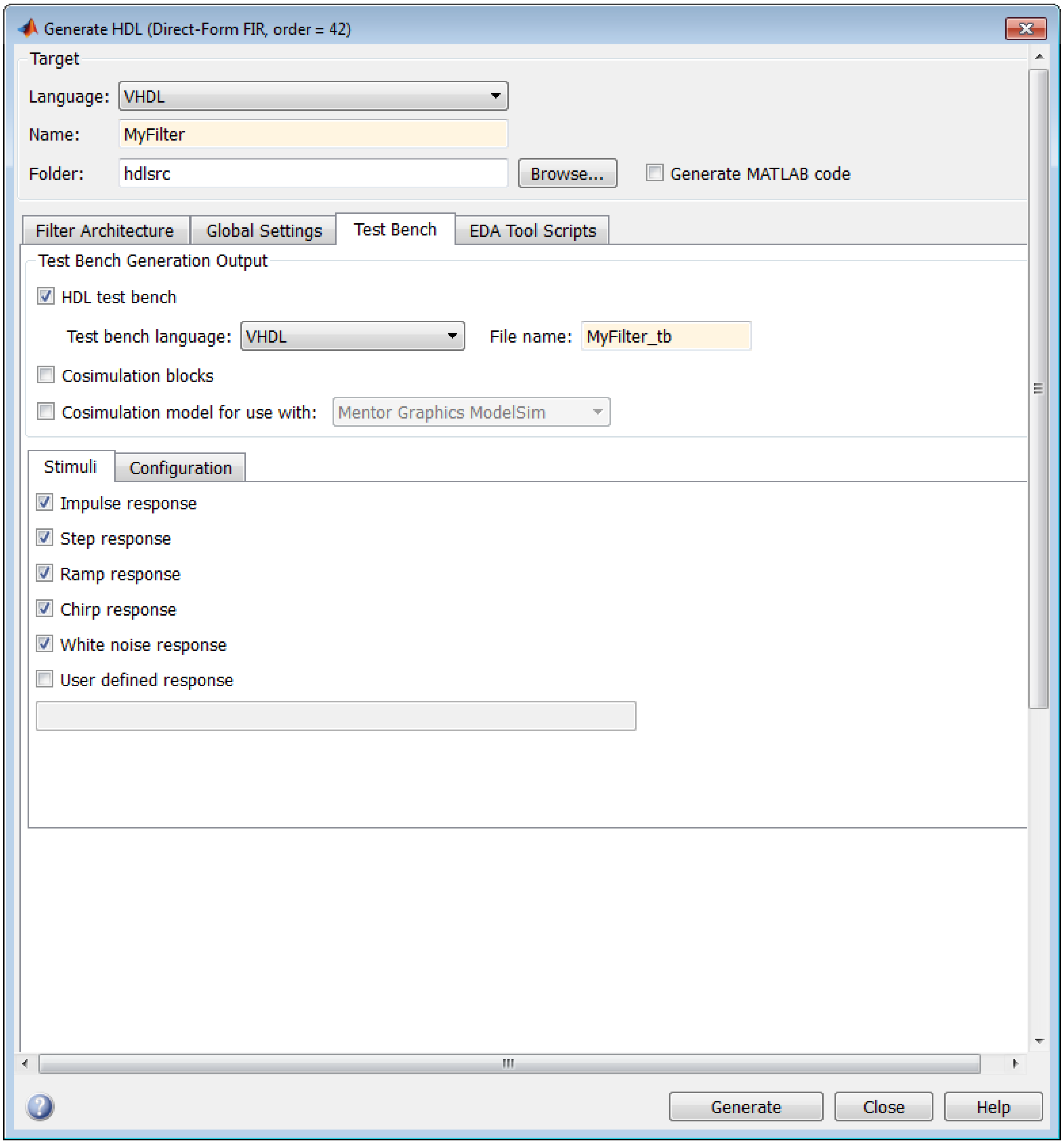
This **wave** window shows the simulation results as HDL waveforms.



Enabling Test Bench Generation

To enable generation of an HDL test bench:

1. Select the **Test Bench** pane in the Generate HDL tool.
2. Select the **HDL test bench** option, as shown in this figure.



3 Click **Generate** to generate HDL and test bench code.

Tip By default, **HDL test bench** is selected.

Command-Line Alternative: Use the `generatehdl` function with the property `GenerateHDLTestBench` to generate an HDL test bench.

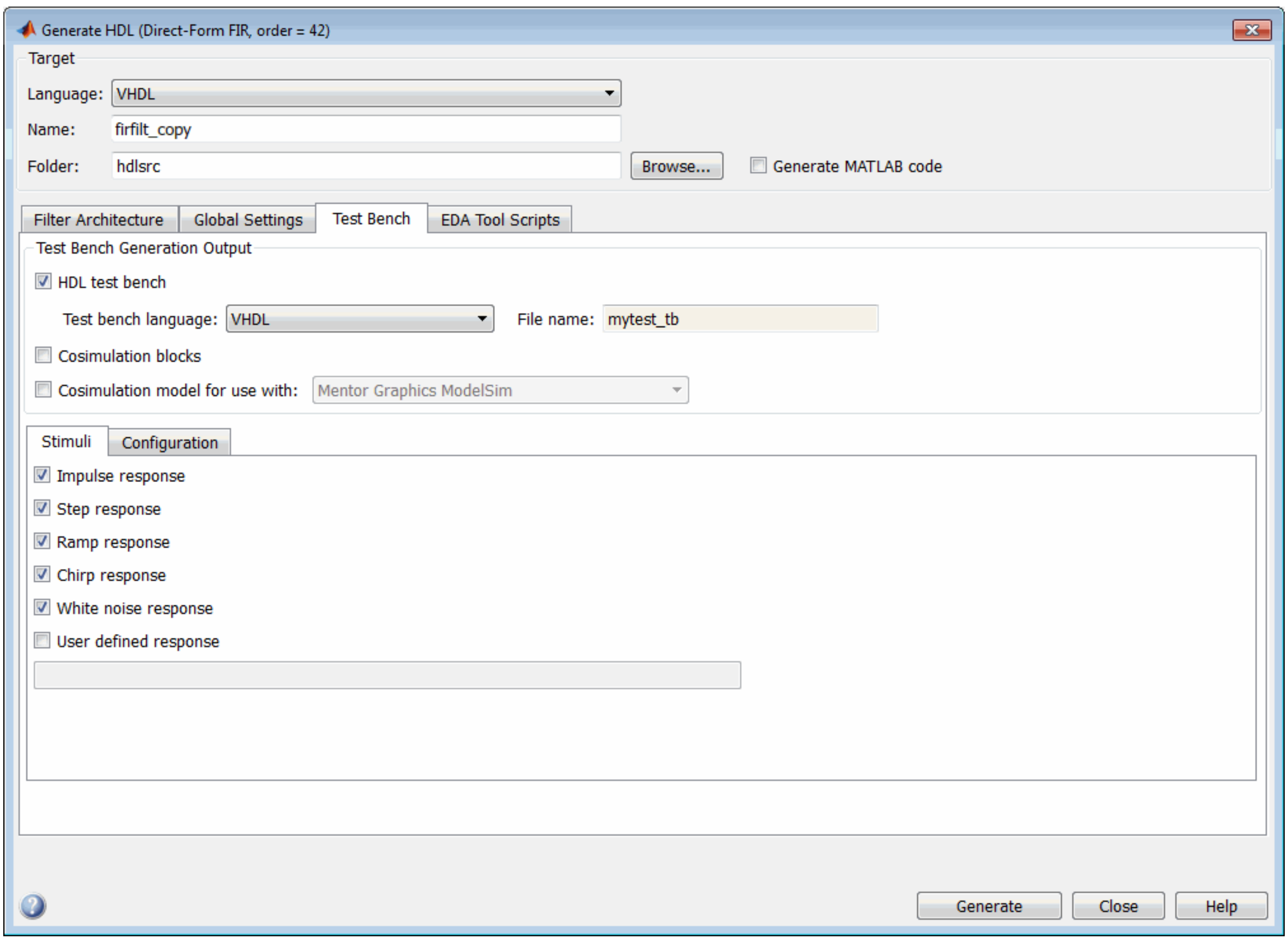
Renaming the Test Bench

The coder derives the name of the test bench file by appending the postfix `_tb` to the name of the quantized filter object. The file type extension depends on the type of test bench that is being generated.

If the Test Bench Is a...	The Extension Is...
Verilog file	Defined by the **Verilog file extension** field in the **General** subpane of the **Global Settings** pane of the Generate HDL tool
VHDL file	Defined by the **VHDL file extension** field in the **Global Settings** pane of the Generate HDL tool

The file is placed in the folder defined by the **Folder** option in the **Target** pane of the Generate HDL tool.

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

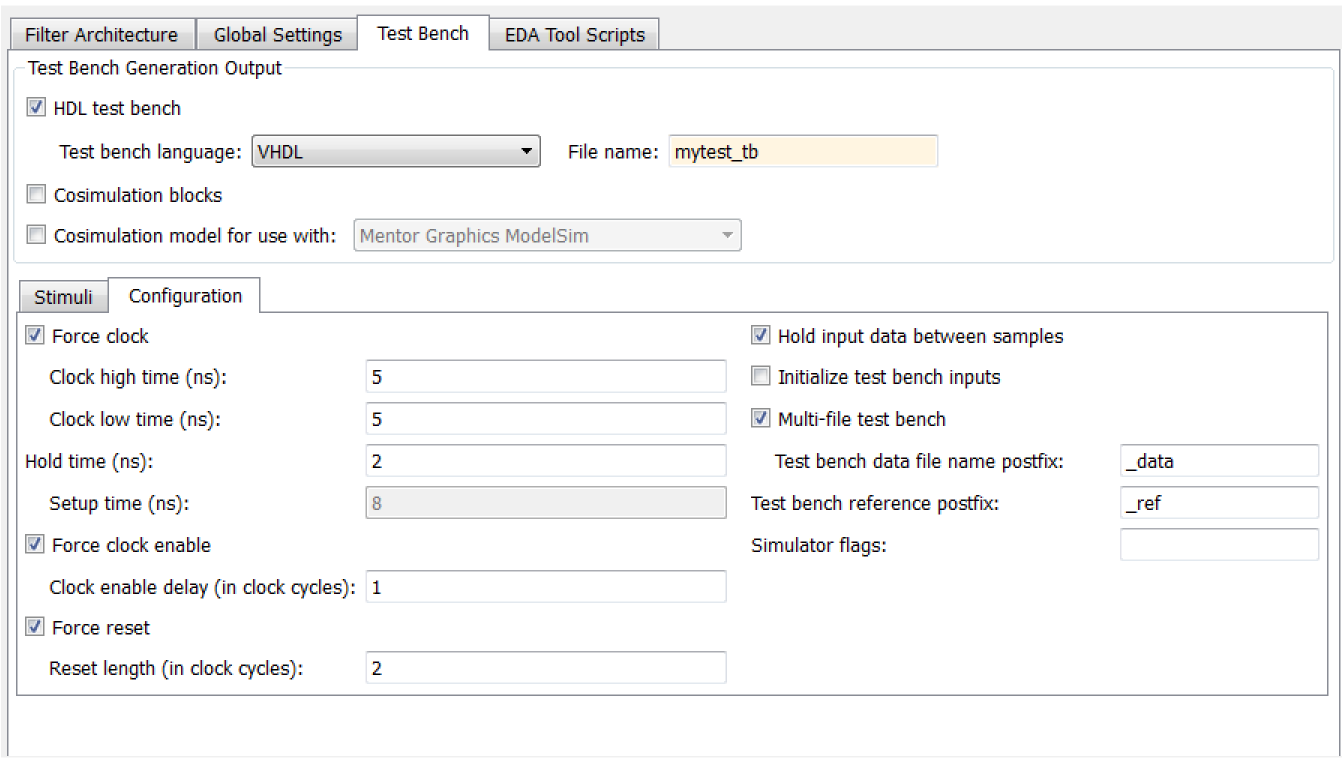


Note If you enter a character vector that is a VHDL or Verilog reserved word, the coder corrects the identifier by appending the reserved word postfix to the character vector.

Command-Line Alternative: Use the `generatehdl` property `TestBenchName` to specify a name for your test bench.

Splitting Test Bench Code and Data into Separate Files

By default, the coder generates a single test bench file, containing test bench helper functions, data, and test bench code. You can split these elements into separate files by selecting the **Multi-file test bench** option in the **Configuration** subpane of the **Test Bench** pane of the Generate HDL tool.



When you select the **Multi-file test bench** option, the **Test bench data file name postfix** option is enabled. The test bench file names are then derived from the name of the test bench and the postfix setting, *`TestBenchName_TestBenchDataPostfix`*.

For example, if the test bench name is `my_fir_filt`, and the target language is VHDL, the default test bench file names are:

- `my_fir_filt_tb.vhd`: test bench code
- `my_fir_filt_tb_pkg.vhd`: helper functions package
- `my_fir_filt_tb_data.vhd`: data package

If the filter name is `my_fir_filt` and the target language is Verilog, the default test bench file names are:

- `my_fir_filt_tb.v`: test bench code
- `my_fir_filt_tb_pkg.v`: helper functions package
- `my_fir_filt_tb_data.v`: test bench data

Command-Line Alternative: Use the `generatehdl` properties `MultifileTestBench`, `TestBenchDataPostfix`, and `TestBenchName` to generate and name separate test bench helper functions, data, and test bench code files.

Configuring the Clock

Based on default settings, the coder configures the clock for a filter test bench such that it:

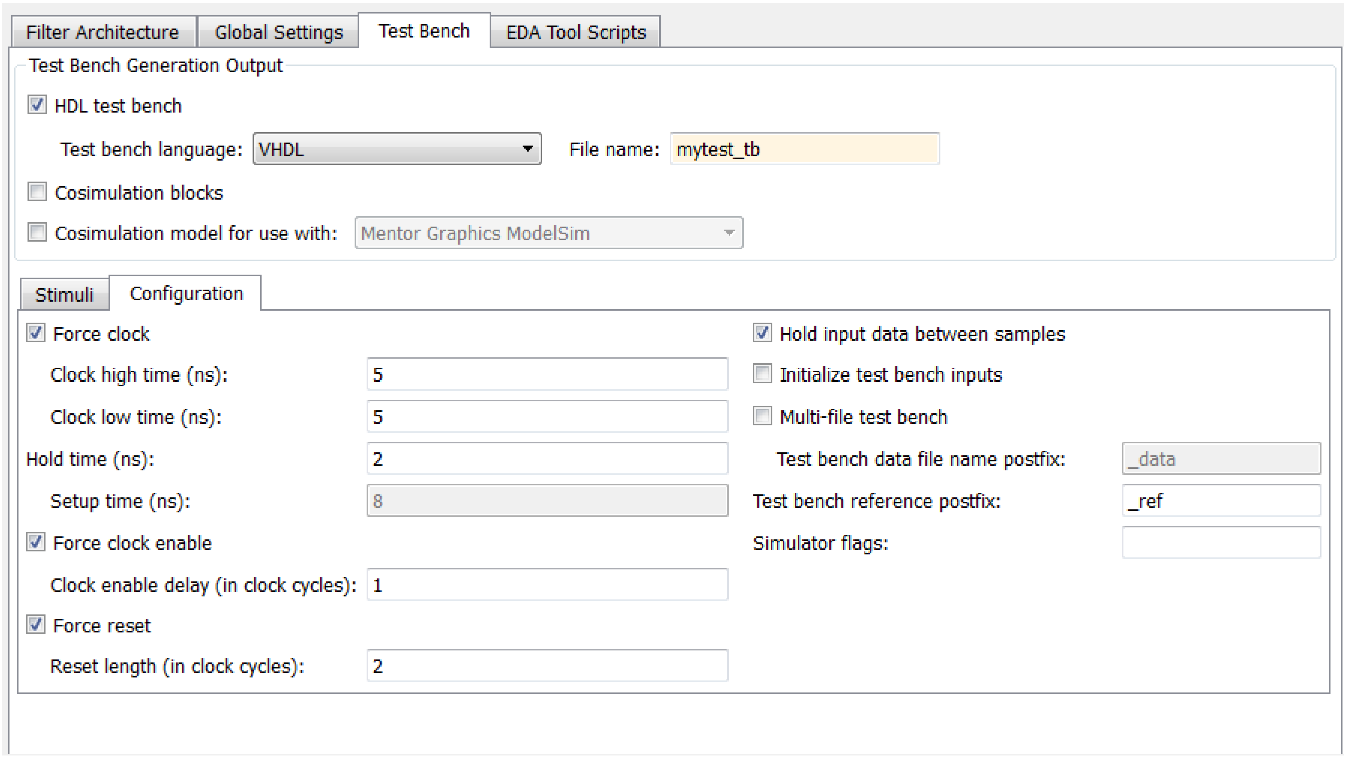
- Forces clock enable input signals to active high (1).
- Asserts the clock enable signal 1 clock cycle after deassertion of the reset signal.
- Forces clock input signals low (0) for a duration of 5 nanoseconds and high (1) for a duration of 5 nanoseconds.

To change these clock configuration settings:

1. Click **Configuration** in the **Test bench** pane of the Generate HDL tool.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. Make the configuration changes shown in the table.

If You Want to...	Then...
Disable the forcing of clock enable input signals	Clear **Force clock enable**.
Disable the forcing of clock input signals	Clear **Force clock**.
Reset the number of nanoseconds that the test bench drives the clock input signals low (0)	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point) in the **Clock low time** field.
Reset the number of nanoseconds that the test bench drives the clock input signals high (1)	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point) in the **Clock high time** field.
Change the delay time elapsed between the deassertion of the reset signal and the assertion of clock enable signal.	Specify a positive integer in the **Clock enable delay** field.

This figure highlights the applicable options.



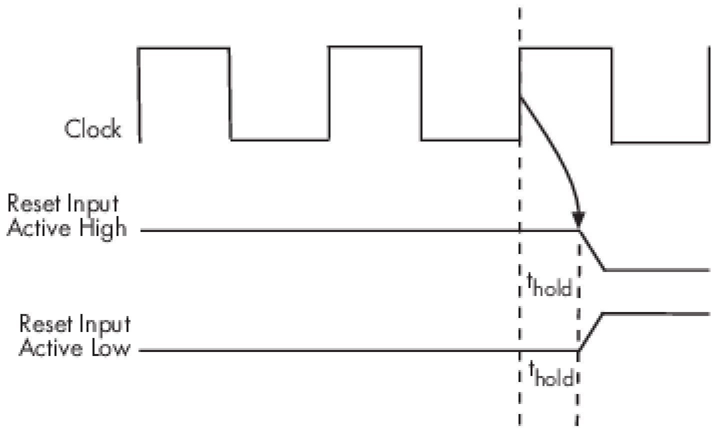
Command-Line Alternative: Use the `generatehdl` properties `ForceClock`, `ClockHighTime`, `ForceClockEnable`, and `TestBenchClockEnableDelay` to reconfigure the test bench clock.

Configuring Resets

Based on default settings, the coder configures the reset for a filter test bench such that it:

- Forces reset input signals to active high (1). (Set the test bench reset input levels with the **Reset asserted level** option).
- Asserts reset input signals for a duration of 2 clock cycles.
- Applies a hold time of 2 nanoseconds for reset input signals.

Hold time is the amount of time the test bench holds the reset input signals past the clock rising edge. The figure shows the application of a hold time (t_{hold}) for reset input signals in the active high and active low cases. The test bench asserts reset after some initial clock cycles defined by the **Reset length** option. The default **Reset length** of 2 clock cycles is shown.

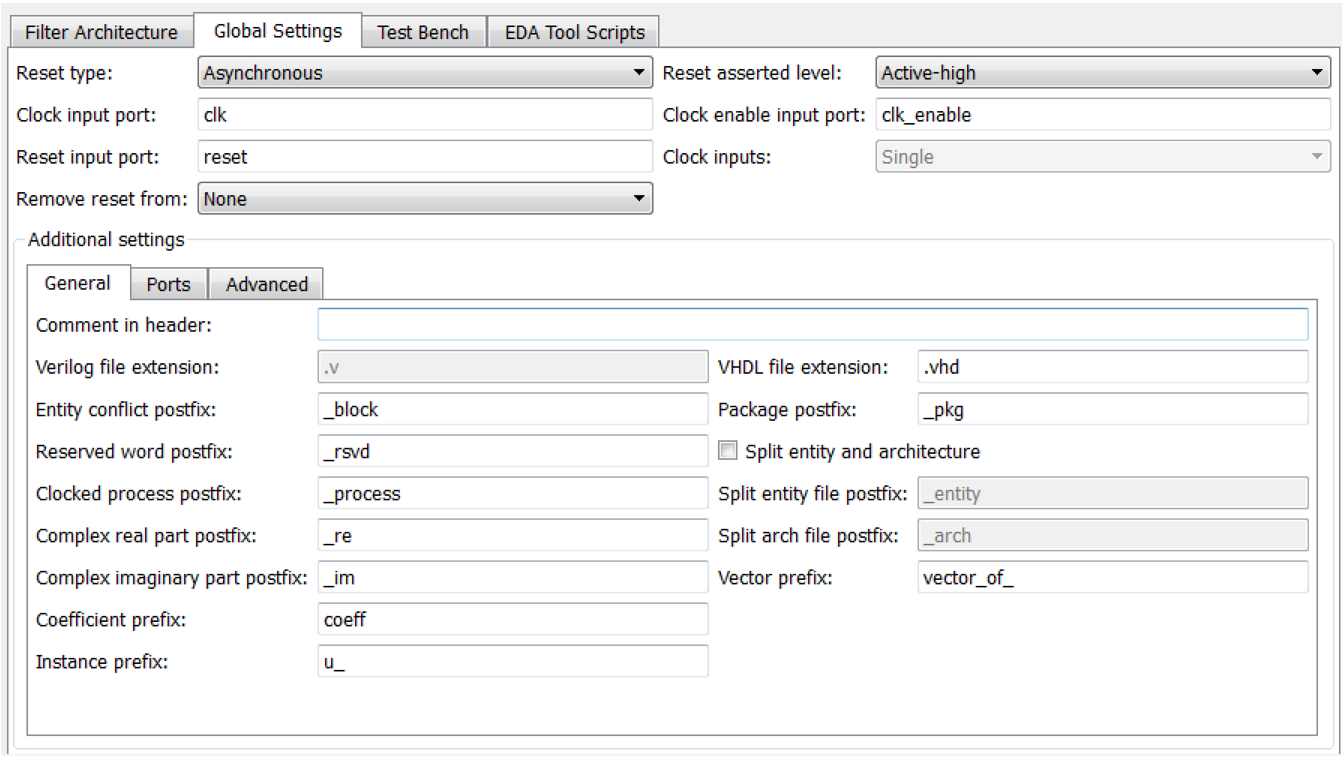


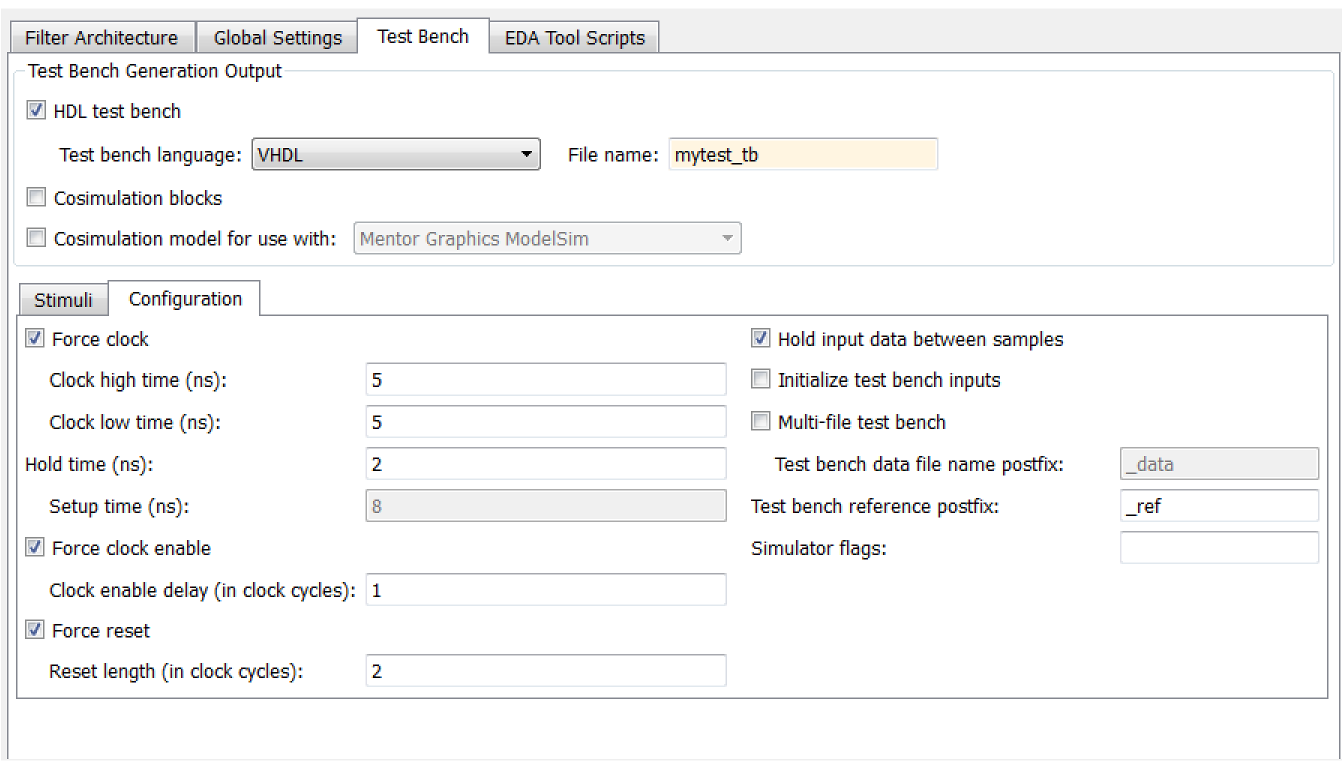
Note The hold time applies to reset input signals only if the forcing of reset input signals is enabled.

The table summarizes the reset configuration settings.

If You Want to...	Then...
Disable the forcing of reset input signals	Clear **Force reset** in the **Test Bench** pane of the Generate HDL tool.
Change the length of time (in clock cycles) during which reset is asserted	Set **Reset length (in clock cycles)** to an integer greater than or equal to 0. This option is located in the **Test Bench** pane of the Generate HDL tool.
Change the reset value to active low (0)	Select `Active-low` from the **Reset asserted level** menu in the **Global Settings** pane of the Generate HDL tool (see "Setting the Asserted Level for the Reset Input Signal" on page 5-19)
Set the hold time	Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point), representing nanoseconds, in the **Hold time** field. When the **Hold time** changes, the **Setup time (ns)** value is updated. The **Setup time (ns)** value computed as `(clock period - HoldTime)` in nanoseconds. These options are in the **Test Bench** pane of the Generate HDL tool.

These figures highlight the applicable options.



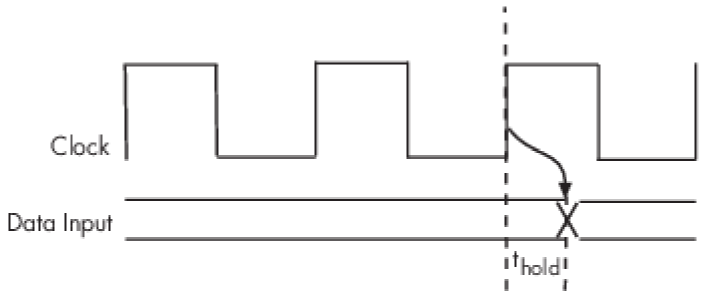


Note The hold time and setup time settings also apply to data input signals.

Command-Line Alternative: Use the `generatehdl` properties `ForceReset`, `ResetLength`, and `HoldTime` to reconfigure test bench resets.

Setting a Hold Time for Data Input Signals

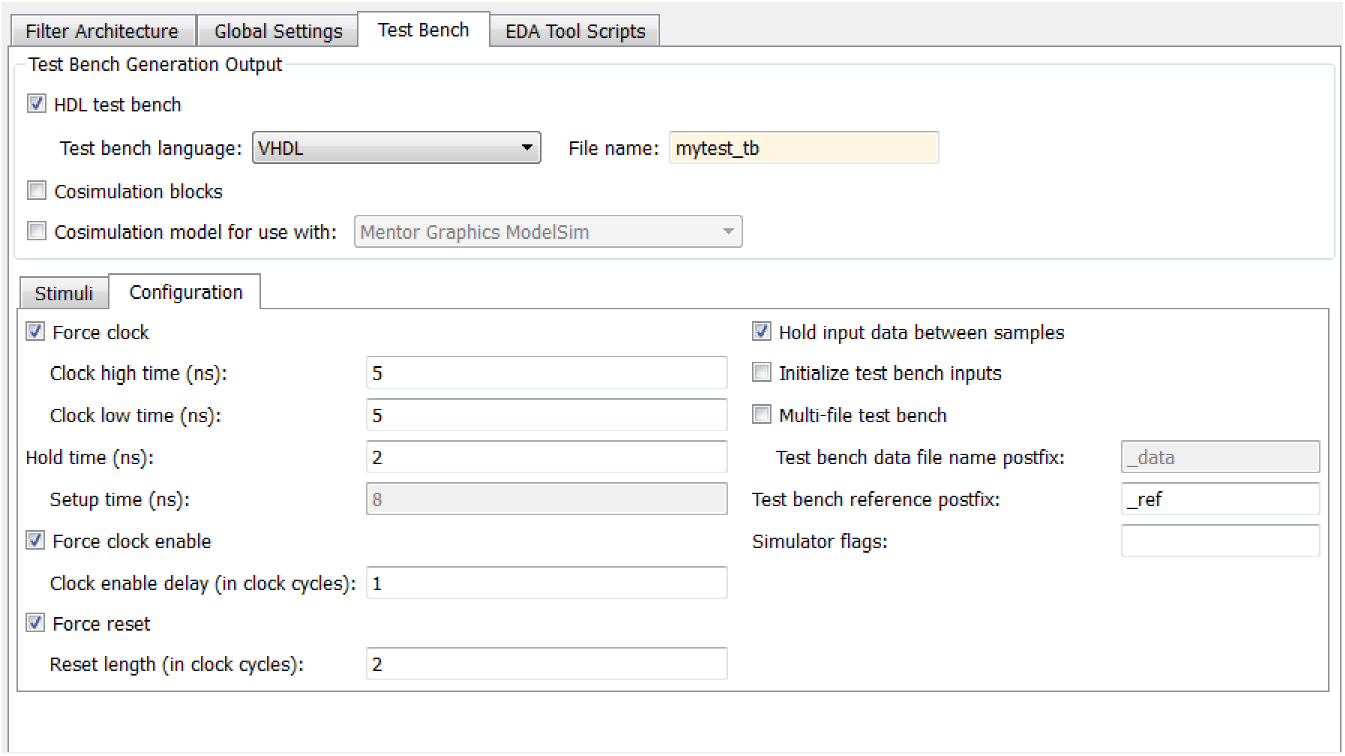
By default, the coder applies a hold time of 2 nanoseconds for filter data input signals. The hold time is the amount of time that data input signals are to be held past the clock rising edge. This figure shows the application of a hold time (t_{hold}) for data input signals.



To change the hold time setting,

1. Click the **Test Bench** tab in the Generate HDL tool.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. Specify a positive integer or double (with a maximum of 6 significant digits after the decimal point), representing nanoseconds, in the **Hold time** field. In this figure, the hold time is set to 2 nanoseconds.

   When the **Hold time** changes, the **Setup time (ns)** value updates. The coder computes the **Setup time (ns)** value as `(clock period - HoldTime)` in nanoseconds. **Setup time (ns)** is a display-only field.



Note When you enable forcing of reset input signals, the hold time and setup time settings also apply to the reset signals.

Command-Line Alternative: Use the `generatehdl` property `HoldTime` to adjust the hold time setting.

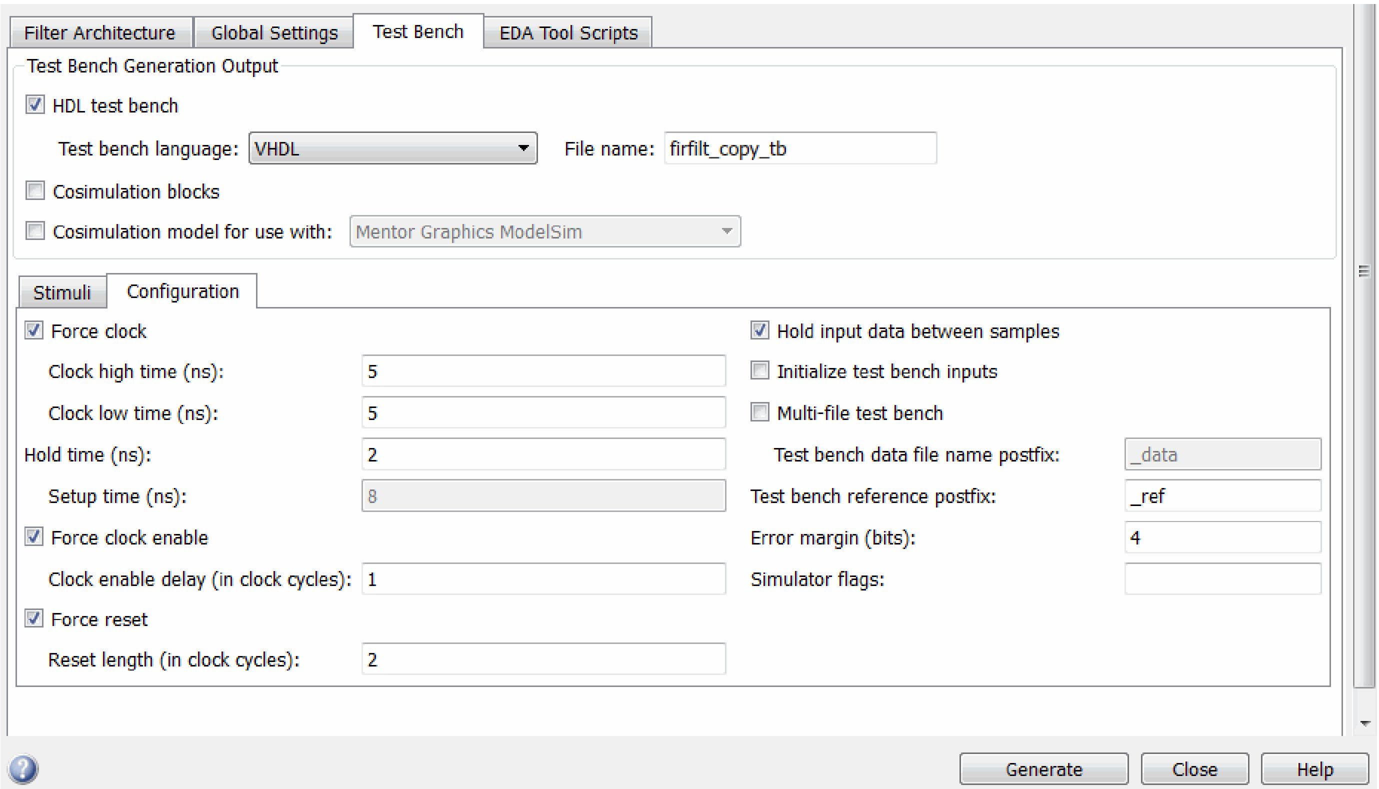
Setting an Error Margin for Optimized Filter Code

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

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

To account for differences in numeric results, consider setting an error margin for the generated test bench. The error margin is the number of least significant bits the test bench ignores when comparing the results. To set an error margin:

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

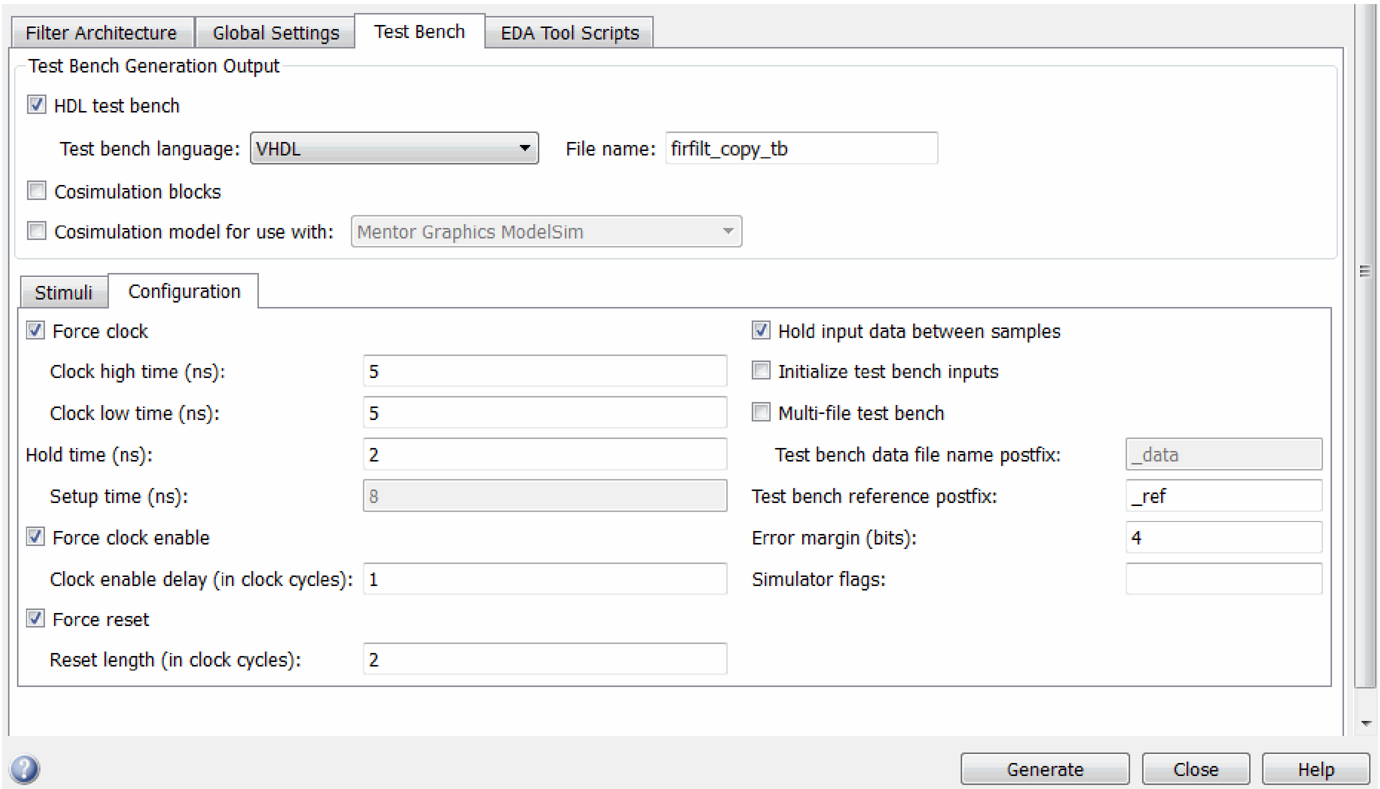


Command-Line Alternative: Use the `generatehdl` property `ErrorMargin` to specify the number of bits of tolerable error.

Setting an Initial Value for Test Bench Inputs

By default, the initial value driven on test bench inputs is `'X'` (unknown). Alternatively, you can specify that the initial value driven on test bench inputs is `0`, as follows:

1. Select the **Test Bench** pane in the Generate HDL tool.
2. Within the **Test Bench** pane, select the **Configuration** subpane.



3 To set an initial test bench input value of `0`, select the **Initialize test bench inputs** option.

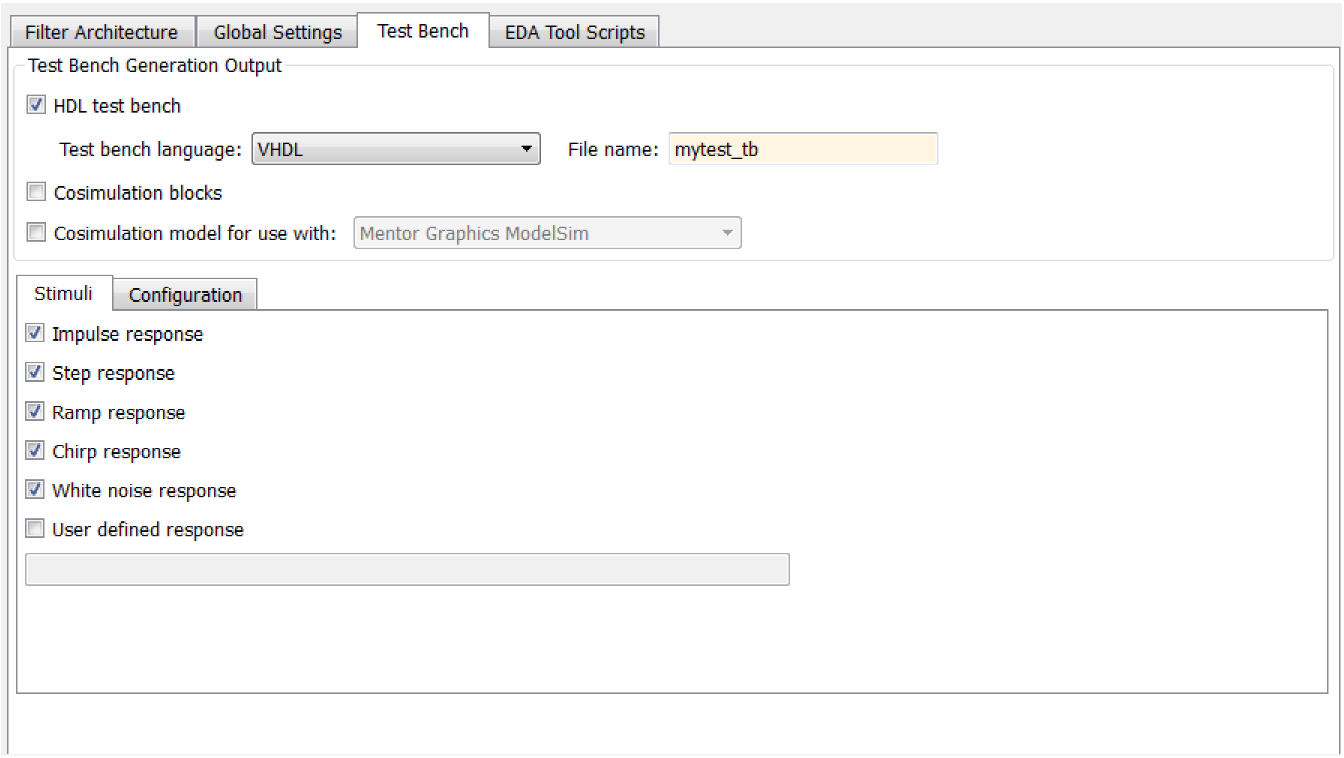
To set an initial test bench input value of `'X'`, clear the **Initialize test bench inputs** option.

Command-Line Alternative: Use the `generatehdl` property `InitializeTestBenchInputs` to set the initial test bench input value.

Setting Test Bench Stimuli

By default, the coder generates a filter test bench that includes stimuli that correspond to the given filter type. However, you can adjust the stimuli settings or specify user-defined stimuli, if desired.

To modify the stimuli included in a test bench, select one or more response types on the **Stimuli** subpane of the **Test bench** tab of the Generate HDL tool. The figure highlights this pane of the tool.



If you select **User defined response**, specify an expression or function that returns a vector of values to be applied to the filter. The values specified in the vector are quantized and scaled based on the quantization settings of the filter.

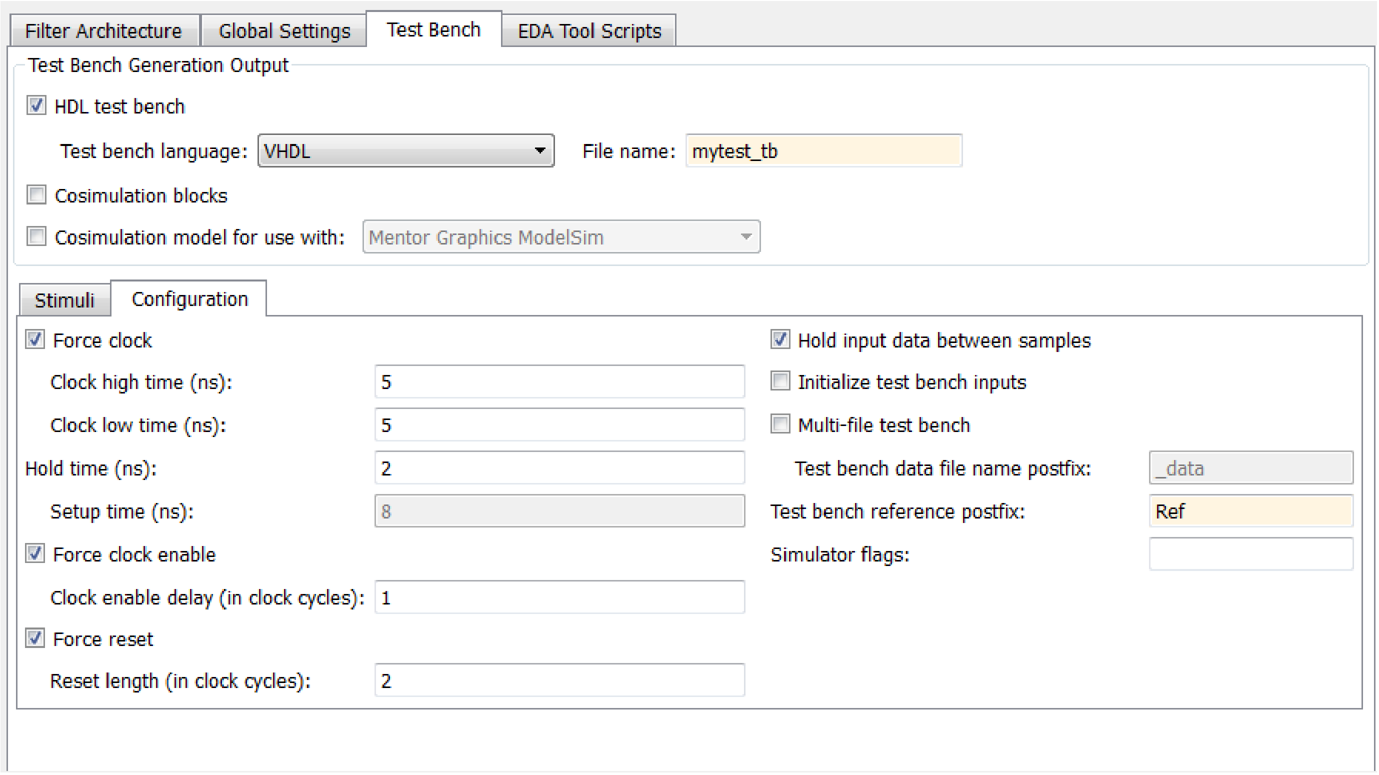
Command-Line Alternative: Use the `generatehdl` properties `TestBenchStimulus` and `TestBenchUserStimulus` to adjust stimuli settings.

Setting a Postfix for Reference Signal Names

Reference signal data is represented as arrays in the generated test bench code. The character vector specified by **Test bench reference postfix** is appended to the generated signal names. The default is `'_ref'`.

You can set the postfix to a value other than `'_ref'`. To change this parameter:

1. Select the **Test Bench** pane in the Generate HDL tool.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. Enter a new character vector in the **Test bench reference postfix** field, as shown in this figure.



Command-Line Alternative: Use the `generatehdl` property `TestBenchReferencePostfix` to change the postfix character vector.

See Also

More About

- "Integration with Third-Party EDA Tools" on page 6-28

Cosimulation of HDL Code with HDL Simulators

In this section...

Generating HDL Cosimulation Blocks for Use with HDL Simulators

The coder supports generation of Simulink HDL Cosimulation blocks. You can use the generated HDL Cosimulation blocks to cosimulate your filter design using Simulink with an HDL simulator. To use this feature, you must have an HDL Verifier™ license.

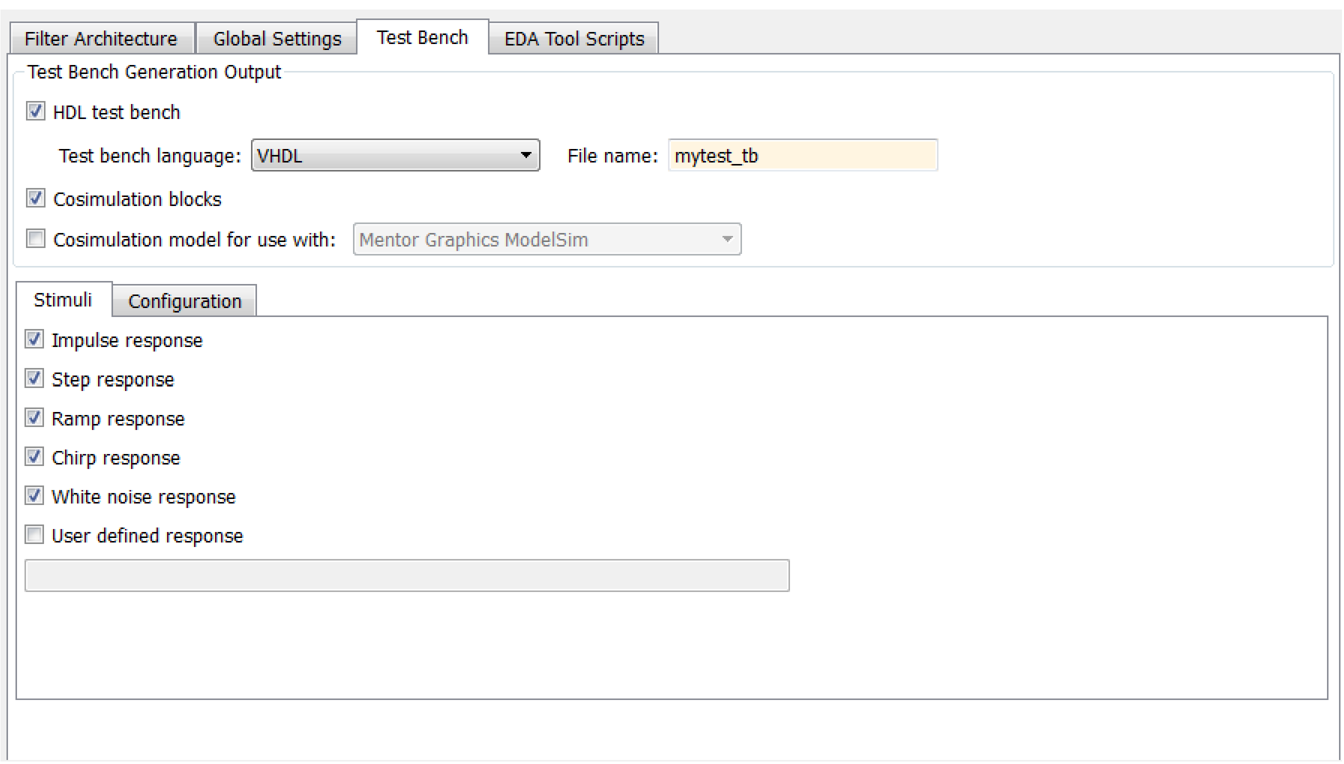
The generated HDL Cosimulation blocks are configured to conform to the port and data type interface of the filter selected for code generation. By connecting an HDL Cosimulation block to a Simulink model in place of the filter, you can cosimulate your design with the desired HDL simulator.

To generate HDL Cosimulation blocks:

1. Select the **Test Bench** pane in the Generate HDL tool.
2. Select the **Cosimulation blocks** option.

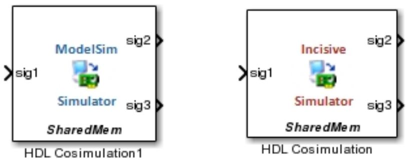
   When this option is selected, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for each supported HDL simulator.
3. If you want to generate HDL Cosimulation blocks only (without generating HDL test bench code), clear **HDL test bench**.

   This figure shows both **HDL test bench** and **Cosimulation blocks** selected.



4 In the Generate HDL tool, click **Generate** to generate HDL and test bench code.

5 In addition to the usual code files, the coder generates a Simulink model containing an HDL Cosimulation block for each HDL simulator supported by HDL Verifier.



6 The generated model is untitled and exists in memory only. Be sure to save it to a destination folder if you want to preserve the model and blocks for use in future sessions.

To configure HDL Cosimulation block parameters, such as timing, latency, and data types, see "Define HDL Cosimulation Block Interface" (HDL Verifier).

Command-Line Alternative: Use the `generatehdl` function with the property `GenerateCosimBlock` to generate HDL Cosimulation blocks.

Generating a Simulink Model for Cosimulation with an HDL Simulator

---

Note To use this feature, you must have an HDL Verifier license.

The coder generates a Simulink model, that runs a Simulink simulation of your filter design, and also a cosimulation of your design with an HDL simulator. The model compares the outputs of the Simulink filter with the results of the HDL simulation.

The generated model includes:

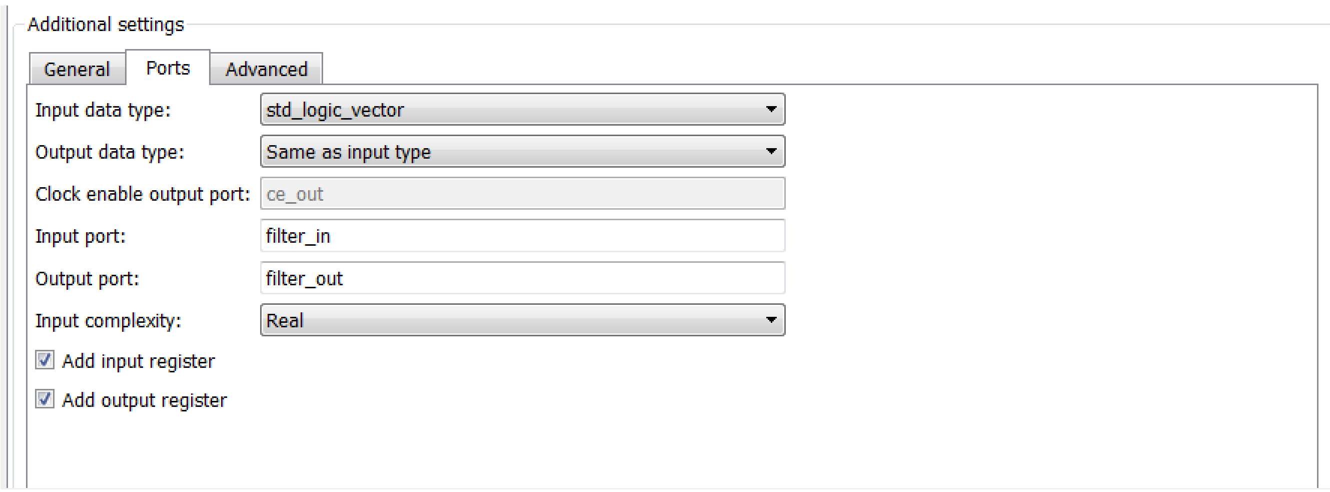
- A behavioral model of the filter design, realized in a Simulink subsystem. The subsystem implements the filter design using basic blocks such as adders and delays.
- A corresponding HDL Cosimulation block. The coder configures this block to cosimulate the filter design using Simulink with either of these HDL simulators.
  - Mentor Graphics ModelSim
  - Cadence Incisive®
- Test input data, calculated from the test bench stimulus you specify. The coder stores the test data in the model workspace variable `inputdata`. A From Workspace block routes test data to the filter subsystem and HDL Cosimulation blocks.
- A Scope block that lets you observe and compare the test input signal with the outputs of the Filter block and the HDL cosimulation. The scope also shows the difference (error) between these two outputs.

Generating the Model

Generation of a cosimulation model requires registered inputs and/or outputs (see "Limitations" on page 6-26). Before generating the model, make sure that your model meets this requirement, as follows:

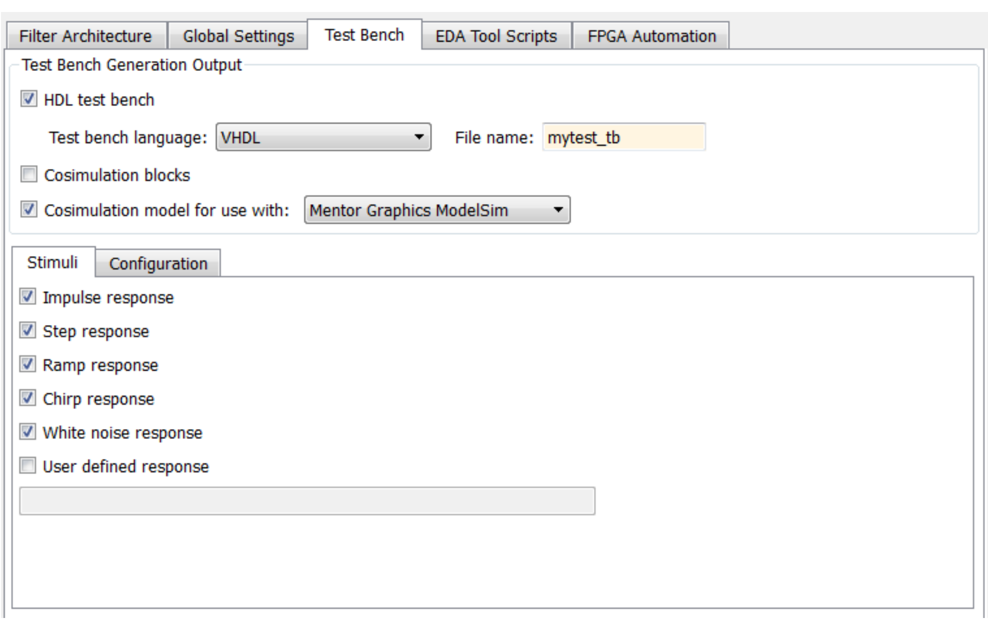
1 Select the **Global Settings** pane the Generate HDL tool.
2 In the **Global Settings** pane, click the **Ports** tab. Port options appear.
3 Select both of these options.

- **Add input register**
- **Add output register**



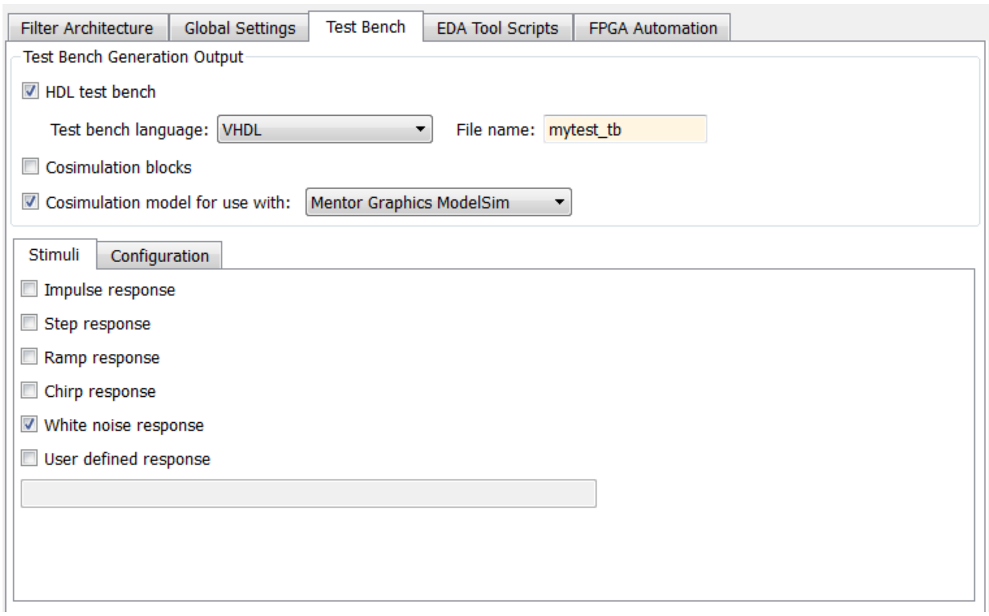
To generate the model:

1 In the Generate HDL tool, configure other code generation and test bench parameters as required by your design.
2 Select the **Test bench** pane of the Generate HDL tool.
3 Select the **Cosimulation model for use with:** option. Selecting this option enables the adjacent drop-down menu, where you can select `Mentor Graphics ModelSim` or `Cadence Incisive`.



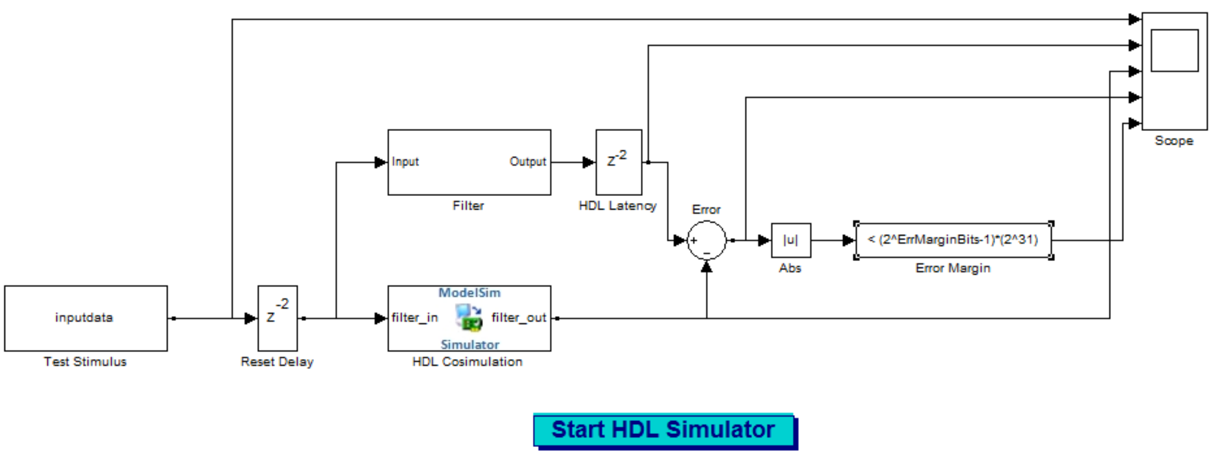
4 Using the drop-down menu, select which type of HDL Cosimulation block you want in the generated model. Select either `Mentor Graphics ModelSim` (the default) or `Cadence Incisive`.

In this figure, the cosimulation model type is `Mentor Graphics ModelSim`, and the stimulus signal is **White noise response**.



5 In the Generate HDL tool, click **Generate** to generate HDL and test bench code.

In addition to the usual code files, the coder generates and opens a Simulink model. This figure shows the model generated from the coder configuration shown in the previous step.



6 The generated model is untitled and exists in memory only. Be sure to save it to a destination folder if you want to preserve the model and blocks for use in future sessions.

To configure HDL Cosimulation block parameters, such as timing, latency, and data types, see “Define HDL Cosimulation Block Interface” (HDL Verifier).

Details of the Generated Model

The generated model contains these blocks.

- `Test Stimulus`: This From Workspace block routes test data in the model workspace variable `inputdata` to the filter subsystem and HDL Cosimulation blocks.
- `Filter`: This subsystem realizes a behavioral model of the filter design.
- `HDL Cosimulation`: This block cosimulates the generated HDL code. The table HDL Cosimulation Block Settings describes how the coder configures the cosimulation block parameters.
- `Reset Delay`: The Tcl commands specified in the `HDL Cosimulation` block apply the reset signal. Reset is high at 0 ns and low at 22 ns (before the third rising clock edge). The Simulink simulation starts feeding the input at 0, 10, 20 ns. The `Reset Delay` block adds a delay such that the first sample is available to the RTL simulation when it is ready after the reset is applied.
- `HDL Latency`: This delay represents the difference between the latency of the RTL simulation and the Simulink behavioral block.
- `Error`: Computes the difference between the outputs of the `Filter` block and the `HDL Cosimulation` block.
- `Abs`: Absolute value of the error computation.
- `Error margin:`: Indicator comparing the absolute value of the error with the test bench error margin value (see “Setting an Error Margin for Optimized Filter Code” on page 6-16).
- `Scope`: Displays the input signal, outputs from the `Filter` block and the `HDL Cosimulation` blocks, and the difference (if one exists) between the two.
- `Start HDL Simulator` button: Starts your HDL cosimulation software.

HDL Cosimulation Block Settings

Pane	Settings
`Ports`	Port names: same as the names in the generated code for the filter. Input/Output data types: `Inherit` Input sample time: `Inherit` Output sample time: Same as Simulink fixed step size.
`Clocks`	Clock port name: same as the name in the generated code for the filter. Active clock edge: `Rising` Period: same as the Simulink sample time.
`Timescales`	1 second in Simulink corresponds to 1 tick in the HDL simulator
`Connection`	Connection Mode: `Full Simulation` Connection Method: `Shared memory`

`Tcl` (Pre-simulation commands)	`force /Hlp/clk_enable 1;`
	`force /Hlp/reset 1 0 ns, 0 22 ns;`
	`puts ----------------------------------------`
	`puts "Running Simulink Cosimulation block.";`
	`puts [clock format [clock seconds]]`

`Tcl` (Post-simulation commands)	`force /Hlp/reset 1` `puts [clock format [clock seconds]]`

Generated Model Settings

The generated model has these nondefault settings:

- **Solver**: `Discrete (no continuous states)`.
- Solver **Type**: `Fixed-step`.
- **Stop Time**: `Ts * StimLen`, where `Ts` is the Simulink sample time and `StimLen` is the stimulus length.
- **Sample Time Colors**: enabled
- **Port Data Types**: enabled
- **Hardware Implementation**: ASIC/FPGA

Limitations

- A cosimulation that runs without encountering errors requires that outputs from the generated HDL code are synchronous with the clock. Before generating code, make sure that both of these options are selected:

  - **Add input register**
  - **Add output register**

  If you do not select either of these options, the coder terminates model generation with an error. However, test bench code generation is completed.

- The coder does not support generation of a cosimulation model when the target language is Verilog and data of type double is generated.

Command-Line Alternative

Use the `generatehdl` function, passing in one of these values for the property `GenerateCosimModel`.

- ```
  generatehdl(filtSysObj,'InputDataType',numerictype(1,16,15), ...
      'GenerateCosimModel','Incisive');
  ```
- ```
  generatehdl(filtSysObj,'InputDataType',numerictype(1,16,15), ...
      'GenerateCosimModel','ModelSim');
  ```

Integration with Third-Party EDA Tools

In this section...

Generate a Default Script

The coder generates scripts as part of the code and test bench generation process. Script files are generated in the target folder.

When HDL code is generated for a filter, `filt`, the coder writes these script files.

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

When test bench code is generated for a filter `filt`, the coder writes these script files.

- `filt_tb_compile.do`: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
- `filt_tb_sim.do`: Mentor Graphics ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.

You can enable or disable script generation and customize the names and content of generated script files by:

- Passing properties as `'Name',Value` arguments to the `generatehdl` function. See Compilation and Simulation.
- Setting the corresponding options in the Generate HDL tool. Select the **EDA Tool Scripts** tab, and click **Compilation script** or **Simulation script** from the menu in the left column. See “Customize Scripts for Compilation and Simulation” on page 6-29.

Structure of Generated Script Files

A generated EDA script consists of three sections, which are generated and executed in this order.

1. An initialization (`Init`) phase. The `Init` phase performs required setup actions, such as creating a design library or a project file.
2. A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file.
3. A termination phase (`Term`). This phase 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 coder generates scripts by passing format character vectors to the `fprintf` function. Using the UI options (or `generatehdl` properties) summarized in these sections, you can pass in customized format character vectors to the script generator. Some of these format character vectors take arguments, such as the top-level entity or module name.

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

Customize Scripts for Compilation and Simulation

To view and set options for generated scripts:

1. Open the Generate HDL tool.
2. Click the **EDA Tool Scripts** tab.

   The **Compilation script** options group is selected, as shown.
3. The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected, as shown in the preceding image.

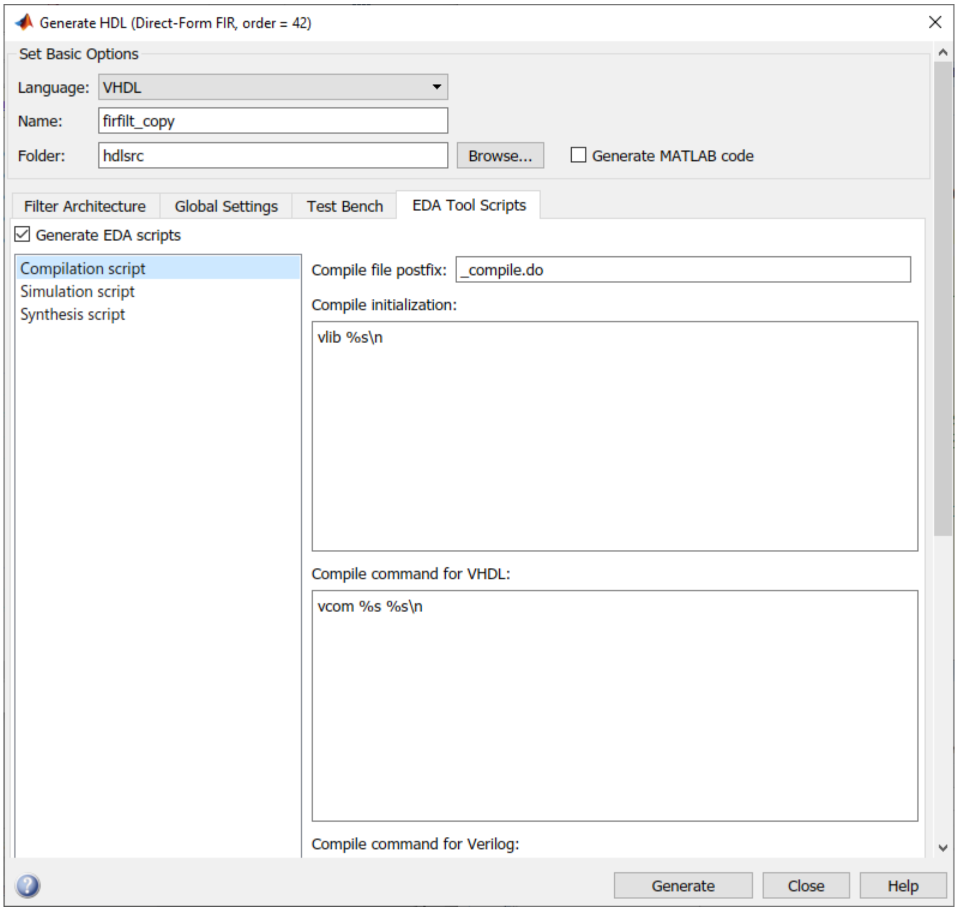
   If you want to disable script generation, clear this check box.
4. The list on the left of the tool lets you select from several categories. Select a category and set the options as desired. The categories are:

   - **Compilation script**: customize scripts for compilation of generated VHDL or Verilog code. See "Compilation Script Options" on page 6-29.
   - **Simulation script**: customize scripts for HDL simulators. See "Simulation Script Options" on page 6-31 .
   - **Synthesis script**: customizing scripts for synthesis tools. See "Automation Scripts for Third-Party Synthesis Tools" on page 7-2 .
5. The custom character vectors for each section are passed to `fprintf` to write each section of the selected script. You can use format character vectors supported by the `fprintf` function. Some of the character vectors include implicit arguments.

Option	Implicit arguments
Compile initialization	Library name
Compile command for VHDL and **Compile command for Verilog**	• Contents of the **Simulator flags** option (an empty character vector, `''`, by default) • File name of the current module
Compile termination	No implicit argument
Compile initialization	No implicit argument
Simulation command	• Library name • Top-level module or entity name
Simulation termination	No implicit argument

Compilation Script Options

The figure shows the **Compilation script** pane, with the options set to their default values.



The coder generates a script called `firfilt_copy_compile.do`:

```
vlib work
vcom  firfilt_copy.vhd
```

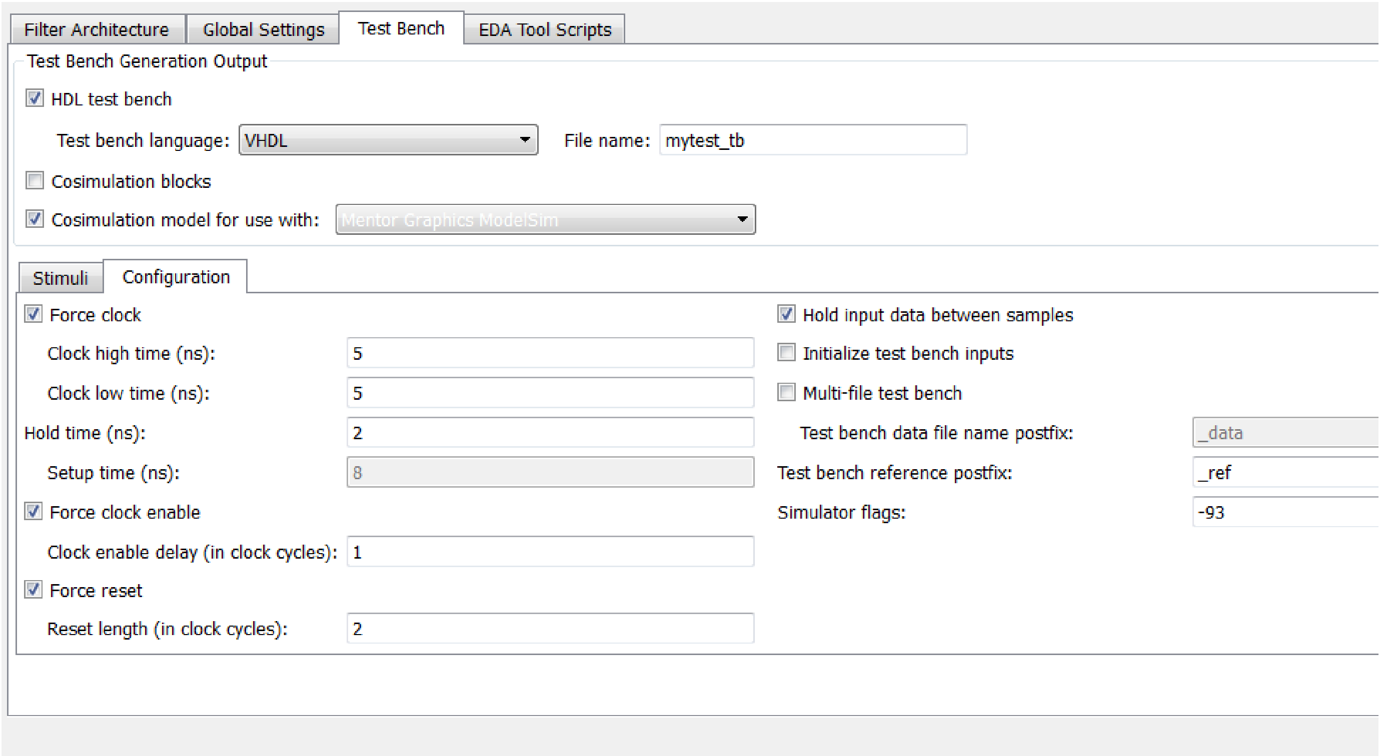
If you generate a test bench for your filter, the coder also generates a script called `firfilt_copy_tb_compile.do`

```
vlib work
vcom  firfilt_copy.vhd
vcom  firfilt_copy_tb.vhd
```

Setting Simulator Flags for Compilation Scripts

You have the option of inserting simulator flags into your generated compilation scripts. This option is included in the compilation scripts for both the standalone filter and the test bench. For example, you can specify a compiler version. To specify the flags:

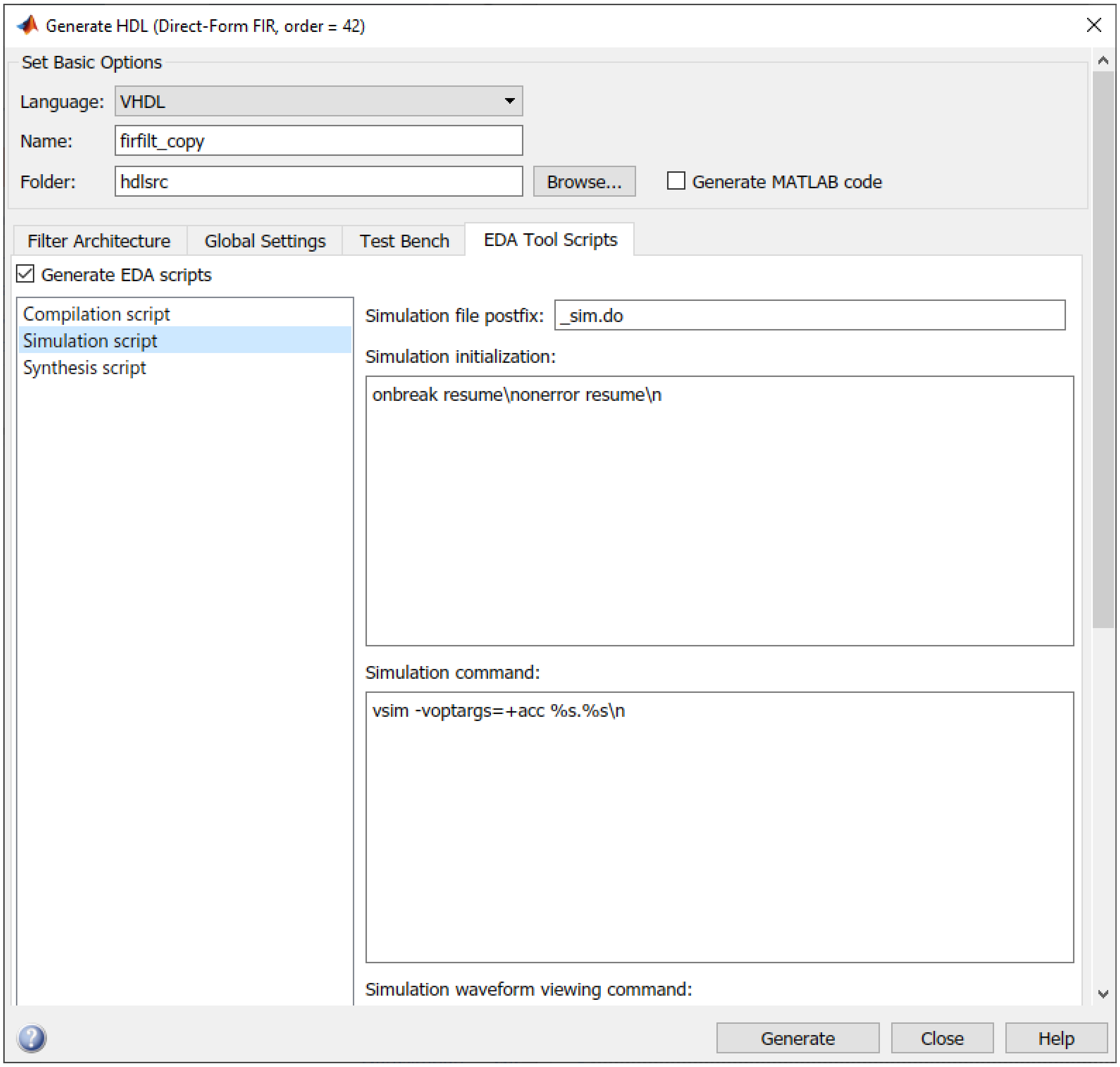
1. Click **Test Bench** in the Generate HDL tool.
2. Type the flags of interest in the **Simulator flags** field. In the figure, the tool specifies that the Mentor Graphics ModelSim simulator use the `-93` compiler option for compilation.



Command-Line Alternative: Specify simulator flags with the `SimulatorFlags` property of the `generatehdl` function.

Simulation Script Options

The coder generates a simulation script when you generate a test bench. The figure shows the **Simulation script** pane, with the options set to their default values.



The coder generates a script called `firfilt_copy_tb_sim.do`:

```
onbreak resume
onerror resume
vsim -voptargs=+acc work.firfilt_copy_tb
add wave sim:/firfilt_copy_tb/u_firfilt_copy/clk
add wave sim:/firfilt_copy_tb/u_firfilt_copy/clk_enable
add wave sim:/firfilt_copy_tb/u_firfilt_copy/reset
add wave sim:/firfilt_copy_tb/u_firfilt_copy/filter_in
add wave sim:/firfilt_copy_tb/u_firfilt_copy/filter_out
add wave sim:/firfilt_copy_tb/filter_out_ref
run -all
```

Synthesis Script Options

For information about synthesis script options, see “Automation Scripts for Third-Party Synthesis Tools” on page 7-2.

7

Synthesis and Workflow Automation

Automation Scripts for Third-Party Synthesis Tools

In this section...

Select a Synthesis Tool

You can enable or disable generation of synthesis scripts, and select the synthesis tool for which the coder generates scripts. To do so, in the Generate HDL tool, select the **EDA Tool Scripts** tab. Then select **Synthesis script** from the menu on the left side, and select your synthesis tool from the **Choose synthesis tool** drop-down menu.

Supported Synthesis Tools
`Xilinx ISE`
`Xilinx Vivado`
`Microsemi Libero`
`Mentor Graphics Precision`
`Altera Quartus II`
`Synopsis Synplify Pro`

When you select a synthesis tool, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to correspond with the tool you selected.
- Fills in the **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** fields with default Tcl script code for the tool.

If you select `None`, the coder does not generate a synthesis script. The coder clears and disables the fields in the **Synthesis script** pane.

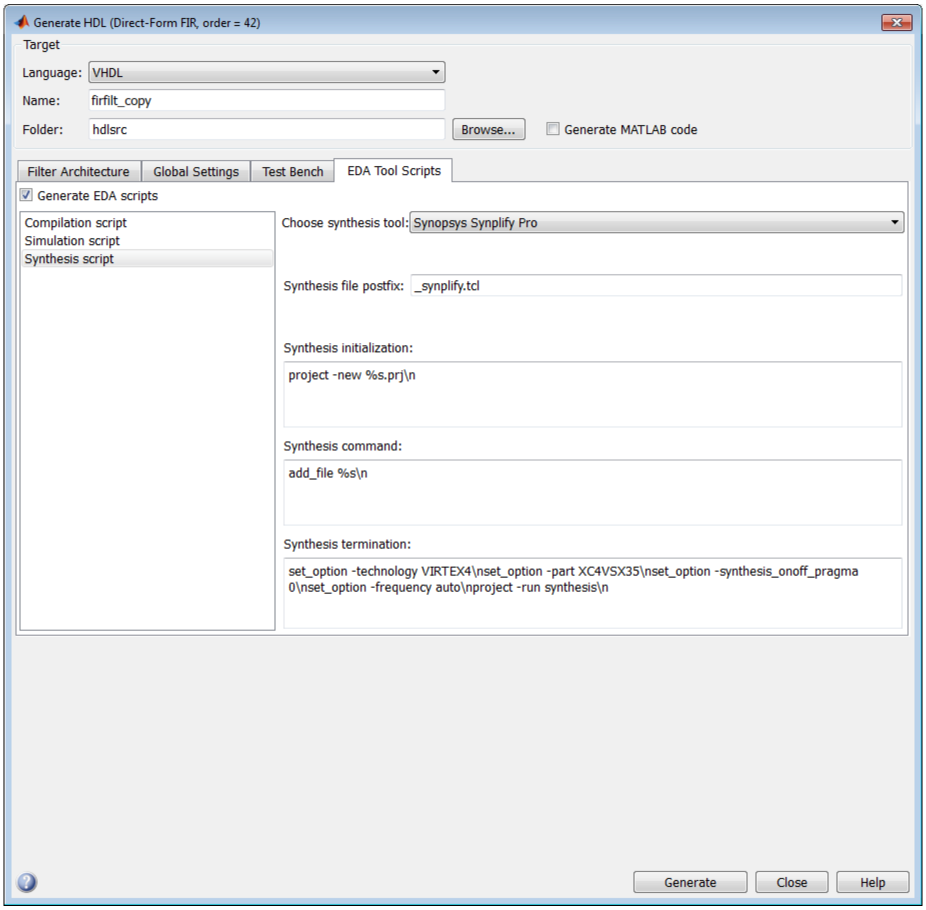
You can also select `'Custom'`, and set the **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** Tcl code fields to generate a script that supports your tool.

Customize Synthesis Script Generation

You can customize the script according to your target device, constraints, etc., by modifying the Tcl code in the **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** fields. To see these options in the Generate HDL tool, select the **EDA Tool Scripts** tab, and click **Synthesis script** from the menu in the left column.

The coder prints the three sections of the script in the order shown in the tool. The script file is named according to the name of your module or entity combined with the text in **Synthesis file postfix**. The custom character vectors for each section are passed to `fprintf` to write each section of the synthesis script. You can use format character vectors supported by the `fprintf` function. In **Synthesis initialization**, you can use an implicit argument that is the name of your top-level module or entity. In **Synthesis command**, you can use an implicit argument that is the name of the file that contains your generated HDL code.

The figure shows the **Synthesis script** pane, with the options set to their default values.



The coder generates a script called `firfilt_copy_synplify.tcl`:

```
project -new firfilt_copy.prj
add_file firfilt_copy.vhd
set_option -technology VIRTEX4
set_option -part XC4VSX35
set_option -synthesis_onoff_pragma 0
set_option -frequency auto
project -run synthesis
```

Programmatic Synthesis Automation

You can also specify the synthesis tool and script options as `'Name',Value` arguments to the `generatehdl` function. For programmatic use with `generatehdl`, see Synthesis and Workflow Automation Properties.

8

Filter Design HDL Coder Featured Examples

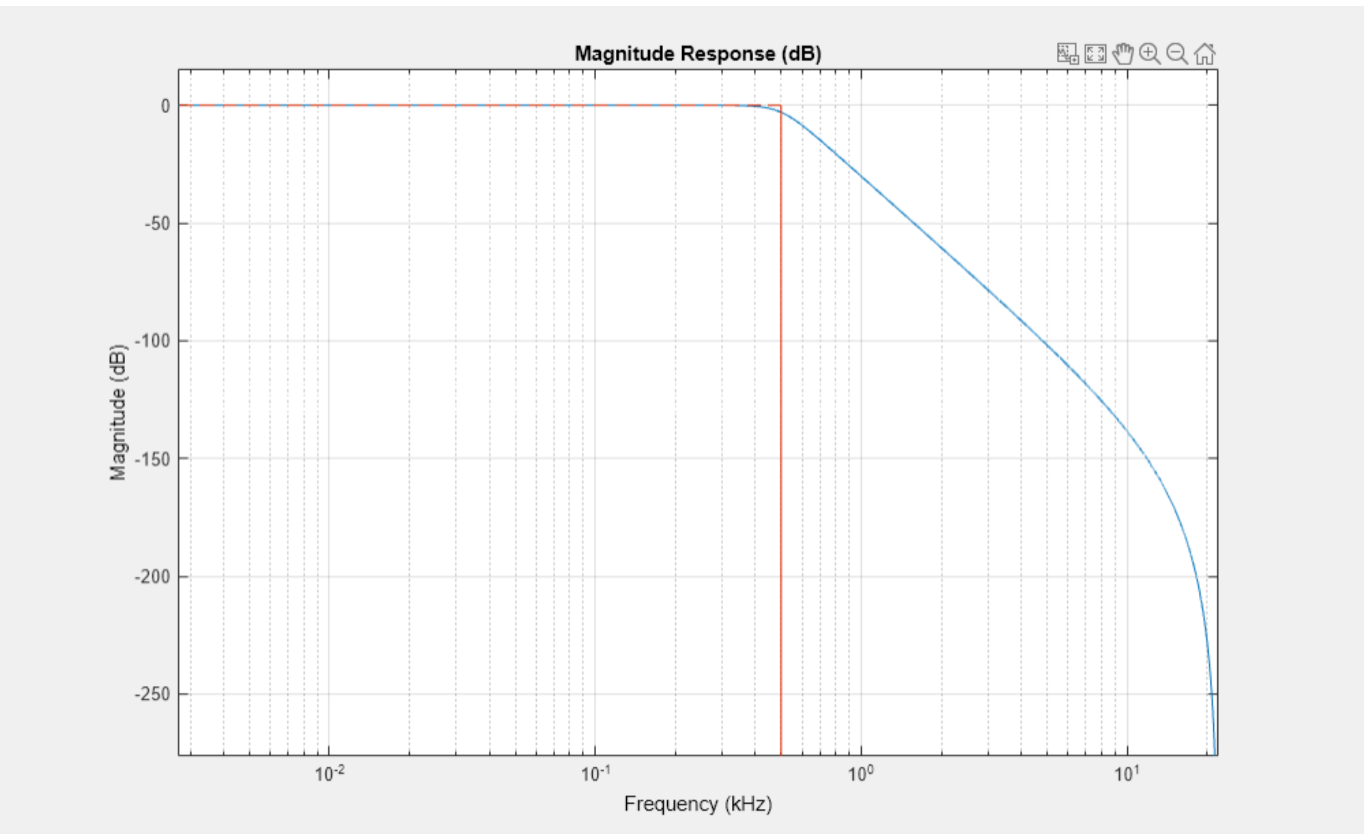
HDL Butterworth Filter

This example illustrates how to generate HDL code for a 5th order Butterworth filter. The cutoff-frequency for this filter is very low relative to the sample rate, leading to a filter that is difficult to make practical. Also, because the filter has small input (8-bit) and output (9-bit) word sizes, the quantized filter requires scaling to be realizable.

Design the Filter

Use the CD sampling rate of 44.1 kHz and a cut-off frequency of 500 Hz. First, create the filter design object, then create a biquad filter System object™. Finally, use `fvtool` to examine the response in log frequency.

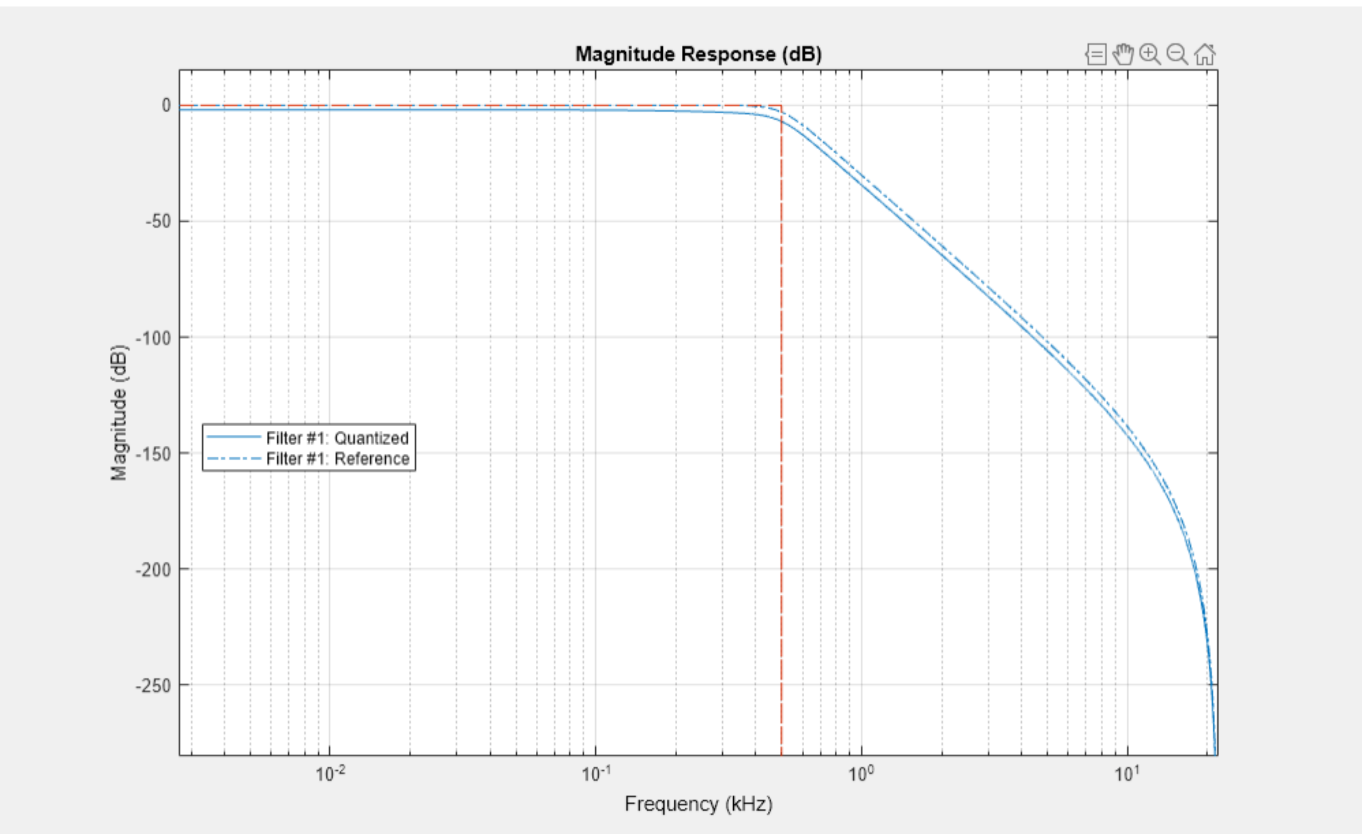
```
Fs = 44100;
F3db = 500;
filtdes = fdesign.lowpass('n,f3db', 5, F3db, Fs);
butterFilter = design(filtdes,'butter',...
    'SystemObject',true,'FilterStructure','df1sos','UseLegacyBiquadFilter',true);

fvtool(butterFilter,'Fs',Fs,'FrequencyScale','log');
```



Create the Quantized Filter

Apply the fixed point settings to the filter object. This example uses 9-bit fixed-point output data with 12-bit coefficients, 20-bit states, full-precision products, and 32-bit adders. Check the response by using `fvtool`.

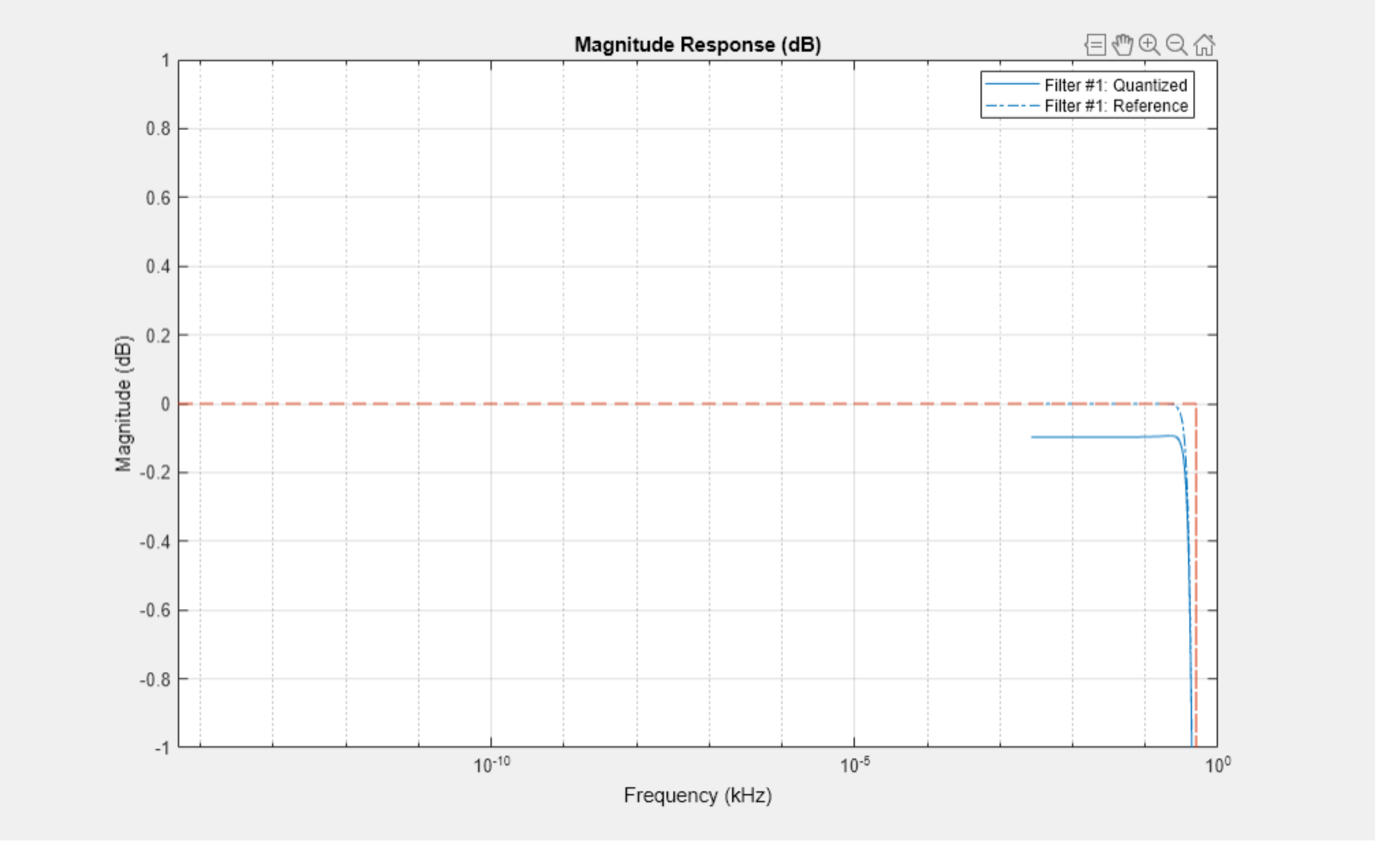
```
butterFilter.NumeratorCoefficientsDataType = 'Custom';
butterFilter.CustomNumeratorCoefficientsDataType = numerictype([],12);
butterFilter.CustomDenominatorCoefficientsDataType = numerictype([],12);
butterFilter.CustomScaleValuesDataType = numerictype([],12);
butterFilter.SectionInputDataType = 'Custom';
butterFilter.CustomSectionInputDataType = numerictype([],20,15);
butterFilter.SectionOutputDataType = 'Custom';
butterFilter.CustomSectionOutputDataType = numerictype([],20,15);
butterFilter.NumeratorProductDataType = 'Full precision';
butterFilter.DenominatorProductDataType = 'Full precision';
butterFilter.NumeratorAccumulatorDataType = 'Custom';
butterFilter.CustomNumeratorAccumulatorDataType = numerictype([],32,24);
butterFilter.DenominatorAccumulatorDataType = 'Custom';
butterFilter.CustomDenominatorAccumulatorDataType = numerictype([],32,25);
butterFilter.OutputDataType = 'Custom';
butterFilter.CustomOutputDataType = numerictype([],9,7);
butterFilter.RoundingMethod = 'nearest';
butterFilter.OverflowAction = 'wrap';
fvtool(butterFilter,'Fs',Fs,'FrequencyScale','log','Arithmetic','fixed');
```



Requantize the Filter

In the plot above, `fvtool` shows that the quantized passband is approximately 2 dB lower than the desired response. Adjust the coefficient word length from 12 to 16 to get the quantized response closer to the reference double-precision response and zoom in on the passband response. The quantized filter is now just over 0.1 dB lower than the reference filter.

```
butterFilter.CustomNumeratorCoefficientsDataType = numerictype([],16);
butterFilter.CustomDenominatorCoefficientsDataType = numerictype([],16);
butterFilter.CustomScaleValuesDataType = numerictype([],16);

h = fvtool(butterFilter,'Fs',Fs,'FrequencyScale','log','Arithmetic','fixed');
h.zoom([0 1.0 -1 1]);
```



Examine the Scale Values

A key step for hardware realization of the filter design is to check whether the scale values are reasonable and adjust the scale value if needed. First, examine the quantized scale values relative to the input specification. The input data are 8-bit values with fraction length of 7 bits. Since the first two scale values are smaller than can be represented with these input settings, most of the input values are quantized away. To correct this behavior, the filter must be scaled.

```
scaless = butterFilter.ScaleValues .* 2^7 %#ok<*NASGU>
```

```
scaless = 4×1

    0.1588
    0.1535
    4.4042
  128.0000
```

Now scale the filter using the frequency domain infinity norm. In this case, after scaling, the scale values are all equal to one.

```
scale(butterFilter,'Linf');
scaless = butterFilter.ScaleValues
```

```
scaless = 4×1

    1.0000
    1.0000
    1.0000
    1.0000
```

Generate HDL Code and Test Bench from the Quantized Filter

Starting with the correctly quantized filter, generate VHDL® or Verilog® code. You have the option of generating a VHDL or Verilog test bench to verify that the HDL design matches the MATLAB® filter.

To generate Verilog instead, change the value of the `TargetLanguage` property, from `'VHDL'` to `'Verilog'`.

Since the passband of this filter is low relative to the sampling rate, a custom input stimulus is a better way to test the filter implementation. Build the test input with one cycle of each of 50 to 300 Hz, in 50 Hz steps.

Generate 8-bit signed fixed-point input with 7 bits of fractional length.

Generate VHDL code for the filter and a VHDL test bench to verify that the results match the MATLAB results exactly.

```
userstim = [];
for n = [50, 100, 150, 200, 250, 300]
  userstim = [userstim,sin(2*pi*n/Fs*(0:Fs/n))]; %#ok
end

generatehdl(butterFilter,'Name','hdlbutter', ...
    'TargetLanguage','VHDL', ...
    'GenerateHDLTestbench','on', ...
    'TestBenchUserStimulus',userstim, ...
    'InputDataType',numerictype(1,8,7));
```

```
### Starting VHDL code generation process for filter: hdlbutter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex42204542\hdlsrc\h
### Starting generation of hdlbutter VHDL entity
### Starting generation of hdlbutter VHDL architecture
### First-order section, # 1
### Second-order section, # 2
### Second-order section, # 3
### Successful completion of VHDL code generation process for filter: hdlbutter
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 2166 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex422045
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

Generate HDL Code and Test Bench Using FDHDLTool

Alternatively, you can generate HDL code and test bench by using the `fdhdhltool` command. This command opens a dialog that allows you to customize and generate Verilog or VHDL code and test benches for the quantized filter.

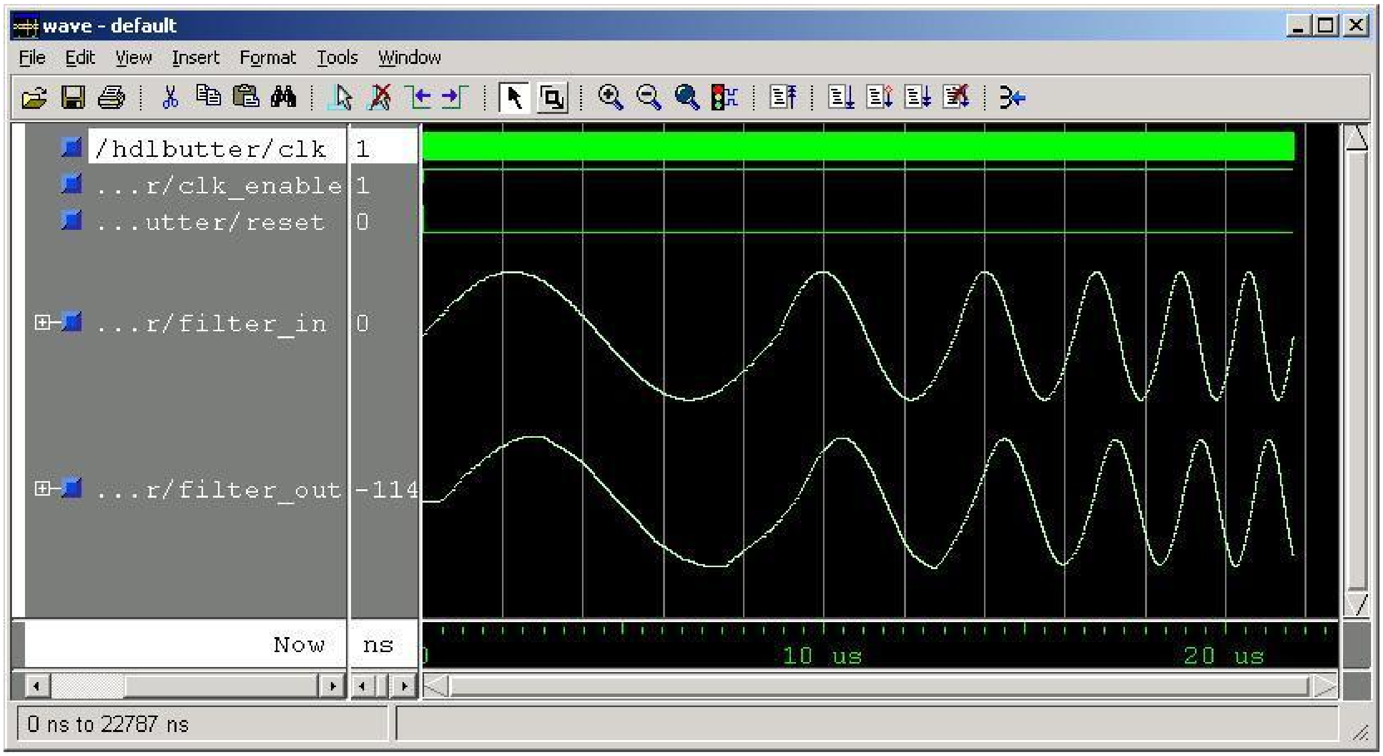
When you specify a type of filter, the tool is customized to show only the relevant options for that filter type.

```
fdhdltool(butterFilter,numerictype(1,8,7));
```

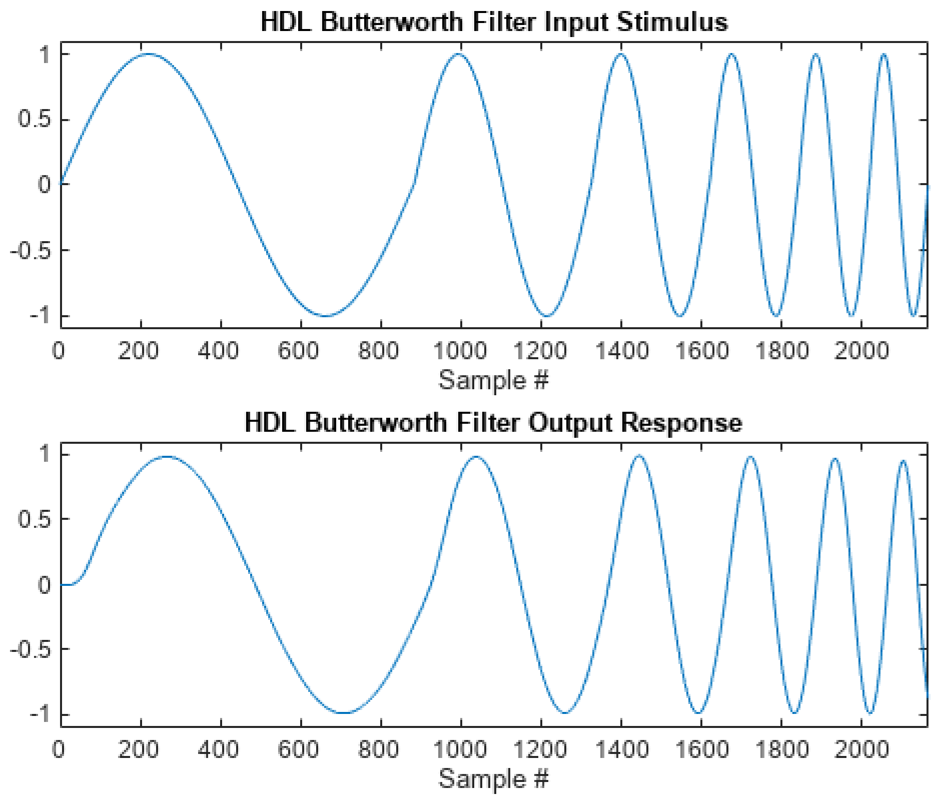
You can modify the default settings and click **Generate** to generate HDL code and/or a test bench.

ModelSim® Simulation Results

The image shows the ModelSim HDL simulator after running the VHDL test bench. Compare the ModelSim result with the MATLAB result.




```
xrange = (0:length(userstim) - 1);
y = butterFilter(fi(userstim.',1,8,7));
subplot(2,1,1); plot(xrange,userstim);
axis([0 length(userstim) -1.1 1.1]);
title('HDL Butterworth Filter Input Stimulus');
xlabel('Sample #');
subplot(2,1,2); plot(xrange, y);
axis([0 length(userstim) -1.1 1.1]);
title('HDL Butterworth Filter Output Response');
xlabel('Sample #');
```



Conclusion

You designed a Butterworth filter to meet the given specification. You then quantized the filter and discovered that the passband requirement was not met. Requantizing the coefficients and scaling the filter fixed this issue. You then generated VHDL code for the filter and a VHDL test bench.

You can use the ModelSim HDL Simulator, to verify these results. You can also experiment with VHDL and Verilog for both filters and test benches.

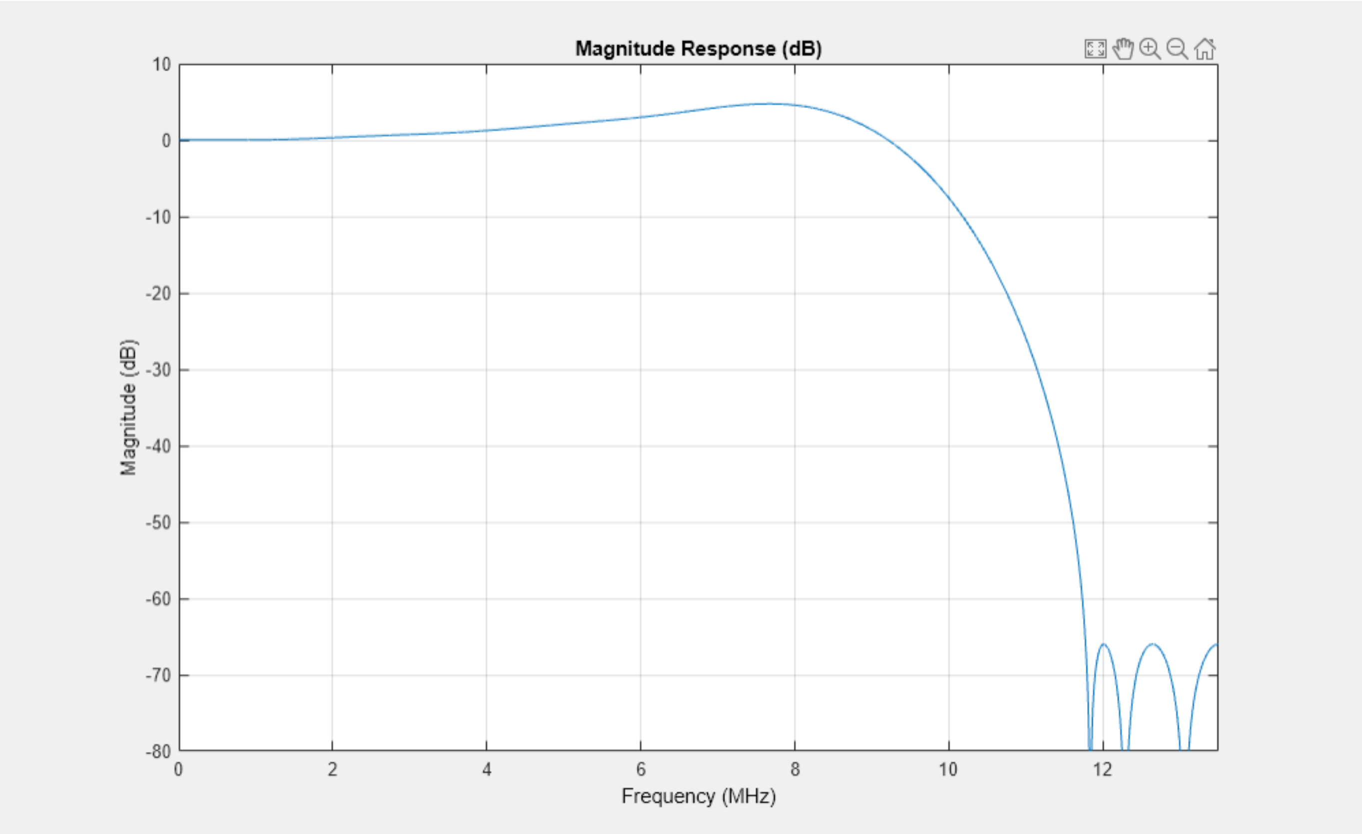
HDL Inverse Sinc Filter

This example illustrates how to generate HDL code for an inverse sinc (sin x/x) peaking filter that adds preemphasis to compensate for the inherent sinc response of the digital-to-analog converter (DAC). The input is a 10-bit video signal and the output is scaled to accommodate the gain of the inverse sinc response.

Design the Filter

Use a video sampling rate of 27 MHz and a passband edge frequency of 7.2 MHz. Set the allowable peak-to-peak passband ripple to 0.1 dB and the stopband attenuation to -66 dB. Then, design the filter using firceqrip, and create a symmetric FIR filter. Finally, examine the response using fvtool.

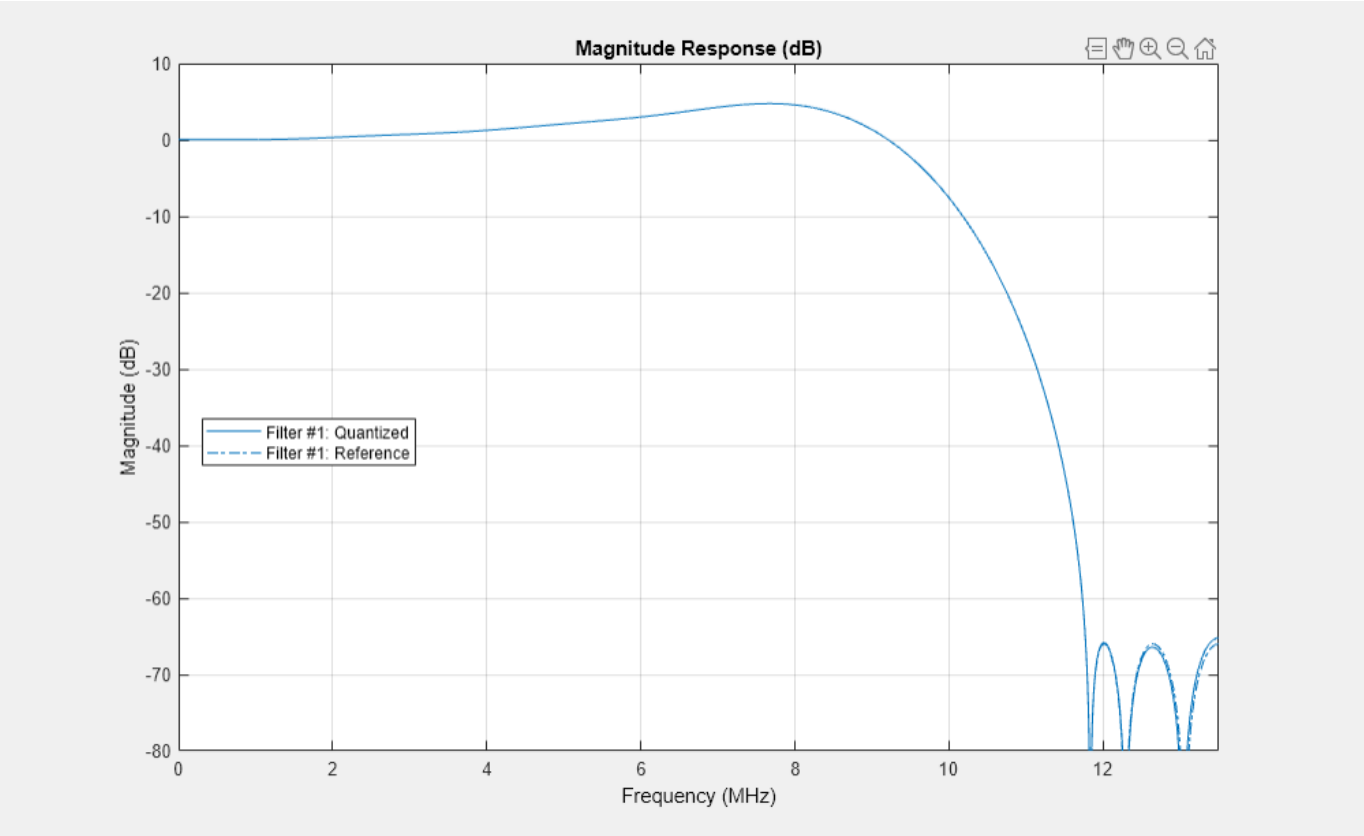
```
Fs           = 27e6;                  % Sampling Frequency in MHz
N            = 20;                    % Order
Fpass        = 7.2e6;                 % Passband Frequency in MHz
slope        = 0;                     % Stopband Slope
spectype     = 'passedge';            % Frequency Specification Type
isincffactor = 1;                     % Inverse Sinc Frequency Factor
isincpower   = 1;                     % Inverse Sinc Power
Dstop        = 10^(-66/20);           % Stopband Attenuation -66 dB
ripple       = 10^(0.1/20);           % Passband Ripple 0.1 dB p-p
Dpass        = (ripple - 1) / (ripple + 1);

% Calculate the coefficients using the FIRCEQRIP function.
b  = firceqrip(N, Fpass/(Fs/2), [Dpass, Dstop], 'slope', slope, ...
               spectype, 'invsinc', isincffactor, isincpower);
inverseSincFilter = dsp.FIRFilter('Numerator',b,'Structure','Direct form symmetric');
h = fvtool(inverseSincFilter,'Fs',Fs);
h.zoom([0 Fs/2e6 -80 10]);
```



Create the Quantized Filter

Use the infinity norm of freqz to find the maximum inverse sinc gain, and then scale this gain into bits, rounding up. Next, apply fixed point settings to the filter. Check the response with fvtool.

```
Gbits = ceil(log2(norm(freqz(inverseSincFilter), inf)));

specifyall(inverseSincFilter);
inverseSincFilter.CustomCoefficientsDataType = numerictype(1,16,15);
inverseSincFilter.CustomOutputDataType = numerictype(1,10+Gbits,9);
inverseSincFilter.CustomProductDataType = numerictype(1,32,30);
inverseSincFilter.CustomAccumulatorDataType = numerictype(1,33,30);

h = fvtool(inverseSincFilter,'Fs',Fs,'Arithmetic','fixed');
h.zoom([0 Fs/2e6 -80 10]);
```



Generate HDL Code from the Quantized Filter

Starting with the quantized filter, generate VHDL or Verilog code. You also have the option of generating a VHDL or Verilog test bench to verify that the HDL design matches the MATLAB® filter.

To generate VHDL instead, change the value of the property 'TargetLanguage', from 'Verilog' to 'VHDL'.

Generate a Verilog test bench to make sure that the result match the response you see in MATLAB exactly. Since this is a video filter, build and specify a stimulus similar to a line of video as the test stimulus.

Create a temporary work directory. After generating the HDL code (selecting Verilog in this case) and test bench, open the generated Verilog files in the editor.

```
workingdir = tempname;

Fsub        = 5e6*63/88;                     % 3.579545 MHz
VoltsperIRE = (7 + 1/7)/1000;                % IRE steps are 7.14mV
Nsamples    = 1716;                          % 27 MS/s video line
userstim = zeros(1,Nsamples);                % predefine our array

% 8 Sample raised-cosine -40 IRE
syncedge = ((cos(pi/2 *(0:7)/8).^2) - 1) * 40 * VoltsperIRE;
burst    = 20 * VoltsperIRE * sin(2*pi * Fsub/Fs * (0:Fs/(Fsub/9)));

userstim(33:40)     = syncedge;
```

```
userstim(41:170)   = repmat(-40 * VoltsperIRE, 1, 130);
userstim(171:178)  = syncedge(end:-1:1);
userstim(180:247)  = burst;
% Ramp with chroma over 1416 samples from 7.5 to 80 IRE with a 20 IRE chroma
actlen = 1416;
active = 1:actlen;
userstim(260:1675) = (((active/actlen * 72.5)+7.5) + ...
                      20 * sin(2*pi * Fsub/Fs * active)) * VoltsperIRE;
userstim(1676:Nsamples) = 72.5 * VoltsperIRE * (41:-1:1)/41;

generatehdl(inverseSincFilter, 'Name', 'hdlinvsinc', 'TargetLanguage', 'Verilog',...
    'GenerateHDLTestbench','on', ...
    'TestBenchUserStimulus', userstim,...
    'TargetDirectory', workingdir,...
    'InputDataType',numerictype(1,10,9));
```

```
### Starting Verilog code generation process for filter: hdlinvsinc
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp60b72018_afff_43ac_8d64_e118b973feac\h
### Starting generation of hdlinvsinc Verilog module
### Starting generation of hdlinvsinc Verilog module body
### Successful completion of Verilog code generation process for filter: hdlinvsinc
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1716 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp60b72018_afff_43ac_8d64_e11
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
```

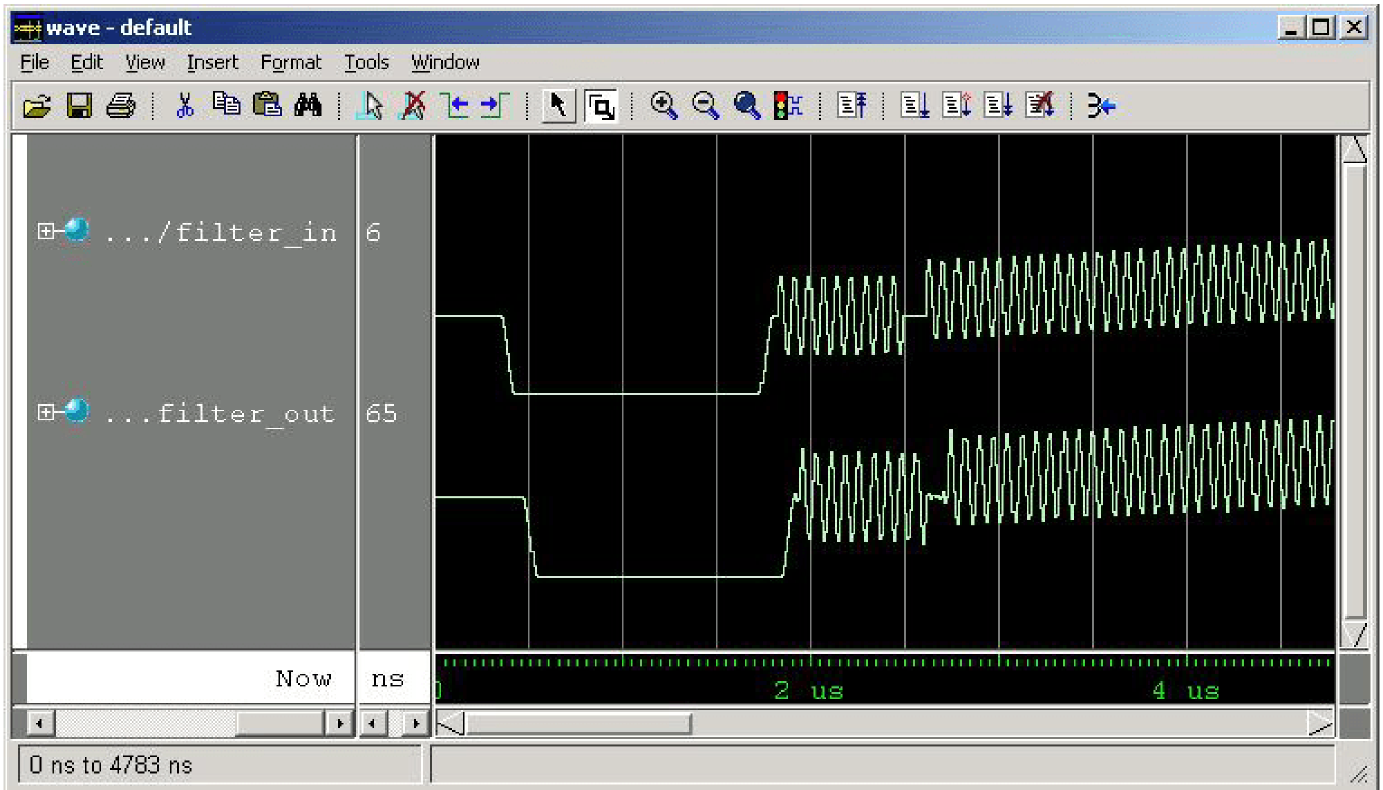
To open the generated Verilog source and test bench files in the MATLAB editor, use these commands.

```
edit(fullfile(workingdir,'hdlinvsinc.v'));
```

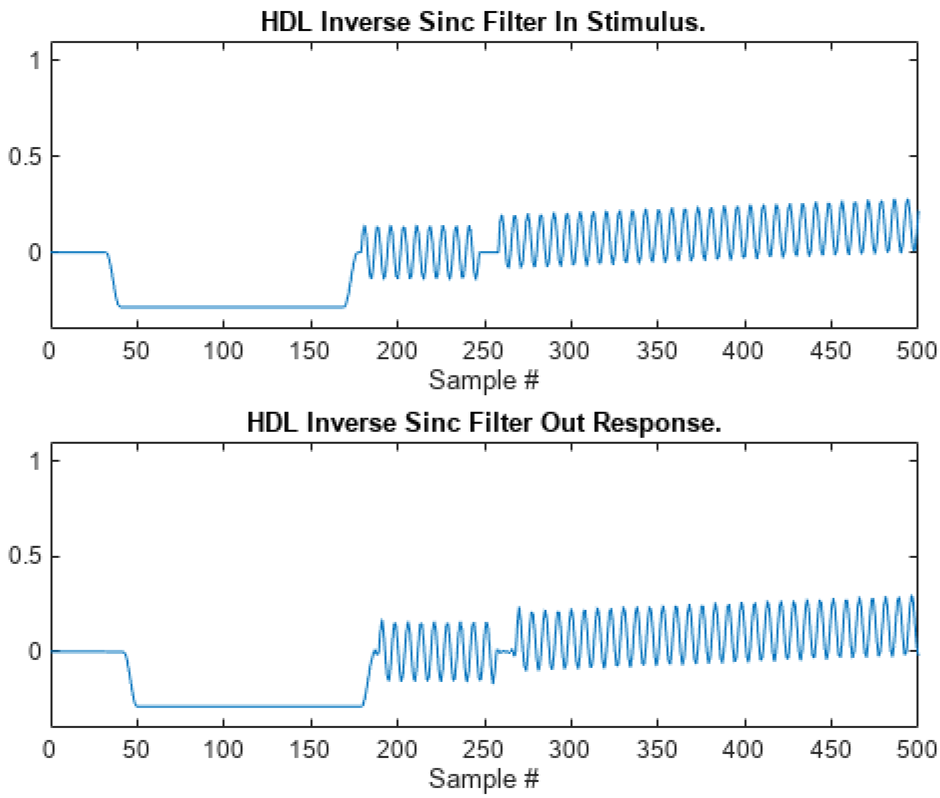
```
edit(fullfile(workingdir,'hdlinvsinc_tb.v'));
```

ModelSim® Simulation Results

The following display show the ModelSim HDL simulator waveform after running the test bench. Compare the ModelSim result with the MATLAB result below.




```
xrange = 0:Nsamples-1;
y = inverseSincFilter(fi(userstim.',1,10,9));
subplot(2,1,1); plot(xrange, userstim);
axis([0 500 -0.4 1.1]);
title('HDL Inverse Sinc Filter In Stimulus.');
xlabel('Sample #');
subplot(2,1,2); plot(xrange, y);
axis([0 500 -0.4 1.1]);
title('HDL Inverse Sinc Filter Out Response.');
xlabel('Sample #');
```



Conclusion

You designed an inverse sinc filter to meet the given specification. You generated Verilog code and a Verilog test bench using an approximation of a video line as the test stimulus.

You can use an HDL Simulator, to verify these results. You can also experiment with VHDL and Verilog for both filters and test benches.

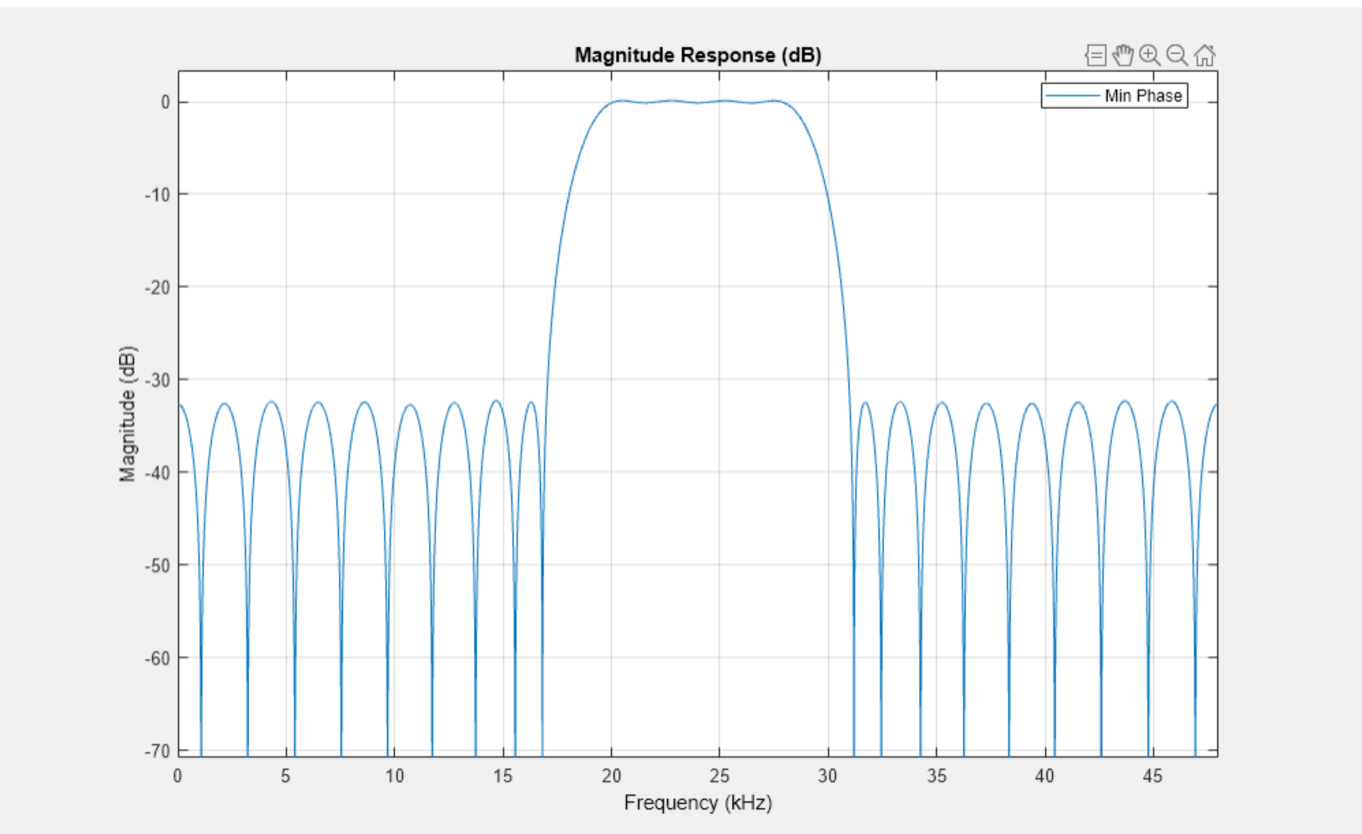
HDL Minimum Phase FIRT Filter

This example illustrates how to generate HDL code for a minimum phase FIRT filter with 10-bit input data. This is a bandpass filter with sample rate of 96 kHz and passband from approximately 19 kHz to 29 kHz. This type of filter is commonly used in feedback loops where linear phase is not sufficient and minimum phase or as close as is achievable is required.

Set up the Coefficients

Design the filter using firgr, which uses the generalized Remez design method. The use of the 'minphase' argument to firgr forces a minimum phase filter design. Then, use fvtool to visualize the filter response.

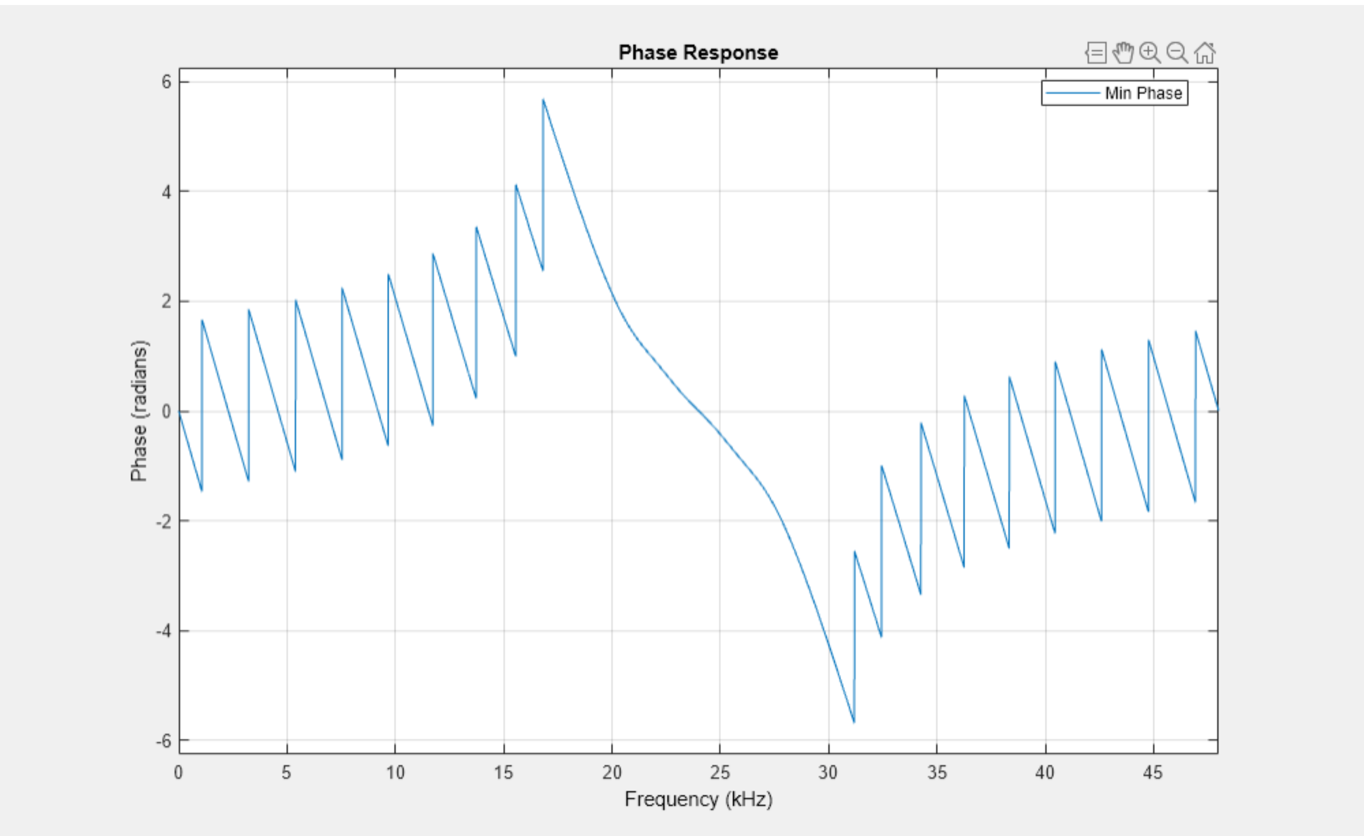
```
Fs   = 96000;
Fn   = Fs/2;
f    = [0 17000 20000 28000 31000 Fn]/Fn;
a    = [0     0     1     1     0  0];
w    = [5 1 5];
b    = firgr(44, f, a, w, 'minphase');
hfvt = fvtool(b,'Fs', Fs,...
              'MagnitudeDisplay', 'Magnitude (dB)',...
              'legend','on');
legend(hfvt,'Min Phase');
```



Examine the Phase Response

Check the phase response of the filter.

```
hfvt = fvtool(b,'Fs', Fs,...
              'Analysis', 'phase',...
              'legend','on');
legend(hfvt, 'Min Phase');
```



Create the Quantized FIRT Fixed-Point Filter

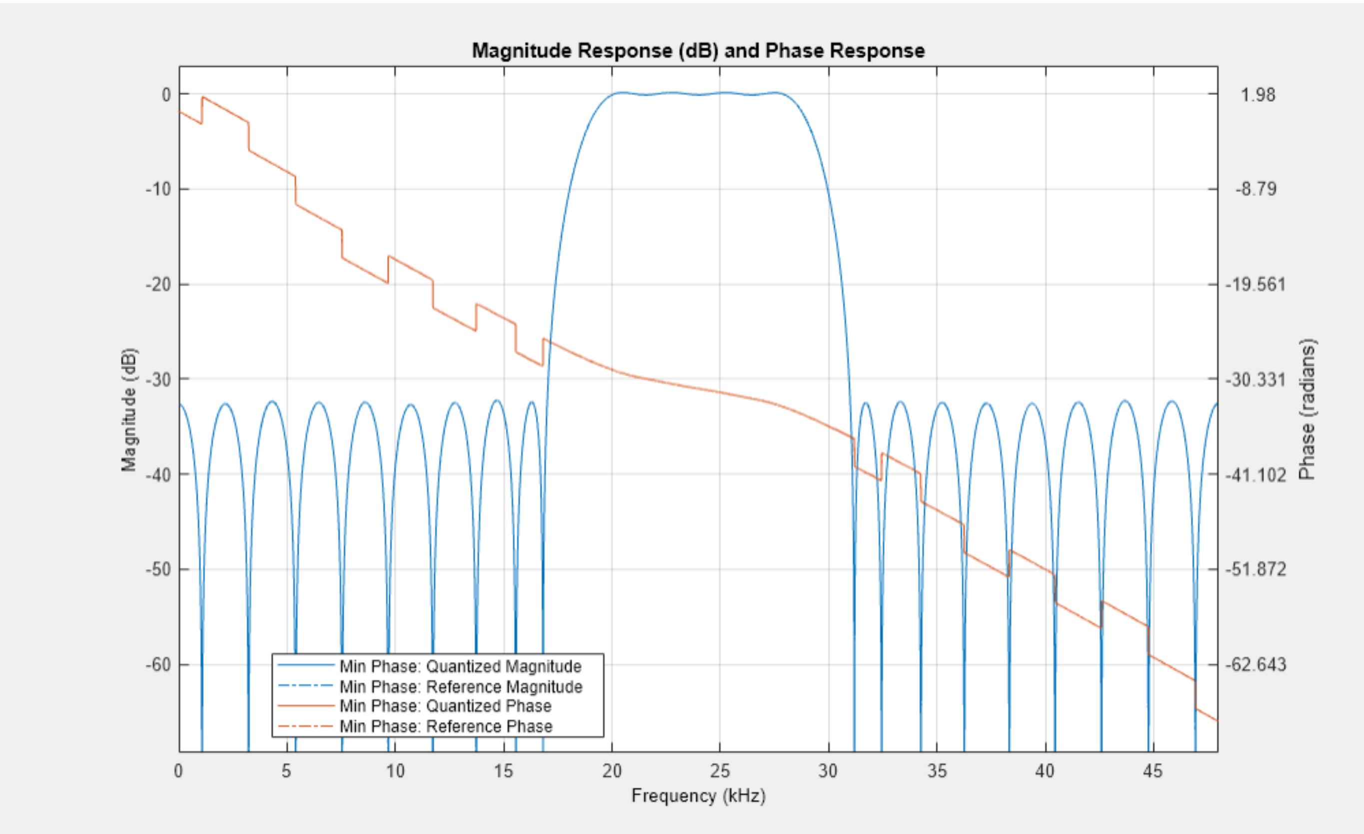
Having checked the minimum phase filter design, construct a FIR filter System object with 'Direct form transposed' structure. Set the coefficient word length to 15, and use full precision for the other filter settings.

```
b_fixed = fi(b,1,15); % use best precision fraction length
T_coeff = numerictype(b_fixed);
minPhaseFilter = dsp.FIRFilter('Structure','Direct form transposed');
minPhaseFilter.Numerator = double(b_fixed);
minPhaseFilter.FullPrecisionOverride = false;
minPhaseFilter.CoefficientsDataType = 'Custom';
minPhaseFilter.CustomCoefficientsDataType = T_coeff;
minPhaseFilter.ProductDataType = 'Full precision';
minPhaseFilter.AccumulatorDataType = 'Full precision';
minPhaseFilter.OutputDataType = 'Same as accumulator';
```

Check the Fixed-Point Filter Relative to the Reference Design

Check the quantized filter relative to the reference design. The magnitude response is correct but the phase response is no longer min-phase, due to quantization.

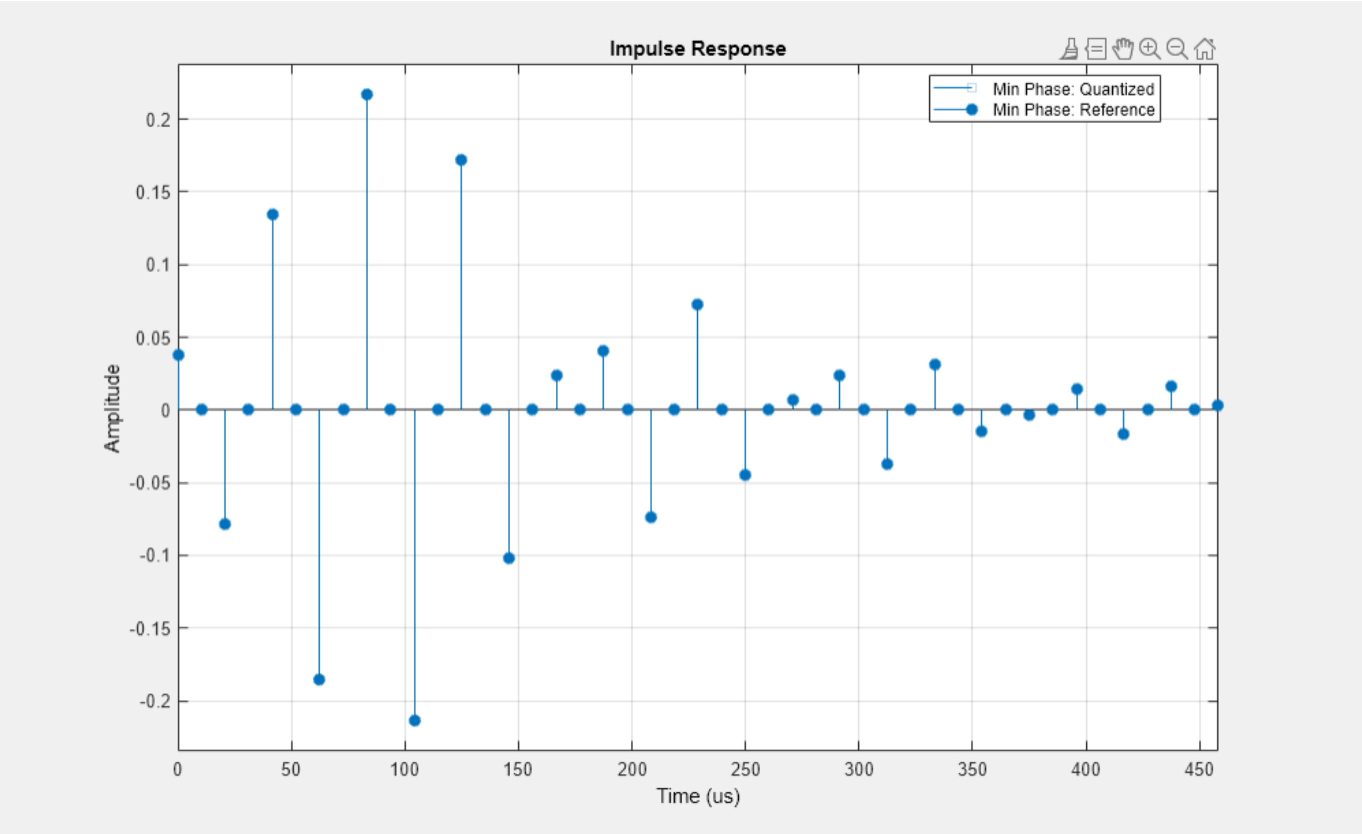
```
hfvt = fvtool(minPhaseFilter, 'Fs', Fs, ...
              'Analysis', 'freq', ...
              'legend', 'on', ...
              'Arithmetic', 'fixed');
legend(hfvt, 'Min Phase');
```



Check the Impulse Response

Plot the impulse response of the quantized filter. Many of the coefficients have quantized to zero and the overall response still meets the specification, even though the zeros have made the phase response non-minimum. These zeros lead to a smaller implementation because HDL code is not generated for multiplies by zero.

```
hfvt = fvtool(minPhaseFilter, 'Fs', Fs, ...
              'Analysis', 'Impulse', ...
              'legend', 'on', ...
              'Arithmetic', 'fixed');
legend(hfvt, 'Min Phase');
```



Generate HDL Code and Test Bench from the Quantized Filter

Starting from the quantized filter, generate VHDL or Verilog.

Create a temporary work directory. After generating the HDL code (Verilog in this case), open the generated file in the editor.

Generate a Verilog test bench to verify that the results match the results in MATLAB. Use the chirp predefined input stimulus.

To generate VHDL code and VHDL test bench instead, change the value for 'TargetLanguage' property from 'Verilog' to 'VHDL'.

Assume an input of 10-bit word length with 9-bit fractional bits.

```
workingdir = tempname;
generatehdl(minPhaseFilter,'Name', 'hdlminphasefilt', ...
               'TargetLanguage', 'Verilog', ...
               'GenerateHDLTestbench','on', ...
               'TestBenchStimulus', 'chirp', ...
               'TargetDirectory', workingdir, ...
               'InputDataType', numerictype(1,10,9));
```

```
### Starting Verilog code generation process for filter: hdlminphasefilt
### Starting Verilog code generation process for filter: hdlminphasefilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp73a4108a_7571_46fc_98b6_1d789b21b8d8\
### Starting generation of hdlminphasefilt Verilog module
```

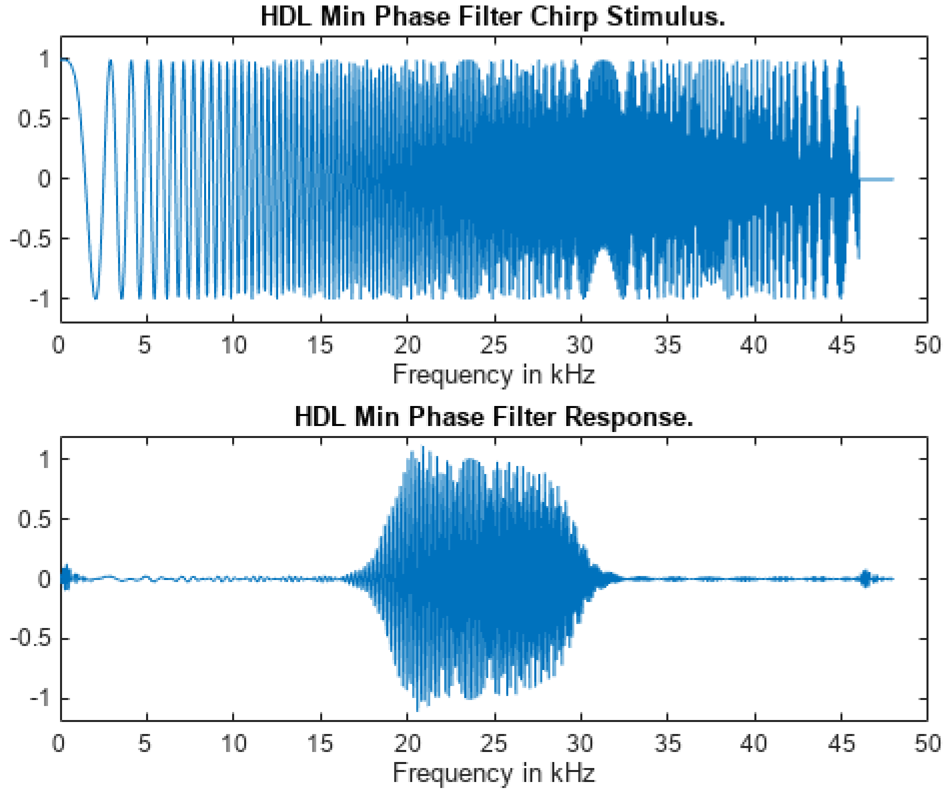
```
### Starting generation of hdlminphasefilt Verilog module body
### Successful completion of Verilog code generation process for filter: hdlminphasefilt
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1069 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp73a4108a_7571_46fc_98b6_1d
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
```

```
edit(fullfile(workingdir, 'hdlminphasefilt.v'));
```

Plot the Test Bench Stimulus and the Filter Response

Plot the filter input stimulus and output response on separate plots.

```
x = generatetbstimulus(minPhaseFilter,'TestBenchStimulus','chirp','InputDataType', numerictype(1
xrange = (0:length(x) - 1).*( Fn / (length(x) - 1))/1e3;
y = minPhaseFilter(x.');
subplot(2,1,1); plot(xrange, x); ylim(ylim.*1.1);
axis([0,50,-1.2,1.2]);
title('HDL Min Phase Filter Chirp Stimulus.');
xlabel('Frequency in kHz');
subplot(2,1,2); plot(xrange, y); ylim(ylim.*1.1);
axis([0,50,-1.2,1.2]);
title('HDL Min Phase Filter Response.');
xlabel('Frequency in kHz');
```



Conclusion

You designed a minimum phase filter and then converted it to a FIR filter System object with transposed structure. You then generated Verilog code for the filter design and a Verilog test bench to functionally verify the results.

You can use a Verilog simulator, such as ModelSim®, to verify these results.

HDL Tone Control Filter Bank

This example illustrates how to generate HDL code for bank of 24 first-order shelving filters that implement an audio tone control with 1 dB steps from -6 dB to +6 dB for bass and treble.

The filters are analytically designed using a simple formula for a first-order filter with one pole and one zero on the real axis.

A filter bank is designed since changing the filter coefficients on-the-fly can lead to transients in the audio (clicks and pops) as the boost/cut control is moved. With a bank of filters running continuously, the appropriate filter is selected from the bank of filters when the output is near any zero crossing to avoid these transients.

Set up the Parameters

Use the CD sampling rate of 44.1 kHz with bass and treble corners at 100 Hz and 1600Hz.

```
Fs  = 44100;                          % all in Hz
Fcb =   100;
Fct =  1600;
```

Define the Tangent Frequency Mapping Parameters

Map the corner frequencies by the tangent to move from the analog to the digital domain. Then, define the range of cut and boost to be applied, choosing a 12 dB total range in 1 dB steps. Convert decibels to linear gain and separate the boost and cut vectors.

```
basstan   = tan(pi*Fcb/Fs);
trebletan = tan(pi*Fct/Fs);

dbrange  = [-6:-1, +1:+6];            % -6 dB to +6 dB
linrange = 10.^(dbrange/20);
boost = linrange(linrange>1);
cut   = linrange(linrange<=1);
Nfilters = 2 * length(dbrange);       % 2X for bass and treble
```

Design the Filter Bank

Complete the bilinear transform on the poles, then compute the zeros of the filters based on the desired boost or cut. Since boost and cut are vectors, we can design all the filters at the same time using vector arithmetic. Note that a1 is always one in these filters.

```
a2_bass_boost    = (basstan - 1) / (basstan + 1);
b1_bass_boost   = 1 + ((1 + a2_bass_boost) .* (boost - 1)) / 2;
b2_bass_boost   = a2_bass_boost + ...
                  ((1 + a2_bass_boost) .* (boost - 1)) / 2;

a2_bass_cut      = (basstan - cut) / (basstan + cut);
b1_bass_cut     = 1 + ((1 + a2_bass_cut) .* (cut - 1)) / 2;
b2_bass_cut     = a2_bass_cut + ((1 + a2_bass_cut) .* (cut - 1)) / 2;

a2_treble_boost  = (trebletan - 1) / (trebletan + 1);
b1_treble_boost   = 1 + ((1 - a2_treble_boost) .* (boost - 1)) / 2;
b2_treble_boost   = a2_treble_boost + ...
                    ((a2_treble_boost - 1) .* (boost - 1)) / 2;
```

```
a2_treble_cut      = (cut .* trebletan - 1) / (cut .* trebletan + 1);
b1_treble_cut     = 1 + ((1 - a2_treble_cut) .* (cut - 1)) / 2;
b2_treble_cut     = a2_treble_cut + ...
                    ((a2_treble_cut - 1) .* (cut - 1)) / 2;
```

Build the Filter Bank

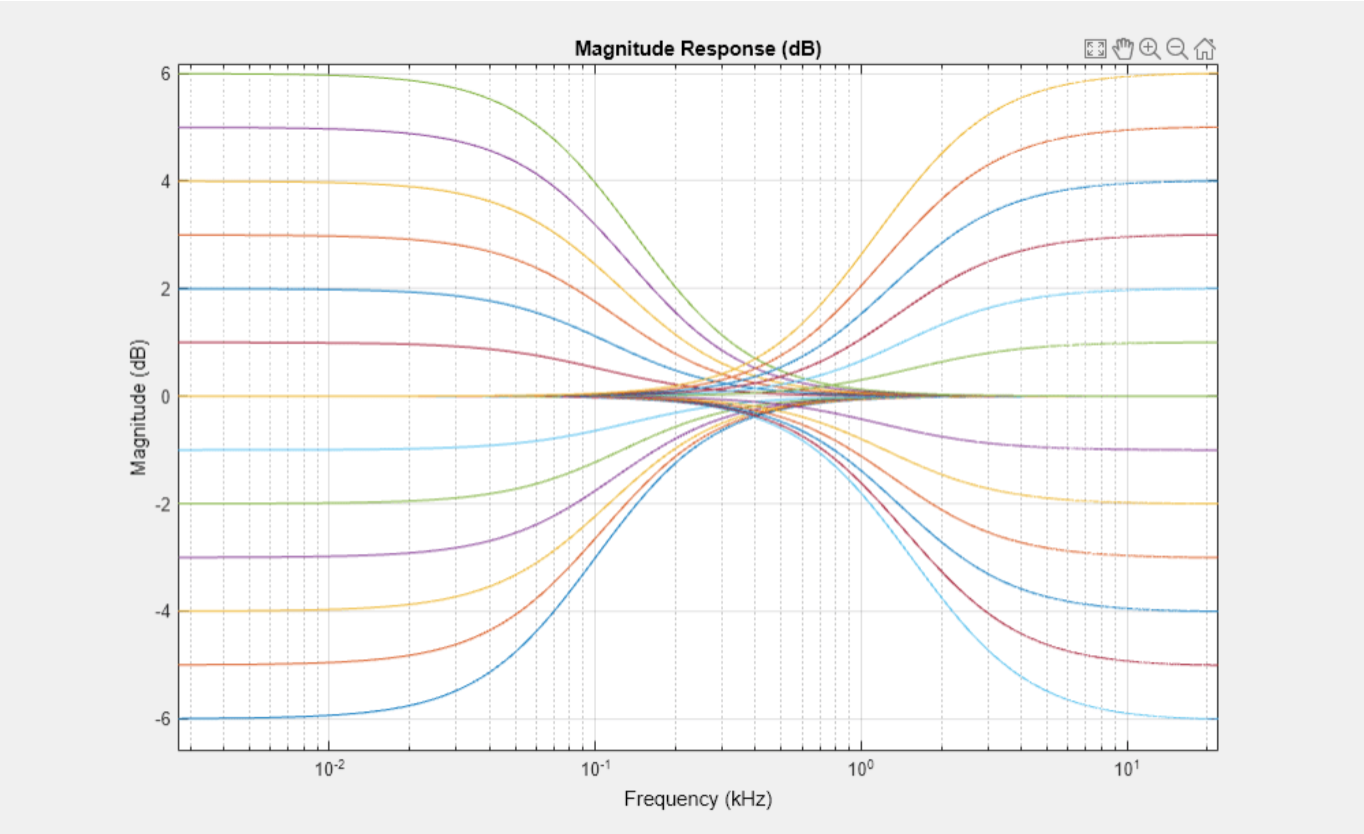
Build the numerator and denominator arrays for the entire filter bank. Then build a cell array of filters in {b,a,b,a,...} form for fvtool. Preallocate the cell array for speed.

```
filterbank = cell(1, 2*Nfilters);     % 2X for numerator and denominator
% Duplicate a2's into vectors
a2_bass_boost   = repmat(a2_bass_boost,   1, length(boost));
a2_bass_cut     = repmat(a2_bass_cut,     1, length(cut));
a2_treble_boost = repmat(a2_treble_boost, 1, length(boost));
a2_treble_cut   = repmat(a2_treble_cut,   1, length(cut));

filterbank_num = [b1_bass_cut, b1_bass_boost, b1_treble_cut, b1_treble_boost ; ...
                  b2_bass_cut, b2_bass_boost, b2_treble_cut, b2_treble_boost ]';
% a1 is always one
filterbank_den = [ones(1, Nfilters); ...
                  a2_bass_cut, a2_bass_boost, a2_treble_cut, a2_treble_boost]';

filterbank(1:2:end) = num2cell(filterbank_num, 2);
filterbank(2:2:end) = num2cell(filterbank_den, 2);
```

Check the Response of the Filter Bank

Use fvtool in log frequency mode to see the audio band more clearly. Also set the sampling frequency.

```
fvtool(filterbank{:}, 'FrequencyScale', 'log', 'Fs', Fs);
```



Create the Quantized Filter Bank

Create a quantized filter for each double-precision filter designed above. Assume CD-quality input of 16 bits and an output word length of 18 bits to allow for the +6 dB gain with some headroom.

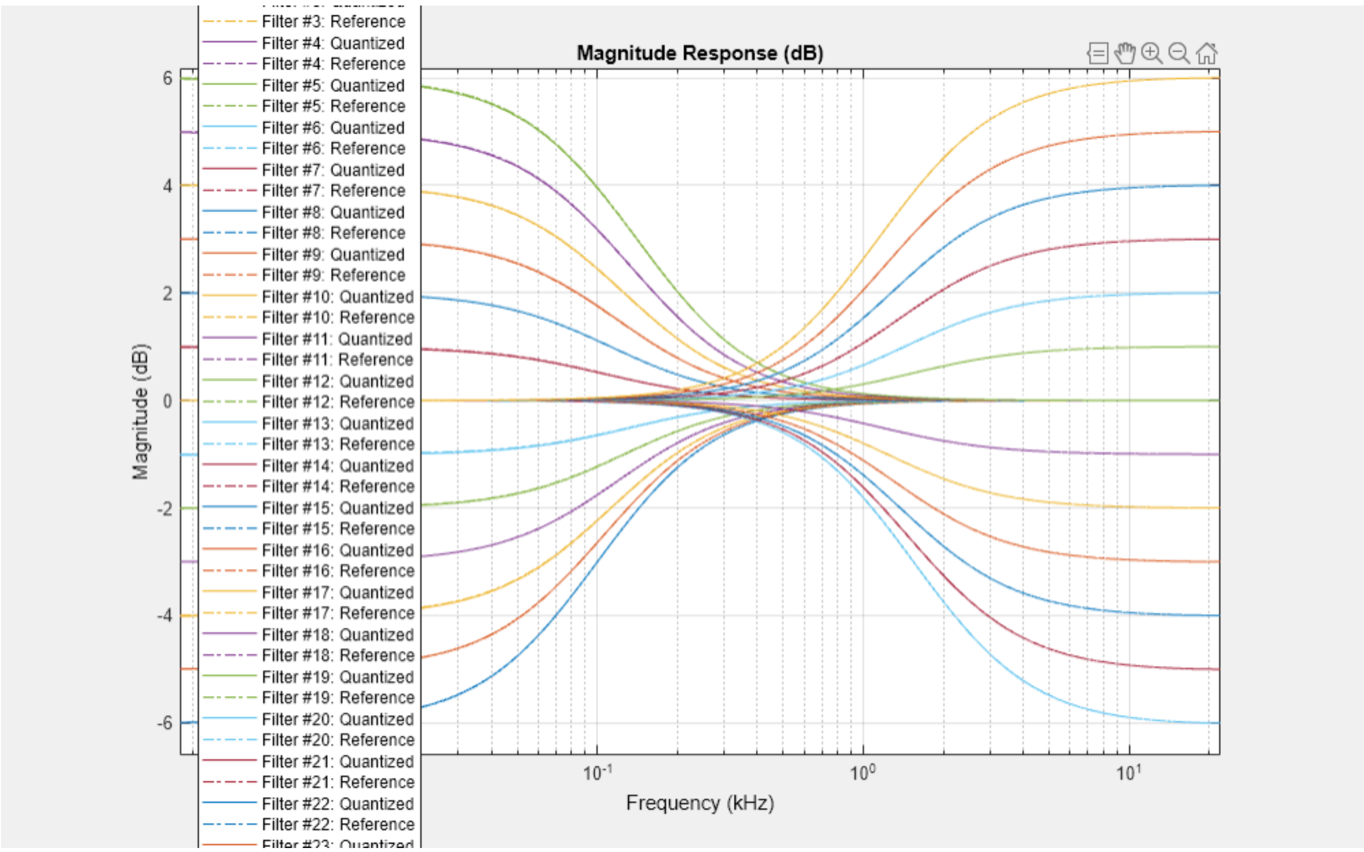
```
quantizedfilterbank = cell(1, Nfilters);
for n = 1:Nfilters
  quantizedfilterbank{n} = dsp.BiquadFilter('Structure','Direct Form I');
  quantizedfilterbank{n}.SOSMatrix = [filterbank_num(n,:),0,...
                                      filterbank_den(n,:),0];

  quantizedfilterbank{n}.NumeratorCoefficientsDataType         = 'Custom';
  quantizedfilterbank{n}.CustomNumeratorCoefficientsDataType   = numerictype([],16);
  quantizedfilterbank{n}.CustomDenominatorCoefficientsDataType = numerictype([],16);
  quantizedfilterbank{n}.CustomScaleValuesDataType             = numerictype([],16);

  quantizedfilterbank{n}.OutputDataType       = 'Custom';
  quantizedfilterbank{n}.CustomOutputDataType = numerictype([],18,15);

  quantizedfilterbank{n}.SectionOutputDataType       = 'Custom';
  quantizedfilterbank{n}.CustomSectionOutputDataType = numerictype([],18,15);
  quantizedfilterbank{n}.NumeratorProductDataType    = 'Full precision';
  quantizedfilterbank{n}.DenominatorProductDataType  = 'Full precision';

  quantizedfilterbank{n}.NumeratorAccumulatorDataType         = 'Custom';
  quantizedfilterbank{n}.CustomNumeratorAccumulatorDataType   = numerictype([],34,30);
  quantizedfilterbank{n}.DenominatorAccumulatorDataType       = 'Custom';
```

```
    quantizedfilterbank{n}.CustomDenominatorAccumulatorDataType = numerictype([],34,29);

    quantizedfilterbank{n}.RoundingMethod  = 'Floor';
    quantizedfilterbank{n}.OverflowAction  = 'Wrap';
end
```

Check the Response of the Quantized Filter Bank

Check the quantized filter bank using fvtool again in log frequency mode with the sampling rate set.

```
fvtool(quantizedfilterbank{:}, 'FrequencyScale', 'log', 'Fs', Fs,'Arithmetic','fixed');
```



Generate HDL for the Filter Bank and Test Benches

Generate HDL for each of the 24 first-order filters and test benches to check each design. The target language here is Verilog.

Use the canonic sign-digit (CSD) techniques to avoid using multipliers in the design. Specify this with the 'CoeffMultipliers', 'CSD' property-value pair. Since the results of using this optimization are not always numerically identical to regular multiplication that results in overflows, set the test bench 'ErrorMargin' property to 1 bit of allowable error.

Create a custom stimulus to illustrate the gain of filters by generating one-half cycle of a 20 Hz tone and 250 cycles of a 10 kHz tone. Use the low frequency tone for the bass boost/cut filters and the high frequency tone for the treble boost/cut filters.

Create a temporary work directory.

To generate VHDL code instead, change the property 'TargetLanguage', from 'Verilog' to 'VHDL'.

```
bassuserstim = sin(2*pi*20/Fs*(0:Fs/40));
trebuserstim = sin(2*pi*10000/Fs*(0:Fs/40));
workingdir   = tempname;

for n = 1:Nfilters/2
  generatehdl(quantizedfilterbank{n},...
              'Name', ['tonecontrol', num2str(n)],...
              'TargetDirectory', workingdir,...
              'InputDataType', numerictype(1,16,15),...
              'TargetLanguage', 'Verilog',...
              'CoeffMultipliers','CSD', ...
              'GenerateHDLTestbench','on', ...
              'TestBenchUserStimulus', bassuserstim, ...
              'ErrorMargin', 1);
end
```

```
### Starting Verilog code generation process for filter: tonecontrol1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol1 Verilog module
### Starting generation of tonecontrol1 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol1
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol2 Verilog module
### Starting generation of tonecontrol2 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol2
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol3 Verilog module
### Starting generation of tonecontrol3 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol3
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol4
```

```
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\
### Starting generation of tonecontrol4 Verilog module
### Starting generation of tonecontrol4 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol4
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol5
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\
### Starting generation of tonecontrol5 Verilog module
### Starting generation of tonecontrol5 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol5
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol6
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\
### Starting generation of tonecontrol6 Verilog module
### Starting generation of tonecontrol6 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol6
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol7
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\
### Starting generation of tonecontrol7 Verilog module
### Starting generation of tonecontrol7 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol7
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol8
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\
### Starting generation of tonecontrol8 Verilog module
### Starting generation of tonecontrol8 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol8
### HDL latency is 2 samples
```

```
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol9
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol9 Verilog module
### Starting generation of tonecontrol9 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol9
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol10
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol10 Verilog module
### Starting generation of tonecontrol10 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol10
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol11
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol11 Verilog module
### Starting generation of tonecontrol11 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol11
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol12
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol12 Verilog module
### Starting generation of tonecontrol12 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol12
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
```

```
for n = Nfilters/2+1:Nfilters
  generatehdl(quantizedfilterbank{n},...
              'Name', ['tonecontrol', num2str(n)],...
              'TargetDirectory', workingdir,...
              'InputDataType', numerictype(1,16,15),...
              'TargetLanguage', 'Verilog',...
              'CoeffMultipliers','CSD', ...
              'GenerateHDLTestbench','on', ...
              'TestBenchUserStimulus', bassuserstim, ...
              'ErrorMargin', 1);
end
```

```
### Starting Verilog code generation process for filter: tonecontrol13
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol13 Verilog module
### Starting generation of tonecontrol13 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol13
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol14
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol14 Verilog module
### Starting generation of tonecontrol14 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol14
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol15
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol15 Verilog module
### Starting generation of tonecontrol15 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol15
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol16
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol16 Verilog module
### Starting generation of tonecontrol16 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol16
```

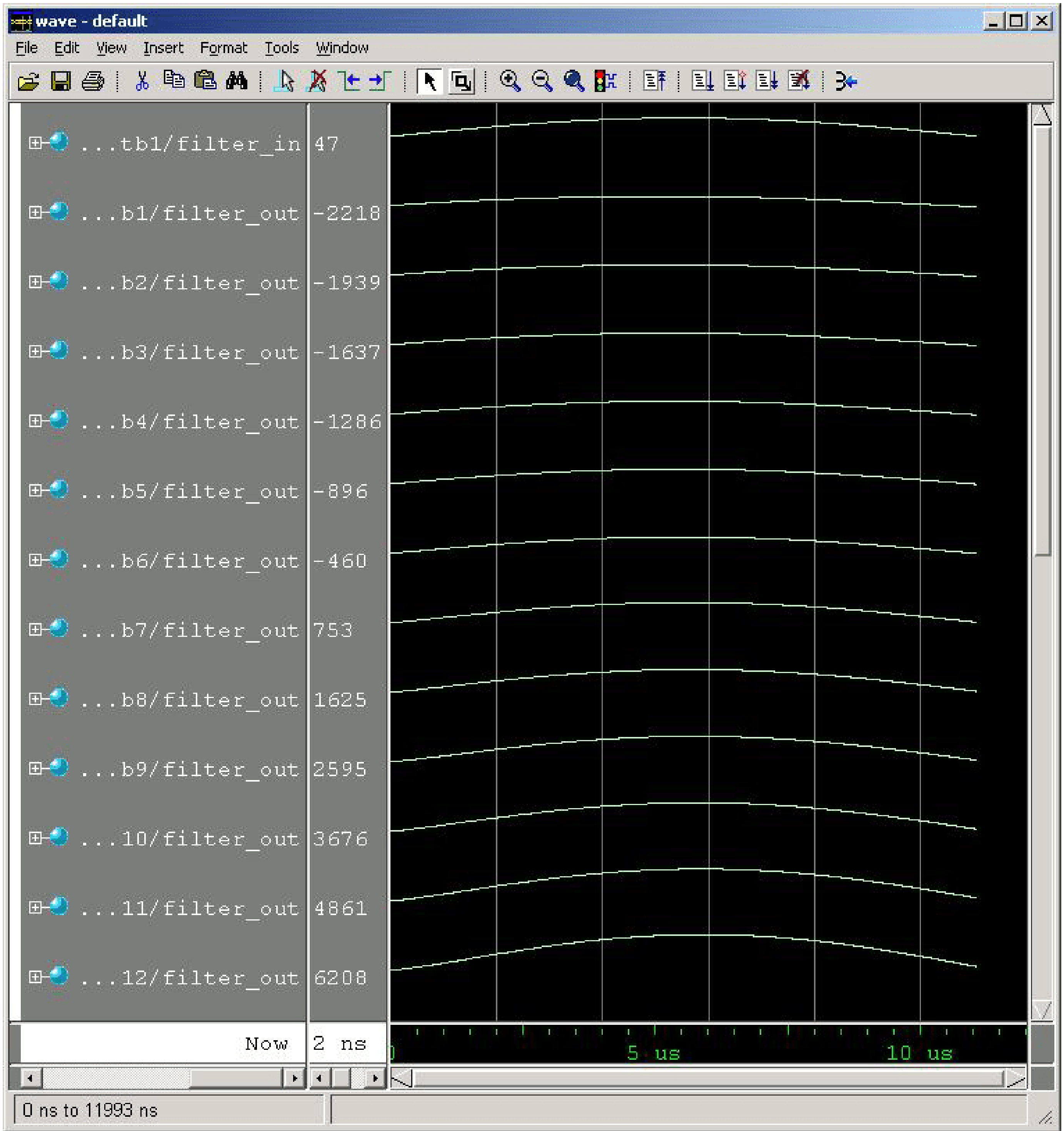
```
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol17
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol17 Verilog module
### Starting generation of tonecontrol17 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol17
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol18
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol18 Verilog module
### Starting generation of tonecontrol18 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol18
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol19
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol19 Verilog module
### Starting generation of tonecontrol19 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol19
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol20
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol20 Verilog module
### Starting generation of tonecontrol20 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol20
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
```

```
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol21
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol21 Verilog module
### Starting generation of tonecontrol21 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol21
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol22
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol22 Verilog module
### Starting generation of tonecontrol22 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol22
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol23
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol23 Verilog module
### Starting generation of tonecontrol23 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol23
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
### Starting Verilog code generation process for filter: tonecontrol24
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6c62838017\t
### Starting generation of tonecontrol24 Verilog module
### Starting generation of tonecontrol24 Verilog module body
### First-order section, # 1
### Successful completion of Verilog code generation process for filter: tonecontrol24
### HDL latency is 2 samples
### Starting generation of VERILOG Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1103 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp46415296_8ed3_4bfa_bd2d_ba6
### Creating stimulus vectors ...
### Done generating VERILOG Test Bench.
```

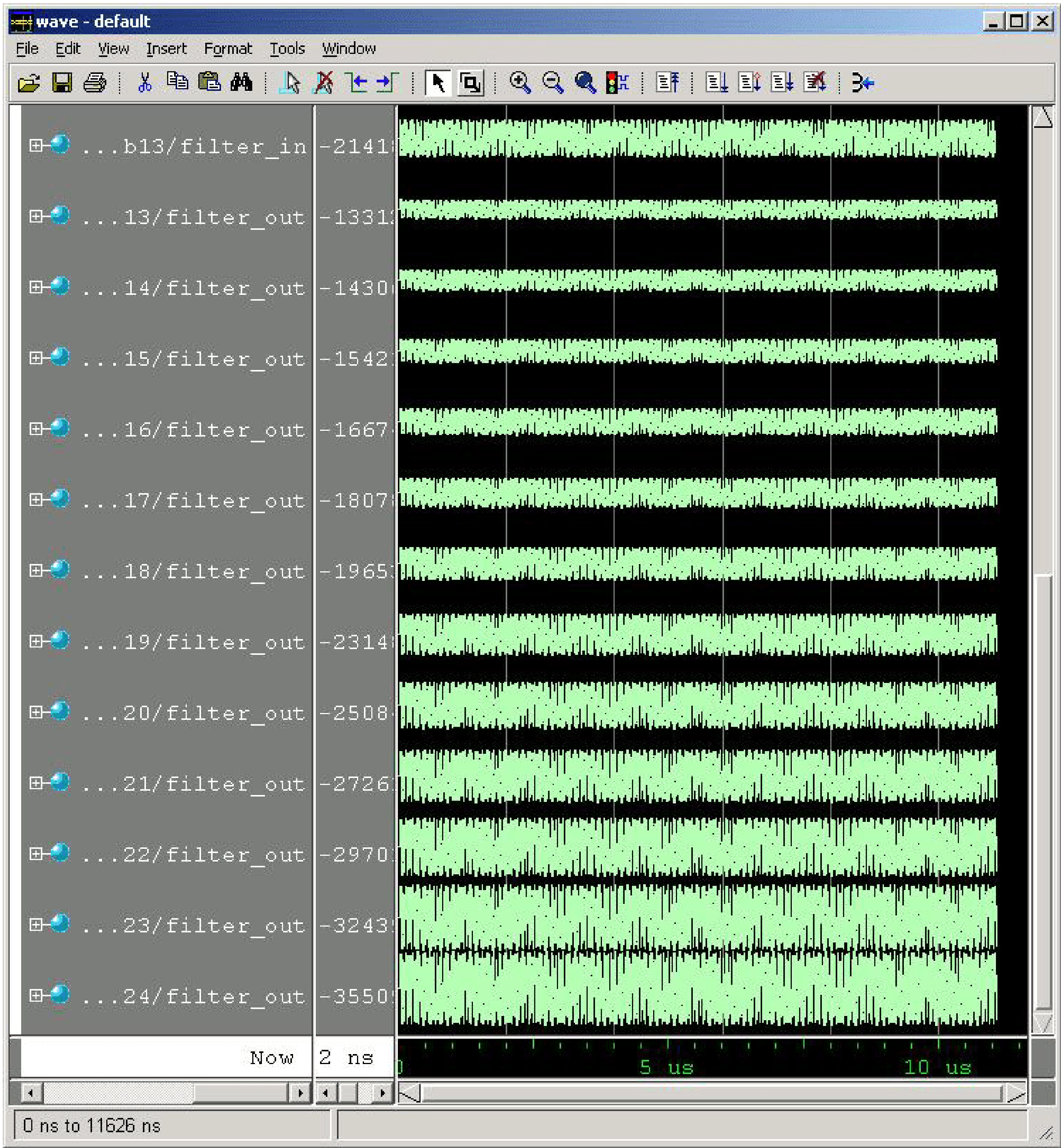
ModelSim® Simulation Results

The following display shows the ModelSim® HDL simulator running these test benches.

Bass response to the 20 Hz tone:



Treble response to the 10 kHz tone:



Conclusion

You designed a filter bank of double-precision bass and treble boost/cut first order filters directly using the bilinear transform. You then used the filter coefficients to create a bank of quantized filters with CD-quality 16-bit inputs and 18-bit outputs. After checking the response of the quantized filters, you generated Verilog code for each filter in the filter bank along with a Verilog test bench that used a custom input stimulus for the bass and treble filters.

To complete the solution of providing tone controls to an audio system, you can add a cross-fader to the outputs of each section of the filter bank. These cross-faders should take several sample times to switch smoothly from one boost or cut step to the next.

Using a full bank of filters is only one approach to solving this type of problem. Another approach would be to use two filters for each band (bass and treble) with programmable coefficients that can be changed under software control. One of the two filters would be the current setting, while the other would be the next setting. As you adjusted the tone controls, the software would ping-pong between the filters exchanging current and next with a simple fader. The trade-off is that the constant coefficient filter bank shown above is uses no multipliers while the seemingly simpler ping-pong scheme requires several multipliers.

HDL Video Filter

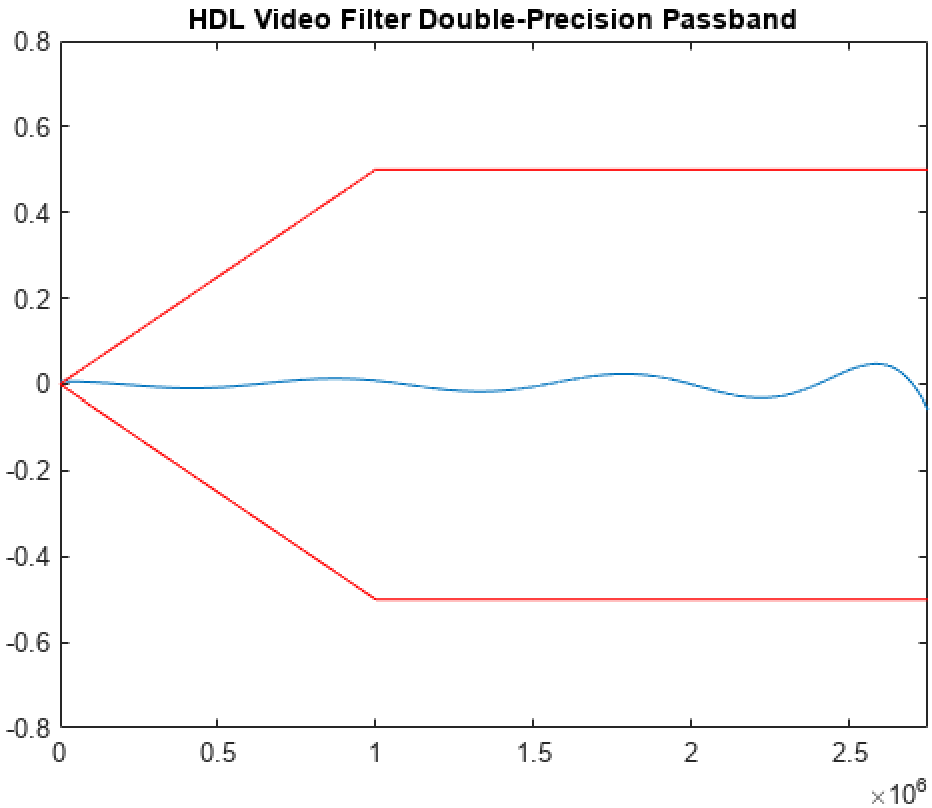
This example illustrates how to generate HDL code for an ITU-R BT.601 luma filter with 8-bit input data and 10-bit output data. This filter is a low-pass filter with a -3 dB point of 3.2 MHz with a 13.5 MHz sampling frequency and a specified range for both passband ripple and stopband attenuation shown in the ITU specification. The filter coefficients were designed using the DSP System Toolbox™. This example focuses on quantization effects and generating HDL code for the System object filter.

Set up the Coefficients

Assign the previously designed filter coefficients to variable b. This is a halfband filter and, therefore, every other coefficient is zero with the exception of the coefficient at the filter midpoint, which is exactly one-half.

Check that double-precision filter design meets the ITU-R BT.601 template for passband ripple and stopband attenuation using freqz and plot the passband. The red lines show the allowed variation in the specification.

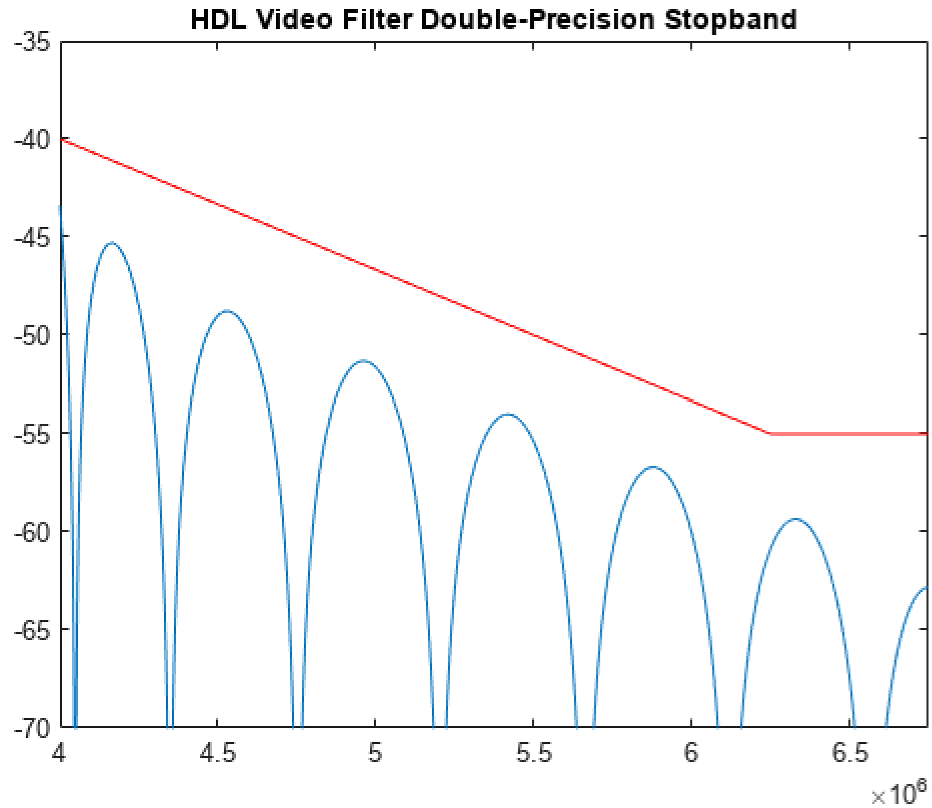
```
b = [0.00303332064210658,  0,...
    -0.00807786494715095,  0,...
     0.0157597395206364,   0,...
    -0.028508691397868,    0,...
     0.0504985344927114,   0,...
    -0.0977926818362618,   0,...
     0.315448742029959,...
     0.5,...
     0.315448742029959,    0,...
    -0.0977926818362618,   0,...
     0.0504985344927114,   0,...
    -0.028508691397868,    0,...
     0.0157597395206364,   0,...
    -0.00807786494715095,  0,...
     0.00303332064210658];

f = 0:100:2.75e6;
H = freqz(b,1,f,13.5e6);
plot(f,20*log10(abs(H)));
title('HDL Video Filter Double-Precision Passband');
axis([0 2.75e6 -.8 .8]);
passbandrange = {[2.75e6;  1e6;   0;  1e6; 2.75e6],...
                 [  -0.5; -0.5;   0;  0.5;    0.5]};
line(passbandrange{:}, 'Color', 'red');
```



Plot the Stopband

The red line shows a "not to exceed" limit on the stopband.

```
f = 4e6:100:6.75e6;
H = freqz(b,1,f,13.5e6);
plot(f,20*log10(abs(H)));
title('HDL Video Filter Double-Precision Stopband');
axis([4e6 6.75e6 -70 -35]);
stopbandrange = {[4e6;   6.25e6;  6.75e6],...
                 [-40;      -55;     -55]};
line(stopbandrange{:}, 'Color', 'red');
```



Create the Quantized Filter

Create a FIR filter System object filter with previously defined coefficients. Experiment with the coefficient word length to get the desired response for 8-bit input data and 10-bit output data.

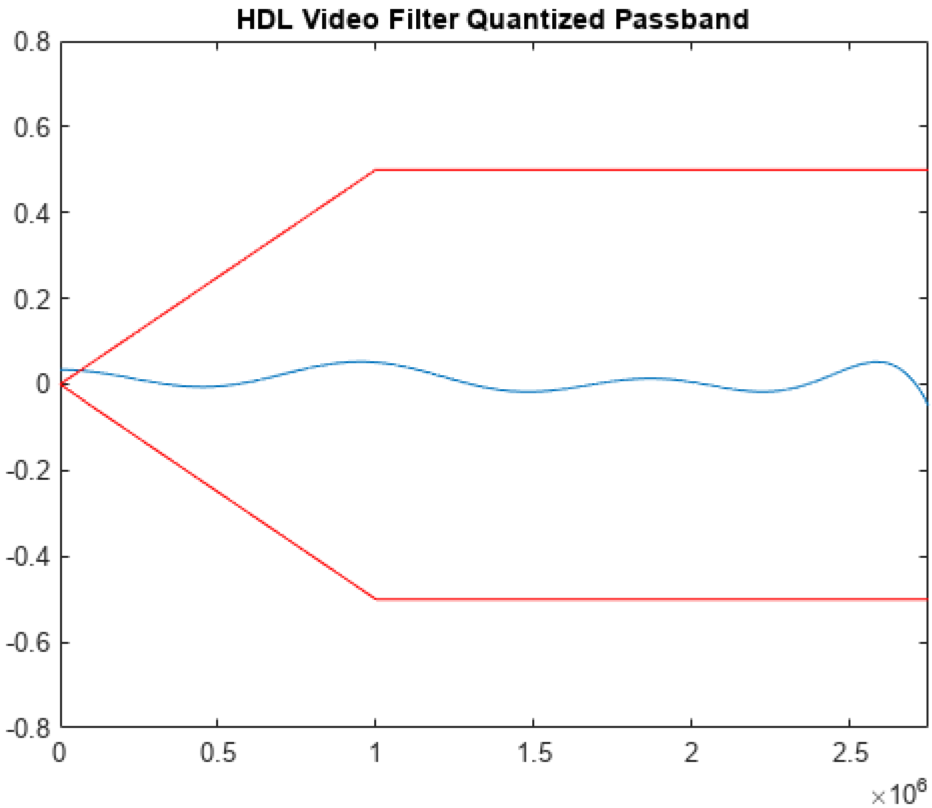
```
videoFilter = dsp.FIRFilter;
videoFilter.Numerator = b;
videoFilter.Structure = 'Direct form symmetric';

%Try 10-bit coefficients
videoFilter.CoefficientsDataType = 'Custom';
videoFilter.CustomCoefficientsDataType = numerictype(1,10);
```

Plot the Quantized Filter Response

Now examine the passband and stopband response of the quantized filter relative to the specification. Plot and check the quantized passband first.

The quantized design meets the passband specifications except at DC, where it misses the specification by about 0.035 dB.

```
f = 0:100:2.75e6;
H = freqz(videoFilter,f,13.5e6,'Arithmetic','fixed');
plot(f,20*log10(abs(H)));
title('HDL Video Filter Quantized Passband');
axis([0 2.75e6 -.8 .8]);
line(passbandrange{:}, 'Color', 'red');
```

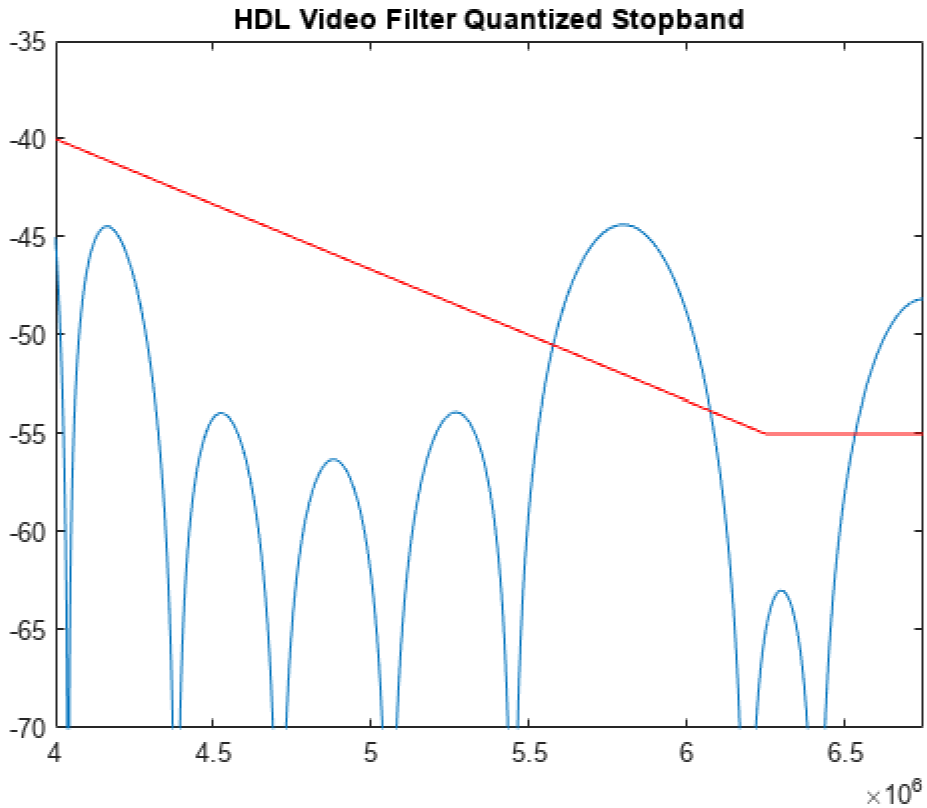


Plot the Quantized Stopband

The red lines again show a "not to exceed" limit on the stopband.

The stopband limit is violated, which indicates a problem with the quantization settings.

```
f = 4e6:100:6.75e6;
H = freqz(videoFilter,f,13.5e6,'Arithmetic','fixed');
plot(f,20*log10(abs(H)));
title('HDL Video Filter Quantized Stopband');
axis([4e6 6.75e6 -70 -35]);
line(stopbandrange{:}, 'Color', 'red');
```

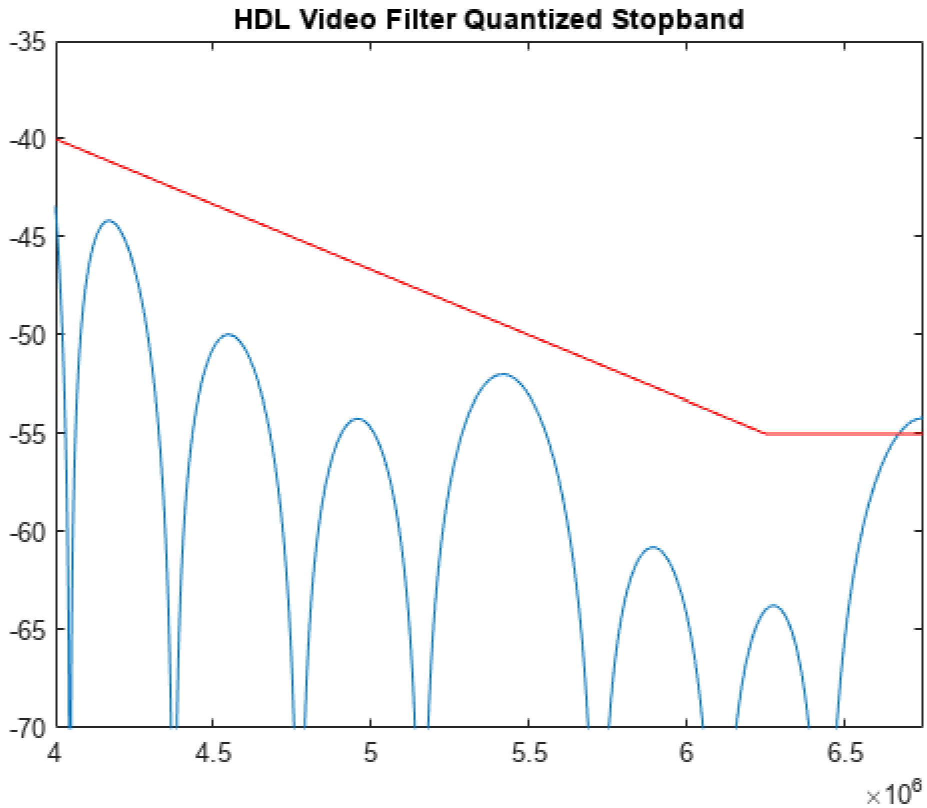


Change the Coefficient Quantizer Settings

Adding more bits to the coefficient word length enables the filter to meet the specification. Increment the word length by one and replot the stopband.

This just misses the specification at the end of the stopband. This small deviation from the specification might be acceptable if you know that some other part of your system applies a lowpass filter to this signal.

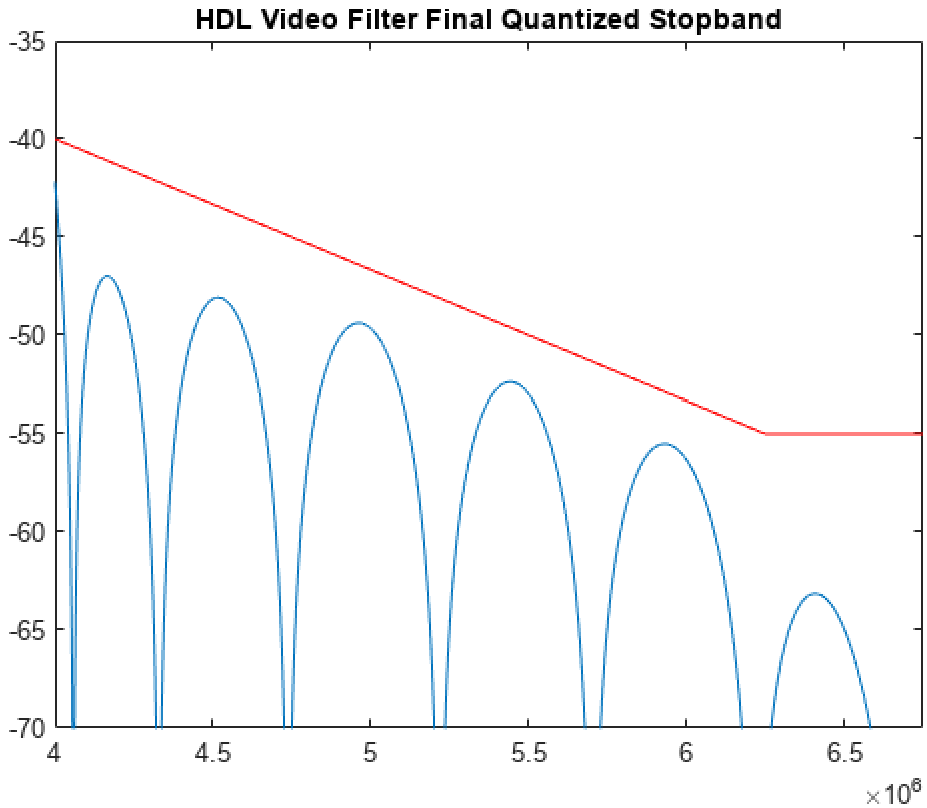
```
videoFilter.CustomCoefficientsDataType = numerictype(1,11);
f = 4e6:100:6.75e6;
H = freqz(videoFilter,f,13.5e6,'Arithmetic','fixed');
plot(f,20*log10(abs(H)));
title('HDL Video Filter Quantized Stopband');
axis([4e6 6.75e6 -70 -35]);
line(stopbandrange{:}, 'Color', 'red');
```



Set the Final Coefficient Quantizer Word Length

Add one more bit to the coefficient quantizer word length and replot the stopband. This should meet the specification.

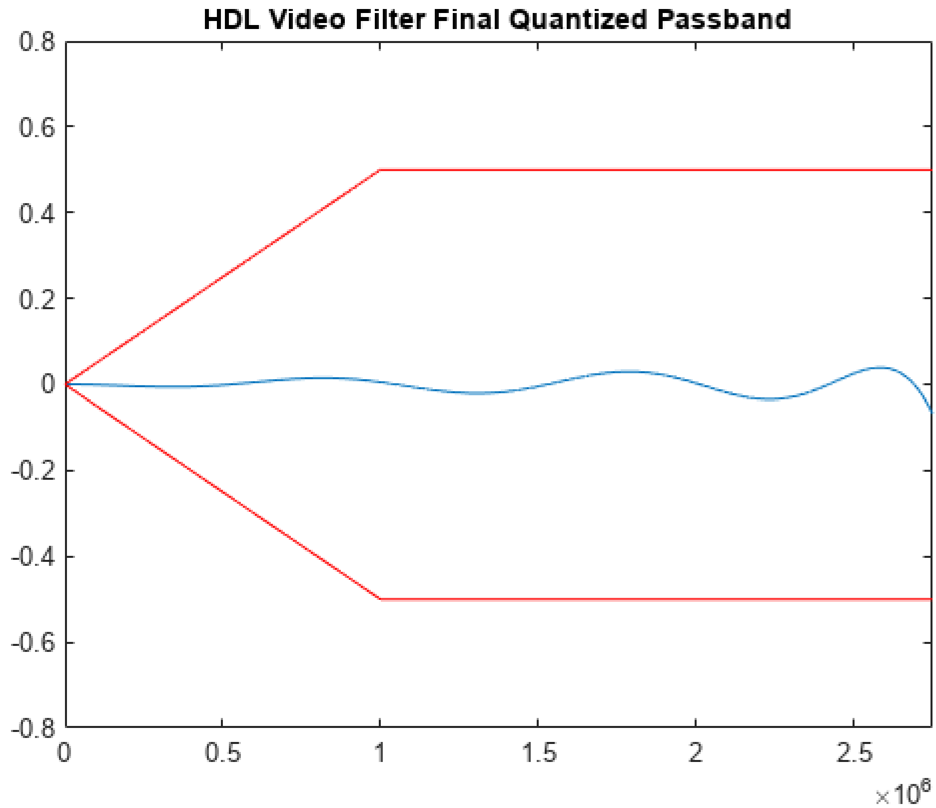
```
videoFilter.CustomCoefficientsDataType = numerictype(1,12);
f = 4e6:100:6.75e6;
H = freqz(videoFilter,f,13.5e6,'Arithmetic','fixed');
plot(f,20*log10(abs(H)));
title('HDL Video Filter Final Quantized Stopband');
axis([4e6 6.75e6 -70 -35]);
line(stopbandrange{:}, 'Color', 'red');
```



Perform a Final Check on the Passband Response

Recheck the passband to be sure the changes have improved the problems in the response near DC. The response now passes the specification.

```
f = 0:100:2.75e6;
H = freqz(videoFilter,f,13.5e6,'Arithmetic','fixed');
plot(f,20*log10(abs(H)));
title('HDL Video Filter Final Quantized Passband');
axis([0 2.75e6 -.8 .8]);
line(passbandrange{:}, 'Color', 'red');
```



Generate HDL Code and Test Bench from the Quantized Filter

Starting from the quantized filter, generate VHDL or Verilog code.

Create a temporary work directory. After generating the HDL (selecting VHDL in this case), open the generated VHDL file in the editor.

Generate a VHDL test bench to make sure that it matches the response in MATLAB exactly. Select the default input stimulus, which for FIR is impulse, step, ramp, chirp, and noise inputs.

To generate Verilog code and Verilog test bench instead, change value for 'TargetLanguage' property from 'VHDL' to 'Verilog'.

The warnings indicate that by selecting the symmetric structure for the filter and generating HDL, may result in smaller area or a higher clock rate.

Assume an input of 8-bit word length with 7-bit fractional bits.

```
workingdir = tempname;
generatehdl(videoFilter, 'Name', 'hdlvideofilt', ...
                'InputDataType' ,numerictype(1,8,7), ...
                'TargetLanguage', 'VHDL', ...
                'TargetDirectory', workingdir, ...
                'GenerateHDLTestbench', 'on');
```

```
### Starting VHDL code generation process for filter: hdlvideofilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp7cbc97b7_1222_4086_9998_e6bc953e9b81\h
```

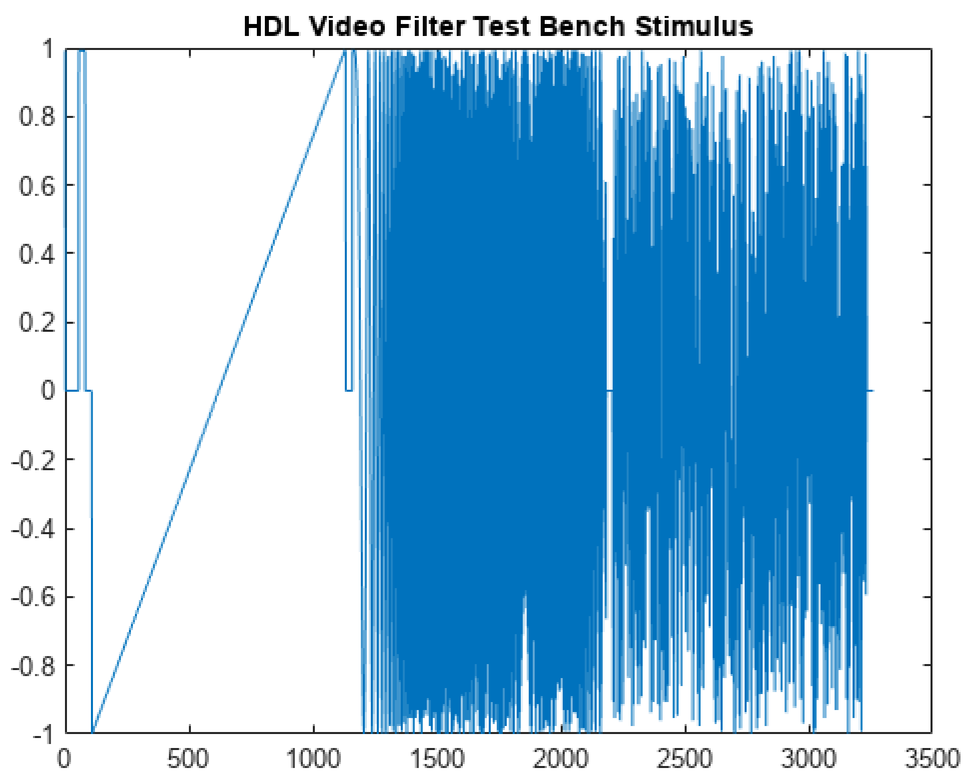
```
### Starting generation of hdlvideofilt VHDL entity
### Starting generation of hdlvideofilt VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlvideofilt
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 3261 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp7cbc97b7_1222_4086_9998_e6l
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

```
edit(fullfile(workingdir, 'hdlvideofilt.vhd'));
```

Plot the Test Bench Stimulus

Plot the default test bench stimulus used by the above command, using the generatetbstimulus function.

```
tbstim = generatetbstimulus(videoFilter,'InputDataType', numerictype(1,8,7));
plot(tbstim);
title('HDL Video Filter Test Bench Stimulus');
```



Conclusion

You designed a double precision filter to meet the ITU-R BT.601 luma filter specification and then created a FIR filter System object that also met the specification. You generated VHDL code and a VHDL test bench that functionally verified the filter.

You can use a VHDL simulator, such as ModelSim®, to verify these results. You can also experiment with Verilog. You can use many optimizations to get smaller and faster HDL results by removing the constraint that the generated HDL be exactly true to MATLAB. When you use these optimizations, the HDL test bench can check the filter response to be within a specified error margin of the MATLAB response.

HDL Digital Up-Converter (DUC)

This example illustrates how to generate HDL code for a Digital Up-Converter (DUC). A DUC is a digital circuit which converts a digital baseband signal to a passband signal. The input baseband signal is sampled at a relatively low sampling rate, typically the digital modulation symbol rate. The baseband signal is filtered and converted to a higher sampling rate before modulating a direct digitally synthesized (DDS) carrier frequency.

The input signals are passed through three filtering stages. Each stage first filters the signals with a lowpass interpolating filter and then performs a sampling rate change. The DUC in this example is a cascade of two FIR Interpolation Filters and one CIC Interpolation Filter. The first FIR Interpolation Filter is a pulse shaping FIR filter that increases the sampling rate by 2 and performs transmitter Nyquist pulse shaping. The second FIR Interpolation Filter is a compensation FIR filter that increases the sampling rate by 2 and compensates for the distortion of the following CIC filter. The CIC Interpolation Filter increases the sampling rate by 32.

The filters are implemented in fixed-point mode. The input/output word length and fraction length are specified. The internal settings of the first two filters are specified, while the internal settings of the CIC filter are calculated automatically to preserve full precision.

Create Pulse Shaping FIR Filter

Create a 32-tap FIR Interpolator with interpolation factor of 2.

```
pulseShapingFIR = dsp.FIRInterpolator;
pulseShapingFIR.InterpolationFactor = 2;
pulseShapingFIR.Numerator = [...
     0.0007    0.0021   -0.0002   -0.0025   -0.0027    0.0013    0.0049    0.0032 ...
    -0.0034   -0.0074   -0.0031    0.0060    0.0099    0.0029   -0.0089   -0.0129 ...
    -0.0032    0.0124    0.0177    0.0040   -0.0182   -0.0255   -0.0047    0.0287 ...
     0.0390    0.0049   -0.0509   -0.0699   -0.0046    0.1349    0.2776    0.3378 ...
     0.2776    0.1349   -0.0046   -0.0699   -0.0509    0.0049    0.0390    0.0287 ...
    -0.0047   -0.0255   -0.0182    0.0040    0.0177    0.0124   -0.0032   -0.0129 ...
    -0.0089    0.0029    0.0099    0.0060   -0.0031   -0.0074   -0.0034    0.0032 ...
     0.0049    0.0013   -0.0027   -0.0025   -0.0002    0.0021    0.0007 ];
```

Create Compensation Fir Filter

Create an 11-tap FIR Interpolator with interpolation factor of 2.

```
compensationFIR = dsp.FIRInterpolator;
compensationFIR.InterpolationFactor = 2;
compensationFIR.Numerator = [...
    -0.0007   -0.0009    0.0039    0.0120    0.0063   -0.0267   -0.0592   -0.0237 ...
     0.1147    0.2895    0.3701    0.2895    0.1147   -0.0237   -0.0592   -0.0267 ...
     0.0063    0.0120    0.0039   -0.0009   -0.0007];
```

Create CIC Interpolating Filter

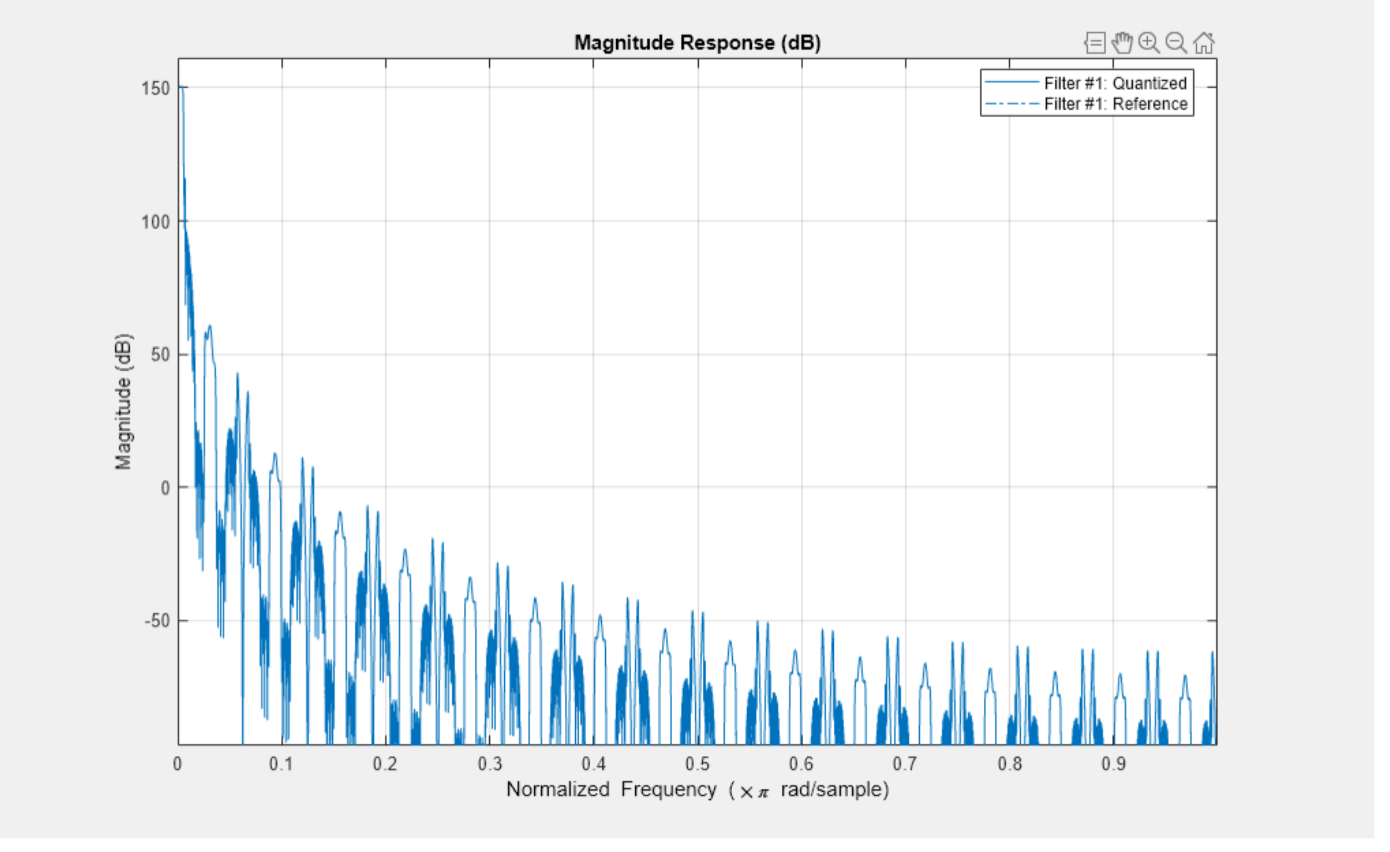
Create a 5-stage CIC Interpolator with interpolation factor of 32.

```
CICFilter = dsp.CICInterpolator;
CICFilter.InterpolationFactor = 32;
CICFilter.NumSections = 5;
```

Cascade of the Filters

Create a cascade filter including the above three filters. Check the frequency response of the cascade filter.

```
DUC = dsp.FilterCascade(pulseShapingFIR, compensationFIR, CICFilter);
fvtool(DUC);
```



Generate VHDL Code for DUC and Test Bench

Generate synthesizable and portable VHDL code for the cascade filter.

You have the option of generating a VHDL, Verilog, or ModelSim® .do file test bench to verify that the HDL design matches the MATLAB® filter.

To generate Verilog instead of VHDL, change the value of the property 'TargetLanguage', from 'VHDL' to 'Verilog'.

Generate a stimulus signal for the filter. The length of the stimulus should be greater than the total latency of the filter.

Generate a VHDL test bench to verify that the results match the MATLAB results exactly. This is done by passing another property 'GenerateHDLTestbench' and setting its value to 'on'. The stimulus to test bench is specified using the 'TestBenchUserStimulus' property.

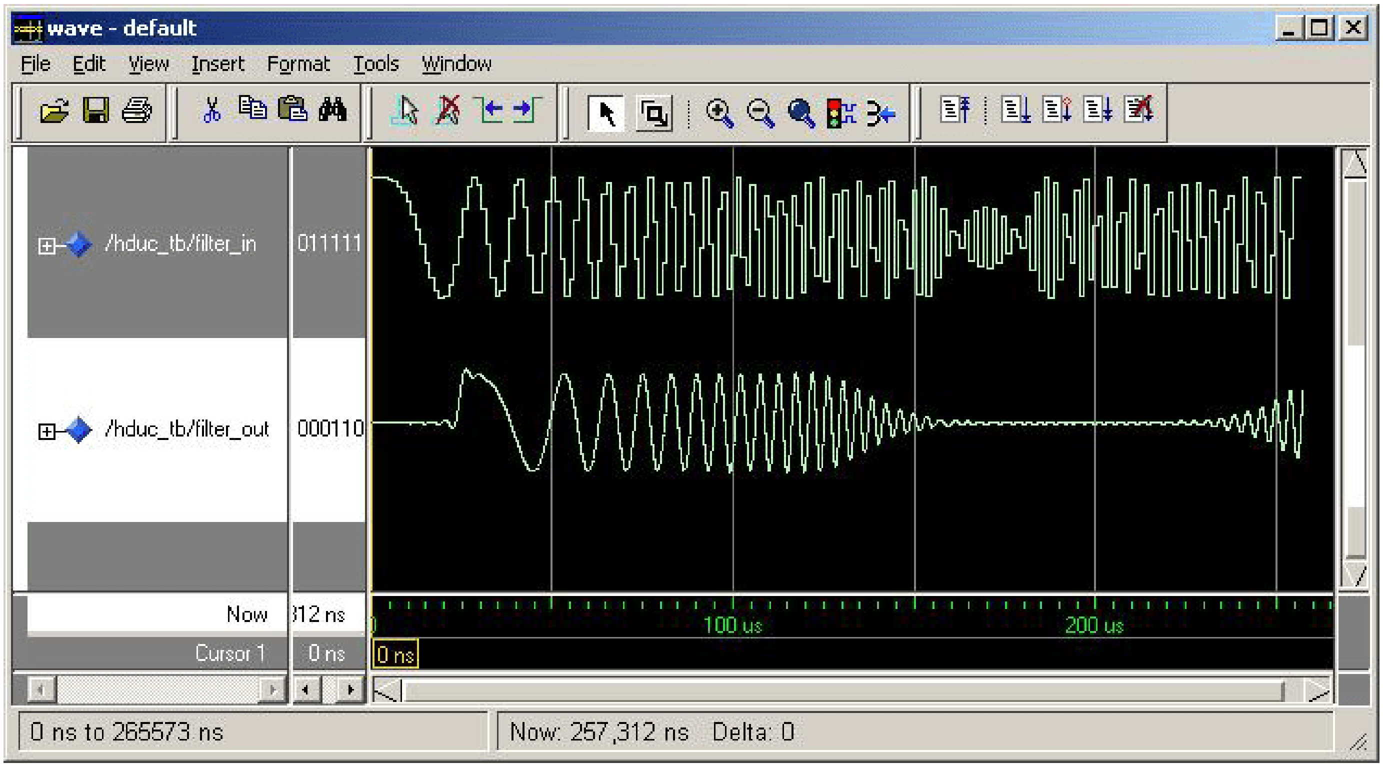
Assume 16-bit signed fixed-point input with 15 bits of fraction.

```
t = 0.005:0.005:1.5;
stim = chirp(t, 0, 1, 150);

workingdir = tempname;
generatehdl(DUC, 'Name', 'hdlduc',...
                 'TargetLanguage', 'VHDL',...
                 'TargetDirectory', workingdir, ...
                 'GenerateHDLTestbench', 'on', ...
                 'TestBenchUserStimulus', stim, ...
                 'InputDataType', numerictype(1,16,15));

### Starting VHDL code generation process for filter: hdlduc
### Cascade stage # 1
### Starting VHDL code generation process for filter: hdlduc_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa7eb6849_683d_4fb8_add2_797b5f7b5f9f\h
### Starting generation of hdlduc_stage1 VHDL entity
### Starting generation of hdlduc_stage1 VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlduc_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: hdlduc_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa7eb6849_683d_4fb8_add2_797b5f7b5f9f\h
### Starting generation of hdlduc_stage2 VHDL entity
### Starting generation of hdlduc_stage2 VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlduc_stage2
### Cascade stage # 3
### Starting VHDL code generation process for filter: hdlduc_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa7eb6849_683d_4fb8_add2_797b5f7b5f9f\h
### Starting generation of hdlduc_stage3 VHDL entity
### Starting generation of hdlduc_stage3 VHDL architecture
### Section # 1 : Comb
### Section # 2 : Comb
### Section # 3 : Comb
### Section # 4 : Comb
### Section # 5 : Comb
### Section # 6 : Integrator
### Section # 7 : Integrator
### Section # 8 : Integrator
### Section # 9 : Integrator
### Section # 10 : Integrator
### Successful completion of VHDL code generation process for filter: hdlduc_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa7eb6849_683d_4fb8_add2_797b5f7b5f9f\h
### Starting generation of hdlduc VHDL entity
### Starting generation of hdlduc VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlduc
### HDL latency is 225 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 300 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa7eb6849_683d_4fb8_add2_797
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

ModelSim® Simulation Results

The following display shows the ModelSim HDL simulator running these test benches.

DUC response to a chirp stimulus:



Conclusion

In this example, you designed three individual interpolation filters, cascaded them into a Digital-Up Converter, verified the frequency response of the filter, and called Filter Design HDL Coder™ functions to generate VHDL code for the filter and a VHDL test bench to verify the VHDL code against its MATLAB result. The simulation result of the VHDL code proved that the generated VHDL filter produced a bit-true implementation of the MATLAB filter.

HDL Fractional Delay (Farrow) Filter

This example illustrates how to generate HDL code for a fractional delay (Farrow) filter for timing recovery in a digital modem. A Farrow filter structure provides variable fractional delay for the received data stream prior to downstream symbol sampling. This special FIR filter structure permits simple handling of filter coefficients by an efficient polynomial interpolation formula implementation to provide variable fractional resampling.

In a typical digital modem application, the fractionally resampled data output from a Farrow filter is passed along to a symbol sampler with optional carrier recovery. For more details of this complete application please refer to "Timing Recovery Using Fixed-Rate Resampling" for Simulink® and Communications Toolbox™.

Design the Filter

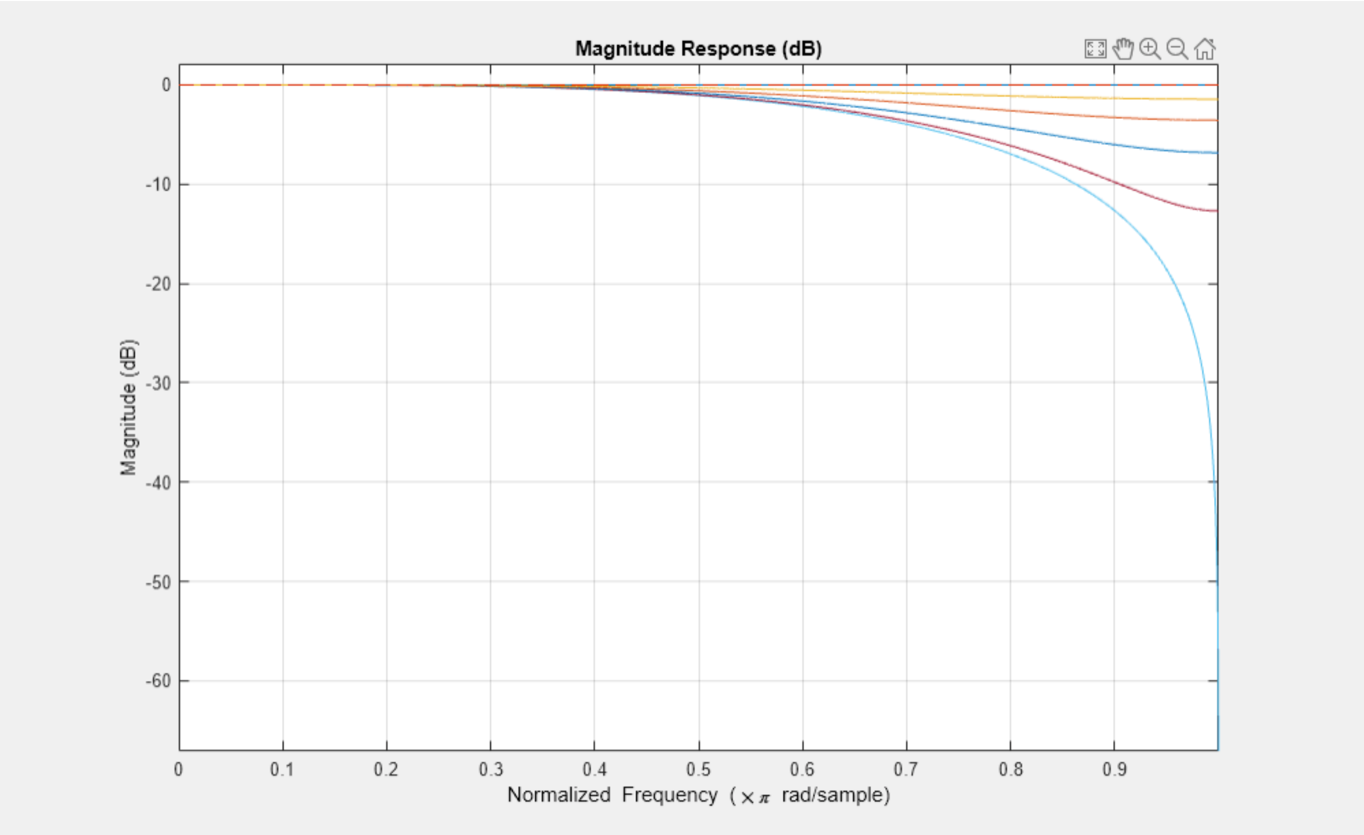
To design a fractional delay filter using the Cubic Lagrange interpolation method, first create a specification object with filter order `3` and an arbitrary fractional delay of `0.3`. Next, create a farrow filter object `Hd`, using the design method of the specification object with argument `lagrange`. This method is also called with property `FilterStructure` and its value `fd`. You can look into the details of the filter object `Hd` by using the `info` command.

```
fDelay = 0.3;
filtdes = fdesign.fracdelay(fDelay, 'N', 3);
Hd = design(filtdes,'lagrange', 'FilterStructure', 'farrowfd');
info(Hd)
```

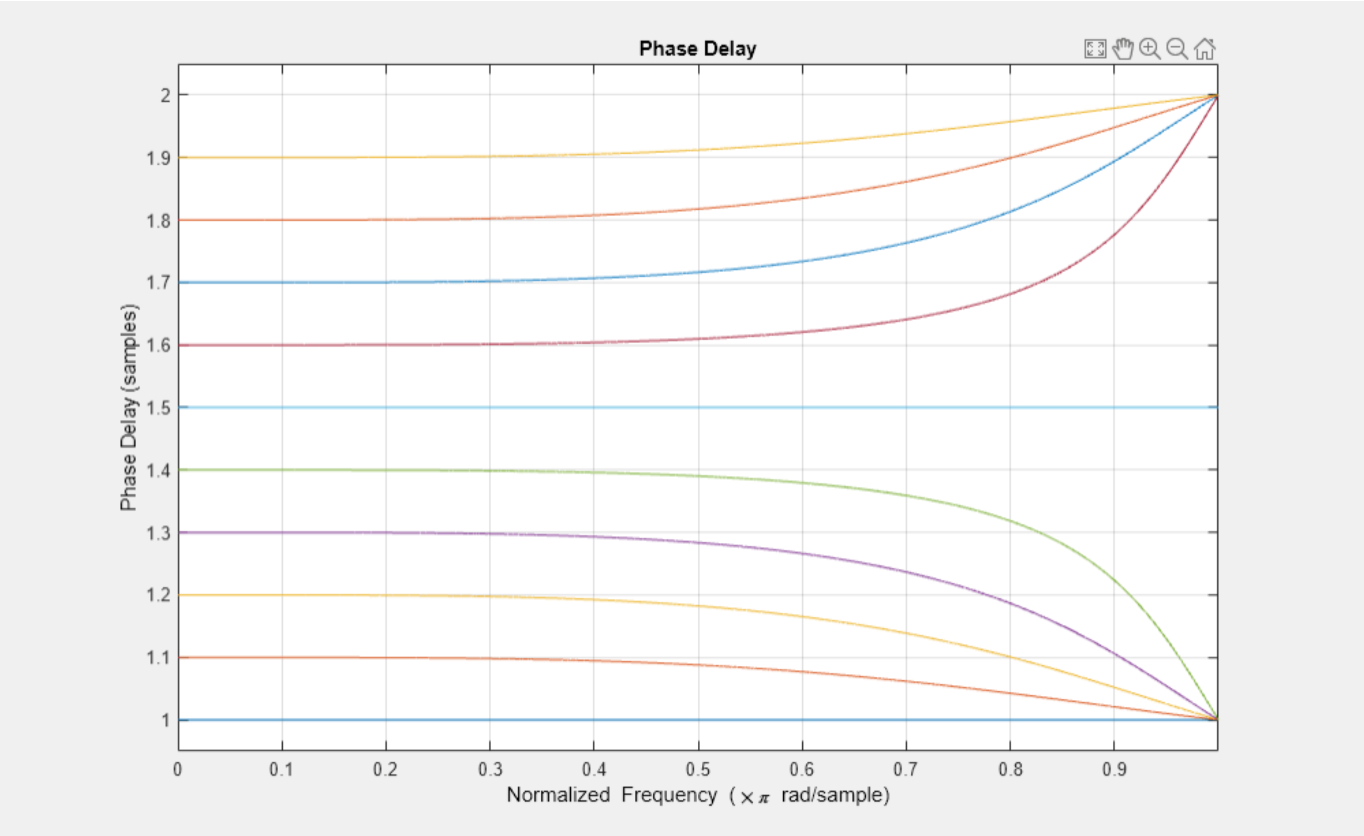
```
Discrete-Time FIR Farrow Filter (real)
--------------------------------------
Filter Structure  : Farrow Fractional Delay
Filter Length     : 4
Stable            : Yes
Linear Phase      : No

Arithmetic        : double
```

The fractional delay for the Farrow filter is tunable and can be altered to lead a different magnitude response. You can see this by creating a set of filters that are copies of the pre-designed filter `Hd` with each differing in their `fracdelay` values.

```
h = repmat(Hd,1,10); % preallocating size
for d=0:9
    h(d+1) = copy(Hd); % create unique filter objects
    h(d+1).fracdelay = d/10;
end
fvtool(h)
```




```
fvtool(h,'Analysis','PhaseDelay')
```



Quantize the Filter

Set the filter object to fixed-point mode to quantize it. Assume eight-bit input data with eight-bit coefficients and six-bit fractional delay. Modify the fixed-point data word lengths and fraction lengths accordingly. The `CoeffFracLength` property is set automatically because coefficient autoscaling is set to `on` by default by the property `CoeffAutoscale`. Switch off `FDAutoScale` and set `FDFracLength` to six, allowing a fractional delay in the range 0 to 1 to be represented.

```
Hd.arithmetic          = 'fixed';
Hd.InputWordLength     = 8;
Hd.InputFracLength     = 7;
Hd.CoeffWordLength     = 8;
Hd.FDWordLength        = 6;
Hd.FDAutoScale         = false;
Hd.FDFracLength        = 6;
```

Generate HDL Code from the Quantized Filter

Starting with the correctly quantized filter, you can generate VHDL or Verilog code using the `generatehdl` command. You create a temporary work directory and then use the `generatehdl` command using the appropriate property-value pairs. After generating the HDL code using `VHDL` for the `TargetLanguage` property in this case, you can open the generated VHDL file in the editor by clicking on the hyperlink displayed in the command line display messages.

```
workingdir = tempname;
generatehdl(Hd, 'Name', 'hdlfarrow', ...
```

```
                'TargetLanguage', 'VHDL',...
                'TargetDirectory', workingdir);
```

```
### Starting VHDL code generation process for filter: hdlfarrow
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1efc68c5a3\
### Starting generation of hdlfarrow VHDL entity
### Starting generation of hdlfarrow VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlfarrow
### HDL latency is 2 samples
```

Generate HDL Test Bench

To verify the HDL code, you can generate an HDL test bench to simulate the HDL code using an HDL simulator. The test bench will verify the results of the HDL code with the results of the MATLAB® `filter` command. The stimuli for the filter input `filter_in` port and fractional delay `filter_fd` port can be specified using properties `TestbenchStimulus`, `TestbenchUserStimulus` and `TestbenchFracDelayStimulus`.

Predefined stimulus for the filter input `filter_in` port can be specified for input data stimulus using the property `TestbenchStimulus` as with the other filter structures. You can specify your own stimulus for the input data by using the property `TestbenchUserStimulus` and passing a MATLAB vector as the value.

You can specify the fractional delay stimulus using the property `TestbenchFracdelayStimulus`. A vector of double between `0` and `1` is generated automatically by specifying either `RandSweep` or `RampSweep`. The default behavior is to provide a fractional delay stimulus of a constant set to the `fracdelay` value of the filter object.

The following command specifies the input stimulus to `chirp`, and the fractional delay vector is set to a constant `0.3` for all the simulation time. This is the default behavior when the `TestbenchFracDelayStimulus` property is not set otherwise.

```
generatehdl(Hd, 'Name', 'hdlfarrow', ...
    'GenerateHDLTestbench', 'on', ...
    'TestBenchName', 'hdlfarrow_default_tb',...
    'TargetLanguage', 'VHDL',...
    'TargetDirectory', workingdir);
```

```
### Starting VHDL code generation process for filter: hdlfarrow
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1efc68c5a3\
### Starting generation of hdlfarrow VHDL entity
### Starting generation of hdlfarrow VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlfarrow
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 3100 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

To automatically generate the test vector for a fractional delay port, specify `RampSweep` for `TestBenchFracDelayStimulus`. It generates a vector of values between `0` and `1` sweeping in a linear fashion. The length of this vector is equal to the input stimulus vector.

```
generatehdl(Hd, 'Name', 'hdlfarrow', ...
    'GenerateHDLTestbench', 'on', ...
```

```
    'TestBenchName', 'hdlfarrow_rampsweep_tb',...
    'TargetLanguage', 'VHDL',...
    'TestBenchStimulus', 'chirp',...
    'TestbenchFracDelaystimulus', 'Rampsweep', ...
    'TargetDirectory', workingdir);
```

```
### Starting VHDL code generation process for filter: hdlfarrow
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1efc68c5a3\h
### Starting generation of hdlfarrow VHDL entity
### Starting generation of hdlfarrow VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlfarrow
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 1028 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

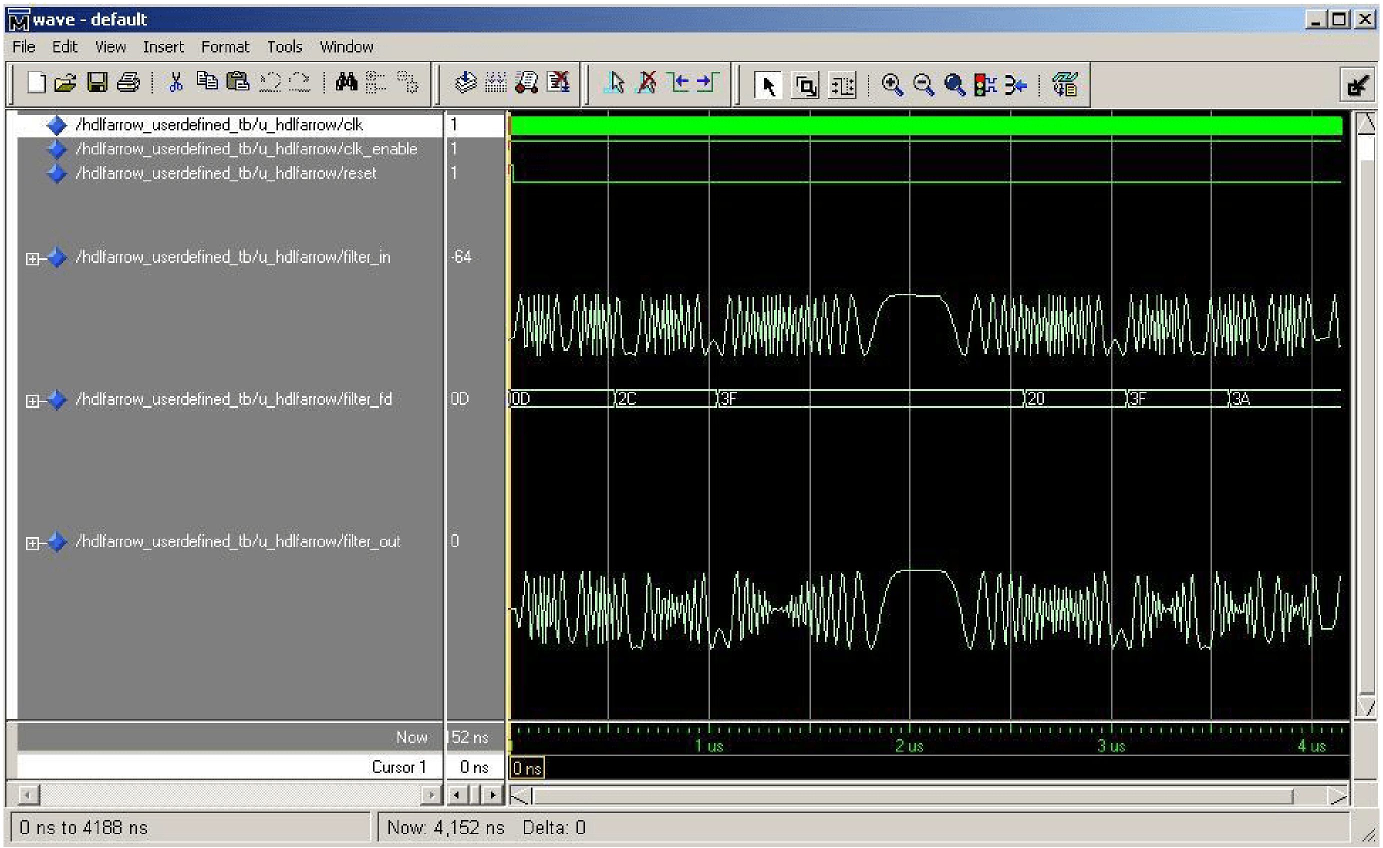
You can generate a customized input stimulus vector using MATLAB commands and pass it to the test bench stimulus properties for the user defined input and `fracdelay` stimuli. An input test vector `userinputstim` is generated using the `chirp` command and the fractional delay test vector `userfdstim` is generated of length equal to the input test vector.

```
t=-2:0.01:2;                               % +/-2 secs @ 100 Hz sample rate
userinputstim = chirp(t,100,1,200,'q'); % Start @100Hz, cross 200Hz at t=1sec
leninput = length(userinputstim);
samplefdvalues = [0.1, 0.34, 0.78, 0.56, 0.93, 0.25, 0.68, 0.45];
samplesheld = ceil(leninput/length(samplefdvalues));
ix = 1;
for n = 1:length(samplefdvalues)-1
    userfdstim(ix: ix + samplesheld-1) =  repmat(samplefdvalues(n),1, samplesheld);
    ix = ix + samplesheld;
end
userfdstim(ix:leninput)= repmat(samplefdvalues(end),1 , leninput-length(userfdstim));

generatehdl(Hd, 'Name', 'hdlfarrow', ...
    'GenerateHDLTestbench', 'on', ...
    'TestBenchName', 'hdlfarrow_userdefined_tb',...
    'TargetLanguage', 'VHDL',...
    'TestBenchUserStimulus', userinputstim,...
    'TestbenchFracDelaystimulus', userfdstim, ...
    'TargetDirectory', workingdir);
```

```
### Starting VHDL code generation process for filter: hdlfarrow
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1efc68c5a3\h
### Starting generation of hdlfarrow VHDL entity
### Starting generation of hdlfarrow VHDL architecture
### Successful completion of VHDL code generation process for filter: hdlfarrow
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 401 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpaff7e366_b07b_4f62_820c_6e1
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

ModelSim® Simulation Results

The following display shows the ModelSim® HDL simulator after running the VHDL test bench.



Conclusion

In this example, we showed how you can design a double-precision fractional delay filter to meet the given specifications. We also showed how you can quantize the filter and generate VHDL code. Then we showed how you can generate VHDL test benches using several options to specify the input and fracdelay stimulus vector.

You can use any HDL simulator to verify these results. You can also experiment with Verilog for both filters and test benches.

HDL Sample Rate Conversion Using Farrow Filters

This example shows how you can design and implement hardware efficient sample rate converters for an arbitrary factor using polynomial-based (Farrow) structures. Sample rate conversion (SRC) between arbitrary factors is useful for many applications including symbol synchronizations in digital receivers, speech coding, audio sampling, etc. This example shows how you can convert the sampling rate of an audio signal from 8kHz to 44.1 kHz.

Overview

To resample the incoming signal from `8 kHz` to `44.1 kHz`, you must interpolate by 441 and decimate by 80. This SRC can be implemented using polyphase structures. However using the polyphase structures for any arbitrary factor usually results in large number of coefficients leading to a lot of memory requirement and area. This example shows how to efficiently implement SRC with a mix of polyphase and Farrow filter structures.

Design of Interpolating Stages

First, interpolate the original `8 kHz` signal by a factor of `4` using a cascade of FIR halfband filters. This interpolation results in an intermediate signal of `32 kHz`. Polyphase filters are particularly well adapted for interpolation or decimation by an integer factor and for fractional rate conversions when the interpolation and the decimation factors are low. Design the interpolating stages for the specifications given.

```
L     = 4;                     % Interpolation factor
FsOut = 32e3;                  % Output sample rate
TW    = 0.125 * 2 * FsOut / L; % Transition width (Hz)
Astop = 50;                    % Minimum stopband attenuation (dB)

FIRCascade = designMultistageInterpolator(L,FsOut,TW,Astop,'CostMethod','design');
```

Design of Farrow Filter

The signal output from the above interpolating stages needs to be further interpolated from `32 kHz` to `44.1 kHz`. This operation is done by a Farrow rate converter filter designed with a cubic Lagrange polynomial.

```
FsInp = 32e3;   % Input sample rate
FsOut = 44.1e3; % Output sample rate
TOL   = 0;      % Output rate tolerance
NP    = 3;      % Polynomial order

farrowFilter = dsp.FarrowRateConverter(FsInp, FsOut, TOL, NP);
```

To prevent the datapath from growing to very large word lengths, quantize the filter stages such that the inputs to each stage are 12 bits and the outputs are 12 bits.

```
cascadeOutNT = numerictype([],12,11);
FIRCascade.Stage1.FullPrecisionOverride = false;
FIRCascade.Stage1.OutputDataType        = 'Custom';
```

```
FIRCascade.Stage1.CustomOutputDataType  = cascadeOutNT;
FIRCascade.Stage2.FullPrecisionOverride = false;
FIRCascade.Stage2.OutputDataType        = 'Custom';
FIRCascade.Stage2.CustomOutputDataType  = cascadeOutNT;

farrowOutNT = numerictype(1,12,11);
farrowFilter.OutputDataType = farrowOutNT;
```

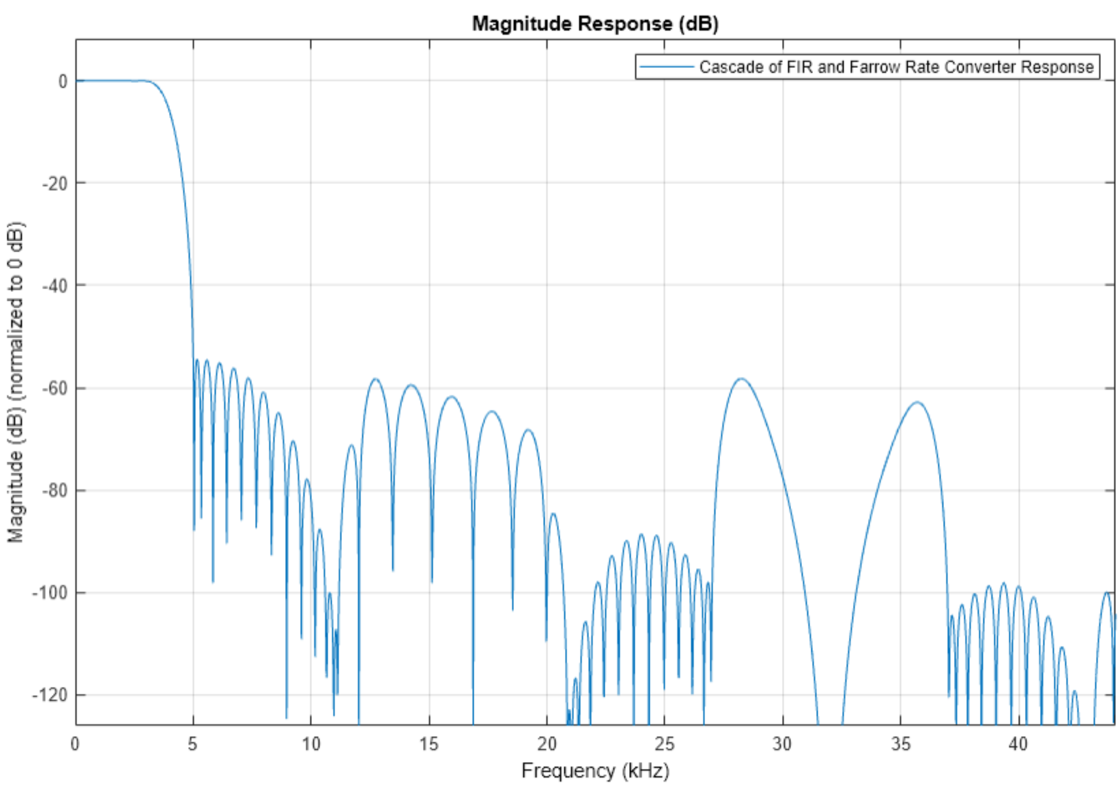
Cascade of Complete SRC and Magnitude Response

The overall filter is obtained by creating a cascade of the interpolating stages and the Farrow filter.

```
sampleRateConverter = cascade(FIRCascade.Stage1, FIRCascade.Stage2, farrowFilter);
```

The magnitude response of the cascaded SRC filter shows that it meets the 50 dB minimum stopband attenuation specification.

```
Fs = 32e3*441;                  % The highest clock rate is 14.112 MHz
W = linspace(0,44.1e3,2048);  % Define the frequency range analysis

fvt = fvtool(sampleRateConverter,'FrequencyRange','Specify freq. vector', ...
    'FrequencyVector',W,'Fs',Fs, ...
    'NormalizeMagnitudeto1','on', 'Color', 'white');

legend(fvt,'Cascade of FIR and Farrow Rate Converter Response',...
    'Location','NorthEast')
```



Generate HDL & Test Bench

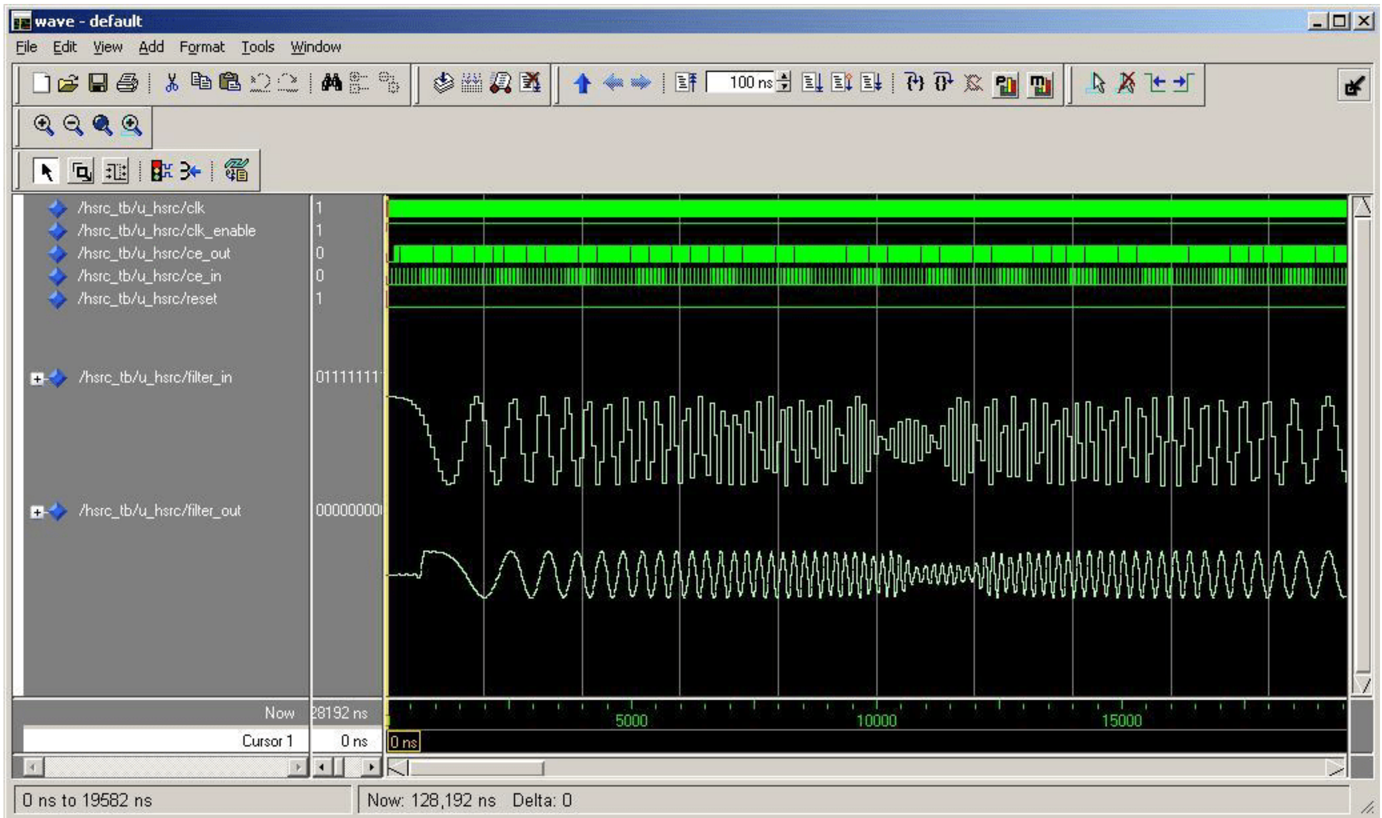
You can now generate VHDL code for the cascaded SRC using the `generatehdl` command. You can also generate the VHDL test bench by passing the `'TestBenchUserStimulus'` and `'GenerateHDLTestbench'` properties into the `generatehdl` command. The VHDL code can be simulated in any HDL simulator to verify the results.

```
workingdir = tempname;
inpFrameSz = 640;
tVector    = linspace(0.005, 7.5, inpFrameSz);
srcTBStim  = (chirp(tVector, 0, 1, 150))';

generatehdl(sampleRateConverter, ...
    'TargetDirectory',       workingdir, ...
    'InputDataType',         numerictype(1,12,11), ...
    'OptimizeForHDL',        'on', ...
    'GenerateHDLTestbench',  'on', ...
    'TestBenchUserStimulus', srcTBStim, ...
    'ErrorMargin',           2);
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa1204d4a_923f_4ed2_bcd9_4bc09578f42e\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa1204d4a_923f_4ed2_bcd9_4bc09578f42e\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Cascade stage # 3
### Starting VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa1204d4a_923f_4ed2_bcd9_4bc09578f42e\c
### Starting generation of casfilt_stage3 VHDL entity
### Starting generation of casfilt_stage3 VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa1204d4a_923f_4ed2_bcd9_4bc09578f42e\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 1325 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 640 samples.

Warning: HDL optimization may cause small numeric differences that will be flagged as errors when

### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tpa1204d4a_923f_4ed2_bcd9_4bc
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

ModelSim® Simulation Results

The following display shows the ModelSim® HDL simulator results after running the VHDL test bench.



Conclusion

This example showed how you can design a sample rate converter using a Farrow structure, and how to analyze the response, quantize it, and generate bit-accurate VHDL code and test bench.

HDL Serial Architectures for FIR Filters

This example illustrates how to generate HDL code for a symmetrical FIR filter with fully parallel, fully serial, partly serial, and cascade-serial architectures for a lowpass filter for an audio filtering application.

Design the Filter

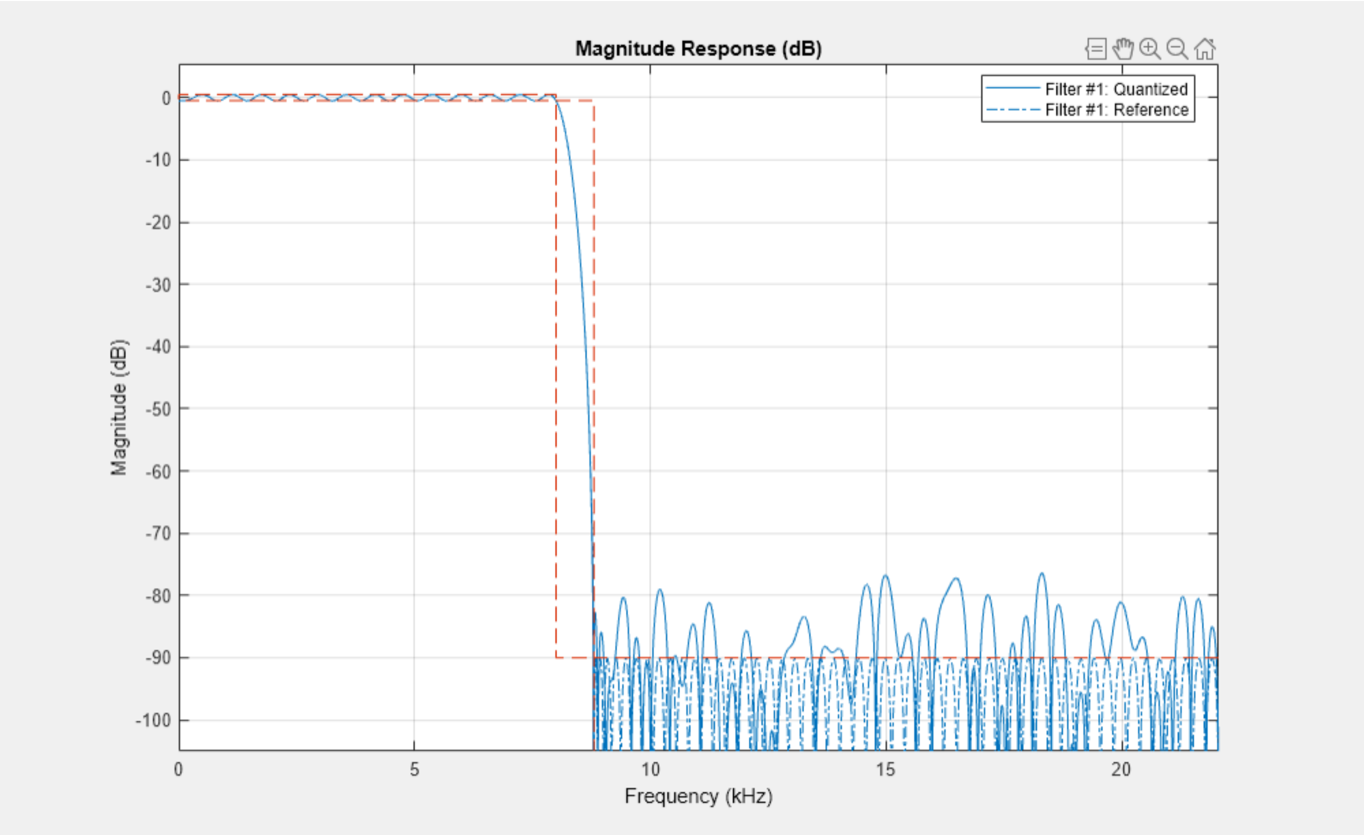
Use an audio sampling rate of 44.1 kHz and a passband edge frequency of 8.0 kHz. Set the allowable peak-to-peak passband ripple to 1 dB and the stopband attenuation to -90 dB. Then, design the filter using `fdesign.lowpass`, and create the FIR filter System object™ using the `'equiripple'` method with the `'Direct form symmetric'` structure.

```
Fs           = 44.1e3;         % Sampling Frequency in Hz
Fpass        = 8e3;            % Passband Frequency in Hz
Fstop        = 8.8e3;          % Stopband Frequency in Hz
Apass        = 1;              % Passband Ripple in dB
Astop        = 90;             % Stopband Attenuation in dB

fdes = fdesign.lowpass('Fp,Fst,Ap,Ast',...
    Fpass, Fstop, Apass, Astop, Fs);
lpFilter = design(fdes,'equiripple', 'FilterStructure', 'dfsymfir', ...
    'SystemObject', true);
```

Quantize the Filter

Assume that the input for the audio filter comes from a 12 bit ADC and output is a 12 bit DAC.

```
nt_in = numerictype(1,12,11);
nt_out = nt_in;
lpFilter.FullPrecisionOverride = false;
lpFilter.CoefficientsDataType = 'Custom';
lpFilter.CustomCoefficientsDataType = numerictype(1,16,16);
lpFilter.OutputDataType = 'Custom';
lpFilter.CustomOutputDataType = nt_out;

% Check the response with fvtool.
fvtool(lpFilter,'Fs',Fs, 'Arithmetic', 'fixed');
```



Generate Fully Parallel HDL Code from the Quantized Filter

Starting with the correctly quantized filter, generate VHDL® or Verilog® code. Create a temporary work directory. After generating the HDL code (selecting VHDL in this case), open the generated VHDL file in the editor by clicking on the link displayed in the command line display messages.

The default settings generate a fully parallel architecture. There is a dedicated multiplier for each filter tap in direct form FIR filter structure and one for every two symmetric taps in symmetric FIR structure. This results in a lot of chip area (78 multipliers, in this example). You can implement the filter in a variety of serial architectures to obtain the desired speed/area trade-off. These architecture options are shown in further sections of this example.

```
workingdir = tempname;
% fully parallel (default)
generatehdl(lpFilter, 'Name', 'fullyparallel', ...
            'TargetLanguage', 'VHDL', ...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: fullyparallel
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\
### Starting generation of fullyparallel VHDL entity
### Starting generation of fullyparallel VHDL architecture
### Successful completion of VHDL code generation process for filter: fullyparallel
### HDL latency is 2 samples
```

Generate a Test Bench from the Quantized Filter

Generate a VHDL test bench to make sure that the result matches the response you see in MATLAB® exactly. The generated VHDL code and VHDL test bench can be compiled and simulated using a simulator.

Generate DTMF tones to be used as test stimulus for the filter. A DTMF signal consists of the sum of two sinusoids - or tones - with frequencies taken from two mutually exclusive groups. Each pair of tones contains one frequency of the low group (697 Hz, 770 Hz, 852 Hz, 941 Hz) and one frequency of the high group (1209 Hz, 1336 Hz, 1477Hz) and represents a unique symbol. This code generates all the DTMF signals and uses only one of them (digit 1 here) for test stimulus. This choice keeps the length of test stimulus to reasonable limit.

```
symbol = {'1','2','3','4','5','6','7','8','9','*','0','#'};

lfg = [697 770 852 941]; % Low frequency group
hfg = [1209 1336 1477];  % High frequency group

% Generate a matrix containing all possible combinations of high and low
% frequencies, where each column represents one combination.
f  = zeros(2,12);
for c=1:4
    for r=1:3
        f(:,3*(c-1)+r) = [lfg(c); hfg(r)];
    end
end
```

Next, let's generate the DTMF tones

```
Fs  = 8000;       % Sampling frequency 8 kHz
N = 800;          % Tones of 100 ms
t   = (0:N-1)/Fs; % 800 samples at Fs
pit = 2*pi*t;

tones = zeros(N,size(f,2));
for toneChoice=1:12
    % Generate tone
    tones(:,toneChoice) = sum(sin(f(:,toneChoice)*pit))';
end

% Taking the tone for digit '1' for test stimulus.
userstim = tones(:,1);

generatehdl(lpFilter, 'Name', 'fullyparallel',...
            'GenerateHDLTestbench','on', ...
            'TestBenchUserStimulus', userstim,...
            'TargetLanguage', 'VHDL',...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: fullyparallel
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\
### Starting generation of fullyparallel VHDL entity
### Starting generation of fullyparallel VHDL architecture
### Successful completion of VHDL code generation process for filter: fullyparallel
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
```

```
### Generating input stimulus
### Done generating input stimulus; length 800 samples.

Warning: Wrap on overflow detected. This originated from 'DiscreteFir'
Suggested Actions:
    • Suppress future instances of this diagnostic from this source. - Suppress


### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

Information Regarding Serial Architectures

Serial architectures present a variety of ways to share the hardware resources at the expense of increasing the clock rate with respect to the sample rate. In FIR filters, we will share the multipliers between the inputs of each serial partition. This will have an effect of increasing the clock rate by a factor known as folding factor.

You can use the `hdlfilterserialinfo` function to get information regarding various filter lengths based on the value of coefficients. This function also displays an exhaustive table of possible options to specify SerialPartition property with corresponding values of folding factor and number of multipliers.

```
hdlfilterserialinfo(lpFilter, 'InputDataType', nt_in);
```

```
   | Total Coefficients | Zeros | A/Symm | Effective |
   ---------------------------------------------------
   |        156         |   0   |   78   |     78    |

Effective filter length for SerialPartition value is 78.

  Table of 'SerialPartition' values with corresponding values of
  folding factor and number of multipliers for the given filter.

   | Folding Factor | Multipliers |   SerialPartition   |
   ------------------------------------------------------
   |        1       |      78     |ones(1,78)           |
   |        2       |      39     |ones(1,39)*2         |
   |        3       |      26     |ones(1,26)*3         |
   |        4       |      20     |[ones(1,19)*4, 2]    |
   |        5       |      16     |[ones(1,15)*5, 3]    |
   |        6       |      13     |ones(1,13)*6         |
   |        7       |      12     |[ones(1,11)*7, 1]    |
   |        8       |      10     |[ones(1,9)*8, 6]     |
   |        9       |      9      |[9 9 9 9 9 9 9 9 6]  |
   |       10       |      8      |[ones(1,7)*10, 8]    |
   |       11       |      8      |[ones(1,7)*11, 1]    |
   |       12       |      7      |[ones(1,6)*12, 6]    |
   |       13       |      6      |[13 13 13 13 13 13]  |
   |       14       |      6      |[14 14 14 14 14 8]   |
   |       15       |      6      |[15 15 15 15 15 3]   |
   |       16       |      5      |[16 16 16 16 14]     |
   |       17       |      5      |[17 17 17 17 10]     |
   |       18       |      5      |[18 18 18 18 6]      |
   |       19       |      5      |[19 19 19 19 2]      |
   |       20       |      4      |[20 20 20 18]        |
   |       21       |      4      |[21 21 21 15]        |
```

```
|       22        |      4      |[22 22 22 12]        |
|       23        |      4      |[23 23 23 9]         |
|       24        |      4      |[24 24 24 6]         |
|       25        |      4      |[25 25 25 3]         |
|       26        |      3      |[26 26 26]           |
|       27        |      3      |[27 27 24]           |
|       28        |      3      |[28 28 22]           |
|       29        |      3      |[29 29 20]           |
|       30        |      3      |[30 30 18]           |
|       31        |      3      |[31 31 16]           |
|       32        |      3      |[32 32 14]           |
|       33        |      3      |[33 33 12]           |
|       34        |      3      |[34 34 10]           |
|       35        |      3      |[35 35 8]            |
|       36        |      3      |[36 36 6]            |
|       37        |      3      |[37 37 4]            |
|       38        |      3      |[38 38 2]            |
|       39        |      2      |[39 39]              |
|       40        |      2      |[40 38]              |
|       41        |      2      |[41 37]              |
|       42        |      2      |[42 36]              |
|       43        |      2      |[43 35]              |
|       44        |      2      |[44 34]              |
|       45        |      2      |[45 33]              |
|       46        |      2      |[46 32]              |
|       47        |      2      |[47 31]              |
|       48        |      2      |[48 30]              |
|       49        |      2      |[49 29]              |
|       50        |      2      |[50 28]              |
|       51        |      2      |[51 27]              |
|       52        |      2      |[52 26]              |
|       53        |      2      |[53 25]              |
|       54        |      2      |[54 24]              |
|       55        |      2      |[55 23]              |
|       56        |      2      |[56 22]              |
|       57        |      2      |[57 21]              |
|       58        |      2      |[58 20]              |
|       59        |      2      |[59 19]              |
|       60        |      2      |[60 18]              |
|       61        |      2      |[61 17]              |
|       62        |      2      |[62 16]              |
|       63        |      2      |[63 15]              |
|       64        |      2      |[64 14]              |
|       65        |      2      |[65 13]              |
|       66        |      2      |[66 12]              |
|       67        |      2      |[67 11]              |
|       68        |      2      |[68 10]              |
|       69        |      2      |[69 9]               |
|       70        |      2      |[70 8]               |
|       71        |      2      |[71 7]               |
|       72        |      2      |[72 6]               |
|       73        |      2      |[73 5]               |
|       74        |      2      |[74 4]               |
|       75        |      2      |[75 3]               |
|       76        |      2      |[76 2]               |
|       77        |      2      |[77 1]               |
|       78        |      1      |[78]                 |
```

You can use the optional properties 'Multipliers' and 'FoldingFactor' to display the specific information.

```
hdlfilterserialinfo(lpFilter, 'Multipliers', 4, ...
                    'InputDataType', nt_in);
```

```
Serial Partition: [20 20 20 18], Folding Factor:   20, Multipliers:    4
```

```
hdlfilterserialinfo(lpFilter, 'Foldingfactor', 6, ...
                    'InputDataType', nt_in);
```

```
Serial Partition: ones(1,13)*6, Folding Factor:    6, Multipliers:   13
```

Fully Serial Architecture

In fully serial architecture, instead of having a dedicated multiplier for each tap, the input sample for each tap is selected serially and is multiplied with the corresponding coefficient. For symmetric (and antisymmetrical) structures the input samples corresponding to each set of symmetric taps are preadded (for symmetric) or pre-subtracted (for anti-symmetric) before multiplication with the corresponding coefficients. The product is accumulated sequentially using a register and the final result is stored in a register before the next set of input samples arrive. This implementation needs a clock rate that is as many times faster than input sample rate as the number of products to be computed. This results in reducing the required chip area as the implementation involves just one multiplier with a few additional logic elements like multiplexers and registers. The clock rate will be 78 times the input sample rate (foldingfactor of 78) equal to 3.4398 MHz for this example.

To implement fully serial architecture, use the `hdlfilterserialinfo` function and set the 'Multipliers' property to 1. You can also set the 'SerialPartition' property equal to the effective filter length, which in this case is 78. The function also returns the folding factor and number of multipliers used for that serial partition setting.

```
[spart, foldingfact, nMults] = hdlfilterserialinfo(lpFilter, 'Multipliers', 1, ...
                                    'InputDataType', nt_in); %#ok<ASGLU>

generatehdl(lpFilter,'Name', 'fullyserial', ...
           'SerialPartition', spart, ...
           'TargetLanguage', 'VHDL', ...
           'TargetDirectory', workingdir, ...
           'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: fullyserial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\
### Starting generation of fullyserial VHDL entity
### Starting generation of fullyserial VHDL architecture
### Clock rate is 78 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: fullyserial
### HDL latency is 3 samples
```

Generate the test bench the same way, as in the fully parallel case. It is important to generate a test bench again for each architecture implementation.

Partly Serial Architecture

Fully parallel and fully serial represent two extremes of implementations. While Fully serial is very low area, it inherently needs a faster clock rate to operate. Fully parallel takes a lot of chip area but has very good performance. Partly serial architecture covers all the cases that lie between these two extremes.

The input taps are divided into sets. Each set is processed in parallel by a serial partition consisting of multiply accumulate and a multiplexer. Here, a set of serial partitions process a given set of taps. These serial partitions operate in parallel with respect to each other but process each tap sequentially to accumulate the result corresponding to the taps served. Finally, the result of each serial partition is added together using adders.

Partly Serial Architecture for Resource Constraint

Let us assume that you want to implement this filter on an FPGA which has only 4 multipliers available for the filter. You can implement the filter using 4 serial partitions, each using one multiply accumulate circuit.

```
hdlfilterserialinfo(lpFilter, 'Multipliers', 4, ...
                    'InputDataType', nt_in);
```

```
Serial Partition: [20 20 20 18], Folding Factor:   20, Multipliers:    4
```

The input taps that are processed by these serial partitions will be [20 20 20 18]. You will specify SerialPartition with this vector indicating the decomposition of taps for serial partitions. The clock rate is determined by the largest element of this vector. In this case the clock rate will be 20 times the input sample rate, 0.882 MHz.

```
[spart, foldingfact, nMults] = hdlfilterserialinfo(lpFilter, 'Multipliers', 4, ...
                                    'InputDataType', nt_in);

generatehdl(lpFilter,'Name', 'partlyserial1',...
            'SerialPartition', spart,...
            'TargetLanguage', 'VHDL', ...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: partlyserial1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\p
### Starting generation of partlyserial1 VHDL entity
### Starting generation of partlyserial1 VHDL architecture
### Clock rate is 20 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: partlyserial1
### HDL latency is 3 samples
```

Partly Serial Architecture for Speed Constraint

Assume that you have a constraint on the clock rate for filter implementation and the maximum clock frequency is 2 MHz. This means that the clock rate can't be more than 45 times the input sample rate. For such a design constraint, specify the `'SerialPartition'` as `[45 33]`. Note that this results in an additional serial partition hardware, implying additional circuitry to multiply-accumulate 33 taps. You can specify the `'SerialPartition'` property using `hdlfilterserialinfo` and its property `'Foldingfactor'` as follows.

```
spart = hdlfilterserialinfo(lpFilter, 'Foldingfactor', 45, ...
                            'InputDataType', nt_in);

generatehdl(lpFilter,'Name', 'partlyserial2', ...
            'SerialPartition', spart,...
            'TargetLanguage', 'VHDL',...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: partlyserial2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\p
### Starting generation of partlyserial2 VHDL entity
### Starting generation of partlyserial2 VHDL architecture
### Clock rate is 45 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: partlyserial2
### HDL latency is 3 samples
```

In general, you can specify any arbitrary decomposition of taps for serial partitions depending on other constraints. The only requirement is that the sum of elements of the vector should be equal the effective filter length.

Cascade-Serial Architecture

The accumulators in serial partitions can be re-used to add the result of the next serial partition. This is possible if the number of taps being processed by one serial partition must be more than that by serial partition next to it by at least 1. The advantage of this technique is that the set of adders required to add the result of all serial partitions are removed. However, this increases the clock rate by 1, as an additional clock cycle is required to complete the additional accumulation step.

Cascade-Serial architecture can be specified using the property `'ReuseAccum'`. This can be done in two ways.

Add `'ReuseAccum'` to generatehdl method and specify it as `'on'`. Note that the value specified for `'SerialPartition'` property has to be such that the accumulator reuse is feasible. The elements of the vector must be in descending order except for the last two which can be same.

If the property `'SerialPartition'` is not specified and `'ReuseAccum'` is specified as `'on'`, the decomposition of taps for serial partitions is determined internally. This decomposition minimizes the clock rate and reuses the accumulator logic. For this audio filter, the serial partitions are `[12 11 10 9 8 7 6 5 4 3 3]`. Note that it uses 11 serial partitions, implying 11 multiply accumulate circuits. The clock rate will be 13 times the input sample rate, 573.3 kHz.

```
generatehdl(lpFilter,'Name', 'cascadeserial1',...
            'SerialPartition', [45 33],...
            'ReuseAccum', 'on', ...
            'TargetLanguage', 'VHDL', ...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: cascadeserial1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\c
### Starting generation of cascadeserial1 VHDL entity
### Starting generation of cascadeserial1 VHDL architecture
### Clock rate is 46 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: cascadeserial1
### HDL latency is 3 samples
```

Optimal decomposition into as many serial partitions required for minimum clock rate possible for reusing accumulator.

```
generatehdl(lpFilter,'Name', 'cascadeserial2', ...
            'ReuseAccum', 'on',...
            'TargetLanguage', 'VHDL',...
            'TargetDirectory', workingdir, ...
            'InputDataType', nt_in);
```

```
### Starting VHDL code generation process for filter: cascadeserial2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp064f4a40_b74c_4933_861a_1c1014e59248\c
### Starting generation of cascadeserial2 VHDL entity
### Starting generation of cascadeserial2 VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Serial partition # 1 has 12 inputs.
### Serial partition # 2 has 11 inputs.
### Serial partition # 3 has 10 inputs.
### Serial partition # 4 has 9 inputs.
### Serial partition # 5 has 8 inputs.
### Serial partition # 6 has 7 inputs.
### Serial partition # 7 has 6 inputs.
### Serial partition # 8 has 5 inputs.
### Serial partition # 9 has 4 inputs.
### Serial partition # 10 has 3 inputs.
### Serial partition # 11 has 3 inputs.
### Successful completion of VHDL code generation process for filter: cascadeserial2
### HDL latency is 3 samples
```

Conclusion

You designed a lowpass direct form symmetric FIR filter to meet the given specification. You then quantized and checked your design. You generated VHDL code for fully parallel, fully serial, partly serial and cascade-serial architectures. You generated a VHDL test bench using a DTMF tone for one of the architectures.

You can use an HDL simulator to verify the generated HDL code for different serial architectures. You can use a synthesis tool to compare the area and speed of these architectures. You can also experiment with and generating Verilog code and test benches.

HDL Distributed Arithmetic for FIR Filters

This example illustrates how to generate HDL code for a lowpass FIR filter with Distributed Arithmetic (DA) architecture.

Distributed Arithmetic

Distributed Arithmetic is a popular architecture for implementing FIR filters without the use of multipliers. DA realizes the sum of products computation required for FIR filters efficiently using LUTs, shifters and adders. Since these operations map efficiently onto an FPGA, DA is a favored architecture on these devices.

Design the Filter

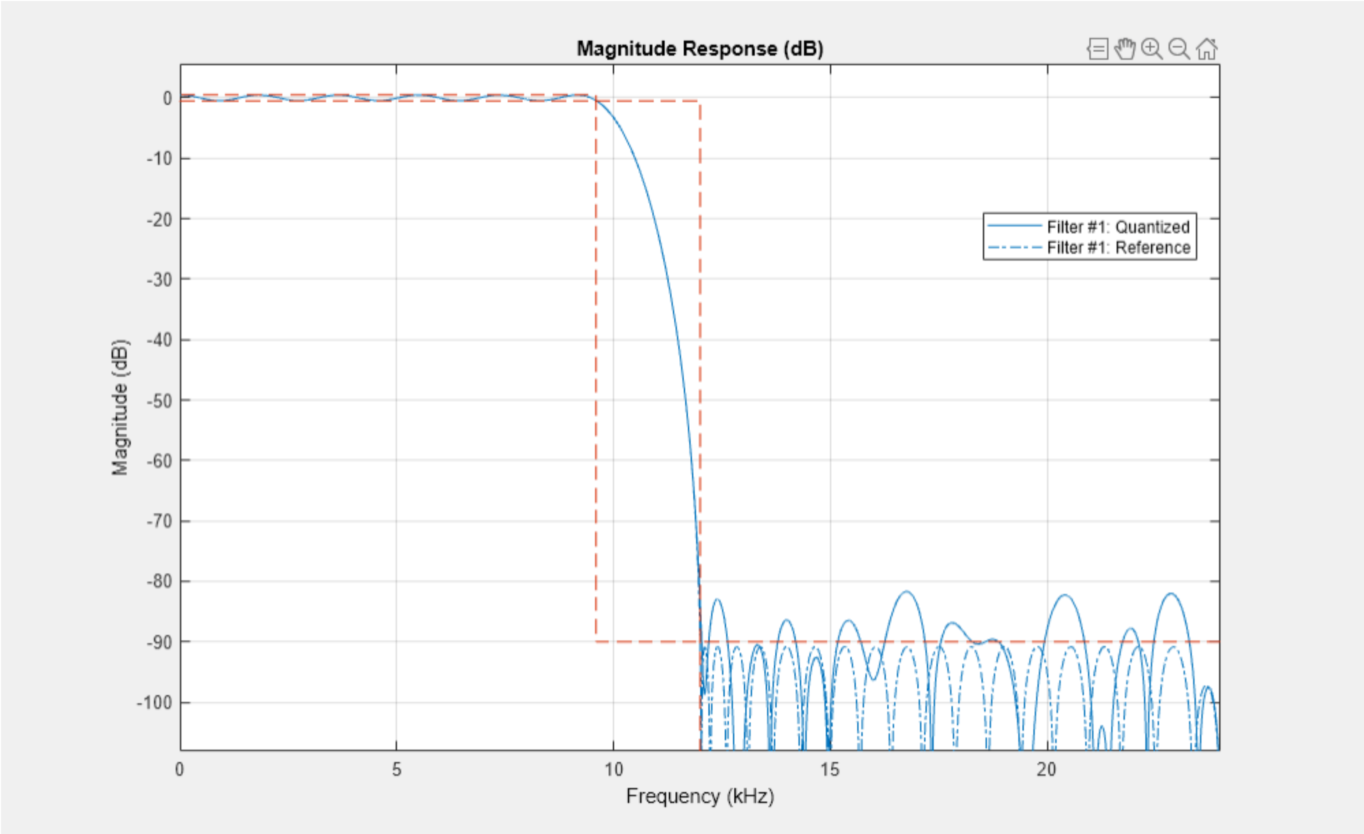
Use a sampling rate of 48 kHz, passband edge frequency of 9.6 kHz and stop frequency of 12k. Set the allowable peak-to-peak passband ripple to 1 dB and the stopband attenuation to -90 dB. Then, design the filter using fdesign.lowpass, and create the System object filter as a direct form FIR filter.

```
Fs          = 48e3;         % Sampling Frequency in Hz
Fpass       = 9.6e3;        % Passband Frequency in Hz
Fstop       = 12e3;         % Stopband Frequency in Hz
Apass       = 1;            % Passband Ripple in dB
Astop       = 90;           % Stopband Attenuation in dB

lpSpec = fdesign.lowpass( 'Fp,Fst,Ap,Ast',...
    Fpass, Fstop, Apass, Astop, Fs);

lpFilter = design(lpSpec, 'equiripple', 'filterstructure', 'dffir',...
    'SystemObject', true);
```

Quantize the Filter

Since DA implements the FIR filter by serializing the input data bits, it requires a quantized filter. Assume that 12 bit input and output word lengths with 11 fractional bits are required (due to of fixed data path requirements or input ADC/output DAC widths). Apply these fixed point settings.

```
inputDataType = numerictype(1,12,11);
outputDataType = inputDataType;
coeffsDataType = numerictype(1,16,16);

lpFilter.FullPrecisionOverride = false;
lpFilter.CoefficientsDataType = 'Custom';
lpFilter.CustomCoefficientsDataType = coeffsDataType;
lpFilter.OutputDataType = 'Custom';
lpFilter.CustomOutputDataType = outputDataType;

% Now check the filter response with fvtool.
fvtool(lpFilter,'Fs',Fs,'Arithmetic','fixed');
```



Generate HDL Code with DA Architecture

To generate HDL Code with DA architecture, invoke the generatehdl command, passing in a valid value to the 'DALUTPartition' property. The 'DALUTPartition' property directs the code generator to use DA architecture, and divides the LUT into a specified number of partitions. The 'DALUTPartition' property specifies the number of LUT partitions, and the number of the taps associated with each partition. For a filter with many taps it is best to divide the taps into a number of LUTs, with each LUT storing the sum of coefficients for only the taps associated with it. The sum of the LUT outputs is computed in a tree structure of adders.

Check the filter length by getting the number of coefficients.

```
FL = length(lpFilter.Numerator);
```

Assume that you have 8 input LUTs; calculate the value of the DALUTPartition property such that you use as many of these LUTs as possible per partition.

```
dalut = [ones(1, floor(FL/8))*8, mod(FL, 8)];
```

Generate HDL with DA architecture. By default, VHDL code is generated. To generate Verilog code, pass in the 'TargetLanguage' property with the value 'Verilog'.

```
workingdir = tempname;
generatehdl(lpFilter, 'DALUTPartition', dalut, ...
                'TargetDirectory', workingdir, ...
                'InputDataType', inputDataType);
```

```
### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for r
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp60801870_710e_42c1_9a68_3d7d4c05e436\t
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 12 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

Convert the Filter Structure to 'Direct form symmetric' and Generate HDL

A symmetrical filter structure offers advantages in hardware, as it halves the number of coefficients to work with. This reduces the hardware complexity substantially. Create a new FIR filter System object 'lpSymFilter' with a 'Direct form symmetric' structure and the same fixed point settings.

```
lpSymFilter = design(lpSpec, 'equiripple', 'filterstructure', 'dfsymfir',...
    'SystemObject', true);

lpSymFilter.FullPrecisionOverride = false;
lpSymFilter.CoefficientsDataType = 'Custom';
lpSymFilter.CustomCoefficientsDataType = coeffsDataType;
lpSymFilter.OutputDataType = 'Custom';
lpSymFilter.CustomOutputDataType = outputDataType;

% Calculate filter length FL for lpSymFilter for the purpose of calculating 'DALUTPartition'
FL = ceil(length(lpSymFilter.Numerator)/2);

% Generate the  value for 'DALUTPartition' as done previously for lpFilter.
dalut_sym = [ones(1, floor(FL/8))*8, mod(FL, 8)];

% Generate HDL code for default radix of 2
generatehdl(lpSymFilter, 'DALUTPartition', dalut_sym, ...
                   'TargetDirectory', workingdir, ...
                   'InputDataType', inputDataType);
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp60801870_710e_42c1_9a68_3d7d4c05e436\t
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

Notice that a symmetrical filter takes one additional clock cycle before the output is obtained. This is because of the carry bit that is added to the input word length as the input data from the symmetrical taps are summed together. The clock rate for 'lpSymFilter' is 13 times the input sample rate, whereas for 'lpFilter' the clock rate was 12 times the input sample rate.

DARadix

The default architecture is a Radix 2 implementation, which operates on one bit of input data on each clock cycle. The number of clock cycles elapsed before an output is obtained is equal to the number of bits in the input data. Thus DA can potentially limit the throughput. To improve the throughput of DA, you can configure DA to process multiple bits in parallel. The 'DARadix' property is provided for this purpose. For example, you can set 'DARadix' to 2^3 to operate on 3 bits in parallel. For a 12 bit input word length, you can specify processing of 1, 2, 3, 4, 6 or 12 bits at a time by specifying corresponding 'DARadix' values of 2^1, 2^2, 2^3, 2^4, 2^6, or 2^12 respectively.

In selecting different 'DARadix' values, you trade off speed vs. area within the DA architecture. The number of bits operated in parallel determines the factor by which the clock rate needs to be increased. This is known as folding factor. For example, the default 'DARadix of 2^1, implying 1 bit at a time, results in a clock rate 12 times the input sample rate or a folding factor of 12. A 'DARadix' of 2^3 results in a clock rate only 4 times the input sample rate, but requires 3 identical sets of LUTs, one for each bit being processed in parallel.

Information Regarding DA Architecture

As explained in previous section, DA architecture presents a lot of options both in terms of LUT sizes and the folding factor. You can use hdlfilterdainfo function to get information regarding various filter lengths based on the value of coefficients. This function also displays two other tables, one for all possible values of DARadix property with corresponding folding factors. The second table displays details of LUT sets with the corresponding values of DALUTPartition property.

```
hdlfilterdainfo(lpFilter, 'InputDataType', inputDataType);
```

```
   | Total Coefficients | Zeros | Effective |
   ------------------------------------------
   |         58         |   0   |     58    |

Effective filter length for SerialPartition value is 58.

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

   | Folding Factor | LUT-Sets Multiple | DARadix |
   ------------------------------------------------
   |        1       |         12        |   2^12  |
   |        2       |         6         |   2^6   |
   |        3       |         4         |   2^4   |
   |        4       |         3         |   2^3   |
   |        6       |         2         |   2^2   |
   |       12       |         1         |   2^1   |

  Details of LUTs with corresponding 'DALUTPartition' values.

   | Max Address Width | Size(bits) |                          LUT Details
   -------------------------------------------------------------------------------------------------------
   |         12        |   259072   |1x1024x13, 1x4096x13, 1x4096x14, 1x4096x15, 1x4096x18
   |         11        |   147544   |2x2048x13, 2x2048x14, 1x2048x18, 1x8x11
   |         10        |    78080   |3x1024x13, 1x1024x16, 1x1024x18, 1x256x13
   |         9         |    43712   |1x16x12, 1x512x12, 2x512x13, 1x512x14, 1x512x15, 1x512x18
   |         8         |    25384   |4x256x13, 1x256x14, 1x256x15, 1x256x18, 1x4x10
   |         7         |    14248   |2x128x12, 3x128x13, 1x128x14, 1x128x16, 1x128x18, 1x4x10
   |         6         |    8000    |1x16x12, 4x64x12, 1x64x13, 2x64x14, 1x64x16, 1x64x17
   |         5         |    4696    |1x32x11, 4x32x12, 3x32x13, 1x32x14, 1x32x15, 1x32x17, 1x8x1
   |         4         |    2904    |3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x16x17, 1x4x1
   |         3         |    1926    |1x2x7, 5x8x11, 8x8x12, 1x8x13, 2x8x14, 2x8x15, 1x8x17
   |         2         |    1412    |2x4x10, 12x4x11, 6x4x12, 2x4x13, 4x4x14, 2x4x15, 1x4x17

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

You can use optional properties for LUT and folding factors to display specific information. You can choose one of the two LUT properties, 'LUTInputs' or 'DALUTPartition' to display all the folding factor options available for the specific LUT inputs.

```
hdlfilterdainfo(lpFilter, 'InputDataType', inputDataType, ...
                    'LUTInputs', 4);
```

```
   | Folding Factor | LUT Inputs | LUT Size |                             LUT Details
   ----------------------------------------------------------------------------------------------
   |        1       |      4     |   34848  |12 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1
   |        2       |      4     |   17424  |6 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
   |        3       |      4     |   11616  |4 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
   |        4       |      4     |   8712   |3 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
   |        6       |      4     |   5808   |2 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
   |       12       |      4     |   2904   |1 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
```

You can also choose one of the two folding factor related properties, 'FoldingFactor' or 'DARadix' to display all the LUT options for the specific folding factor.

```
hdlfilterdainfo(lpFilter, 'InputDataType', inputDataType, ...
                    'Foldingfactor', 6);
```

```
   | Folding Factor | LUT Inputs | LUT Size |                             LUT Details
   ----------------------------------------------------------------------------------------------
   |        6       |     12     |  518144  |2 x (1x1024x13, 1x4096x13, 1x4096x14, 1x4096x15, 1x4
   |        6       |     11     |  295088  |2 x (2x2048x13, 2x2048x14, 1x2048x18, 1x8x11)
   |        6       |     10     |  156160  |2 x (3x1024x13, 1x1024x16, 1x1024x18, 1x256x13)
   |        6       |      9     |   87424  |2 x (1x16x12, 1x512x12, 2x512x13, 1x512x14, 1x512x15
   |        6       |      8     |   50768  |2 x (4x256x13, 1x256x14, 1x256x15, 1x256x18, 1x4x10)
   |        6       |      7     |   28496  |2 x (2x128x12, 3x128x13, 1x128x14, 1x128x16, 1x128x1
   |        6       |      6     |   16000  |2 x (1x16x12, 4x64x12, 1x64x13, 2x64x14, 1x64x16, 1x
   |        6       |      5     |   9392   |2 x (1x32x11, 4x32x12, 3x32x13, 1x32x14, 1x32x15, 1x
   |        6       |      4     |   5808   |2 x (3x16x11, 5x16x12, 2x16x13, 2x16x14, 1x16x15, 1x
   |        6       |      3     |   3852   |2 x (1x2x7, 5x8x11, 8x8x12, 1x8x13, 2x8x14, 2x8x15,
   |        6       |      2     |   2824   |2 x (2x4x10, 12x4x11, 6x4x12, 2x4x13, 4x4x14, 2x4x15
```

Notice that LUT details indicate a factor by which the LUT sets need to be replicated to achieve the corresponding folding factor. Also, total LUT size is calculated with above factor.

You can use output arguments to return the values of DALUTPartition and DARadix for a specific configuration and use it with generatehdl command. Let us assume that you can intend to raise the clock rate by 4 times the sample rate and want to use 6 input LUTs. You can verify that the LUT details meet your area requirements.

```
hdlfilterdainfo(lpFilter, 'InputDataType', inputDataType, ...
                    'FoldingFactor', 4, ...
                    'LUTInputs',  6);
```

```
   | Folding Factor |     LUT Size     |                             LUT Details
   ----------------------------------------------------------------------------------------------
   |        4       | 3 x 8000 = 24000 |3 x (1x16x12, 4x64x12, 1x64x13, 2x64x14, 1x64x16, 1x64x1
```

Now generate HDL with the above constraints by first storing the required values of DALUTPartition and DARadix in variables by using the output arguments to the hdlfilterdainfo function. You can then invoke generatehdl command using these variables.

```
[dalut, dr] = hdlfilterdainfo(lpFilter, 'InputDataType', inputDataType, ...
                                  'FoldingFactor', 4, ...
```

```
                                  'LUTInputs', 6);

generatehdl(lpFilter, 'InputDataType', inputDataType, ...
                'DALUTPartition', dalut, ...
                'DAradix', dr, ...
                'TargetDirectory', workingdir);
```

```
### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for r
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp60801870_710e_42c1_9a68_3d7d4c05e436\t
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

Conclusion

You designed a lowpass direct form FIR filter to meet the given specification. You then quantized and checked your design. You generated VHDL code for DA with various radices and explored speed vs. area trade-offs within DA by replicating LUTs and operating on multiple bits in parallel.

You can generate a test bench with a standard stimulus and/or your own defined stimulus, and use an HDL Simulator to verify the generated HDL code for DA architectures. You can use a synthesis tool to compare the area and speed of these architectures.

HDL Programmable FIR Filter

This example illustrates how to generate HDL code for an FIR filter with a processor interface for loading coefficients. The filter can be programmed to any desired response by loading the coefficients into an internal coefficient memory using the processor interface.

Let us assume that we need to implement a bank of filters, having different responses, on a chip. If all of the filters have a direct-form FIR structure, and the same length, then we can use a processor interface to load the coefficients for each response from a RAM or register file when needed.

This design will add latency of a few cycles before the input samples can be processed with the loaded coefficients. However, it has the advantage that the same filter hardware can be programmed with new coefficients to obtain a different filter response. This saves chip area, as otherwise each filter would be implemented separately on the chip.

In this example, we will consider two FIR filters, one with a highpass response and the other with a lowpass response. We will show how the same filter hardware can be programmed for each response by loading the corresponding set of coefficients. We will generate VHDL code for the filter and show the two responses using the generated VHDL test bench.

Design the Filters

Create the lowpass filter design object, then create the FIR Filter System object (Hlp). Then, transform it to create a FIR Filter System object with a highpass response (Hhp).

```
Fpass = 0.45; % Passband Frequency
Fstop = 0.55; % Stopband Frequency
Apass = 1;    % Passband Attenuation (dB)
Astop = 60;   % Stopband Attenuation (dB)

f = fdesign.lowpass('Fp,Fst,Ap,Ast',Fpass,Fstop,Apass,Astop);
lpFilter = design(f, 'equiripple','FilterStructure', 'dfsymfir','SystemObject',true); % Lowpass

hpcoeffs = firlp2hp(lpFilter.Numerator);
hpFilter = dsp.FIRFilter('Numerator', hpcoeffs); % Highpass
```

Quantize the Filters

Assume the coefficients need be stored in a memory of bit width 14. Using this information, apply fixed point settings to the System object filter.

```
lpFilter.FullPrecisionOverride=false;
lpFilter.CoefficientsDataType='Custom';
lpFilter.CustomCoefficientsDataType=numerictype(1,14,13);
lpFilter.OutputDataType='Same as Accumulator';
lpFilter.ProductDataType='Full precision';
lpFilter.AccumulatorDataType='Full precision';

hpFilter.FullPrecisionOverride=false;
hpFilter.CoefficientsDataType='Custom';
hpFilter.CustomCoefficientsDataType=numerictype(1,14,13);
hpFilter.OutputDataType='Same as Accumulator';
hpFilter.ProductDataType='Full precision';
hpFilter.AccumulatorDataType='Full precision';
```

After applying fixed point settings, it is important to verify that the System object filter still meets the specifications. We will use the function 'measure' to check if this is true.

```
measure(lpFilter,'Arithmetic','fixed')
```

```
ans =
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.45
3-dB Point       : 0.46957
6-dB Point       : 0.48314
Stopband Edge    : 0.55
Passband Ripple  : 0.89243 dB
Stopband Atten.  : 55.3452 dB
Transition Width : 0.1
```

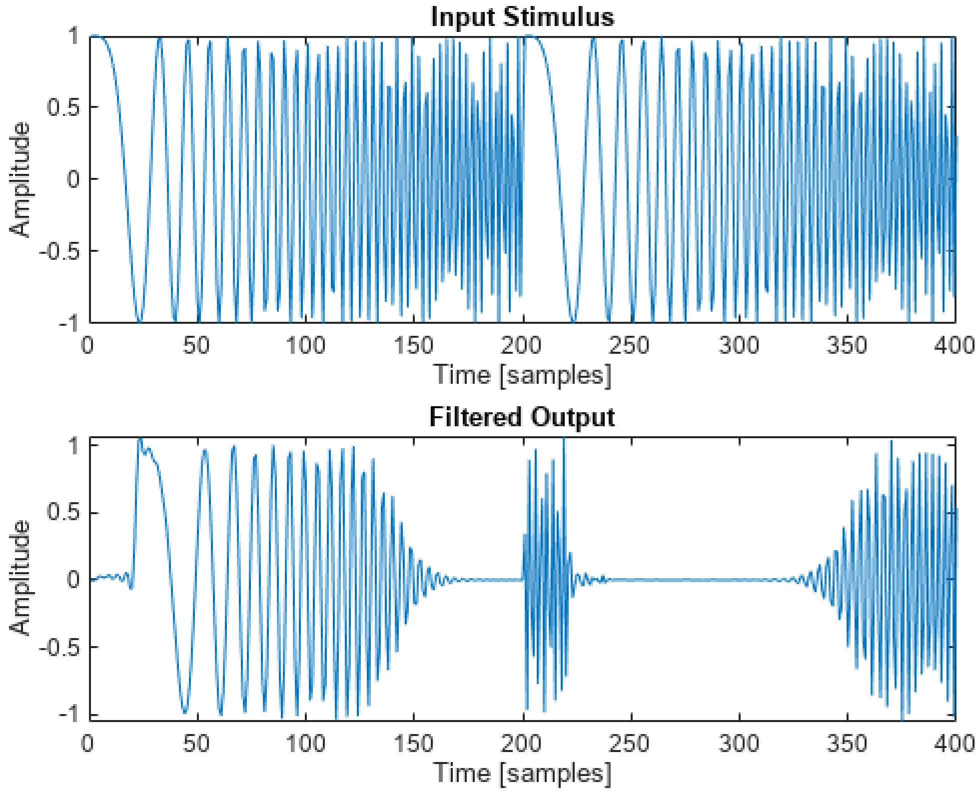
Verify the Filter Output

Generate a linear swept-frequency stimulus signal using chirp. Use this input stimulus for filtering through the lowpass FIR filter first. Then change the coefficients of the filter to obtain a highpass response and use the same input sample to filter again.

For the above two-stage filtering operation our goal is to compare the filter output from MATLAB® with that from the generated HDL code.

Plotting the input samples and the filtered output shows the lowpass and highpass behavior.

```
x = chirp(0:199,0,199,0.4);

lpcoeffs = lpFilter.Numerator;            % store original lowpass coefficients
y1 = lpFilter(fi(x,1,14,13).');           % filter the signal

lpFilter.Numerator = hpFilter.Numerator;  % load the highpass filter coefficients
y2 = lpFilter(fi(x,1,14,13).');           % filter the signal

y = [y1; y2];                             % concatenate output signals

lpFilter.Numerator = lpcoeffs;            % restore original lowpass coefficients

subplot(2,1,1);plot([x,x]);
xlabel('Time [samples]');ylabel('Amplitude'); title('Input Stimulus');
subplot(2,1,2);plot(y);
xlabel('Time [samples]');ylabel('Amplitude'); title('Filtered Output');
```



Generate VHDL Code with Processor Interface and Test Bench

For the quantized lowpass filter, we will generate the VHDL code with a processor interface by setting the property 'CoefficientSource' to 'ProcessorInterface'. This will result in the generated code having additional ports for write_address, write_enable, coeffs_in, and write_done signals. This interface can be used to load the coefficients from a host processor into an internal register file. The HDL has an additional shadow register that is updated from the register file when the 'write_done' signal is high. This enables simultaneous loading and processing of data by the filter entity.

To verify that the filter entity can be successively loaded with two different sets of filter coefficients, we will generate a VHDL test bench. First, the test bench loads the lowpass coefficients and processes the input samples. Then the test bench loads the coefficients corresponding to the highpass filter response, and processes the input samples again.

The generated VHDL code and VHDL test bench can be compiled and simulated using an HDL simulator such as ModelSim®. Notice that the loading of the second set of coefficients and the processing of the last few input samples are performed simultaneously.

In order to generate the required test bench, we set the property 'GenerateHDLTestbench' to 'on' and pass 'TestbenchCoeffStimulus' in the call to the generatehdl command. The value passed in for 'TestbenchCoeffStimulus' is a vector of coefficients that are to be used for subsequent processing of input samples. This example passes in a vector of coefficients corresponding to a highpass filter.

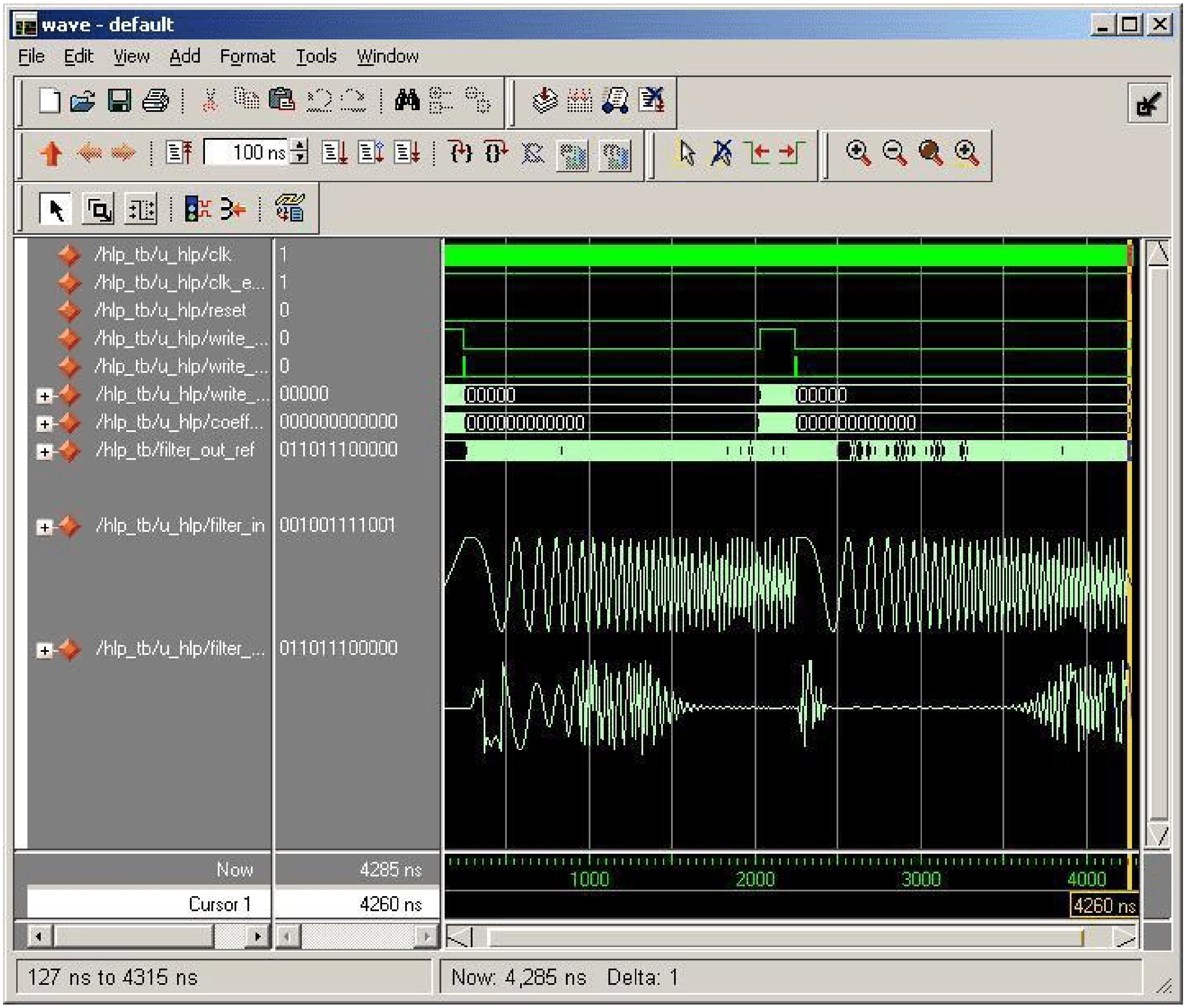
Assume 14-bit signed fixed-point input with 13 bits of fraction precision is needed due to fixed data path requirements of input ADC.

```
%As the symmetric structure is selected, the field 'TestbenchCoeffStimulus'
% has to be the half of length of filter.

workingdir = tempname;

generatehdl(lpFilter,'Name','FilterProgrammable', ...
                 'InputDataType',numerictype(1,14,13), ...
                 'TargetLanguage','VHDL', ...
                 'TargetDirectory',workingdir, ...
                 'CoefficientSource','ProcessorInterface', ...
                 'GenerateHDLTestbench','on', ...
                 'TestBenchUserStimulus',x, ...
                 'TestbenchCoeffStimulus',hpFilter.Numerator(1:(length(hpFilter.Numerator)+1)/2)
```

```
### Starting VHDL code generation process for filter: FilterProgrammable
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp7f286531_6eac_4e49_9b1f_a240c5083220\
### Starting generation of FilterProgrammable VHDL entity
### Starting generation of FilterProgrammable VHDL architecture
### Successful completion of VHDL code generation process for filter: FilterProgrammable
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 200 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp7f286531_6eac_4e49_9b1f_a24
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

ModelSim® Simulation Results

The following display shows the ModelSim HDL simulator after running the generated .do file scripts for the test bench. Compare the ModelSim result with the MATLAB result as plotted before.



Conclusion

We designed highpass and lowpass FIR filters to meet the given specifications. We then quantized the filter and generated VHDL code for the filter, with an interface to load the coefficients from a processor. We then generated a VHDL test bench that showed the processing of input samples after loading lowpass coefficients, repeating the operation with the highpass coefficients. We showed how to generate the VHDL code that implements filter hardware that is reusable for different responses when different sets of coefficients are loaded via the port interface from a host processor.

Properties

Fundamental HDL Code Generation Properties

Customize filter name, destination folder, and specify target language

Description

With the fundamental HDL code generation properties, you can customize filter name, destination folder, and specify the target language.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'TargetLanguage','Verilog');
```

Properties

Target

`TargetLanguage` — HDL language of generated filter code
`'VHDL'` (default) | `'Verilog'`

HDL language of generated filter code, specified as `'VHDL'` or `'Verilog'`.

`Name` — File name of generated HDL code
character vector | string scalar

File name of generated HDL code, specified as a character vector or a string scalar. The coder adds a file type extension to the file name, as specified by the `VerilogFileExtension` or `VHDLFileExtension` properties. The file name also determines the name of the generated VHDL entity or Verilog module for the filter. The file is located in the folder specified by the `TargetDirectory` property.

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see "Resolving HDL Reserved Word Conflicts" on page 5-9.

`TargetDirectory` — Location of generated files
`'hdlsrc'` (default) | character vector | string scalar

Location of generated files, specified as a character vector or a string scalar. Specify the location as a subfolder under the current working folder, or as a complete path to the files.

Language-Specific

`VerilogFileExtension` — File type extension of generated Verilog file
`'.v'` (default) | character vector | string scalar

File type extension of generated Verilog file, specified as a character vector or a string scalar.

`VHDLFileExtension` — File type extension of generated VHDL file

`'.vhd'` (default) | character vector | string scalar

File type extension of generated VHDL file, specified as a character vector or a string scalar.

Data Types

`InputDataType` — Input data type for System object

`numerictype` object

Input data type for System object, specified as a `numerictype` object. This argument is required only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits. For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15));
```

`FractionalDelayDataType` — Fractional delay data type

`numerictype` object

Fractional delay data type, specified as a `numerictype` object. This argument is required only when the input filter is a `dsp.VariableFractionalDelay` System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits. For example:

```
farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17), ...
    'FractionalDelayDataType',numerictype(1,8,7));
```

Tips

If you use the `fdhdltool` function to generate HDL code, you can specify the input and fractional delay data types as arguments, and then set additional properties in the Generate HDL dialog box.

Property	Location in Dialog Box
Language	**Target** section at top of dialog box
Name	
Folder	
Verilog file extension	**Global Settings** tab
VHDL file extension	

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

Topics

“Code Generation Fundamentals”

HDL Filter Configuration Properties

Configure coefficients, complex input ports, and optional ports for specific filter types

Description

With the HDL filter configuration properties, you can configure coefficients, complex input ports, and optional ports for specific filter types. For filter serialization and pipeline properties, see Optimization.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'CoefficientSource','ProcessorInterface');
```

Properties

Coefficients

`CoefficientSource` — Source of programmable filter coefficients
`'Internal'` (default) | `'ProcessorInterface'`

Source of programmable filter coefficients, specified as `'Internal'` or `'ProcessorInterface'`. This property applies only to “Programmable Filter Coefficients for FIR Filters” on page 3-23 and “Programmable Filter Coefficients for IIR Filters” on page 3-30.

- `'Internal'` — The coder obtains the filter coefficients from the filter object. The coefficients are hard-coded in the generated HDL code.
- `'ProcessorInterface'` — The coder generates a memory interface for the filter coefficients. You can drive this interface with an external microprocessor. The generated VHDL entity or Verilog module for the filter includes these ports for the processor interface:

  - `coeffs_in` — Input port for coefficient data
  - `write_address` — Write-address for coefficient memory
  - `write_enable` — Write-enable signal for coefficient memory
  - `write_done` — Signal to indicate completion of coefficient write operation

  If you generate a test bench, you can specify the input stimulus for this interface by using the `TestBenchCoeffStimulus` property.

  For serial FIR filter, you can also specify the memory type for storing the programmable coefficients by setting the `CoefficientMemory` property.

`CoefficientMemory` — Memory type for programmable filter coefficients
`'Registers'` (default) | `'DualPortRAMs'` | `'SinglePortRAMs'`

Memory type for programmable filter coefficients, specified as `'Registers'`, `'DualPortRAMs'`, or `'SinglePortRAMs'`. This property applies only to "Programmable Filter Coefficients for FIR Filters" on page 3-23 with a fully serial, partly serial, or cascade serial architecture.

- `'Registers'` — The coder generates a register file for storing programmable coefficients.
- `'SinglePortRAMs'` or `'DualPortRAMs'` — The coder generates the respective RAM interface for storing programmable coefficients.

Dependencies

This property applies only when you set `CoefficientSource` to `'ProcessorInterface'`. If the coder does not generate an interface for programmable coefficients, this `CoefficientMemory` property is ignored.

Optional Ports

`InputComplex` — Generate complex input data ports

`'off'` (default) | `'on'`

Generate complex input data ports, specified as `'off'` or `'on'`. Use this option when your filter design requires complex input data. See "Using Complex Data and Coefficients" on page 5-26. When you set this property to `'on'`, the coder generates ports and signal paths for the real and imaginary components of a complex signal.

You can customize the port names by setting the `ComplexRealPostfix` and `ComplexImagPostfix` properties.

Dependencies

To generate complex inputs, you must also set `CoefficientSource` to `'Internal'`. Complex inputs are not supported when filter coefficients are obtained from a processor interface.

`ClockInputs` — Type of generated clock inputs

`'Single'` (default) | `'Multiple'`

Type of generated clock inputs, specified as `'Single'` or `'Multiple'`. This property applies only to "Multirate Filters" on page 3-2.

- `'Single'` — The generated VHDL entity or Verilog module for the filter has a single clock input, an associated clock enable input, and a clock enable output. The generated code includes a counter that controls the timing of data transfers to the filter output (for decimation filters) or input (for interpolation filters). The counter behaves as a secondary clock. The decimation or interpolation factor determines the clock rate of the counter. This option provides a self-contained clocking solution for FPGA designs.

  To customize the names of these clock inputs and outputs, see the `ClockInputPort`, `ClockEnableInputPort`, and `ClockEnableOutputPort` properties.

  Interpolators also pass through the clock enable input signal to an output port named `ce_in`. This signal indicates when the object accepted an input sample. You can use this signal to control the upstream data flow. You cannot customize this port name.
- `'Multiple'` — The generated VHDL entity or Verilog module for the filter has separate clock inputs for each rate of the multirate filter. Each clock input has an associated clock enable input. The coder does not generate a clock enable output. Provide input clock signals that correspond to the desired decimation or interpolation factor.

This option provides more flexibility than a single clock input. However, multiple clock inputs assume that you provide higher-level HDL code to drive the input clocks of your filter. The coder does not generate synchronizers between multiple clock domains. If you generate a test bench, examine the `clk_gen` processes for each clock.

The following filters do not support `'Multiple'`:

- Filters with a partly serial architecture
- Multistage sample rate converters: `dsp.FIRRateConverter`, `dsp.FarrowRateConverter`, or multirate `dsp.FilterCascade`

For an example, see “Clock Ports for Multirate Filters” on page 10-22.

`AddRatePort` — Generate rate ports
`'off'` (default) | `'on'`

Generate rate ports, specified as `'off'` or `'on'`. This property applies only to “Variable Rate CIC Filters” on page 3-6.

When you set this property to `'on'`, the coder generates `rate` and `load_rate` ports for the filter. A variable-rate CIC filter has a programmable rate change factor. When you assert the `load_rate` signal, the `rate` port loads in a rate factor. You can generate rate ports only for a full-precision filter.

If you generate a test bench, you can customize the rate port stimulus by setting the `TestBenchRateStimulus` property.

`FracDelayPort` — Name of fractional delay input port
`'filter_fd'` (default) | character vector | string scalar

Name of fractional delay input port, specified as `'filter_fd'`, a character vector, or a string scalar. This property applies only to “Single-Rate Farrow Filters” on page 3-18. For example:

```
farrowfilt = dsp.VariableFractionalDelay('InterpolationMethod','Farrow');
generatehdl(farrowfilt,'InputDataType',numerictype(1,18,17), ...
    'FractionalDelayDataType',numerictype(1,8,7), ...
    'FracDelayPort','fractional_delay');
```

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

If you generate a test bench, you can customize the fractional delay stimulus by setting the `TestBenchFracDelayStimulus` property.

Tips

If you use the `fdhdltool` function to generate HDL code, you can set the corresponding properties in the Generate HDL dialog box.

Filter Type	Property	Location in Dialog Box
FIR or IIR filter with programmable coefficients	**Coefficient source**	**Filter Architecture** tab

Filter Type	**Property**	**Location in Dialog Box**
FIR filter with serial architecture and programmable coefficients	**Coefficient memory**	**Filter Architecture** tab, when **Coefficient source** is set to `Processor interface`
Filter with complex input data	**Input complexity**	**Global Settings** tab > **Ports** tab
Multirate filter	**Clock inputs**	**Global Settings** tab
CIC filter	**Add rate port**	**Filter Architecture** tab
Single-rate Farrow filter	**Fractional delay port**	**Global Settings** tab > **Ports** tab

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

Topics

"Filter Configuration Options"

HDL Optimization Properties

Optimize speed or area of generated HDL code

Description

With the HDL optimization properties, you can specify speed vs. area tradeoffs in the generated code.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'AddPipelineRegisters','on');
```

Properties

Speed Optimization

`AddPipelineRegisters` — Optimize clock rate with pipeline registers

`'off'` (default) | `'on'`

Optimize clock rate with pipeline registers, specified as `'off'` or `'on'`. You cannot use this property with fully serial or cascade serial filters. When you set this property to `'on'`, the coder adds pipeline registers between filter computation stages. Although the registers add to the overall filter latency, they provide significant improvements to the clock rate.

Filter Type	Location of Added Pipeline Register
FIR transposed	Between coefficient multipliers and adders
Direct form FIR, antisymmetric FIR, and symmetric FIR	Between levels of a tree-based final adder For an alternative tree-based summation technique, see also the property `FIRAdderStyle`.
IIR	Between sections
CIC	Between comb sections

For more details, see "Optimizing the Clock Rate with Pipeline Registers" on page 4-25.

`FIRAdderStyle` — Optimize clock rate with summation technique

`'linear'` (default) | `'tree'` | `'pipelined'`

Optimize clock rate with summation technique, specified as `'linear'`, `'tree'`, or `'pipelined'`. This property applies only to direct form FIR, antisymmetric FIR, and symmetric FIR filters. You cannot use this property with fully serial or cascade serial filters. When you set this property to `'tree'`, the coder creates a final adder that performs pairwise addition on successive products that execute in parallel, rather than sequentially. When you set this property to `'pipelined'`, the coder creates a tree-based final adder with pipeline registers between the levels of the tree.

For more details, see "Optimizing Final Summation for FIR Filters" on page 4-26.

Dependencies

This property applies only when the AddPipelineRegisters property is set to 'off'.

AddInputRegister — Extra input register

'on' (default) | 'off'

Extra input register, specified as 'on' or 'off'. When this property is set to 'on', the coder generates a signal named input_register and includes a process statement that controls the register. If the incurred latency is a concern, or if the filter is incorporated into a code that has an existing input register, set this property to 'off'. For more details, see “Specifying or Suppressing Registered Input and Output” on page 4-27.

AddOutputRegister — Extra output register

'on' (default) | 'off'

Extra output register, specified as 'on' or 'off'. When this property is set to 'on', the coder generates a signal named output_register and includes a process statement that controls the register. If the incurred latency is a concern, or if the filter is incorporated into a code that has an existing output register, set this property to 'off'. For more details, see “Specifying or Suppressing Registered Input and Output” on page 4-27.

MultiplierInputPipeline — Number of pipeline stages on multiplier inputs

0 (default) | nonnegative integer

Number of pipeline stages on multiplier inputs, specified as a nonnegative integer. This property applies only to FIR filters. Multiplier pipelining can significantly increase clock rates. For more details, see “Multiplier Input and Output Pipelining for FIR Filters” on page 4-26.

Dependencies

To enable this property, set CoeffMultipliers to 'multipliers'.

MultiplierOutputPipeline — Number of pipeline stages on multiplier outputs

0 (default) | nonnegative integer

Number of pipeline stages on multiplier outputs, specified as a nonnegative integer. This property applies only to FIR filters. Multiplier pipelining can significantly increase clock rates. For more details, see “Multiplier Input and Output Pipelining for FIR Filters” on page 4-26.

Dependencies

To enable this property, set CoeffMultipliers to 'multipliers'.

Area Optimization

OptimizeForHDL — HDL code optimization

'off' (default) | 'on'

HDL code optimization, specified as 'off' or 'on'. By default, the coder generates the literal implementation of the filter with numeric behaviour that matches the filter object exactly. This implementation is not necessarily an optimal HDL implementation. When this property is set to 'on', the coder reduces the area of the hardware implementation and optimizes data types and quantization effects. For more details about the underlying tradeoffs, see “Optimize for HDL” on page 4-29.

`CoeffMultipliers` — Implementation of coefficient multiplications

`'multiplier'` (default) | `'csd'` | `'factored-csd'`

Implementation of coefficient multiplications, specified as `'multiplier'`, `'csd'`, or `'factored-csd'`. You cannot use this property with multirate or serial filters.

- `'multiplier'` — The coder retains multiplier logic in the generated HDL code.
- `'csd'` or `'factored-csd'`— The coder implements multiplication using canonical signed digit (CSD) logic. The CSD technique replaces multipliers with shift and add logic. This technique also minimizes the number of adders used for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This optimization decreases the area used by the filter while maintaining or increasing clock speed.
- `'factored-csd'` — The coder implements multiplication using factored CSD logic. Factored CSD replaces multiplier operations with shift and add operations on prime factors of the coefficients. This option achieves a greater area reduction than CSD, at the cost of decreasing clock speed.

For more details, see “CSD Optimizations for Coefficient Multipliers” on page 4-24.

`SerialPartition` — Partitions for serial filter architectures

`-1` (default) | effective filter length | `[p1 p2 ... pN]` | cell array of serial partitions

Partitions for serial filter architectures, specified as one of the following:

- `-1` — The coder generates a fully parallel architecture. This architecture is equivalent to a serial partition defined as a vector of ones of the size of the effective filter length.
- Effective filter length — The coder generates a fully serial architecture.
- `[p1 p2 ... pN]` — The coder generates a partly serial architecture with *N* partitions. The integers in the vector specify the length of each partition. The sum of the vector elements must be equal to the effective filter length. To reduce the area further, you can generate a cascade-serial architecture by enabling the `ReuseAccum` property. For some examples, see “Generate Serial Partitions for FIR Filter” on page 10-9.
- Cell array of serial partitions — The coder generates partitions for each filter stage in a cascaded filter. Specify the partitions for each filter stage as `-1`, the effective filter length, or a vector of integers. The elements of each vector must sum to the effective filter length of the associated filter in the cascade. For an example, see “Generate Serial Partitions of Cascaded Filter” on page 10-11.

  When the serial partition of a filter stage is set to `-1`, you can specify a LUT partition for that stage by using the `DALUTPartition` and `DARadix` properties. For more details, see “Architecture Options for Cascaded Filters” on page 4-23.

You cannot use this property with IIR SOS filters. To generate serial architectures for IIR SOS filters, use the `FoldingFactor` or `NumMultipliers` properties instead.

Use this table as a guide for calculating the effective filter length. Alternatively, you can use the `hdlfilterserialinfo` function to display the effective filter length and possible partitions for a filter.

Filter Type	Effective Filter Length Calculation
Direct form	`FL = length(find(filt.Numerator~= 0))`
Direct form symmetric	`FL = ceil(length(find(filt.Numerator~= 0))/2)`

Filter Type	Effective Filter Length Calculation
Direct form antisymmetric	

For more details, see “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-5.

For an overview of parallel and serial architectures and a list of filter types supported for each architecture, see “Speed vs. Area Tradeoffs” on page 4-2.

`ReuseAccum` — Accumulator reuse for cascade-serial architecture

`'off'` (default) | `'on'`

Accumulator reuse for cascade-serial architecture, specified as `'off'` or `'on'`. When this property is set to `'on'`, the coder groups filter taps into several serial partitions. The accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of the partitions is therefore computed at the accumulator of the first partition. This technique, called accumulator reuse, saves chip area. If the property `SerialPartition` is not defined, the coder generates an optimal partition. For more details, see “Specifying Speed vs. Area Tradeoffs via generatehdl Properties” on page 4-5.

For an overview of parallel and serial architectures and a list of filter types supported for each architecture, see “Speed vs. Area Tradeoffs” on page 4-2.

`DALUTPartition` — Lookup table partitions for distributed arithmetic

`-1` (default) | effective filter length | `[p1 p2 ... pN]` | `{p1 p2 ... pN; q1 q2 ... qN; ... }` | cell array of DALUT partitions

Lookup table (LUT) partitions for distributed arithmetic (DA), specified as one of the following:

- `-1` — The coder generates a fully parallel architecture.
- Effective filter length — The coder generates a DA implementation without LUT partitioning.
- `[p1 p2 ... pN]` — The coder generates a DA implementation with *N* LUT partitions. The integers in the vector specify the size of each partition. The maximum size for an individual partition is 12. The sum of the vector elements must be equal to the effective filter length. For multirate filters, each polyphase subfilter uses the same LUT partitions. For an example, see “Distributed Arithmetic for Single Rate Filters” on page 10-15.
- `{p1 p2 ... pN; q1 q2 ... qN; ... }` — The coder generates a DA implementation with *N* unique LUT partitions for each polyphase subfilter of a multirate filter. Each row of the matrix specifies the partitions for one subfilter. The elements in each row must sum to the associated subfilter length, `FLi`. For an example, see “Distributed Arithmetic for Multirate Filters” on page 10-15.
- Cell array of DALUT partitions — The coder generates DA implementation with different LUT partitions for each filter stage of the cascade. Specify the LUT partitions for each filter stage as `-1`, the effective filter length, or a vector of integers. The elements of each vector must sum to the effective filter length of the associated filter in the cascade. For an example, see “Distributed Arithmetic for Cascaded Filters” on page 10-16.

  When the LUT partition of a filter stage is set to `-1`, you can specify a serial partition for that stage by using the `SerialPartition` property. For more details, see “Architecture Options for Cascaded Filters” on page 4-23.

Use this table as a guide for calculating the effective filter length. Alternatively, you can use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible `DARadix` values for the filter.

Filter Type	Effective Filter Length Calculation
Direct form	`FL = length(find(filt.Numerator~= 0))`
Direct form symmetric	`FL = ceil(length(find(filt.Numerator~= 0))/2)`
Direct form antisymmetric	
Multirate with uniform LUT partitions for each polyphase subfilter	`FL = size(polyphase(filt),2)`
Multirate with unique LUT partitions for each polyphase subfilter	`p = polyphase(filt)` `FLi = length(find(p(i,:)))`, where `i` is the index to the `i`th row of the polyphase matrix of the filter. The `i`th row of the matrix `p` represents the `i`th subfilter.

For more details, see “Distributed Arithmetic for FIR Filters” on page 4-16.

`DARadix` — Number of bits processed simultaneously in distributed arithmetic

2 (default) | 2^N | {2^N,2^M,...}

Number of bits processed simultaneously in distributed arithmetic (DA), specified as 2, 2^N, or {2^N,2^M,...} where:

- N > 0
- `mod(W,N) = 0`, where `W` is the input word size of the filter
- $2^N <= 2^W$

This property specifies a degree of parallelism in the DA architecture which can improve clock speed at the expense of area.

- 2^1 — The coder implements a fully serial DA architecture that processes 1 bit at a time.
- 2^N — The coder generates a partly serial DA architecture when `1 < N < W`.
- 2^W — The coder generates a fully parallel DA architecture.
- {2^N,2^M,...} — The coder generates a DA implementation with different `DARadix` values for each filter stage in a cascaded filter. For an example, see “Distributed Arithmetic for Cascaded Filters” on page 10-16.

  When the `DARadix` value of a filter stage is set to 2, you can specify a serial architecture for that stage by using the `SerialPartition` property. For more details, see “Architecture Options for Cascaded Filters” on page 4-23.

For more details, see “Distributed Arithmetic for FIR Filters” on page 4-16.

`FoldingFactor` — Folding factor for IIR filter

1 (default) | positive integer

Folding factor for IIR filter, specified as 1 or a positive integer. Use this property to define a serial architecture for direct form I or direct form II SOS filters. To reduce area in a serial architecture implementation, you can share multipliers at the cost of latency. The folding factor specifies the factor by which the clock rate increases in response to area optimization.

You can specify either the `FoldingFactor` property or the `NumMultipliers` property, but not both. If you do not specify either property, the coder generates a fully parallel architecture.

For an example, see “Generate Serial Architectures for IIR Filter” on page 10-13. To obtain information about the `FoldingFactor` options and the corresponding `NumMultipliers`, call the `hdlfilterserialinfo` function.

`NumMultipliers` — Number of shared multipliers for IIR filter
positive integer

Number of shared multipliers for IIR filter, specified as a positive integer. Use this property to define a serial architecture for direct form I or direct form II SOS filters. Shared multipliers reduce area at the cost of an increased clock rate.

You can specify either the `NumMultipliers` property or the `FoldingFactor` property, but not both. If you do not specify either property, the coder generates a fully parallel architecture.

For an example, see “Generate Serial Architectures for IIR Filter” on page 10-13. To obtain information about the `NumMultipliers` options and the corresponding `FoldingFactor`, call the `hdlfilterserialinfo` function.

Tips

If you use the `fdhdltool` function to generate HDL code, you can set the corresponding properties in the Generate HDL dialog box.

Property	Location in Dialog Box
Add input register	**Global Settings** tab > **Ports** tab
Add output register	
Additional optimization properties	**Filter Architecture** tab See also: • “Select Architectures in the Generate HDL Tool” on page 4-7 • “Distributed Arithmetic Options in the Generate HDL Tool” on page 4-19

See Also

`generatehdl` | `fdhdltool`

Topics

“Optimization”
“Cascaded Filter with Multiple Architectures” on page 10-19

HDL Port and Identifier Properties

Customize ports, identifiers, and comments

Description

With the HDL port and identifier properties, you can customize ports, identifiers, and comments in the generated code.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'ClockInputPort','clk_input');
```

Properties

Clocks, Inputs, and Outputs

`ClockEnableInputPort` — Name of clock enable input port
`'clk_enable'` (default) | character vector | string scalar

Name of clock enable input port, specified as `'clk_enable'`, a character vector, or a string scalar.

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

`ClockEnableOutputPort` — Name of clock enable output port
`'ce_out'` (default) | character vector | string scalar

Name of clock enable output port, specified as `'ce_out'`, a character vector, or a string scalar. This property applies only to “Multirate Filters” on page 3-2 that use a single input clock (default behavior of `ClockInputs`). For an example, see “Clock Ports for Multirate Filters” on page 10-22. For more details, see “Code Generation Options for Multirate Filters” on page 3-2.

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

`ClockInputPort` — Name of clock input port
`'clk'` (default) | character vector | string scalar

Name of clock input port, specified as `'clk'`, a character vector, or a string scalar.

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

InputPort — Name of filter input port
'filter_in' (default) | character vector | string scalar

Name of filter input port, specified as 'filter_in', a character vector, or a string scalar.

If you specify a value that is a reserved word in the target language, the coder adds the postfix _rsvd to this value. You can update the postfix value by using the ReservedWordPostfix property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

InputType — Data type of filter input port
'std_logic_vector' (default) | 'signed/unsigned' | 'wire'

Data type of filter input port, specified as one of the following:

- 'std_logic_vector' or 'signed/unsigned' (when the target language is VHDL)
- 'wire' (when the target language is Verilog)

OutputPort — Name of filter output port
'filter_out' (default) | character vector | string scalar

Name of filter output port, specified as 'filter_out', a character vector, or a string scalar.

If you specify a value that is a reserved word in the target language, the coder adds the postfix _rsvd to this value. You can update the postfix value by using the ReservedWordPostfix property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

OutputType — Data type of filter output port
'Same as input data type' (default) | 'std_logic_vector' | 'signed/unsigned' | 'wire'

Data type of filter output port in generated HDL code, specified as one of the following:

- 'Same as input data type', 'std_logic_vector', or 'signed/unsigned' (when the target language is VHDL)
- 'wire' (when the target language is Verilog)

Resets

ResetInputPort — Name of filter reset port
'reset' (default) | character vector | string scalar

Name of filter reset port, specified as 'reset', a character vector, or a string scalar. Use the ResetAssertedLevel property to control the behaviour of this port.

If you specify a value that is a reserved word in the target language, the coder adds the postfix _rsvd to this value. You can update the postfix value by using the ReservedWordPostfix property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

RemoveResetFrom — Suppress generation of resets from shift registers
'none' (default) | 'ShiftRegister'

Suppress the generation of resets from the shift registers, specified as 'none' or 'ShiftRegister'. To omit reset signals from shift registers, set this property to 'ShiftRegister'. Disabling reset signals from shift registers can result in a more efficient FPGA implementation. For more details, see “Suppressing Generation of Reset Logic” on page 5-20.

`ResetAssertedLevel` — Asserted (active) level of reset input signal
'active-high' (default) | 'active-low'

Asserted (active) level of reset input signal, specified as one of the following:

- `'active-high'` — To reset registers in the filter design, the reset input signal must be driven high (1).

  For example, this code 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'` — To reset registers in the filter design, the reset input signal must be driven low (0).

  For example, this code 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'));
  ```

`ResetType` — Reset style for registers
'async' (default) | 'sync'

Reset style for registers, specified as one of the following:

- `'async'` — The coder uses asynchronous resets. The HDL process block does not check for an active clock before performing a reset. For example:

  ```
  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'` — The coder uses a synchronous reset style. In this case, the HDL process block checks for the rising edge of the clock before performing a reset. For example:

  ```
  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;
  ```

Identifiers and Comments

`BlockGenerateLabel` — Postfix to the block section labels
'_gen' (default) | character vector | string scalar

Postfix to the block section labels, specified as `'_gen'`, a character vector, or a string scalar. This property applies only when the target language is VHDL. The coder appends this postfix to the block section labels of VHDL `GENERATE` statements.

`InstanceGenerateLabel` — Postfix to the instance section labels
`'_gen'` (default) | character vector | string scalar

Postfix to the instance section labels, specified as `'_gen'`, a character vector, or a string scalar. This property applies only when the target language is VHDL. The coder appends this postfix to the instance section labels of VHDL `GENERATE` statements.

`OutputGenerateLabel` — Postfix to output assignment block labels
`'outputgen'` (default) | character vector | string scalar

Postfix to output assignment block labels, specified as `'outputgen'`, a character vector, or a string scalar. This property applies only when the target language is VHDL. The coder appends this postfix to the output assignment block labels of VHDL `GENERATE` statements.

`ClockProcessPostfix` — Postfix to HDL clock process names
`'_process'` (default) | character vector | string scalar

Postfix to HDL clock process names, specified as `'_process'`, a character vector, or a string scalar. The coder uses HDL process blocks to modify the content of the registers in the filter. The block label is derived from the register name and this postfix. For example, in the following block declaration, the coder derives the process label from the register name `delay_pipeline` and the default postfix `'_process'`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
```

`CoeffPrefix` — Prefix for filter coefficient names
`'coeff'` (default) | character vector | string scalar

Prefix for filter coefficient names, specified as `'coeff'`, a character vector, or a string scalar. The coder derives the coefficient names by appending filter-specific characteristics to this prefix.

Filter Type	Coefficient Name
FIR	The coder appends the coefficient number to `CoeffPrefix`, starting with 1. For example, the default for the first coefficient is `coeff1`.
IIR	The coder appends the following characters to `CoeffPrefix`: **1** underscore (_) **2** `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) **3** `_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:

```
firfilt = design(fdesign.lowpass,'equiripple', ...
    'FilterStructure','dfsymfir','SystemObject',true);
```

```
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'CoefficientSource','Internal','CoeffPrefix','mycoeff');
```

The coder replaces the default coefficient name prefix with the custom value:

```
ARCHITECTURE rtl OF firfilt IS
  -- Local Functions
  -- Type Definitions
  TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF signed(15 DOWNTO 0); -- sfix16_En15
  -- Constants
  CONSTANT mycoeff1   : signed(15 DOWNTO 0) := to_signed(-159, 16); -- sfix16_En16
  CONSTANT mycoeff2   : signed(15 DOWNTO 0) := to_signed(-137, 16); -- sfix16_En16
  CONSTANT mycoeff3   : signed(15 DOWNTO 0) := to_signed(444, 16); -- sfix16_En16
  CONSTANT mycoeff4   : signed(15 DOWNTO 0) := to_signed(1097, 16); -- sfix16_En16
  ...
```

Dependencies

This property applies only when you set `CoefficientSource` to `'Internal'`.

`ComplexImagPostfix` — Postfix to imaginary part of complex signal names

`'_im'` (default) | character vector | string scalar

Postfix to imaginary part of complex signal names, specified as `'_im'`, a character vector, or a string scalar. See “Using Complex Data and Coefficients” on page 5-26.

`ComplexRealPostfix` — Postfix to real part of complex signal names

`'_re'` (default) | character vector | string scalar

Postfix to real part of complex signal names, specified as `'_re'`, a character vector, or a string scalar. See “Using Complex Data and Coefficients” on page 5-26.

`EntityConflictPostfix` — Postfix to duplicate entity or module names

`'_block'` (default) | character vector | string scalar

Postfix to duplicate entity or module names, specified as `'_block'`, a character vector, or a string scalar. The coder appends this postfix to resolve duplicate VHDL entity or Verilog module names. For example, if the coder detects two entities with the name `MyFilt`, the coder names the first entity `MyFilt` and the second instance `MyFilt_block`.

`InstancePrefix` — Prefix for component instance name

`'u_'` (default) | character vector | string scalar

Prefix for component instance name, specified as `'u_'`, a character vector, or string scalar.

`PackagePostfix` — Postfix to VHDL package file name

`'_pkg'` (default) | character vector | string scalar

Postfix to VHDL package file name, specified as `'_pkg'`, a character vector, or a string scalar. The coder derives the package name by appending this postfix to the filter name. This option applies only if a package file is required for the design.

`ReservedWordPostfix` — Postfix to reserved words

`'_rsvd'` (default) | character vector | string scalar

Postfix to reserved words, specified as `'_rsvd'`, a character vector, or a string scalar. This property applies to name, postfix, or label values specified as a character vector or a string scalar in `Name,Value` pair arguments to `generatehdl`. If a specified value is a reserved word in the target language, the coder appends this postfix to the value. For example, if you call `generatehdl` with the argument pair `'Name','mod'`, the coder forms the name `mod_rsvd` in the generated filter code. See “Reserved Word Tables” on page 5-10.

`SplitEntityArch` — Split VHDL entity and architecture code

`'off'` (default) | `'on'`

Split VHDL entity and architecture code, specified as `'off'` or `'on'`. When this property is set to `'on'`, the coder generates the VHDL entity and architecture code of the filter in two separate files. The coder derives the file names from the filter name by appending the postfixes `_entity` and `_arch` to the base file name. To specify custom postfix values, set the `SplitEntityFilePostfix` and `SplitArchFilePostfix` properties.

`SplitArchFilePostfix` — Postfix to VHDL architecture file name

`'_arch'` (default) | character vector | string scalar

Postfix to VHDL architecture file name, specified as `'_arch'`, a character vector, or a string scalar.

Dependencies

This property applies only when you set `SplitEntityArch` to `'on'`.

`SplitEntityFilePostfix` — Postfix to VHDL entity file name

`'_entity'` (default) | character vector | string scalar

Postfix to VHDL entity file name, specified as `'_entity'`, a character vector, or a string scalar.

Dependencies

This property applies only when you set `SplitEntityArch` to `'on'`.

`UserComment` — Add user comments to generated HDL code

character vector | string scalar

Add user comments to generated HDL code, specified as a character vector or string vector. The user comments appear in the header comment block at the top of the generated files, preceded by leading comment characters specific to the target language. When you include new lines or line feeds in the user comments, the coder emits single-line comments for each new line. For example:

```
firfilt = dsp.FIRFilter;
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'UserComment','This is a comment line.\nThis is a second line.')
```

The resulting header comment block for the filter `firfilt` is as follows:

```
-- -------------------------------------------------------------
--
-- Module: firfilt
-- Generated by MATLAB(R) 9.1 and the Filter Design HDL Coder 3.1.
-- Generated on: 2016-11-08 15:28:25
-- This is a comment line.
-- This is a second line.
--
-- -------------------------------------------------------------

-- -------------------------------------------------------------
-- HDL Code Generation Options:
--
-- TargetLanguage: VHDL
-- Name: firfilt
-- InputDataType: numerictype(1,16,15)
-- UserComment:  User data, length 47
-- GenerateHDLTestBench: off

-- -------------------------------------------------------------
-- HDL Implementation    : Fully parallel
```

```
-- Folding Factor       : 1
-- -------------------------------------------------------------
-- Filter Settings:
--
-- Discrete-Time FIR Filter (real)
-- -------------------------------
-- Filter Structure  : Direct-Form FIR
-- Filter Length     : 2
-- Stable            : Yes
-- Linear Phase      : Yes (Type 2)
-- Arithmetic        : fixed
-- Numerator         : s16,15 -> [-1 1)
-- -------------------------------------------------------------
```

`VectorPrefix` — Prefix for VHDL vector signal names
`'vector_of_'` (default) | character vector | string scalar

Prefix for VHDL vector signal names, specified as `'vector_of_'`, a character vector, or a string scalar.

Tips

If you use the function `fdhdltool` to generate HDL code, you can set the corresponding properties in the Generate HDL dialog box.

Property	Location in Dialog Box
Input data type **Output data type** **Clock enable output port** **Input port** **Output port**	**Global Settings** tab > **Ports** tab
Additional port and identifier properties	Top section of **Global Settings** tab, and **Global Settings** tab > **General** tab

Check out the main **Global Settings** tab, and the **Ports** and **General** tabs of the **Global Settings** tab.

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

Topics

"Customization"

HDL Construct Properties

Customize HDL constructs in generated code

Description

With the HDL construct properties, you can customize VHDL and Verilog constructs in the generated code.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'CastBeforeSum','off');
```

Properties

HDL Coding Style

`CastBeforeSum` — Type casting before addition or subtraction

`'on'` | `'off'`

Type casting before addition or subtraction, specified as one of the following:

- `'on'` — The generated code type-casts input values of addition and subtraction operations to the desired result type before operating on the values. This setting produces numeric results that are typical of DSP processors.
- `'off'` — The generated code preserves the input value types during addition and subtraction operations and then converts the result to the desired type.

By default, the coder sets `CastBeforeSum` based on the **Cast signals before sum** Filter Designer setting of the filter object. Use this property to override the inherited setting, see "Relationship With Cast Before Sum in Filter Designer" on page 5-25. For System objects, the default setting depends on the filter type and structure.

`InlineConfigurations` — Generate inline VHDL configurations

`'on'` (default) | `'off'`

Generate inline VHDL configurations, specified as one of the following:

- `'on'` — The coder includes configurations for the filter entity within the generated VHDL code.
- `'off'` — The coder omits the generation of configurations. Use this option if you are creating your own VHDL configuration files.

`LoopUnrolling` — Loop unrolling in generated VHDL code

`'off'` (default) | `'on'`

Loop unrolling in generated VHDL code, specified as one of the following:

- `'off'` — The coder includes `FOR` and `GENERATE` loops in the generated VHDL code.
- `'on'` — The coder unrolls and omits `FOR` and `GENERATE` loops in the generated VHDL code. Use this option if your EDA tool does not support `GENERATE` loops.

`SafeZeroConcat` — Type-safe syntax for concatenated zeros

`'on'` (default) | `'off'`

Type-safe syntax for concatenated zeros, specified as one of the following:

- `'on'` — The coder uses the `'0' & '0'` syntax for concatenated zeros. This syntax is recommended because it is unambiguous.
- `'off'` — The coder uses the "`000000...`" syntax for concatenated zeros. This syntax can be easier to read and is more compact, but it can lead to ambiguous types.

`UseAggregatesForConst` — Represent constant values by aggregates

`'off'` (default) | `'on'`

Represent constant values by aggregates, specified as one of the following:

- `'off'` — The coder represents constants less than 32 bits as scalars, and constants greater than or equal to 32 bits as aggregates. The following example shows the default scalar declaration for constants of less than 32 bits.

```
CONSTANT coeff1: signed(15 DOWNTO 0) := to_signed(-60, 16); -- sfix16_En16
CONSTANT coeff2: signed(15 DOWNTO 0) := to_signed(-178, 16); -- sfix16_En16
```

- `'on'` — The coder represents constants by aggregates, including constants that are less than 32 bits wide. The following example shows constants of less than 32 bits declared as aggregates.

```
CONSTANT c1: signed(15 DOWNTO 0):= (5 DOWNTO 3 =>'0',1 DOWNTO 0 => '0',OTHERS =>'1');
CONSTANT c2: signed(15 DOWNTO 0):= (7 => '0',5 DOWNTO 4 =>'0',0 => '0',OTHERS =>'1');
```

`UseRisingEdge` — VHDL coding style to check for rising edges

`'off'` (default) | `'on'`

VHDL coding style to check for rising edges when operating on registers, specified as one of the following:

- `'off'` — The generated code checks for clock events when operating on registers. For example:

```
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 ;
```

- `'on'` — The generated code uses the VHDL `rising_edge` function to check for rising edges when operating on registers. For example:

```
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 ;
```

When the clock transitions from `'X'` to `'1'`, the two coding styles have different simulation behavior.

UseVerilogTimescale — Use Verilog `timescale compiler directive

'on' (default) | 'off'

Use Verilog `timescale compiler directive, specified as 'on' or 'off'. The `timescale directive provides a way of specifying different delay values for multiple modules in a Verilog file. When this property is set to 'off', the coder excludes the directive in the generated Verilog code.

Tips

If you use the fdhdltool function to generate HDL code, you can set the corresponding properties on the **Global Settings** > **Advanced** tab in the Generate HDL dialog box.

See Also

generatehdl | fdhdltool

Topics

"HDL Constructs" on page 5-21

HDL Test Bench Properties

Generate and customize HDL test bench

Description

With the HDL test bench properties, you can enable and customize test bench generation.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15), ...
    'GenerateHDLTestBench','on','MultifileTestBench','on');
```

Properties

General

GenerateHDLTestBench — Generate HDL test bench

'off' (default) | 'on'

Generate HDL test bench for your HDL filter code, specified as `'off'` or `'on'`. The test bench applies generated input stimuli to the generated filter code and compares the output with stored MATLAB simulation results.

TestBenchName — File name of generated test bench

*filtername*_tb (default) | character vector | string scalar

File name of generated test bench, specified as *`filtername`*`_tb`, a character vector, or a string scalar. *`filtername`* is the name of the generated VHDL entity or Verilog module. You can customize this name by setting the `Name` property. The coder adds a file type extension to the test bench name, as specified by the `VerilogFileExtension` or `VHDLFileExtension` properties. The test bench file is located in the folder specified by the `TargetDirectory` property.

If you specify a value that is a reserved word in the target language, the coder adds the postfix `_rsvd` to this value. You can update the postfix value by using the `ReservedWordPostfix` property. For more details, see “Resolving HDL Reserved Word Conflicts” on page 5-9.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

ErrorMargin — Error margin for test bench comparison in bits

4 (default) | positive integer

Error margin for test bench comparison in bits, specified as 4 or a positive integer. The test bench compares results with reference signals. These HDL optimizations can generate test bench code that produces numeric results that differ from the results produced by the original filter function:

- `CastBeforeSum`
- `OptimizeForHDL`
- `FIRAdderStyle` set to `'tree'` or `'pipelined'`
- `AddPipelineRegisters` with FIR, asymmetric FIR, and symmetric FIR filters

The error margin specifies an acceptable minimum number of bits by which the numeric results can differ before the test bench issues a warning.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`MultifileTestBench` — Generate multifile test bench

`'off'` (default) | `'on'`

Generate multifile test bench, specified as `'off'` or `'on'`. When this property is set to `'on'`, the coder generates separate files for test bench code, helper functions, and test bench data instead of a single file. The file names are derived from the `TestBenchName` and `TestBenchDataPostfix` properties. For example, if the name of the generated VHDL entity or Verilog module is `my_fir_filt`, the default test bench file names are:

- `my_fir_filt_tb` — Test bench code
- `my_fir_filt_tb_pkg` — Helper functions package
- `my_fir_filt_tb_data` — Test vector data package

The coder appends to these file names the file type extension defined by the `VerilogFileExtension` or `VHDLFileExtension` properties.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`TestBenchDataPostfix` — Postfix to file name of test bench data

`'_data'` (default) | character vector | string scalar

Postfix to file name of test bench data, specified as `'_data'`, a character vector, or a string scalar. The coder generates a test bench data file with a file name obtained by appending this postfix to the `TestBenchName` property value.

Dependencies

This property applies only when the `GenerateHDLTestBench` and `MultifileTestBench` properties are set to `'on'`.

`TestBenchReferencePostfix` — Postfix to reference signal names

`'_ref'` (default) | character vector | string scalar

Postfix to reference signal names, specified as `'_ref'`, a character vector, or string scalar. The coder applies this postfix to the reference output signal in the test bench. The coder represents reference signal data as arrays.

```
CONSTANT filter_out_expected : filter_in_data_log_type :=
    (
         -2.4228738523269194E-03,
```

```
        -2.0832449820793104E-03,
        6.7703446401186345E-03,...
```

For comparison, the test bench accesses one array value at a time.

```
 SIGNAL filter_out_ref                   : real := 0.0; -- double
...
 filter_out_ref <= filter_out_expected(TO_INTEGER(filter_out_addr));
```

Dependencies

This property applies only when the GenerateHDLTestBench property is set to 'on'.

Clocks and Resets

ClockHighTime — Period during which the test bench drives clock input signals high (1) in ns

5 (default) | positive scalar

Period during which the test bench drives clock input signals high (1) in ns, specified as 5 or a positive scalar. You can specify an integer or a double-precision floating-point value with a maximum of 6 significant digits after the decimal point.

Dependencies

This property applies only when the GenerateHDLTestBench and ForceClock properties are set to 'on'.

ClockLowTime — Period during which the test bench drives clock input signals low (0) in ns

5 (default) | positive scalar

Period during which the test bench drives clock input signals low (0) in ns, specified as 5 or a positive scalar. You can specify an integer or a double-precision floating-point value with a maximum of 6 significant digits after the decimal point.

Dependencies

This property applies only when the GenerateHDLTestBench and ForceClock properties are set to 'on'.

ForceClock — Test bench forces clock input signals

'on' (default) | 'off'

Test bench forces clock input signals, specified as one of these values.

- 'on' — The test bench forces the clock input signals. The values of the ClockHighTime and ClockLowTime properties control the clock waveform.
- 'off' — You must drive the clock input signals from an external source.

Dependencies

This property applies only when the GenerateHDLTestBench property is set to 'on'.

ForceClockEnable — Test bench forces clock enable input signals

'on' (default) | 'off'

Test bench forces clock enable input signals, specified as one of these values.

- `'on'` — The test bench forces the clock enable input signals. The polarity is active high (1). This signal also obeys the setting of the `HoldTime` property.
- `'off'` — You must drive the clock enable input signals from an external source.

Dependencies

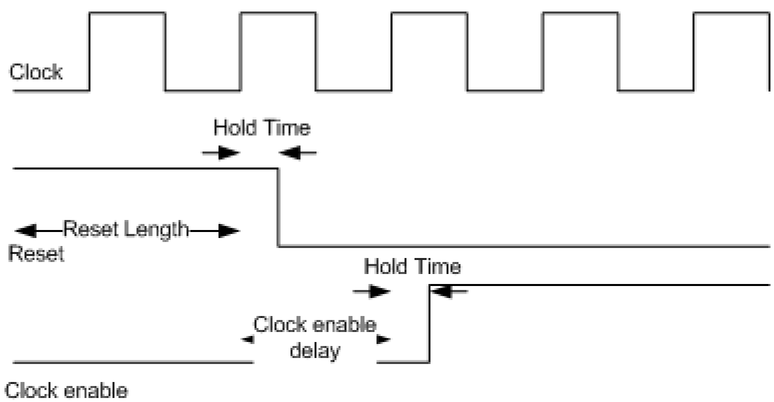
This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`TestBenchClockEnableDelay` — Clock cycles between deassertion of reset and assertion of clock enable

1 (default) | positive integer

Clock cycles between deassertion of reset and assertion of clock enable, specified as `1` or a positive integer. The test bench waits this number of cycles between deasserting the reset signal and asserting the clock enable signal. The `HoldTime` property also applies.

In the figure, the test bench deasserts an active-high reset signal after the interval labeled `Hold Time`. The test bench then asserts clock enable after a further interval, labeled `Clock enable delay`.



Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`ForceReset` — Test bench forces the reset input signals

`'on'` (default) | `'off'`

Test bench forces the reset input signals, specified as one of these values.

- `'on'` — The test bench forces the reset input signals. You can also specify a hold time to control the timing of reset by setting the `HoldTime` property.
- `'off'` — You must drive the reset input signals from an external source.

Dependencies

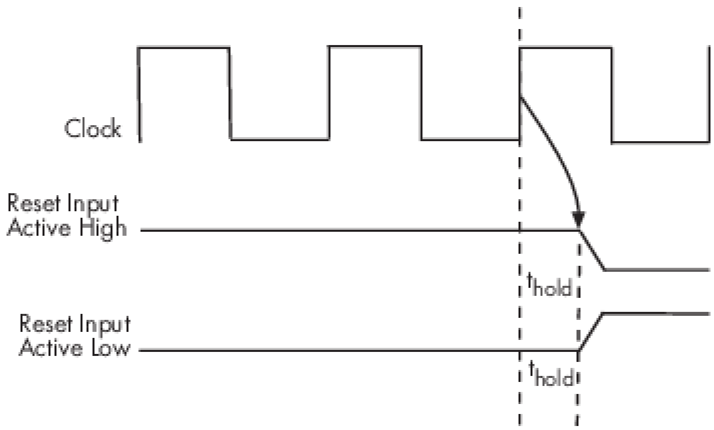
This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`HoldTime` — Hold time for input data values and forced reset signals in ns

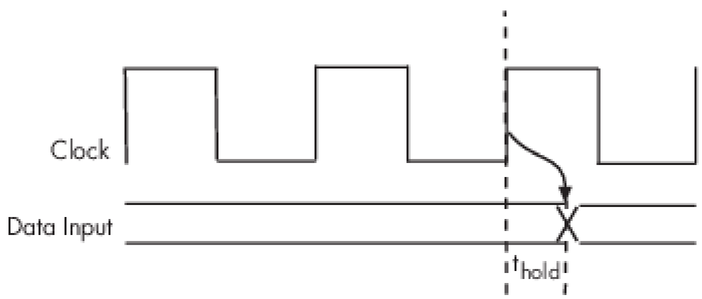
2 (default) | positive scalar

Hold time for input data values and forced reset signals in ns, specified as 2 or a positive scalar. The test bench holds filter data input signals and forced reset input signals for the specified time interval past the rising clock edge. You can specify an integer or a double-precision floating-point value with a maximum of 6 significant digits after the decimal point.

These figures show the application of a hold time, t_{hold}, for reset and data input signals. The signals are forced to active high and active low. The `ResetLength` property is set to 2 cycles, and the test bench asserts the reset signal for a total of 2 cycles plus t_{hold}.



Hold Time for Reset Input Signals



Hold Time for Data Input Signals

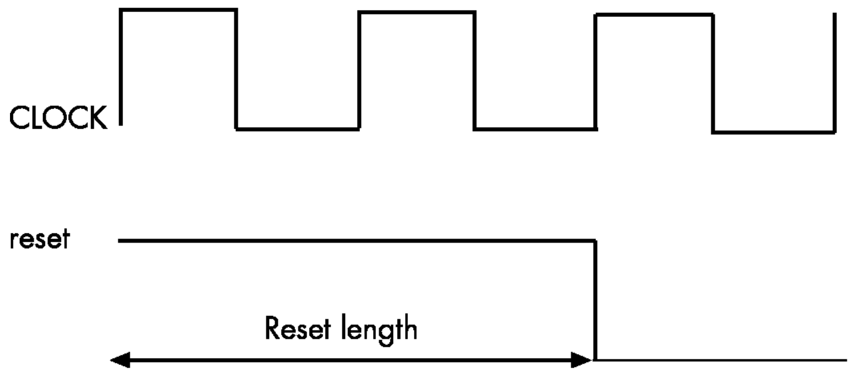
Dependencies

This property applies only when the `GenerateHDLTestBench` and `ForceReset` properties are set to `'on'`.

`ResetLength` — Number of clock cycles that the test bench asserts the reset signal
2 (default) | positive integer

Number of clock cycles that the test bench asserts the reset signal, specified as `2` or a positive integer.

The default test bench asserts an active-high reset signal for 2 clock cycles.



Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`HoldInputDataBetweenSamples` — Test bench holds input data of over-clocked filters in a valid state

`'off'` (default) | `'on'`

Test bench holds input data of over-clocked filters in a valid state, specified as `'off'` or `'on'`. Serial architectures and distributed arithmetic architectures implement internal clock rates higher than the input rate. In such filter implementations, the base clock runs `N` cycles (`N >= 2`) for each input sample. This property relates to the number of clock cycles that the test bench holds the input data in a valid state.

- `'off'` — The test bench holds data values in a valid state for one clock cycle. For the next `N-1` cycles, data is in an unknown state (expressed as `'X'`). Forcing the input data to an unknown state verifies that the generated filter code registers the input data only on the first cycle.
- `'on'` — The test bench holds input data values in a valid state across `N` clock cycles.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`InitializeTestBenchInputs` — Initialize test bench input

`'off'` (default) | `'on'`

Initialize test bench input, specified as one of these values.

- `'off'` — At the start of the simulation, the test bench drives an unknown state (expressed as `'X'`) to the input ports.
- `'on'` — At the start of the simulation, the test bench drives zeros to the input ports.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

Stimulus

TestBenchStimulus — Input stimuli applied to generated filter
{'impulse','step','ramp','chirp','noise'} (default) | cell array of character vectors | string array

Input stimuli applied to generated filter, specified as {'impulse','step','ramp','chirp','noise'}, a cell array of character vectors, or a string array. The cell or string array must be a subset of the default set of stimuli. You can specify combinations of stimuli in any order. For example:

```
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'GenerateHDLTestbench','on', ...
    'TestBenchStimulus',{'ramp','impulse','noise'})
```

You can specify a custom input stimulus by using the TestBenchUserStimulus property. When TestBenchUserStimulus is a nonempty vector, it takes priority over TestBenchStimulus.

Dependencies

This property applies only when the GenerateHDLTestBench property is set to 'on'.

TestBenchUserStimulus — Custom input stimulus
[] (empty vector) (default) | vector of input data

Custom input stimulus, specified as one of these values.

- [] (empty vector) — The test bench uses the TestBenchStimulus property to generate input data.
- Vector of input data — The test bench applies this input stimulus to the generated filter. You can specify the vector as a function call returning a vector.

  For example, this 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)
  ```

  Specify this stimulus when calling generatehdl.

  ```
  generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
     'GenerateHDLTestbench','on', ...
     'TestBenchUserStimulus',repmat([1 1 1 1 0 0 0 0],1,10))
  ```

Dependencies

This property applies only when the GenerateHDLTestBench property is set to 'on'.

TestBenchCoeffStimulus — Coefficient stimulus for FIR or IIR filters
[] (empty vector) (default) | vector of coefficients (FIR filters only) | cell array of coefficient and scale values (IIR filters only)

Coefficient stimulus for FIR or IIR filters, specified as one of these values.

- [] (empty vector) — The test bench uses the filter object coefficients and forces the input stimuli. This sequence shows the response to the input stimuli and verifies that the interface writes one set of coefficients into the coefficient memory as expected.

- Vector of coefficients (FIR filters only) — The filter processes the input stimuli twice: once with the filter object coefficients and once with the coefficient stimulus. The test bench verifies that the interface writes two different sets of coefficients into the coefficient memory. For more details, see “Generating a Test Bench for Programmable FIR Coefficients” on page 3-25.
- Cell array of coefficient and scale values (IIR filters only) — Specify the stimulus as a column vector of scale values and a second-order section (SOS) matrix. The filter processes the input stimuli twice: once with the filter object coefficients and once with the coefficient stimulus. The test bench verifies that the interface writes two different sets of coefficients into the coefficient memory. For more details, see “Generating a Test Bench for Programmable IIR Coefficients” on page 3-31.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'` and the `CoefficientSource` property is set to `'ProcessorInterface'`.

`TestBenchFracDelayStimulus` — Fractional delay stimulus for single-rate Farrow filters

constant numeric value (default) | vector of numeric values | `'RandSweep'` | `'RampSweep'`

Fractional delay stimulus for single-rate Farrow filters, specified as one of these values.

- Constant numeric value— The test bench drives the fractional delay input signal with a constant value obtained from the filter object.
- Vector of numeric values — The test bench drives the fractional delay input signal from this vector. You can specify the vector as a function call returning a vector. The vector must be of the same length as the test bench input signal.
- `'RandSweep'` — The test bench drives the fractional delay input signal by using a vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the input signal to the filter, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.
- `'RampSweep'` — The test bench drives the fractional delay input signal by using a vector of random values from 0 through 1. This stimulus signal has the same duration as the input signal to the filter, but it changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.

See “Code Generation Properties for Farrow Filters” on page 3-19.

Dependencies

This property applies only when the `GenerateHDLTestBench` property is set to `'on'`.

`TestBenchRateStimulus` — Rate input stimulus for CIC filters

maximum rate change factor (default) | integer

Rate input stimulus for Cascaded Integrator-Comb (CIC) filters, specified as the maximum rate change factor or an integer. If you do not specify `TestBenchRateStimulus`, the coder assumes that the filter is designed with the maximum rate expected. The decimation factor (for CIC decimators) or interpolation factor (for CIC interpolators) is set to this maximum rate-change factor.

See “Variable Rate CIC Filters” on page 3-6.

Dependencies

This property applies only to variable-rate CIC filters, when the `GenerateHDLTestBench` and `AddRatePort` properties are set to `'on'`.

Cosimulation

`GenerateCosimBlock` — Generate Simulink model of HDL Cosimulation blocks
`'off'` (default) | `'on'`

Generate Simulink model of HDL Cosimulation blocks, specified as `'off'` or `'on'`. The generated Simulink model contains two HDL Cosimulation blocks: one for Mentor Graphics ModelSim and one for Cadence Incisive. The coder configures these blocks to conform to the port and data type interface of the selected filter. Use these blocks to cosimulate your design with the desired HDL simulator in Simulink.

Dependencies

This feature requires an HDL Verifier license.

`GenerateCosimModel` — Generate Simulink model of realized filter and HDL Cosimulation block
`'none'` (default) | `'ModelSim'` | `'Incisive'`

Generate Simulink model of realized filter and HDL Cosimulation block, specified as `'none'`, `'ModelSim'`, or `'Incisive'`. When you set this property to `'ModelSim'` or `'Incisive'`, the coder generates and opens a Simulink model. The model contains an HDL cosimulation block for the selected simulator, and a behavioral implementation of the filter design. The model applies generated input stimuli and compares the output of the EDA simulator with the output of the behavioral filter subsystem. You can customize the input stimulus and error margin using the same properties as you would for the generated HDL test bench.

See "Generating a Simulink Model for Cosimulation with an HDL Simulator" on page 6-22.

Dependencies

This feature requires an HDL Verifier license.

Tips

If you use the function `fdhdltool` to generate HDL code, you can set the corresponding properties on the **Test Bench** tab in the Generate HDL tool.

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

Topics

"Enabling Test Bench Generation" on page 6-7
"Testing with an HDL Test Bench" on page 6-2

HDL Synthesis and Workflow Automation Properties

Integrate third-party EDA tools into filter design workflow

Description

With the synthesis and workflow automation properties, you can enable and customize the generation of script files for third-party Electronic Design Automation (EDA) tools.

These scripts let you compile, simulate, and synthesize the generated HDL code. You can modify the commands that the coder prints to the scripts by setting the properties on this page. The coder passes the property values to `fprintf` to create the script. You can use control characters supported by the `fprintf` function. For example, `'\n'` inserts a new line into the script file.

Specify these properties as name-value arguments to the `generatehdl` function. `Name` is the property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `'Name1',Value1,...,'NameN',ValueN`.

For example:

```
fir = dsp.FIRFilter('Structure','Direct form antisymmetric');
generatehdl(fir,'InputDataType',numerictype(1,16,15),'VHDLLibraryName','my_work');
```

Properties

Generate Scripts

`EDAScriptGeneration` — Enable script generation for EDA tools

`'on'` (default) | `'off'`

Enable script generation for EDA tools, specified as one of the following:

- `'on'` — The coder generates a compilation script for Mentor Graphics ModelSim. In addition:
  - If the `GenerateHDLTestBench` property is set to `'on'`, the coder generates compilation and simulation scripts for the test bench.
  - If the `HDLSynthTool` property is set to a value other than `'none'`, the coder generates a synthesis script.
- `'off'` — Disables script generation, including compilation, simulation, and synthesis scripts.

See “Integration with Third-Party EDA Tools” on page 6-28.

Compilation Scripts

`VHDLLibraryName` — Library name used in initialization section of compilation script

`'work'` (default) | character vector | string scalar

Library name used in initialization section of compilation script, specified as `'work'`, a character vector, or a string scalar. Use this property to avoid library name conflicts with your existing VHDL code. The coder inserts this name into the HDLCompileInit property value. By default, the coder generates the library specification 'vlib work/n'.

HDLCompileFilePostfix — Postfix to file name of compilation script

'_compile.do' (default) | character vector | string scalar

Postfix to file name of compilation script, specified as '_compile.do', a character vector, or a string scalar. The coder derives the name of the script by appending this postfix to the generated filter name or test bench name. For example, if the generated filter name is my_design, the coder adds the postfix _compile.do to form the name my_design_compile.do.

HDLCompileInit — Initialization section of compilation script

'vlib %s\n' (default) | character vector | string scalar

Initialization section of compilation script, specified as 'vlib %s\n', a character vector, or a string scalar. The coder prints this command to the beginning of the compilation script. The implicit argument, %s, is the name of the library specified by the VHDLLibraryName property. By default, this property generates the library specification 'vlib work/n'. If you compile your filter design with code from other libraries, update the VHDLLibraryName property to avoid library name conflicts.

HDLCompileVerilogCmd — Command written to compilation script for each Verilog file

'vlog %s %s\n' (default) | character vector | string scalar

Command written to compilation script for each Verilog file, specified as 'vlog %s %s\n', a character vector, or a string scalar. This command adds the generated HDL source file to the list of files to be compiled. The coder prints this command to the script once for each generated HDL file. The first implicit argument, %s, takes the value of the SimulatorFlags property. The second implicit argument is the file name of the current module.

HDLCompileVHDLCmd — Command written to compilation script for each VHDL file

'vcom %s %s\n' (default) | character vector | string scalar

Command written to compilation script for each VHDL file, specified as 'vcom %s %s\n', a character vector, or a string scalar. This command adds the generated HDL source file to the list of files to be compiled. The coder prints this command to the script once for each generated HDL file. The first implicit argument, %s, takes the value of the SimulatorFlags property. The second implicit argument is the file name of the current entity.

SimulatorFlags — Simulator options

'' (default) | character vector | string scalar

Simulator options, specified as '', a character vector, or a string scalar. Specify options that are specific to your application and the simulator you are using. For example, if you use the 1076-1993 VHDL compiler, specify the flag '-93'. The coder adds the flags you specify with this option to the compilation command in the generated EDA tool scripts. The HDLCompileVHDLCmd or HDLCompileVerilogCmd properties determine the compilation command.

HDLCompileTerm — Termination section of compilation script

'' (default) | character vector | string scalar

Termination section of compilation script, specified as '', a character vector, or a string scalar. The coder prints this character sequence to the end of the compilation script.

Simulation Scripts

`HDLSimFilePostfix` — Postfix to file name of simulation script

`'_sim.do'` (default) | character vector | string scalar

Postfix to file name of simulation script, specified as `'_sim.do'`, a character vector, or a string scalar. The coder derives the name of the script by appending this postfix to the generated test bench name. For example, if the name of the test bench is `my_design_tb`, the coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

Dependencies

This property applies only when the `EDAScriptGeneration` and `GenerateHDLTestBench` properties are set to `'on'`.

`HDLSimInit` — Initialization section of simulation script

`'onbreak resume\nonerror resume\n'` (default) | character vector | string scalar

Initialization section of simulation script, specified as `'onbreak resume\nonerror resume\n'`, a character vector, or a string scalar. The coder prints this command to the beginning of the simulation script.

Dependencies

This property applies only when the `EDAScriptGeneration` and `GenerateHDLTestBench` properties are set to `'on'`.

`HDLSimCmd` — Command written to simulation script

`'-voptargs=+acc %s.%s\n'` (default) | character vector | string scalar

Command written to simulation script, specified as a character vector or a string scalar to be combined with `vsim`, like `vsim -voptargs=+acc %s.%s\n`. The first implicit argument, `%s`, is the library name. The second implicit argument is the generated test bench name. For Verilog, the library name is `'work'` and cannot be changed. For VHDL, the library name is the value of the `VHDLLibraryName` property. If you compile your filter design with code from other libraries, update `VHDLLibraryName` to avoid library name conflicts.

Note Prior to R2020b, the default HDL simulation command was `vsim -novopt %s.%s\n`. Mentor Graphics ModelSim versions prior to 10.7 support the former syntax. If you use a more recent ModelSim version, you must use the `-voptargs=+acc` syntax.

Dependencies

This property applies only when the `EDAScriptGeneration` and `GenerateHDLTestBench` properties are set to `'on'`.

`HDLSimViewWaveCmd` — Waveform-viewing command written to simulation script

`'add wave sim:%s\n'` (default) | character vector | string scalar

Waveform-viewing command written to simulation script, specified as `'add wave sim:%s\n'`, a character vector, or a string scalar. The implicit argument, `%s`, is a command that adds the signal paths for the DUT top-level input signals, output signals, and output reference signals.

Dependencies

This property applies only when the EDAScriptGeneration and GenerateHDLTestBench properties are set to 'on'.

HDLSimTerm — Termination section of simulation script

'run -all\n' (default) | character vector | string scalar

Termination section of simulation script, specified as 'run -all\n', a character vector, or a string scalar. The coder prints this command to the end of the simulation script.

Dependencies

This property applies only when the EDAScriptGeneration and GenerateHDLTestBench properties are set to 'on'.

Synthesis Scripts

HDLSynthTool — Generate script for synthesis tool

'none' (default) | 'Vivado' | 'ISE' | 'Libero' | 'Precision' | 'Quartus' | 'Synplify' | 'Custom'

Generate script for synthesis tool, specified as one of the following.

HDLSynthTool Value	Synthesis Tool
'none'	N/A. The coder does not generate a synthesis script.
'Vivado'	Xilinx® Vivado®
'ISE'	Xilinx ISE
'Libero'	Microsemi™ Libero®
'Precision'	Mentor Graphics Precision
'Quartus'	Altera® Quartus II
'Synplify'	Synopsys® Synplify Pro®
'Custom'	The coder generates a script that supports your tool, based on the settings of HDLSynthCmd, HDLSynthInit, and HDLSynthTerm properties.

When generating the script, the coder uses tool-specific values set by the HDLSynthCmd, HDLSynthInit, and HDLSynthTerm properties. Customize these properties according to your target device and constraints.

Dependencies

This property applies only when the EDAScriptGeneration property is set to 'on'.

HDLSynthFilePostfix — Postfix to file name of synthesis script

character vector | string scalar

Postfix to file name of synthesis script, specified as a character vector or a string scalar. The coder derives the name of the script by appending this postfix to the generated filter name.

The default postfix value depends on the synthesis tool specified by the HDLSynthTool property. For example, if the value of HDLSynthTool is 'Synplify', then HDLSynthFilePostfix defaults to '_synplify.tcl'. Therefore, if the generated filter name is my_design, the coder adds the postfix _synplify.tcl to form the synthesis script file name my_design_synplify.tcl.

HDLSynthTool Value	Default HDLSynthFilePostfix Value
none	N/A
'Vivado'	'_vivado.tcl'
'ISE'	'_ise.tcl'
'Libero'	'_libero.tcl'
'Precision'	'_precision.tcl'
'Quartus'	'_quartus.tcl'
'Synplify'	'_synplify.tcl'
'Custom'	'_custom.tcl'

Dependencies

This property applies only when the EDAScriptGeneration property is set to 'on' and the HDLSynthTool property is set to a value other than 'none'.

HDLSynthInit — Initialization section of synthesis script

character vector | string scalar

Initialization section of synthesis script, specified as a character vector or a string scalar. The default value of this property depends on the synthesis tool specified by the HDLSynthTool property. For example, if you set HDLSynthTool to 'ISE', this property defaults to:

```
set src_dir [pwd]\nset prj_dir "synprj"\n
file mkdir ../$prj_dir\n
cd ../$prj_dir\n
project new %s.xise\n
project set family Virtex4\n
project set device xc4vsx35\n
project set package ff668\n
project set speed -10\n
```

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

Dependencies

This property applies only when the EDAScriptGeneration property is set to 'on' and the HDLSynthTool property is set to a value other than 'none'.

HDLSynthCmd — Command written to synthesis script for each HDL file

character vector | string scalar

Command written to synthesis script for each HDL file, specified as a character vector or a string scalar. The command adds the generated HDL source file to the list of files to be compiled. The coder prints this command to the script once for each generated HDL file. The default value of this property depends on the synthesis tool specified by the HDLSynthTool property. For example, when HDLSynthTool is set to 'Quartus', this property defaults to 'set_global_assignment -name %s_FILE "$src_dir/%s"\n'. The first implicit argument is the TargetLanguage. The second implicit argument is the name of the HDL file.

Dependencies

This property applies only when the `EDAScriptGeneration` property is set to `'on'` and the `HDLSynthTool` property is set to a value other than `'none'`.

`HDLSynthTerm` — Termination section of synthesis script

character vector | string scalar

Termination section of synthesis script, specified as a character vector or a string scalar. The coder prints this character sequence to the end of the synthesis script. The default value depends on the synthesis tool specified by the `HDLSynthTool` property. For example, if you set `HDLSynthTool` to `'Synplify'`, this property defaults to:

```
set_option -technology VIRTEX4\n
set_option -part XC4VSX35\n
set_option -synthesis_onoff_pragma 0\n
set_option -frequency auto\n
project -run synthesis\n
```

Dependencies

This property applies only when the `EDAScriptGeneration` property is set to `'on'` and the `HDLSynthTool` property is set to a value other than `'none'`.

Tips

If you use the function `fdhdltool` to generate HDL code, you can set the corresponding properties in the Generate HDL dialog box.

Property	Location in Dialog Box
Compilation script	**EDA Tool Scripts** tab, left pane. (`VHDLLibraryName` property does not have a corresponding option in the dialog box.)
Simulation script	**EDA Tool Scripts** tab, left pane. To access simulation flags, see **Test Bench > Configuration** tab.
Synthesis script	**EDA Tool Scripts** tab, left pane.

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

Topics

"Integration with Third-Party EDA Tools" on page 6-28

10

Functions

fdhdltool

Open Generate HDL dialog box

Syntax

```
fdhdltool(filtSO,nt)
fdhdltool(filtSO,nt,fd)
fdhdltool(filterObj)
```

Description

`fdhdltool(filtSO,nt)` opens the Generate HDL dialog box to set options and generate HDL code for the specified filter System object and the input data type, specified by `nt`.

When the dialog box opens, it displays default values for code generation options that apply to the filter. You can then specify code generation options and generate HDL code. You can also use this dialog box to generate HDL test bench code and scripts for third-party EDA tools.

`fdhdltool` operates on a copy of the filter, not the original object in the workspace. After you call `fdhdltool`, changes made to the original filter do not apply to the copy. The Generate HDL dialog box does not update either. The naming convention for the copied filter is *`filt`*`_copy`, where *`filt`* is the name of the original filter.

`fdhdltool(filtSO,nt,fd)` opens the Generate HDL dialog box to set options and generate HDL code for a `dsp.VariableFractionalDelay` filter System object. Specify the input data type by `nt`, and the fractional delay data type by `fd`.

`fdhdltool(filterObj)` opens the Generate HDL dialog box to set options and generate HDL code for the specified `dfilt` filter object.

Examples

Open Generate HDL Dialog Box for FIR Equiripple Filter

Design a direct form symmetric equiripple filter with these specifications:

- Passband frequency of 20 kHz
- Stopband frequency of 24 kHz
- Passband ripple of 0.01 dB
- Stopband attenuation of 80 dB
- Sampling frequency of 96 kHz

The `design` function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
filtSpecs = fdesign.lowpass(20e3,24e3,0.01,80,96e3);
FIRLowpass = design(filtSpecs,'equiripple','FilterStructure','dfsymfir','SystemObject',true)
```

```
FIRLowpass =
  dsp.FIRFilter with properties:
```

```
           Structure: 'Direct form symmetric'
     NumeratorSource: 'Property'
           Numerator: [1.0908e-04 2.1016e-05 -2.3369e-04 -2.8798e-04 9.0899e-05 3.6470e-04 -5.32
   InitialConditions: 0

  Show all properties
```

When the filter is a System object, you must specify a fixed-point data type for the input data.

```
T = numerictype(1,16,15);
```

Open the `Generate HDL` dialog box by passing the filter and the data type as arguments.

```
fdhdltool(FIRLowpass,T)
```

Input Arguments

`filtSO` — Filter
filter System object

Filter from which to generate HDL code, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. You can use the following System objects from DSP System Toolbox:

Single Rate Filters

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`
- `dsp.FilterCascade`
- `dsp.VariableFractionalDelay`

Multirate Filters

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FarrowRateConverter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FilterCascade`
- `dsp.DigitalDownConverter`
- `dsp.DigitalUpConverter`

`nt` — Input data type

`numerictype` object

Input data type, specified as a `numerictype` object. This argument applies only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`fd` — Fractional delay data type

`numerictype` object

Fractional delay data type, specified as a `numerictype` object. This argument applies only when the input filter is a `dsp.VariableFractionalDelay` System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`filterObj` — Filter

`dfilt` object

Filter from which to generate HDL code, specified as a `dfilt` object. You can create this object by using the `design` function. For an overview of supported filter features, see "Filter Configuration Options".

Version History

Introduced in R2007a

See Also

`generatehdl` | `generatetbstimulus`

Topics

"Opening the Filter Design HDL Coder UI Using the fdhdltool Command" on page 2-9

generatehdl

Generate HDL code for quantized filter

Syntax

```
generatehdl(filtSO,'InputDataType',nt)
generatehdl(filtSO,'InputDataType',nt,'FractionalDelayDataType',fd)
generatehdl(filterObj)

generatehdl( ___ ,Name,Value)
```

Description

`generatehdl(filtSO,'InputDataType',nt)` generates HDL code for the specified filter System object and the input data type, `nt`.

The generated file is a single source file that includes the entity declaration and architecture code. You can find this file in your current working folder, inside the `hdlsrc` subfolder.

`generatehdl(filtSO,'InputDataType',nt,'FractionalDelayDataType',fd)` generates HDL code for a `dsp.VariableFractionalDelay` filter System object. Specify the input data type, `nt`, and the fractional delay data type, `fd`.

`generatehdl(filterObj)` generates HDL code for the specified `dfilt` filter object using default settings.

`generatehdl( ___ ,Name,Value)` uses optional name-value arguments, in addition to the input arguments in previous syntaxes. Use these properties to override default HDL code generation settings.

- To customize filter name, destination folder, and to specify target language, see Fundamental HDL Code Generation Properties.
- To configure coefficients, complex input ports, and optional ports for specific filter types, see HDL Filter Configuration Properties.
- To optimize the speed or area of generated HDL code, see HDL Optimization Properties.
- To customize ports, identifiers, and comments, see HDL Ports and Identifiers Properties.
- To customize HDL constructs, see HDL Constructs Properties.
- To generate and customize test bench, see HDL Test Bench Properties.
- To integrate third-party EDA tools into the filter design workflow, see Synthesis and Workflow Automation Properties.

Examples

Generate HDL Code for FIR Equiripple Filter

Design a direct form symmetric equiripple filter with these specifications:

- Normalized passband frequency of 0.2
- Normalized stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

The `design` function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
filtSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
FIRe = design(filtSpecs,'equiripple','FilterStructure','dfsymfir','SystemObject',true)
```

```
FIRe = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [-0.0011 -0.0017 -0.0025 -0.0031 -0.0031 -0.0024 -9.7703e-04 0.0010 0.0031
    InitialConditions: 0

  Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input using the “InputDataType” on page 9-0 property. The coder generates the file `MyFilter.vhd` in the default target folder, `hdlsrc`.

```
generatehdl(FIRe,'InputDataType',numerictype(1,16,15),'Name','MyFilter');
```

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex48836167\hdlsrc\M
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for filter: MyFilter
### HDL latency is 2 samples
```

Generate HDL Code and Test Bench for FIR Equiripple Filter

Design a direct form symmetric equiripple filter with these specifications:

- Normalized passband frequency of 0.2
- Normalized stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

The `design` function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
filtSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
FIRe = design(filtSpecs,'equiripple','FilterStructure','dfsymfir','SystemObject',true)
```

```
FIRe = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
```

```
          Numerator: [-0.0011 -0.0017 -0.0025 -0.0031 -0.0031 -0.0024 -9.7703e-04 0.0010 0.0031
    InitialConditions: 0

  Show all properties
```

Generate VHDL code and a VHDL test bench for the FIR equiripple filter. When the filter is a System object, you must specify a fixed-point data type for the input data type. The coder generates the files `MyFilter.vhd` and `MyFilterTB.vhd` in the default target folder, `hdlsrc`.

```
generatehdl(FIRe,'InputDataType',numerictype(1,16,15),'Name','MyFilter',...
    'GenerateHDLTestbench','on','TestBenchName','MyFilterTB')
```

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex63281302\hdlsrc\M
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### Successful completion of VHDL code generation process for filter: MyFilter
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 4486 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex632813
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

Generate HDL Code for Fully Parallel FIR Filter with Programmable Coefficients

Design a direct form symmetric equiripple filter with fully parallel (default) architecture and programmable coefficients. The `design` function returns a `dsp.FIRFilter` System object™ with default lowpass filter specification.

```
firfilt = design(fdesign.lowpass,'equiripple','FilterStructure','dfsymfir','SystemObject',true)
```

```
firfilt =
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [-0.0024 -0.0021 0.0068 0.0167 0.0111 -0.0062 -0.0084 0.0093 0.0130 -0.010
    InitialConditions: 0

  Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data. To generate a processor interface for the coefficients, you must specify an additional name-value pair argument.

```
generatehdl(firfilt,'InputDataType',numerictype(1,16,15),'CoefficientSource','ProcessorInterface
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex74213987\hdlsrc\f
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
```

```
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
```

The coder generates this VHDL entity for the filter object.

```
ENTITY firfilt 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
         write_enable    :   IN    std_logic;
         write_done      :   IN    std_logic;
         write_address   :   IN    std_logic_vector(4 DOWNTO 0); -- ufix5
         coeffs_in       :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En16
         filter_out      :   OUT   std_logic_vector(36 DOWNTO 0)  -- sfix37_En31
         );

END firfilt;
```

Generate Partly Serial FIR Filter with Programmable Coefficients

Create a direct form antisymmetric filter with coefficients:

```
coeffs = fir1(22,0.45);
firfilt = dsp.FIRFilter('Numerator',coeffs,'Structure','Direct form antisymmetric')
```

```
firfilt = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form antisymmetric'
      NumeratorSource: 'Property'
            Numerator: [3.6133e-04 0.0031 8.4473e-04 -0.0090 -0.0072 0.0203 0.0272 -0.0341 -0.07
    InitialConditions: 0

  Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data. To generate a partly serial architecture, specify a serial partition. To enable `CoefficientMemory` property, you must set `CoefficientSource` to `ProcessorInterface`.

```
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'SerialPartition',[7 4],'CoefficientMemory','DualPortRAMs', ...
    'CoefficientSource','ProcessorInterface')
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex21465785\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 7 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

The generated code includes a dual-port RAM interface for the programmable coefficients.

```
ENTITY firfilt 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
         write_enable  :   IN    std_logic;
         write_done    :   IN    std_logic;
         write_address :   IN    std_logic_vector(3 DOWNTO 0); -- ufix4
         coeffs_in     :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En16
         filter_out    :   OUT   std_logic_vector(35 DOWNTO 0)  -- sfix36_En31
         );

END firfilt;
```

Generate Serial Partitions for FIR Filter

Explore clock rate and latency for different serial implementations of the same filter. Using a symmetric structure also allows the filter logic to share multipliers for symmetric coefficients.

Create a direct form symmetric FIR filter with these specifications:

- Filter order 13
- Normalized cut-off frequency of 0.4 for the 6-dB point

The `design` function returns a `dsp.FIRFilter` System object™ that implements the specification.

```
FIR = design(fdesign.lowpass('N,Fc',13,.4),'FilterStructure','dfsymfir','SystemObject',true)
```

```
FIR = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [0.0037 0.0045 -0.0115 -0.0417 1.0911e-17 0.1776 0.3674 0.3674 0.1776 1.09
    InitialConditions: 0

  Show all properties
```

To generate HDL code, call the `generatehdl` function. When the filter is a System object, you must specify a fixed-point data type for the input data.

For a baseline comparison, first generate a default fully parallel architecture.

```
generatehdl(FIR,'Name','FullyParallel', ...
    'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: FullyParallel
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\
### Starting generation of FullyParallel VHDL entity
### Starting generation of FullyParallel VHDL architecture
```

```
### Successful completion of VHDL code generation process for filter: FullyParallel
### HDL latency is 2 samples
```

Generate a fully serial architecture by setting the partition size to the effective filter length. The system clock rate is six times the input sample rate. The reported HDL latency is one sample greater than the default parallel implementation.

```
generatehdl(FIR,'SerialPartition',6,'Name','FullySerial', ...
    'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: FullySerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\F
### Starting generation of FullySerial VHDL entity
### Starting generation of FullySerial VHDL architecture
### Clock rate is 6 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: FullySerial
### HDL latency is 3 samples
```

Generate a partly serial architecture with three equal partitions. This architecture uses three multipliers. The clock rate is two times the input rate, and the latency is the same as the default parallel implementation.

```
generatehdl(FIR,'SerialPartition',[2 2 2],'Name','PartlySerial', ...
    'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: PartlySerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\P
### Starting generation of PartlySerial VHDL entity
### Starting generation of PartlySerial VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: PartlySerial
### HDL latency is 3 samples
```

Generate a cascade-serial architecture by enabling accumulator reuse. Specify the three partitions in descending order of size. Notice that the clock rate is higher than the rate in the partly serial (without accumulator reuse) example.

```
generatehdl(FIR,'SerialPartition',[3 2 1],'ReuseAccum','on','Name','CascadeSerial', ...
    'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\C
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: CascadeSerial
### HDL latency is 3 samples
```

You can also generate a cascade-serial architecture without specifying the partitions explicitly. The coder automatically selects partition sizes.

```
generatehdl(FIR,'ReuseAccum','on','Name','CascadeSerial', ...
    'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: CascadeSerial
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex94948885\hdlsrc\C
### Starting generation of CascadeSerial VHDL entity
### Starting generation of CascadeSerial VHDL architecture
```

```
### Clock rate is 4 times the input sample rate for this architecture.
### Serial partition # 1 has 3 inputs.
### Serial partition # 2 has 3 inputs.
### Successful completion of VHDL code generation process for filter: CascadeSerial
### HDL latency is 3 samples
```

Generate Serial Partitions of Cascaded Filter

Create a two-stage cascaded filter with these specifications for each filter stage:

- Direct form symmetric FIR filter
- Filter order 8
- Normalized cut-off frequency of 0.4 for the 6-dB point

Each call of the design function returns a dsp.FIRFilter System object™ that implements the specification. The cascade function returns a two-stage cascaded filter.

```
lp = design(fdesign.lowpass('N,Fc',8,.4),'FilterStructure','dfsymfir','SystemObject',true)
```

```
lp = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [-0.0061 -0.0136 0.0512 0.2657 0.4057 0.2657 0.0512 -0.0136 -0.0061]
    InitialConditions: 0

  Show all properties
```

```
hp = design(fdesign.highpass('N,Fc',8,.4),'FilterStructure','dfsymfir','SystemObject',true)
```

```
hp = 
  dsp.FIRFilter with properties:

            Structure: 'Direct form symmetric'
      NumeratorSource: 'Property'
            Numerator: [0.0060 0.0133 -0.0501 -0.2598 0.5951 -0.2598 -0.0501 0.0133 0.0060]
    InitialConditions: 0

  Show all properties
```

```
casc = cascade(lp,hp)
```

```
casc = 
  dsp.FilterCascade with properties:

         Stage1: [1x1 dsp.FIRFilter]
         Stage2: [1x1 dsp.FIRFilter]
    CloneStages: true
```

To generate HDL code, call the generatehdl function for the cascaded filter. When the filter is a System object, you must specify a fixed-point data type for the input data.

Specify different partitions for each cascade stage as a cell array.

```
generatehdl(casc,'InputDataType',numerictype(1,16,15),'SerialPartition',{[3 2],[4 1]})
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 4 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 2 samples
```

To explore the effective filter length and partitioning options for each filter stage of a cascade, call the `hdlfilterserialinfo` function. The function returns a partition vector corresponding to a desired number of multipliers. Request serial partition possibilities for the first stage, and choose a number of multipliers.

```
hdlfilterserialinfo(casc.Stage1,'InputDataType',numerictype(1,16,15))
```

```
   | Total Coefficients | Zeros | A/Symm | Effective |
   ---------------------------------------------------
   |         9          |   0   |    4   |     5     |

Effective filter length for SerialPartition value is 5.

  Table of 'SerialPartition' values with corresponding values of
  folding factor and number of multipliers for the given filter.

   | Folding Factor | Multipliers | SerialPartition |
   --------------------------------------------------
   |        1       |      5      |[1 1 1 1 1]      |
   |        2       |      3      |[2 2 1]          |
   |        3       |      2      |[3 2]            |
   |        4       |      2      |[4 1]            |
   |        5       |      1      |[5]              |
```

Select a serial partition vector for a target of two multipliers, and pass the vectors to the `generatehdl` function. Calling the function this way returns the first possible partition vector, but there are multiple partition vectors that achieve a two-multiplier architecture. Each stage uses a different clock rate based on the number of multipliers. The coder generates a timing controller to derive these clocks.

```
sp1 = hdlfilterserialinfo(casc.Stage1,'InputDataType',numerictype(1,16,15),'Multiplier',2)
```

```
sp1 = 1×2
```

```
     3     2
```

```
sp2 = hdlfilterserialinfo(casc.Stage2,'InputDataType',numerictype(1,16,15),'Multiplier',3)
```

```
sp2 = 1×3

     2     2     1
```

```
generatehdl(casc,'InputDataType',numerictype(1,16,15),'SerialPartition',{sp1,sp2})
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex16715237\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 2 samples
```

Generate Serial Architectures for IIR Filter

Create a direct form I SOS filter with these specifications:

- Sampling frequency of 48 kHz
- Filter order 5
- Cut-off frequency of 10.8 kHz for the 3 dB point

The `design` function returns a `dsp.BiquadFilter` System object™ that implements the specification. The custom accumulator data type avoids quantization error.

```
Fs = 48e3;
Fc = 10.8e3;
N = 5;
lp = design(fdesign.lowpass('n,f3db',N,Fc,Fs),'butter', ...
    'FilterStructure','df1sos','SystemObject',true,'UseLegacyBiquadFilter',true)
```

```
lp = 
  dsp.BiquadFilter with properties:

                     Structure: 'Direct form I'
               SOSMatrixSource: 'Property'
```

```
                   SOSMatrix: [3x6 double]
                 ScaleValues: [4x1 double]
    NumeratorInitialConditions: 0
  DenominatorInitialConditions: 0
      OptimizeUnityScaleValues: true

  Show all properties
```

```
nt_accum = numerictype('Signedness','auto','WordLength',20, ...
    'FractionLength',15);
nt_input = numerictype(1,16,15);
lp.NumeratorAccumulatorDataType = 'Custom';
lp.CustomNumeratorAccumulatorDataType = nt_accum;
lp.DenominatorAccumulatorDataType = 'Custom';
lp.CustomDenominatorAccumulatorDataType = nt_accum;
```

To list all possible serial architecture specifications for this filter, call the `hdlfilterserialinfo` function. When the filter is a System object, you must specify a fixed-point data type for the input data.

```
hdlfilterserialinfo(lp,'InputDataType',nt_input)
```

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

   | Folding factor | Multipliers |
   --------------------------------
   |       6        |      3      |
   |       9        |      2      |
   |      18        |      1      |
```

To generate HDL code, call the `generatehdl` function with one of the serial architectures. Specify either the `NumMultipliers` or `FoldingFactor` property, but not both. For instance, using the `NumMultipliers` property:

```
generatehdl(lp,'NumMultipliers',2,'InputDataType',nt_input)
```

```
### Starting VHDL code generation process for filter: lp
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex90334139\hdlsrc\
### Starting generation of lp VHDL entity
### Starting generation of lp VHDL architecture
### Successful completion of VHDL code generation process for filter: lp
### HDL latency is 2 samples
```

Alternatively, specify the same architecture with the `FoldingFactor` property.

```
generatehdl(lp,'FoldingFactor',9,'InputDataType',nt_input)
```

```
### Starting VHDL code generation process for filter: lp
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex90334139\hdlsrc\
### Starting generation of lp VHDL entity
### Starting generation of lp VHDL architecture
### Successful completion of VHDL code generation process for filter: lp
### HDL latency is 2 samples
```

Both these commands generate a filter that uses a total of two multipliers, with a latency of nine clock cycles. This architecture uses less area than the parallel implementation, at the expense of latency.

Distributed Arithmetic for Single Rate Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

Create a direct-form FIR filter and calculate the filter length, FL.

```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
firfilt = design(filtdes,'FilterStructure','dffir','SystemObject',true);
FL = length(find(firfilt.Numerator ~= 0))
```

```
FL = 31
```

Specify a set of partitions such that the partition sizes add up to the filter length. This is just one partition option, you can specify other combinations of sizes.

```
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'DALUTPartition',[8 8 8 7])
```

```
### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for r
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex00198568\hdlsrc\f
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 16 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

For comparison, create a direct-form symmetric FIR filter. The filter length is smaller in the symmetric case.

```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
firfilt = design(filtdes,'FilterStructure','dfsymfir','SystemObject',true);
FL = ceil(length(find(firfilt.Numerator ~= 0))/2)
```

```
FL = 16
```

Specify a set of partitions such that the partition sizes add up to the filter length. This is just one partition option, you can specify other combinations of sizes. **Tip:** Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible `DARadix` values for a filter.

```
generatehdl(firfilt,'InputDataType',numerictype(1,16,15), ...
    'DALUTPartition',[8 8])
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex00198568\hdlsrc\f
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 17 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

Distributed Arithmetic for Multirate Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

Create a direct-form FIR polyphase decimator, and calculate the filter length.

```
d = fdesign.decimator(4);
filt = design(d,'SystemObject',true);
FL = size(polyphase(filt),2)
```

```
FL = 27
```

Specify distributed arithmetic LUT partitions that add up to the filter size. When you specify partitions as a vector for a polyphase filter, each subfilter uses the same partitions.

```
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'DALUTPartition',[8 8 8 3])
```

```
### Starting VHDL code generation process for filter: firdecim
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex51670151\hdlsrc\
### Starting generation of firdecim VHDL entity
### Starting generation of firdecim VHDL architecture
### Clock rate is 4 times the input and 16 times the output sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firdecim
### HDL latency is 16 samples
```

You can also specify unique partitions for each subfilter. For the same filter, specify subfilter partitioning as a matrix. The length of the first subfilter is 1, and the other subfilters have length 26. **Tip:** Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible `DARadix` values for a filter.

```
d = fdesign.decimator(4);
filt = design(d,'SystemObject',true);
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'DALUTPartition',[1 0 0 0; 8 8 8 2; 8 8 6 4; 8 8 8 2])
```

```
### Starting VHDL code generation process for filter: firdecim
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex51670151\hdlsrc\
### Starting generation of firdecim VHDL entity
### Starting generation of firdecim VHDL architecture
### Clock rate is 4 times the input and 16 times the output sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firdecim
### HDL latency is 16 samples
```

Distributed Arithmetic for Cascaded Filters

Use distributed arithmetic options to reduce the number of multipliers in the filter implementation.

Create Cascaded Filter

Create a two-stage cascaded filter. Define different LUT partitions for each stage, and specify the partition vectors in a cell array.

```
lp = design(fdesign.lowpass('N,Fc',8,.4),'filterstructure','dfsymfir', ...
    'SystemObject',true);
hp = design(fdesign.highpass('N,Fc',8,.4),'filterstructure','dfsymfir', ...
```

```
    'SystemObject',true);
casc = cascade(lp,hp);
nt1 = numerictype(1,12,10);
generatehdl(casc,'InputDataType',nt1,'DALUTPartition',{[3 2],[2 2 1]})

### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 29 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples
```

Distributed Arithmetic Options

Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible `DARadix` values for each filter stage of a cascade. The function returns a LUT partition vector corresponding to a desired number of address bits.

Request LUT partition possibilities for the first stage.

```
 hdlfilterdainfo(casc.Stage1,'InputDataType',nt1);

   | Total Coefficients | Zeros | A/Symm | Effective |
   ------------------------------------------------
   |         9          |   0   |    4   |     5     |

Effective filter length for SerialPartition value is 5.

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

   | Folding Factor | LUT-Sets Multiple | DARadix |
   ------------------------------------------------
   |        1       |         12        |   2^12  |
   |        3       |         6         |   2^6   |
   |        4       |         4         |   2^4   |
   |        5       |         3         |   2^3   |
   |        7       |         2         |   2^2   |
   |       13       |         1         |   2^1   |

  Details of LUTs with corresponding 'DALUTPartition' values.

   | Max Address Width | Size(bits) |      LUT Details      | DALUTPartition |
   -----------------------------------------------------------------------
```

```
   |         5          |     416    |1x32x13               |[5]            |
   |         4          |     216    |1x16x12, 1x2x12       |[4 1]          |
   |         3          |     124    |1x4x13, 1x8x9         |[3 2]          |
   |         2          |     104    |1x2x12, 1x4x12, 1x4x8 |[2 2 1]        |

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

To request LUT partition possibilities for the second stage, you must first determine the input data type of the second stage.

```
y = casc.Stage1(fi(0,nt1));
nt2 = y.numerictype;
hdlfilterdainfo(casc.Stage2,'InputDataType',nt2);
```

```
   | Total Coefficients | Zeros | A/Symm | Effective |
   ---------------------------------------------------
   |          9         |   0   |    4   |     5     |

Effective filter length for SerialPartition value is 5.

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

   | Folding Factor | LUT-Sets Multiple | DARadix |
   ------------------------------------------------
   |        1       |         28        |   2^28  |
   |        3       |         14        |   2^14  |
   |        5       |         7         |   2^7   |
   |        8       |         4         |   2^4   |
   |       15       |         2         |   2^2   |
   |       29       |         1         |   2^1   |

  Details of LUTs with corresponding 'DALUTPartition' values.

   | Max Address Width | Size(bits) |       LUT Details      | DALUTPartition |
   ----------------------------------------------------------------------------
   |         5          |     896    |1x32x28                 |[5]             |
   |         4          |     488    |1x16x27, 1x2x28         |[4 1]           |
   |         3          |     304    |1x4x28, 1x8x24          |[3 2]           |
   |         2          |     256    |1x2x28, 1x4x23, 1x4x27  |[2 2 1]         |

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

Different LUT Partitions for Each Stage

Select address widths and folding factors to obtain LUT partition for each stage. The first stage uses LUTs with a maximum address size of five bits. The second stage uses LUTs with a maximum address size of three bits. They run at the same clock rate, and have different LUT partitions.

```
dp1 = hdlfilterdainfo(casc.Stage1,'InputDataType',nt1, ...
    'LUTInputs',5,'FoldingFactor',3);
dp2 = hdlfilterdainfo(casc.Stage2,'InputDataType',nt1, ...
    'LUTInputs',3,'FoldingFactor',5);
generatehdl(casc,'InputDataType',nt1,'DALUTPartition',{dp1,dp2});
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 13 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 29 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples
```

Different DARadix Values for Each Stage

You can also specify different `DARadix` values for each filter in a cascade. You can only specify different cascade partitions on the command-line. When you specify partitions in the **Generate HDL** dialog box, all cascade stages use the same partitions. Inspect the results of `hdlfilterdainfo` to set `DARadix` values for each stage.

```
generatehdl(casc,'InputDataType',nt1, ...
'DALUTPartition',{[3 2],[2 2 1]},'DARadix',{2^3,2^7})
```

```
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 5 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex17169310\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 4 samples
```

Cascaded Filter with Multiple Architectures

Specify different filter architectures for the different stages of a cascaded filter. You can specify a mix of serial, distributed arithmetic (DA), and parallel architectures depending upon your hardware constraints.

Create Cascaded Filter

Create a three-stage filter. Each stage is a different type.

```
h1 = dsp.FIRFilter('Numerator',[0.05 -.25 .88 0.9 .88 -.25 0.05]);
h2 = dsp.FIRFilter('Numerator',[-0.008 0.06 -0.44 0.44 -0.06 0.008], ...
    'Structure','Direct form antisymmetric');
h3 = dsp.FIRFilter('Numerator',[-0.008 0.06 0.44 0.44 0.06 -0.008], ...
    'Structure','Direct form symmetric');
casc = cascade(h1,h2,h3);
```

Specify Architecture for Each Stage

Specify a DA architecture for the first stage, a serial architecture for the second stage, and a fully parallel (default) architecture for the third stage.

To obtain `DARadix` values for the first architecture, use `hdlfilterdainfo`, then pick a value from `dr`.

```
nt = numerictype(1,12,10);
[dp,dr,lutsize,ff] = hdlfilterdainfo(casc.Stage1, ...
    'InputDataType',numerictype(1,12,10));
dr
```

```
dr = 6x1 cell
    {'2^12'}
    {'2^6' }
    {'2^4' }
    {'2^3' }
    {'2^2' }
    {'2^1' }
```

Set the property values as cell arrays, where each cell applies to a stage. To disable a property for a particular stage, use default values (-1 for the partitions and 2 for `DARadix`).

```
generatehdl(casc,'InputDataType',nt, ...
    'SerialPartition',{-1,3,-1}, ...
    'DALUTPartition',{[4 3],-1,-1}, ...
    'DARadix',{2^6,2,2});
```

```
### Structure fir has symmetric coefficients, consider converting to structure symmetricfir for r
### Starting VHDL code generation process for filter: casfilt
### Cascade stage # 1
### Starting VHDL code generation process for filter: casfilt_stage1
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\c
### Starting generation of casfilt_stage1 VHDL entity
### Starting generation of casfilt_stage1 VHDL architecture
### Clock rate is 2 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage1
### Cascade stage # 2
### Starting VHDL code generation process for filter: casfilt_stage2
```

```
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\c
### Starting generation of casfilt_stage2 VHDL entity
### Starting generation of casfilt_stage2 VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: casfilt_stage2
### Cascade stage # 3
### Starting VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\c
### Starting generation of casfilt_stage3 VHDL entity
### Starting generation of casfilt_stage3 VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt_stage3
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex13094988\hdlsrc\c
### Starting generation of casfilt VHDL entity
### Starting generation of casfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: casfilt
### HDL latency is 3 samples
```

Test Bench for FIR Filter with Programmable Coefficients

You can specify input coefficients to test a filter with programmable coefficients.

Create a direct-form symmetric FIR filter with a fully parallel (default) architecture. Define the coefficients for the filter object in the vector `b`. The coder generates test bench code to test the coefficient interface using a second set of coefficients, `c`. The coder trims `c` to the effective length of the filter.

```
b = [-0.01 0.1 0.8 0.1 -0.01];
c = [-0.03 0.5 0.7 0.5 -0.03];
c = c(1:ceil(length(c)/2));
filt = dsp.FIRFilter('Numerator',b,'Structure','Direct form symmetric');
generatehdl(filt,'InputDataType',numerictype(1,16,15), ...
    'GenerateHDLTestbench','on', ...
    'CoefficientSource','ProcessorInterface','TestbenchCoeffStimulus',c)
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex66247050\hdlsrc\f
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
### Starting generation of VHDL Test Bench.
### Generating input stimulus
### Done generating input stimulus; length 3107 samples.
### Generating Test bench: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex66247(
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

IIR Filter with Programmable Coefficients

Create a filter specification. When you generate HDL code, specify a programmable interface for the coefficients.

```
Fs = 48e3;
Fc = 10.8e3;
```

```
N = 5;
f_lp = fdesign.lowpass('n,f3db',N,Fc,Fs);
filtiir = design(f_lp,'butter','FilterStructure','df2sos','SystemObject',true,'UseLegacyBiquadFi
filtiir.OptimizeUnityScaleValues = 0;
generatehdl(filtiir,'InputDataType',numerictype(1,16,15), ...
    'CoefficientSource','ProcessorInterface')
```

```
### Starting VHDL code generation process for filter: filtiir
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex92389569\hdlsrc\
### Starting generation of filtiir VHDL entity
### Starting generation of filtiir VHDL architecture
### Second-order section, # 1
### Second-order section, # 2
### First-order section, # 3
### Successful completion of VHDL code generation process for filter: filtiir
### HDL latency is 2 samples
```

The coder generates this VHDL entity for the filter object.

```
ENTITY filtiir 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
         write_enable    :   IN    std_logic;
         write_done      :   IN    std_logic;
         write_address   :   IN    std_logic_vector(4 DOWNTO 0); -- ufix5
         coeffs_in       :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16
         filter_out      :   OUT   std_logic_vector(15 DOWNTO 0)  -- sfix16_En15
         );

END filtiir;
```

Clock Ports for Multirate Filters

Explore various ways to specify clock ports for multirate filters.

Default Setting

Create a polyphase sample rate converter. By default, the coder generates a single input clock (`clk`), an input clock enable (`clk_enable`), and a clock enable output signal named `ce_out`. The `ce_out` signal indicates when an output sample is ready. The `ce_in` output signal indicates when an input sample was accepted. You can use this signal to control the upstream data flow.

```
firrc = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3);
generatehdl(firrc,'InputDataType',numerictype(1,16,15))
```

```
### Starting VHDL code generation process for filter: firrc
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\
### Starting generation of firrc VHDL entity
### Starting generation of firrc VHDL architecture
### Successful completion of VHDL code generation process for filter: firrc
### HDL latency is 2 samples
```

The generated entity has the following signals:

```
ENTITY firrc 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(35 DOWNTO 0); -- sfix36_En31
         ce_in                        :   OUT   std_logic;
         ce_out                       :   OUT   std_logic
         );

END firrc;
```

Custom Clock Names

You can provide custom names for the input clock enable and the output clock enable signals. You cannot rename the `ce_in` signal.

```
firrc = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3)
```

```
firrc =
  dsp.FIRRateConverter with properties:

   Main
    InterpolationFactor: 5
       DecimationFactor: 3
        NumeratorSource: 'Property'
              Numerator: [0 -6.6976e-05 -1.6044e-04 -2.2552e-04 -1.8884e-04 0 3.2095e-04 6.5785e-

  Show all properties
```

```
generatehdl(firrc,'InputDataType',numerictype(1,16,15),...
            'ClockEnableInputPort','clk_en1', ...
            'ClockEnableOutputPort','clk_en2')
```

```
### Starting VHDL code generation process for filter: firrc
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\
### Starting generation of firrc VHDL entity
### Starting generation of firrc VHDL architecture
### Successful completion of VHDL code generation process for filter: firrc
### HDL latency is 2 samples
```

The generated entity has the following signals:

```
ENTITY firrc IS
   PORT( clk                              :   IN    std_logic;
         clk_en1                          :   IN    std_logic;
         reset                            :   IN    std_logic;
         filter_in                        :   IN    std_logic_vector(15 DOWNTO 0); -- sfix16_En15
         filter_out                       :   OUT   std_logic_vector(35 DOWNTO 0); -- sfix36_En31
         ce_in                            :   OUT   std_logic;
         clk_en2                          :   OUT   std_logic
         );

END firrc;
```

Multiple Clock Inputs

To generate multiple clock input signals for a supported multirate filter, set the `ClockInputs` property to `'Multiple'`. In this case, the coder does not generate any output clock enable ports.

```
decim = dsp.CICDecimator(7,1,4);
generatehdl(decim,'InputDataType',numerictype(1,16,15), ...
    'ClockInputs','Multiple')
```

```
### Starting VHDL code generation process for filter: cicDecOrIntFilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex09049114\hdlsrc\c
### Starting generation of cicDecOrIntFilt VHDL entity
### Starting generation of cicDecOrIntFilt VHDL architecture
### Section # 1 : Integrator
### Section # 2 : Integrator
### Section # 3 : Integrator
### Section # 4 : Integrator
### Section # 5 : Comb
### Section # 6 : Comb
### Section # 7 : Comb
### Section # 8 : Comb
### Successful completion of VHDL code generation process for filter: cicDecOrIntFilt
### HDL latency is 7 samples
```

The generated entity has the following signals:

```
ENTITY cicdecimfilt 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(27 DOWNTO 0)  -- sfix28_En15
         );

END cicdecimfilt;
```

Generate Default Altera Quartus II Synthesis Script

Create a filter object. Then call `generatehdl` , and specify a synthesis tool.

```
lpf = fdesign.lowpass('fp,fst,ap,ast',0.45,0.55,1,60);
firfilt = design(lpf,'equiripple','FilterStructure','dfsymfir', ...
    'SystemObject',true);
generatehdl(firfilt,'InputDataType',numerictype(1,14,13), ...
    'HDLSynthTool','Quartus');
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex92219095\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
```

The coder generates a script file named `firfilt_quartus.tcl`, using the default script properties for the Altera® Quartus II synthesis tool.

```
type hdlsrc/firfilt_quartus.tcl
```

```
load_package flow
set top_level firfilt
set src_dir "./hdlsrc"
set prj_dir "q2dir"
file mkdir ../$prj_dir
cd ../$prj_dir
project_new $top_level -revision $top_level -overwrite
set_global_assignment -name FAMILY "Stratix II"
set_global_assignment -name DEVICE EP2S60F484C3
set_global_assignment -name TOP_LEVEL_ENTITY $top_level
set_global_assignment -name vhdl_FILE "../$src_dir/firfilt.vhd"
execute_flow -compile
project_close
```

Construct Customized Synthesis Script

You can set the script automation properties to dummy values to illustrate how the coder constructs the synthesis script from the properties.

Design a filter and generate HDL. Specify a synthesis tool and custom text to include in the synthesis script.

```
lpf = fdesign.lowpass('fp,fst,ap,ast',0.45,0.55,1,60);
firfilt = design(lpf,'equiripple','FilterStructure','dfsymfir', ...
    'Systemobject',true);
generatehdl(firfilt,'InputDataType',numerictype(1,14,13), ...
    'HDLSynthTool','ISE', ...
    'HDLSynthInit','init line 1 : module name is %s\ninit line 2\n', ...
    'HDLSynthCmd','command : HDL filename is %s\n', ...
    'HDLSynthTerm','term line 1\nterm line 2\n');
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex64737676\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 2 samples
```

The coder generates a script file named `firfilt_ise.tcl`. Note the locations of the custom text you specified. You can use this feature to add synthesis instructions to the generated script.

```
type hdlsrc/firfilt_ise.tcl
```

```
init line 1 : module name is firfilt
init line 2
command : HDL filename is firfilt.vhd
term line 1
term line 2
```

Input Arguments

`filtSO` — Filter
filter System object

Filter from which to generate HDL code, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. You can use the following System objects from DSP System Toolbox:

Single Rate Filters

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`
- `dsp.FilterCascade`
- `dsp.VariableFractionalDelay`

Multirate Filters

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.FIRRateConverter`
- `dsp.FarrowRateConverter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FilterCascade`
- `dsp.DigitalDownConverter`
- `dsp.DigitalUpConverter`

`nt` — Input data type
`numerictype` object

Input data type, specified as a `numerictype` object. This argument applies only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`fd` — Fractional delay data type
`numerictype` object

Fractional delay data type, specified as a `numerictype` object. This argument applies only when the input filter is a `dsp.VariableFractionalDelay` System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`filterObj` — Filter
`dfilt` object

Filter from which to generate HDL code, specified as a `dfilt` object. You can create this object by using the `design` function. For an overview of supported filter features, see “Filter Configuration Options”.

Alternatives

You can use the `fdhdltool` function to generate HDL code instead (requires Filter Design HDL Coder). Specify the input and fractional delay data types as arguments, and then set additional properties in the Generate HDL dialog box.

Version History

Introduced before R2006a

See Also

`generatetbstimulus` | `fdhdltool`

generatetbstimulus

Generate HDL test bench stimulus

Syntax

```
dataIn = generatetbstimulus(filtSO,'InputDataType',nt)
dataIn = generatetbstimulus(filterObj)

dataIn = generatetbstimulus( ___ ,Name,Value)
```

Description

`dataIn = generatetbstimulus(filtSO,'InputDataType',nt)` generates a test bench stimulus for the specified filter System object and the input data type, specified by `nt`.

The coder chooses a default set of stimuli, depending on your filter type. The default set is `{'impulse','step','ramp','chirp','noise'}`. For IIR filters, `'impulse'` and `'step'` are excluded.

`dataIn = generatetbstimulus(filterObj)` generates a test bench stimulus for the specified `dfilt` filter object.

`dataIn = generatetbstimulus( ___ ,Name,Value)` uses optional name-value arguments, in addition to any of the input arguments in previous syntaxes. Use these options to change the default set of stimuli used by the coder.

Examples

Generate Test Bench Stimulus for FIR Filter

Design a lowpass filter and construct a direct-form FIR filter System object™, `fir_lp` .

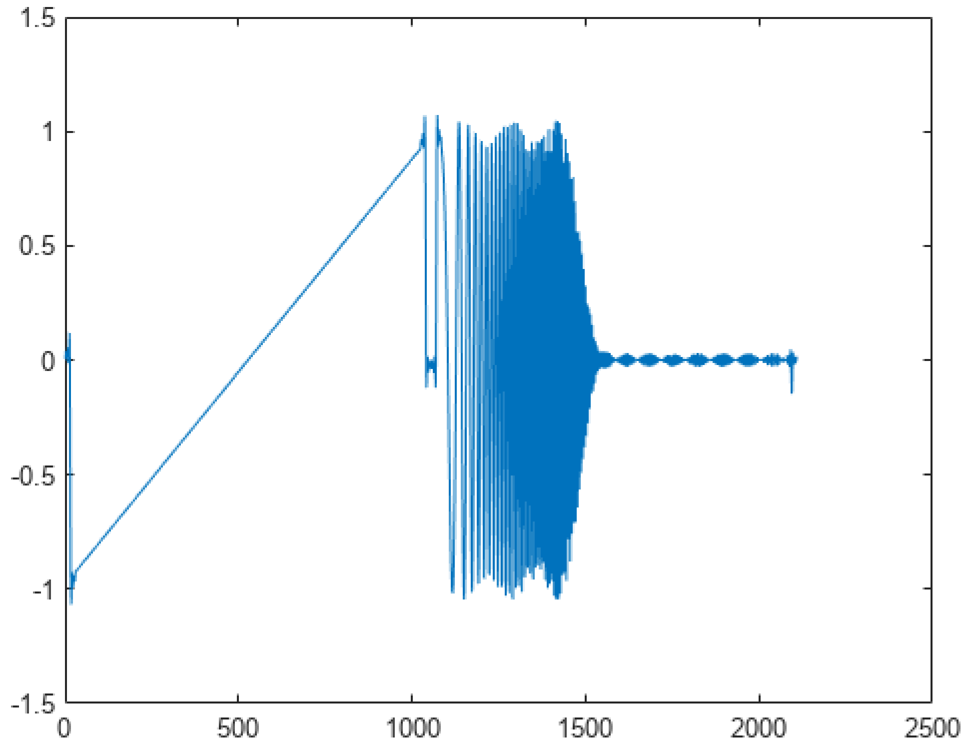
```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
fir_lp = design(filtdes,'FilterStructure','dffir','SystemObject',true);
```

Generate test bench input data. The call to `generatetbstimulus` generates ramp and chirp stimuli and returns the results. Specify the fixed-point input data type as a `numerictype` object.

```
rc_stim = generatetbstimulus(fir_lp,'InputDataType',numerictype(1,12,10),'TestBenchStimulus',{'ra
```

Apply the quantized filter to the data and plot the results. The call to the `step` function computes the filtered response to the input stimulus. The input data for the step function must be a column-vector to indicate samples over time. A row-vector would represent independent data channels.

```
plot(step(fir_lp,rc_stim'))
```



Input Arguments

`filtSO` — Filter
filter System object

Filter for which to generate a test bench stimulus, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. You can use the following System objects from DSP System Toolbox:

Single Rate Filters

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.HighpassFilter`
- `dsp.LowpassFilter`
- `dsp.FilterCascade`
- `dsp.VariableFractionalDelay`

Multirate Filters

- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`

- `dsp.FIRRateConverter`
- `dsp.FarrowRateConverter`
- `dsp.CICDecimator`
- `dsp.CICInterpolator`
- `dsp.CICCompensationDecimator`
- `dsp.CICCompensationInterpolator`
- `dsp.FilterCascade`
- `dsp.DigitalDownConverter`
- `dsp.DigitalUpConverter`

`nt` — Input data type
`numerictype` object

Input data type, specified as a `numerictype` object. This argument applies only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`filterObj` — Filter
`dfilt` object

Filter for which to generate a test bench stimulus, specified as a `dfilt` object. You can create this object by using the `design` function. For an overview of supported filter features, see "Filter Configuration Options".

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TestBenchStimulus',{'ramp','impulse'}`

`TestBenchStimulus` — Input stimuli
`'impulse'` | `'step'` | `'ramp'` | `'chirp'` | `'noise'` | cell array of character vectors | string array

Input stimuli that the generated test bench applies to the filter, specified as `'impulse'`, `'step'`, `'ramp'`, `'chirp'`, or `'noise'`. You can specify combinations of these stimuli in a cell array of character vectors or string array, in any order.

You can also specify a custom input vector by using the `TestBenchUserStimulus` property. When `TestBenchUserStimulus` is a non-empty vector, it takes priority over `TestBenchStimulus`.

Example: `'TestBenchStimulus',{'ramp','impulse','noise'}`

`TestBenchUserStimulus` — Custom vector of input data
`[]` (default) | function call

Custom vector of input data that the generated test bench applies to the filter, specified as the empty vector or a function call that returns a vector. When this argument is set to the empty vector, the test bench uses the `TestBenchStimulus` property to generate input data.

For example, this 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)
```

Specify this stimulus when you call `generatetbstimulus`.

```
generatetbstimulus(filt,'InputDataType',numerictype(1,16,15), ...
    'TestBenchUserStimulus',repmat([1 1 1 1 0 0 0 0],1,10))
```

Output Arguments

dataIn — Test bench stimulus
single array | double array | fi array

Test bench stimulus for the filter, returned as a `single`, `double`, or `fi` array. If the input filter is a `dfilt` filter object, the results are quantized using the arithmetic property of the filter object. If the input filter is a filter System object, the stimulus is quantized by `nt`.

Version History

Introduced before R2006a

See Also

`generatehdl` | `fdhdltool`

hdlfilterdainfo

Distributed arithmetic information for filter architectures

Syntax

```
hdlfilterdainfo(filtSO,'InputDataType',nt)
hdlfilterdainfo(filtObj)
hdlfilterdainfo( ___ ,Name,Value)

[dp,dr,lutsize,ff] = hdlfilterdainfo( ___ )
```

Description

`hdlfilterdainfo(filtSO,'InputDataType',nt)` displays distributed arithmetic (DA) information for the specified filter System object and the input data type, specified by `nt`. The information consists of an exhaustive table of `DARadix` values with corresponding folding factors and multiplies for LUT sets, and a table with details of LUTs with corresponding `DALUTPartition` values. This information helps you to define optimal DA settings for the filter.

`hdlfilterdainfo(filtObj)` displays DA information for the specified `dfilt` filter object.

`hdlfilterdainfo( ___ ,Name,Value)` uses optional name-value arguments, in addition to any of the input arguments in previous syntaxes. Use these options to query for DA LUT partition and DA radix information calculated for a given folding factor or LUT specification.

`[dp,dr,lutsize,ff] = hdlfilterdainfo( ___ )` stores filter architecture details in output variables.

Examples

Explore DA Options for a Filter

Construct a direct-form FIR filter, and pass it to `hdlfilterdainfo`. The command displays the results at the command line.

```
firfilt = design(fdesign.lowpass('N,Fc',8,.4),'SystemObject',true);
hdlfilterdainfo(firfilt,'InputDataType',numerictype(1,12,10))
```

```
   | Total Coefficients | Zeros | Effective |
   ------------------------------------------
   |         9          |   0   |     9     |

Effective filter length for SerialPartition value is 9.

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

   | Folding Factor | LUT-Sets Multiple | DARadix |
   ------------------------------------------------
   |       1        |        12         |   2^12  |
```

```
|       2       |         6         |   2^6   |
|       3       |         4         |   2^4   |
|       4       |         3         |   2^3   |
|       6       |         2         |   2^2   |
|      12       |         1         |   2^1   |

  Details of LUTs with corresponding 'DALUTPartition' values.

   | Max Address Width | Size(bits) |             LUT Details              | DALUTPartition |
   ----------------------------------------------------------------------------------------
   |         9         |    7168    |1x512x14                              |[9]             |
   |         8         |    3596    |1x256x14, 1x2x6                       |[8 1]           |
   |         7         |    1824    |1x128x14, 1x4x8                       |[7 2]           |
   |         6         |     904    |1x64x13, 1x8x9                        |[6 3]           |
   |         5         |     608    |1x16x12, 1x32x13                      |[5 4]           |
   |         4         |     412    |1x16x12, 1x16x13, 1x2x6               |[4 4 1]         |
   |         3         |     248    |1x8x13, 2x8x9                         |[3 3 3]         |
   |         2         |     180    |1x2x6, 1x4x12, 1x4x13, 1x4x8, 1x4x9   |[2 2 2 2 1]     |

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

Generate Code for Filter with Distributed Arithmetic Settings

Create a direct-form FIR filter.

```
firfilt = design(fdesign.lowpass('N,Fc',8,.4),'filterstructure','dfsymfir','SystemObject',true);
```

Call `hdlfilterdainfo`.

```
lutip = 4;
ff = 3;
[dp,dr,lutsize,ff] = hdlfilterdainfo(firfilt, ...
    'InputDataType',numerictype(1,12,10), ...
    'FoldingFactor',ff,'LUTInputs', lutip);
```

Pass the returned DA LUT partition (`dp`) and DA radix (`dr`) values into `generatehdl`. The generated HDL code has DA architecture and implements LUTs with the specified max address width (`lutip`) and folding factor (`ff`).

```
generatehdl(firfilt,'InputDataType',numerictype(1,12,10), ...
    'DALUTPartition',dp,'DARadix',dr);
```

```
### Starting VHDL code generation process for filter: firfilt
### Generating: C:\TEMP\Bdoc23a_2213998_3568\ib570499\33\tp913ae594\hdlfilter-ex76912192\hdlsrc\
### Starting generation of firfilt VHDL entity
### Starting generation of firfilt VHDL architecture
### Clock rate is 3 times the input sample rate for this architecture.
### Successful completion of VHDL code generation process for filter: firfilt
### HDL latency is 3 samples
```

Input Arguments

`filtSO` — Filter
filter System object

Filter for which to display distributed arithmetic information, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. The following System objects from DSP System Toolbox support distributed arithmetic:

- `dsp.FIRFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`

For more information, see "Distributed Arithmetic for FIR Filters" on page 4-16.

`nt` — Input data type
`numerictype` object

Input data type, specified as a `numerictype` object. This argument applies only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`filtObj` — Filter
`dfilt` object

Filter for which to display distributed arithmetic information, specified as a `dfilt` object. See "Distributed Arithmetic for FIR Filters" on page 4-16 for filter types that support distributed arithmetic. You can create this object using the `design` function.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FoldingFactor',2,'DALUTPartition',9`.

You can only specify one folding factor argument and one LUT argument at a time.

Folding Factor Arguments

`FoldingFactor` — Hardware folding factor
integer greater than `1` | `Inf`

Hardware folding factor, specified as `Inf` or an integer greater than `1`. Given the folding factor, the coder displays an exhaustive table of corresponding LUT input values, sizes, and details. If the folding factor is `inf`, the coder uses the maximum folding factor.

Example: `'FoldingFactor',2`

`DARadix` — DA radix value
integer power of `2`

DA radix value, specified as an integer power of 2. Given the DA radix, the coder displays for the corresponding folding factor value an exhaustive table of LUT input values, sizes, and details.

Example: `'DARadix',4`

LUT Arguments

`LUTInputs` — LUT input value

integer greater than 1

LUT input value, specified as an integer greater than 1. Given the LUT input value, the coder displays an exhaustive table of the corresponding folding factor values, LUT sizes, and details.

Example: `'LUTInputs',3`

`DALUTPartition` — DA LUT partition value

integer greater than 1

DA LUT partition value, specified as an integer greater than 1. Given the DA LUT partition value, the coder displays an exhaustive table of the corresponding folding factor values, LUT sizes, and details.

Example: `'DALUTPartition',9`

Output Arguments

dp — DA LUT partition

cell array

DA LUT partition values, returned as a cell array.

`dr` — DA radix

cell array

DA radix values, returned as a cell array.

`lutsize` — LUT size

cell array

LUT size values, returned as a cell array.

`ff` — Folding factor

cell array

Folding factor values, returned as a cell array.

Version History

Introduced in R2011a

See Also

`hdlfilterserialinfo`

Topics

"Distributed Arithmetic for FIR Filters" on page 4-16

hdlfilterserialinfo

Serial partition information for filter architectures

Syntax

```
hdlfilterserialinfo(filtSO,'InputDataType',nt)
hdlfilterserialinfo(filtSO,'InputDataType',nt,'FoldingFactor',ff)
hdlfilterserialinfo(filtSO,'InputDataType',nt,'Multipliers',mult)
hdlfilterserialinfo(filtSO,'InputDataType',nt,'SerialPartition',[p1 ... pN])

hdlfilterserialinfo(filtObj)
hdlfilterserialinfo(filtObj,'FoldingFactor',ff)
hdlfilterserialinfo(filtObj,'Multipliers',mult)
hdlfilterserialinfo(filtObj,'SerialPartition',[p1 ... pN])

[sp,fold,nm] = hdlfilterserialinfo(___)
```

Description

`hdlfilterserialinfo(filtSO,'InputDataType',nt)` displays an exhaustive table of serial partition values with corresponding folding factors and numbers of multipliers for the specified filter System object and the input data type, specified by `nt`. This information helps you to define optimal serial architecture for the filter in the generated HDL code.

`hdlfilterserialinfo(filtSO,'InputDataType',nt,'FoldingFactor',ff)` displays only those serial partition values that correspond to the specified folding factor.

`hdlfilterserialinfo(filtSO,'InputDataType',nt,'Multipliers',mult)` displays only those serial partition values that correspond to the specified number of multipliers.

`hdlfilterserialinfo(filtSO,'InputDataType',nt,'SerialPartition',[p1 ... pN])` displays the folding factor and number of multipliers corresponding to the serial partition vector.

`hdlfilterserialinfo(filtObj)` displays serial partition information for the specified `dfilt` filter object.

`hdlfilterserialinfo(filtObj,'FoldingFactor',ff)` displays only those serial partition values that correspond to the specified folding factor.

`hdlfilterserialinfo(filtObj,'Multipliers',mult)` displays only those serial partition values that correspond to the specified number of multipliers.

`hdlfilterserialinfo(filtObj,'SerialPartition',[p1 ... pN])` displays the folding factor and number of multipliers corresponding to the serial partition vector.

`[sp,fold,nm] = hdlfilterserialinfo(___)` captures serial partition values with their corresponding folding factors and numbers of multipliers, for any of the input argument combinations in previous syntaxes.

Examples

Explore Serial Partition Options

To display valid serial partitions, pass the filter, with no other arguments, to `hdlfilterserialinfo`.

```
filt = design(fdesign.lowpass('N,Fc',8,.4),'SystemObject',true);
hdlfilterserialinfo(filt,'InputDataType',numerictype(1,12,10))
```

```
   | Total Coefficients | Zeros | Effective |
   ------------------------------------------
   |         9          |   0   |     9     |

Effective filter length for SerialPartition value is 9.

  Table of 'SerialPartition' values with corresponding values of
  folding factor and number of multipliers for the given filter.

   | Folding Factor | Multipliers |   SerialPartition   |
   ------------------------------------------------------
   |        1       |      9      |[1 1 1 1 1 1 1 1 1]  |
   |        2       |      5      |[2 2 2 2 1]          |
   |        3       |      3      |[3 3 3]              |
   |        4       |      3      |[4 4 1]              |
   |        5       |      2      |[5 4]                |
   |        6       |      2      |[6 3]                |
   |        7       |      2      |[7 2]                |
   |        8       |      2      |[8 1]                |
   |        9       |      1      |[9]                  |
```

Explore Serial Partitions for a Fixed Number of Multipliers

Design a filter and pass it to `hdlfilterserialinfo`. Request serial partition parameters for a design that uses three multipliers.

```
filt = design(fdesign.lowpass('N,Fc',8,.4),'SystemObject',true);
hdlfilterserialinfo(filt,'InputDataType',numerictype(1,12,10),'Multipliers',3)
```

```
Serial Partition: [3 3 3], Folding Factor:    3, Multipliers:    3
```

Explore Serial Partitions for a Fixed Folding Factor

Design a filter and pass it to `hdlfilterserialinfo`. Request serial partition parameters for a design that uses a folding factor of four.

```
filt = design(fdesign.lowpass('N,Fc',8,.4),'SystemObject',true);
hdlfilterserialinfo(filt,'InputDataType',numerictype(1,12,10),'FoldingFactor',4)
```

```
Serial Partition: [4 4 1], Folding Factor:    4, Multipliers:    3
```

Return Serial Partition Options to a Cell Array

Pass the filter and data type, with no additional arguments, to `hdlfilterserialinfo`. You can return the results to a cell array.

```
filt = design(fdesign.lowpass('N,Fc',8,.4),'SystemObject',true);
[sp,ff,nm] = hdlfilterserialinfo(filt,'InputDataType',numerictype(1,12,10))
```

```
sp = 9x1 cell
    {'[1 1 1 1 1 1 1 1 1]'}
    {'[2 2 2 2 1]'        }
    {'[3 3 3]'            }
    {'[4 4 1]'            }
    {'[5 4]'              }
    {'[6 3]'              }
    {'[7 2]'              }
    {'[8 1]'              }
    {'[9]'                }


ff = 9x1 cell
    {'1'}
    {'2'}
    {'3'}
    {'4'}
    {'5'}
    {'6'}
    {'7'}
    {'8'}
    {'9'}


nm = 5x1 cell
    {'1'}
    {'2'}
    {'3'}
    {'5'}
    {'9'}
```

You can also use this syntax while specifying a number of multipliers or folding factor.

```
[sp_ff4,ff4,nm_ff4] = hdlfilterserialinfo(filt,'InputDataType',numerictype(1,12,10), ...
    'FoldingFactor',4)
```

```
sp_ff4 = 1×3

     4     4     1


ff4 = 4

nm_ff4 = 3
```

Input Arguments

`filtSO` — Filter
filter System object

Filter for which to display serial partition information, specified as a filter System object. To create a filter System object, use the `design` function or see the reference page of the object. The following System objects from DSP System Toolbox support serial architectures:

- `dsp.FIRFilter`
- `dsp.FIRDecimator`
- `dsp.FIRInterpolator`
- `dsp.BiquadFilter`

For more information, see “Speed vs. Area Tradeoffs” on page 4-2.

`nt` — Input data type
`numerictype` object

Input data type, specified as a `numerictype` object. This argument applies only when the input filter is a System object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

`filtObj` — Filter
`dfilt` object

Filter for which to display serial partition information, specified as a `dfilt` object. See “Speed vs. Area Tradeoffs” on page 4-2 for filter types that support serial architectures. You can create this object using the `design` function.

`ff` — Hardware folding factor
integer greater than `1` | `Inf`

Hardware folding factor, specified as an integer greater than `1` or `Inf`. Given the folding factor, the coder computes the serial partition and the number of multipliers. If the folding factor is `Inf`, the coder uses the maximum folding factor.

`mult` — Desired number of multipliers
integer greater than `1` | `Inf`

Desired number of multipliers, specified as an integer greater than `1` or `Inf`. Given the number of multipliers, the coder computes the serial partition and the folding factor. If the number of multipliers is `inf`, the coder uses the maximum number of multipliers.

`[p1 ... pN]` — Serial partitions
vector of `N` integers

Serial partitions, specified as a vector of *N* integers, where *N* is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition.

Output Arguments

`sp` — Serial partition
cell array of vectors

Available serial partitioning options, returned as a cell array of vectors.

`fold` — Folding factor
cell array

Available folding factor values, returned as a cell array.

`nm` — Number of multipliers
cell array

Available multiplier values, returned as a cell array.

Version History

Introduced in R2010b

See Also

`hdlfilterdainfo`

Topics

"Speed vs. Area Tradeoffs" on page 4-2