# Filter Design HDL Coder™

## User's Guide

MATLAB®

R2016a

MathWorks®

## How to Contact MathWorks

| | |
|---|---|
| Latest news: | www.mathworks.com |
| Sales and services: | www.mathworks.com/sales_and_services |
| User community: | www.mathworks.com/matlabcentral |
| Technical support: | www.mathworks.com/support/contact_us |
| ☎ Phone: | 508-647-7000 |

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

*Filter Design HDL Coder™ User's Guide*

### Trademarks

### Patents

## 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) |

# 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

# 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, an architect or designer uses DSP System Toolbox™ tools (FDATool or `filterbuilder`) to design a filter algorithm targeted for hardware. Then, a designer uses the Filter Design HDL Coder app (`fdhdltool`) or command-line interface (`generatehdl`) to configure code generation options and generate 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`

## Related Examples

- “Starting Filter Design HDL Coder” on page 2-2
- “Generating HDL Code” on page 2-14

# Basic FIR Filter

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

**In this section...**

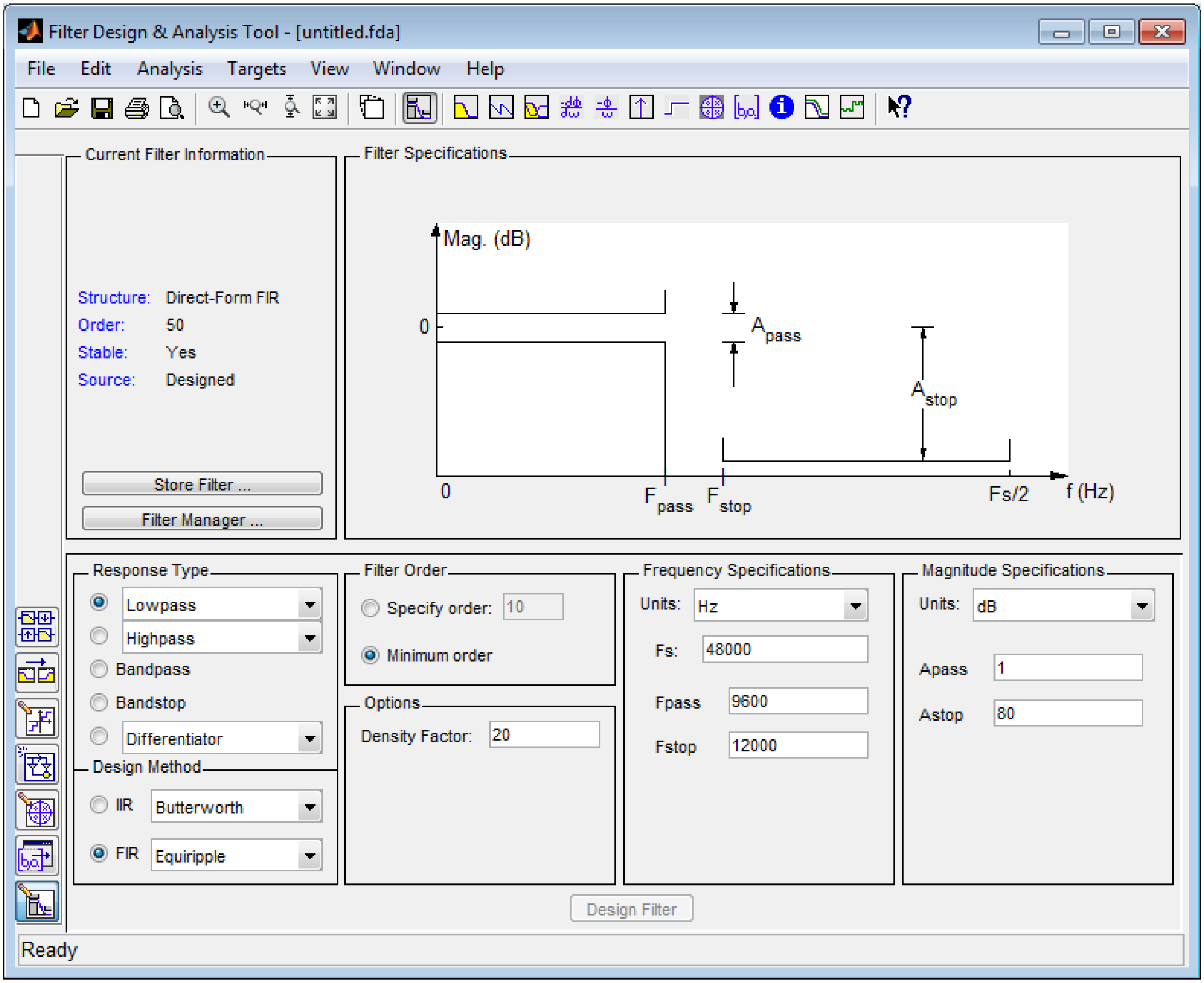
## Create a Folder for Your Tutorial Files

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 FDATool

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

1. Start the MATLAB software.
2. Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-4.
3. Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



Current Filter Information

Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Filter Specifications

Response Type

● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method

○ IIR Butterworth
● FIR Equiripple

Filter Order

○ Specify order: 10
● Minimum order

Options

Density Factor: 20

Frequency Specifications

Units: Hz
Fs: 48000
Fpass: 9600
Fstop: 12000

Magnitude Specifications

Units: dB
Apass 1
Astop 80

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

| Option | Value |
|---|---|
| **Response Type** | `Lowpass` |
| **Design Method** | `FIR Equiripple` |

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

These settings are for the default filter design that the FDATool 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 1-6.

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

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

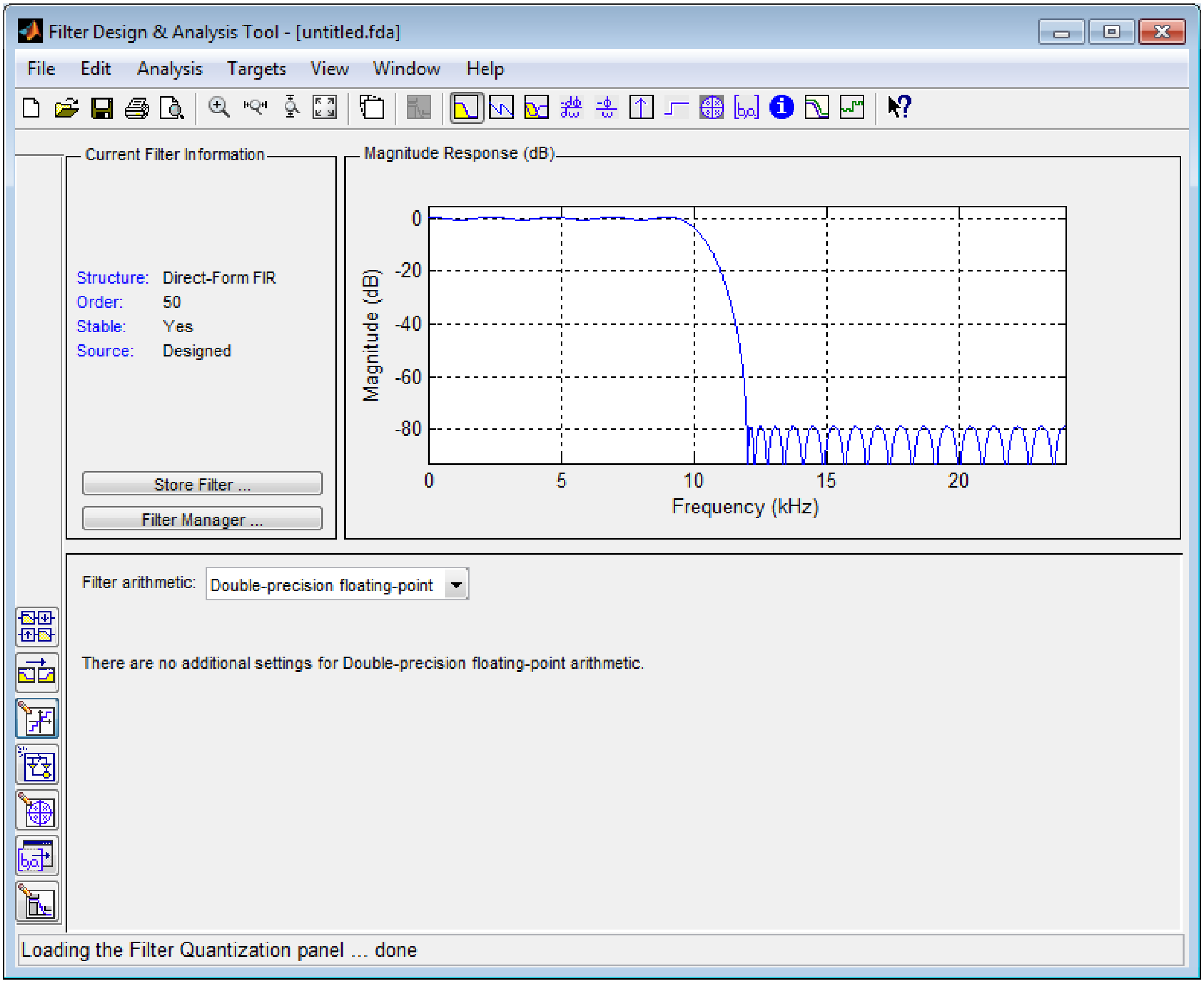
For more information on designing filters with the FDATool, see the DSP System Toolbox documentation.

## 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 FDATool" on page 1-4.

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



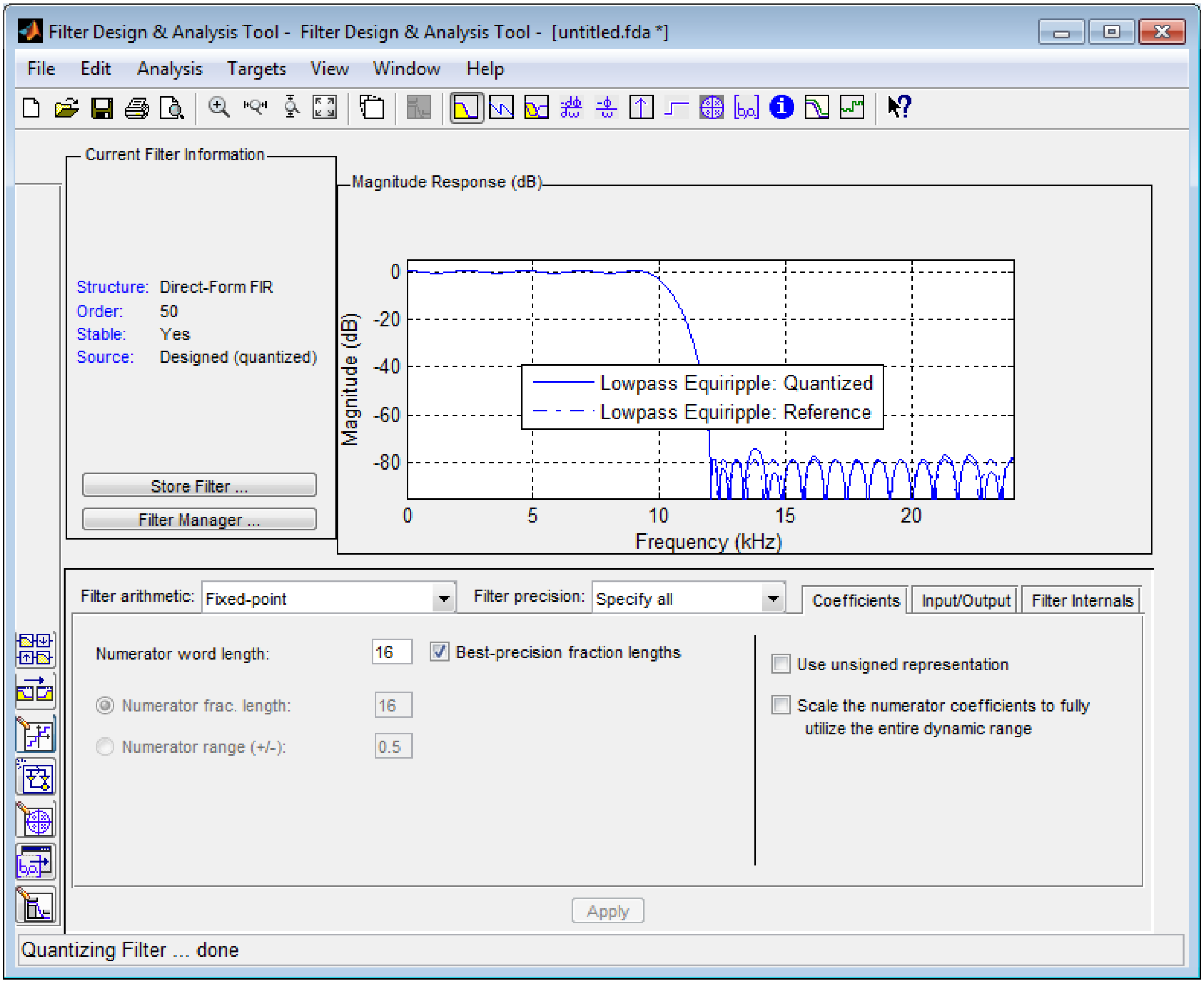
Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Filter arithmetic: Double-precision floating-point

There are no additional settings for Double-precision floating-point arithmetic.

Loading the Filter Quantization panel ... done

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



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed (quantized)

Filter arithmetic: Fixed-point

Filter precision: Specify all

Numerator word length: 16

☑ Best-precision fraction lengths

● Numerator frac. length: 16

○ Numerator range (+/-): 0.5

□ Use unsigned representation

□ Scale the numerator coefficients to fully utilize the entire dynamic range

Quantizing Filter ... done

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` |
| | **Accum. word length** | `40` |

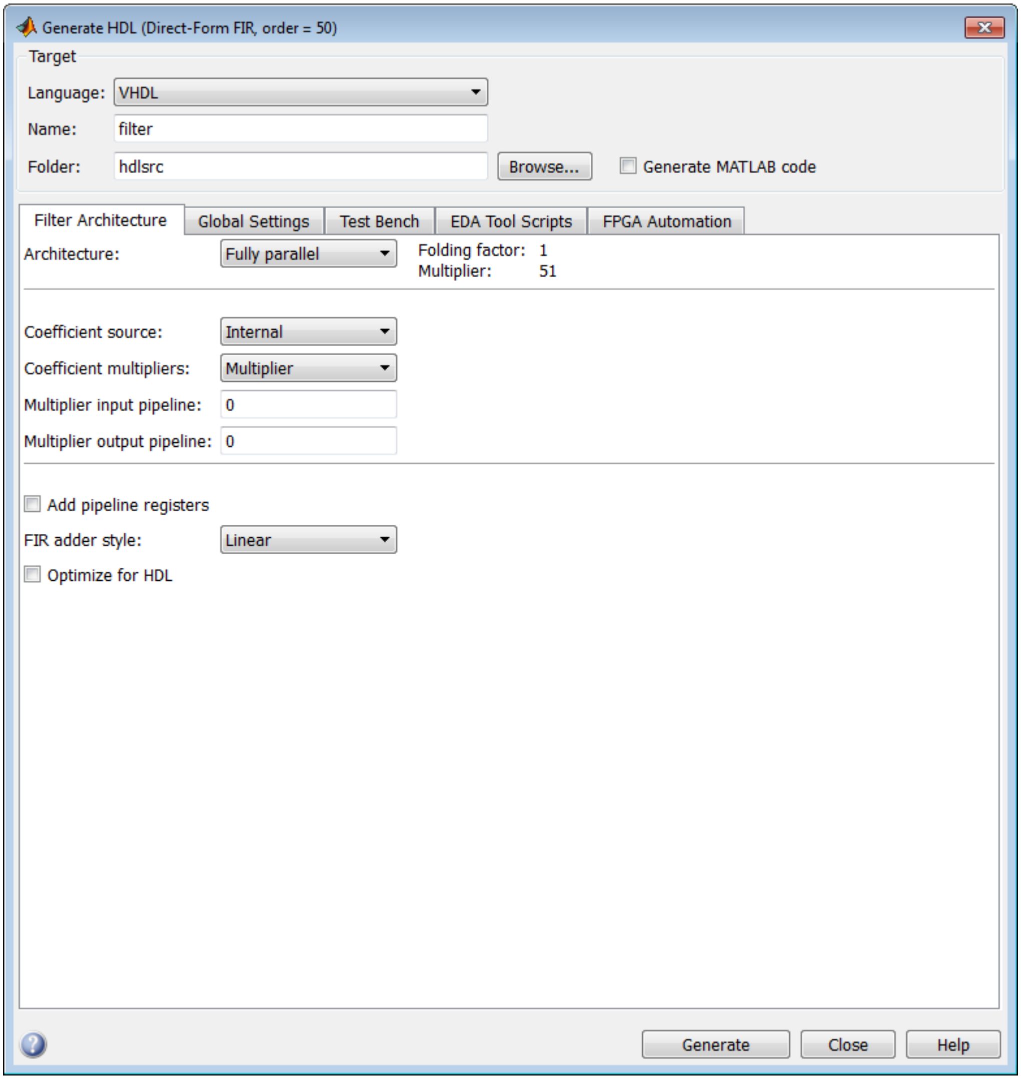
4 Click **Apply**.

For more information on quantizing filters with the FDATool, 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 GUI, 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 FDATool” on page 1-4 and “Quantize the Filter” on page 1-6.

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



Generate HDL (Direct-Form FIR, order = 50)

Target

Language: VHDL

Name: filter

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 51

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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**.

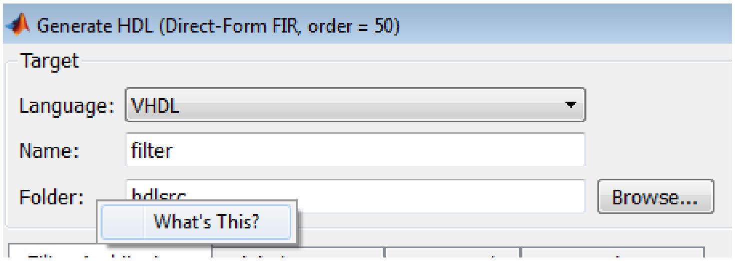
   **b** In the **Contents** pane of the **Help** browser, select the **Filter Design HDL Coder** entry.

   **c** Minimize the **Help** browser.

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

**4** Close the **Help** window.

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



Generate HDL (Direct-Form FIR, order = 50)

Target

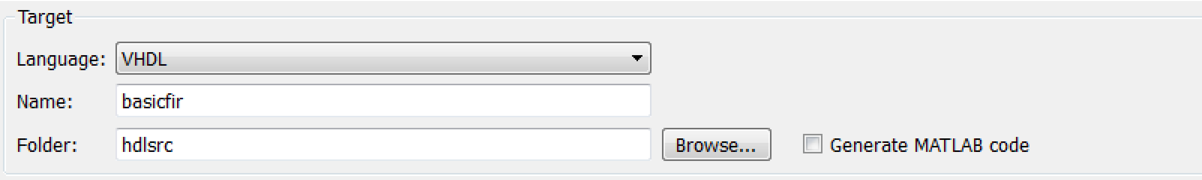
Language: VHDL

Name: filter

Folder: hdlsrc

**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.



Target

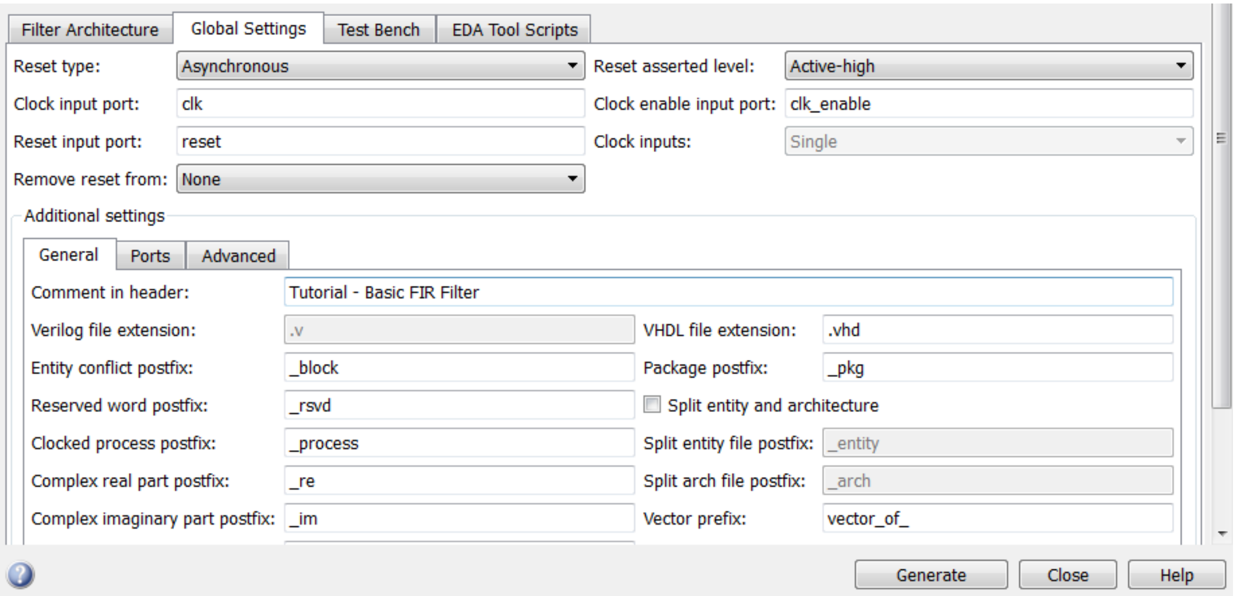
Language: VHDL

Name: basicfir

Folder: hdlsrc

□ Generate MATLAB code

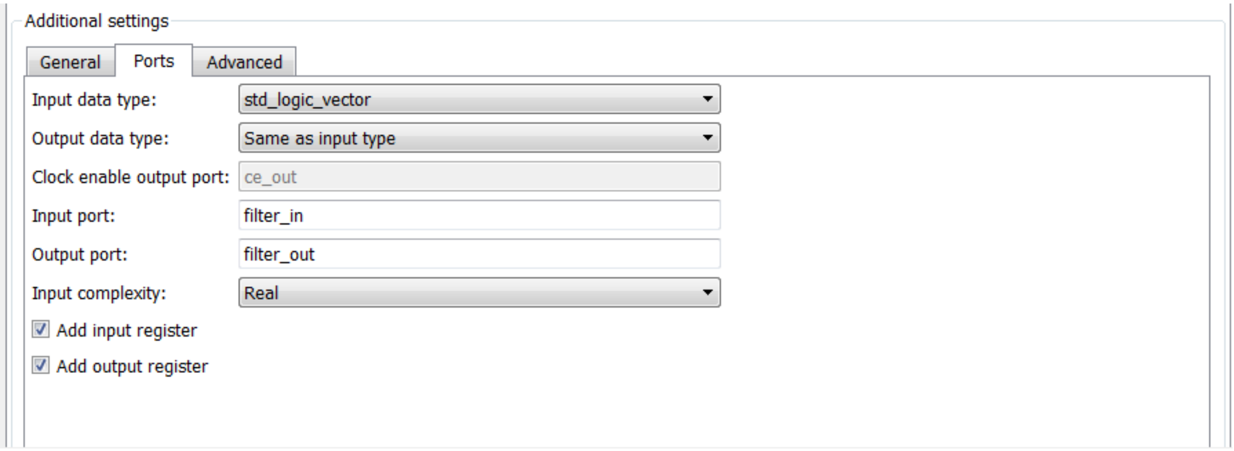
**8** Select the **Global settings** tab of the GUI. Then select the **General** tab of the **Additional settings** section of the GUI. Type `Tutorial - Basic FIR Filter` in the **Comment in header** text box. The coder adds the comment to the end of the header comment block in each generated file.



Reset type: Asynchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Remove reset from: None

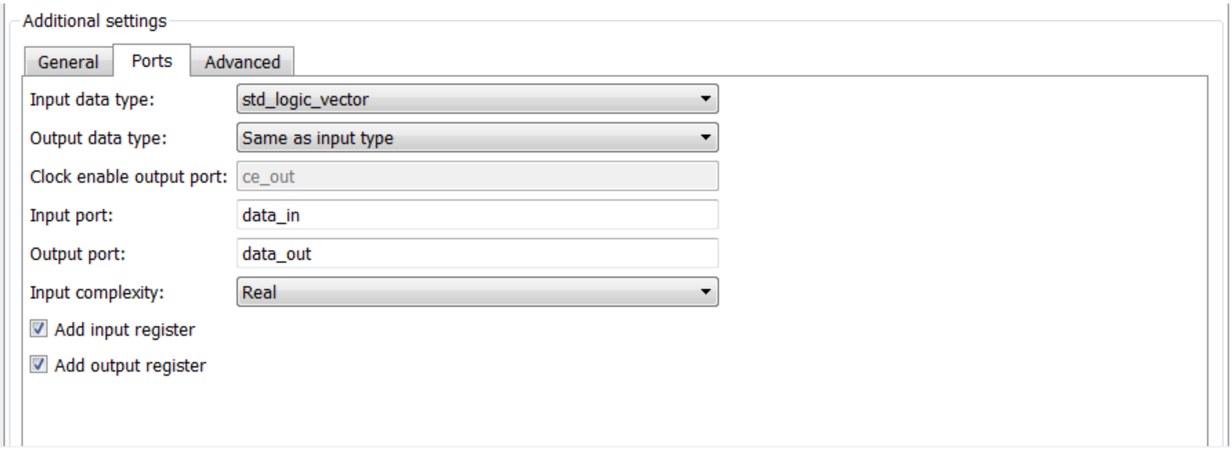
Comment in header: Tutorial - Basic FIR Filter
Verilog file extension: .v
VHDL file extension: .vhd
Entity conflict postfix: _block
Package postfix: _pkg
Reserved word postfix: _rsvd
□ Split entity and architecture
Clocked process postfix: _process
Split entity file postfix: _entity
Complex real part postfix: _re
Split arch file postfix: _arch
Complex imaginary part postfix: _im
Vector prefix: vector_of_

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



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

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`.



Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: data_in

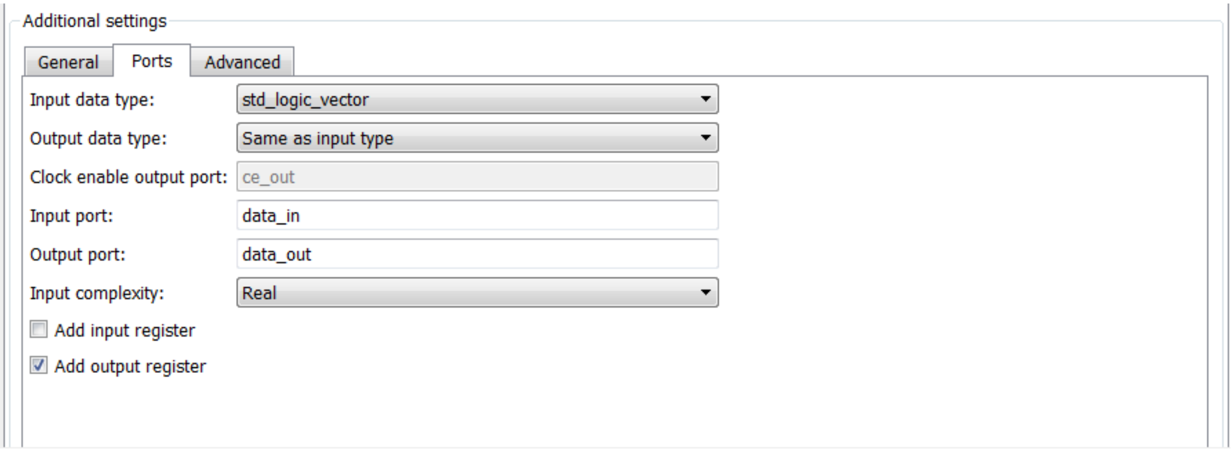
Output port: data_out

Input complexity: Real

☑ Add input register

☑ Add output register

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



Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: data_in

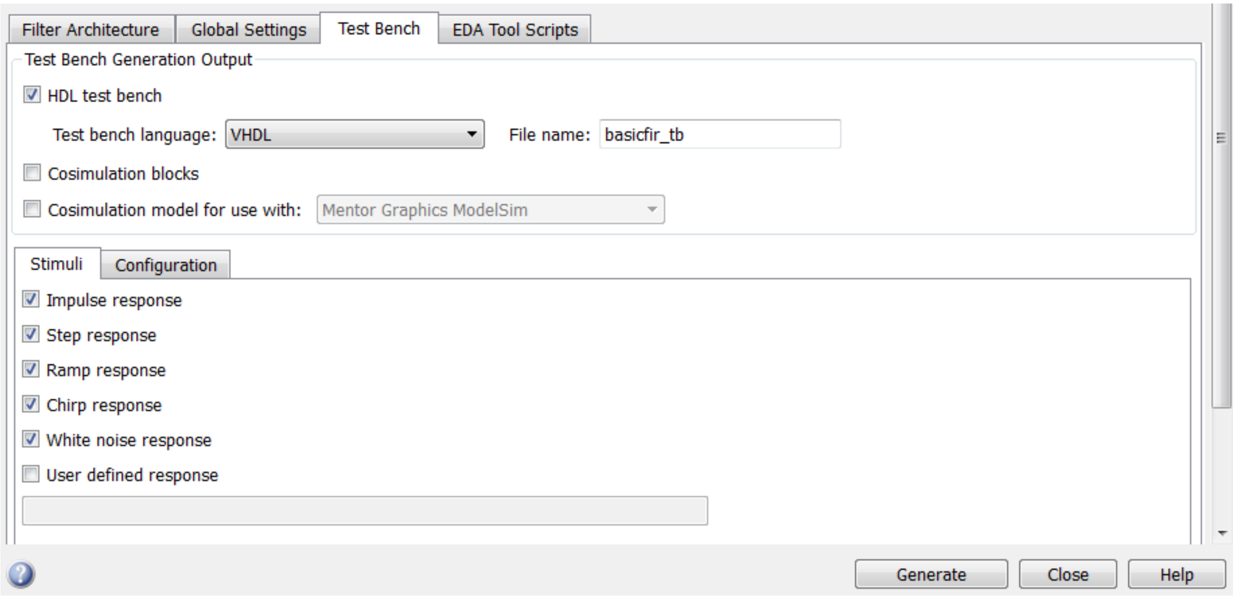
Output port: data_out

Input complexity: Real

□ Add input register

☑ Add output register

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



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL    File name: basicfir_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

**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 the following characteristics:

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

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

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

The generated test bench:

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

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

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

## 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 string you specified for the **Name** option in the **Target** pane. See step 5 in “Configure and Generate VHDL Code” on page 1-9.

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 1-9.

4 Search for `HDL Code`. This section lists coder options you modified in “Configure and Generate VHDL Code” on page 1-9.

5 Search for `Filter Settings`. This section describes the filter design and quantization settings as you specified in “Design a FIR Filter in FDATool” on page 1-4 and “Quantize the Filter” on page 1-6.

6 Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **Target** pane. See step 5 in “Configure and Generate VHDL Code” on page 1-9.

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 strings. 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 dialog box. See step 12 in “Configure and Generate VHDL Code” on page 1-9.

8 Search for `Constants`. This section 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 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 string `_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 1-9, you cleared the **Add input register** option, but left the **Add output register** selected. Also note that the `PROCESS` block name `Output_Register_process` includes the default `PROCESS` block postfix string `_process`.

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

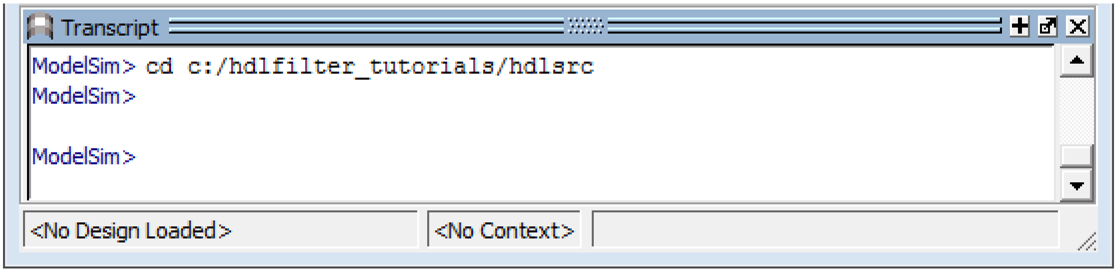
## Verify the Generated VHDL Code

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 the following steps:

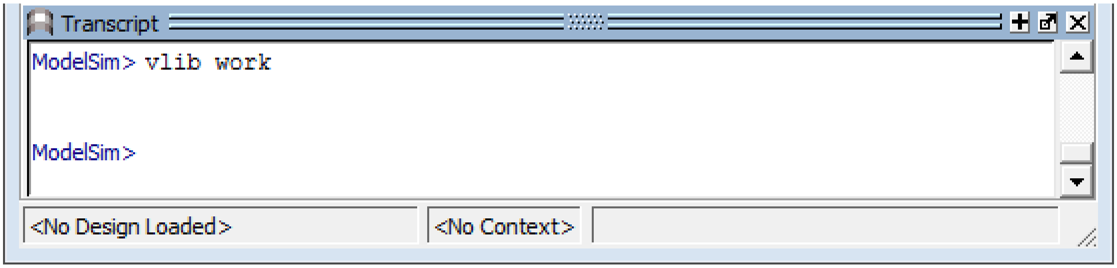
1 Start your Mentor Graphics ModelSim simulator.

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



```
ModelSim> cd c:/hdlfilter_tutorials/hdlsrc
ModelSim>

ModelSim>
```

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.

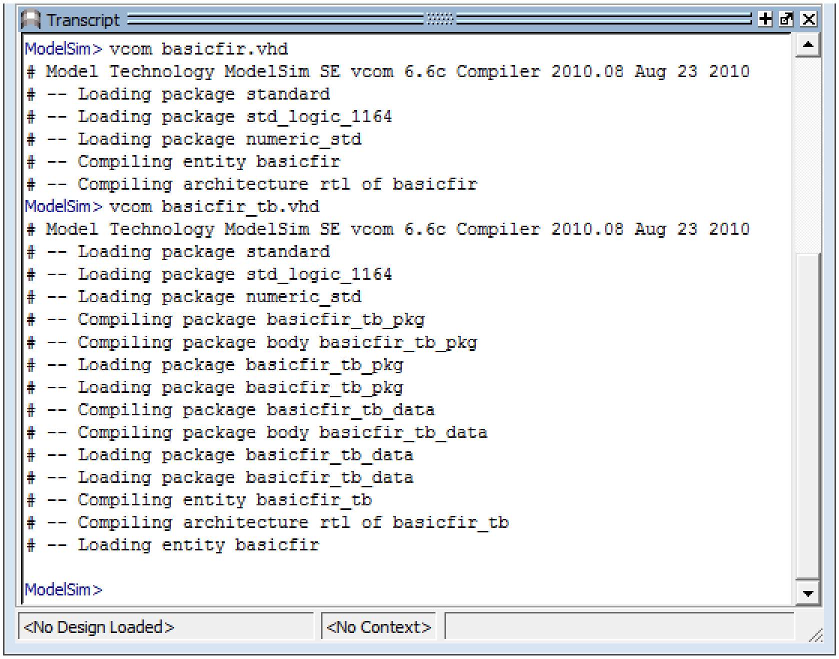


```
ModelSim> vlib work


ModelSim>
```

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

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

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

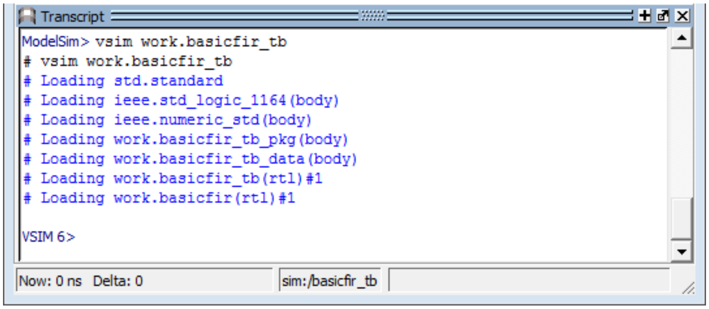


```
ModelSim> vcom basicfir.vhd
# Model Technology ModelSim SE vcom 6.6c Compiler 2010.08 Aug 23 2010
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity basicfir
# -- Compiling architecture rtl of basicfir
ModelSim> vcom basicfir_tb.vhd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

ModelSim>
```

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
```

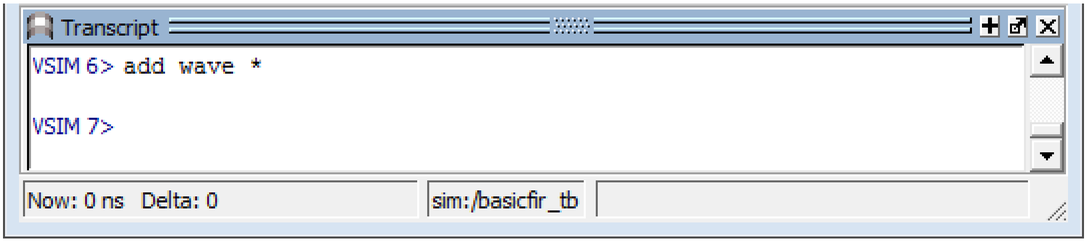
The following figure shows the results of loading `work.basicfir_tb` with the `vsim` command.



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

VSIM 6>
```

Now: 0 ns Delta: 0 | sim:/basicfir_tb

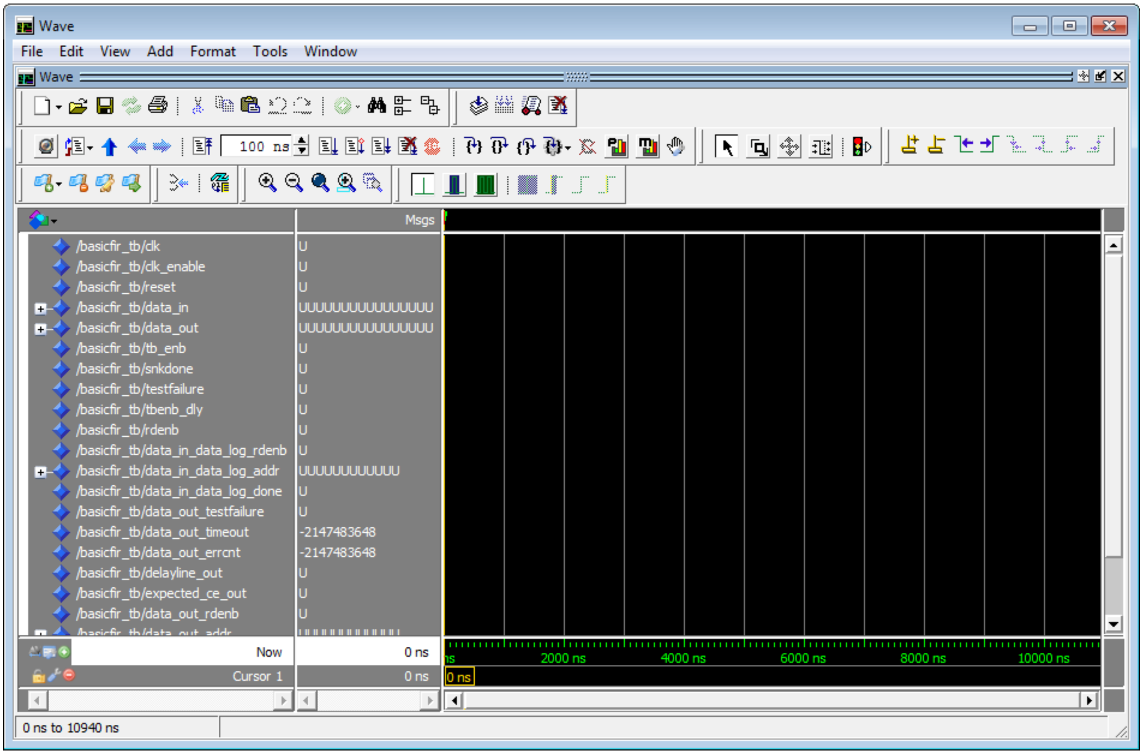
6 Open a display window for monitoring the simulation as the test bench runs. In the Mentor Graphics ModelSim simulator, use the following command to open a **wave** window and view the results of the simulation as HDL waveforms.



```
VSIM 6> add wave *

VSIM 7>
```

Now: 0 ns Delta: 0 | sim:/basicfir_tb

The following **wave** window displays.

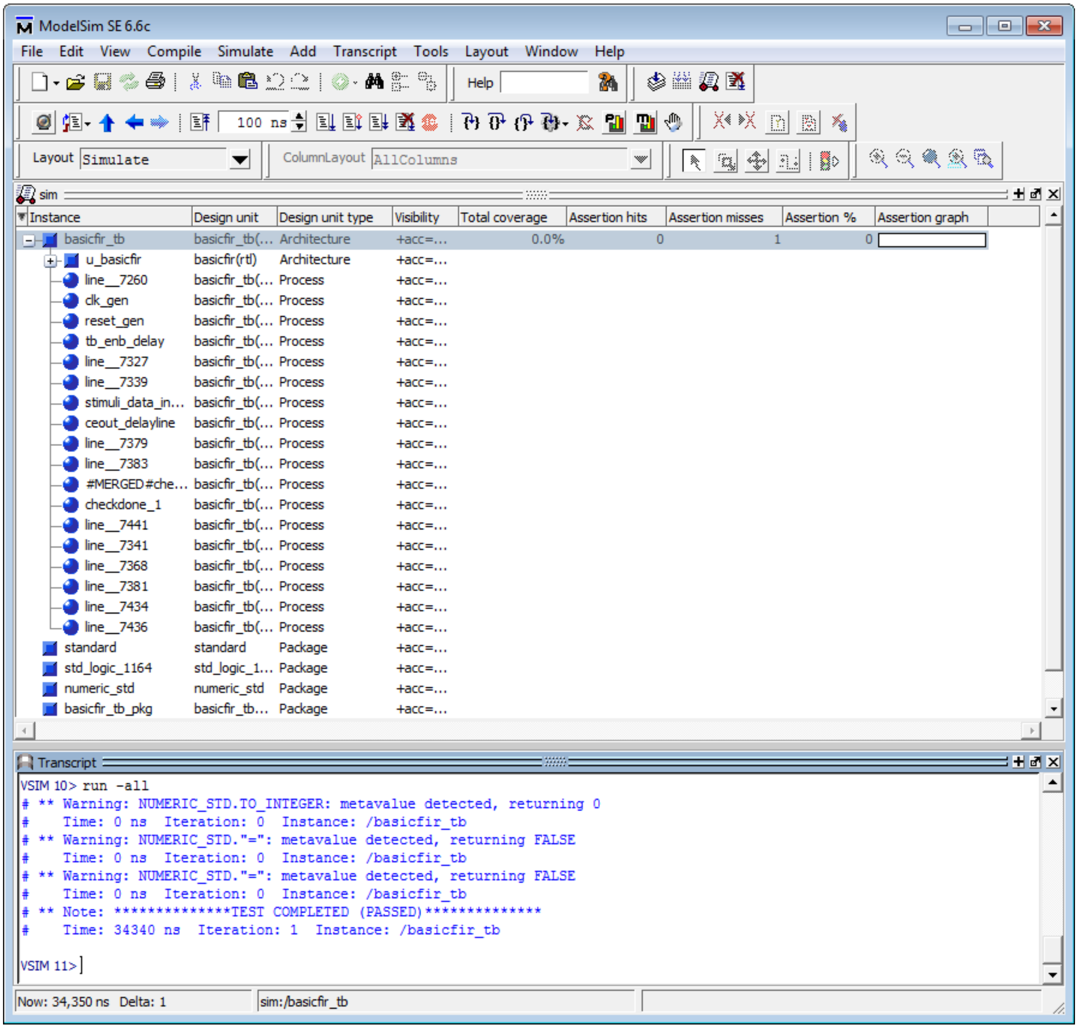


|  | Msgs |
|---|---|
| /basicfir_tb/clk | U |
| /basicfir_tb/clk_enable | U |
| /basicfir_tb/reset | U |
| /basicfir_tb/data_in | UUUUUUUUUUUUUUUU |
| /basicfir_tb/data_out | UUUUUUUUUUUUUUUU |
| /basicfir_tb/tb_enb | U |
| /basicfir_tb/snkdone | U |
| /basicfir_tb/testfailure | U |
| /basicfir_tb/tbenb_dly | U |
| /basicfir_tb/rdenb | U |
| /basicfir_tb/data_in_data_log_rdenb | U |
| /basicfir_tb/data_in_data_log_addr | UUUUUUUUUUUU |
| /basicfir_tb/data_in_data_log_done | U |
| /basicfir_tb/data_out_testfailure | U |
| /basicfir_tb/data_out_timeout | -2147483648 |
| /basicfir_tb/data_out_errcnt | -2147483648 |
| /basicfir_tb/delayline_out | U |
| /basicfir_tb/expected_ce_out | U |
| /basicfir_tb/data_out_rdenb | U |
| /basicfir_tb/data_out_addr | UUUUUUUUUUUU |
| Now | 0 ns |
| Cursor 1 | 0 ns |

0 ns to 10940 ns

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

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



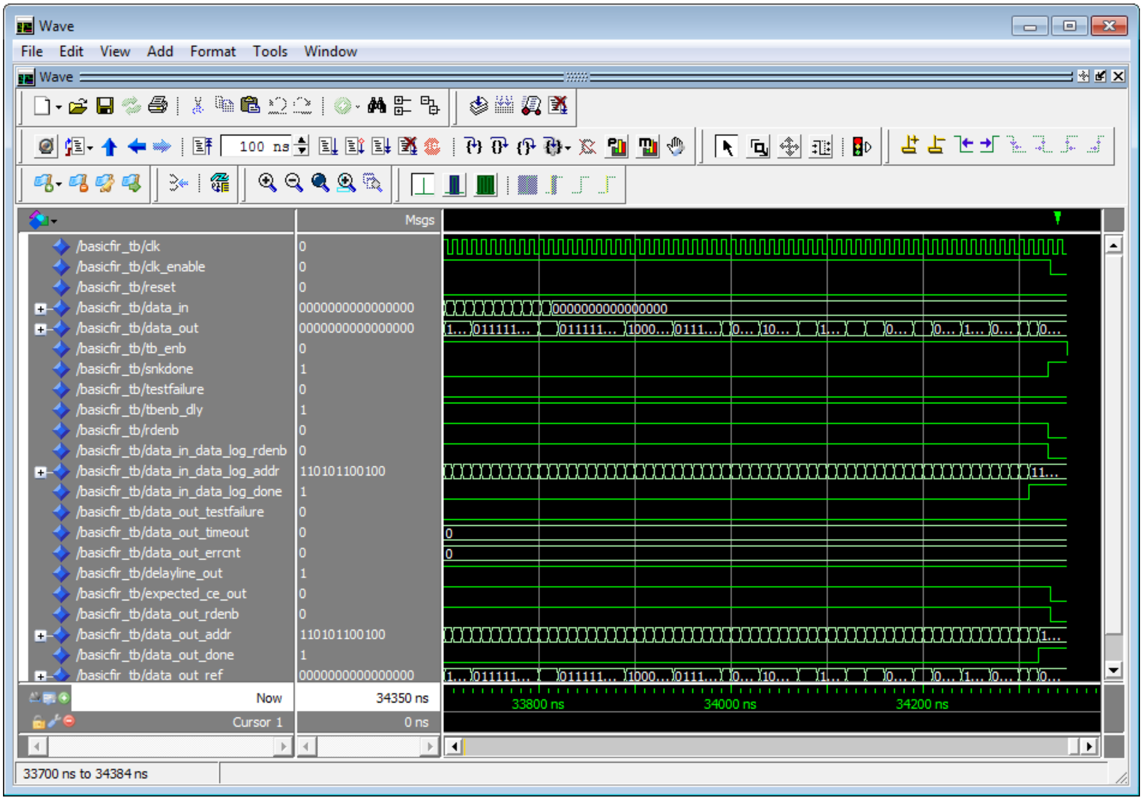
| Instance | Design unit | Design unit type | Visibility | Total coverage | Assertion hits | Assertion misses | Assertion % | Assertion graph |
|---|---|---|---|---|---|---|---|---|
| basicfir_tb | basicfir_tb(... | Architecture | +acc=... | 0.0% | 0 | 1 | 0 | |
| u_basicfir | basicfir(rtl) | Architecture | +acc=... | | | | | |
| line__7260 | basicfir_tb(... | Process | +acc=... | | | | | |
| clk_gen | basicfir_tb(... | Process | +acc=... | | | | | |
| reset_gen | basicfir_tb(... | Process | +acc=... | | | | | |
| tb_enb_delay | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7327 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7339 | basicfir_tb(... | Process | +acc=... | | | | | |
| stimuli_data_in... | basicfir_tb(... | Process | +acc=... | | | | | |
| ceout_delayline | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7379 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7383 | basicfir_tb(... | Process | +acc=... | | | | | |
| #MERGED#che... | basicfir_tb(... | Process | +acc=... | | | | | |
| checkdone_1 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7441 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7341 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7368 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7381 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7434 | basicfir_tb(... | Process | +acc=... | | | | | |
| line__7436 | basicfir_tb(... | Process | +acc=... | | | | | |
| standard | standard | Package | +acc=... | | | | | |
| std_logic_1164 | std_logic_1... | Package | +acc=... | | | | | |
| numeric_std | numeric_std | Package | +acc=... | | | | | |
| basicfir_tb_pkg | basicfir_tb... | Package | +acc=... | | | | | |

```
VSIM 10> run -all
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ns  Iteration: 0  Instance: /basicfir_tb
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /basicfir_tb
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /basicfir_tb
# ** Note: *************TEST COMPLETED (PASSED)*************
#    Time: 34340 ns  Iteration: 1  Instance: /basicfir_tb

VSIM 11>
```

Now: 34,350 ns  Delta: 1    sim:/basicfir_tb

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.

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



```
/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
33700 ns to 34384 ns
```

# Optimized FIR Filter

**In this section...**

## 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 FDATool

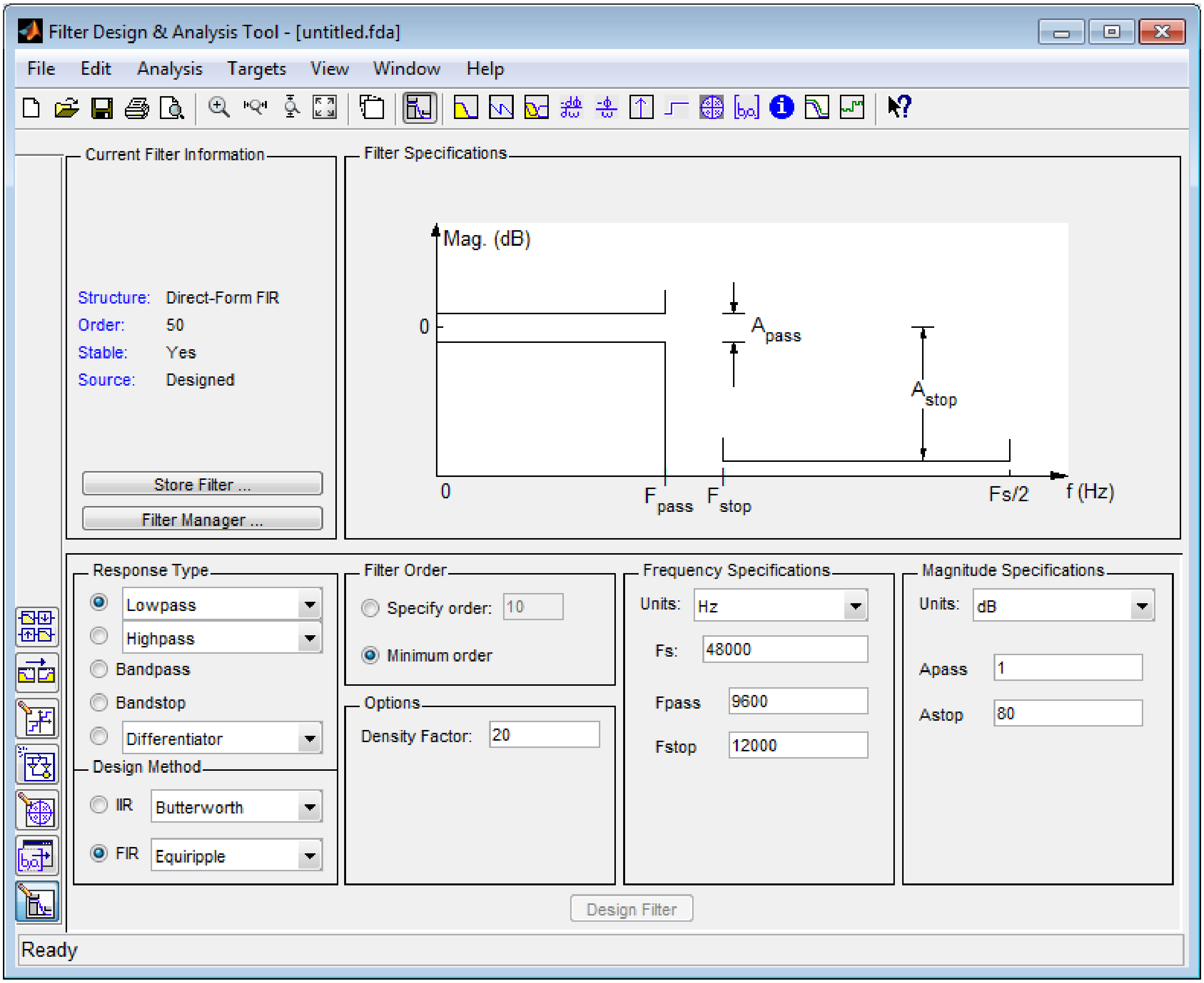
This tutorial guides you through the steps for designing an optimized quantized discrete-time FIR filter, generating Verilog code for the filter, and verifying the Verilog code with a generated test bench.

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

1 Start the MATLAB software.

2 Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-24.

3 Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response Type
- ● Lowpass
- ○ Highpass
- ○ Bandpass
- ○ Bandstop
- ○ Differentiator

Design Method
- ○ IIR Butterworth
- ● FIR Equiripple

Filter Order
- ○ Specify order: 10
- ● Minimum order

Options
Density Factor: 20

Frequency Specifications
Units: Hz
Fs: 48000
Fpass: 9600
Fstop: 12000

Magnitude Specifications
Units: dB
Apass: 1
Astop: 80

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

| Option | Value |
|---|---|
| **Response Type** | `Lowpass` |
| **Design Method** | `FIR Equiripple` |
| **Filter Order** | **Minimum order** |

| Option | Value |
|---|---|
| **Options** | `Density Factor: 20` |
| **Frequency Specifications** | **Units**: Hz<br>**Fs**: 48000<br>**Fpass**: 9600<br>**Fstop**: 12000 |
| **Magnitude Specifications** | **Units**: dB<br>**Apass**: 1<br>**Astop**: 80 |

These settings are for the default filter design that the FDATool 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 1-6.

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

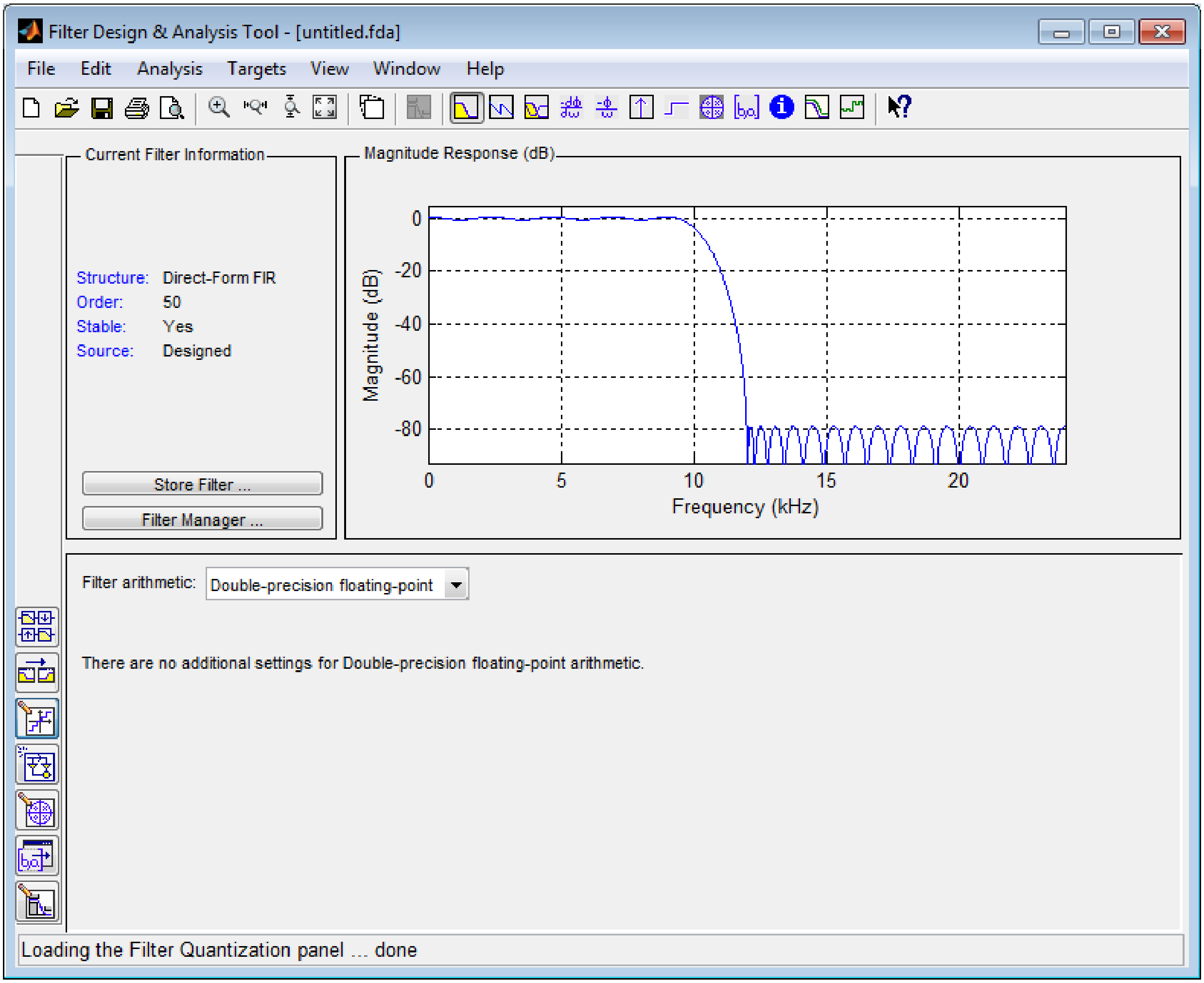
   ```
   Designing Filter... Done
   ```

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

## 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 FDATool" on page 1-24 if it is not already open.
2. Click the Set Quantization Parameters button in the left-side toolbar. The FDATool displays a **Filter arithmetic** menu in the bottom half of its dialog box.



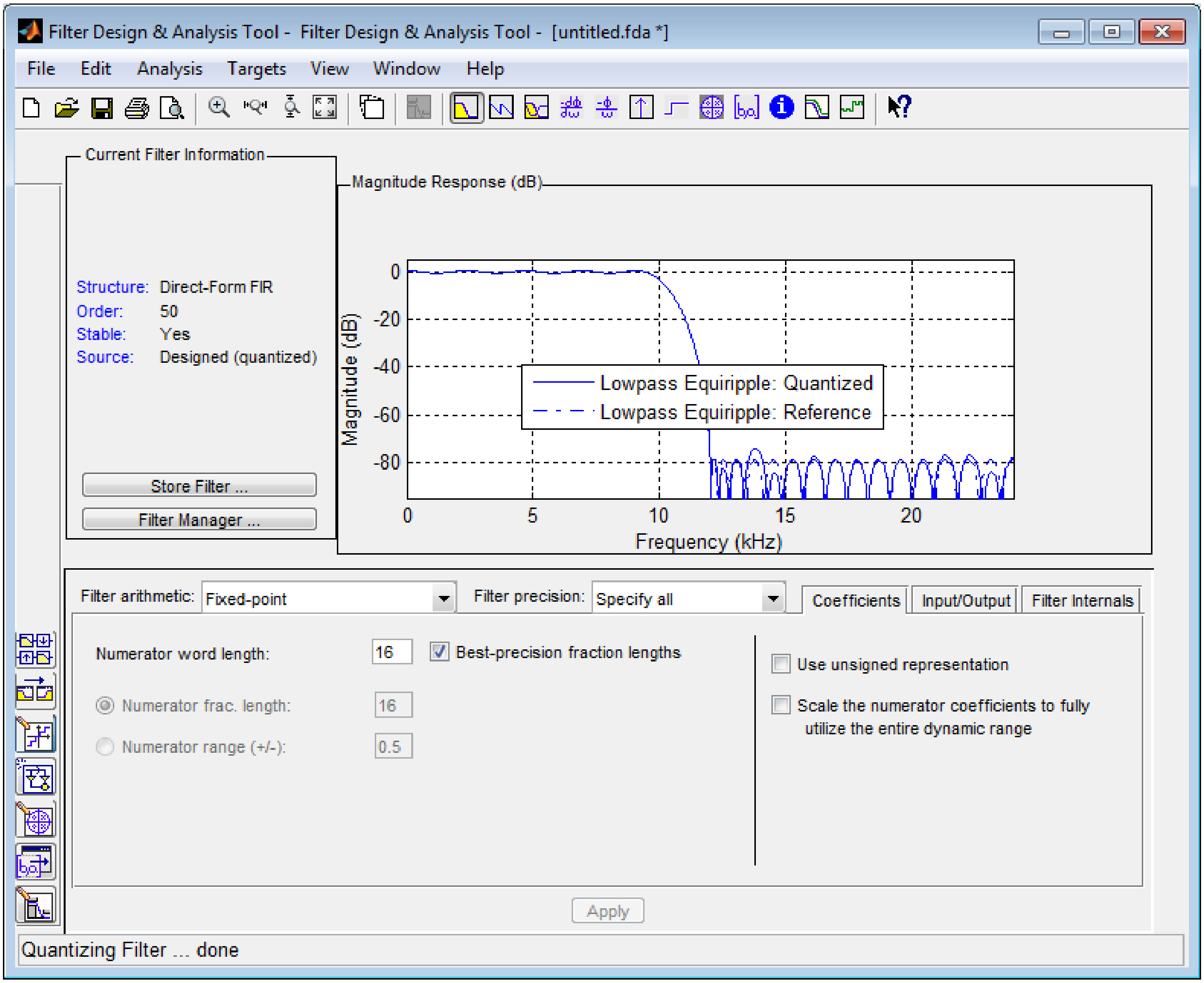
Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Filter arithmetic: Double-precision floating-point

There are no additional settings for Double-precision floating-point arithmetic.

Loading the Filter Quantization panel ... done

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



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed (quantized)

Filter arithmetic: Fixed-point

Filter precision: Specify all

Numerator word length: 16

☑ Best-precision fraction lengths

● Numerator frac. length: 16

○ Numerator range (+/-): 0.5

□ Use unsigned representation

□ Scale the numerator coefficients to fully utilize the entire dynamic range

Quantizing Filter ... done

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` |
| | **Accum. word length** | `40` |

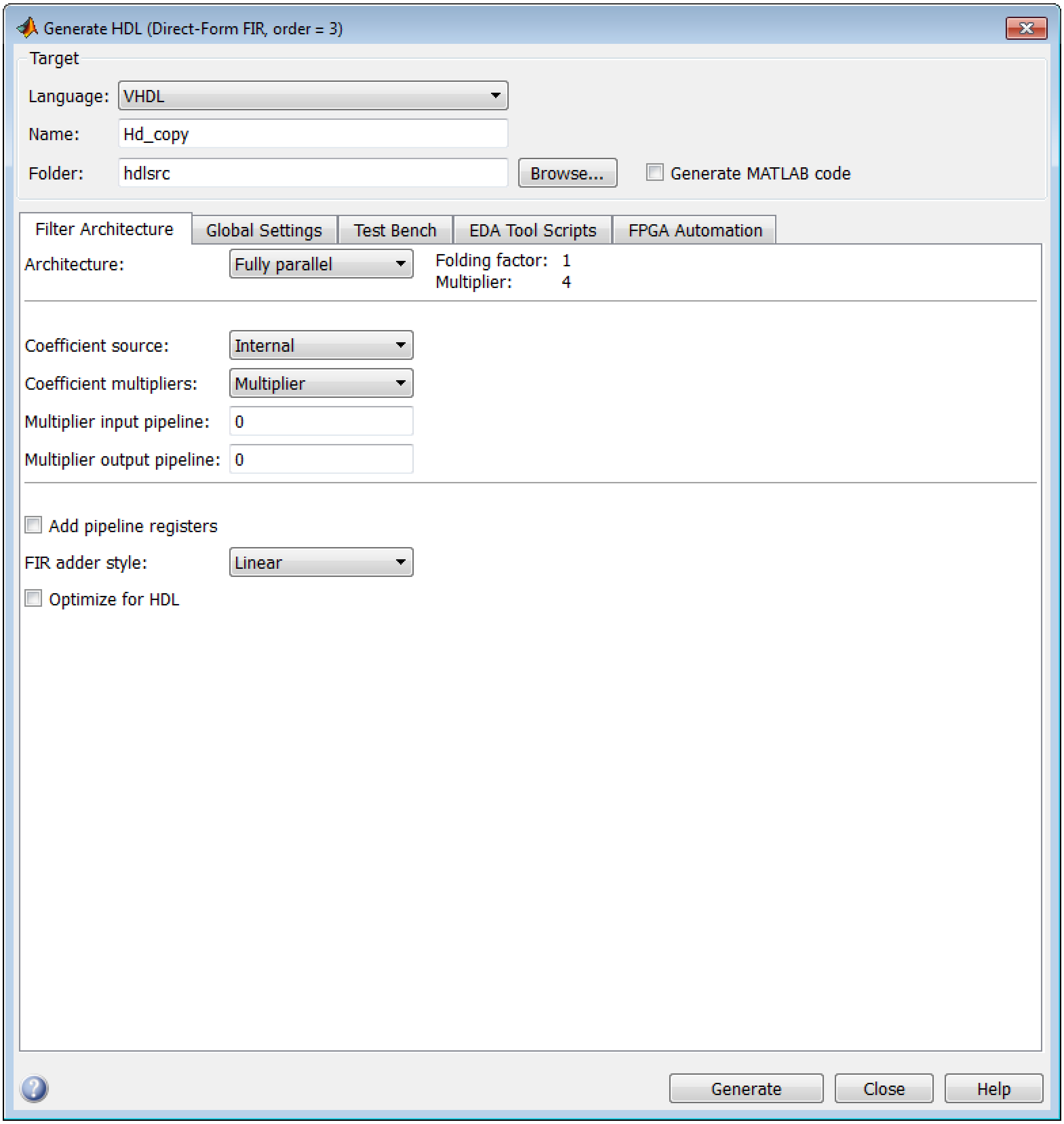
5 Click **Apply**.

For more information on quantizing filters with the FDATool, 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 GUI, 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 FDATool” on page 1-24 and “Quantize the FIR Filter” on page 1-26.

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



Generate HDL (Direct-Form FIR, order = 3)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 4

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

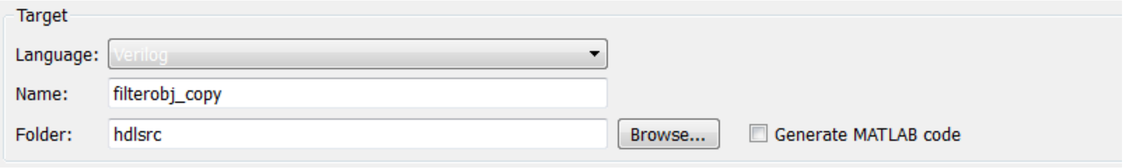
Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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



Target

Language: Verilog

Name: filterobj_copy

Folder: hdlsrc

□ Generate MATLAB code

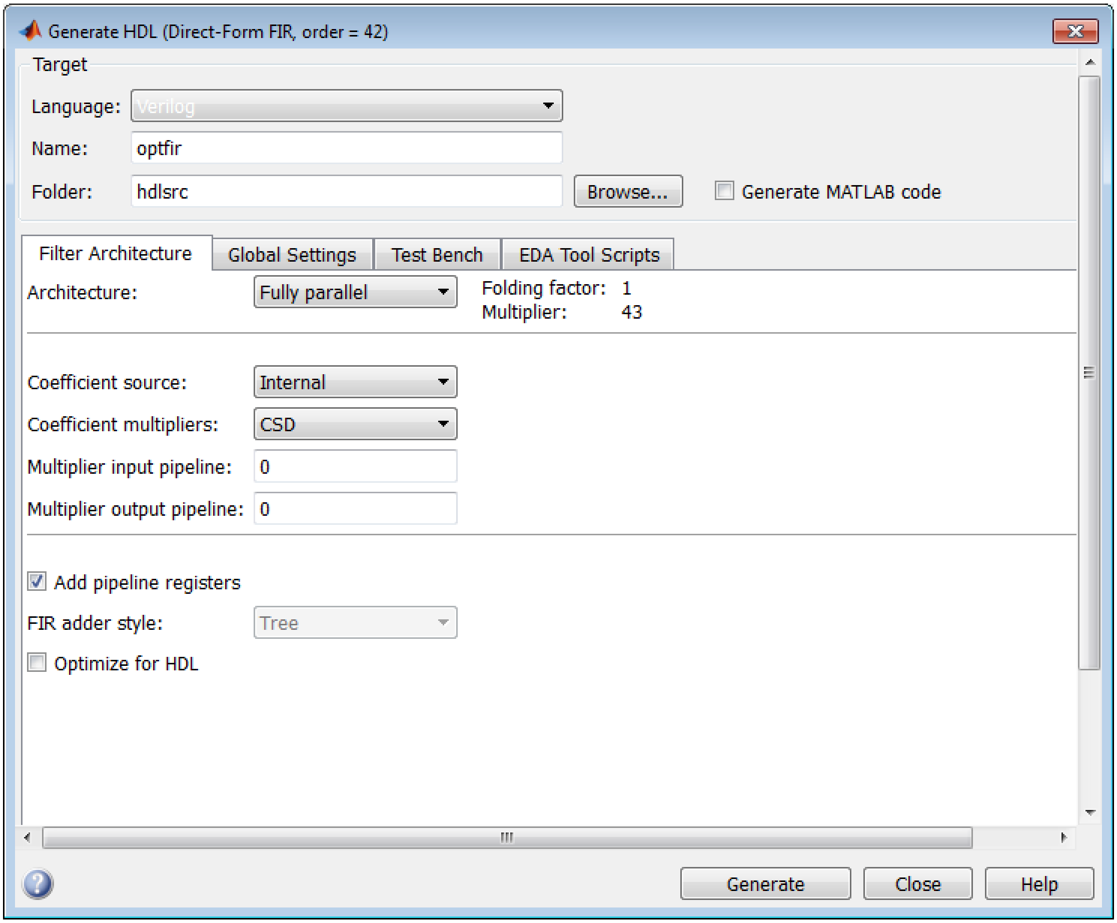
3 In the **Name** text box of the **Target** pane, replace the default name with `optfir`. This option names the Verilog module and the file that contains the Verilog code for the filter.

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 dialog box now appears as shown.



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

Target

Language: Verilog

Name: optfir

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 43

Coefficient source: Internal

Coefficient multipliers: CSD

Multiplier input pipeline: 0

Multiplier output pipeline: 0

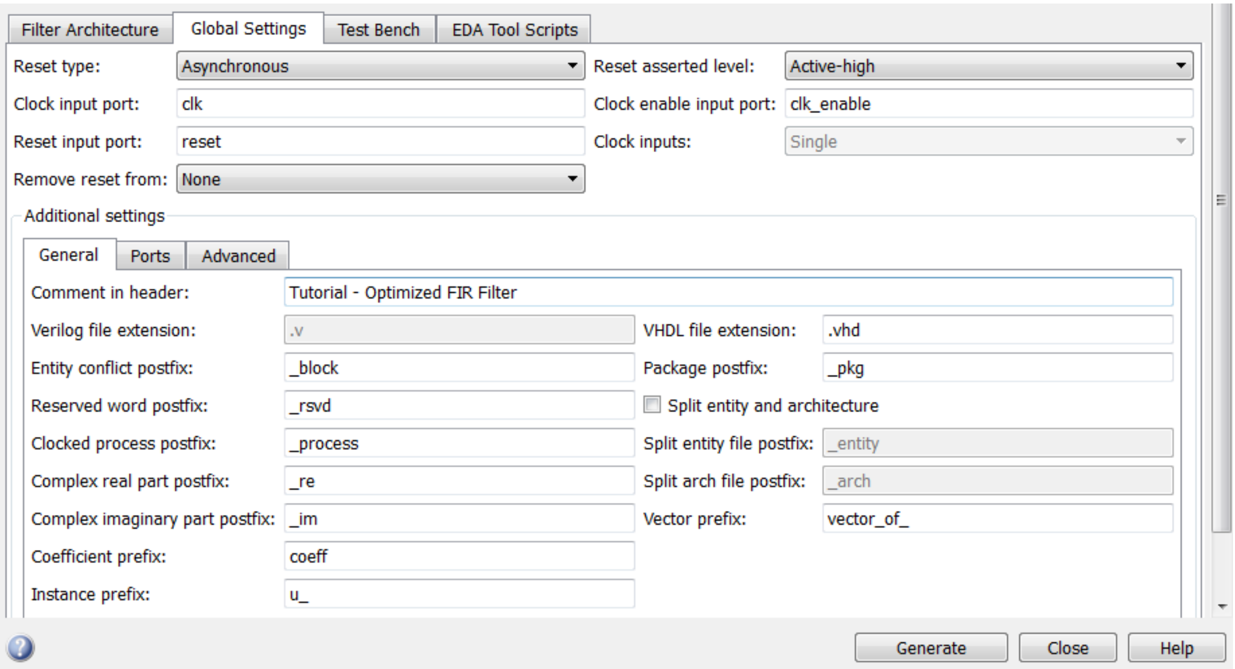
☑ Add pipeline registers

FIR adder style: Tree

□ Optimize for HDL

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

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



Reset type: Asynchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

Comment in header: Tutorial - Optimized FIR Filter

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

□ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

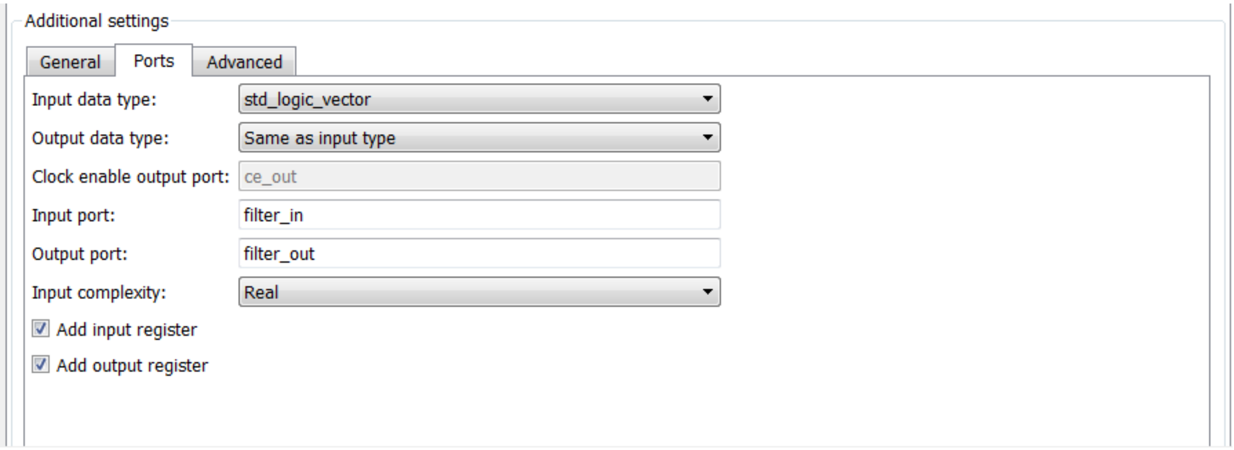
Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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



Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

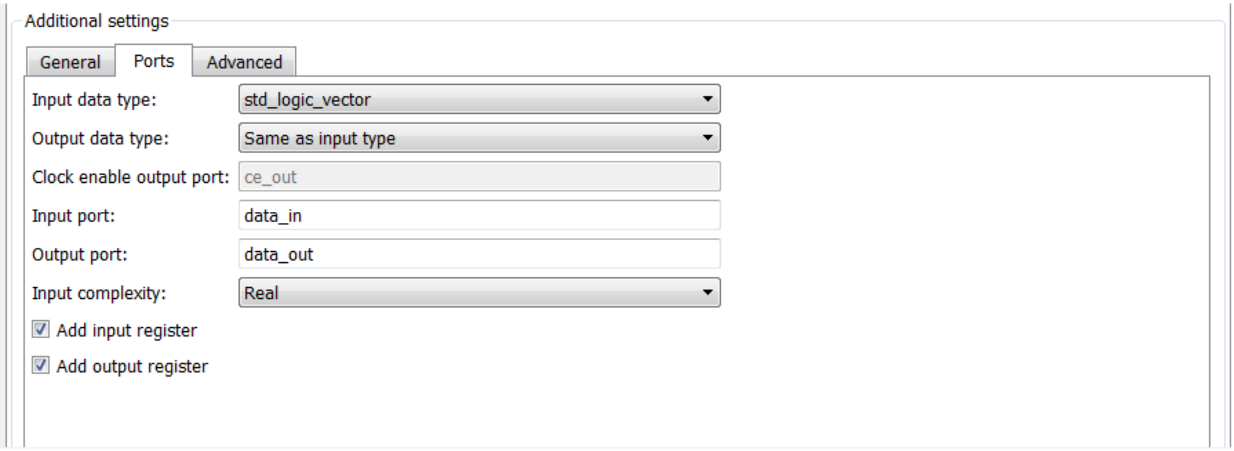
Output port: filter_out

Input complexity: Real

☑ Add input register

☑ Add output register

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



Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: data_in

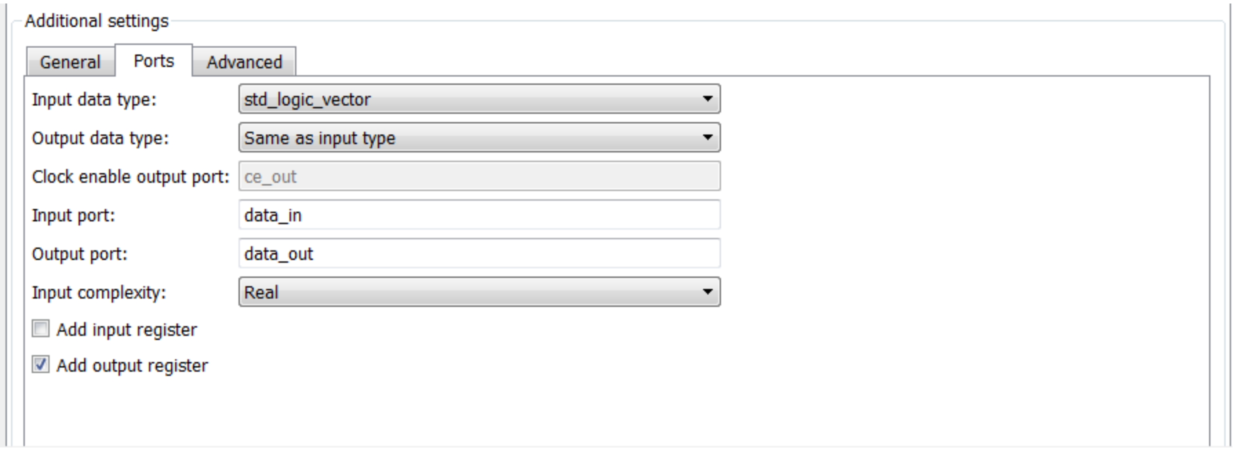
Output port: data_out

Input complexity: Real

☑ Add input register

☑ Add output register

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



Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: data_in

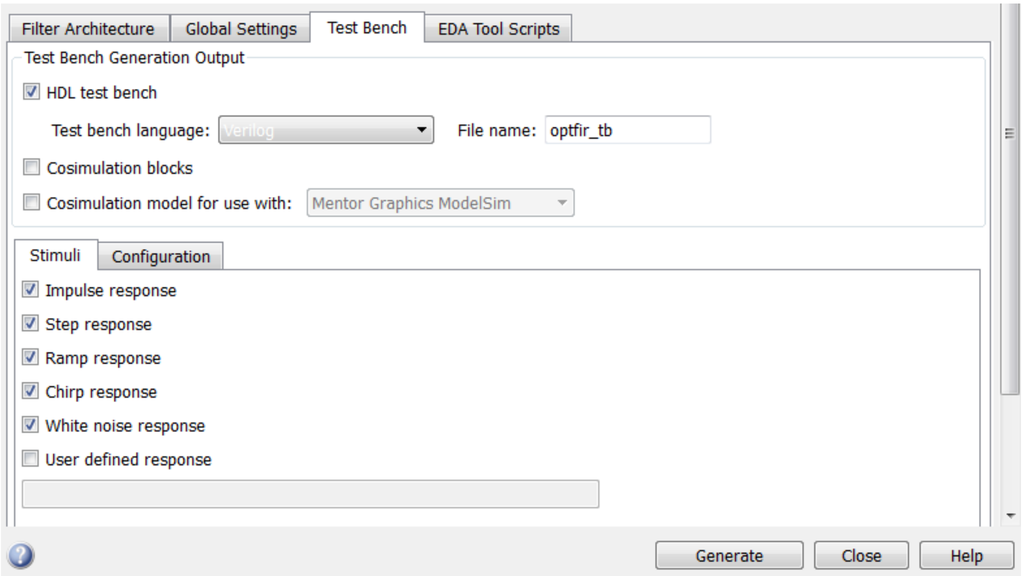
Output port: data_out

Input complexity: Real

□ Add input register

☑ Add output register

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



Test Bench Generation Output

☑ HDL test bench

Test bench language: Verilog

File name: optfir_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

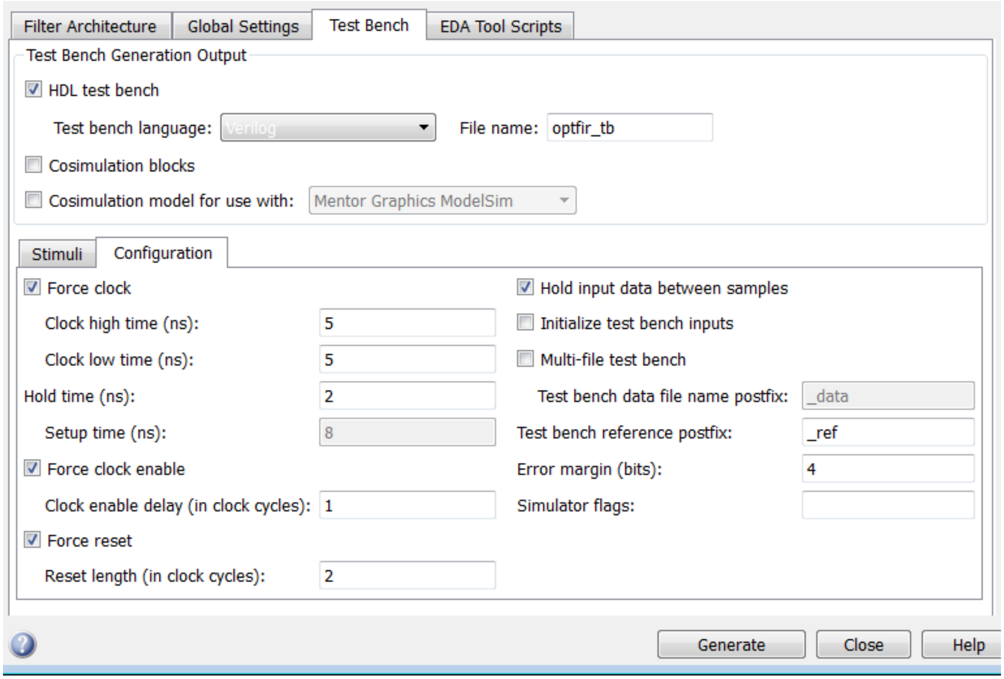
☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

**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.



Test Bench Generation Output

☑ HDL test bench

Test bench language: Verilog    File name: optfir_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Error margin (bits): 4

Simulator flags:

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

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

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

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

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

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

The generated Verilog code has the following characteristics:

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

| Verilog Port | Name |
|---|---|
| Input | `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 string `_process` is appended to sequential (`begin`) block names.

The generated test bench:

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

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

## 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 string you specified for the **Name** option in the **Target** pane. See step 3 in "Configure and Generate Optimized Verilog Code" on page 1-29.
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 1-29.
4. Search for `HDL Code`. This section lists the coder options you modified in "Configure and Generate Optimized Verilog Code" on page 1-29.
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 FDATool" on page 1-24 and "Quantize the FIR Filter" on page 1-26.
6. Search for `module`. This line names the Verilog module, using the string you specified for the **Name** option in the **Target** pane. This line also declares the list of ports, as defined by options on the **Ports** pane of the Generate HDL dialog box. The ports for data input and output are named with the strings you specified for the **Input port** and **Output port** options on the **Ports** tab of the Generate HDL dialog box. See steps 3 and 11 in "Configure and Generate Optimized Verilog Code" on page 1-29.
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 1-29.

11 Search for `process`. The `block` name `Delay_Pipeline_process` includes the default `block` postfix string `_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 1-29.

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 1-29 , you cleared the **Add input register** option, but left the **Add output register** selected. Also note that the process name `Output_Register_process` includes the default `process` postfix string `_process`.

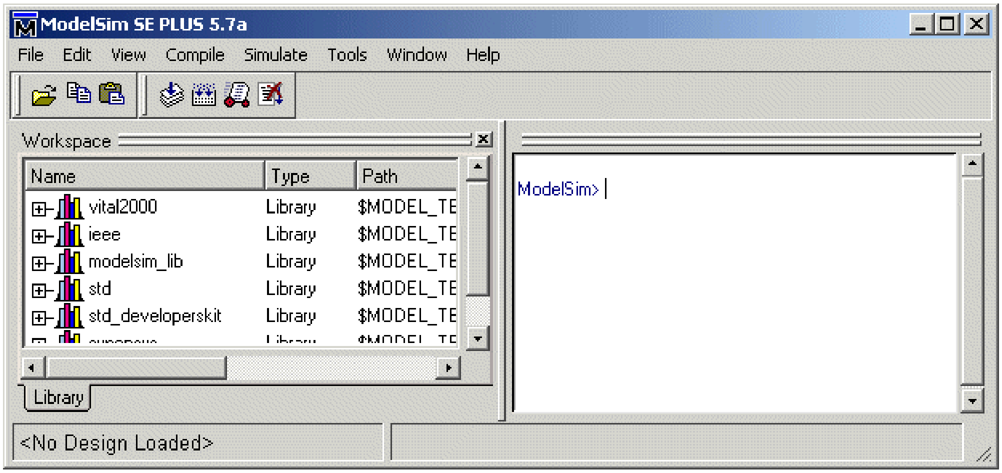
16 Search for `data_out`. This code drives the output data of the filter.

## 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 the following steps:

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



| Name | Type | Path |
|---|---|---|
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |

```
ModelSim>
```

<No Design Loaded>

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

```
cd hdlsrc
```

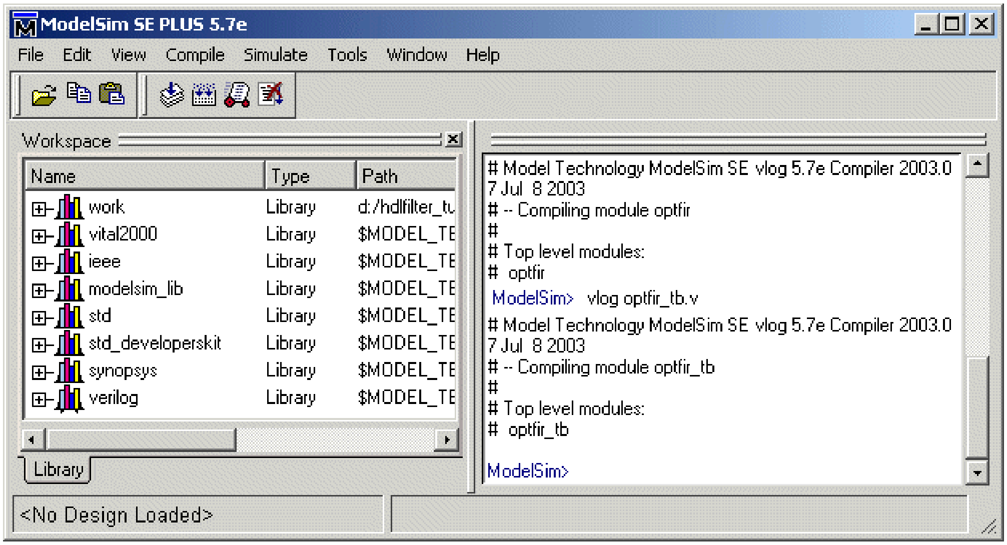
3 If desired, create a design library to store the compiled Verilog modules. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

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

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

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



| Name | Type | Path |
|---|---|---|
| work | Library | d:/hdlfilter_tu |
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |
| synopsys | Library | $MODEL_TE |
| verilog | Library | $MODEL_TE |

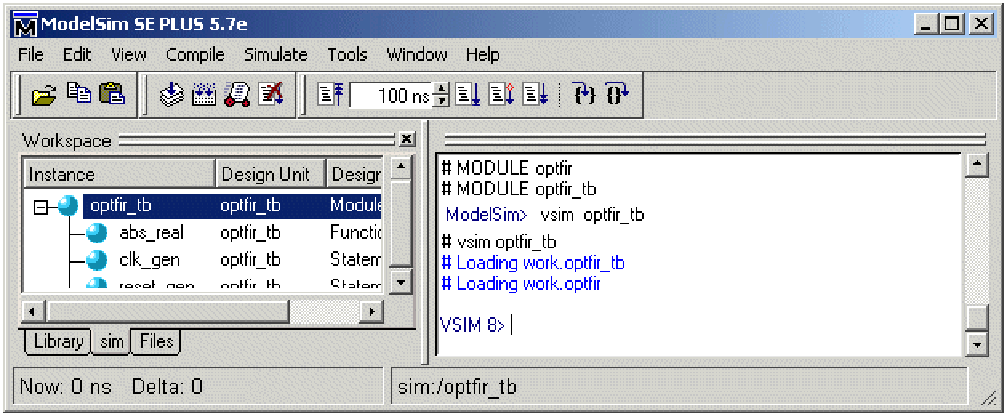
```
# Model Technology ModelSim SE vlog 5.7e Compiler 2003.0
7 Jul  8 2003
# -- Compiling module optfir
#
# Top level modules:
#   optfir
 ModelSim>  vlog optfir_tb.v
# Model Technology ModelSim SE vlog 5.7e Compiler 2003.0
7 Jul  8 2003
# -- Compiling module optfir_tb
#
# Top level modules:
#   optfir_tb

ModelSim>
```

<No Design Loaded>

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
```

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



| Instance | Design Unit | Desigr |
|---|---|---|
| optfir_tb | optfir_tb | Module |
| abs_real | optfir_tb | Functic |
| clk_gen | optfir_tb | Staterr |
| reset_gen | optfir_tb | Statem |

```
# MODULE optfir
# MODULE optfir_tb
 ModelSim>  vsim  optfir_tb
# vsim optfir_tb
# Loading work.optfir_tb
# Loading work.optfir

VSIM 8> |
```

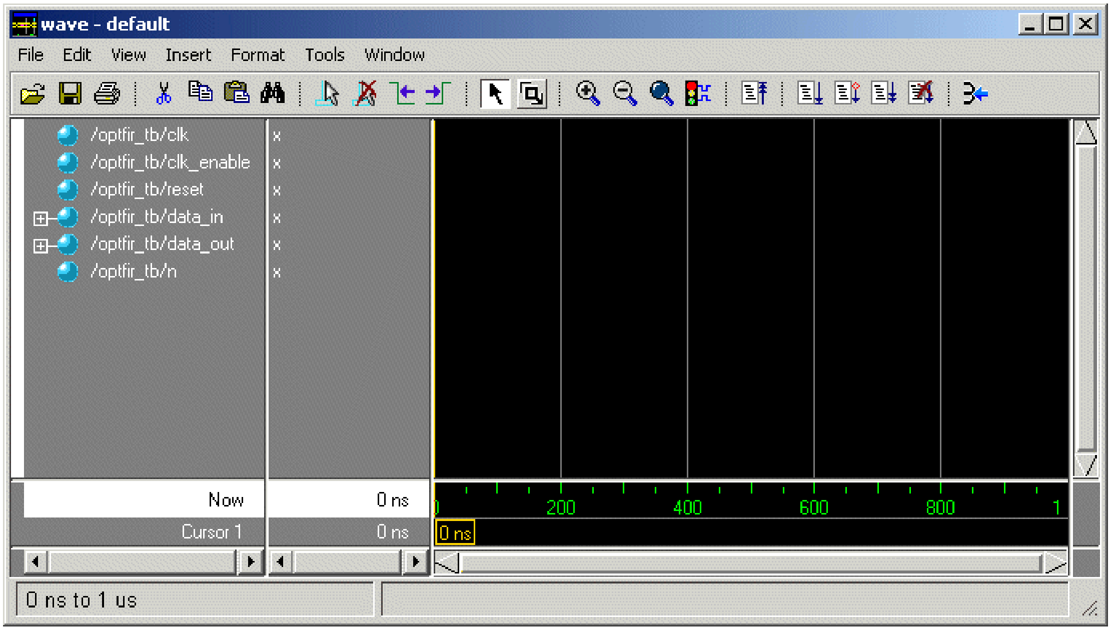
Now: 0 ns   Delta: 0

sim:/optfir_tb

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

```
add wave *
```

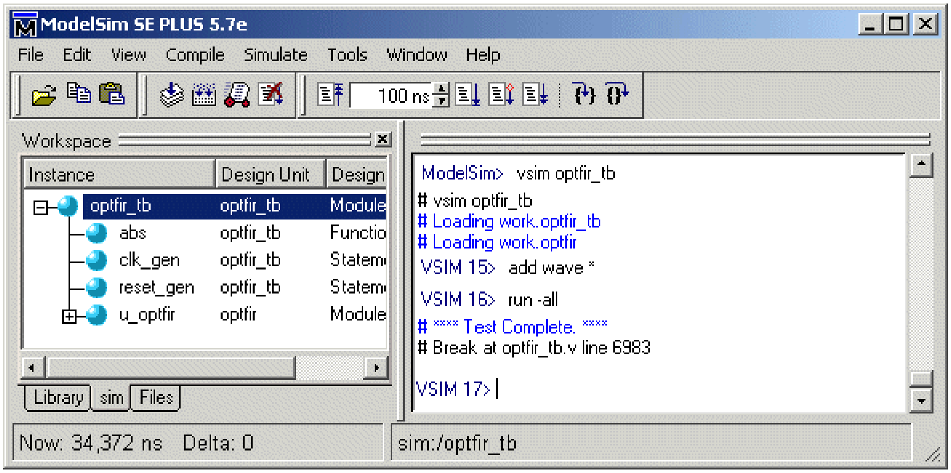
The following **wave** window opens:



```
/optfir_tb/clk         x
/optfir_tb/clk_enable  x
/optfir_tb/reset       x
/optfir_tb/data_in     x
/optfir_tb/data_out    x
/optfir_tb/n           x

Now                    0 ns
Cursor 1               0 ns    0 ns

0 ns to 1 us
```

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

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



| Instance | Design Unit | Design |
|---|---|---|
| optfir_tb | optfir_tb | Module |
| abs | optfir_tb | Functio |
| clk_gen | optfir_tb | Statem |
| reset_gen | optfir_tb | Statem |
| u_optfir | optfir | Module |

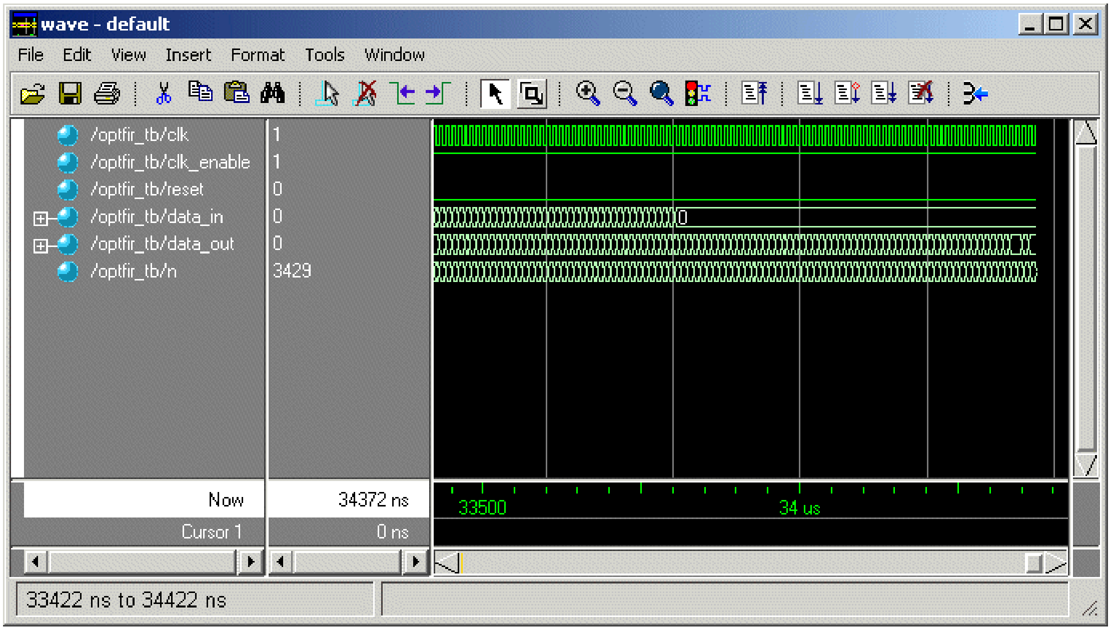
```
ModelSim> vsim optfir_tb
# vsim optfir_tb
# Loading work.optfir_tb
# Loading work.optfir
VSIM 15> add wave *
VSIM 16> run -all
# **** Test Complete. ****
# Break at optfir_tb.v line 6983
VSIM 17> |
```

Now: 34,372 ns Delta: 0

sim:/optfir_tb

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.

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



/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

33422 ns to 34422 ns

# IIR Filter

In this section...

## Create a Folder for Your Tutorial Files

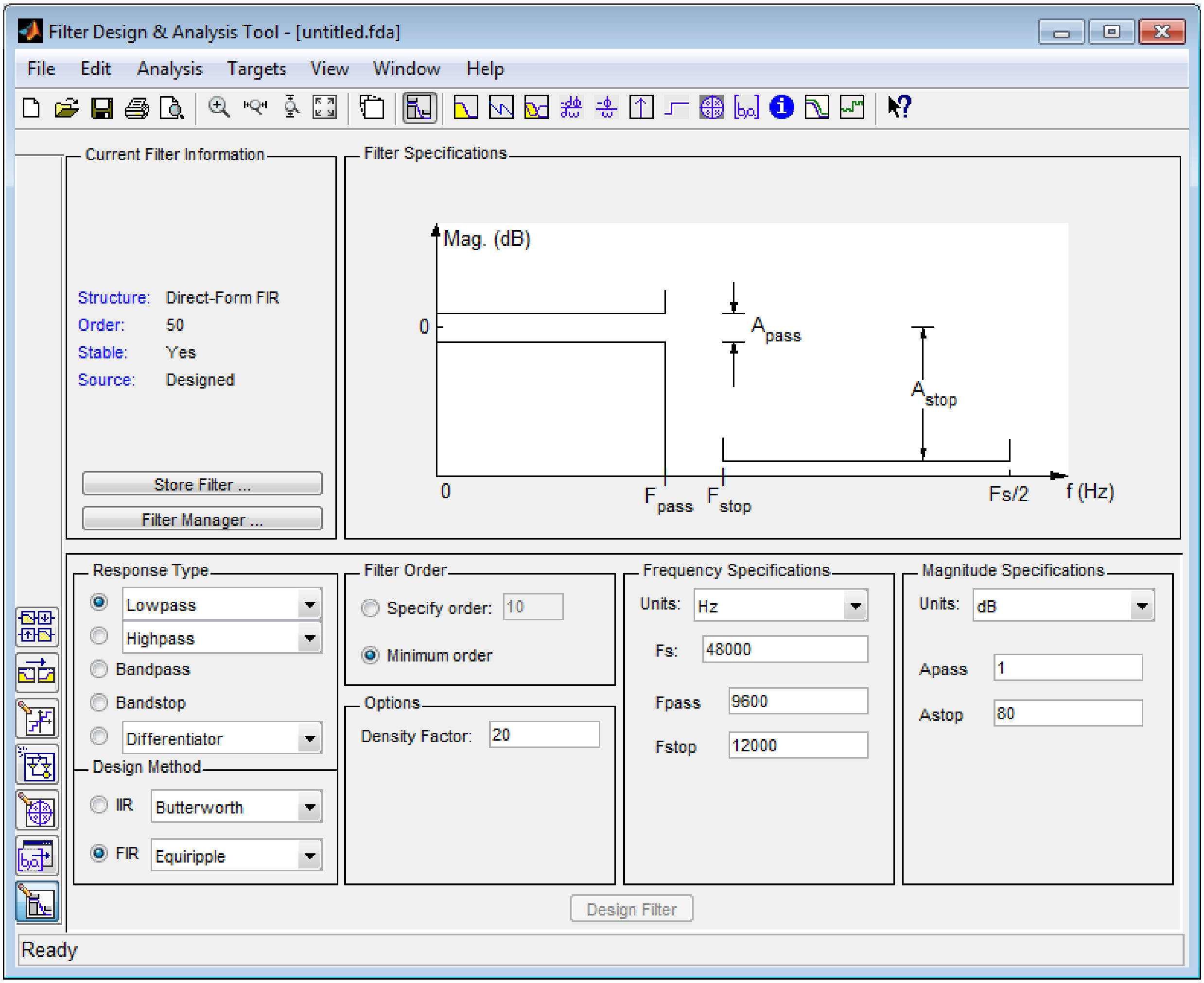
Set up a writable working folder outside your MATLAB installation folder to store files that will be generated as you complete your tutorial work. The tutorial instructions assume that you create the folder `hdlfilter_tutorials` on drive C.

## Design an IIR Filter in FDATool

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

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

1. Start the MATLAB software.
2. Set your current folder to the folder you created in "Create a Folder for Your Tutorial Files" on page 1-45.
3. Start the FDATool by entering the `fdatool` command in the MATLAB Command Window. The Filter Design & Analysis Tool dialog box appears.



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response Type
- ● Lowpass
- ○ Highpass
- ○ Bandpass
- ○ Bandstop
- ○ Differentiator

Design Method
- ○ IIR Butterworth
- ● FIR Equiripple

Filter Order
- ○ Specify order: 10
- ● Minimum order

Options
Density Factor: 20

Frequency Specifications
Units: Hz
Fs: 48000
Fpass: 9600
Fstop: 12000

Magnitude Specifications
Units: dB
Apass: 1
Astop: 80

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

| Option | Value |
|---|---|
| **Response Type** | `Highpass` |
| **Design Method** | `IIR Butterworth` |
| **Filter Order** | `Specify order: 5` |

| Option | Value |
|---|---|
| **Frequency Specifications** | **Units**: Hz<br>**Fs**: 48000<br>**Fc**: 10800 |

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

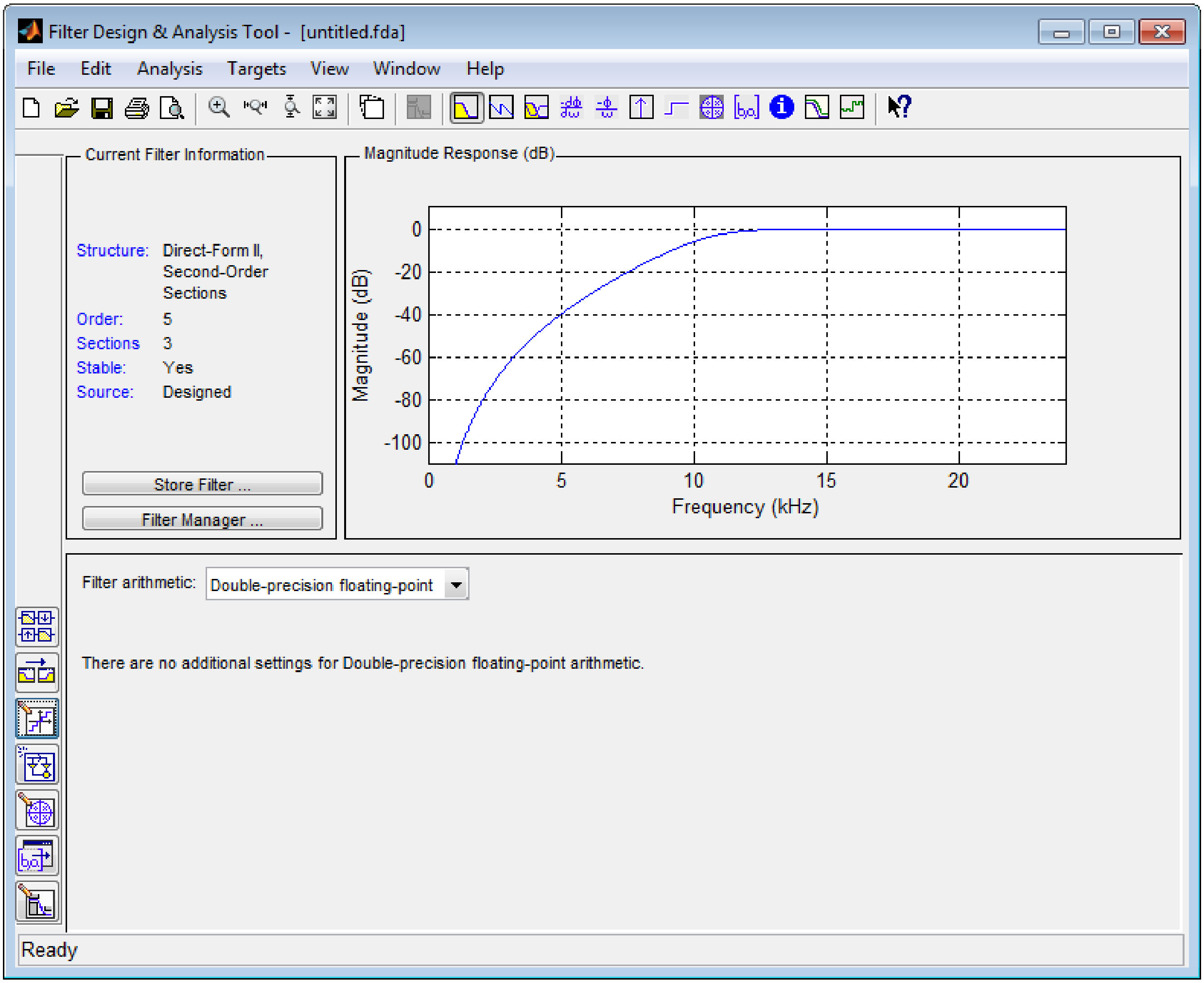
```
Designing Filter... Done
```

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

## 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 FDATool” on page 1-45 if it is not already open.
2. Click the Set Quantization Parameters button in the left-side toolbar. The FDATool displays the **Filter arithmetic** list in the bottom half of its dialog box.



Structure: Direct-Form II, Second-Order Sections

Order: 5

Sections 3

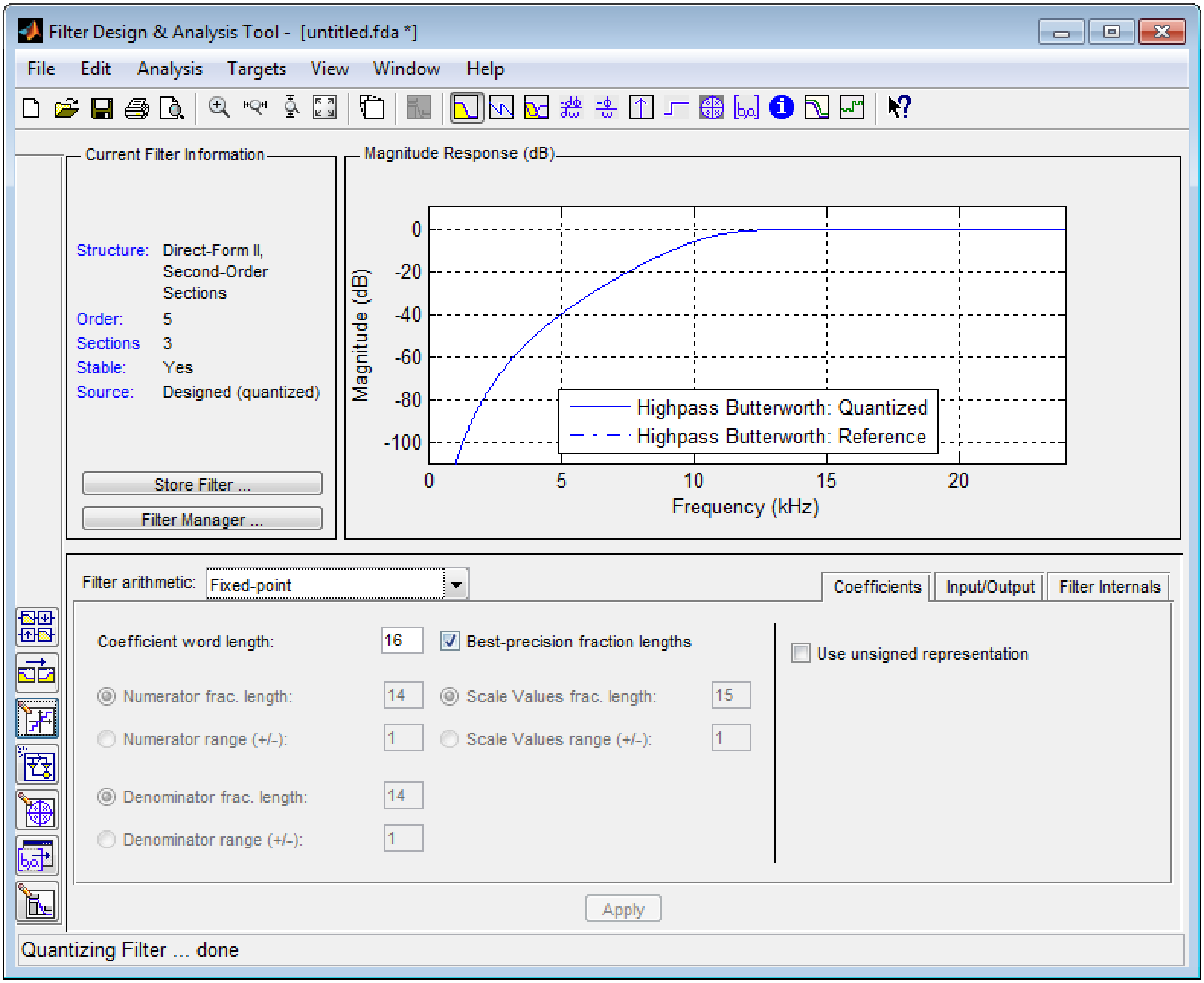
Stable: Yes

Source: Designed

Filter arithmetic: Double-precision floating-point

There are no additional settings for Double-precision floating-point arithmetic.

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



Current Filter Information

Structure: Direct-Form II, Second-Order Sections
Order: 5
Sections 3
Stable: Yes
Source: Designed (quantized)

Filter arithmetic: Fixed-point

Coefficient word length: 16

☑ Best-precision fraction lengths

□ Use unsigned representation

● Numerator frac. length: 14

○ Numerator range (+/-): 1

● Scale Values frac. length: 15

○ Scale Values range (+/-): 1

● Denominator frac. length: 14

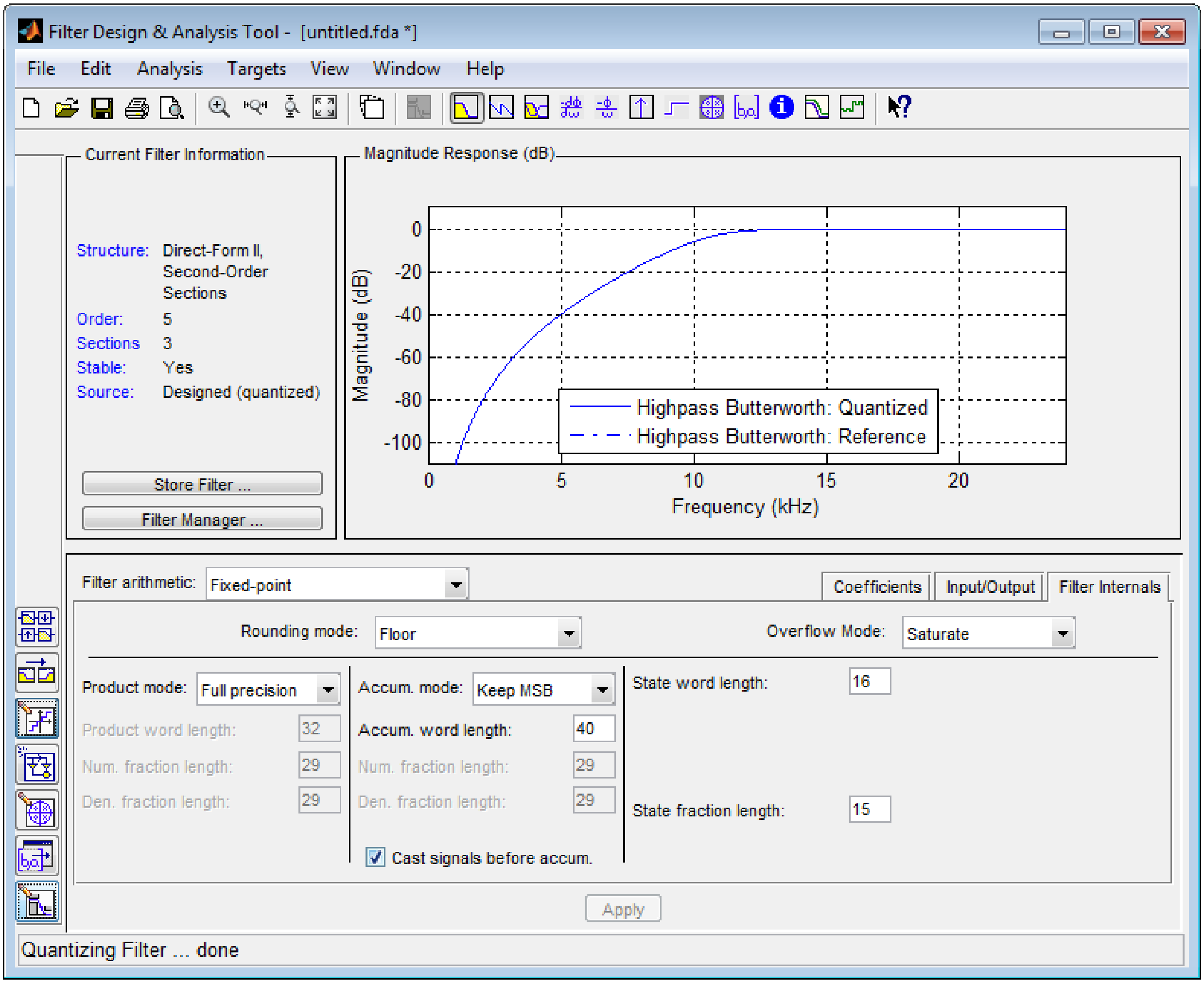
○ Denominator range (+/-): 1

Quantizing Filter ... done

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.



Filter Design & Analysis Tool - [untitled.fda *]

Current Filter Information

Structure: Direct-Form II, Second-Order Sections

Order: 5

Sections 3

Stable: Yes

Source: Designed (quantized)

Filter arithmetic: Fixed-point

Rounding mode: Floor

Overflow Mode: Saturate

Product mode: Full precision

Product word length: 32

Num. fraction length: 29

Den. fraction length: 29

Accum. mode: Keep MSB

Accum. word length: 40

Num. fraction length: 29

Den. fraction length: 29

☑ Cast signals before accum.

State word length: 16

State fraction length: 15

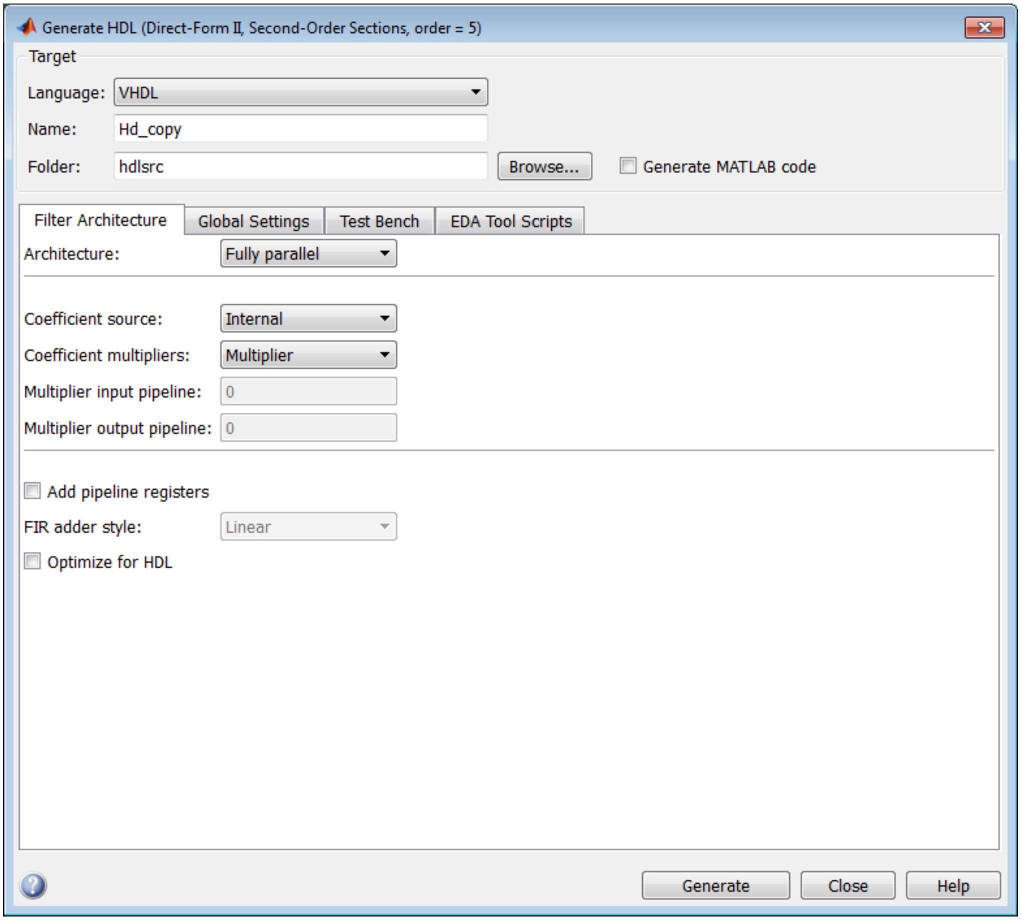
Quantizing Filter ... done

For more information on quantizing filters with the FDATool, see “Use FDATool with DSP System Toolbox Software” in 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. This section guides you through starting the Filter Design HDL Coder GUI, 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 FDATool” on page 1-45 and “Quantize the IIR Filter” on page 1-47.

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



Generate HDL (Direct-Form II, Second-Order Sections, order = 5)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

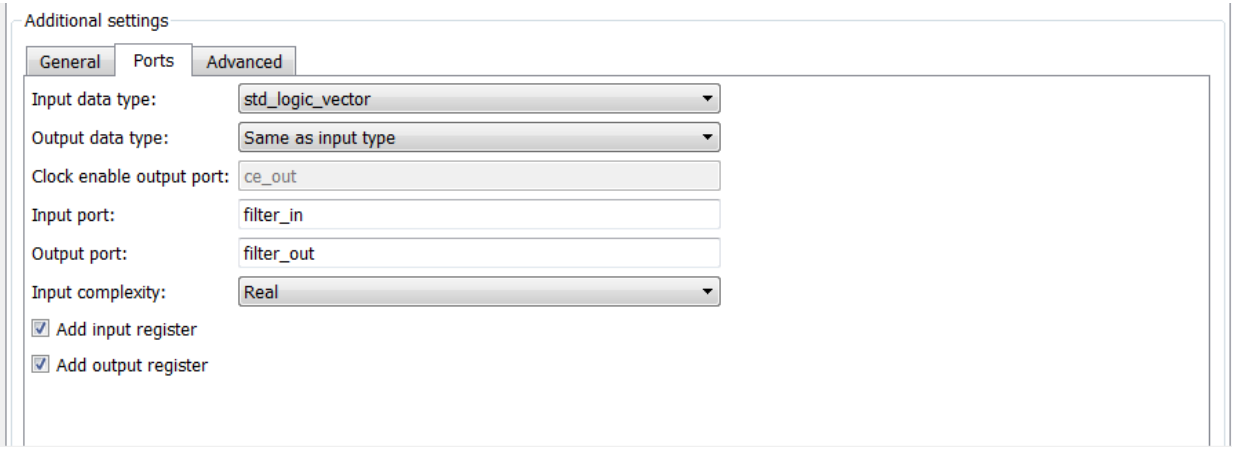
□ Optimize for HDL

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 GUI. Then select the **General** tab of the **Additional settings** section.

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

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



Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

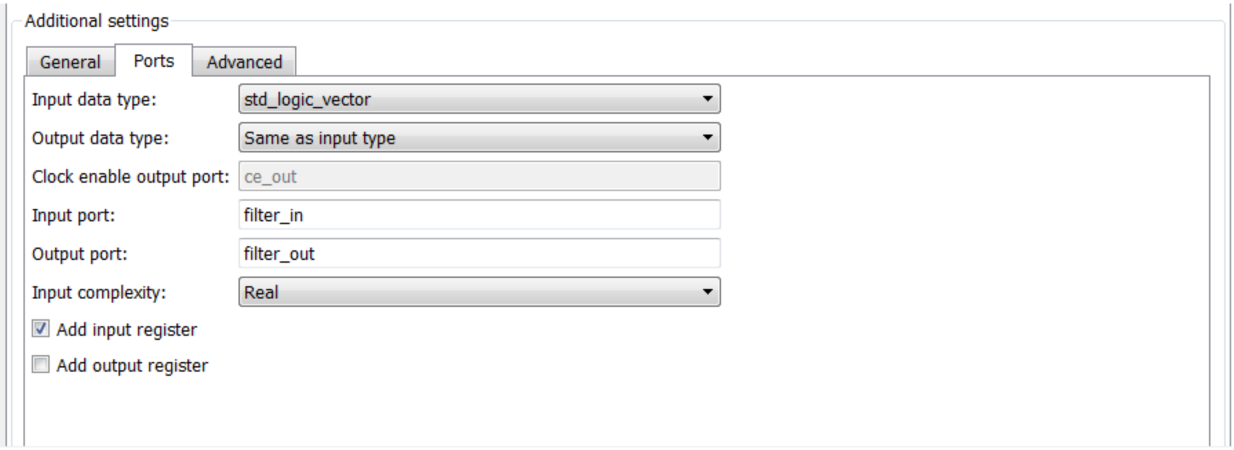
Output port: filter_out

Input complexity: Real

☑ Add input register

☑ Add output register

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



Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

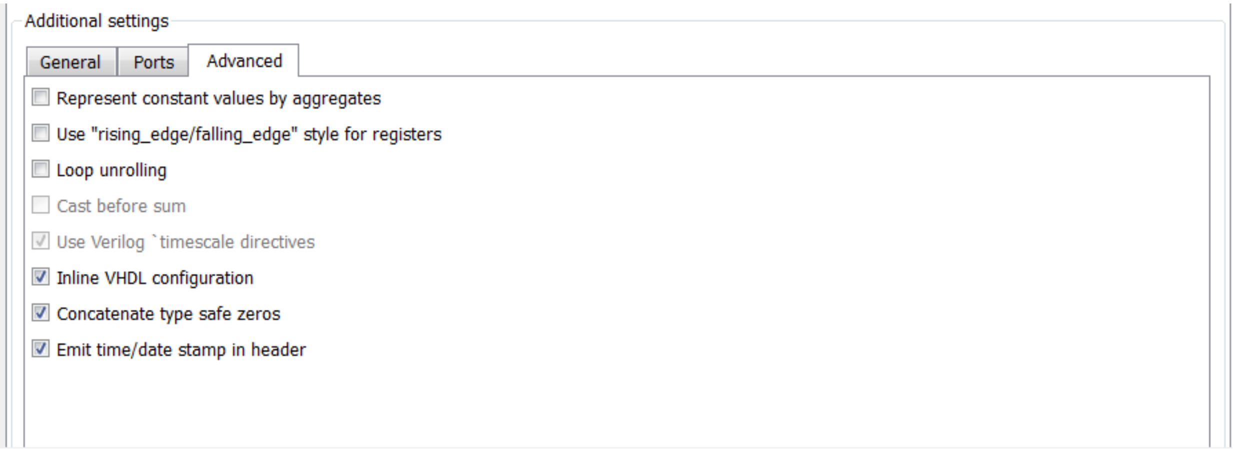
Output port: filter_out

Input complexity: Real

☑ Add input register

□ Add output register

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



Additional settings

□ Represent constant values by aggregates

□ Use "rising_edge/falling_edge" style for registers

□ Loop unrolling

□ Cast before sum

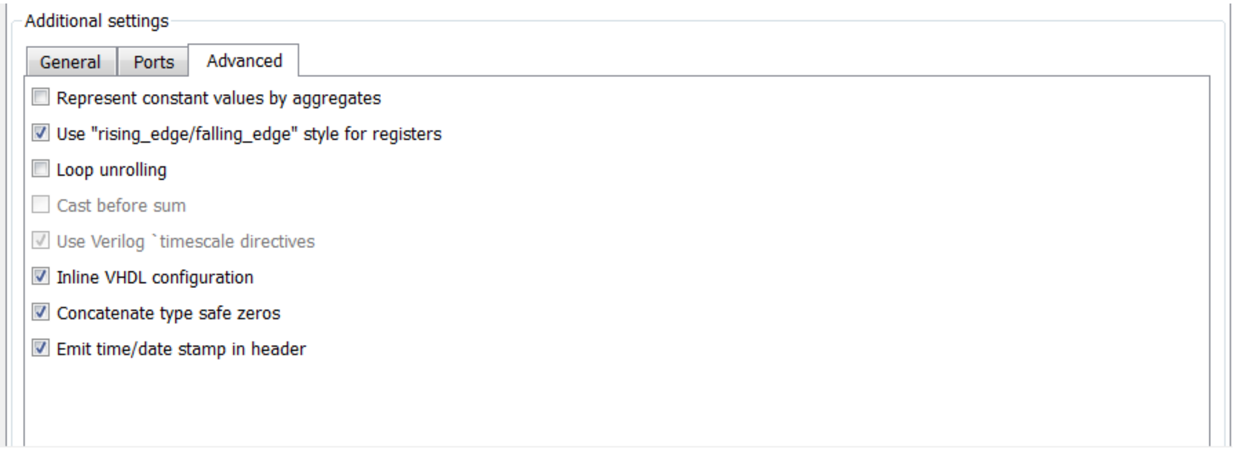
☑ Use Verilog `timescale directives

☑ Inline VHDL configuration

☑ Concatenate type safe zeros

☑ Emit time/date stamp in header

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

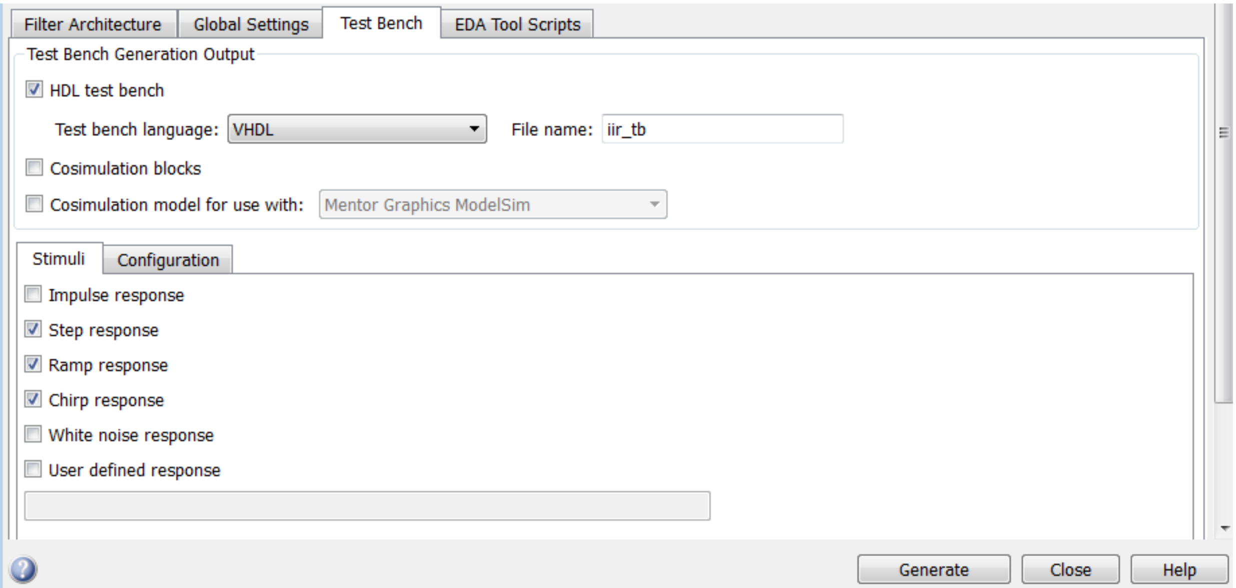


Additional settings

General | Ports | Advanced

- □ Represent constant values by aggregates
- ☑ Use "rising_edge/falling_edge" style for registers
- □ Loop unrolling
- □ Cast before sum
- ☑ Use Verilog `timescale directives
- ☑ Inline VHDL configuration
- ☑ Concatenate type safe zeros
- ☑ Emit time/date stamp in header

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



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

Test Bench Generation Output

- ☑ HDL test bench
- Test bench language: VHDL  File name: iir_tb
- □ Cosimulation blocks
- □ Cosimulation model for use with: Mentor Graphics ModelSim

Stimuli | Configuration

- □ Impulse response
- ☑ Step response
- ☑ Ramp response
- ☑ Chirp response
- □ White noise response
- □ User defined response

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

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

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

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

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

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

The generated VHDL code has the following characteristics:

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

| VHDL Port | Name |
|---|---|
| Input | `filter_in` |
| Output | `filter_out` |
| Clock input | `clk` |

| VHDL Port | Name |
|---|---|
| Clock enable input | `clk_enable` |
| Reset input | `reset` |

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

The generated test bench:

- Is a portable VHDL file.
- Forces clock, clock enable, and reset input signals.
- Forces the clock enable input signal to active high.
- Drives the clock input signal high (1) for 5 nanoseconds and low (0) for 5 nanoseconds.
- Forces the reset signal for two cycles plus a hold time of 2 nanoseconds.
- Applies a hold time of 2 nanoseconds to data input signals.
- 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 string you specified for the **Name** option in the **Target** pane. See step 2 in “Configure and Generate VHDL Code” on page 1-51.

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 1-51.

4 Search for `HDL Code`. This section lists coder options you modified in“Configure and Generate VHDL Code” on page 1-51.

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 FDATool” on page 1-45 and “Quantize the IIR Filter” on page 1-47.

6 Search for `ENTITY`. This line names the VHDL entity, using the string you specified for the **Name** option in the **Target** pane. See step 2 in “Configure and Generate VHDL Code” on page 1-51.

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

8 Search for `CONSTANT`. This 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 string `_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 1-51, 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 dialog box. See step 10 in “Configure and Generate VHDL Code” on page 1-51.

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.

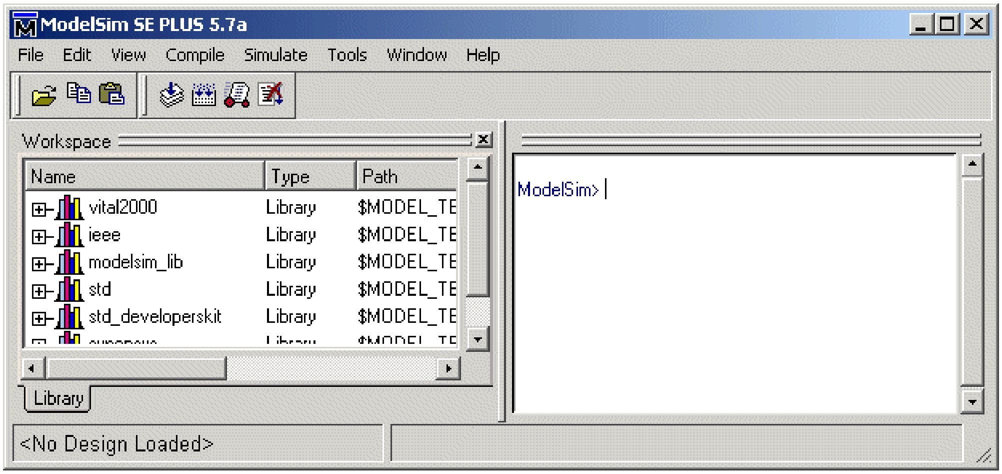
14 Search for `filter_out`. This code drive the filter output data.

## 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 the following steps:

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



| Name | Type | Path |
|---|---|---|
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |

```
ModelSim>
```

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

```
cd hdlsrc
```

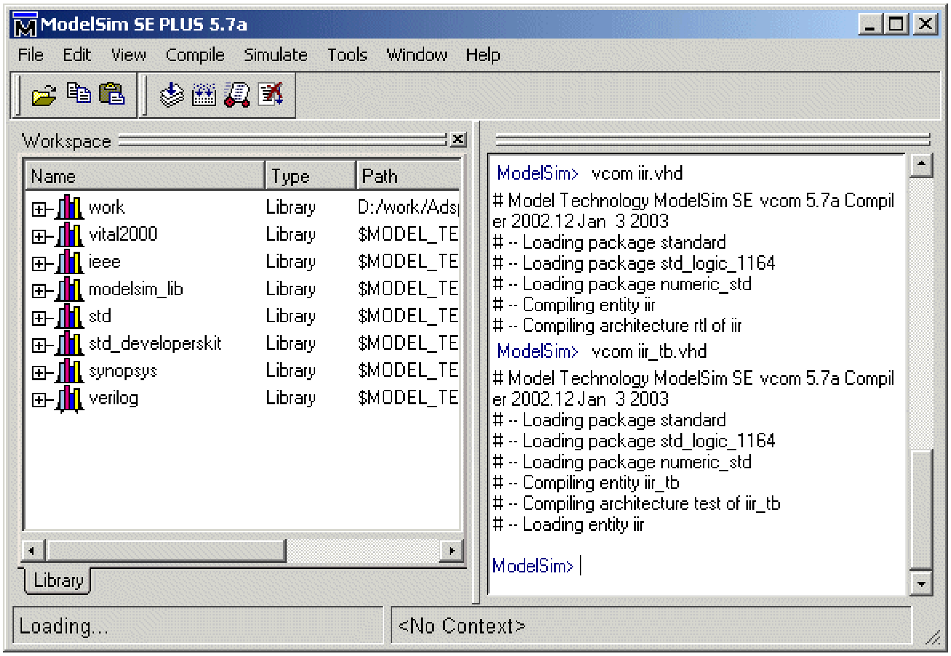
3 If desired, create a design library to store the compiled VHDL entities, packages, architectures, and configurations. In the Mentor Graphics ModelSim simulator, you can create a design library with the `vlib` command.

```
vlib work
```

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

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

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



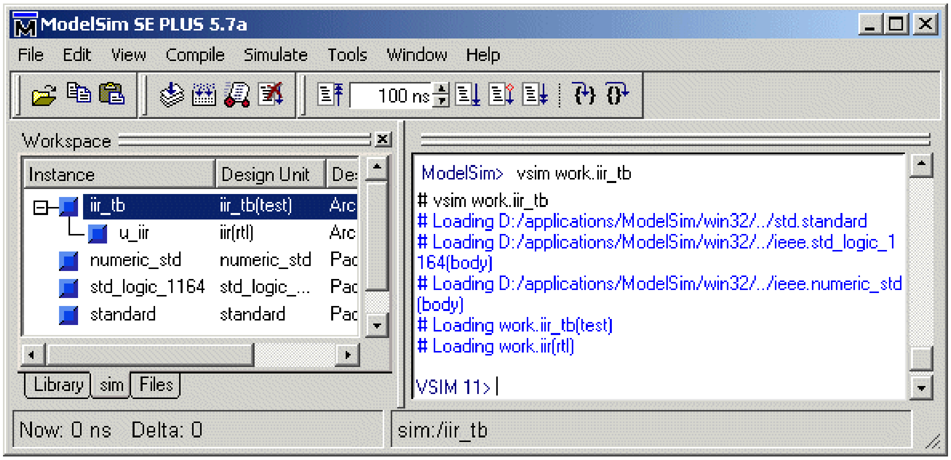
| Name | Type | Path |
|---|---|---|
| work | Library | D:/work/Ads |
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |
| synopsys | Library | $MODEL_TE |
| verilog | Library | $MODEL_TE |

```
ModelSim> vcom iir.vhd
# Model Technology ModelSim SE vcom 5.7a Compil
er 2002.12 Jan  3 2003
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity iir
# -- Compiling architecture rtl of iir
ModelSim> vcom iir_tb.vhd
# Model Technology ModelSim SE vcom 5.7a Compil
er 2002.12 Jan  3 2003
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity iir_tb
# -- Compiling architecture test of iir_tb
# -- Loading entity iir

ModelSim> |
```

**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
```

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



| Instance | Design Unit | De: |
|---|---|---|
| iir_tb | iir_tb(test) | Arc |
| u_iir | iir(rtl) | Arc |
| numeric_std | numeric_std | Pac |
| std_logic_1164 | std_logic_... | Pac |
| standard | standard | Pac |

```
ModelSim>  vsim work.iir_tb
# vsim work.iir_tb
# Loading D:/applications/ModelSim/win32/../std.standard
# Loading D:/applications/ModelSim/win32/../ieee.std_logic_1
164(body)
# Loading D:/applications/ModelSim/win32/../ieee.numeric_std
(body)
# Loading work.iir_tb(test)
# Loading work.iir(rtl)

VSIM 11> |
```

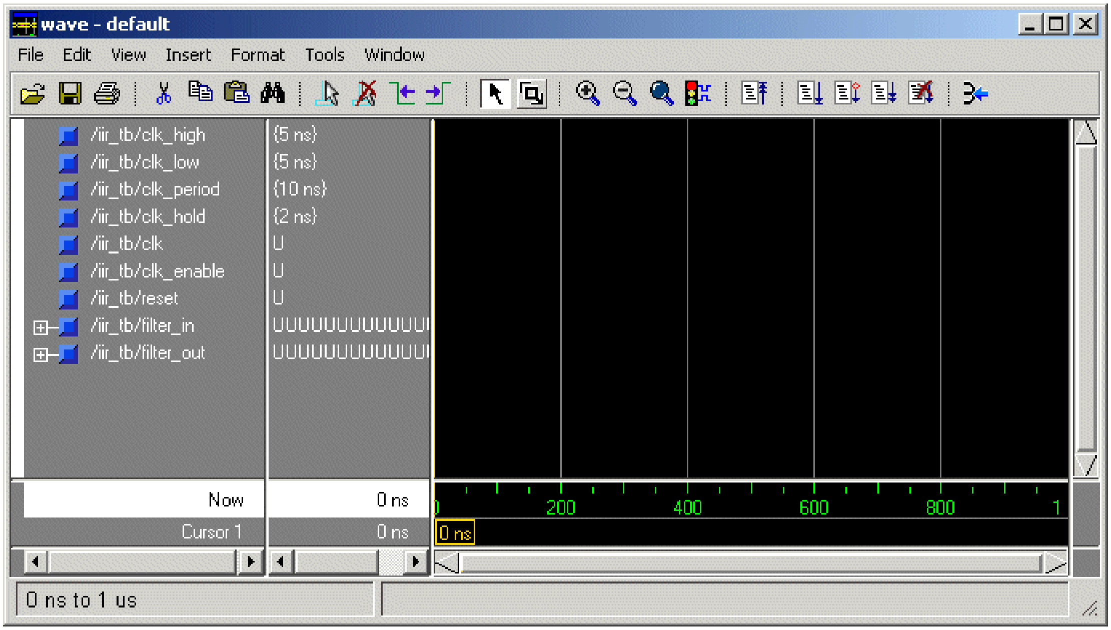
Now: 0 ns   Delta: 0

sim:/iir_tb

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

```
add wave *
```

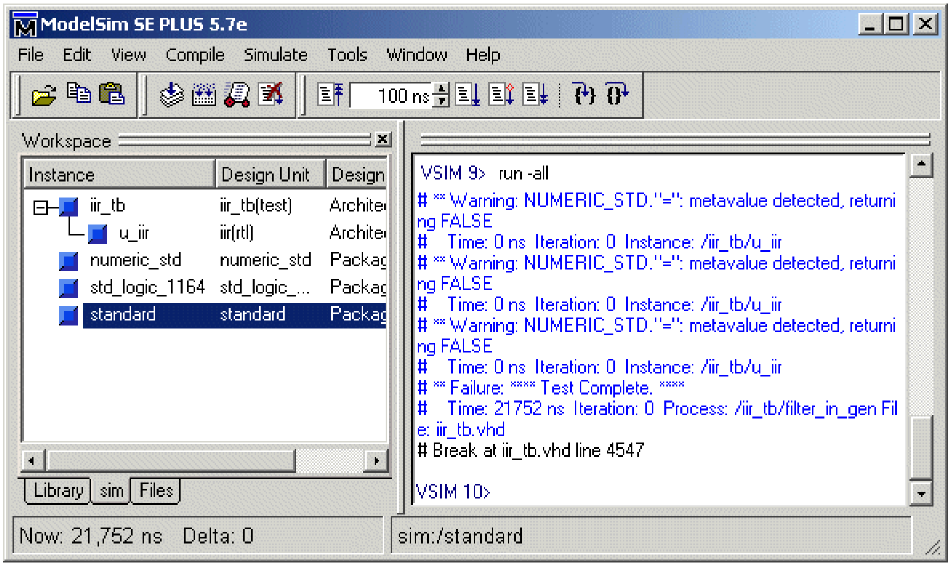
The following **wave** window displays.



```
/iir_tb/clk_high     {5 ns}
/iir_tb/clk_low      {5 ns}
/iir_tb/clk_period   {10 ns}
/iir_tb/clk_hold     {2 ns}
/iir_tb/clk          U
/iir_tb/clk_enable   U
/iir_tb/reset        U
/iir_tb/filter_in    UUUUUUUUUUUUU
/iir_tb/filter_out   UUUUUUUUUUUUU
Now                  0 ns
Cursor 1             0 ns    0 ns
0 ns to 1 us
```

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

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



| Instance | Design Unit | Design |
|---|---|---|
| iir_tb | iir_tb(test) | Archite |
| u_iir | iir(rtl) | Archite |
| numeric_std | numeric_std | Packag |
| std_logic_1164 | std_logic_... | Packag |
| standard | standard | Packag |

```
VSIM 9> run -all
# ** Warning: NUMERIC_STD."=": metavalue detected, returni
ng FALSE
#    Time: 0 ns  Iteration: 0  Instance: /iir_tb/u_iir
# ** Warning: NUMERIC_STD."=": metavalue detected, returni
ng FALSE
#    Time: 0 ns  Iteration: 0  Instance: /iir_tb/u_iir
# ** Warning: NUMERIC_STD."=": metavalue detected, returni
ng FALSE
#    Time: 0 ns  Iteration: 0  Instance: /iir_tb/u_iir
# ** Failure: **** Test Complete. ****
#    Time: 21752 ns  Iteration: 0  Process: /iir_tb/filter_in_gen Fil
e: iir_tb.vhd
# Break at iir_tb.vhd line 4547

VSIM 10>
```

Now: 21,752 ns   Delta: 0

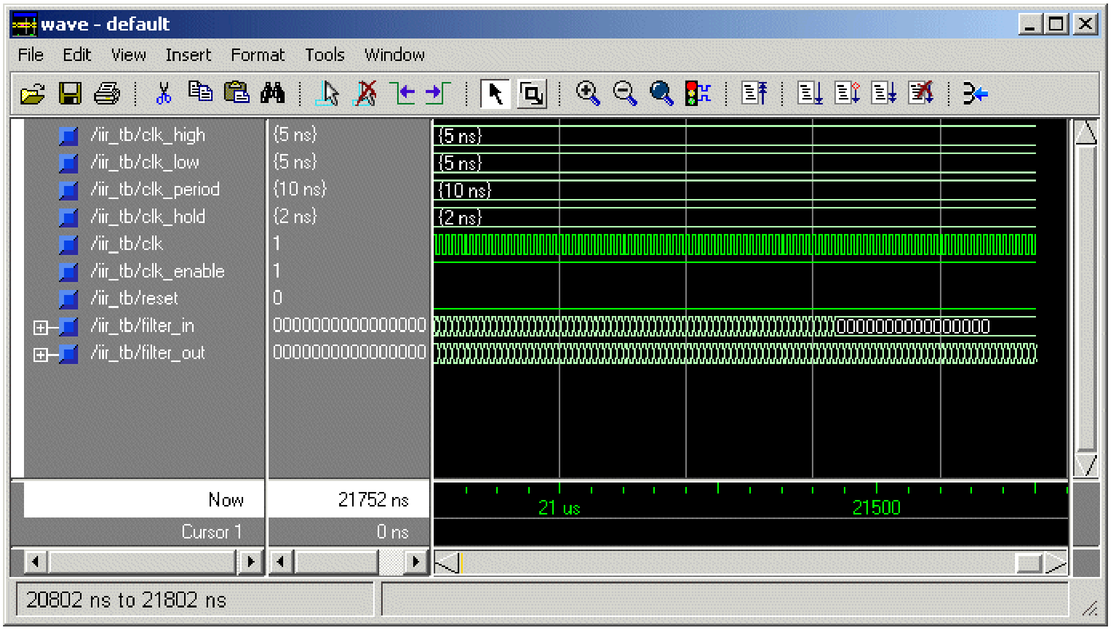
sim:/standard

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 string `Test Complete`, the test bench has run to completion without encountering an error. The `Failure` part of the message is tied to the mechanism that the coder uses to end the simulation.

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



```
/iir_tb/clk_high      {5 ns}
/iir_tb/clk_low       {5 ns}
/iir_tb/clk_period    {10 ns}
/iir_tb/clk_hold      {2 ns}
/iir_tb/clk           1
/iir_tb/clk_enable    1
/iir_tb/reset         0
/iir_tb/filter_in     0000000000000000
/iir_tb/filter_out    0000000000000000
Now                   21752 ns
Cursor 1              0 ns
20802 ns to 21802 ns
```

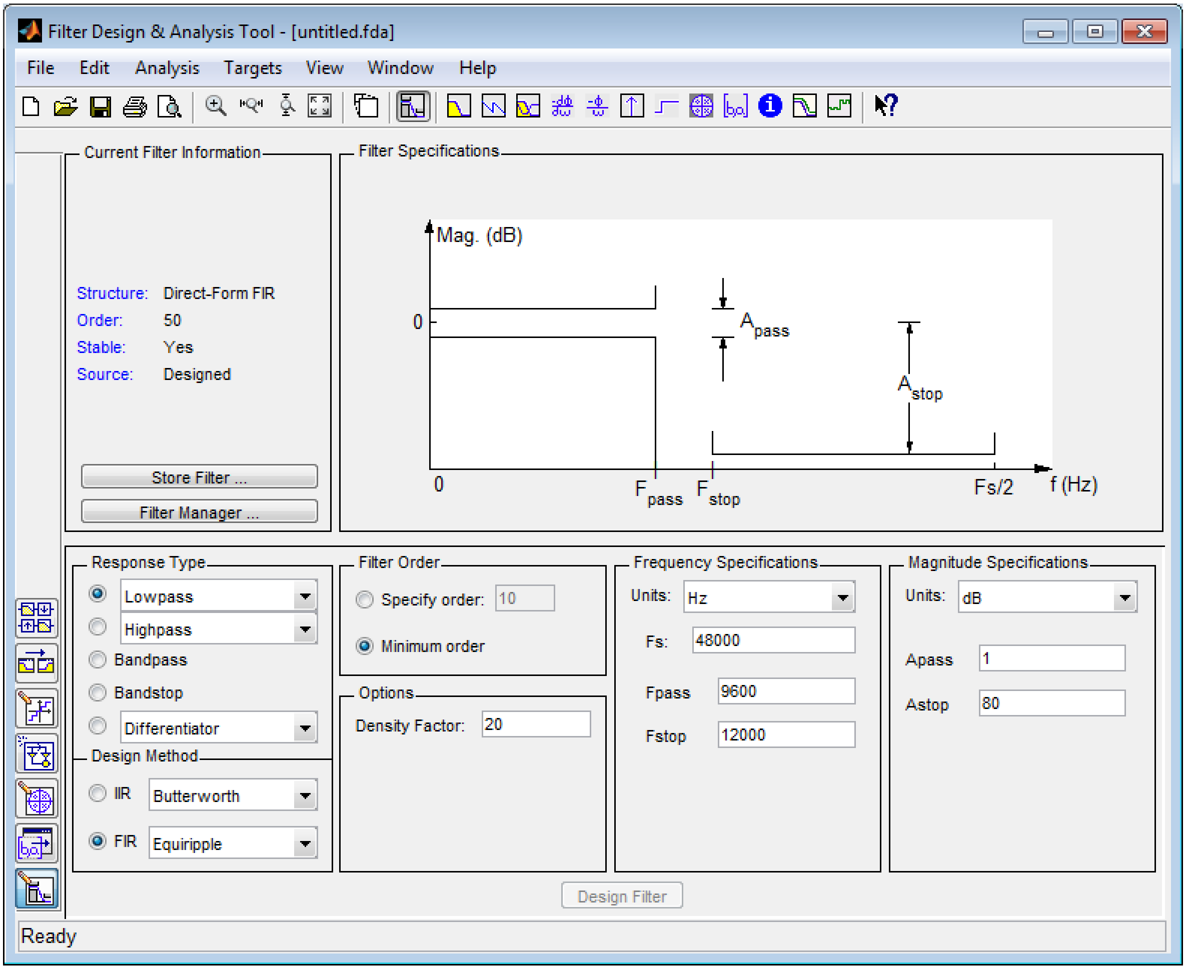
# HDL Filter Code Generation Fundamentals

# Starting Filter Design HDL Coder

## Opening the Filter Design HDL Coder GUI from FDATool

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

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



Current Filter Information

Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response Type

● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method

○ IIR Butterworth
● FIR Equiripple

Filter Order

○ Specify order: 10
● Minimum order

Options

Density Factor: 20

Frequency Specifications

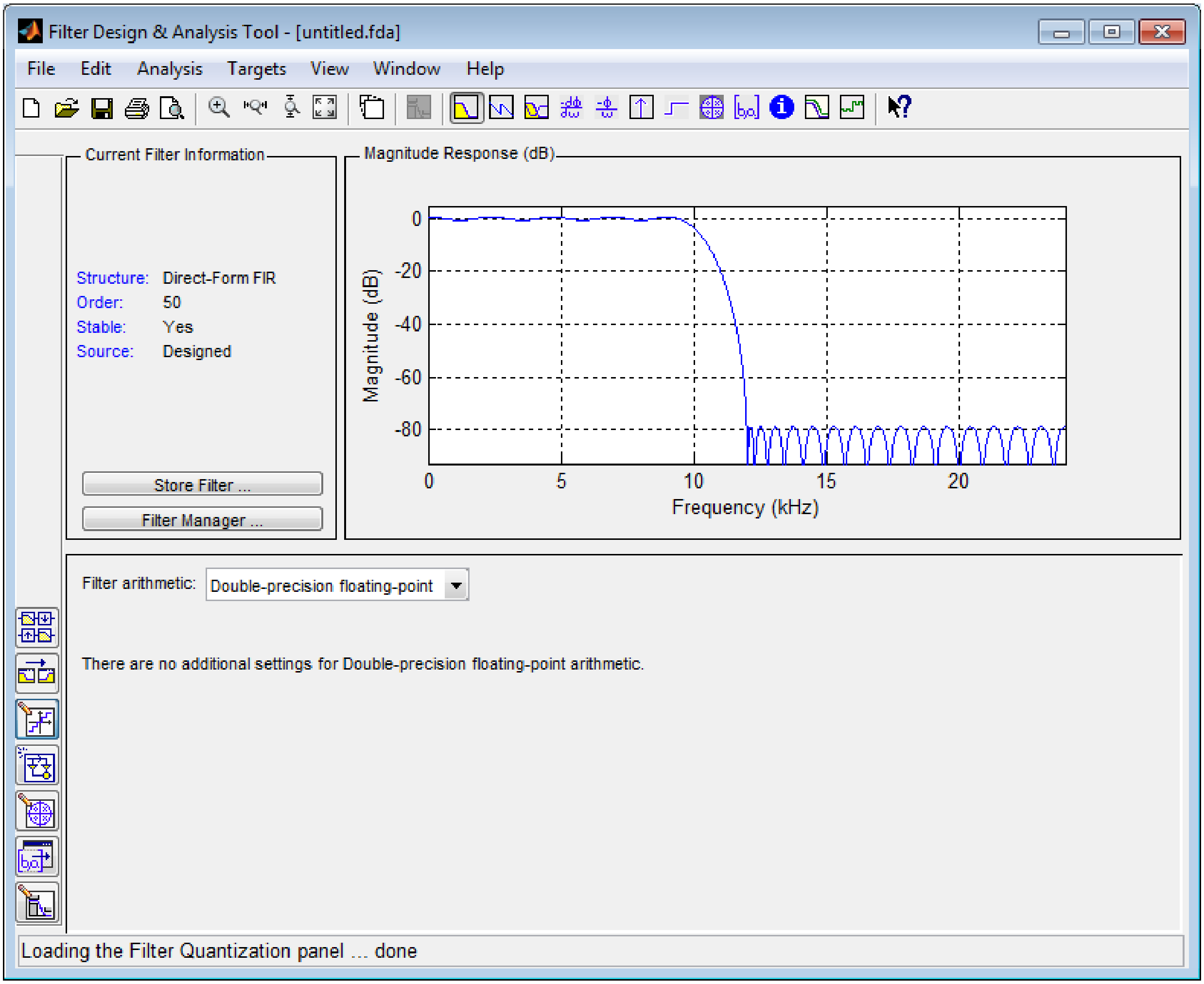
Units: Hz
Fs: 48000
Fpass: 9600
Fstop: 12000

Magnitude Specifications

Units: dB
Apass: 1
Astop: 80

Ready

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



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

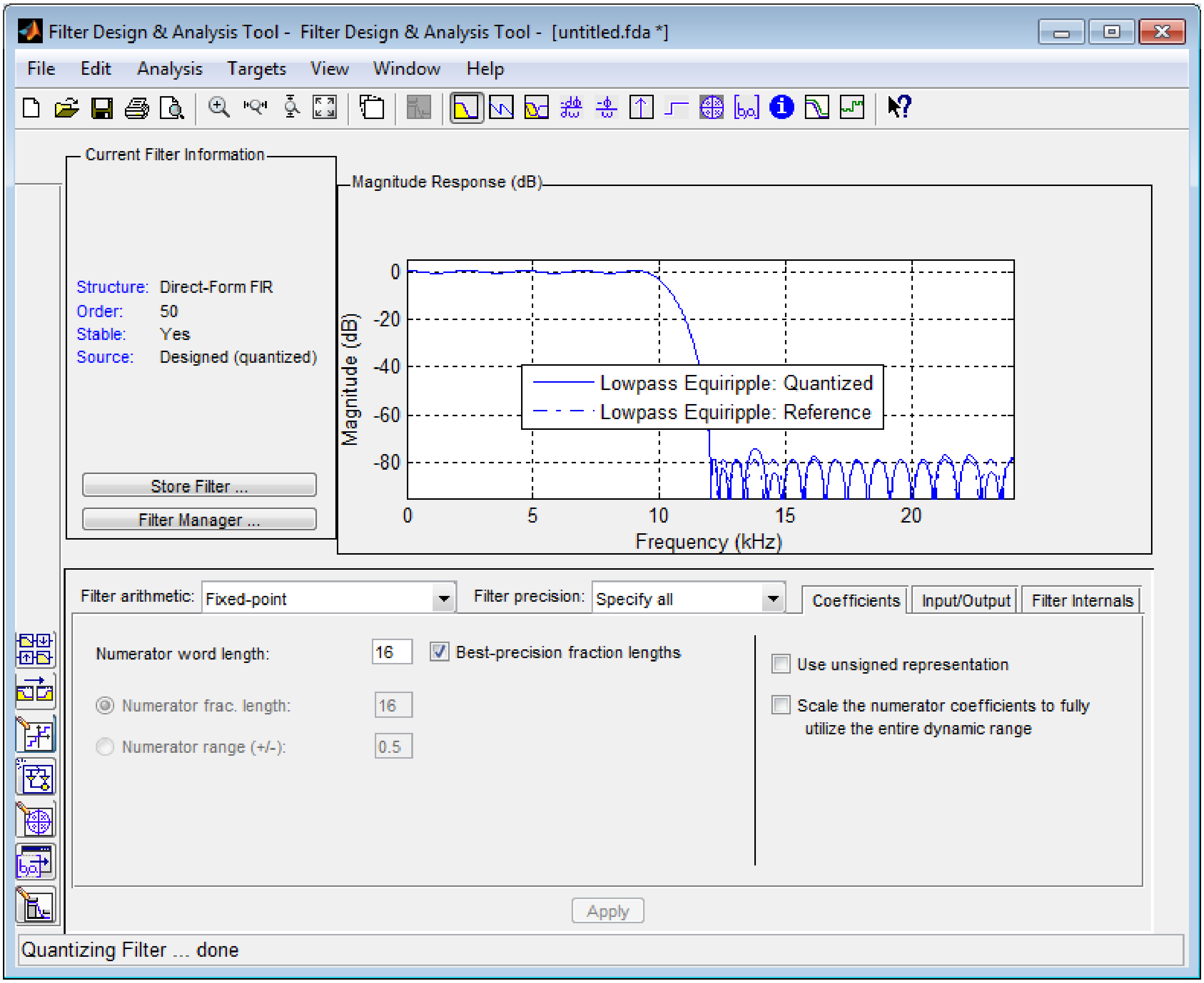
Filter arithmetic: Double-precision floating-point

There are no additional settings for Double-precision floating-point arithmetic.

Loading the Filter Quantization panel ... done

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

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



Structure: Direct-Form FIR

Order: 50

Stable: Yes

Source: Designed (quantized)

Filter arithmetic: Fixed-point

Filter precision: Specify all

Numerator word length: 16

☑ Best-precision fraction lengths

● Numerator frac. length: 16

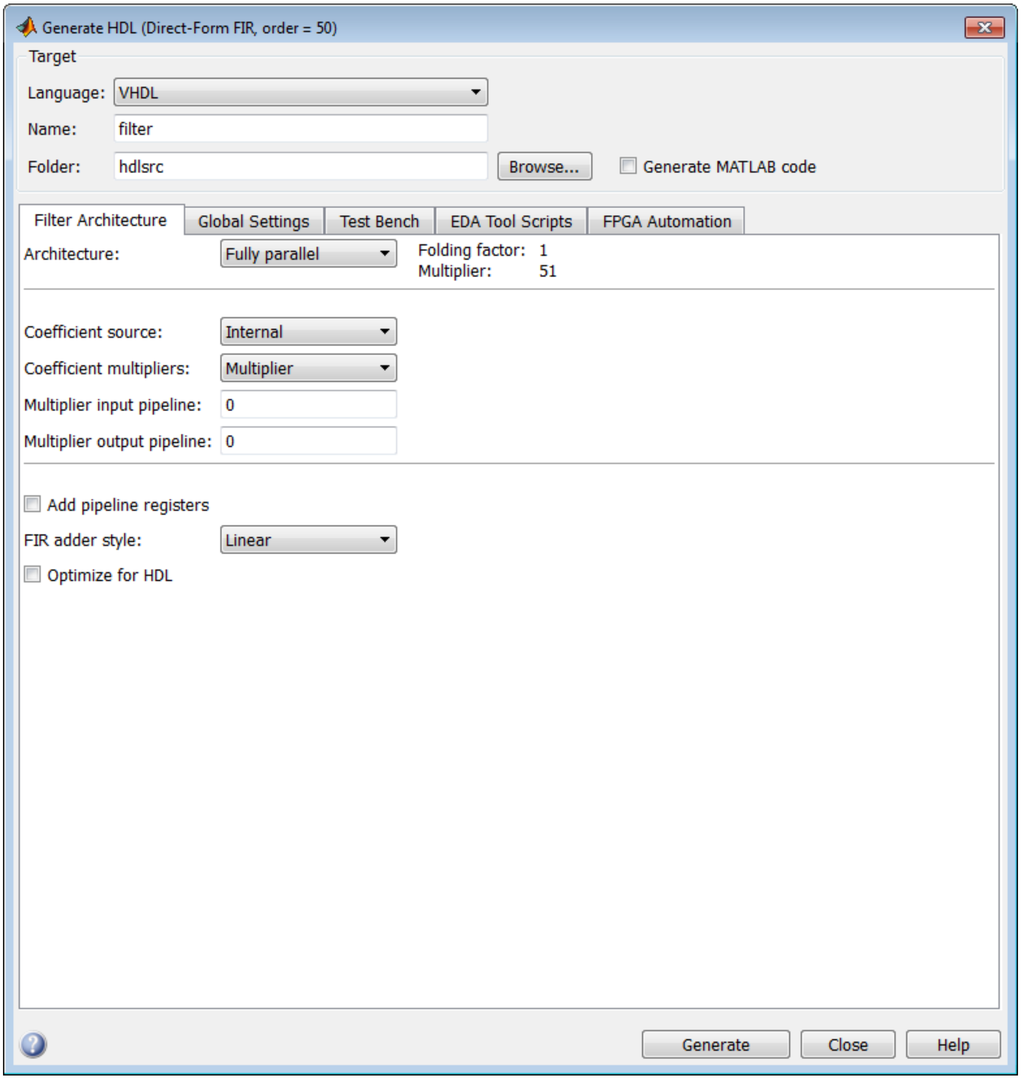
○ Numerator range (+/-): 0.5

□ Use unsigned representation

□ Scale the numerator coefficients to fully utilize the entire dynamic range

Quantizing Filter ... done

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



Language: VHDL

Name: filter

Folder: hdlsrc Browse...

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 51

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

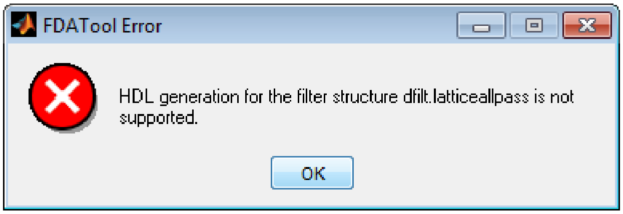
Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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



HDL generation for the filter structure dfilt.latticeallpass is not supported.

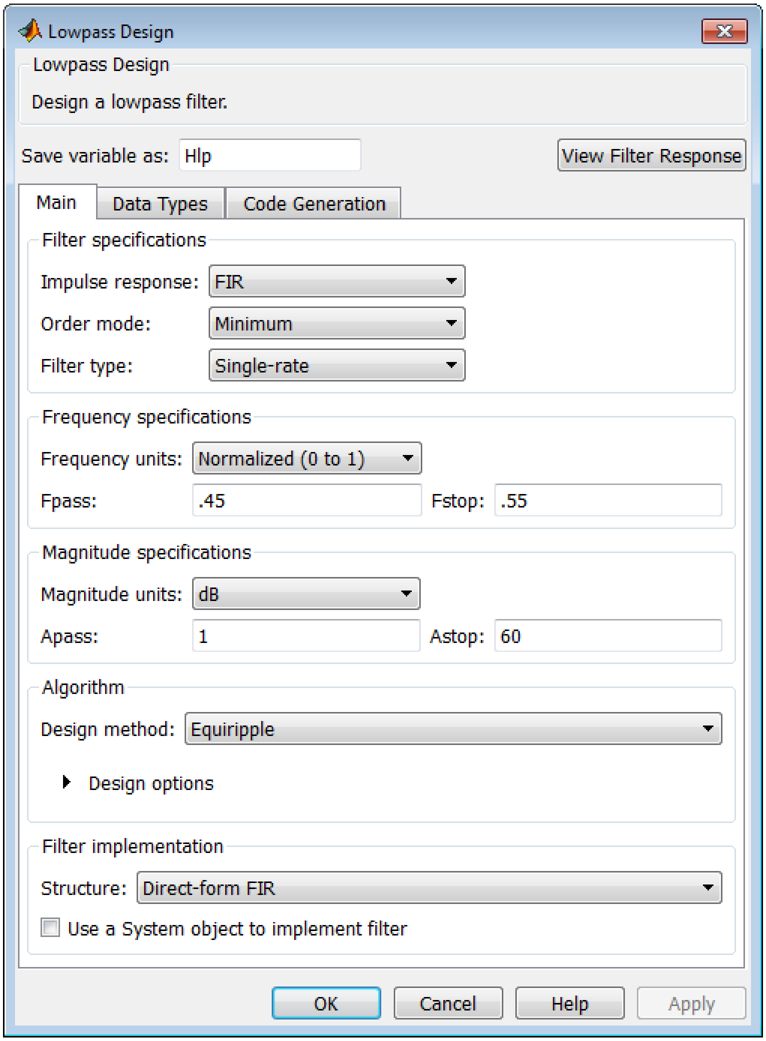
## Opening the Filter Design HDL Coder GUI from the `filterbuilder` GUI

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

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

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

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



Lowpass Design

Lowpass Design

Design a lowpass filter.

Save variable as: Hlp

Filter specifications

Impulse response: FIR

Order mode: Minimum

Filter type: Single-rate

Frequency specifications

Frequency units: Normalized (0 to 1)

Fpass: .45 Fstop: .55

Magnitude specifications

Magnitude units: dB

Apass: 1 Astop: 60

Algorithm

Design method: Equiripple

Filter implementation

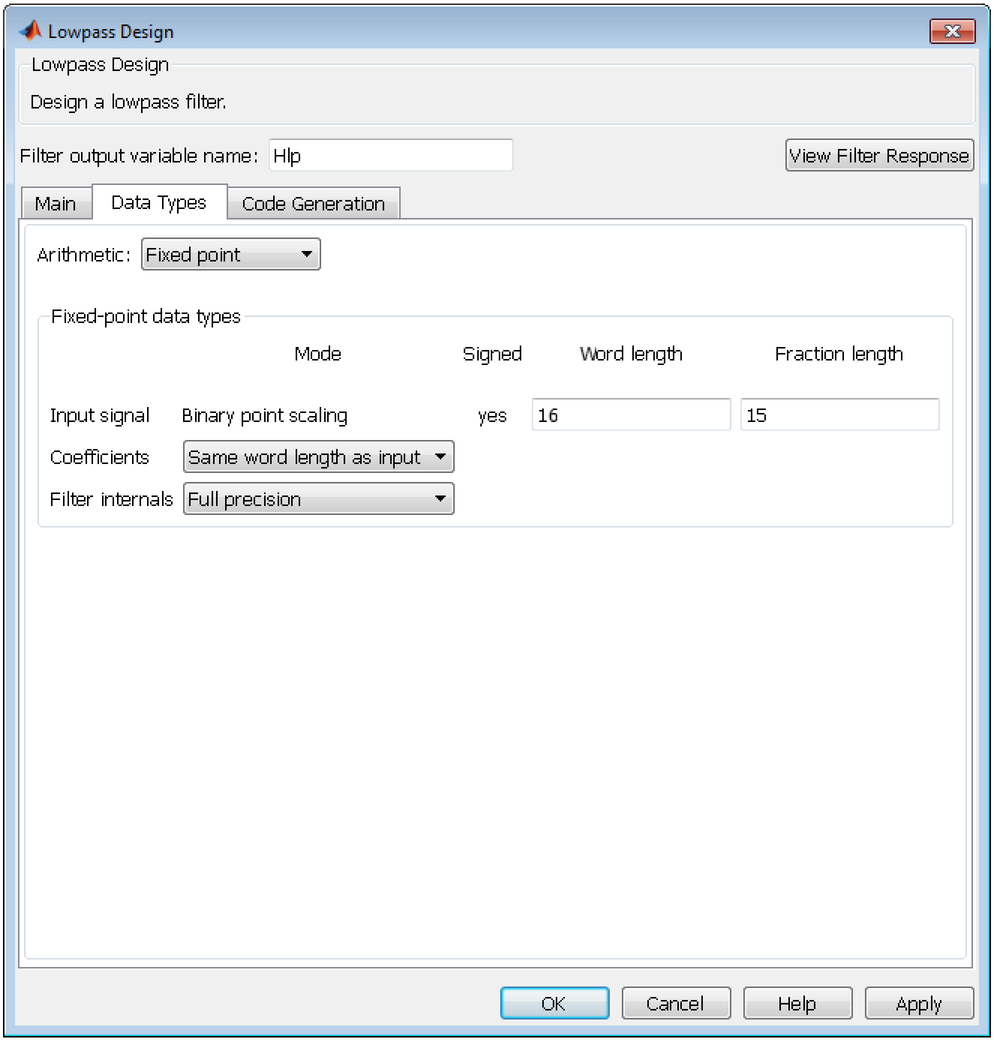
Structure: Direct-form FIR

□ Use a System object to implement filter

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.

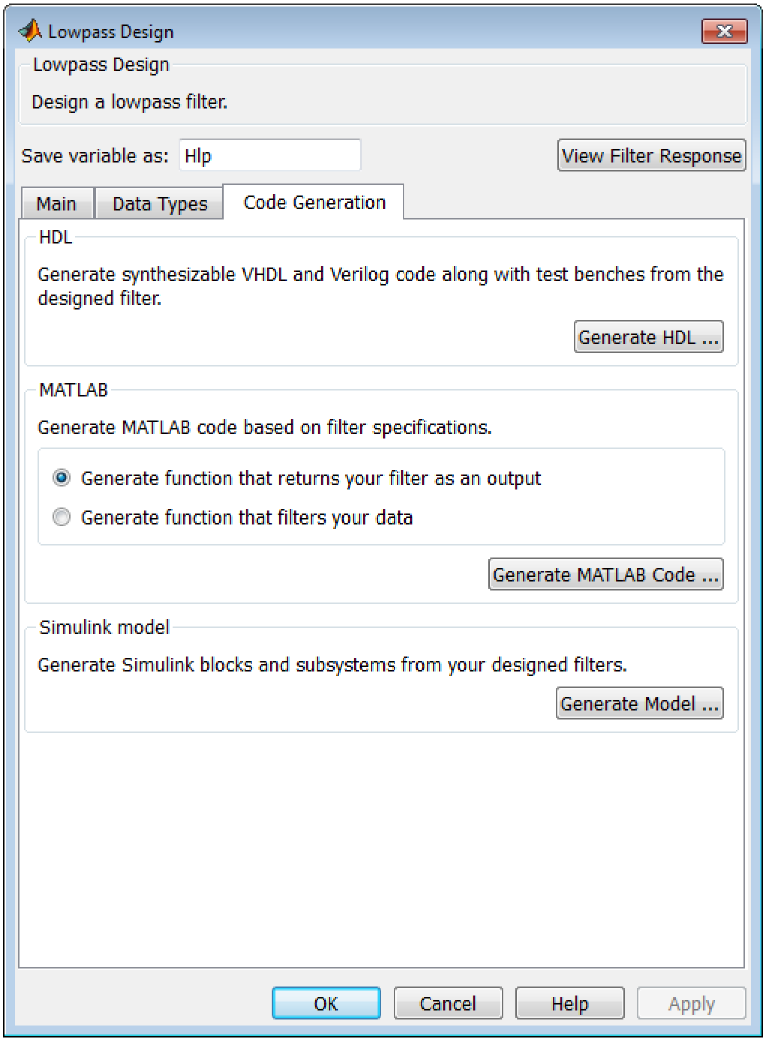


Filter output variable name: Hlp

Arithmetic: Fixed point

|  | Mode | Signed | Word length | Fraction length |
|---|---|---|---|---|
| Input signal | Binary point scaling | yes | 16 | 15 |
| Coefficients | Same word length as input |  |  |  |
| Filter internals | Full precision |  |  |  |

5 Click the **Code Generation** tab.



Design a lowpass filter.

Save variable as: Hlp

Generate synthesizable VHDL and Verilog code along with test benches from the designed filter.

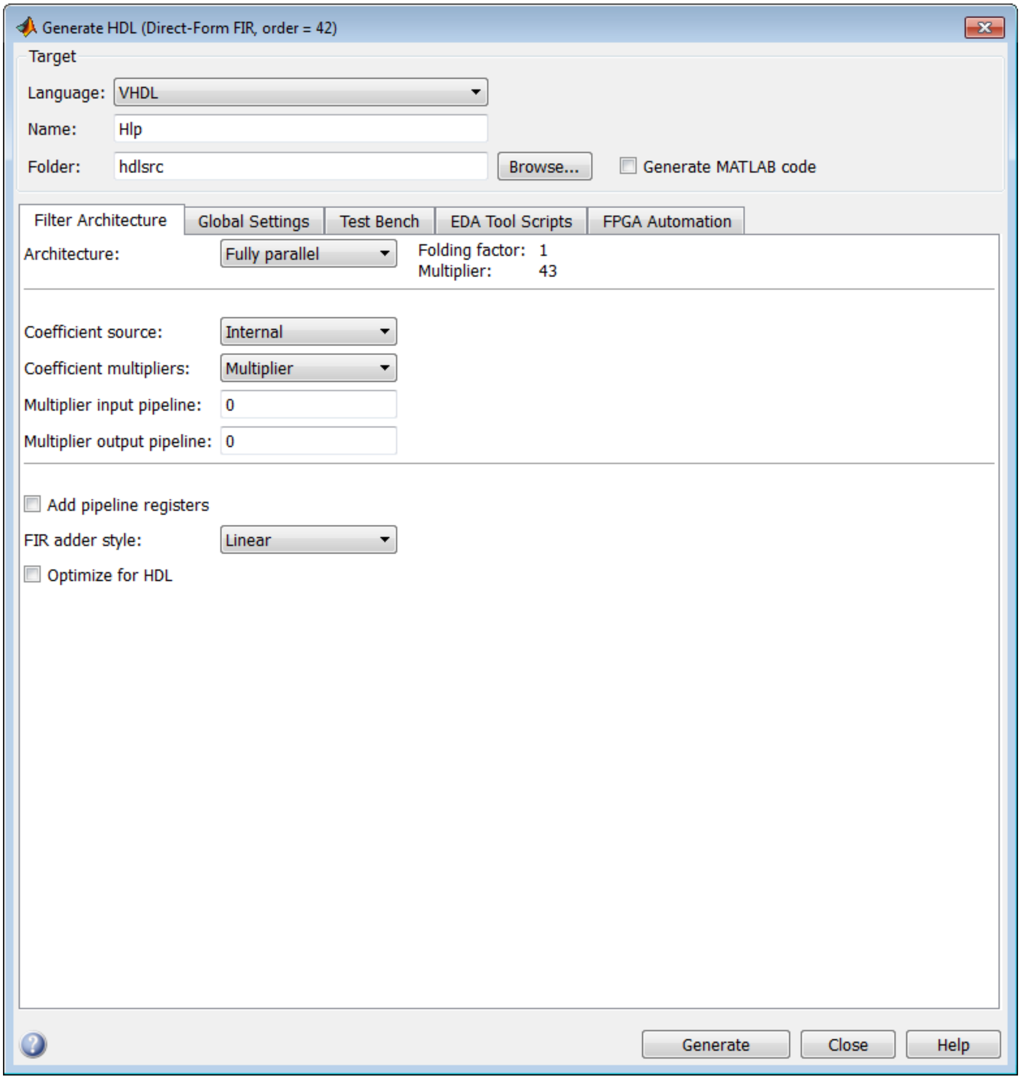
Generate MATLAB code based on filter specifications.

● Generate function that returns your filter as an output

○ Generate function that filters your data

Generate Simulink blocks and subsystems from your designed filters.

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



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

Target

Language: VHDL

Name: Hlp

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 43

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

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

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

```
fdhdltool(Hd)
```

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

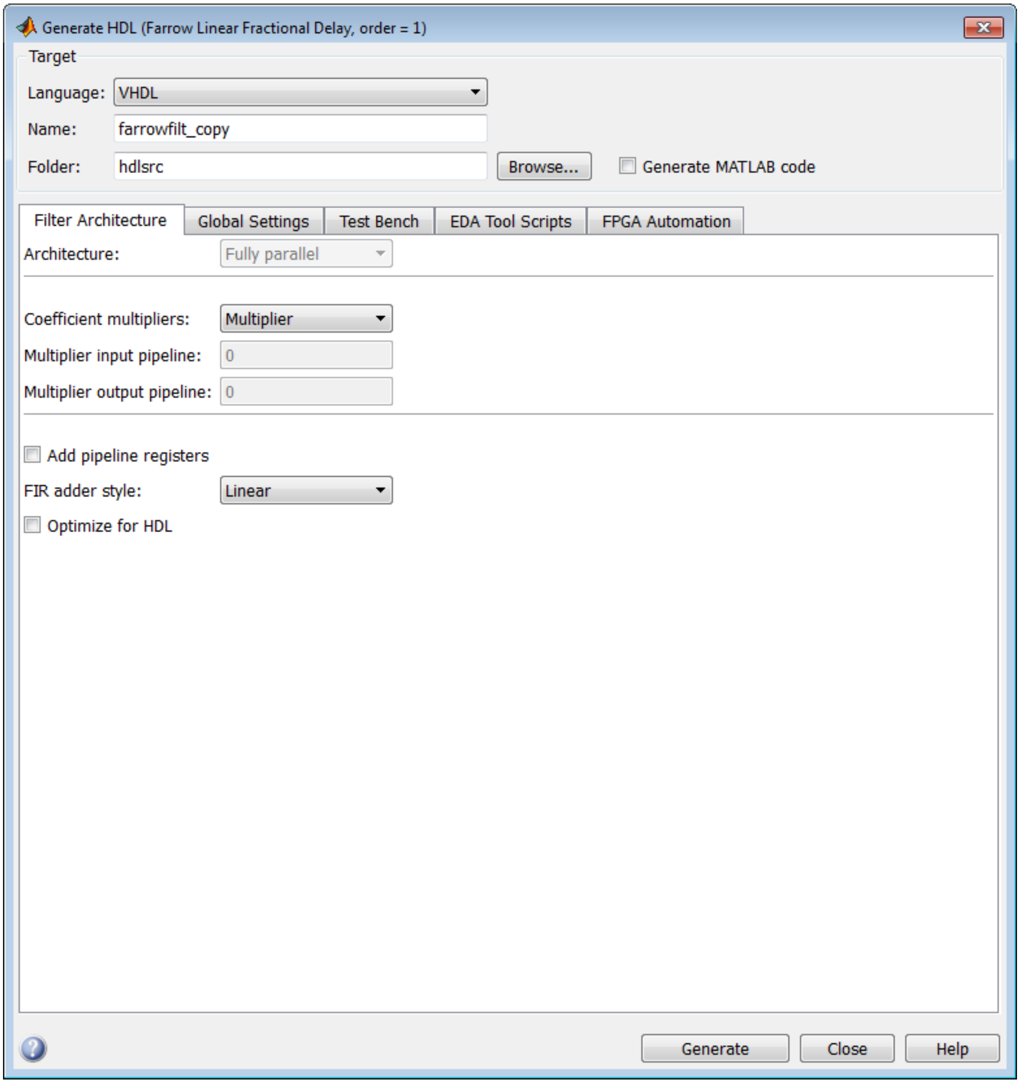
```
fdhdltool(FIRLowpass,numerictype(1,16,15))
```

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

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

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

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



Generate HDL (Farrow Linear Fractional Delay, order = 1)

Target

Language: VHDL

Name: farrowfilt_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

# Selecting Target Language

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

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

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

# Generating HDL Code

**In this section...**

## Applying Your Settings

When you generate HDL, either from the GUI 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 2-16. **Generate MATLAB code** is available only in the GUI. The function `generatehdl` does not have an equivalent property.

## Generating HDL Code from the GUI

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

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

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

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

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

## Generating HDL Code Using `generatehdl`

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

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60)
Hd = design(d,'equiripple','filterstructure','dfsymfir','Systemobject',true)
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'Name','MyFilter',...
               'TargetLanguage','Verilog','GenerateHDLTestbench', 'on')
```

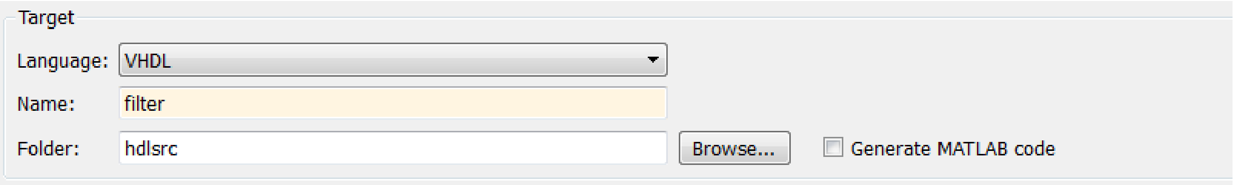
As code generation proceeds, a sequence of messages similar to the following appears in the MATLAB Command Window:

```
#### Starting Verilog code generation process for filter: MyFilter
### Generating: H:\hdlsrc\MyFilter.v
### 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: H:\hdlsrc\MyFilter_tb.v
### Creating stimulus vectors ...
### Done generating VERILOG 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.

# Capturing Code Generation Settings

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



Target

Language: VHDL

Name: filter

Folder: hdlsrc

□ Generate MATLAB code

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 GUI 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 GUI. The function `generatehdl` does not have an equivalent property.

## More About

- "Generating HDL Code" on page 2-14

# 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 2-16.

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

## More About

- “Starting Filter Design HDL Coder” on page 2-2
- “Generating HDL Code from the GUI” on page 2-14

# HDL Code for Supported Filter Structures

# Generate HDL from Filter System Objects

You can generate HDL code from the following System objects:

### Single Rate Filters

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

### Multirate Filters

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

After you design a filter System object, use `generatehdl` or `fdhdltool` to set HDL code generation options and generate code. When you input a System object into either function, you must also specify the input data type for the object. The HDL code generation tool quantizes the input signal to this data type. The data type argument is an object of `numerictype` class. 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.

For instance, if `Hd` is a `dsp.BiquadFilter` object, as in the “HDL Butterworth Filter” example, call `generatehdl` to generate HDL code for the filter.

```
generatehdl(Hd,'Name','hdlbutter',...
                'TargetLanguage','VHDL',...
```

```
                  'TargetDirectory',workingdir,...
                  'GenerateHDLTestbench','on',...
                  'TestBenchUserStimulus',userstim,...
                  'InputDataType',numerictype(1,8,7));
```

The call to `numerictype(1,8,7)` specifies a signed 8-bit number with 7 fractional bits.

Alternatively, call `fdhdltool` to open a GUI which enables you to set code generation options and generate HDL code.

```
fdhdltool(Hd,numerictype(1,8,7));
```

You can also create a filter System object and generate HDL code from it, by calling `filterbuilder` and then setting 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.

Functions for exploring filter architectures, and generating test bench stimulus also take an input data type argument when you call them with a System object. See `hdlfilterserialinfo`, `hdlfilterdainfo`, and `generatetbstimulus`.

## See Also

`fdhdltool` | `generatehdl` | `numerictype`

## Related Examples

- "HDL Inverse Sinc Filter"
- "HDL Tone Control Filter Bank"
- "HDL Sample Rate Conversion Using Farrow Filters"

# Multirate Filters

## Supported Multirate Filter Types

HDL code generation is supported for the following 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 FDATool, `filterbuilder`, or the MATLAB command line.

After you have created the filter, open the Generate HDL dialog box, set the desired code generation properties, and generate code. See “Code Generation Options for Multirate Filters” on page 3-4.

To generate code using the `generatehdl` function, specify multirate filter code generation properties that are functionally equivalent to the GUI options. See “`generatehdl` Properties for Multirate Filters” on page 3-8.

## Code Generation Options for Multirate Filters

When a multirate filter of a supported type (see “Supported Multirate Filter Types” on page 3-4) is designed, the enabled/disabled state of several options in the Generate HDL dialog box changes.

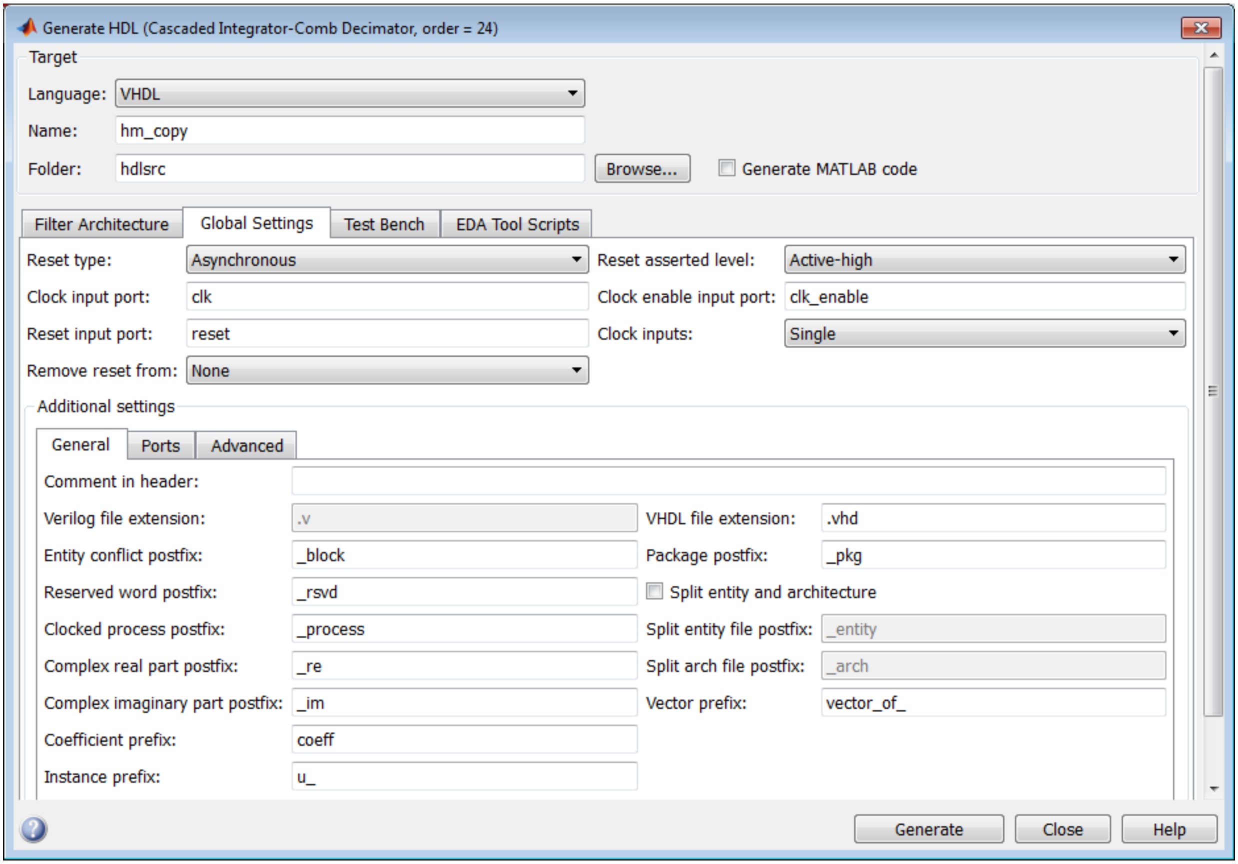
- On the **Global settings** tab, the **Clock inputs** pull-down menu is enabled. This menu provides two alternatives for generating clock inputs for multirate filters.

**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.

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



Generate HDL (Cascaded Integrator-Comb Decimator, order = 24)

Target

Language: VHDL

Name: hm_copy

Folder: hdlsrc

☐ Generate MATLAB code

Reset type: Asynchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

Additional settings

Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

☐ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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 3-8. 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.

  The following code excerpts were 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';
```

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

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

- `Multiple`: When 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.

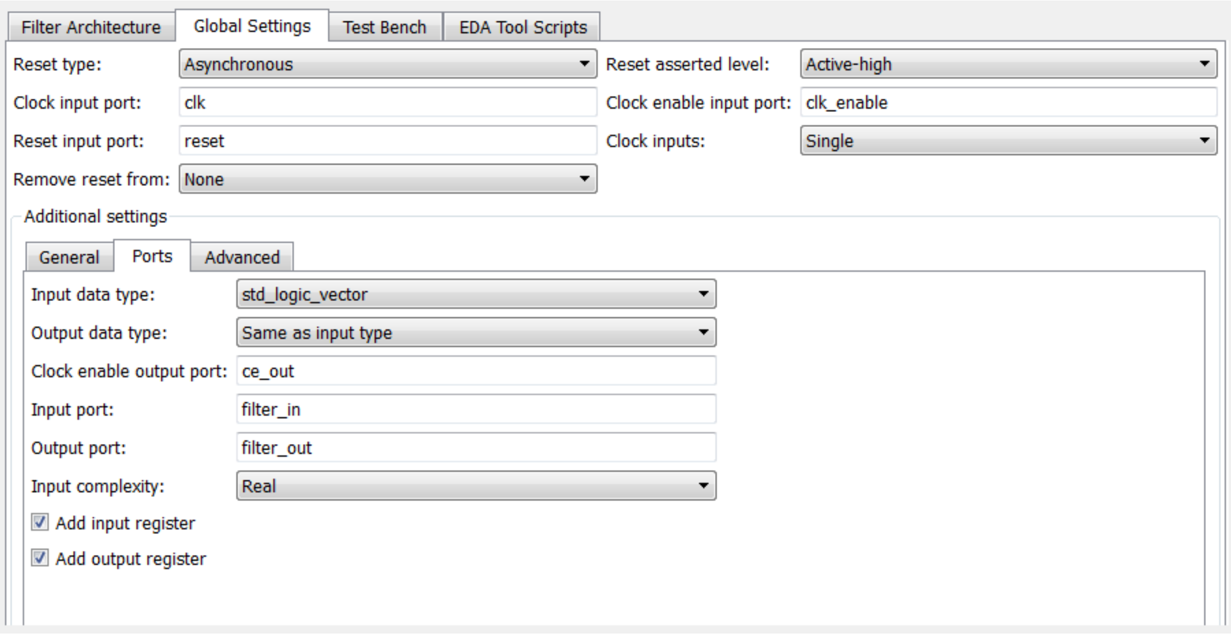
  The following`ENTITY` declaration was generated from a CIC decimation filter with **Clock inputs** set to `Multiple`.

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

END cic_decim_4_1_multi;
```

### Setting the Clock Enable Output Name

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

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



Reset type: Asynchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Remove reset from: None

Additional settings

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

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 the following properties to specify clock generation options:

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

- ClockEnableInputPort corresponds to the **Clock enable input port** field; specifies the name of the clock enable input port.

# Variable Rate CIC Filters

**In this section...**

## Supported Variable Rate CIC Filter Types

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

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

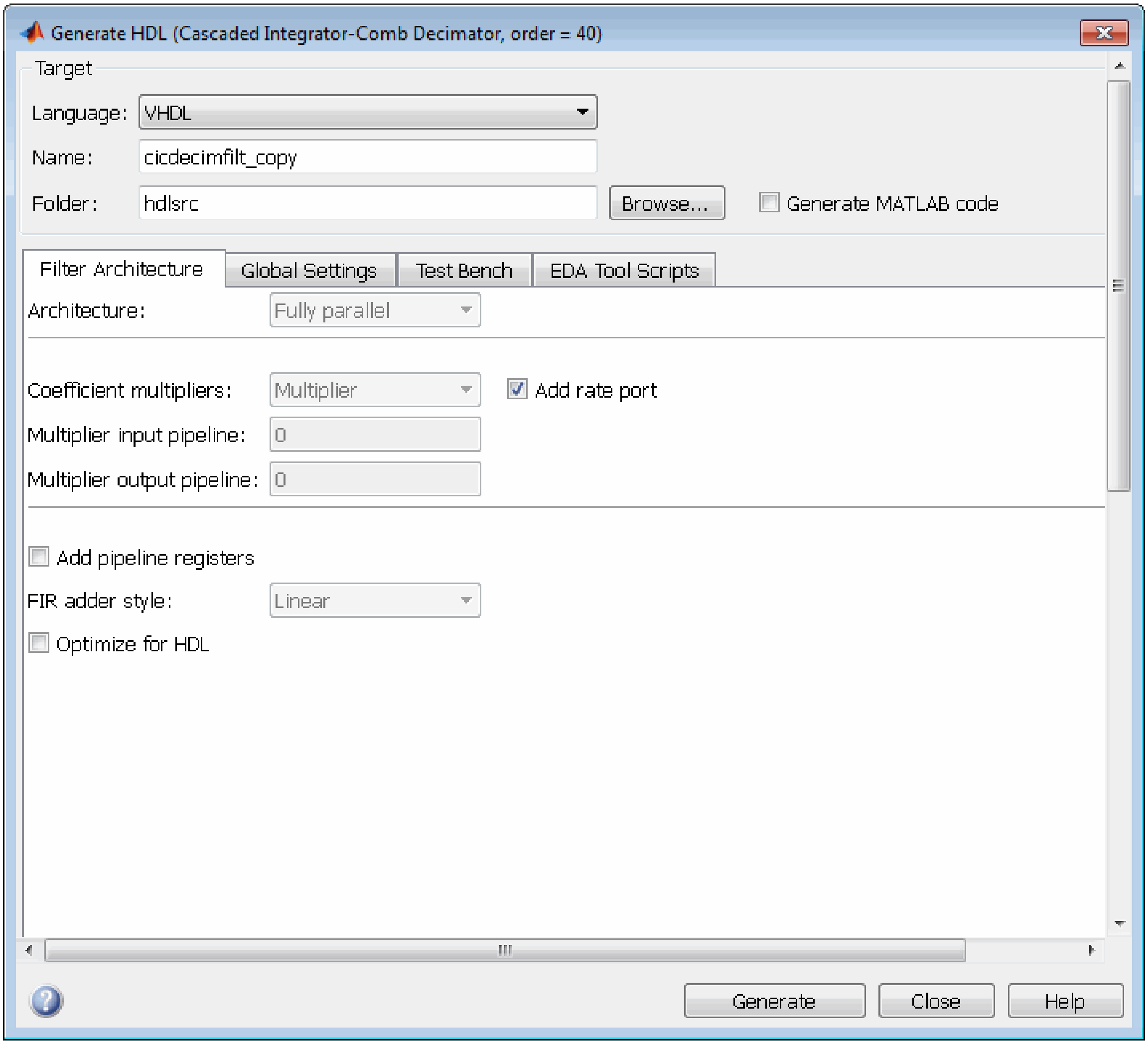
## Code Generation Options for Variable Rate CIC Filters

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 GUI 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

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient multipliers: Multiplier

☑ Add rate port

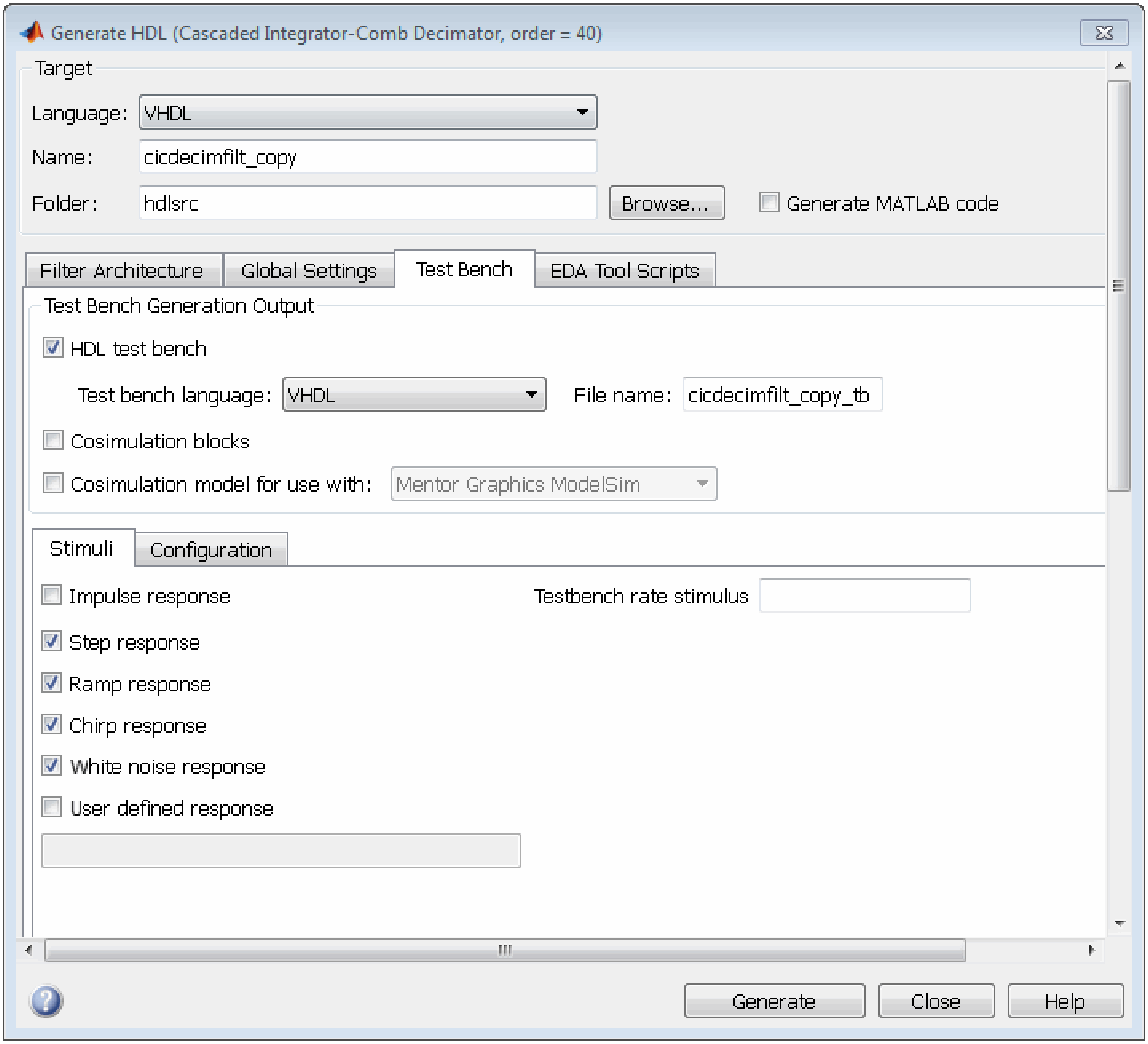
Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL



Generate HDL (Cascaded Integrator-Comb Decimator, order = 40)

Target

Language: VHDL

Name: cicdecimfilt_copy

Folder: hdlsrc

☐ Generate MATLAB code

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

☐ Impulse response

Testbench rate stimulus

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

☐ User defined response

# 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 3-14 before creating your filter stages and cascade filter object.

### Generating Cascade Filter Code with the `fdhdltool` Function

Call `fdhdltool` to open the Generate HDL dialog box, 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

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

- You can generate code for cascades that combine the following filter types:

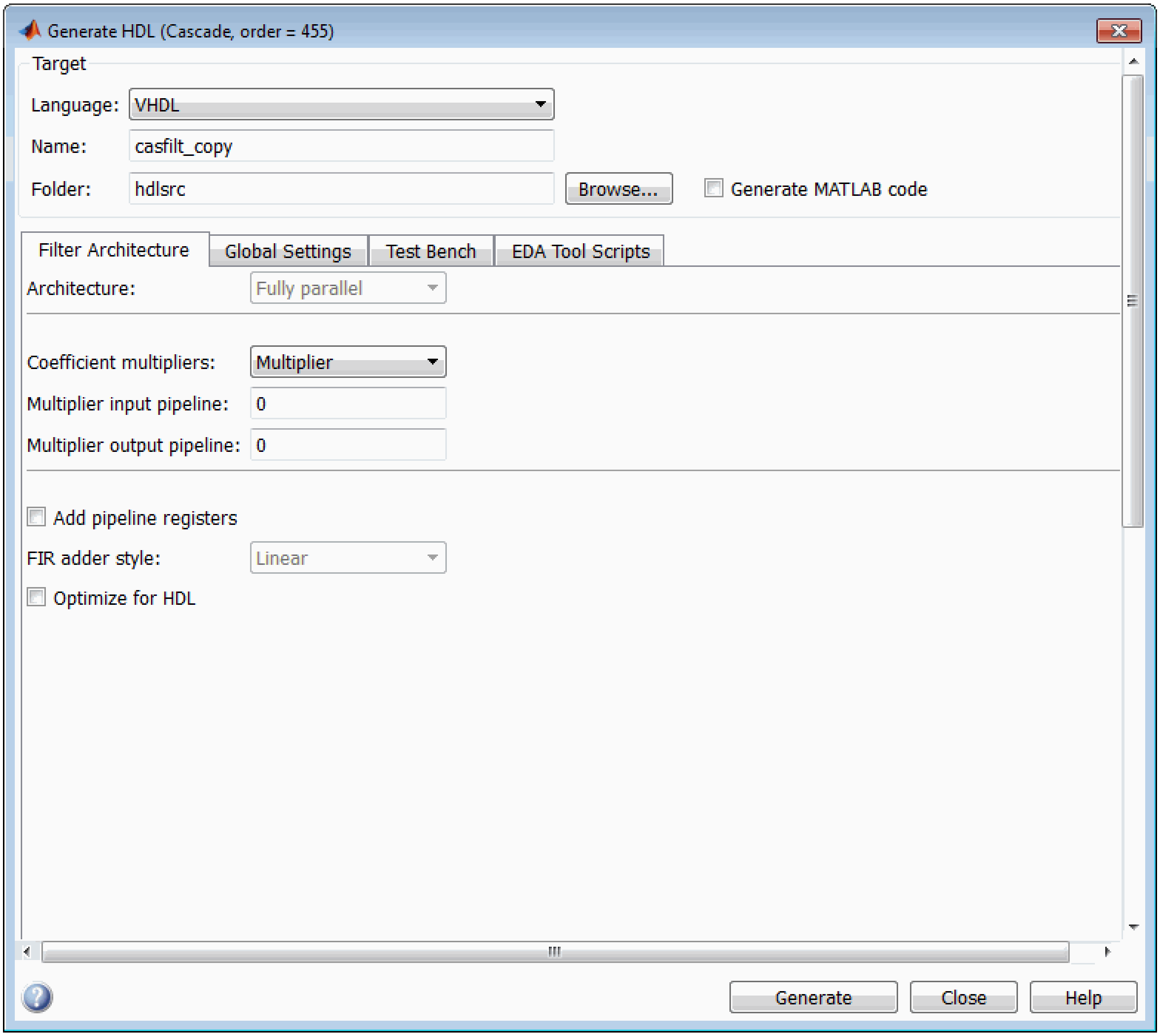
  - Decimators and/or single-rate filter structures
  - Interpolators and/or single-rate filter structures

  Code generation for cascades that include both decimators and interpolators is not 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 dialog box. 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 string `_stage1`, `_stage2`, ... `_stage`*`N`* to the filter name.

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



Generate HDL (Cascade, order = 455)

Target

Language: VHDL

Name: casfilt_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

# Polyphase Sample Rate Converters

| In this section... |
|---|
| "Code Generation for Polyphase Sample Rate Converter" on page 3-16 |
| "HDL Implementation for Polyphase Sample Rate Converter" on page 3-16 |

## 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

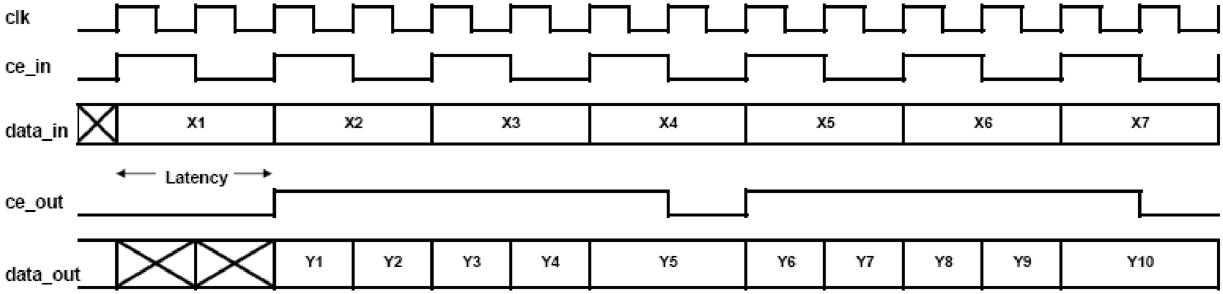
### Signal Flow, Latency, and Timing

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.

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



### Clock Rate

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

```
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 dialog box 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 dialog box 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

The following 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
```

The following 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 3-23), 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.

The following 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')
```

The following 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.

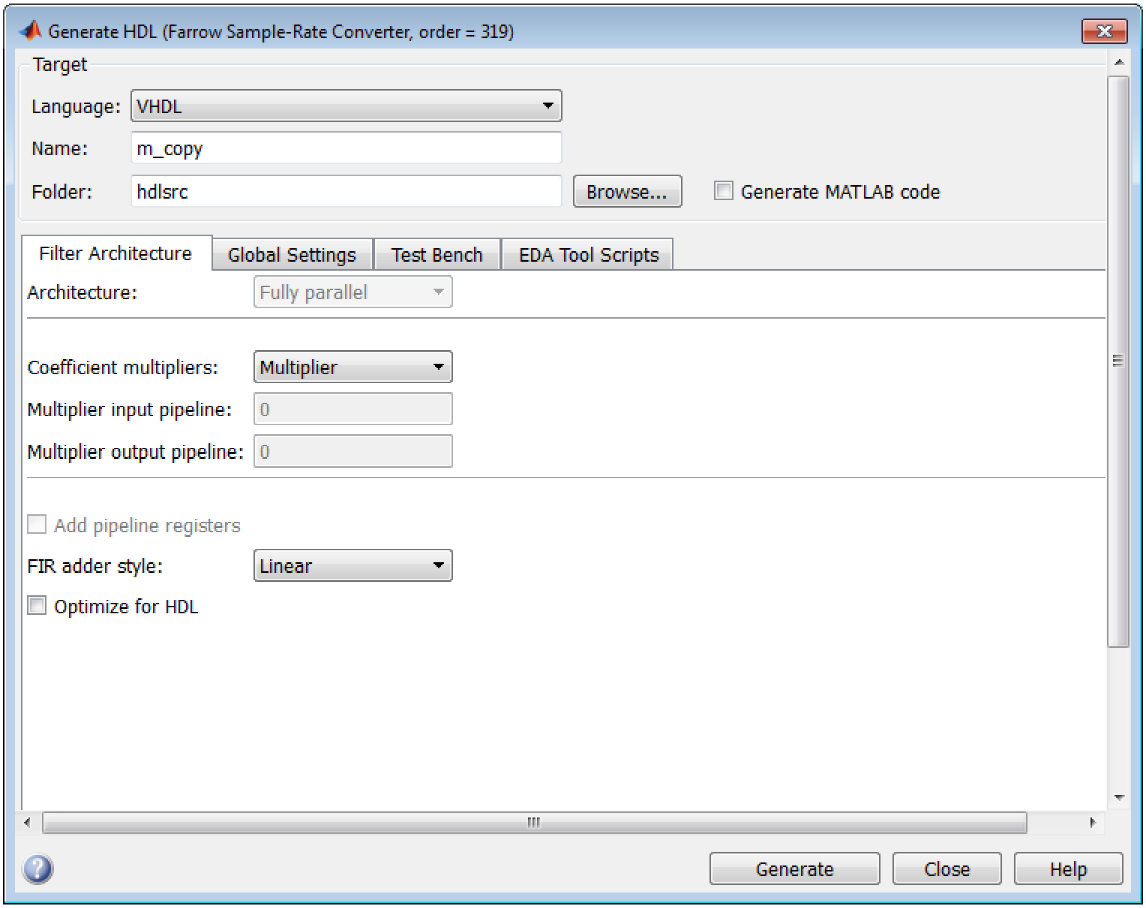
```
interp_cascade.addStage(hfar);
generatehdl(interp_cascade,'InputDataType',numerictype(1,16,15),...
                           'GenerateHDLTestbench','on');
```

## Generating Code for `dsp.FarrowRateConverter` Filters in the GUI

`fdatool` and `filterbuilder` do not currently support `dsp.FarrowRateConverter` filters. To generate code for a `dsp.FarrowRateConverter` filter in the HDL code generation GUI, use the `fdhdltool` command, as in the following example:

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

`fdhdltool` opens the Generate HDL dialog box for the `dsp.FarrowRateConverter` filter, as shown in the following figure.



Generate HDL (Farrow Sample-Rate Converter, order = 319)

Target

Language: VHDL

Name: m_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

- 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 3-23
- “Multirate Filters” on page 3-4

# Single-Rate Farrow Filters

In this section...

## About Code Generation for Single-Rate Farrow Filters

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

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

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

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

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

## Code Generation Properties for Farrow Filters

The following properties support Farrow filter code generation:

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

```
D = .3;
```

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

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

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

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

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

  - `'RandSweep'`: A vector of random values 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.
  - `'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 call to a function that returns a vector. Alternatively, create the vector in the workspace and pass it in as shown in the following code example:

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

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

    ```
    D = .3;
    hd = dfilt.farrowfd(D);
    inputdata = generatetbstimulus(hd, 'TestBenchStimulus', {'ramp'});
    ```

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

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

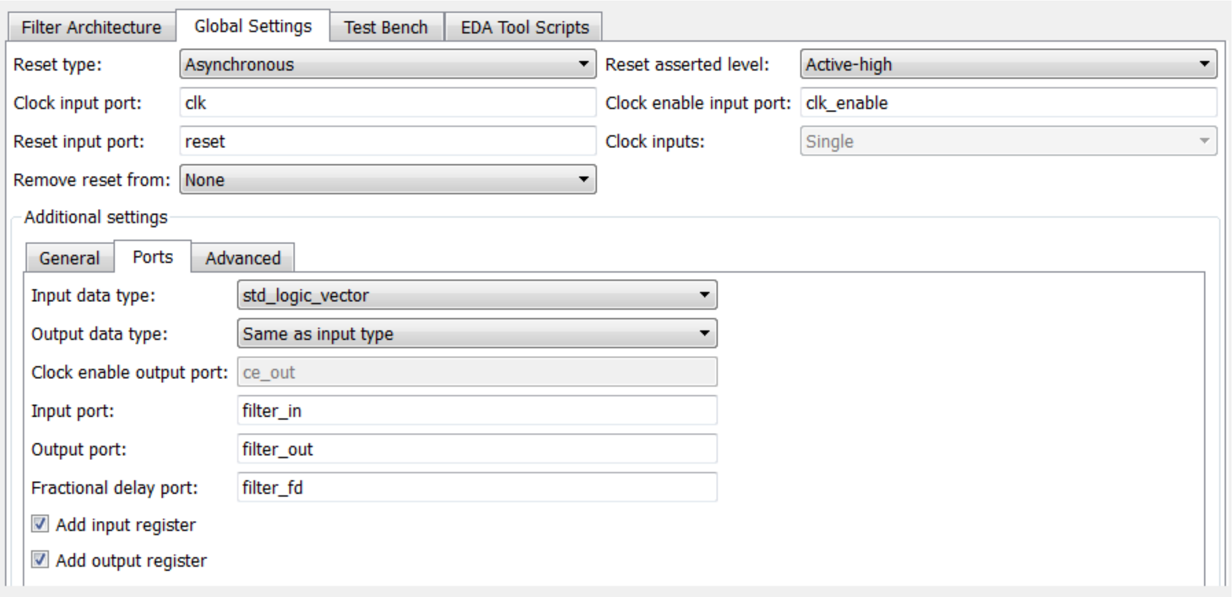
## GUI Options for Farrow Filters

This section describes Farrow filter code generation options that are available in the Filter Design HDL Coder GUI. These options correspond to the properties described in "Code Generation Properties for Farrow Filters" on page 3-23.

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

The Farrow filter options are:

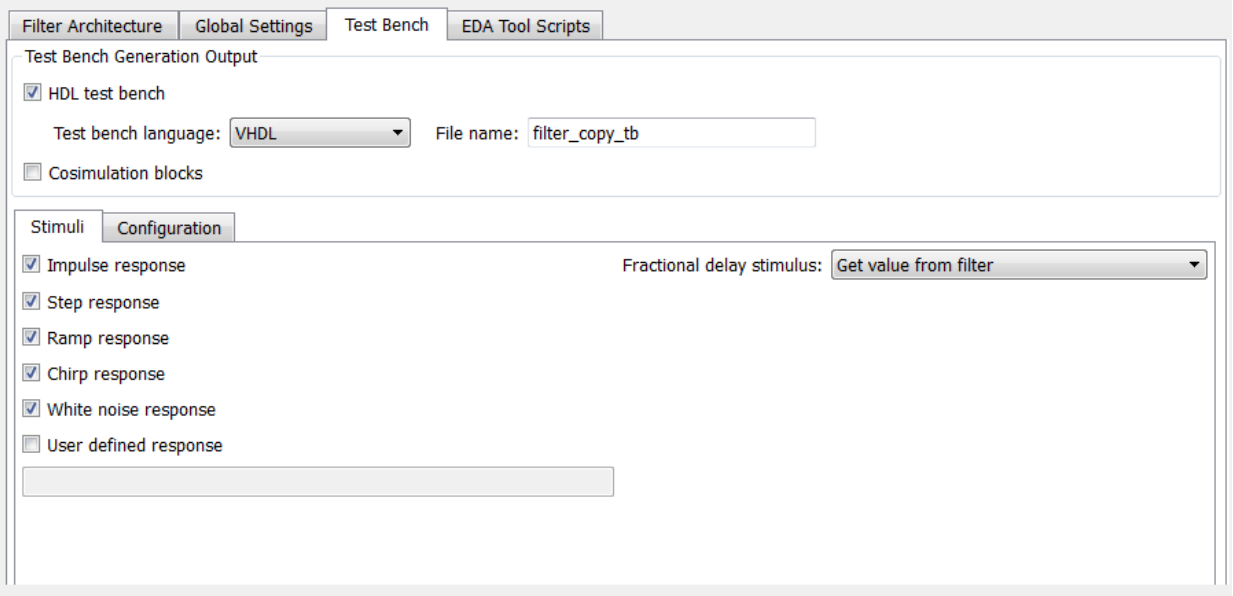
- The **Fractional delay port** field in the **Ports** pane of the Generate HDL dialog box (shown) specifies the name of the fractional delay port in generated code. The default name is `filter_fd`.



Reset type: Asynchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Remove reset from: None

Input data type: std_logic_vector
Output data type: Same as input type
Clock enable output port: ce_out
Input port: filter_in
Output port: filter_out
Fractional delay port: filter_fd
☑ Add input register
☑ Add output register

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



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: filter_copy_tb

☐ Cosimulation blocks

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

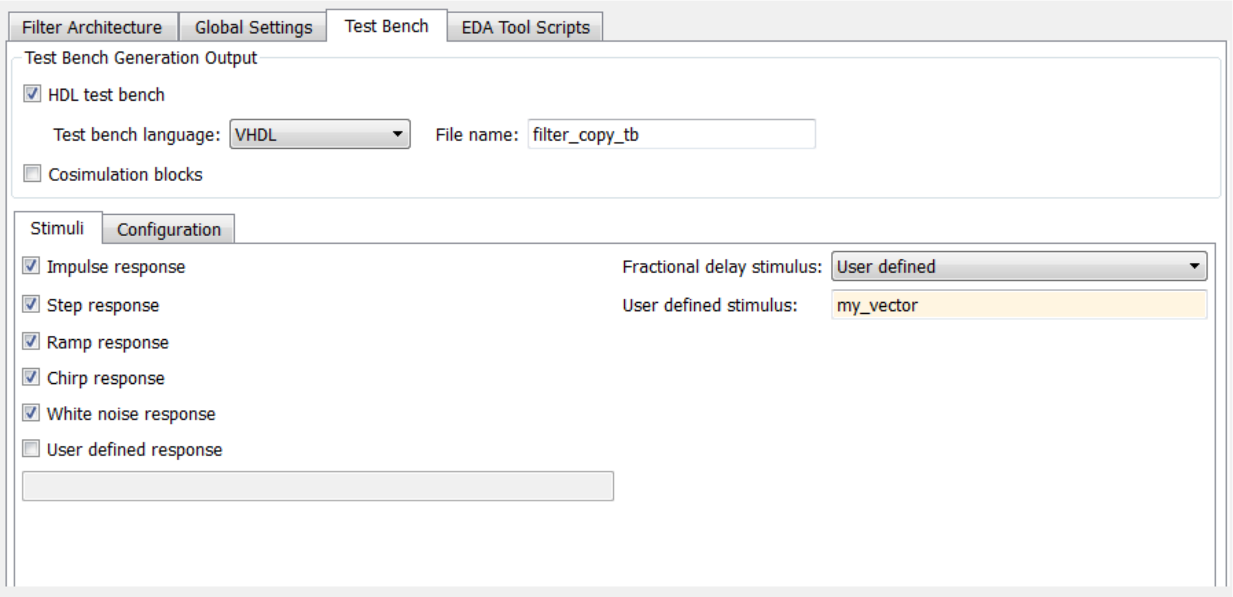
☐ User defined response

Fractional delay stimulus: Get value from filter

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

- **Get value from filter**: (default). A constant value is obtained from the `FracDelay` property of the Farrow filter object, and applied to the fractional delay port.
- **Ramp sweep**. A vector of values incrementally increasing over the range from 0 to 1. This stimulus signal has the same duration as the 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** field is enabled. You can enter a call to a function that returns a vector in the **User defined stimulus** field. Alternatively, create the vector as a workspace variable and enter the variable name, as shown in the following figure.



☑ HDL test bench

Test bench language: VHDL

File name: filter_copy_tb

□ Cosimulation blocks

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

Fractional delay stimulus: User defined

User defined stimulus: my_vector

## Farrow Filter Code Generation Mechanics

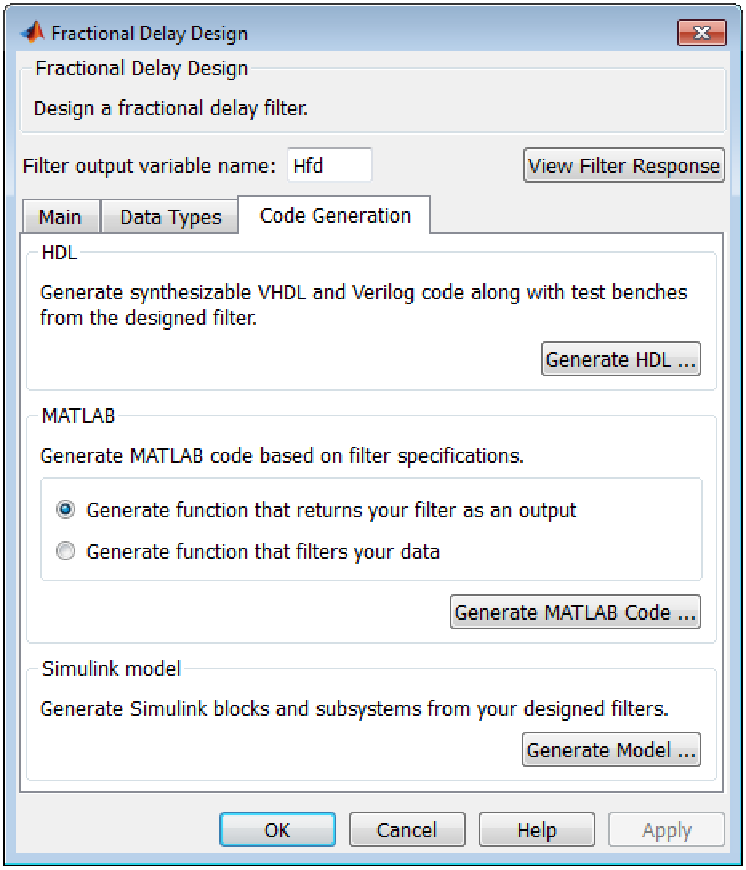
FDATool does not support design or import of Farrow filters. To generate HDL code for a Farrow filter, use one of the following methods:

- Use the MATLAB command line to create a Farrow filter object, initiate code generation, and set Farrow-related properties, as in the examples shown in “Code Generation Properties for Farrow Filters” on page 3-23.
- Use the MATLAB command line to create a Farrow filter object. Then open the Generate HDL dialog box. For example, these commands create a Farrow linear fractional delay filter object `Hd` and pass it in to `fdhdltool`:

```
D = .3
Hd = dfilt.farrowlinearfd(D)
Hd.arithmetic = 'fixed'
```

```
fdhdltool(Hd)
```

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



Fractional Delay Design

Fractional Delay Design

Design a fractional delay filter.

Filter output variable name: Hfd

HDL

Generate synthesizable VHDL and Verilog code along with test benches from the designed filter.

MATLAB

Generate MATLAB code based on filter specifications.

● Generate function that returns your filter as an output

○ Generate function that filters your data

Simulink model

Generate Simulink blocks and subsystems from your designed filters.

### Options Disabled for Farrow Filters

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

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

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

# Programmable Filter Coefficients for FIR Filters

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

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

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

Programmable filter coefficients are supported for the following 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 the following 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 3-30.

You can also generate a processor interface for loading IIR filter coefficients. See “Programmable Filter Coefficients for IIR Filters” on page 3-40.

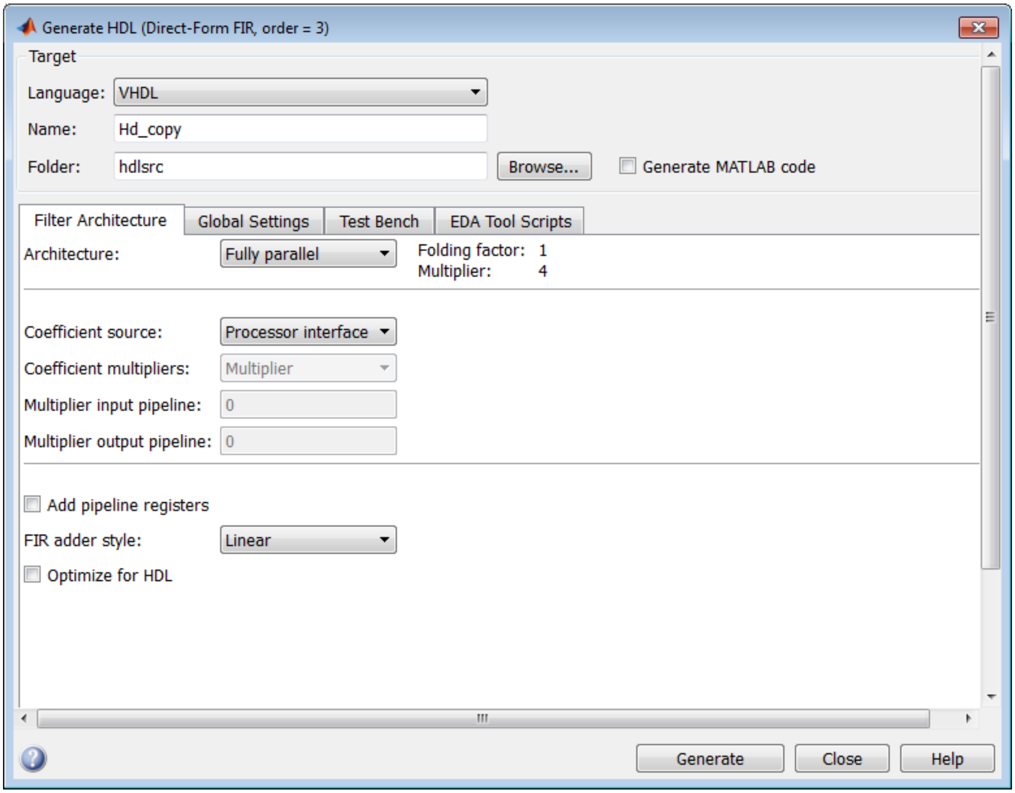
**In this section...**

## GUI Options for Programmable Coefficients

The following GUI options let you specify programmable coefficients:

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

  The corresponding command-line property is CoefficientSource.



Generate HDL (Direct-Form FIR, order = 3)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 4

Coefficient source: Processor interface

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

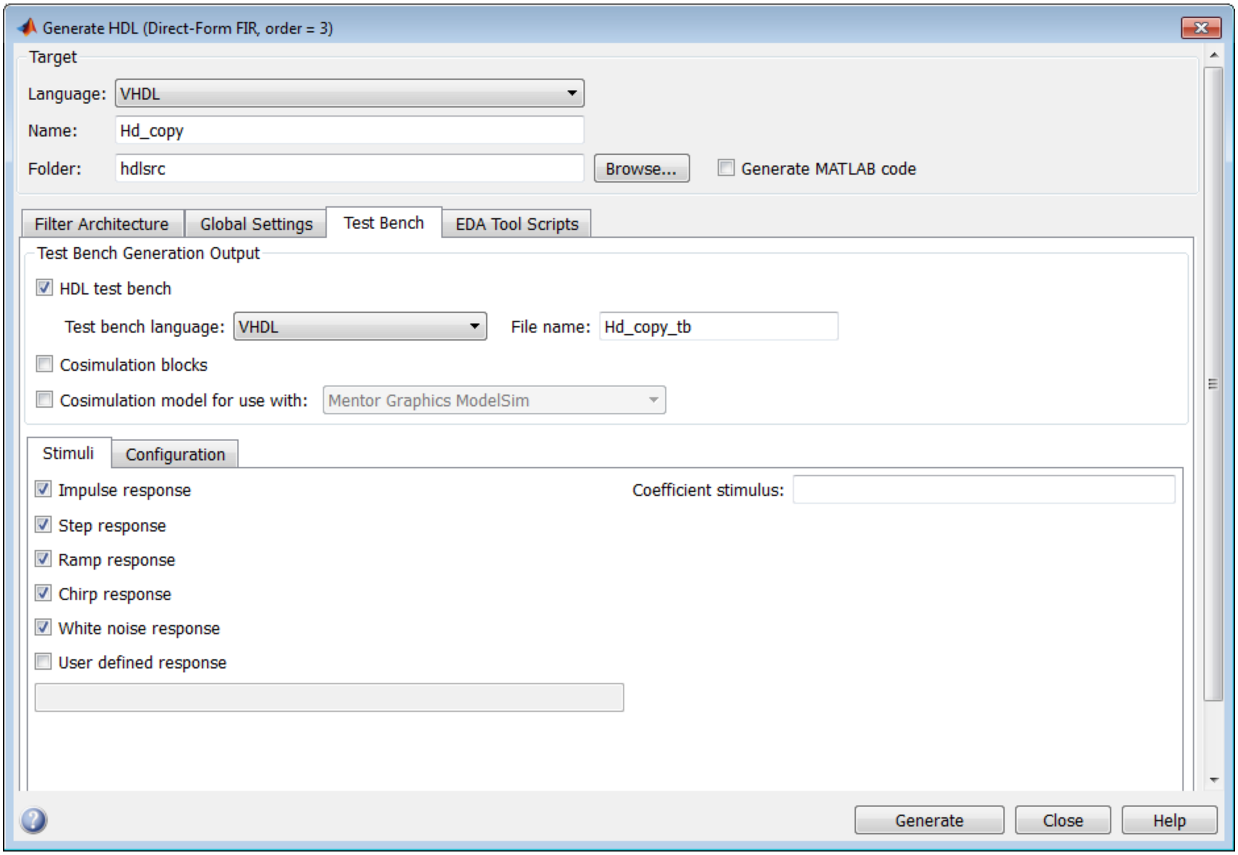
□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

  The corresponding command-line property is TestBenchCoeffStimulus.



Generate HDL (Direct-Form FIR, order = 3)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: Hd_copy_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

Coefficient stimulus:

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

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

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

For an example, see "Test Bench for FIR Filter with Programmable Coefficients" on page 9-13.

## Using Programmable Coefficients with Serial FIR Filter Architectures

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

- Fully serial

- `Partly serial`
- `Cascade serial`

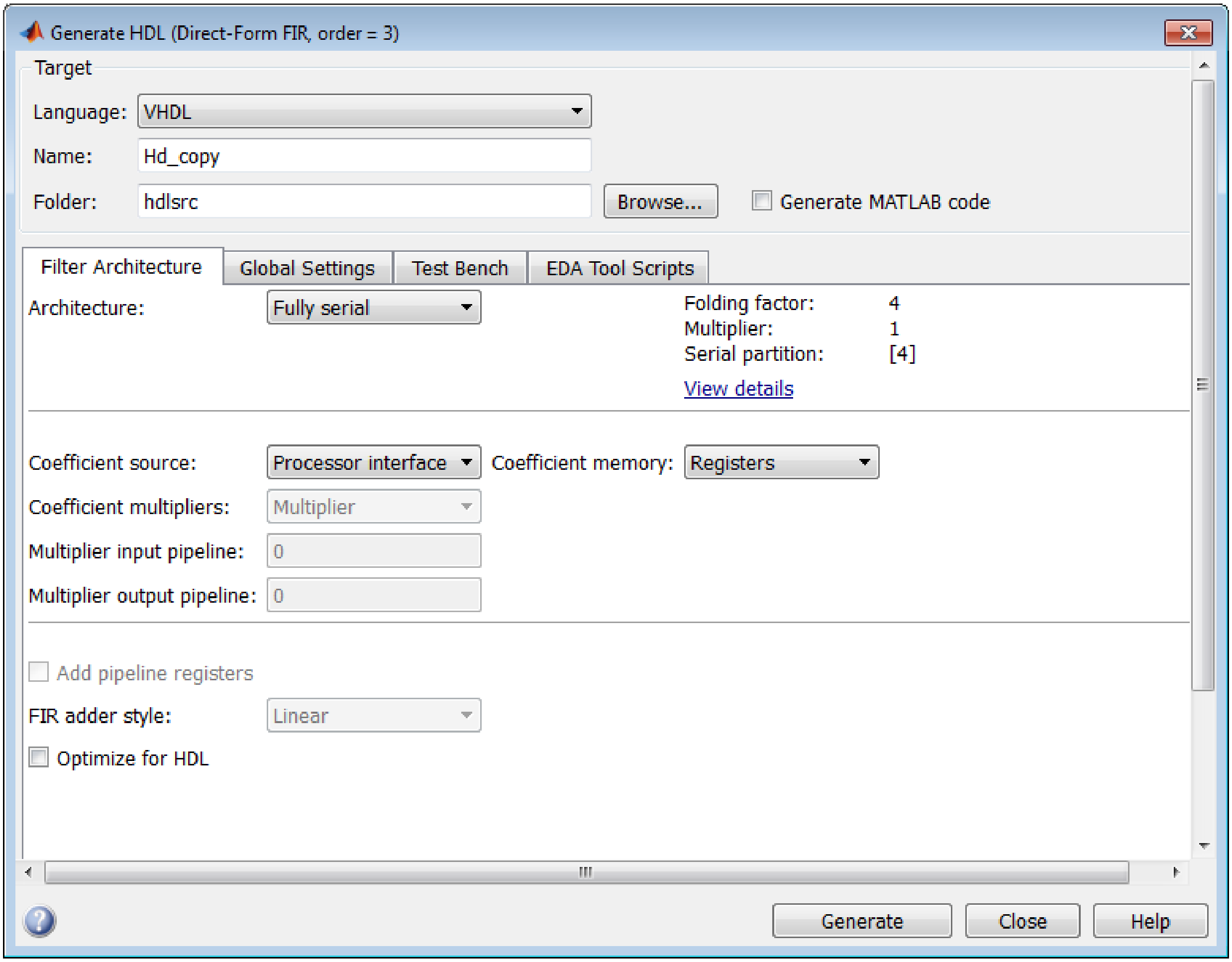
### Specifying Memory for Programmable Coefficients

By default, the processor interface for programmable coefficients loads the coefficients from a register file. The **Coefficient memory** 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.



Generate HDL (Direct-Form FIR, order = 3)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

☐ Generate MATLAB code

Architecture: Fully serial

Folding factor: 4

Multiplier: 1

Serial partition: [4]

Coefficient source: Processor interface

Coefficient memory: Registers

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

☐ Add pipeline registers

FIR adder style: Linear

☐ Optimize for HDL

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. |

| Coefficient memory Selection | Description |
|---|---|
|  | 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.<br>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 the following points:

- When `write_enable` asserts, coefficients load from the`coeffs_in` port into coefficient memory at the address specified by `write_address`.
- `write_done` can assert 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 filter, `Hd`, that is an asymmetric filter with 11 coefficients.

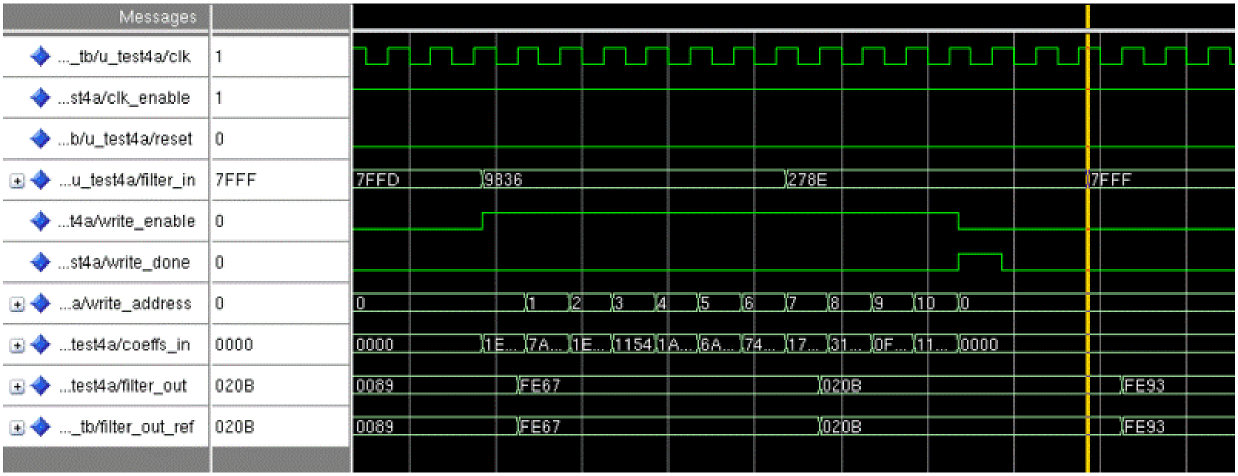
```
rng(13893,'v5uniform')
b = rand(1,23)
Hd = dfilt.dfasymfir(b)
Hd.Arithmetic = 'fixed'
```

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

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

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

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

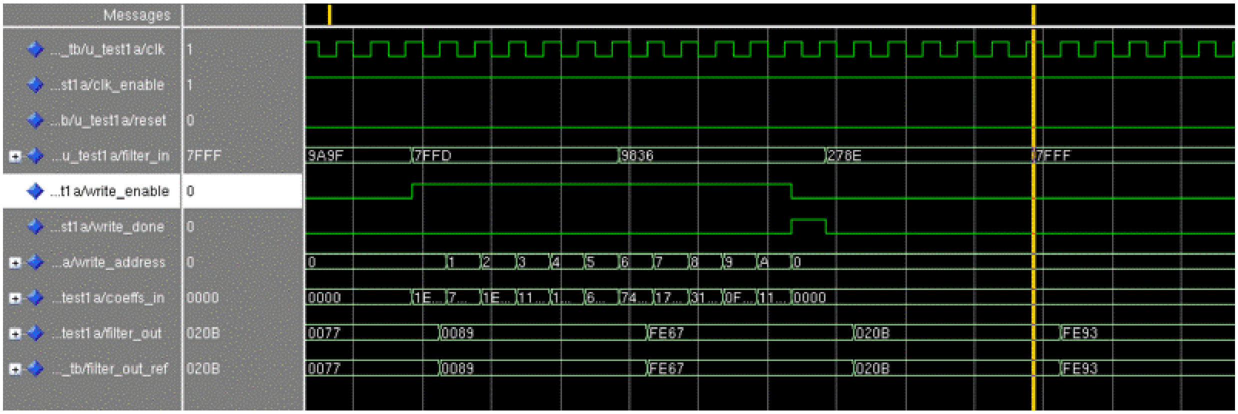
The timing diagram illustrates the rate of coefficient loading relative to the rate of input data samples. While `write_enable` is asserted, 11 coefficient values are loaded, via `coeffs_in`, to 11 sequential memory addresses. On the next `clk` cycle, `write_enable` is deasserted and `write_done` is 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(Hd,'SerialPartition',[6 5],'CoefficientSource','ProcessorInterface')

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



| Messages | |
|---|---|
| ..._tb/u_test1a/clk | 1 |
| ...st1a/clk_enable | 1 |
| ...b/u_test1a/reset | 0 |
| ...u_test1a/filter_in | 7FFF |
| ...t1a/write_enable | 0 |
| ...st1a/write_done | 0 |
| ...a/write_address | 0 |
| ...test1a/coeffs_in | 0000 |
| ...test1a/filter_out | 020B |
| ..._tb/filter_out_ref | 020B |

# Programmable Filter Coefficients for IIR Filters

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

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

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

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

The following 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:
  ```
  Hd.OptimizeScaleValues = 0
  ```
  Check that the filter still has the desired response, using the `fvtool` and `filter`, commands. Disabling `Hd.OptimizeScaleValues` may add quantization at section inputs and outputs.

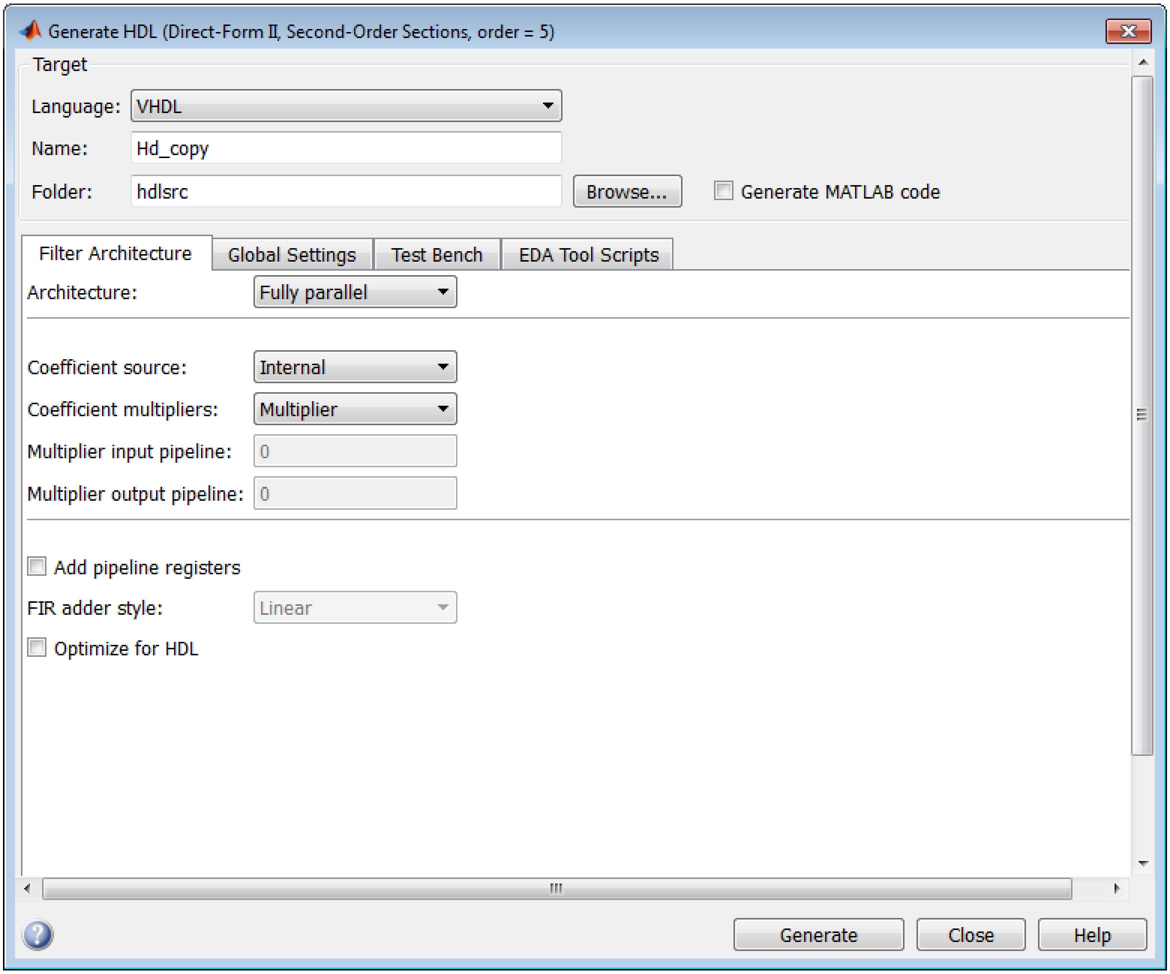
You can also generate a processor interface for loading FIR filter coefficients.“Specifying Memory for Programmable Coefficients” on page 3-35 for further information.

## Generate a Processor Interface for a Programmable IIR Filter

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

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

  The corresponding command-line property is CoefficientSource.



Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

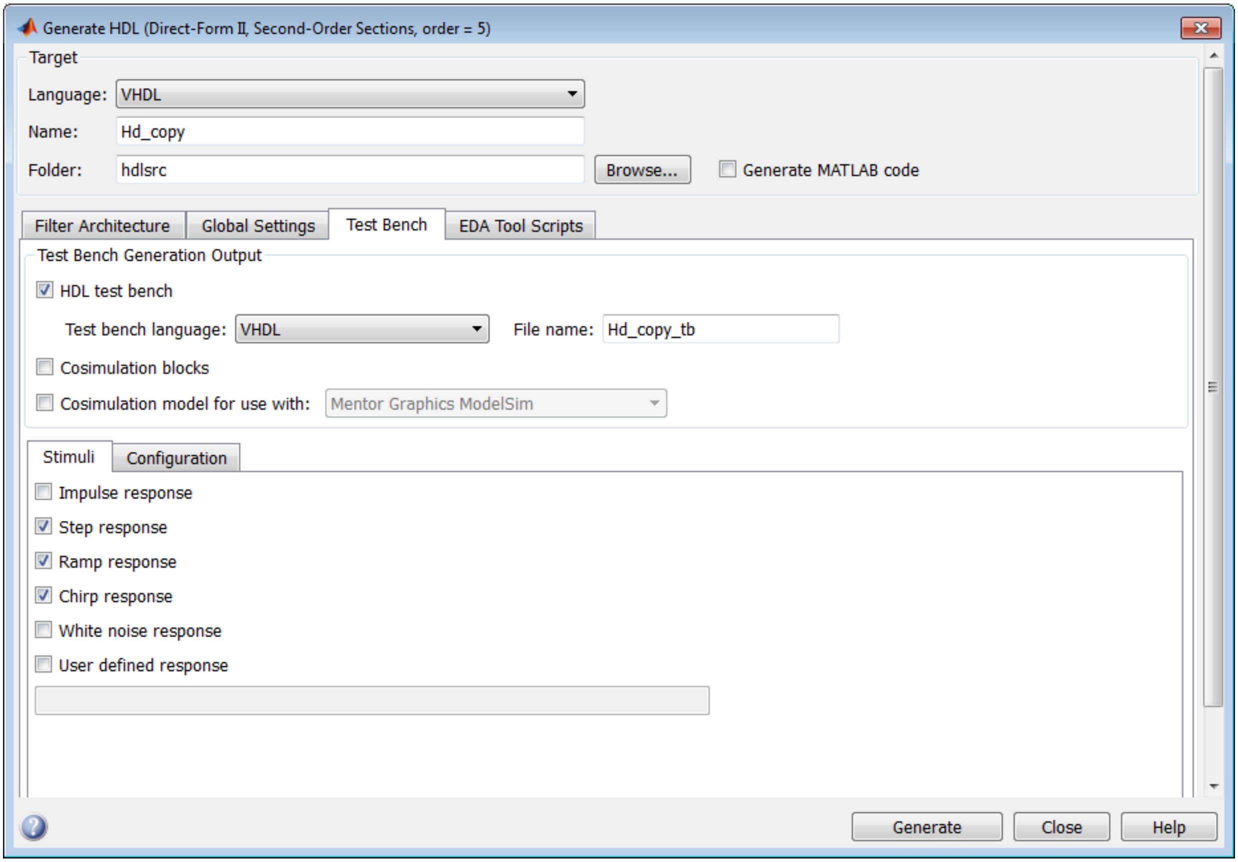
□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

  The corresponding command-line property is TestBenchCoeffStimulus.



Language: VHDL

Name: Hd_copy

Folder: hdlsrc

☐ Generate MATLAB code

☑ HDL test bench

Test bench language: VHDL

File name: Hd_copy_tb

☐ Cosimulation blocks

☐ Cosimulation model for use with: Mentor Graphics ModelSim

☐ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☐ White noise response

☐ User defined response

## 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 the following elements:

  - New_Hd.ScaleValues: column vector of scale values for the IIR filter
  - New_Hd.sosMatrix: second-order section (SOS) matrix for the IIR filter

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

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

  In this case, the filter processes the input stimuli twice. First, the test bench loads the coefficients from the filter object and forces the input stimuli to show the response. Then, the filter loads the set of coefficients specified in the TestbenchCoeffStimulus cell array, and shows 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_Hd.ScaleValues, the test bench uses the scale values loaded from the filter object twice. Likewise, if you omit New_Hd.sosMatrix, the test bench uses the SOS matrix loaded from the filter object twice.

## Addressing Scheme for Loading IIR Coefficients

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

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 $\mathtt{ceil(log_2N)}$ bits, where `N` is the number of sections.
- Coefficient address: Width is three bits.

The total width of the `write_address` port is therefore $\mathtt{ceil(log_2N)\ +\ 3}$bits.

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

# 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.

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

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

| Architecture | Available for Filter Types... |
|---|---|
| `Cascade serial` | • direct form<br>• direct form symmetric<br>• 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 4-6.

Alternatively, you can use the **Architecture** pop-up menu on the Generate HDL dialog box to choose parallel and serial architecture options, as described in "Select Architectures in the Generate HDL Dialog Box" on page 4-9.

**Note:** The coder also supports distributed arithmetic (DA), another highly efficient architecture for realizing filters. See "Distributed Arithmetic for FIR Filters" on page 4-21.

## Parallel and Serial Architectures

### Fully Parallel Architecture

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

### Serial Architectures

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

You can select from these serial architecture options:

- *Fully serial*: A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design would use a single multiplier and adder, executing a multiply/accumulate operation once for each tap. The multiply/accumulate section of the design runs at four times the 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 the following ways:

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

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

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

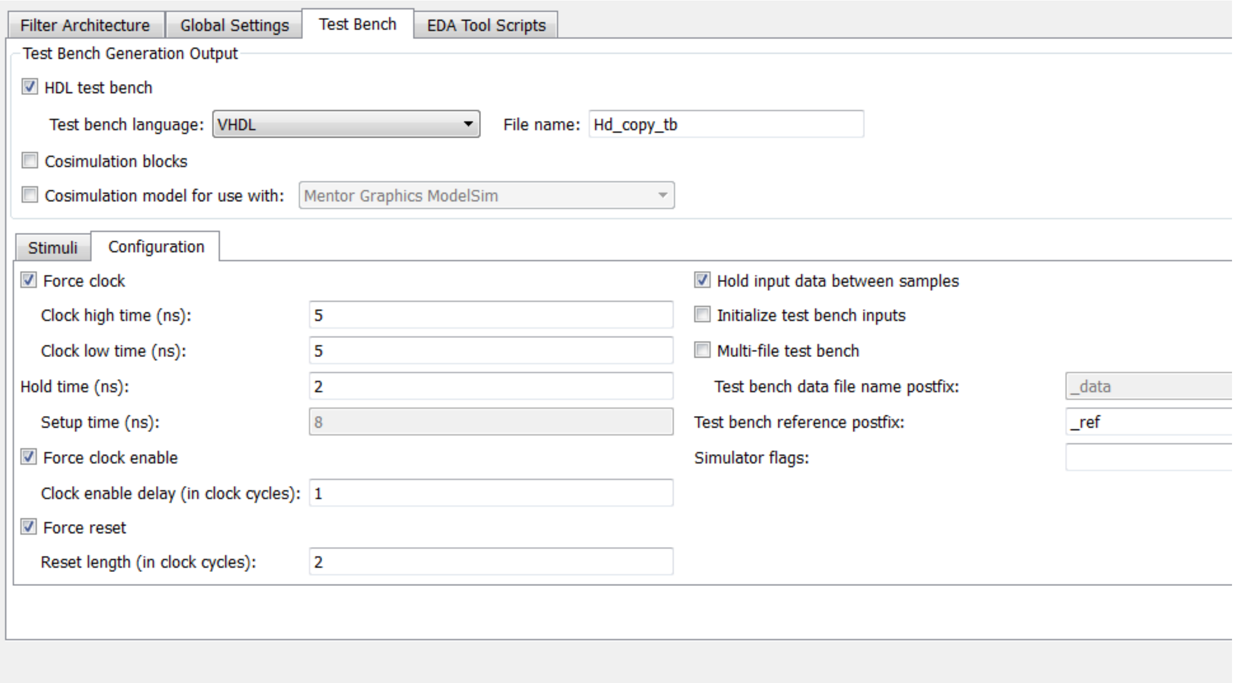
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the sequential products. An additional final register is used to store the summed result of each of the serial partitions. The operation 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 dialog box, with **Hold input data between samples** set to its default setting.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: Hd_copy_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Simulator flags:

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'` |
| Partly serial | `[p1 p2 p3...pN]`: a vector of `N`integer elements, where `N` is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. When you define the partitioning for a partly serial architecture, consider the following:<br>• 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]`.<br>• 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.<br>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'` |

| To Generate This Architecture... | Set SerialPartition to... | Set ReuseAccum to... |
|---|---|---|
| Cascade-serial with explicitly specified partitioning | `[p1 p2 p3...pN]`: a vector of integers having `N` elements, where `N` is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must equal the length of the filter. The values of the vector elements must appear in descending order, except that the last two elements must be equal. For example, for a filter of length 9, partitions such as`[5 4]` or `[4 3 2]` would be legal, but the partitions `[3 3 3]` or `[3 2 4]` 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 “Compare Serial Architectures for FIR Filter” on page 9-7.

### 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 the following parameters:

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

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

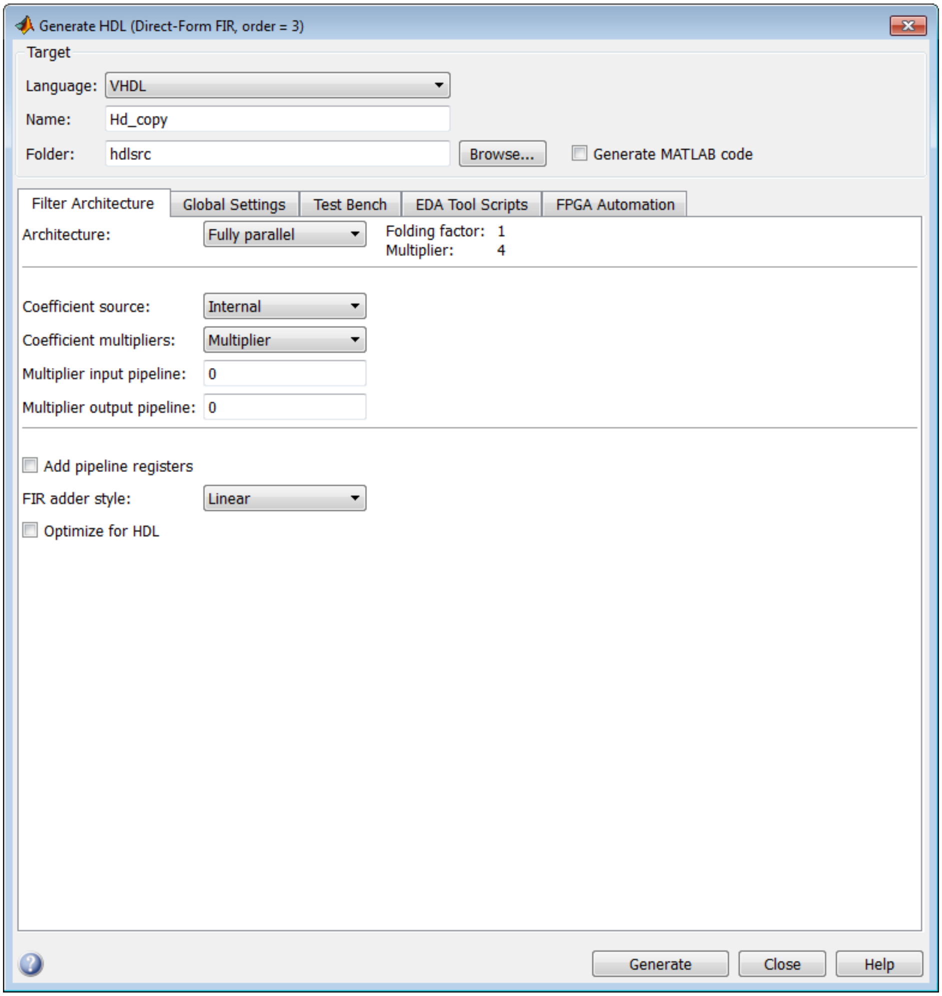
For an example, see “Serial Architecture for IIR Filter” on page 9-9.

## Select Architectures in the Generate HDL Dialog Box

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

### Specifying a Fully Parallel Architecture

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



Generate HDL (Direct-Form FIR, order = 3)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 4

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

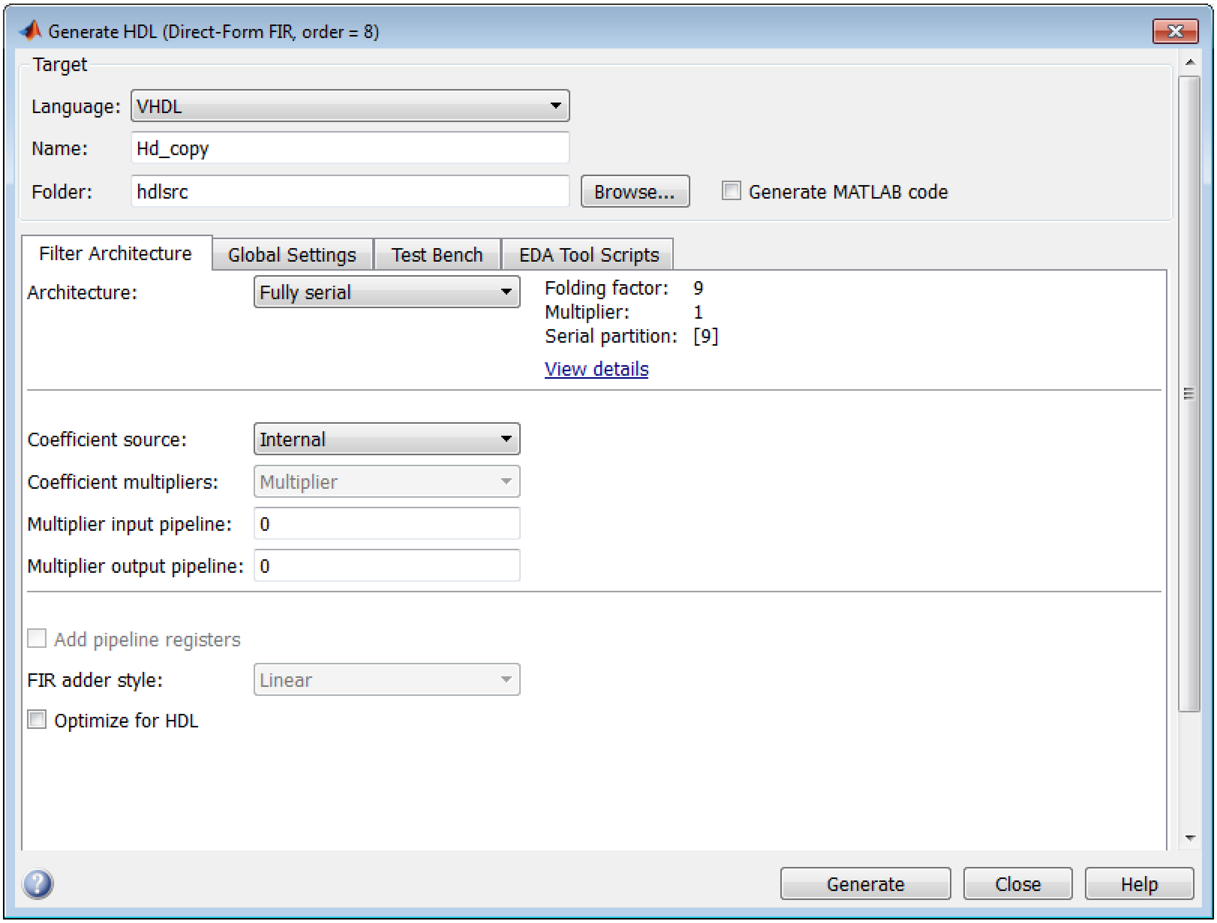
FIR adder style: Linear

□ Optimize for HDL

### Specifying a Fully Serial Architecture

When you select the `Fully serial`, **Architecture** options, the Generate HDL dialog box 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 the following figure.

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



Generate HDL (Direct-Form FIR, order = 8)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

☐ Generate MATLAB code

Architecture: Fully serial

Folding factor: 9
Multiplier: 1
Serial partition: [9]

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

☐ Add pipeline registers

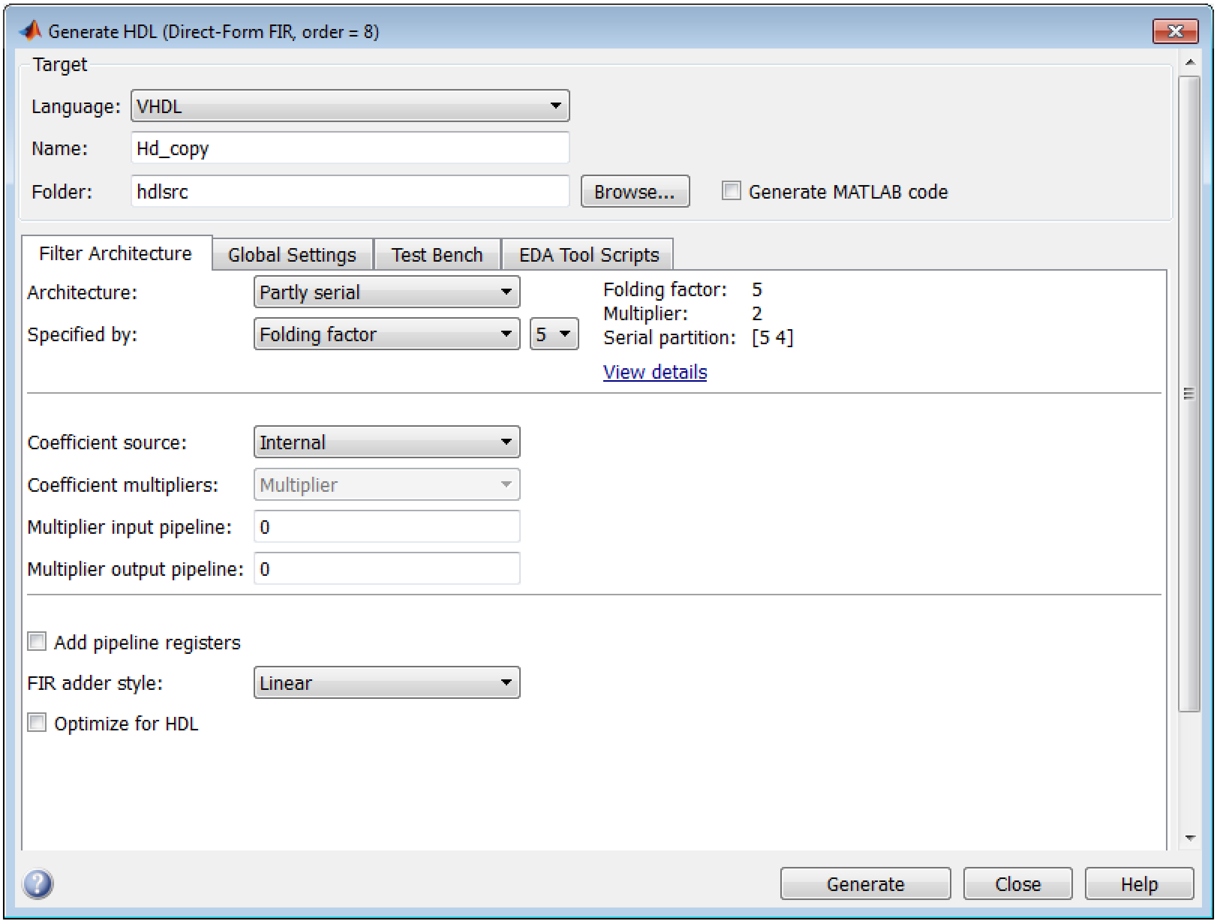
FIR adder style: Linear

☐ Optimize for HDL

### Specify Partitions for a Partly Serial Architecture

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

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



Generate HDL (Direct-Form FIR, order = 8)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Partly serial

Specified by: Folding factor 5

Folding factor: 5

Multiplier: 2

Serial partition: [5 4]

View details

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

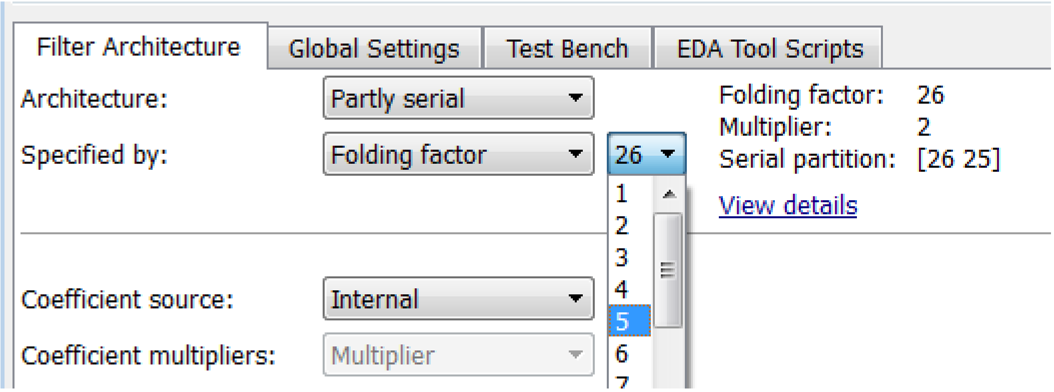
□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

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



Architecture: Partly serial

Specified by: Folding factor 26

Folding factor: 26

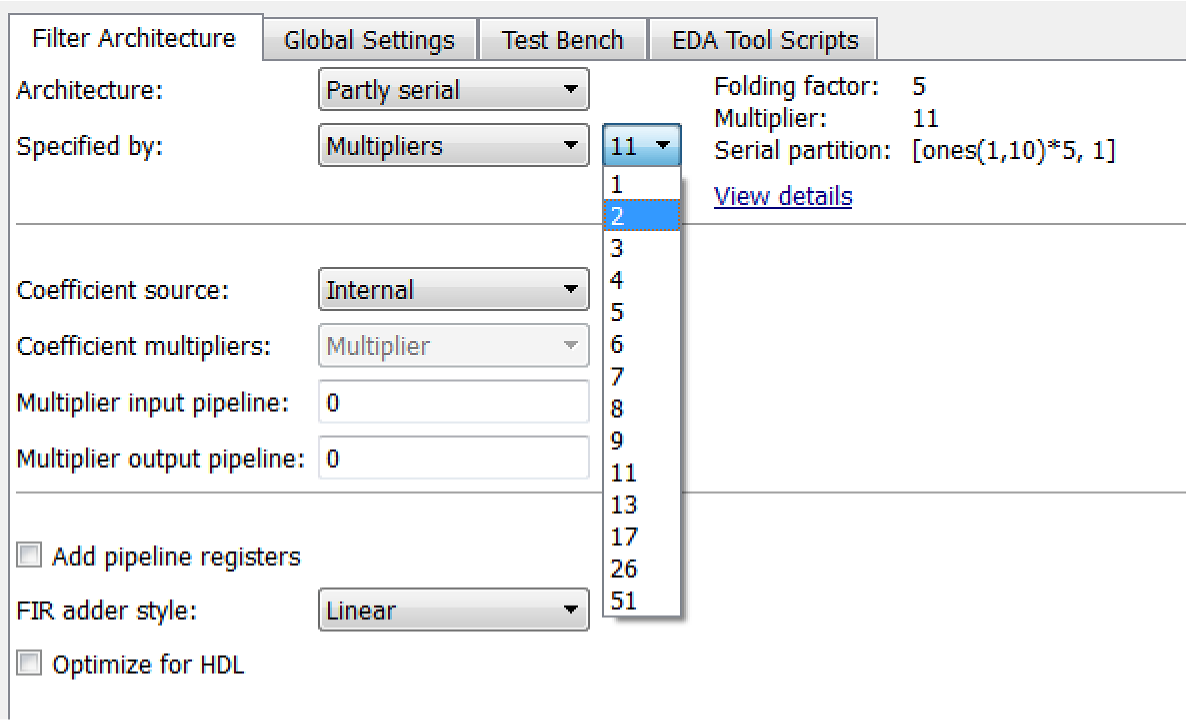
Multiplier: 2

Serial partition: [26 25]

Coefficient source: Internal

Coefficient multipliers: Multiplier

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



Architecture: Partly serial

Specified by: Multipliers 11

Folding factor: 5

Multiplier: 11

Serial partition: [ones(1,10)*5, 1]

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

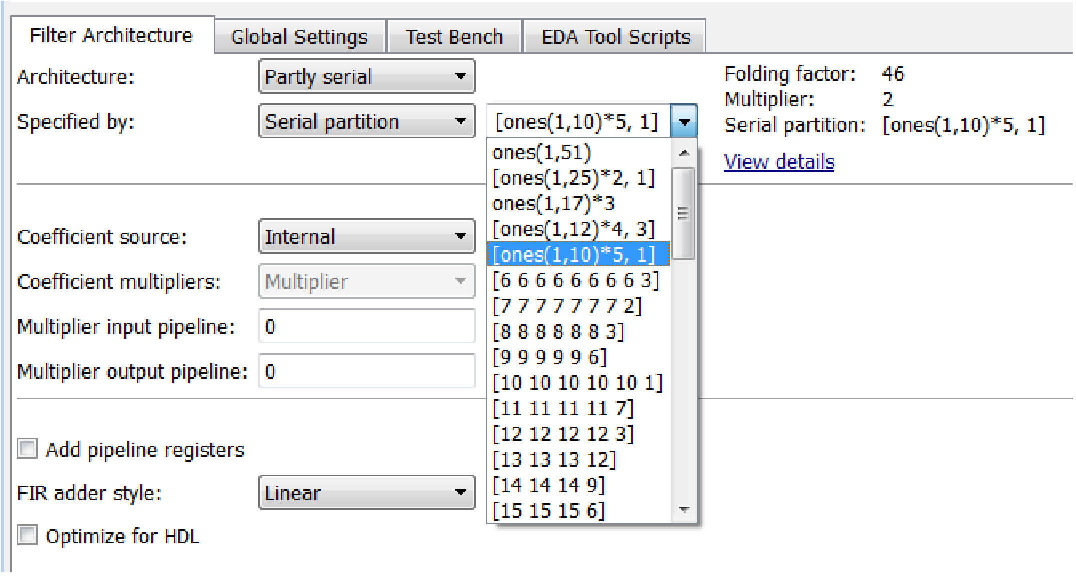
Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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



Architecture: Partly serial

Specified by: Serial partition [ones(1,10)*5, 1]

Folding factor: 46
Multiplier: 2
Serial partition: [ones(1,10)*5, 1]

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

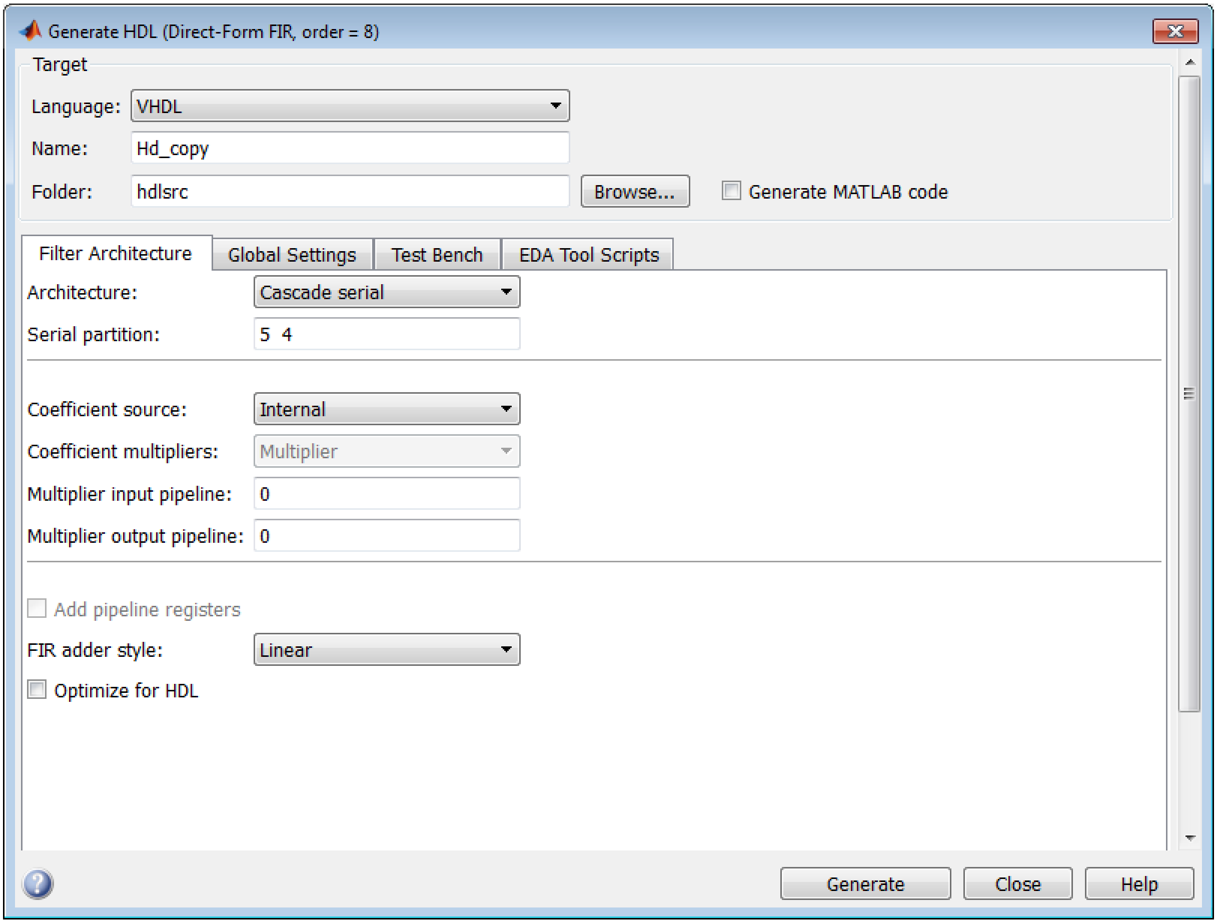
□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

## Specifying a Cascade Serial Architecture

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



Generate HDL (Direct-Form FIR, order = 8)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Cascade serial

Serial partition: 5 4

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

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

### Specifying Serial Architectures for IIR SOS Filters

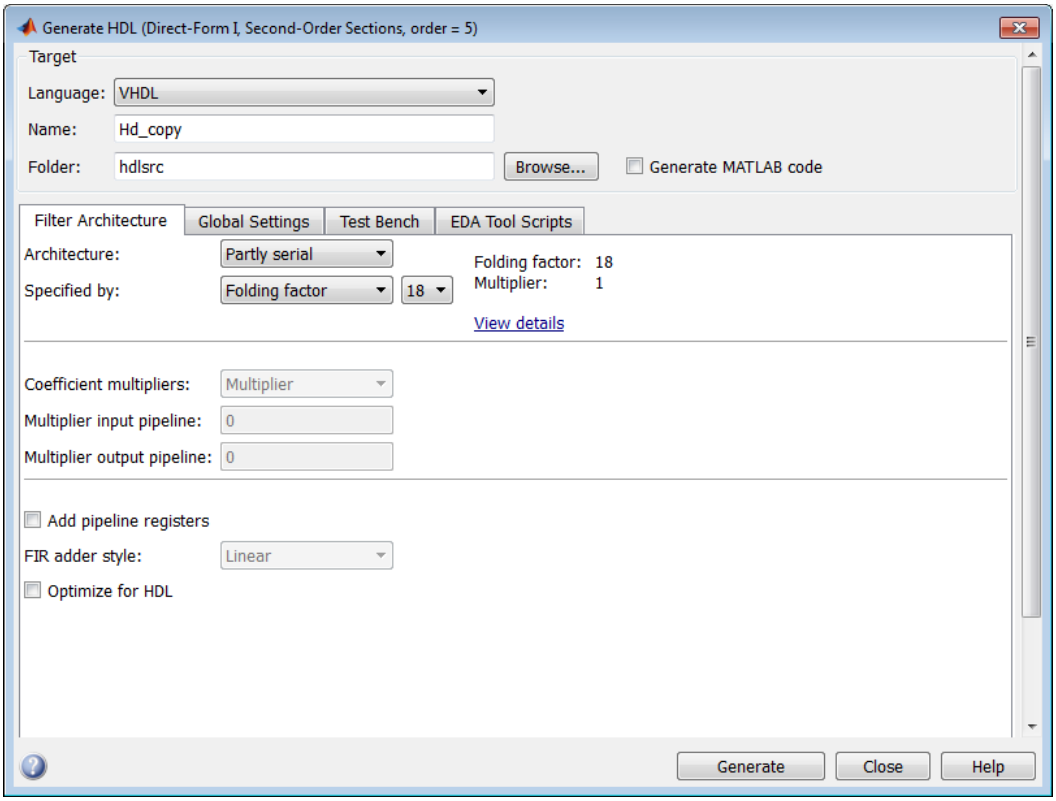
To specify a partly or fully serial architecture for an IIR SOS filter structure in the GUI, you set the following options:

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

### Example: Direct Form I SOS Filter

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

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


Generate HDL (Direct-Form I, Second-Order Sections, order = 5)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Partly serial

Specified by: Folding factor 18

Folding factor: 18
Multiplier: 1

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

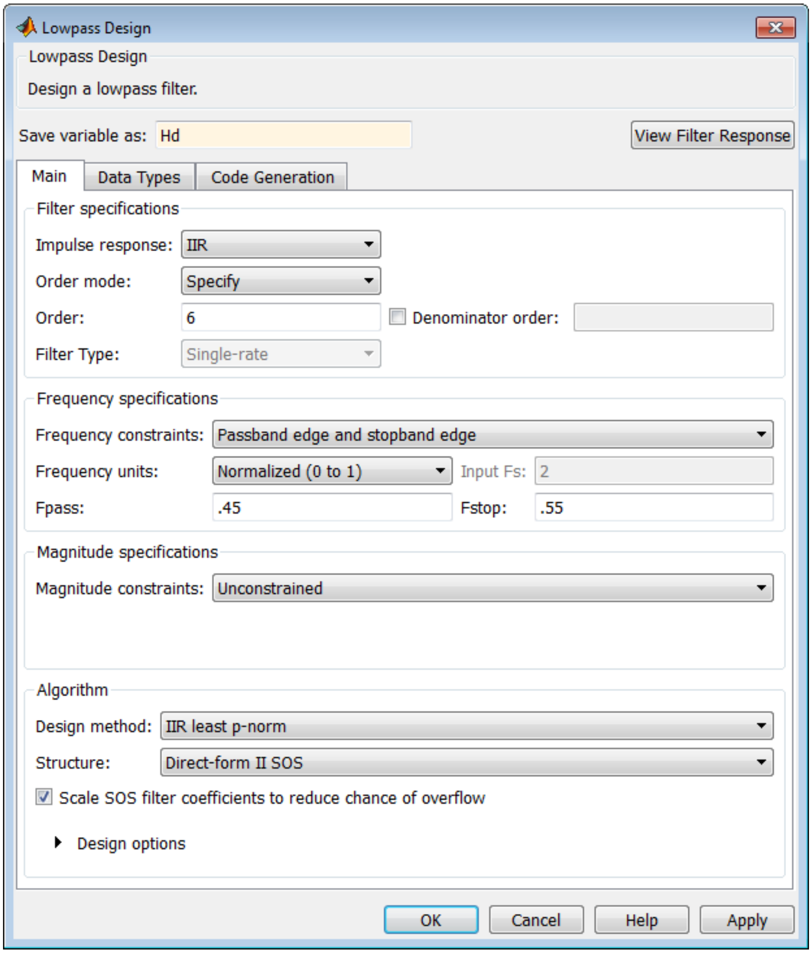
□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

### Example: Direct Form II SOS Filter

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



Lowpass Design

Lowpass Design

Design a lowpass filter.

Save variable as: Hd

Filter specifications

Impulse response: IIR

Order mode: Specify

Order: 6

□ Denominator order:

Filter Type: Single-rate

Frequency specifications

Frequency constraints: Passband edge and stopband edge

Frequency units: Normalized (0 to 1)

Input Fs: 2

Fpass: .45

Fstop: .55

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: IIR least p-norm

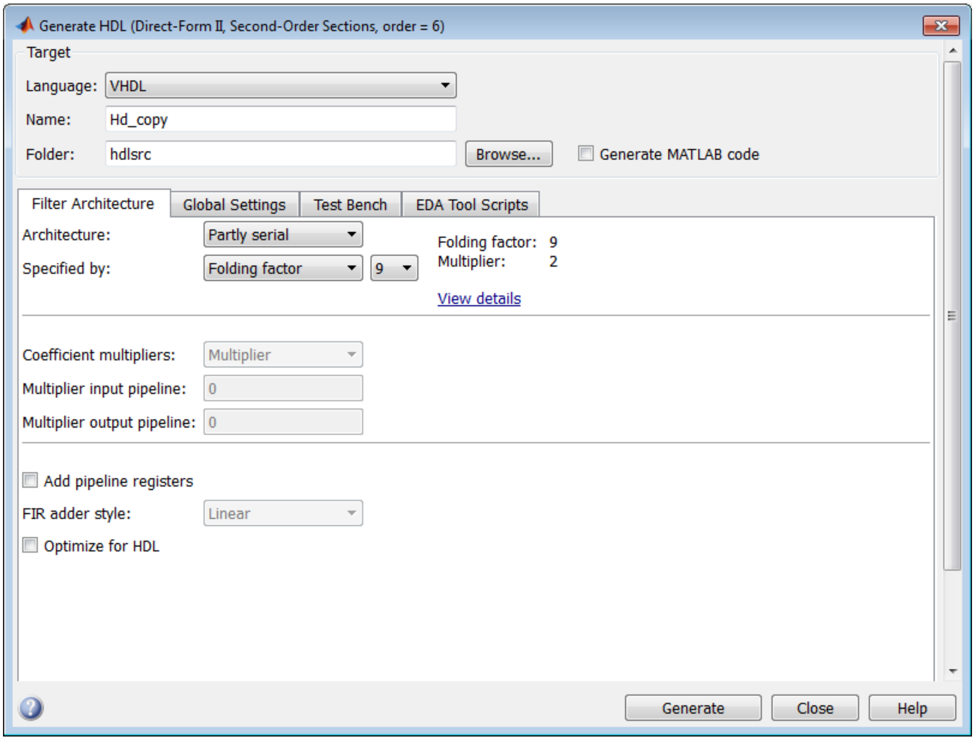
Structure: Direct-form II SOS

☑ Scale SOS filter coefficients to reduce chance of overflow

▸ Design options

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

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



Generate HDL (Direct-Form II, Second-Order Sections, order = 6)

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Partly serial

Specified by: Folding factor 9

Folding factor: 9
Multiplier: 2

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

### 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 4-21) 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, the following options are set to their default values and disabled:

  - **Coefficient multipliers**
  - **Add pipeline registers**
  - **FIR adder style**

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

# Distributed Arithmetic for FIR Filters

**In this section...**

## Distributed Arithmetic Overview

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 4-23. )

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

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

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

The delay line is tapped (based on the input word size `W`), to form a `W`-bit address that indexes into a lookup table (LUT). The LUT stores the possible sums of partial products over the filter coefficients space. 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 dialog box options). Before using these properties, review the following general requirements, restrictions, and other considerations for generation of DA code.

### Supported Filter Types

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

- 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.

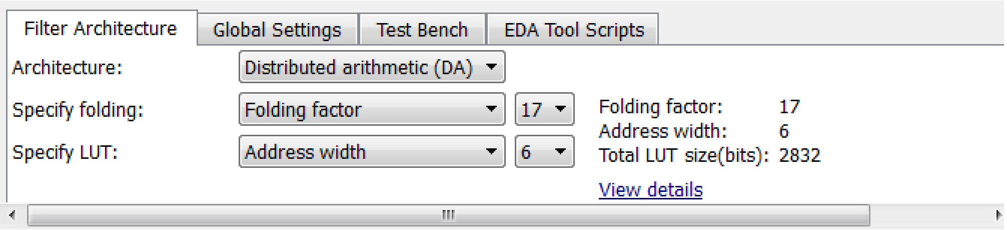
You can use the helper function `hdlfilterdainfo` to explore possible partitions and radix settings for your filter.

For examples, see

- "Distributed Arithmetic for Single Rate Filters" on page 9-10
- "Distributed Arithmetic for Multirate Filters" on page 9-11
- "Distributed Arithmetic for Cascaded Filters" on page 9-11

## Distributed Arithmetic Options in the Generate HDL Dialog Box

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



Architecture: Distributed arithmetic (DA)

Specify folding: Folding factor 17

Specify LUT: Address width 6

Folding factor: 17
Address width: 6
Total LUT size(bits): 2832

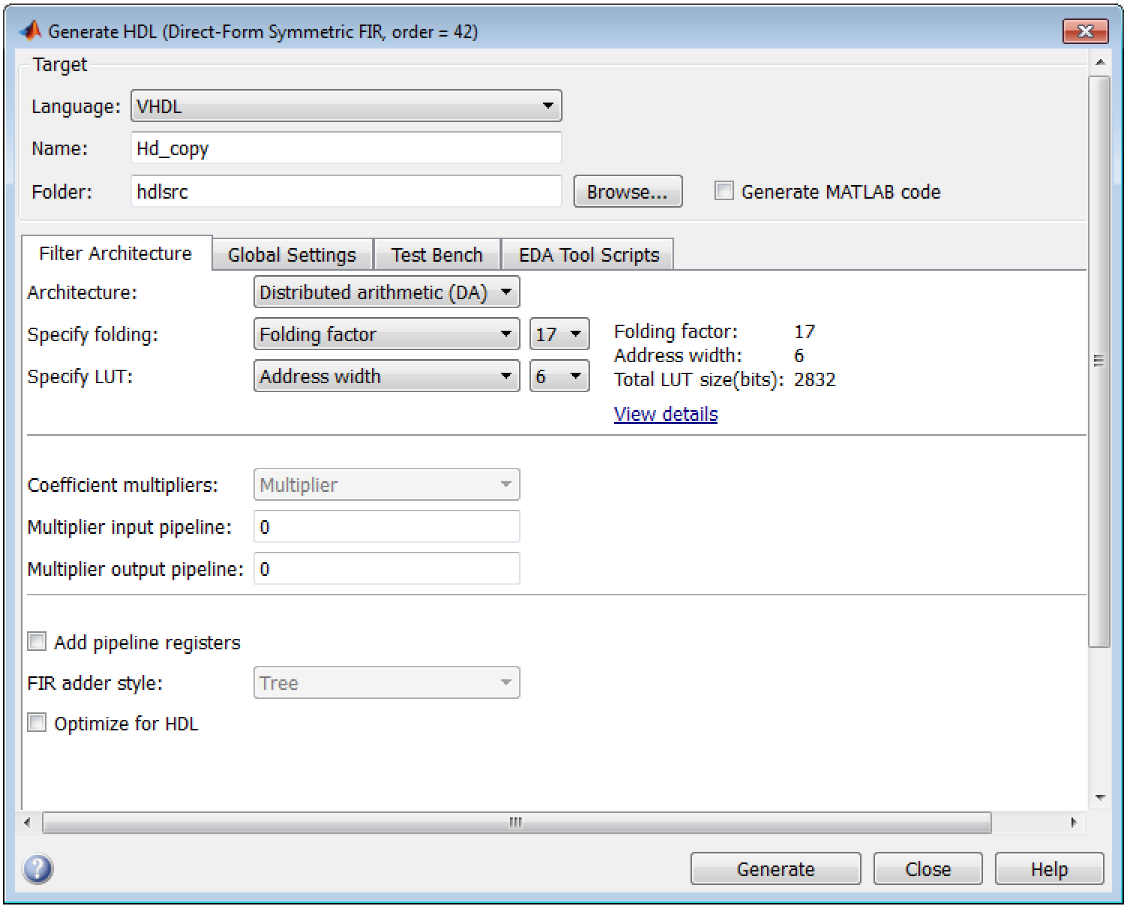
- The **Architecture** pop-up menu, which lets you enable DA code generation and displays related options.
- The **Specify folding** drop-down menu, which lets you directly specify the folding factor, or set a value for the DARadix property.
- The **Specify LUT** drop-down menu, which lets you directly 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 dialog box 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 dialog box, follow these steps:

1. Design a FIR filter (using FDATool, `filterbuilder`, or MATLAB commands) that meets the requirements described in "Requirements and Considerations for Generating Distributed Arithmetic Code" on page 4-23.
2. Open the Generate HDL dialog box.
3. Select `Distributed Arithmetic (DA)` from the **Architecture** pop-up menu.

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



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

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Distributed arithmetic (DA)

Specify folding: Folding factor 17

Specify LUT: Address width 6

Folding factor: 17
Address width: 6
Total LUT size(bits): 2832

Coefficient multipliers: Multiplier

Multiplier input pipeline: 0

Multiplier output pipeline: 0

□ Add pipeline registers

FIR adder style: Tree

□ Optimize for HDL

**4** Select one of the following options from the **Specify folding** drop-down menu:

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

**5** Select one of the following options from the **Specify LUT** drop-down menu:

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

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

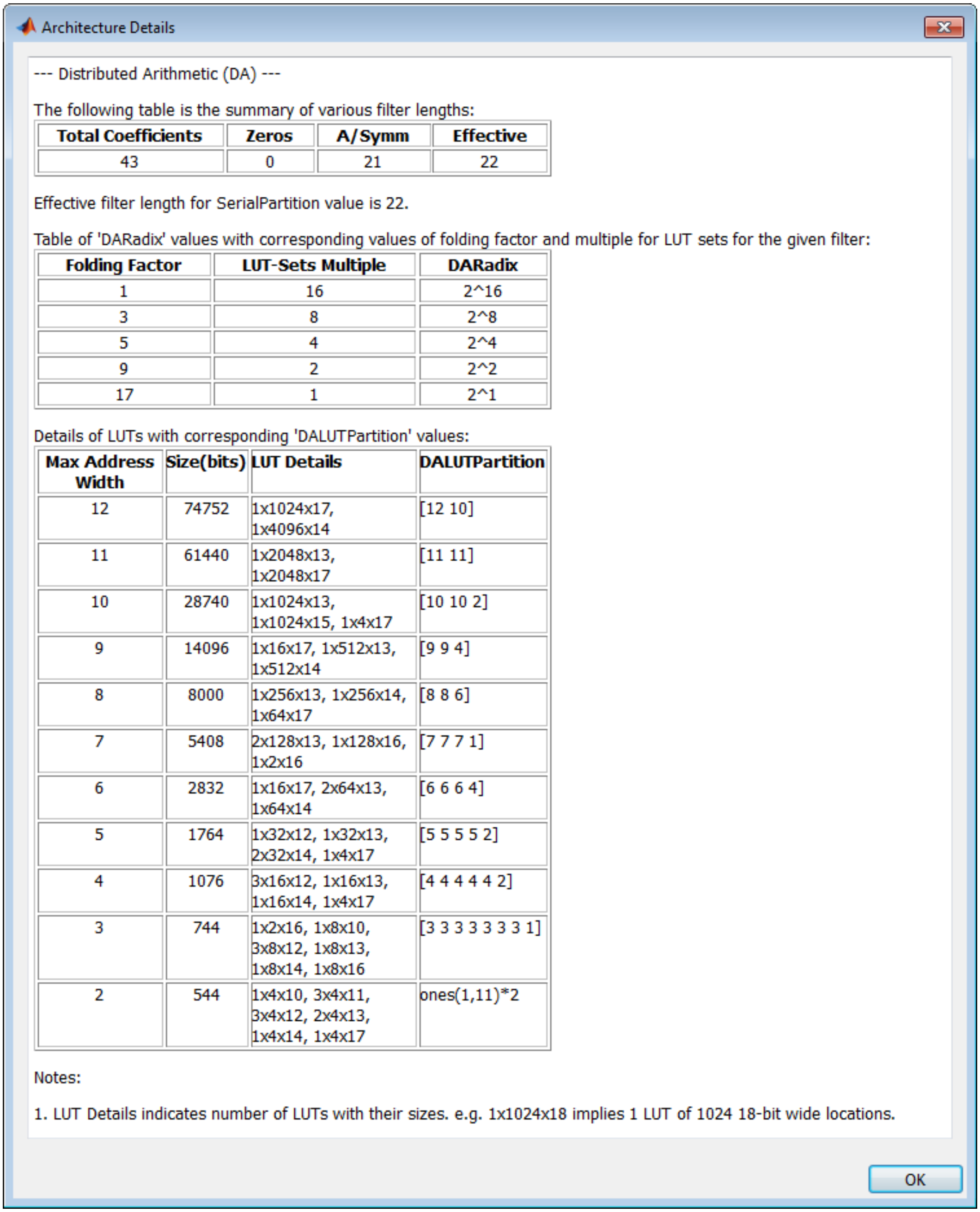
### Viewing Detailed DA Options

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

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

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

The following figure shows a typical report.



--- 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.

### 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 dialog box, all stages of a cascade use 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 the following 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 9-11
- "Serial Partitions for Cascaded Filter" on page 9-8
- "Cascaded Filter with Multiple Architectures" on page 9-13

# 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 dialog box.
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 dialog box. 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 4-33 for information on a related optimization.

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

# Improving Filter Performance with Pipelining

**In this section...**

## Optimizing the Clock Rate with Pipeline Registers

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

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

For example, for a sixth order IIR filter, the coder adds two pipeline registers. 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 4-34.

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

   a. Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.
   b. 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.

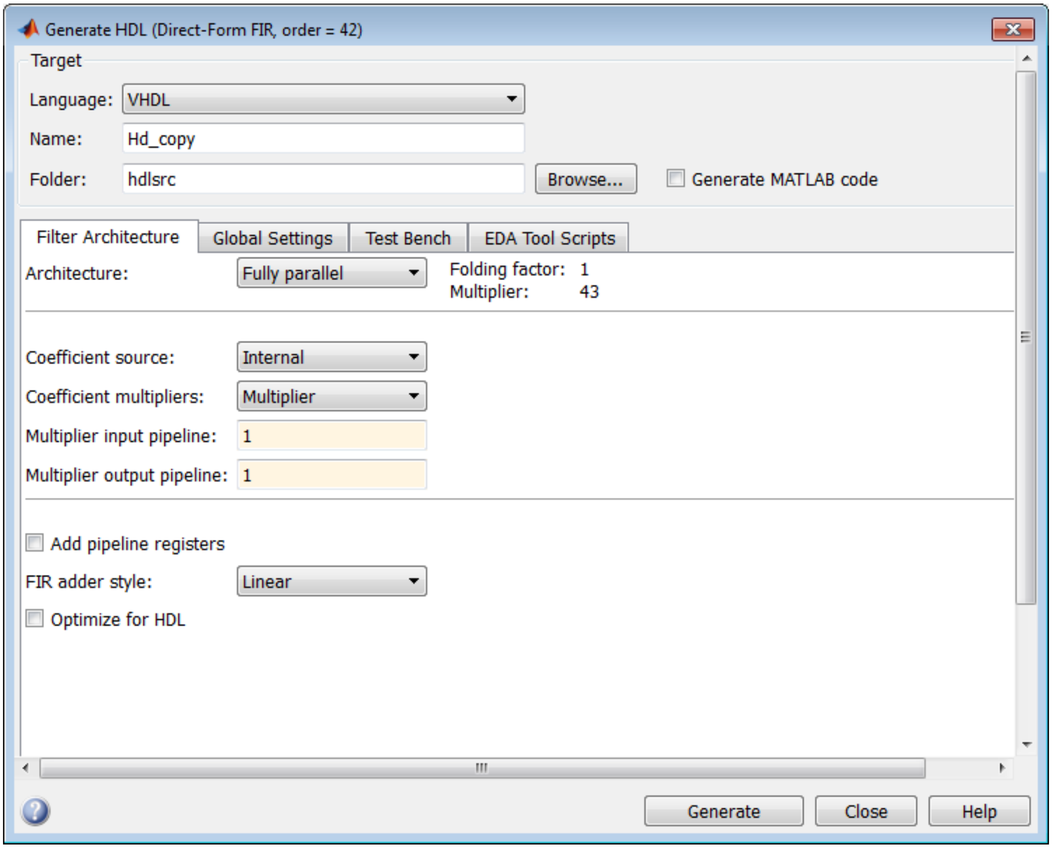
**Command-Line Alternative:** Use the `generatehdl` function with the property AddPipelineRegisters to optimize the filters with pipeline registers.

## 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.

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

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



Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Architecture: Fully parallel

Folding factor: 1

Multiplier: 43

Coefficient source: Internal

Coefficient multipliers: Multiplier

Multiplier input pipeline: 1

Multiplier output pipeline: 1

□ Add pipeline registers

FIR adder style: Linear

□ Optimize for HDL

**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 dialog box.

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

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

   a Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.

   b 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.

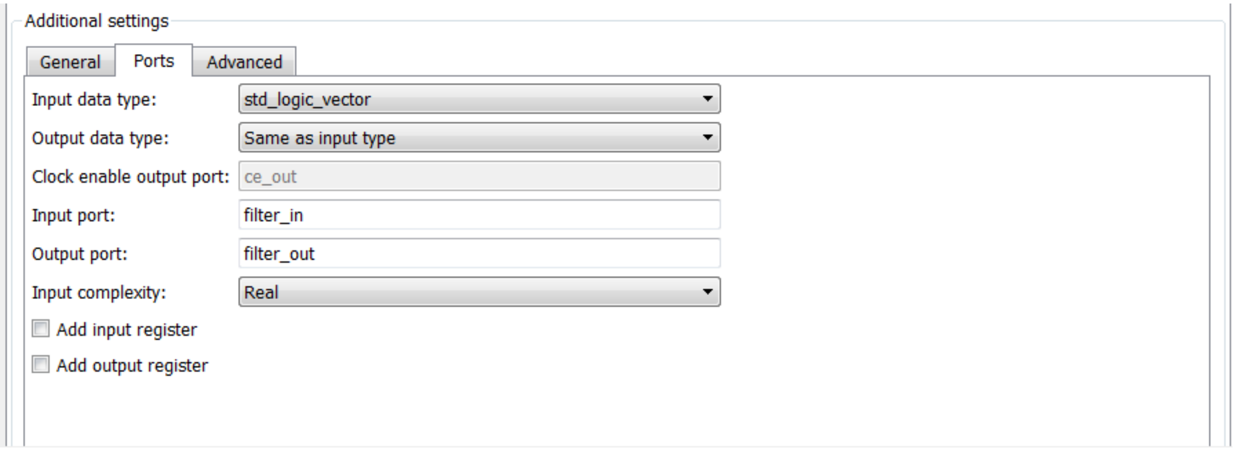
## Specifying or Suppressing Registered Input and Output

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

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

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

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



Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

Output port: filter_out

Input complexity: Real

□ Add input register

□ Add output register

**Command-Line Alternative:** Use the `generatehdl` and function with the properties AddInputRegister andAddOutputRegister 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:

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

To optimize generated code for clock speed or space requirements:

1. Select **Optimize for HDL** in the **Filter architecture** pane of the Generate HDL dialog box.
2. Consider setting an error margin for the generated test bench. The error margin is the number of least significant bits the test bench ignores when comparing the results. To set an error margin,

    a. Select the **Test Bench** pane in the Generate HDL dialog box. Then click the **Configuration** tab.

    b. 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.

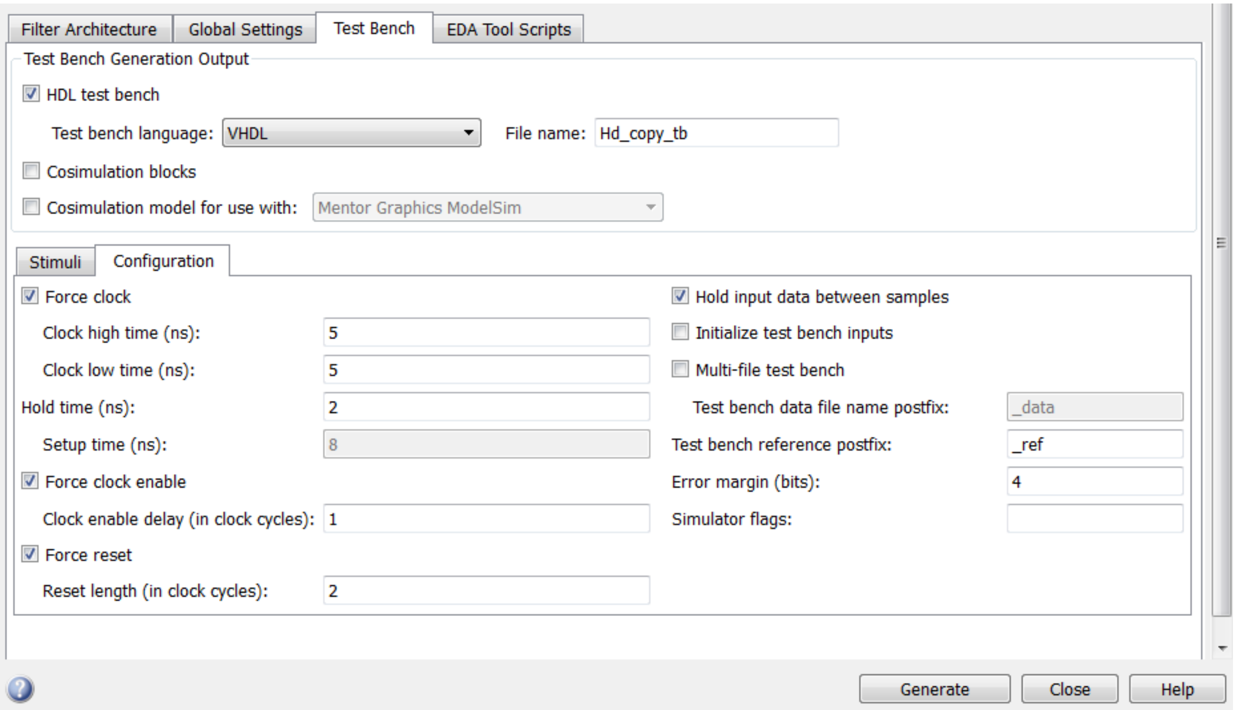
## Set Error Margin for Test Bench

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 dialog box.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. For fixed-point filters, the initial **Error margin (bits)** field has a default value of 4. 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.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: Hd_copy_tb

☐ Cosimulation blocks

☐ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

☐ Initialize test bench inputs

☐ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Error margin (bits): 4

Simulator flags:

**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

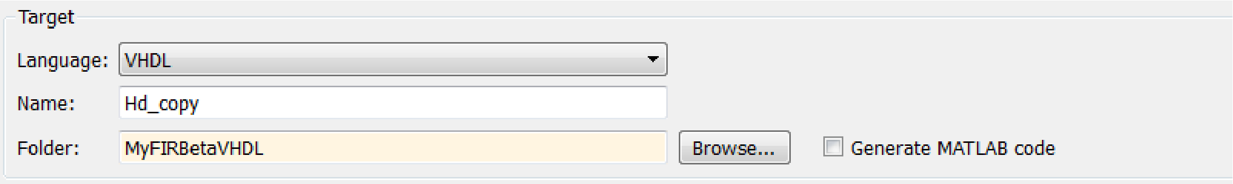
By default, the coder places generated HDL files in the subfolder `hdlsrc` under your current working folder. To direct the coder output to a folder other than the default target folder, use either the **Folder** field or the **Browse** button in the **Target** pane of the Generate HDL dialog box.

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 the following:

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

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



Target

Language: VHDL

Name: Hd_copy

Folder: MyFIRBetaVHDL

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

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

## Naming the Generated Files and Filter Entity

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

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

### Derivation of File Names

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

| Language | Generated File | File Name | Example |
|---|---|---|---|
| Verilog | Source file for the quantized filter | *`dfilt_name`*`.v` | `Hq.v` |
| | Source file for the test bench | *`dfilt_name`*`_tb.v` | `Hq_tb.v` |
| VHDL | Source file for the quantized filter | *`dfilt_name`*`.vhd` | `Hq.vhd` |
| | Source file for the test bench | *`dfilt_name`*`_tb.vhd` | `Hq_tb.vhd` |

| Language | Generated File | File Name | Example |
|---|---|---|---|
| | Package file, if required by the filter design | *dfilt_name*_pkg.vhd | Hq_pkg.vhd |

By default, the coder generates a single test bench file, containing test bench helper functions, data, and test bench code. You can split these elements into separate files, as described in “Splitting Test Bench Code and Data into Separate Files” on page 6-13.

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 Hd, the coder writes the VHDL code for the filter to files Hd_entity.vhd and Hd_arch.vhd (see “Splitting Entity and Architecture Code Into Separate Files” on page 5-6).

### Derivation of Entity Names

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

## Set HDL File Name Extensions

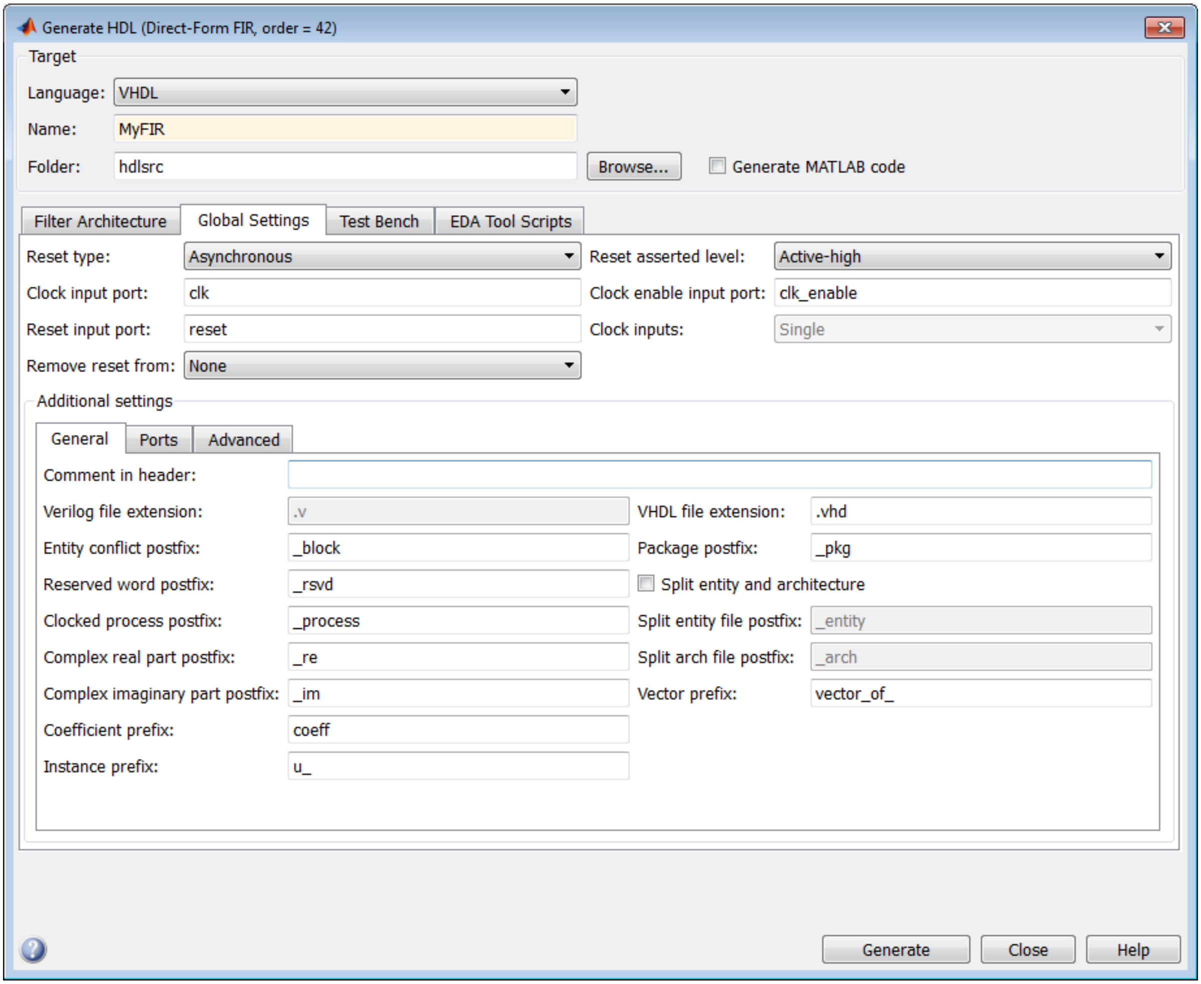
- “Set File Name Extension Via the Generate HDL Tool” on page 5-4
- “Set HDL File Name Extensions Via the Command-Line” on page 5-6

### 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 dialog box.
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.



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

Target

Language: VHDL

Name: MyFIR

Folder: hdlsrc

□ Generate MATLAB code

Reset type: Asynchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

Additional settings

Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

□ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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

### Set HDL File Name Extensions Via the Command-Line

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

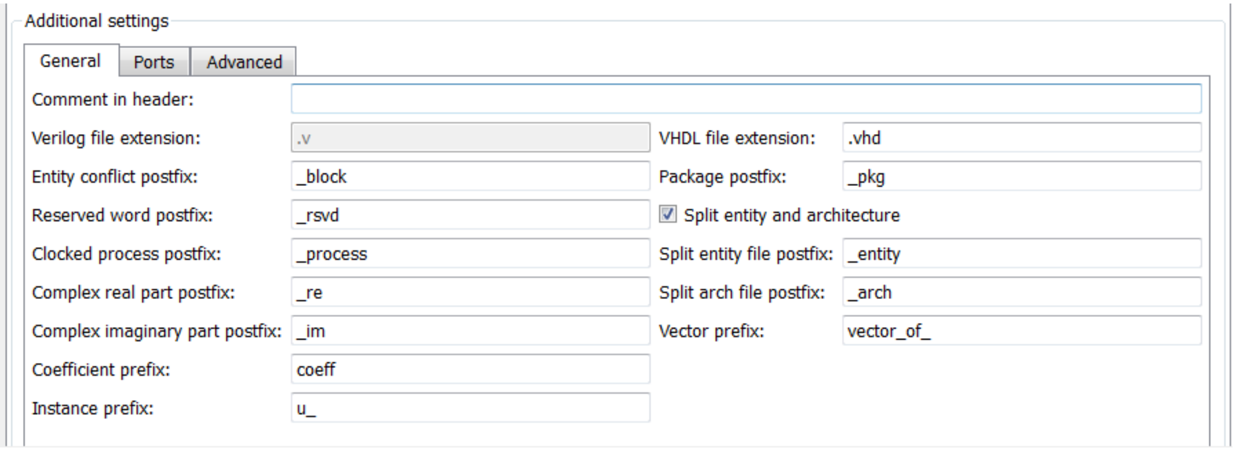
By default, the coder includes a VHDL entity and architecture code in the same generated VHDL file. Alternatively, you can instruct the coder to place the entity and architecture code in separate files. For example, instead of generated code residing in `MyFIR.vhd`, you can specify that the code reside 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 dialog box.
- Default postfix string values `_entity` and `_arch`.
- The VHDL file type extension, as specified by the **VHDL file extension** field on the **General** pane of the Generate HDL dialog box.

To split the filter source file, do the following:

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



Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

☑ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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

---

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

---

**Command-Line Alternative:** Use the `generatehdl` function with the property SplitEntityArch to split the 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 string, 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:

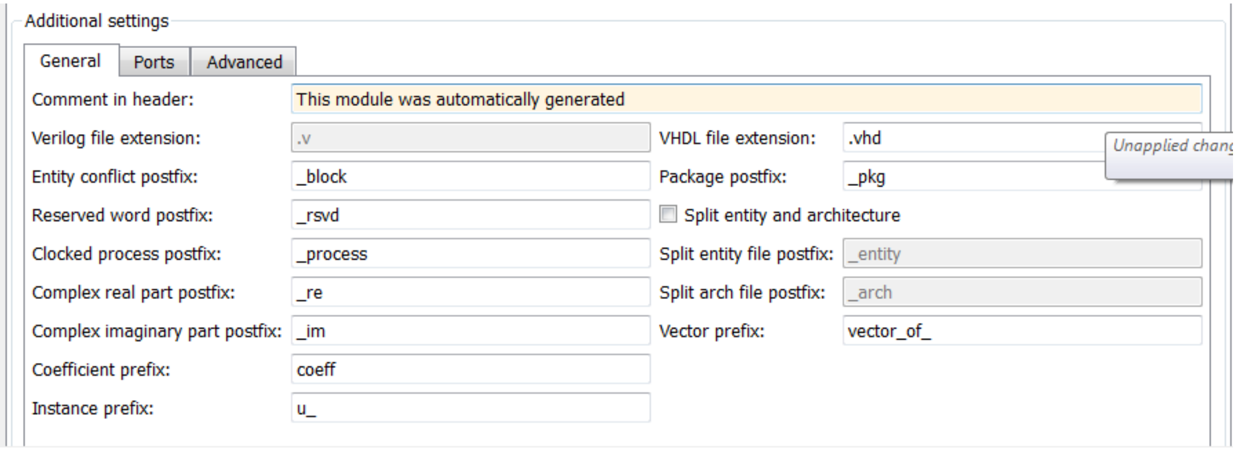
```
-- --------------------------------------------------------------
--
-- Module: Hlp
--
-- Generated by MATLAB(R) 7.11 and the Filter Design HDL Coder 2.7.
--
-- Generated on: 2010-08-31 13:32:16
--
-- This module was automatically generated
--
-- --------------------------------------------------------------

-- --------------------------------------------------------------
-- 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 dialog box.

2 Select the **General** tab in the **Additional settings** pane.

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



Additional settings

Comment in header: This module was automatically generated

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

□ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

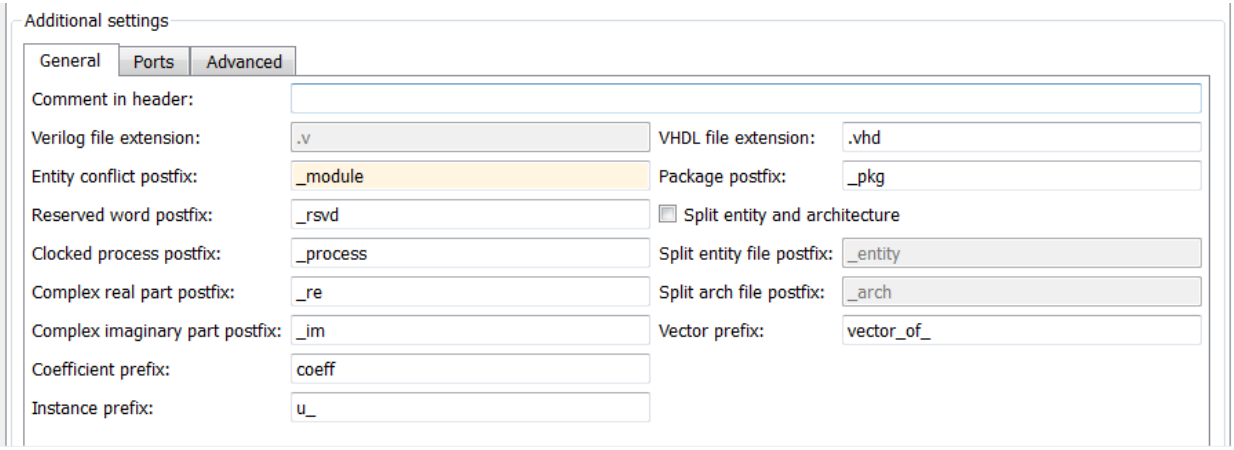
**Command-Line Alternative:** Use the `generatehdl` function with the property UserComment to add a comment string to the end of the header comment block in each generated HDL file.

## Resolving Entity or Module Name Conflicts

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

To change the postfix string:

1. Select the **Global Settings** tab on the Generate HDL dialog box.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new string in the **Entity conflict postfix** field, as shown in the following figure.



Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _module

Package postfix: _pkg

Reserved word postfix: _rsvd

☐ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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

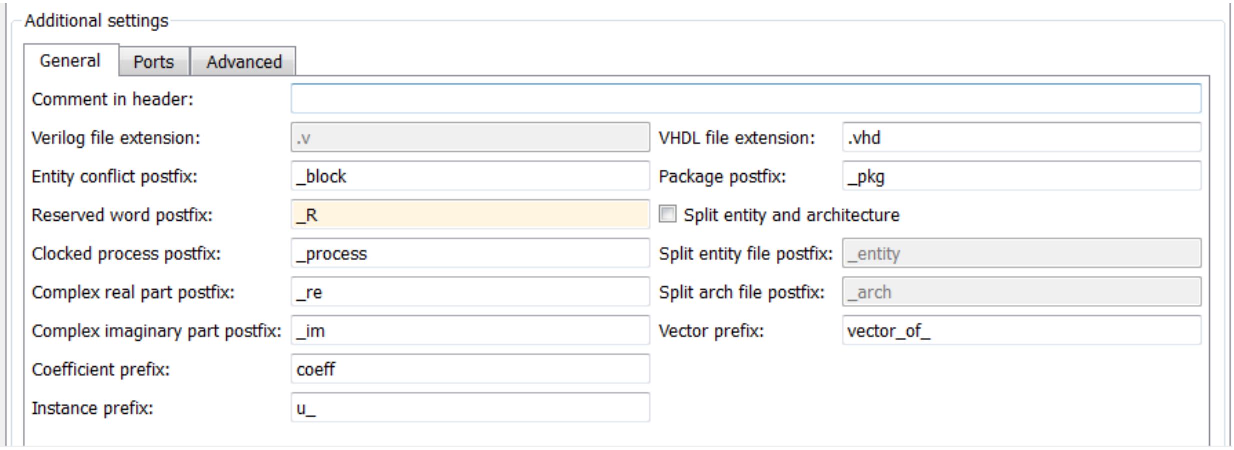
## Resolving HDL Reserved Word Conflicts

The coder checks whether strings that you specify as names, postfix values, or labels are VHDL or Verilog reserved words. See "Reserved Word Tables" on page 5-12 for listings of VHDL and Verilog reserved words.

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

To change the postfix string:

1. Select the **Global Settings** tab on the Generate HDL dialog box.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new string in the **Reserved word postfix** field, as shown in the following figure.



Additional settings

Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _R

☐ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_

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

### Reserved Word Tables

The following tables list VHDL and Verilog reserved words.

#### VHDL Reserved Words

| | | | | |
|---|---|---|---|---|
| `abs` | `access` | `after` | `alias` | `all` |
| `and` | `architecture` | `array` | `assert` | `attribute` |
| `begin` | `block` | `body` | `buffer` | `bus` |
| `case` | `component` | `configuration` | `constant` | `disconnect` |
| `downto` | `else` | `elsif` | `end` | `entity` |
| `exit` | `file` | `for` | `function` | `generate` |
| `generic` | `group` | `guarded` | `if` | `impure` |
| `in` | `inertial` | `inout` | `is` | `label` |
| `library` | `linkage` | `literal` | `loop` | `map` |
| `mod` | `nand` | `new` | `next` | `nor` |
| `not` | `null` | `of` | `on` | `open` |
| `or` | `others` | `out` | `package` | `port` |

| | | | | |
|---|---|---|---|---|
| postponed | procedure | process | pure | range |
| record | register | reject | rem | report |
| return | rol | ror | select | severity |
| signal | shared | sla | sll | sra |
| srl | subtype | then | to | transport |
| type | unaffected | units | until | use |
| variable | wait | when | while | with |
| xnor | xor | | | |

### Verilog Reserved Words

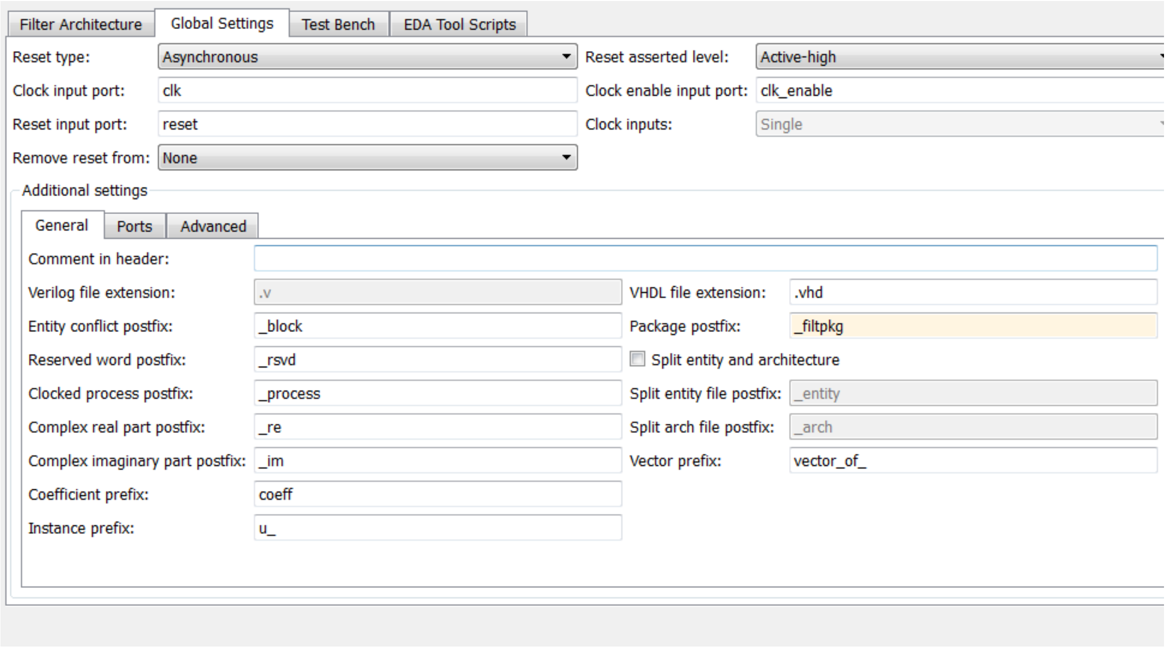
| | | | | |
|---|---|---|---|---|
| always | and | assign | automatic | begin |
| buf | bufif0 | bufif1 | case | casex |
| casez | cell | cmos | config | deassign |
| default | defparam | design | disable | edge |
| else | end | endcase | endconfig | endfunction |
| endgenerate | endmodule | endprimitive | endspecify | endtable |
| endtask | event | for | force | forever |
| fork | function | generate | genvar | highz0 |
| highz1 | if | ifnone | incdir | include |
| initial | inout | input | instance | integer |
| join | large | liblist | library | localparam |
| macromodule | medium | module | nand | negedge |
| nmos | nor | noshowcancelled | not | notif0 |
| notif1 | or | output | parameter | pmos |
| posedge | primitive | pull0 | pull1 | pulldown |
| pullup | pulsestyle_onevent | pulsestyle_ondetect | rcmos | real |
| realtime | reg | release | repeat | rnmos |
| rpmos | rtran | rtranif0 | rtranif1 | scalared |
| showcancelled | signed | small | specify | specparam |
| strong0 | strong1 | supply0 | supply1 | table |

```
task      time      tran      tranif0   tranif1
tri       tri0      tri1      triand    trior
trireg    unsigned  use       vectored  wait
wand      weak0     weak1     while     wire
wor       xnor      xor
```

## Setting the Postfix String for VHDL Package Files

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

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



Reset type: Asynchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Remove reset from: None

Additional settings

Comment in header:
Verilog file extension: .v
VHDL file extension: .vhd
Entity conflict postfix: _block
Package postfix: _filtpkg
Reserved word postfix: _rsvd
□ Split entity and architecture
Clocked process postfix: _process
Split entity file postfix: _entity
Complex real part postfix: _re
Split arch file postfix: _arch
Complex imaginary part postfix: _im
Vector prefix: vector_of_
Coefficient prefix: coeff
Instance prefix: u_

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

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

## Specifying a Prefix for Filter Coefficients

The coder declares 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.

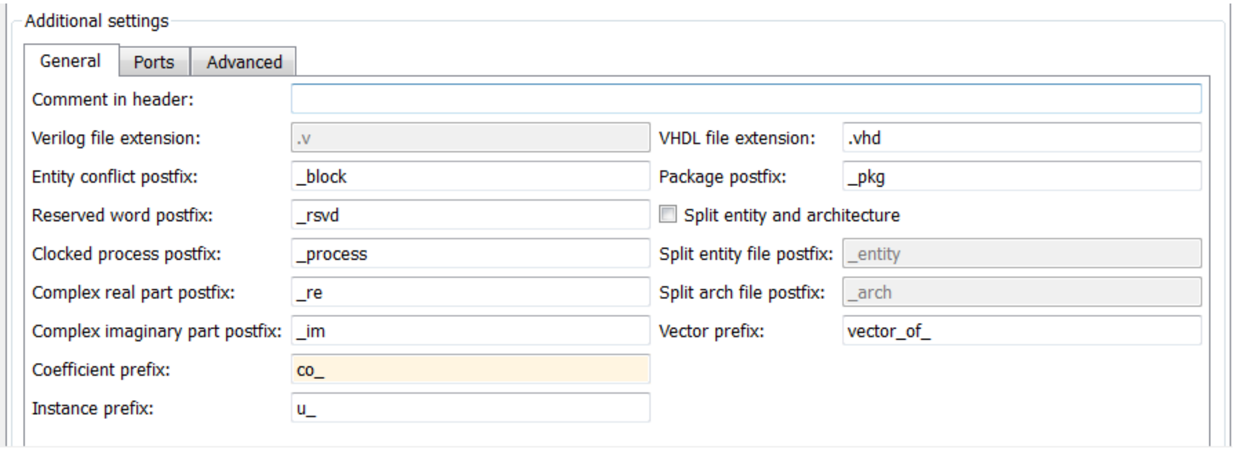
| For... | The Prefix Is Concatenated with... |
|---|---|
| FIR filters | Each coefficient number, starting with 1.<br>Examples: `coeff1`, `coeff22` |
| IIR filters | An underscore (_) and an `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) followed by the string `_section`*n*, where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

For example:

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

To use a prefix other than `coeff`,

1. Select the **Global Settings** tab on the Generate HDL dialog box.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new string in the **Coefficient prefix** field, as shown in the following figure.



Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

□ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: co_

Instance prefix: u_

The string that you specify

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

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

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

## Specifying a Postfix String for Process Block Labels

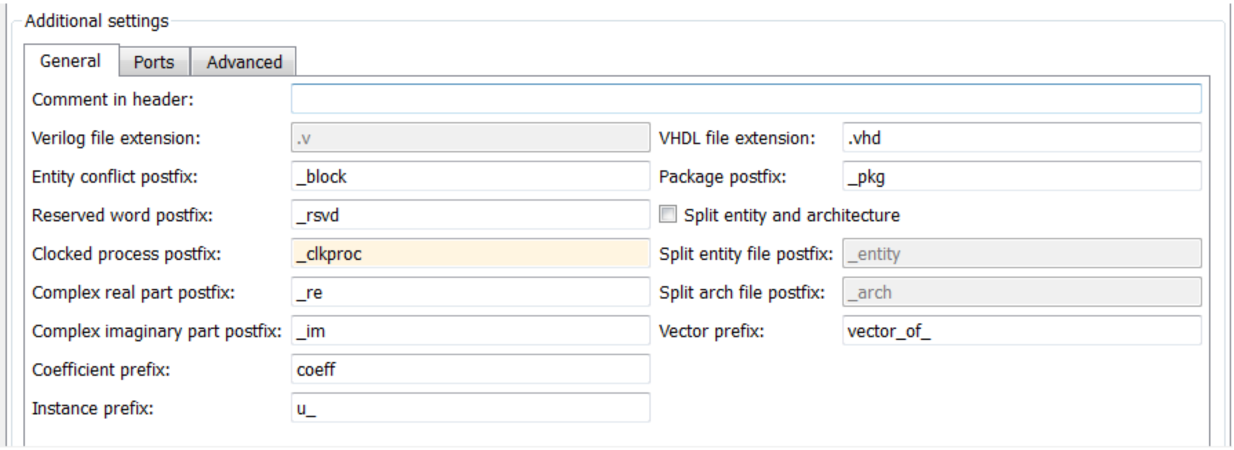
The coder generates process blocks to modify the content of the registers. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, the coder derives the label `delay_pipeline_process` in the following block from the register name `delay_pipeline` and the postfix string `_process`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
```

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

The **Clocked process postfix** property lets you change the postfix string to a value other than `_process`. For example, to change the postfix string to `_clkproc`, do the following:

1. Select the **Global Settings** tab on the Generate HDL dialog box.
2. Select the **General** tab in the **Additional settings** pane.
3. Enter a new string in the **Clocked process postfix** field, as shown in the following figure.



| | | | |
|---|---|---|---|
| Comment in header: | | | |
| Verilog file extension: | .v | VHDL file extension: | .vhd |
| Entity conflict postfix: | _block | Package postfix: | _pkg |
| Reserved word postfix: | _rsvd | □ Split entity and architecture | |
| Clocked process postfix: | _clkproc | Split entity file postfix: | _entity |
| Complex real part postfix: | _re | Split arch file postfix: | _arch |
| Complex imaginary part postfix: | _im | Vector prefix: | vector_of_ |
| Coefficient prefix: | coeff | | |
| Instance prefix: | u_ | | |

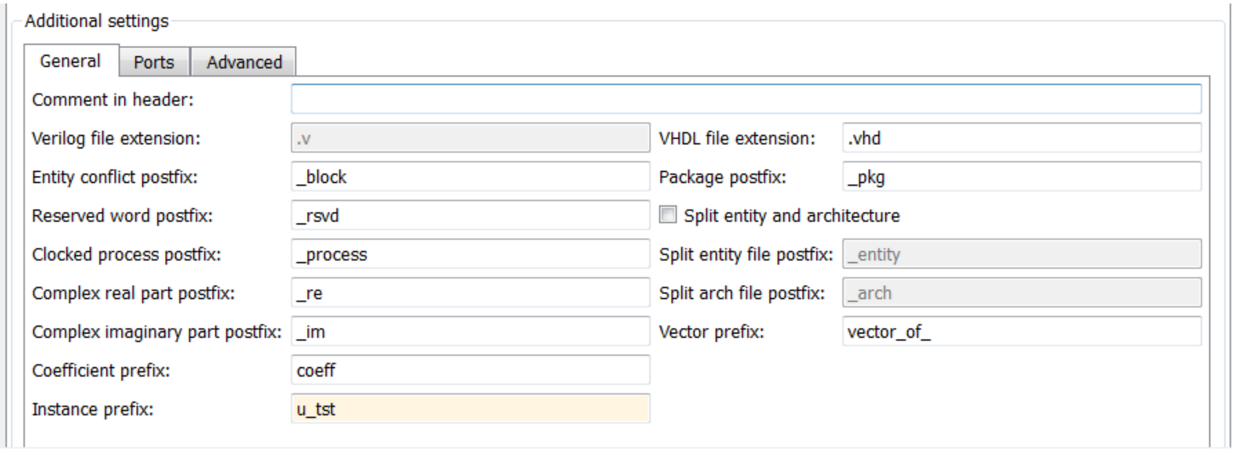
**Command-Line Alternative:** Use the `generatehdl` function with the property ClockProcessPostfix to change the postfix string appended to process labels.

## Setting a Prefix for Component Instance Names

**Instance prefix** specifies a string to be prefixed to component instance names in generated code. The default string is `u_`.

You can of set the postfix string to a value other than u_. To change the string:

1 Select the **Global Settings** tab on the Generate HDL dialog box.
2 Select the **General** tab in the **Additional settings** pane.
3 Enter a new string in the **Instance prefix** field, as shown in the following figure.



Additional settings

Comment in header:
Verilog file extension: .v
VHDL file extension: .vhd
Entity conflict postfix: _block
Package postfix: _pkg
Reserved word postfix: _rsvd
□ Split entity and architecture
Clocked process postfix: _process
Split entity file postfix: _entity
Complex real part postfix: _re
Split arch file postfix: _arch
Complex imaginary part postfix: _im
Vector prefix: vector_of_
Coefficient prefix: coeff
Instance prefix: u_tst

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

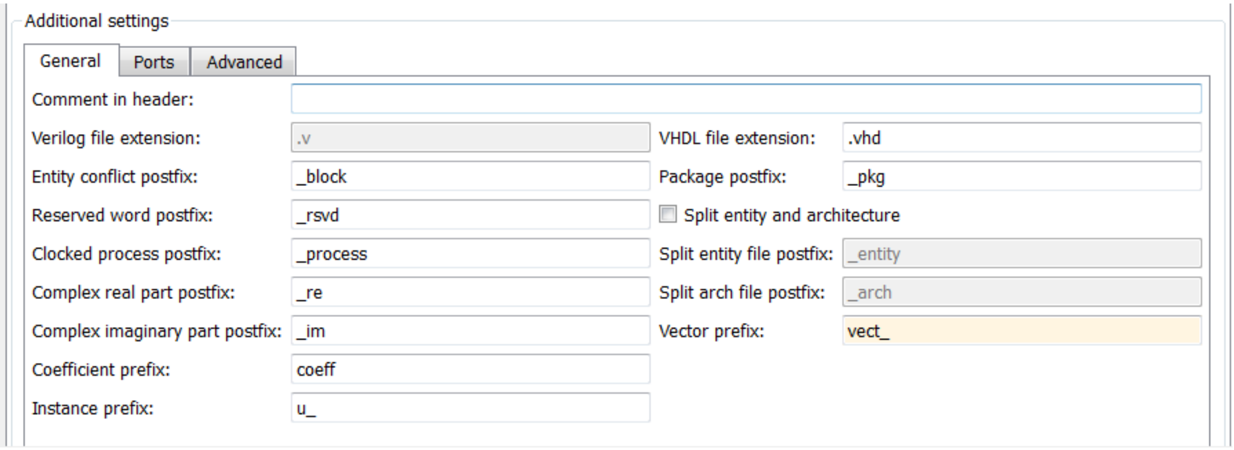
## Setting a Prefix for Vector Names

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

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

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

1 Select the **Global Settings** tab on the Generate HDL dialog box.
2 Select the **General** tab in the **Additional settings** pane.
3 Enter a new string in the **Vector prefix** field, as shown in the following figure.



Additional settings

Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

□ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vect_

Coefficient prefix: coeff

Instance prefix: u_

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

# Ports and Resets

**In this section...**

## Naming HDL Ports

The default names for filter HDL ports are as follows:

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

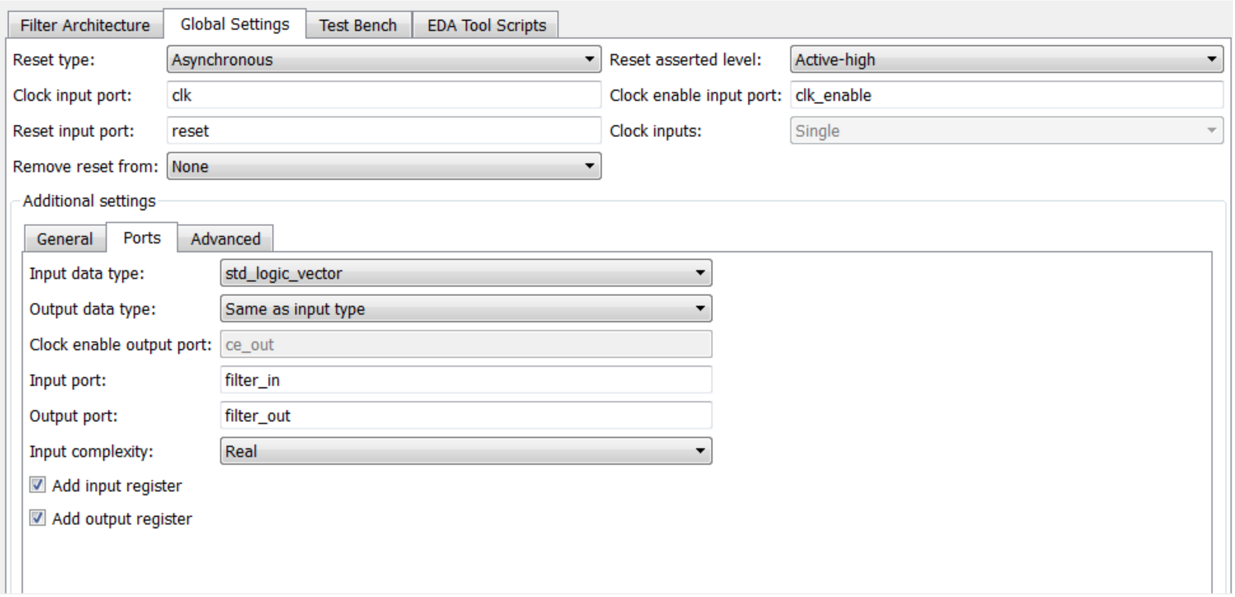
For example, the default VHDL declaration for entity `Hd` looks like the following.

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

To change port names,

1 Select the **Global Settings** tab on the Generate HDL dialog box.

2 Select the **Ports** tab in the **Additional settings** pane. The following figure highlights the port name fields for **Input port**, **Output port**, **Clock input port**, **Reset input port**, and **Clock enable output port**.



Reset type: Asynchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

Output port: filter_out

Input complexity: Real

☑ Add input register

☑ Add output register

3 Enter new strings 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,

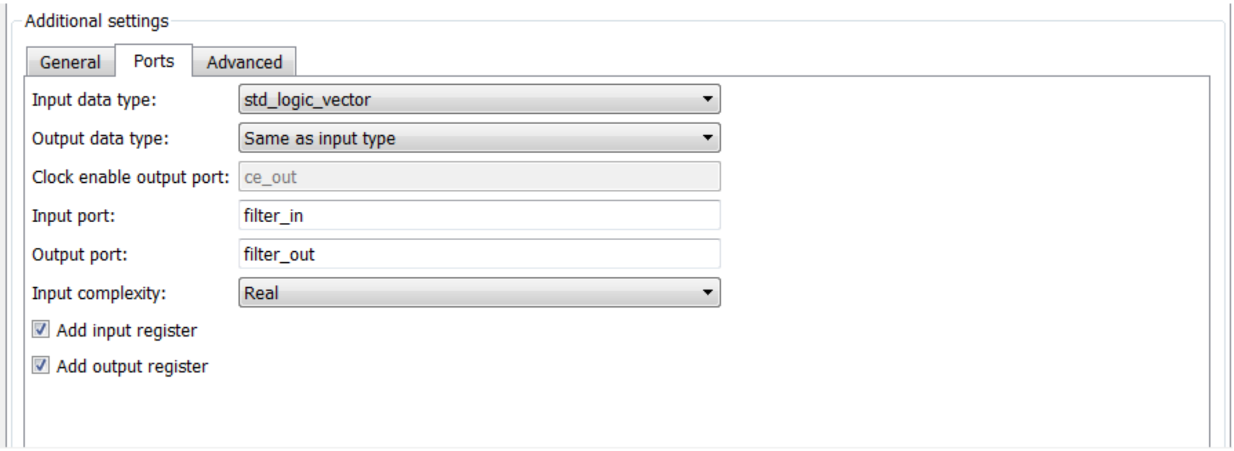
1 Select the **Global Settings** tab on the Generate HDL dialog box.

2 Select the **Ports** tab in the **Additional settings** pane.

3 Select a data type from the **Input data type** or **Output data type** menu identified in the following figure.

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

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



Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

Output port: filter_out

Input complexity: Real

☑ Add input register

☑ Add output register

---

**Note:** 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.

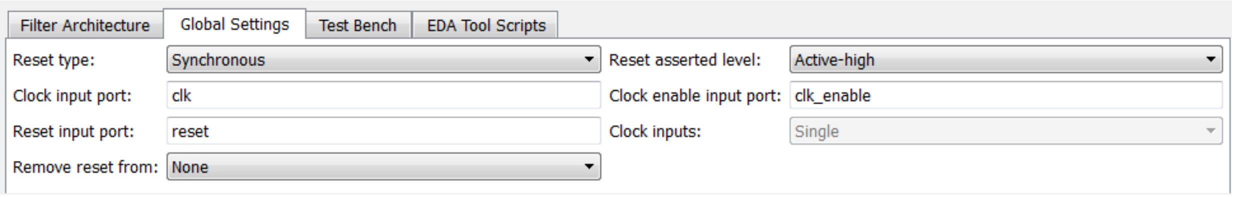
## Selecting Asynchronous or Synchronous Reset Logic

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.

The following 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 dialog box.



Reset type: Synchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

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

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

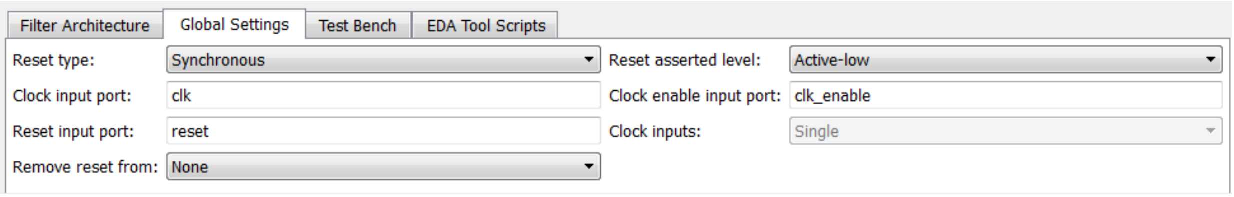
**Command-Line Alternative:** Use the `generatehdl` function with the property ResetType to set the reset style for the registers in the generated HDL code.

## 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, the following code fragment checks whether reset is active high before populating the delay_pipeline register:

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

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



Reset type: Synchronous
Reset asserted level: Active-low
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Remove reset from: None

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

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

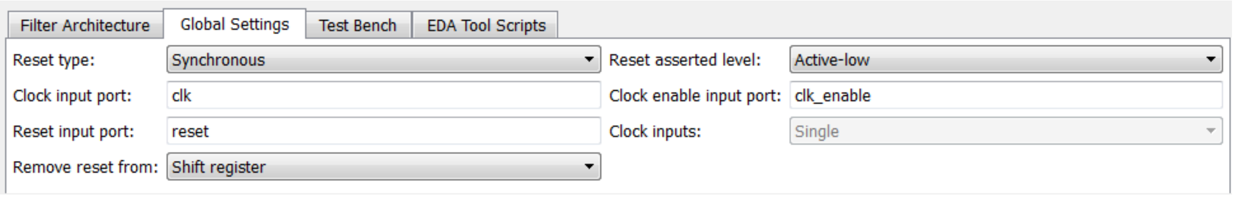
**Note:** The **Reset asserted level** setting also determines the reset level for test bench reset input signals.

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

## Suppressing Generation of Reset Logic

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

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



Reset type: Synchronous

Reset asserted level: Active-low

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: Shift register

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

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

# HDL Constructs

In this section...

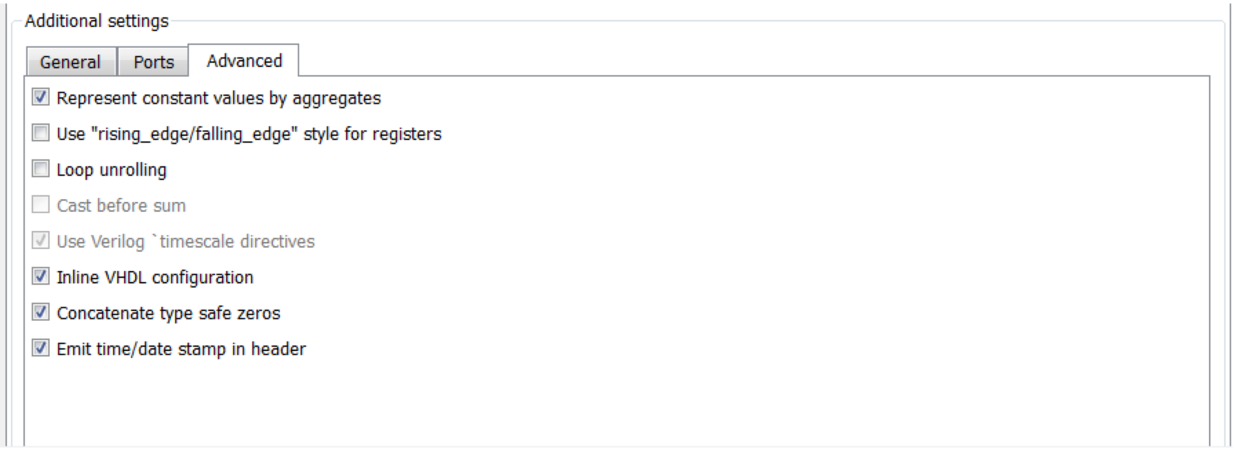
## Representing VHDL Constants with Aggregates

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. The following VHDL constant declarations are examples of declarations generated by default for values less than 32 bits:

```
CONSTANT coeff1: signed(15 DOWNTO 0) := to_signed(-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 dialog box.
2. Select the **Advanced** tab.
3. Select **Represent constant values by aggregates**, as shown the following figure.



☑ Represent constant values by aggregates

□ Use "rising_edge/falling_edge" style for registers

□ Loop unrolling

□ Cast before sum

☑ Use Verilog `timescale directives

☑ Inline VHDL configuration

☑ Concatenate type safe zeros

☑ Emit time/date stamp in header

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');
```

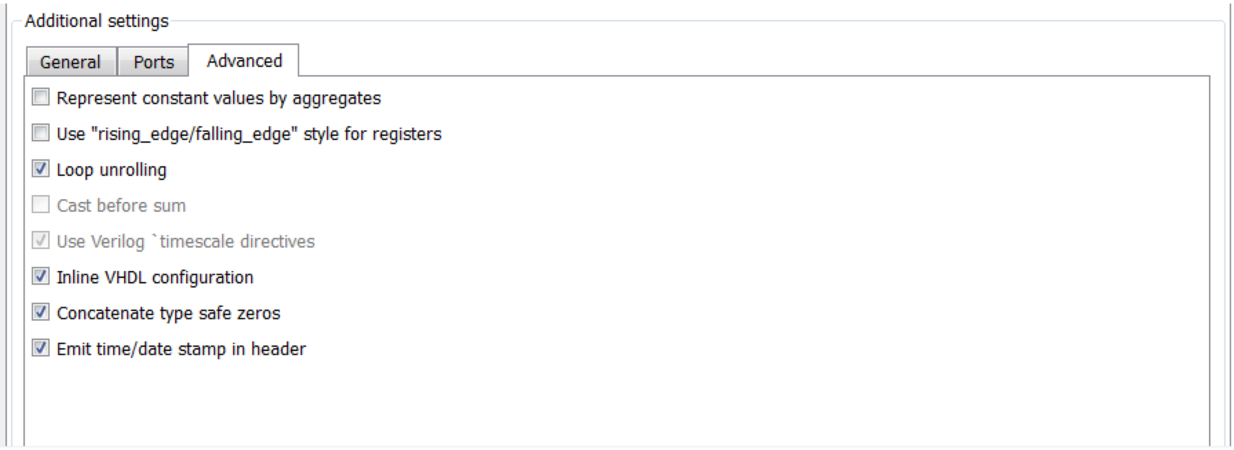
**Command-Line Alternative:** Use the `generatehdl` function with the property UseAggregatesForConst to represent constants in the HDL code as aggregates.

## 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 dialog box.
2. Select the **Advanced** tab. The **Advanced** pane appears.
3. Select **Loop unrolling**, as shown in the following figure.



☐ Represent constant values by aggregates
☐ Use "rising_edge/falling_edge" style for registers
☑ Loop unrolling
☐ Cast before sum
☑ Use Verilog `timescale directives
☑ Inline VHDL configuration
☑ Concatenate type safe zeros
☑ Emit time/date stamp in header

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

## Using the VHDL rising_edge Function

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

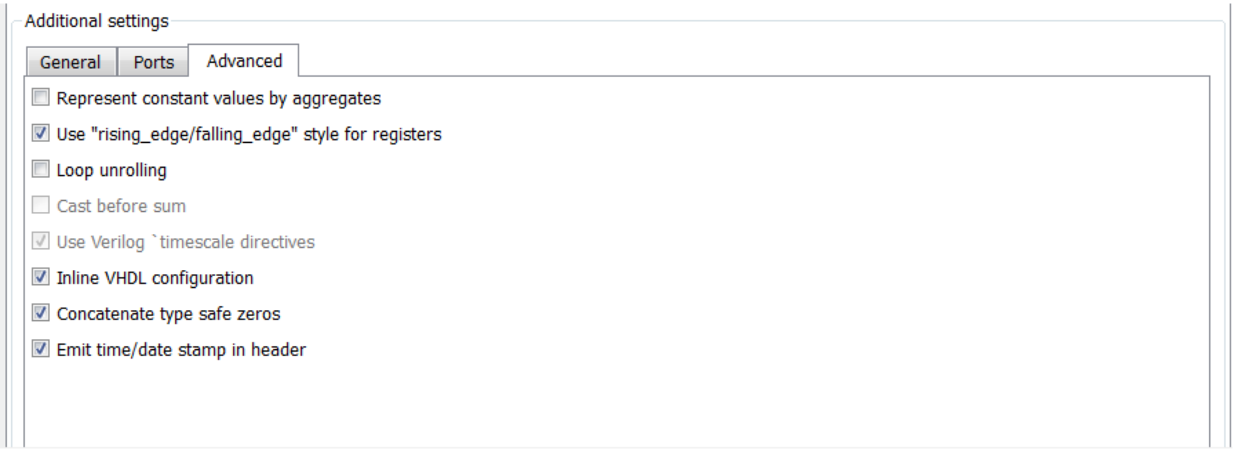
```
Delay_Pipeline_Process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    delay_pipeline(0 TO 50) <= (OTHERS => (OTHERS => '0'));
  ELSEIF clk'event AND clk = '1' THEN
    IF clk_enable = '1' THEN
      delay_pipeline(0) <= signed(filter_in);
  delay_pipeline(1 TO 50) <= delay_pipeline(0 TO 49);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

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

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

To use the `rising_edge` function,

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



Additional settings

☐ Represent constant values by aggregates

☑ Use "rising_edge/falling_edge" style for registers

☐ Loop unrolling

☐ Cast before sum

☑ Use Verilog `timescale directives

☑ Inline VHDL configuration

☑ Concatenate type safe zeros

☑ Emit time/date stamp in header

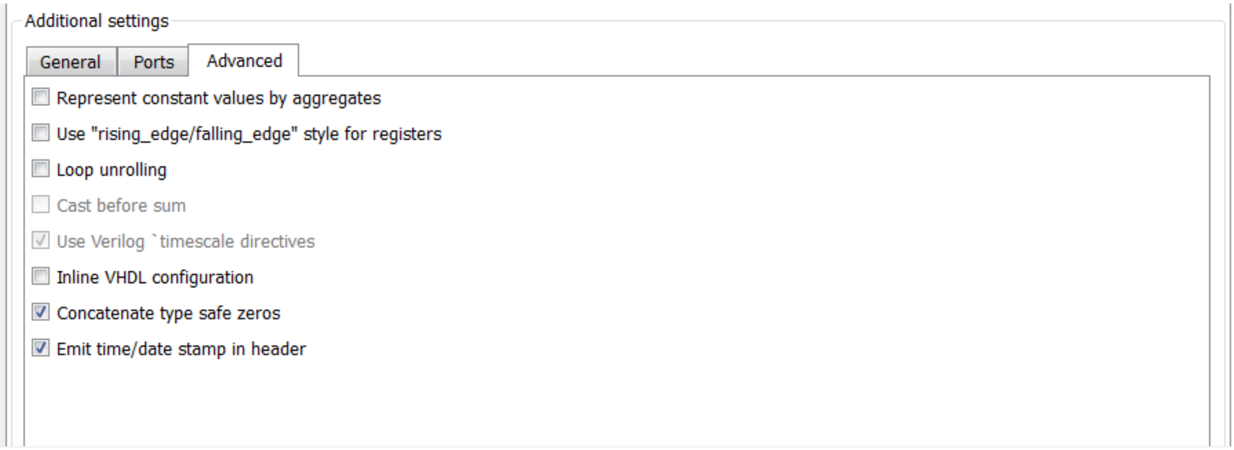
**Command-Line Alternative:** Use the `generatehdl` function with the property UseRisingEdge to use the VHDL `rising_edge` function to check for rising edges during register operations.

## 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 dialog box.
2. Select the **Advanced** tab. The **Advanced** pane appears.
3. Clear **Inline VHDL configuration**, as shown in the following figure.



Additional settings

General | Ports | Advanced

☐ Represent constant values by aggregates

☐ Use "rising_edge/falling_edge" style for registers

☐ Loop unrolling

☐ Cast before sum

☑ Use Verilog `timescale directives

☐ Inline VHDL configuration

☑ Concatenate type safe zeros

☑ Emit time/date stamp in header

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

## Specifying VHDL Syntax for Concatenated Zeros

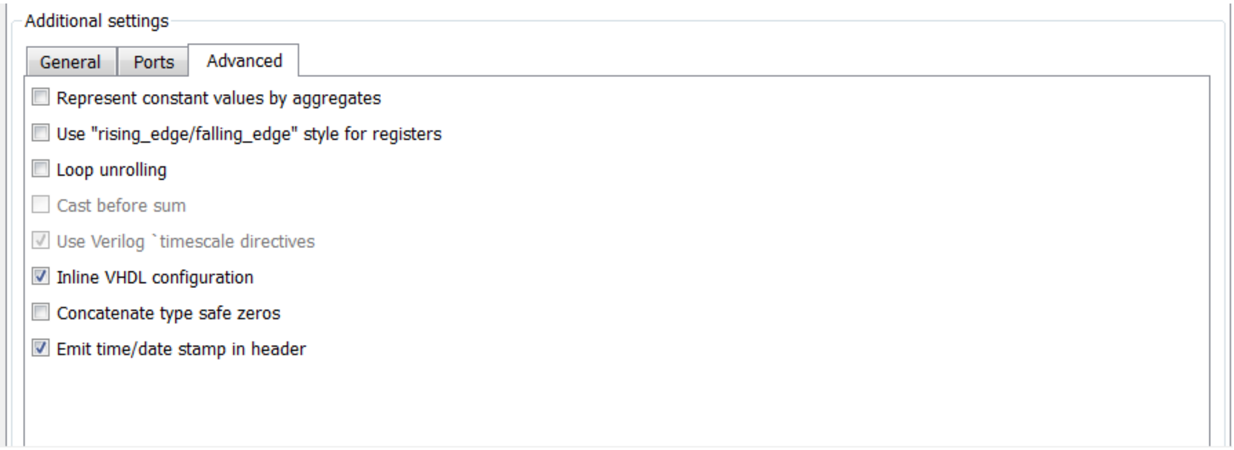
In VHDL, the concatenation of zeros can be represented in two syntax forms. One form, `'0' & '0'`, is type-safe. This syntax is the default. The alternative syntax, `"000000..."`, can be easier to read and is more compact, but can lead to ambiguous types.

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

1 Select the **Global Settings** tab on the Generate HDL dialog box.

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

3 Clear **Concatenate type safe zeros**, as shown in the following figure.



□ Represent constant values by aggregates
□ Use "rising_edge/falling_edge" style for registers
□ Loop unrolling
□ Cast before sum
☑ Use Verilog `timescale directives
☑ Inline VHDL configuration
□ Concatenate type safe zeros
☑ Emit time/date stamp in header

**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

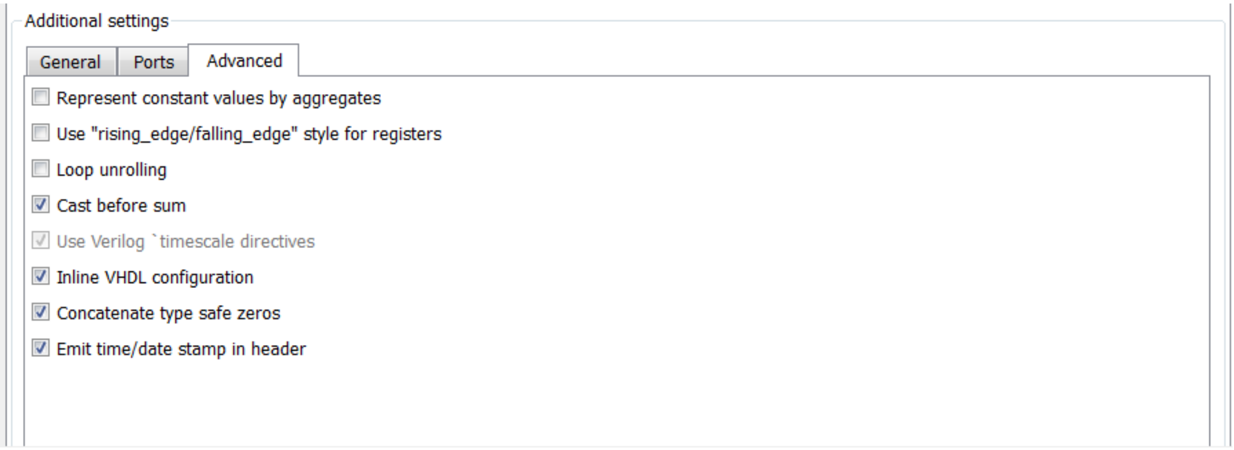
By default, generated HDL code operates on input data using data types as specified by the filter design, and then converts the result to the specified result type.

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 dialog box.

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

3 Select **Cast before sum**, as shown in the following figure.



Additional settings

General | Ports | Advanced

☐ Represent constant values by aggregates

☐ Use "rising_edge/falling_edge" style for registers

☐ Loop unrolling

☑ Cast before sum

☑ Use Verilog `timescale directives

☑ Inline VHDL configuration

☑ Concatenate type safe zeros

☑ Emit time/date stamp in header

**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 FDATool

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

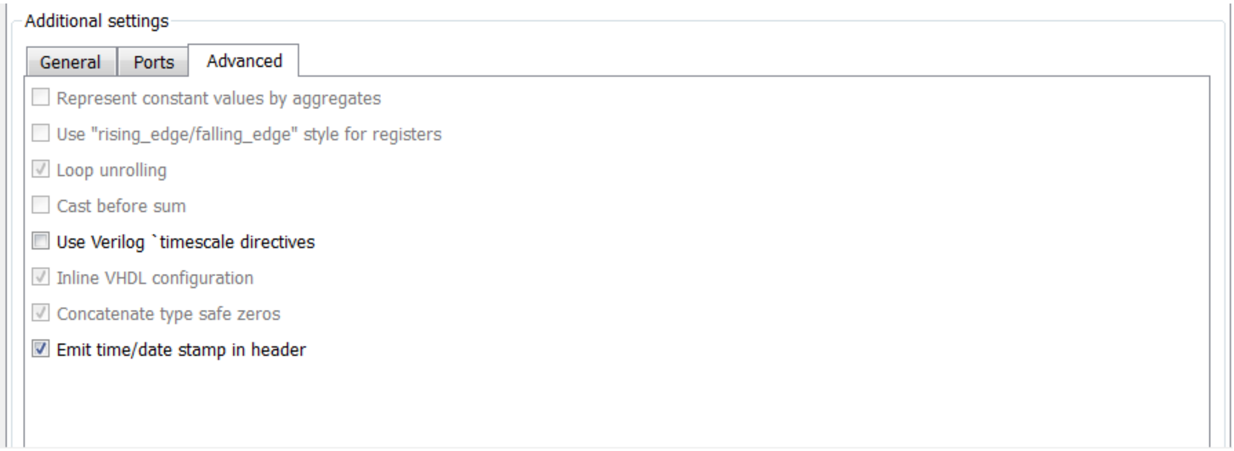
- Some filter object types do not have the **Cast signals before sum** property. For such filter objects, **Cast before sum** is effectively off when HDL code is generated; it is not relevant to the filter.
- Where the filter object does have the **Cast signals before sum** property, the coder by default follows the setting of **Cast signals before sum** in the filter object. This setting is visible in the GUI. 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 FDATool.

## Suppressing Verilog Time Scale Directives

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

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

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



Additional settings

□ Represent constant values by aggregates
□ Use "rising_edge/falling_edge" style for registers
☑ Loop unrolling
□ Cast before sum
□ Use Verilog `timescale directives
☑ Inline VHDL configuration
☑ Concatenate type safe zeros
☑ Emit time/date stamp in header

**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 the following code example:

```
Hd = design(fdesign.lowpass,'equiripple','Filterstructure','dffir')
```

```
generatehdl(Hd,'InputComplex','on')
```

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

```
ENTITY Hd IS
   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         filter_in_re                    :   IN    real; -- double
         filter_in_im                    :   IN    real; -- double
         filter_out_re                   :   OUT   real; -- double
         filter_out_im                   :   OUT   real  -- double
         );

END Hd;
```

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

### Setting the Port Name Postfix for Complex Ports

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

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

# Verification of Generated HDL Filter Code

# Testing with an HDL Test Bench

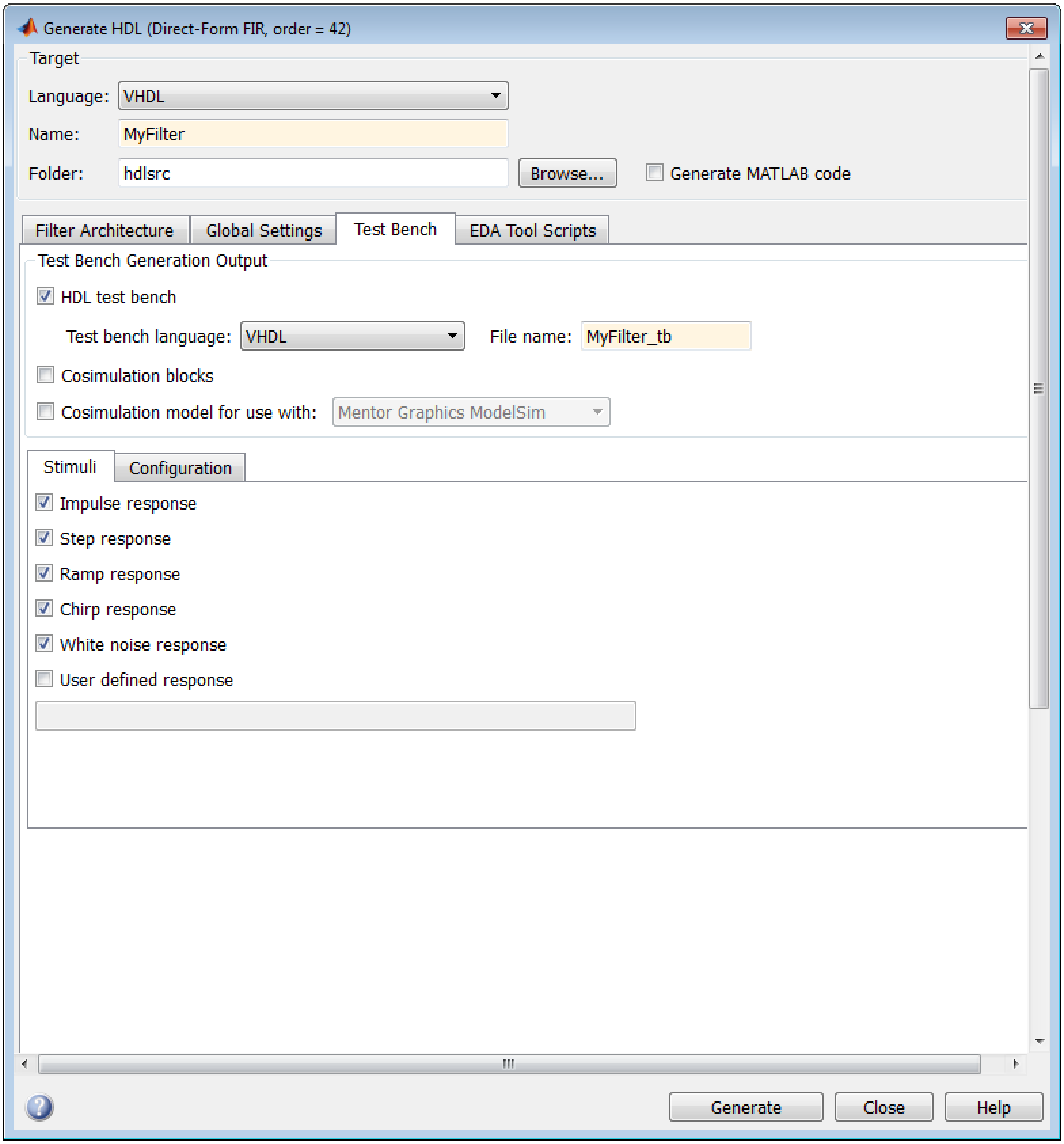
**In this section...**

## Workflow for Testing with an HDL Test Bench

### Generating the Filter and Test Bench HDL Code

Use the Filter Design HDL Coder GUI or command-line interface to generate the HDL code for your filter design and test bench. The GUI generates a VHDL or Verilog test bench file, depending on your language selection for the generated HDL code. To specify a different test bench language, select the **Test bench language** option in the **Test Bench** pane of the Generate HDL dialog box. There is no command-line option for generating the test bench in a different language than the generated HDL code.

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



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

Target

Language: VHDL

Name: MyFilter

Folder: hdlsrc

□ Generate MATLAB code

Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: MyFilter_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

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

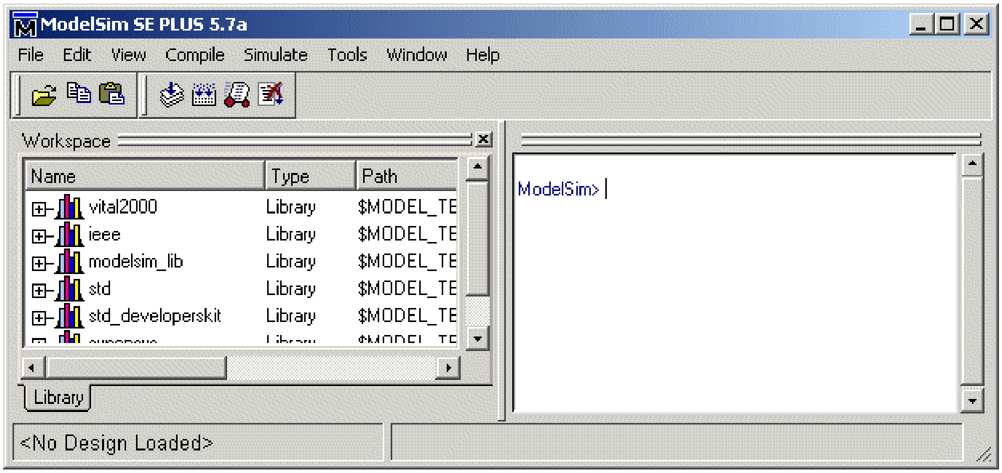
```
### Starting VHDL code generation process for filter: MyFilter
### Generating: C:\Work\sl_hdlcoder_work\hdlsrc\MyFilter.vhd
### Starting generation of MyFilter VHDL entity
### Starting generation of MyFilter VHDL architecture
### HDL latency is 2 samples
### Successful completion of VHDL code generation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



| Name | Type | Path |
|---|---|---|
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |

```
ModelSim> |
```

<No Design Loaded>

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.

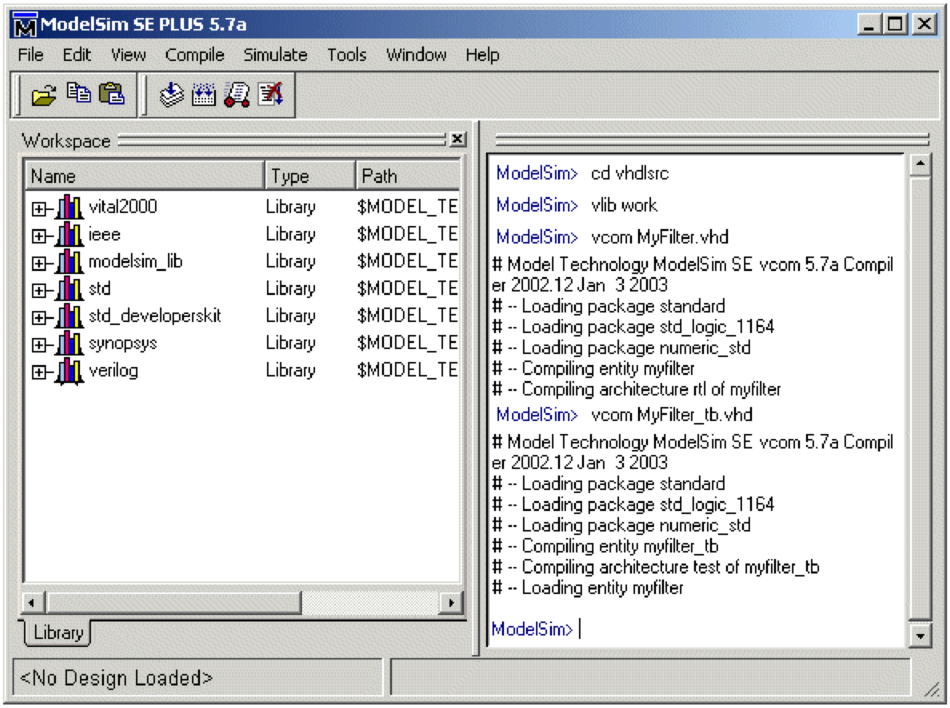
The following Mentor Graphics ModelSim command sequence changes the current folder to `hdlsrc`, creates the design library `work`, and compiles VHDL filter and filter test bench code. The `vlib` command creates the design library work and the `vcom` commands initiate the compilations.

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

---

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

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



| Name | Type | Path |
|---|---|---|
| vital2000 | Library | $MODEL_TE |
| ieee | Library | $MODEL_TE |
| modelsim_lib | Library | $MODEL_TE |
| std | Library | $MODEL_TE |
| std_developerskit | Library | $MODEL_TE |
| synopsys | Library | $MODEL_TE |
| verilog | Library | $MODEL_TE |

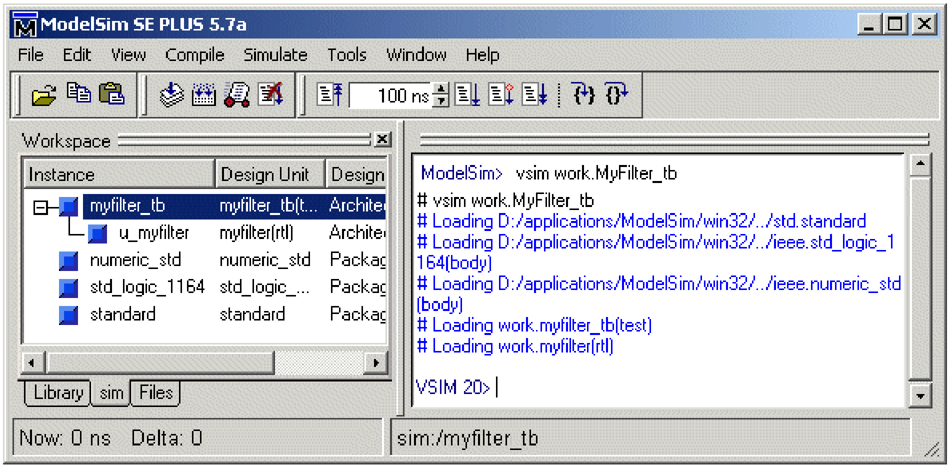
```
ModelSim> cd vhdlsrc
ModelSim> vlib work
ModelSim> vcom MyFilter.vhd
# Model Technology ModelSim SE vcom 5.7a Compil
er 2002.12 Jan  3 2003
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity myfilter
# -- Compiling architecture rtl of myfilter
ModelSim> vcom MyFilter_tb.vhd
# Model Technology ModelSim SE vcom 5.7a Compil
er 2002.12 Jan  3 2003
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package numeric_std
# -- Compiling entity myfilter_tb
# -- Compiling architecture test of myfilter_tb
# -- Loading entity myfilter

ModelSim> |
```

### 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
```

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



| Instance | Design Unit | Design |
|---|---|---|
| myfilter_tb | myfilter_tb(t... | Archite |
| u_myfilter | myfilter(rtl) | Archite |
| numeric_std | numeric_std | Packag |
| std_logic_1164 | std_logic_... | Packag |
| standard | standard | Packag |

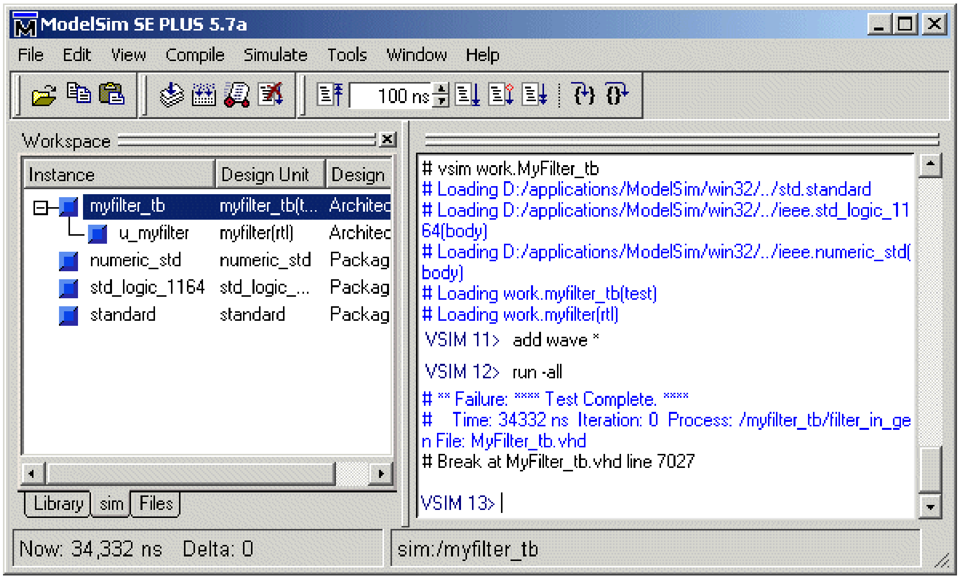
```
ModelSim> vsim work.MyFilter_tb
# vsim work.MyFilter_tb
# Loading D:/applications/ModelSim/win32/../std.standard
# Loading D:/applications/ModelSim/win32/../ieee.std_logic_1
164(body)
# Loading D:/applications/ModelSim/win32/../ieee.numeric_std
(body)
# Loading work.myfilter_tb(test)
# Loading work.myfilter(rtl)

VSIM 20>
```

```
Now: 0 ns   Delta: 0        sim:/myfilter_tb
```

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.

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



| Instance | Design Unit | Design |
|---|---|---|
| myfilter_tb | myfilter_tb(t... | Architec |
| u_myfilter | myfilter(rtl) | Architec |
| numeric_std | numeric_std | Packag |
| std_logic_1164 | std_logic_... | Packag |
| standard | standard | Packag |

```
# vsim work.MyFilter_tb
# Loading D:/applications/ModelSim/win32/../std.standard
# Loading D:/applications/ModelSim/win32/../ieee.std_logic_11
64(body)
# Loading D:/applications/ModelSim/win32/../ieee.numeric_std(
body)
# Loading work.myfilter_tb(test)
# Loading work.myfilter(rtl)
VSIM 11> add wave *

VSIM 12> run -all
# ** Failure: **** Test Complete. ****
#    Time: 34332 ns  Iteration: 0  Process: /myfilter_tb/filter_in_ge
n File: MyFilter_tb.vhd
# Break at MyFilter_tb.vhd line 7027

VSIM 13> |
```

Now: 34,332 ns  Delta: 0    sim:/myfilter_tb

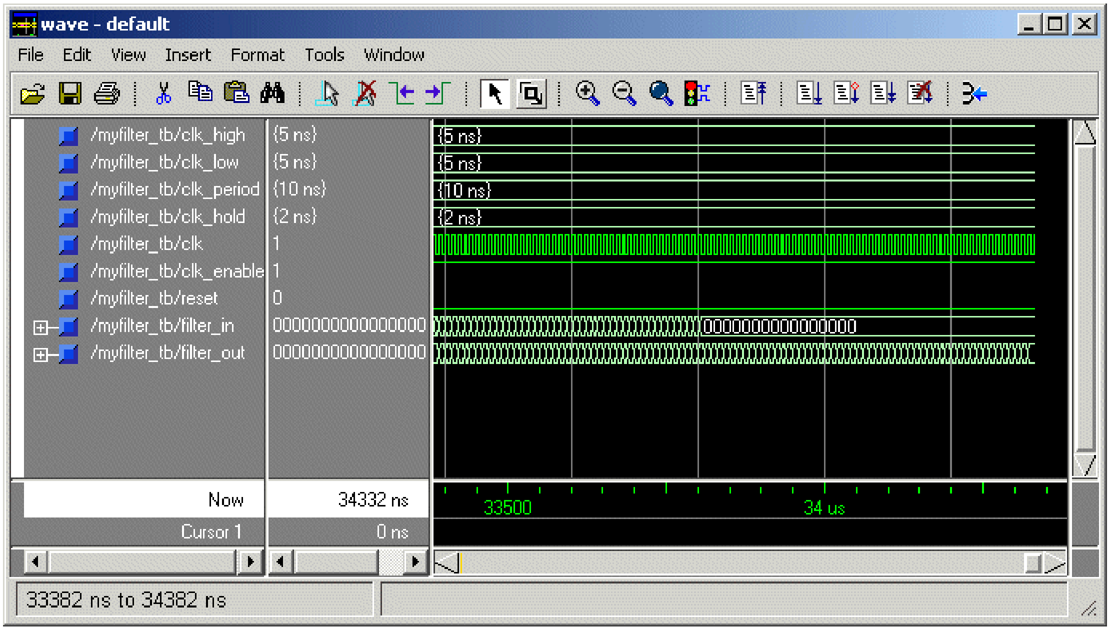
As your test bench simulation runs, watch for error messages. If error messages appear, interpret them as they pertain to your filter design and the code generation options you applied. For example, some HDL optimization options can produce numeric results that differ from the results produced by the 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 the following is returned:

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

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

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

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



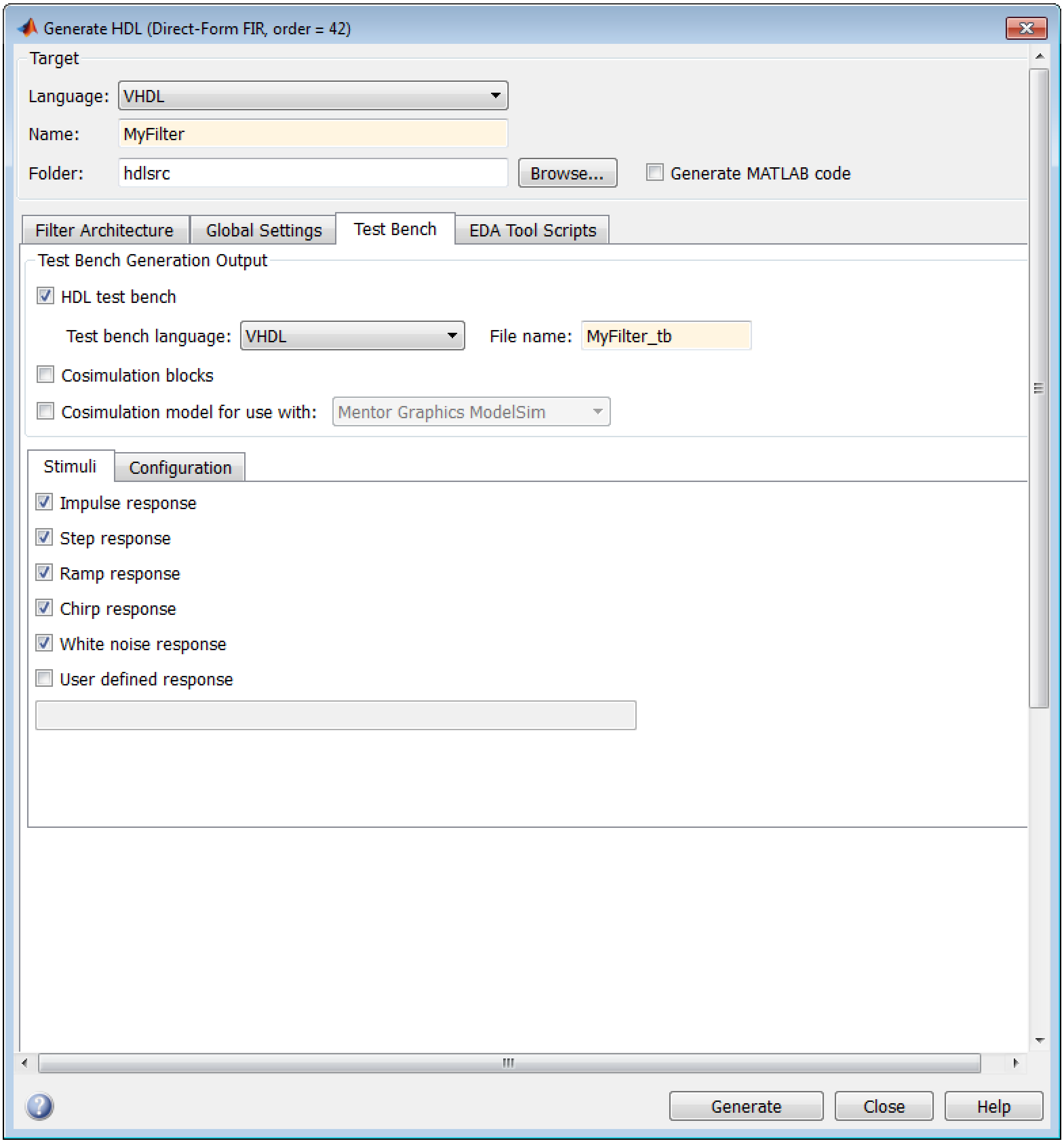
```
/myfilter_tb/clk_high      {5 ns}
/myfilter_tb/clk_low       {5 ns}
/myfilter_tb/clk_period    {10 ns}
/myfilter_tb/clk_hold      {2 ns}
/myfilter_tb/clk           1
/myfilter_tb/clk_enable    1
/myfilter_tb/reset         0
/myfilter_tb/filter_in     0000000000000000
/myfilter_tb/filter_out    0000000000000000

Now        34332 ns
Cursor 1   0 ns

33382 ns to 34382 ns
```

## Enabling Test Bench Generation

To enable generation of an HDL test bench:

1 Select the **Test Bench** pane in the Generate HDL dialog box.

2 Select the **HDL test bench** option, as shown in the following figure.



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

Target

Language: VHDL

Name: MyFilter

Folder: hdlsrc

□ Generate MATLAB code

Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: MyFilter_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

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.

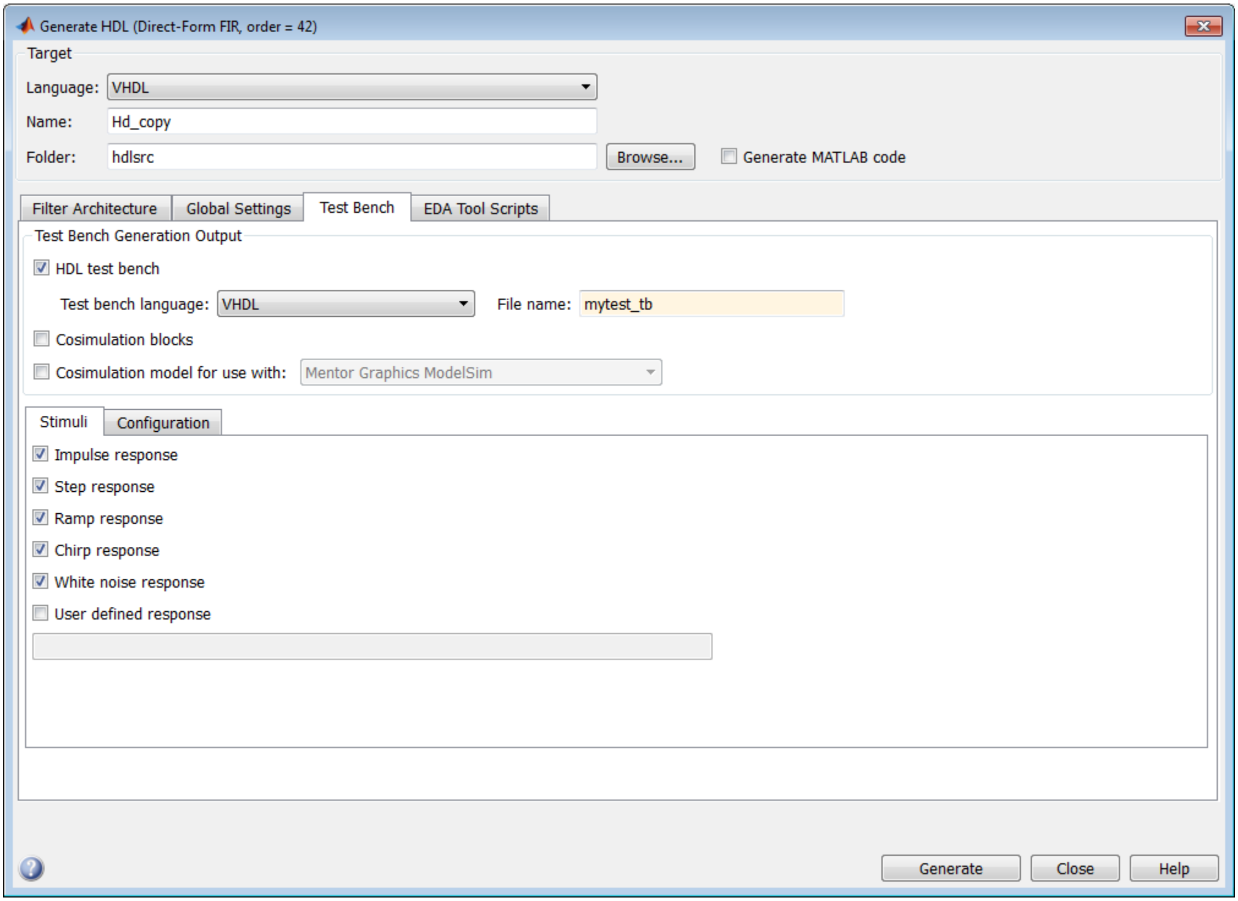
## Renaming the Test Bench

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 dialog box |
| VHDL file | Defined by the **VHDL file extension** field in the **Global Settings** pane of the Generate HDL dialog box |

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

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



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

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

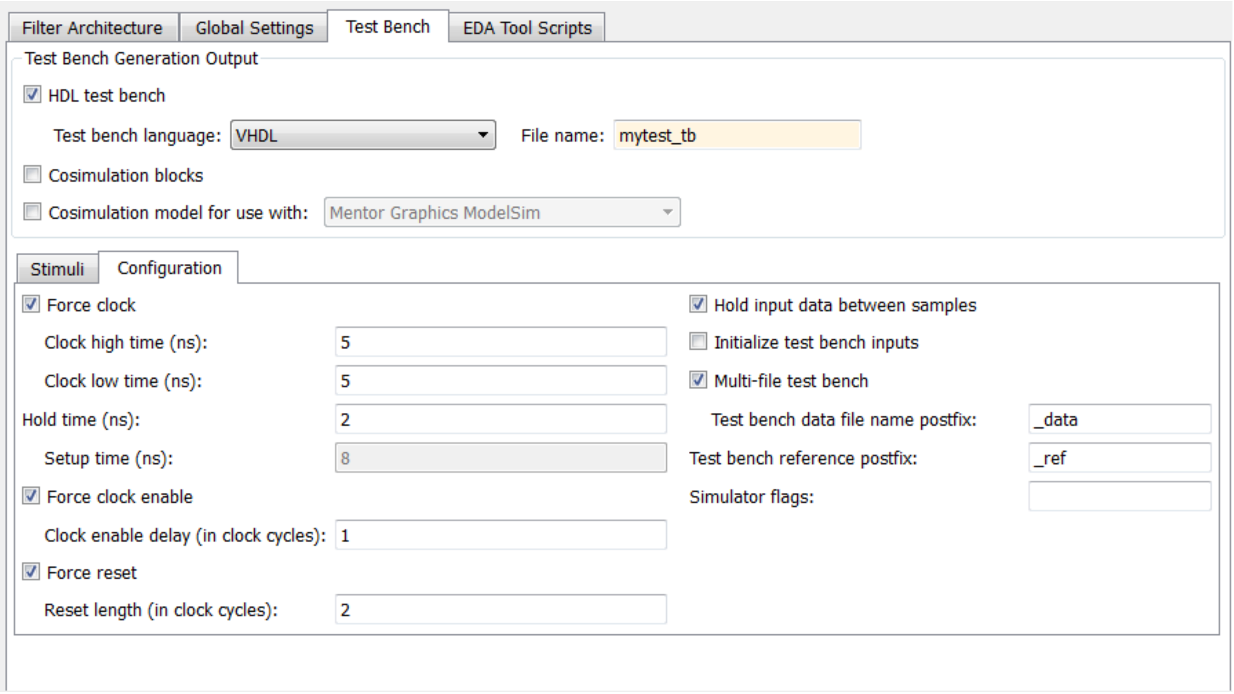
□ User defined response

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

**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 dialog box.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL    File name: mytest_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

☑ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Simulator flags:

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

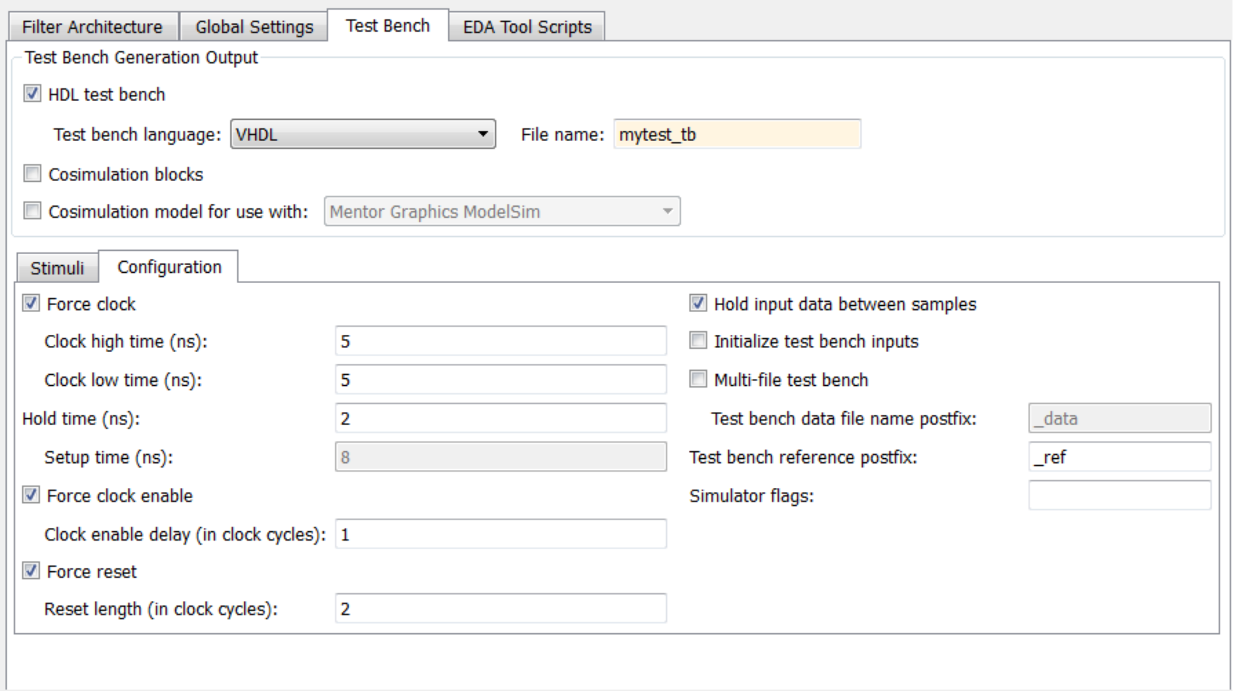
To change these clock configuration settings:

1. Click **Configuration** in the **Test bench** pane of the Generate HDL dialog box.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. Make the following configuration changes as described in the following table:

| If You Want to... | Then... |
|---|---|
| Disable the forcing of clock enable input signals | Clear **Force clock enable**. |
| Disable the forcing of clock input signals | Clear **Force clock**. |
| 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. |

| If You Want to... | Then... |
|---|---|
| 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. |

The following figure highlights the applicable options.



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

Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

Stimuli | Configuration

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Simulator flags:

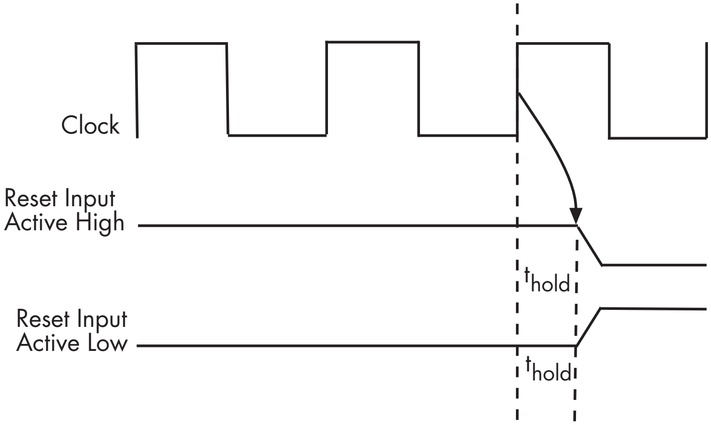
**Command-Line Alternative:** Use the `generatehdl` properties ForceClock, ClockHighTime, ForceClockEnable, and TestBenchClockEnableDelay to reconfigure the test bench clock.

## 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.



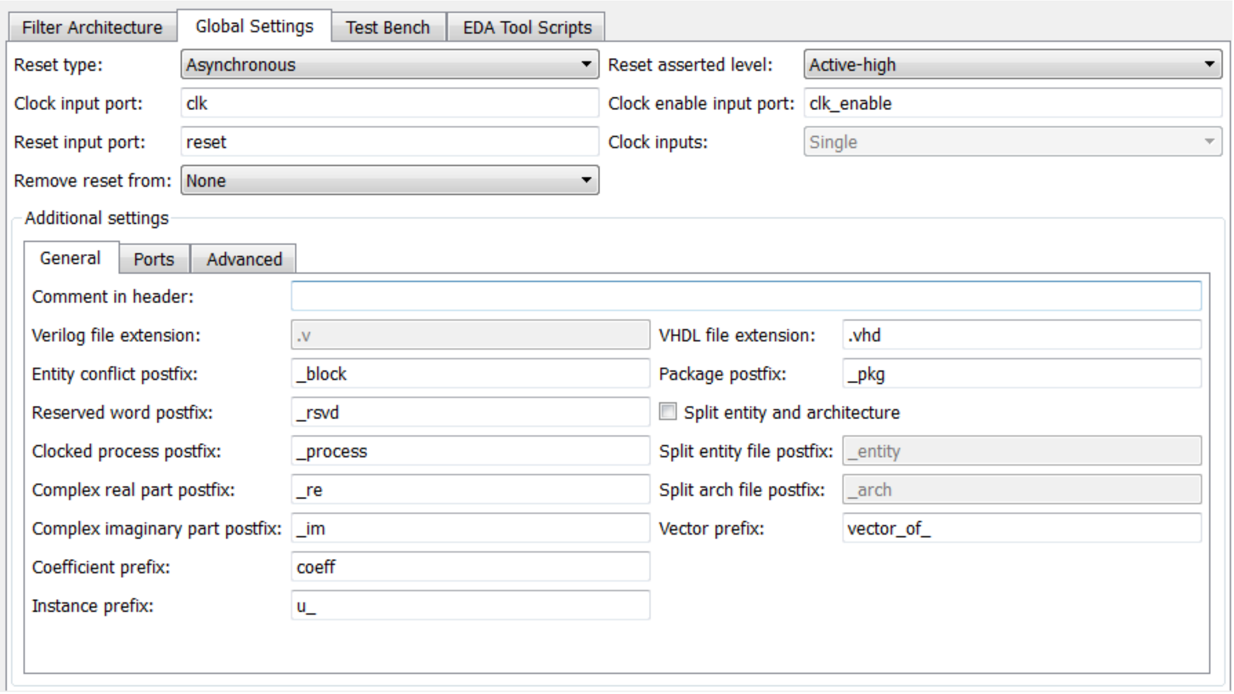
**Note:** The hold time applies to reset input signals only if the forcing of reset input signals is enabled.

The following table summarizes the reset configuration settings,

| If You Want to... | Then... |
|---|---|
| Disable the forcing of reset input signals | Clear **Force reset** in the **Test Bench** pane of the Generate HDL dialog box. |

| If You Want to... | Then... |
|---|---|
| Change the length of time (in clock cycles) during which reset is asserted | Set **Reset length (in clock cycles)** to an integer greater than or equal to 0. This option is located in the **Test Bench** pane of the Generate HDL dialog box. |
| Change the reset value to active low (0) | Select `Active-low` from the **Reset asserted level** menu in the **Global Settings** pane of the Generate HDL dialog box (see "Setting the Asserted Level for the Reset Input Signal" on page 5-23) |
| 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 dialog box. |

The following figures highlight the applicable options.



Reset type: Asynchronous

Reset asserted level: Active-high

Clock input port: clk

Clock enable input port: clk_enable

Reset input port: reset

Clock inputs: Single

Remove reset from: None

Additional settings

Comment in header:

Verilog file extension: .v

VHDL file extension: .vhd

Entity conflict postfix: _block

Package postfix: _pkg

Reserved word postfix: _rsvd

☐ Split entity and architecture

Clocked process postfix: _process

Split entity file postfix: _entity

Complex real part postfix: _re

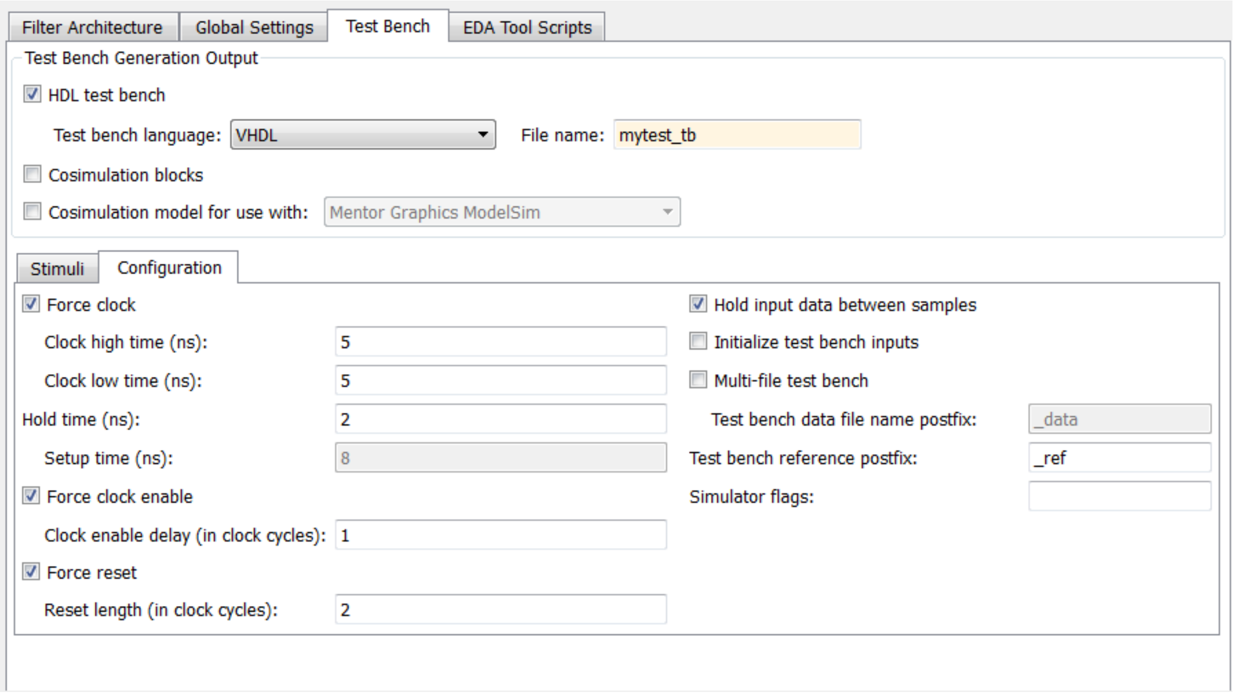
Split arch file postfix: _arch

Complex imaginary part postfix: _im

Vector prefix: vector_of_

Coefficient prefix: coeff

Instance prefix: u_



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

☐ Cosimulation blocks

☐ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

☐ Initialize test bench inputs

☐ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

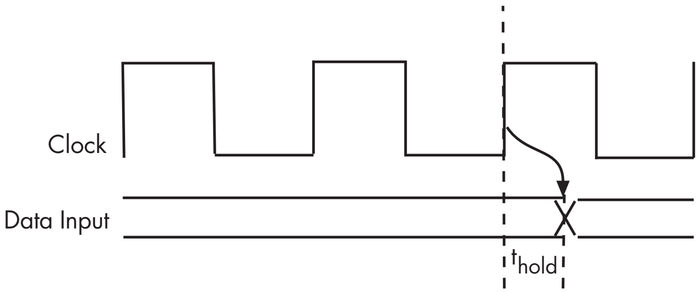
Simulator flags:

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

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

## Setting a Hold Time for Data Input Signals

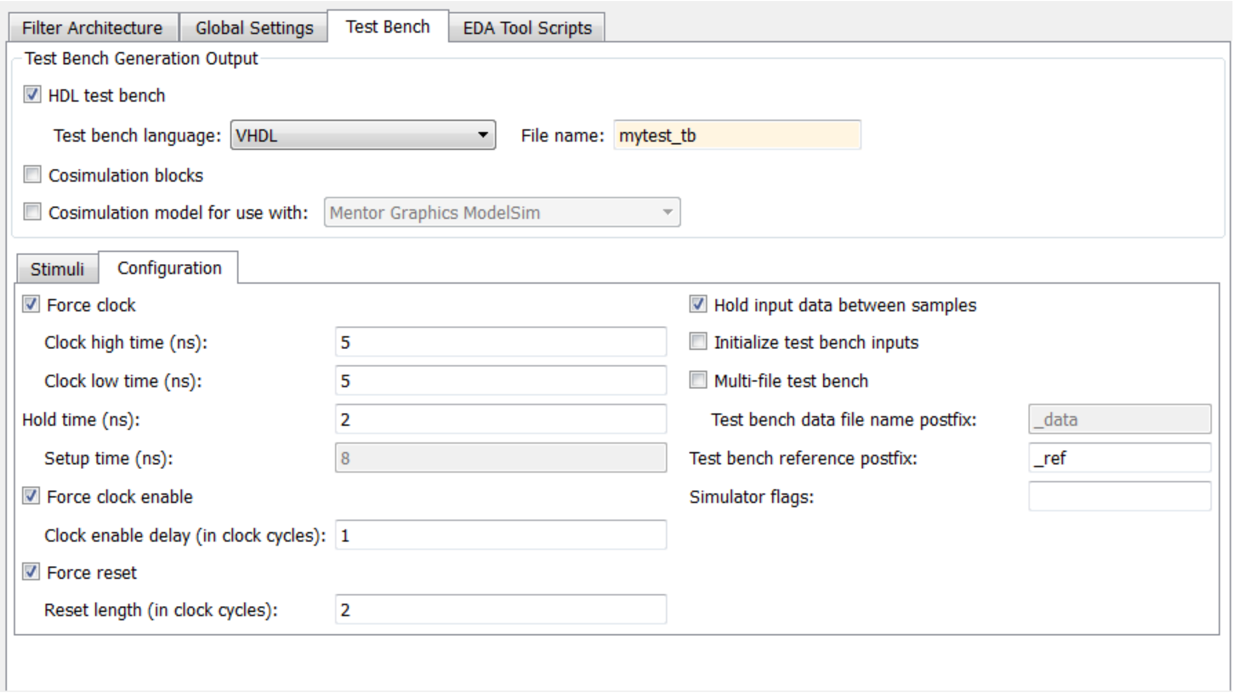
By default, the coder applies a hold time of 2 nanoseconds for filter data input signals. The hold time is the amount of time that data input signals are to be held past the clock rising edge. The following figure shows the application of a hold time ($t_{hold}$) for data input signals.



To change the hold time setting,

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

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



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Simulator flags:

**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

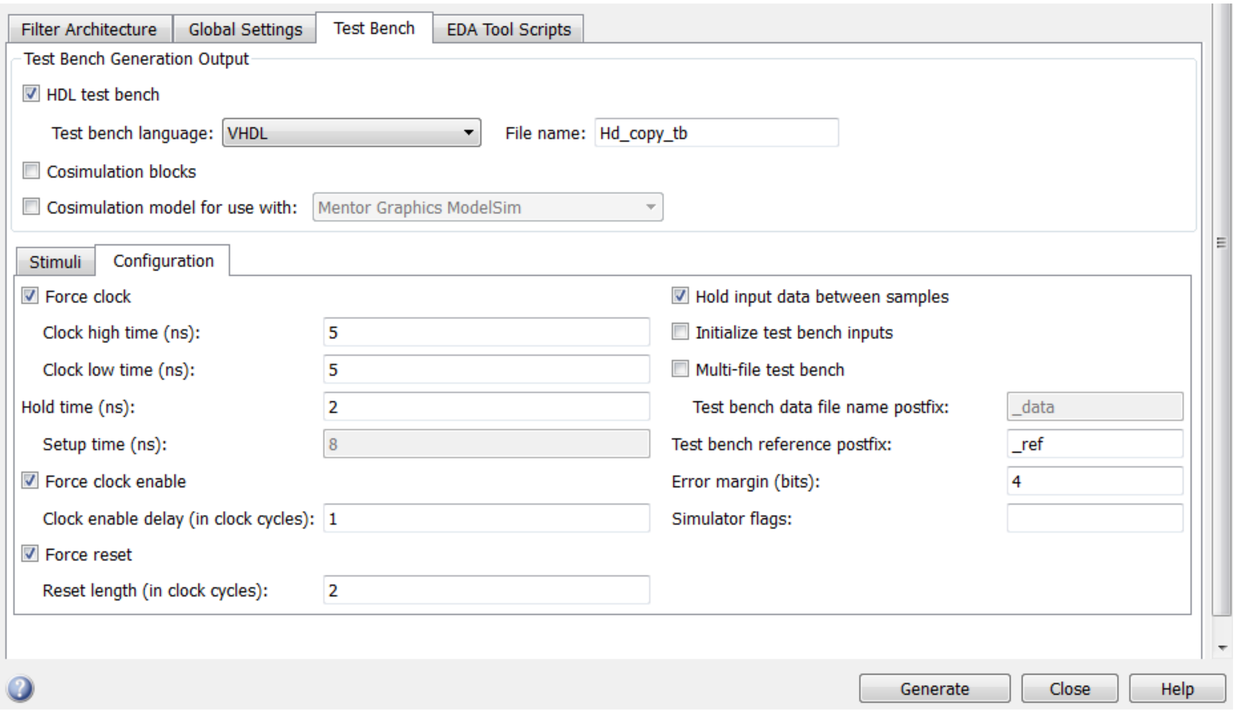
Customizations that provide optimizations can generate test bench code that produces numeric results that differ from results produced by the original filter object. These options include:

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



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: Hd_copy_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Error margin (bits): 4

Simulator flags:

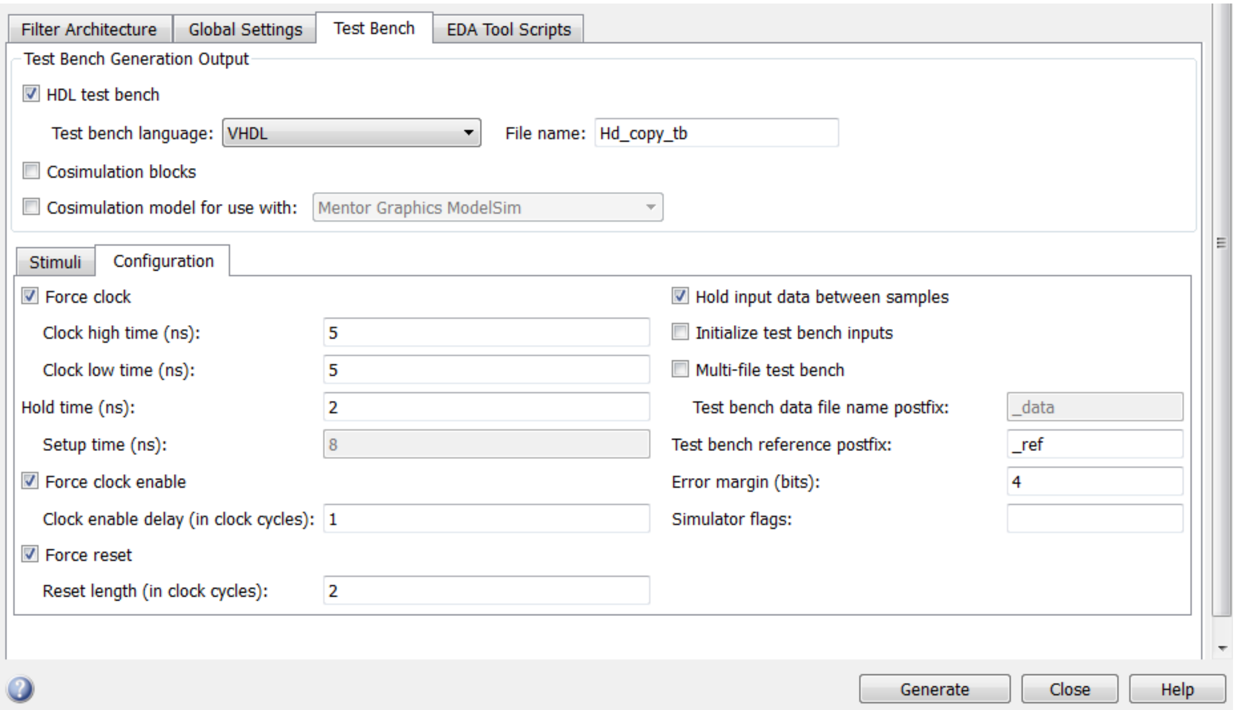
**Command-Line Alternative:** Use the `generatehdl` property ErrorMargin to specify the number of bits of tolerable error.

## 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 dialog box.

2 Within the **Test Bench** pane, select the **Configuration** subpane.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL  File name: Hd_copy_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

□ Initialize test bench inputs

□ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: _ref

Error margin (bits): 4

Simulator flags:

3 To set an initial test bench input value of `0`, select the **Initialize test bench inputs** option.

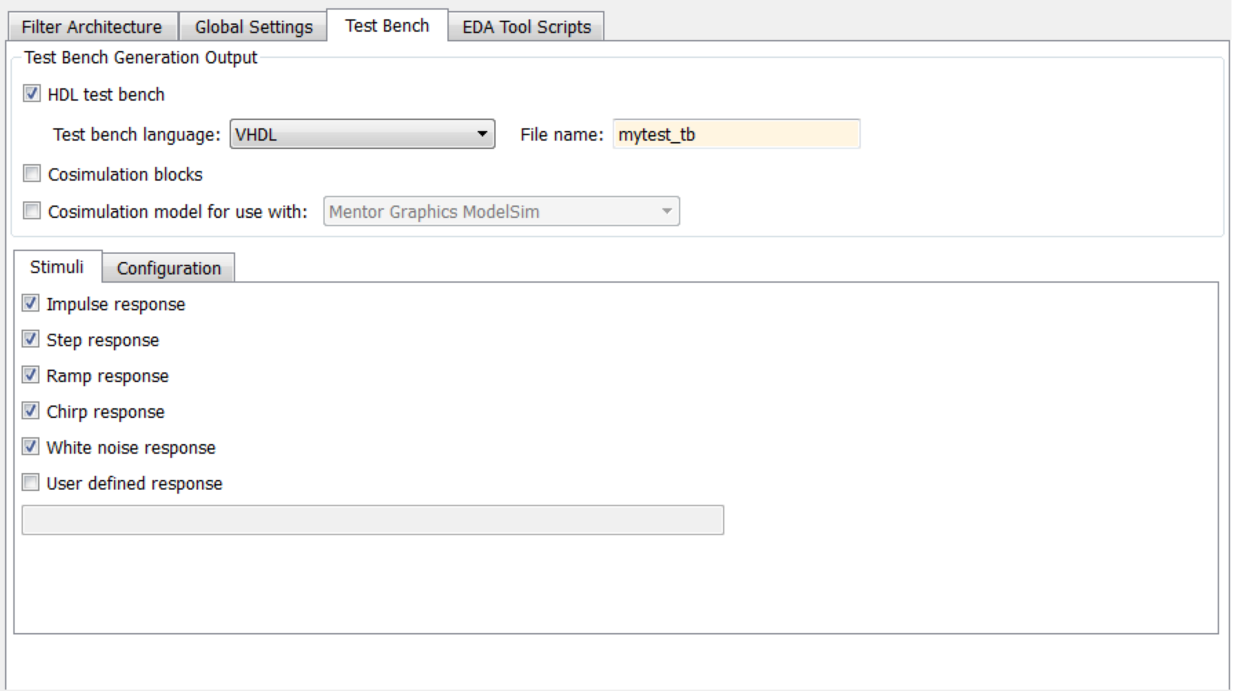
To set an initial test bench input value of `'X'`, clear the **Initialize test bench inputs** option.

**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 dialog box. The figure highlights this pane of the dialog box.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

□ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

☑ Chirp response

☑ White noise response

□ User defined response

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.

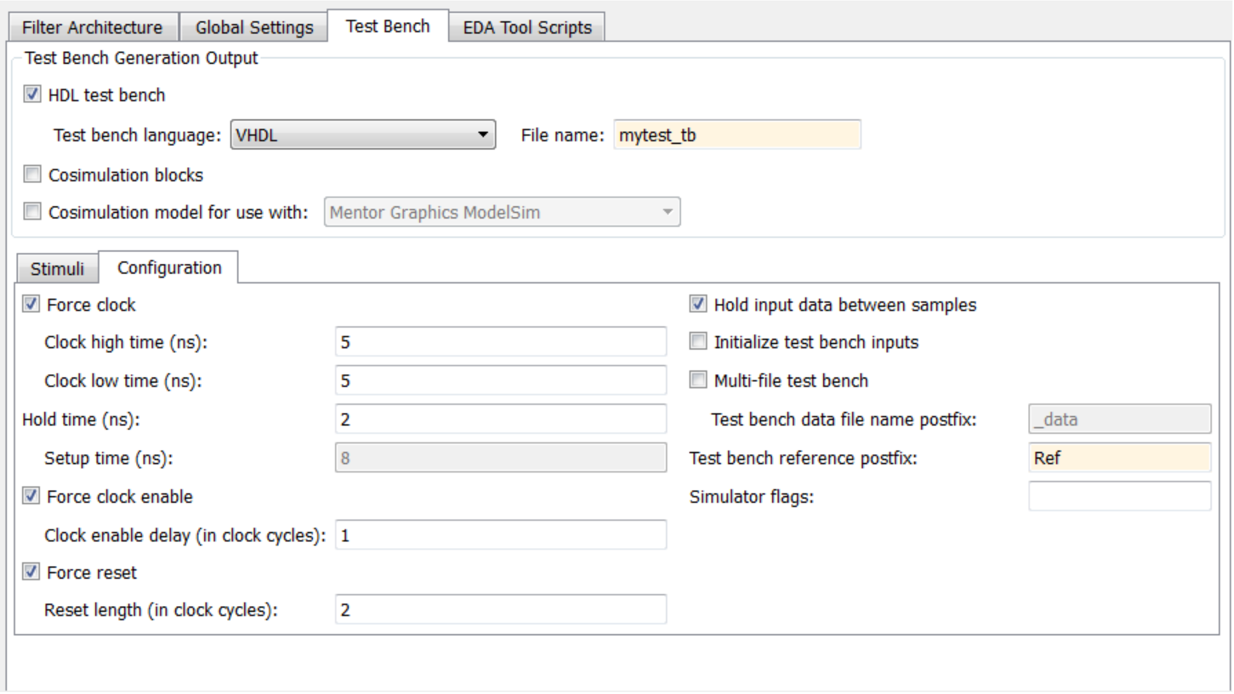
**Command-Line Alternative:** Use the `generatehdl` properties TestBenchStimulus and TestBenchUserStimulus to adjust stimuli settings.

## Setting a Postfix for Reference Signal Names

Reference signal data is represented as arrays in the generated test bench code. The string specified by **Test bench reference postfix** is appended to the generated signal names. The default string is `_ref`.

You can set the postfix string to a value other than `_ref`. To change the string:

1. Select the **Test Bench** pane in the Generate HDL dialog box.
2. Within the **Test Bench** pane, select the **Configuration** subpane.
3. Enter a new string in the **Test bench reference postfix** field, as shown in the following figure.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL

File name: mytest_tb

☐ Cosimulation blocks

☐ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

☐ Initialize test bench inputs

☐ Multi-file test bench

Test bench data file name postfix: _data

Test bench reference postfix: Ref

Simulator flags:

**Command-Line Alternative:** Use the `generatehdl` property TestBenchReferencePostfix to change the postfix string.

## More About

- "Integration with Third-Party EDA Tools" on page 6-36

# 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.

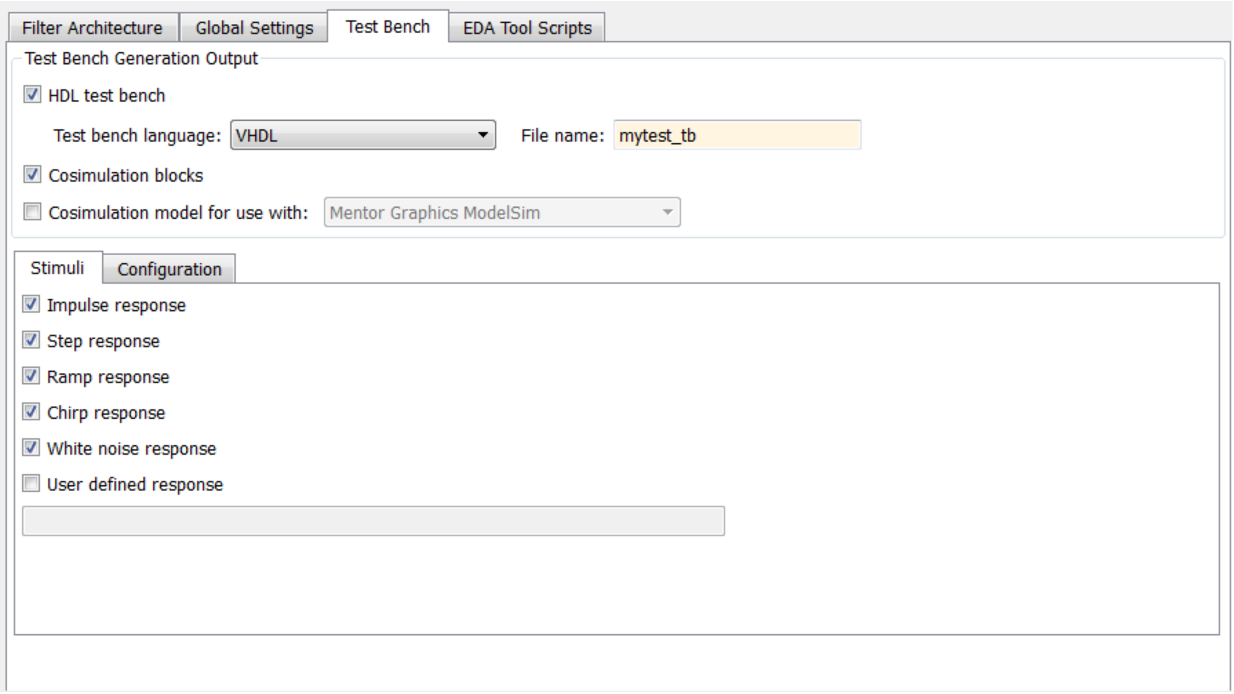
The generated HDL Cosimulation blocks are configured to conform to the port and data type interface of the filter selected for code generation. By connecting an HDL Cosimulation block to a Simulink model in place of the filter, you can cosimulate your design with the desired HDL simulator.

To generate HDL Cosimulation blocks:

1. Select the **Test Bench** pane in the Generate HDL dialog box.
2. Select the **Cosimulation blocks** option.

   When this option is selected, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for each supported HDL simulator.
3. If you want to generate HDL Cosimulation blocks only (without generating HDL test bench code), clear **HDL test bench**.

   The following figure shows both **HDL test bench** and **Cosimulation blocks** selected.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL    File name: mytest_tb

☑ Cosimulation blocks

□ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

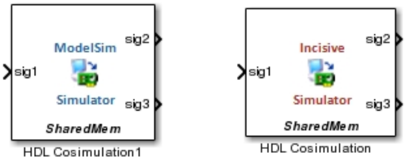
☑ Chirp response

☑ White noise response

□ User defined response

4 In the Generate HDL dialog box, click **Generate** to generate HDL and test bench code.

5 In addition to the usual code files, the coder generates a Simulink model containing 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”.

**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 the following:
    - 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

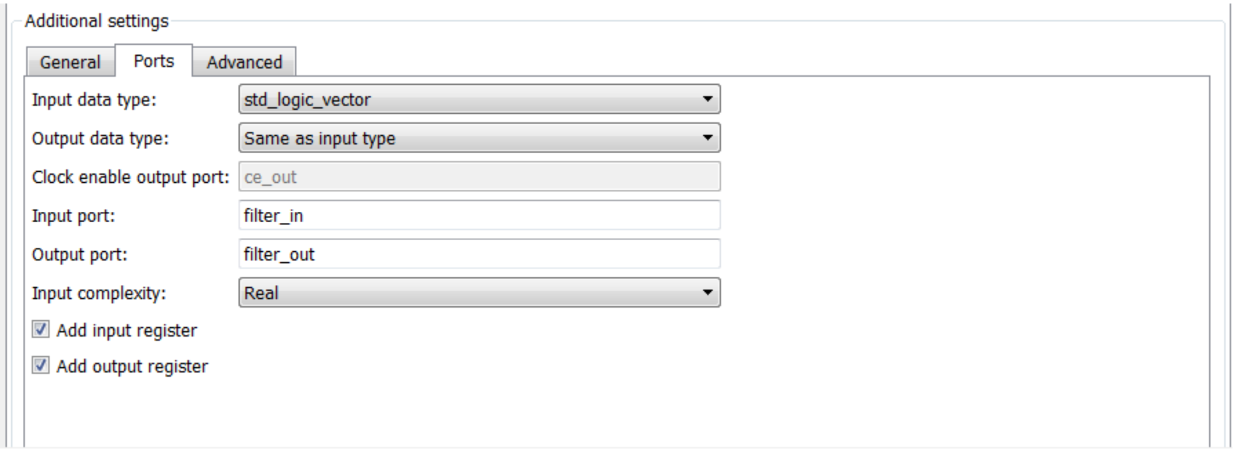
Generation of a cosimulation model requires registered inputs and/or outputs (see “Limitations” on page 6-35). Before generating the model, make sure that your model meets this requirement, as follows:

1 Select the **Global Settings** pane the Generate HDL dialog box.

2 In the **Global Settings** pane, click the **Ports** tab. Port options appear.

3 Select both of the following options:

- **Add input register**
- **Add output register**



Additional settings

Input data type: std_logic_vector

Output data type: Same as input type

Clock enable output port: ce_out

Input port: filter_in

Output port: filter_out

Input complexity: Real

☑ Add input register

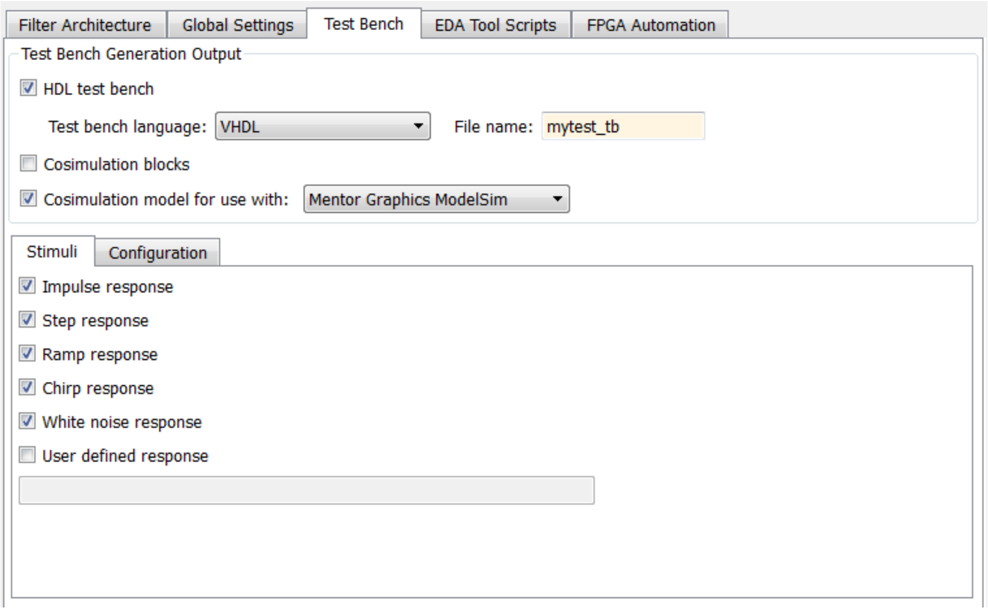
☑ Add output register

To generate the model:

1 In the Generate HDL dialog box, configure other code generation and test bench parameters as required by your design.

2 Select the **Test bench** pane of the Generate HDL dialog box.

3 Select the **Cosimulation model for use with:** option. Selecting this option enables the adjacent drop-down menu, where you can select `Mentor Graphics ModelSim` or `Cadence Incisive`.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL    File name: mytest_tb

□ Cosimulation blocks

☑ Cosimulation model for use with: Mentor Graphics ModelSim

☑ Impulse response

☑ Step response

☑ Ramp response

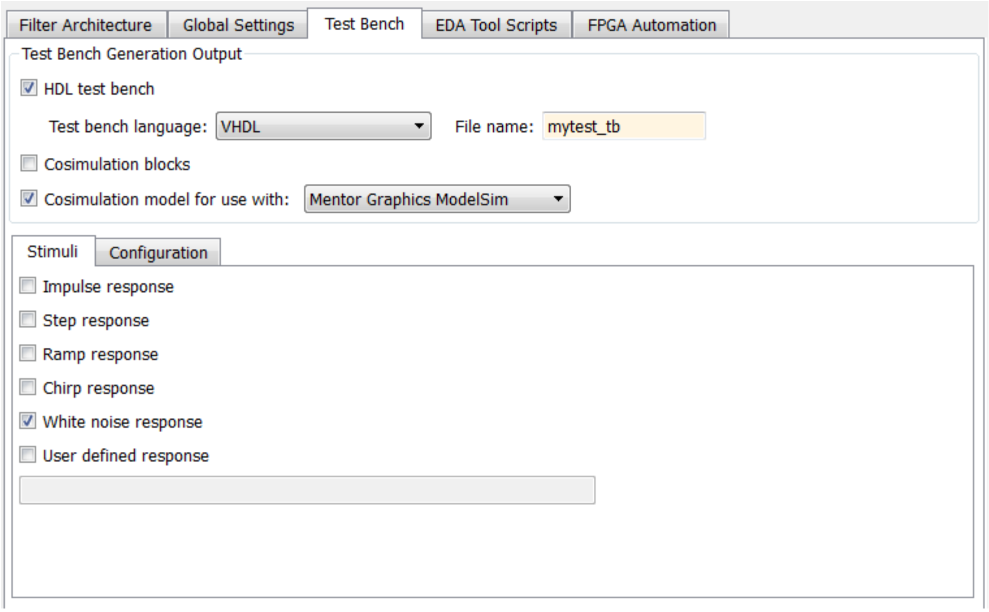
☑ Chirp response

☑ White noise response

□ User defined response

**4** Using the drop-down menu, select which type of HDL Cosimulation block you want in the generated model. Select either `Mentor Graphics ModelSim` (the default) or `Cadence Incisive`.

In the following figure, the cosimulation model type is `Mentor Graphics ModelSim`, and the stimulus signal is **White noise response**.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL    File name: mytest_tb

□ Cosimulation blocks

☑ Cosimulation model for use with: Mentor Graphics ModelSim

□ Impulse response

□ Step response

□ Ramp response

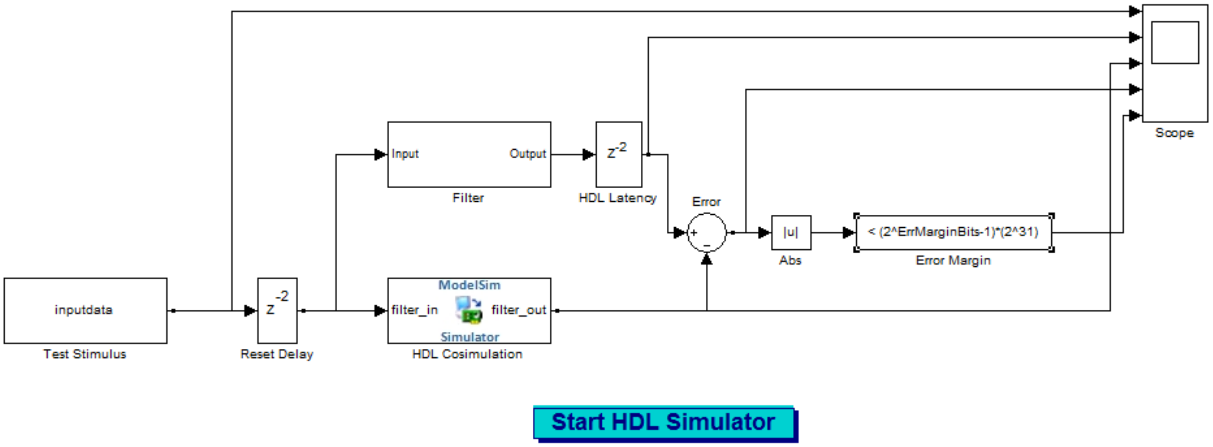
□ Chirp response

☑ White noise response

□ User defined response

**5** In the Generate HDL dialog box, click **Generate** to generate HDL and test bench code.

In addition to the usual code files, the coder generates and opens a Simulink model. The following figure shows the model generated from the coder configuration shown in the previous step.



6 The generated model is untitled and exists in memory only. Be sure to save it to a destination folder if you want to preserve the model and blocks for use in future sessions.

To configure HDL Cosimulation block parameters, such as timing, latency, and data types, see “Define HDL Cosimulation Block Interface”.

### Details of the Generated Model

The generated model contains the following blocks:

- `Test Stimulus`: This `From Workspace` block routes test data in the model workspace variable `inputdata` to the filter subsystem and `HDL Cosimulation` blocks.
- `Filter`: This subsystem realizes a behavioral model of the filter design.
- `HDL Cosimulation`: This block cosimulates the generated HDL code. The table HDL Cosimulation Block Settings describes how the 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 6-21).
- `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.<br>Input/Output data types: `Inherit`<br>Input sample time: `Inherit`<br>Output sample time: Same as Simulink fixed step size. |
| `Clocks` | Clock port name: same as the name in the generated code for the filter.<br>Active clock edge: `Rising`<br>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`<br>Connection Method: `Shared memory` |
| `Tcl` (Pre-simulation commands) | `force /Hlp/clk_enable 1;`<br>`force /Hlp/reset 1 0 ns, 0 22 ns;`<br>`puts ----------------------------------------`<br>`puts "Running Simulink Cosimulation block.";` |

| Pane | Settings |
|---|---|
| | `puts [clock format [clock seconds]]` |
| `Tcl` (Post-simulation commands) | `force /Hlp/reset 1`<br>`puts [clock format [clock seconds]]` |

### Generated Model Settings

The generated model has the following nondefault settings:

- **Solver**: `Discrete (no continuous states)`.
- Solver **Type**: `Fixed-step`.
- **Stop Time**: `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 the following options are selected:

  - **Add input register**
  - **Add output register**

  If you do not select either of these options, the coder terminates model generation with an error. 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 the following values for the property GenerateCosimModel.

- `generatehdl(filterObj,'GenerateCosimModel','Incisive');`
- `generatehdl(filterObj,'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, `Hd`, the coder writes the following script files:

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

When test bench code is generated for a filter `Hd`, the coder writes the following script files:

- `Hd_tb_compile.do`: Mentor Graphics ModelSim compilation script. This script contains commands to compile the generated filter and test bench code.
- `Hd_tb_sim.do`: Mentor Graphics ModelSim simulation script. This script contains commands to run a simulation of the generated filter and test bench code.

You can enable or disable script generation and customize the names and content of generated script files by:

- Passing properties as `'Name',Value` arguments to the `generatehdl` function. See Compilation and Simulation Properties.
- Setting the corresponding options in the Generate HDL dialog box. 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 6-37.

### Structure of Generated Script Files

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

1. An initialization (`Init`) phase. The `Init` phase performs required setup actions, such as creating a design library or a project file.

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 strings to the `fprintf` function. Using the GUI options (or `generatehdl` properties) summarized in the following sections, you can pass in customized format strings to the script generator. Some of these format strings take arguments, such as the top-level entity or module name.

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

## Customize Scripts for Compilation and Simulation

To view and set options in the **EDA Tool Scripts** dialog box:

1. Open the Generate HDL dialog box.
2. Click the **EDA Tool Scripts** tab.

   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.

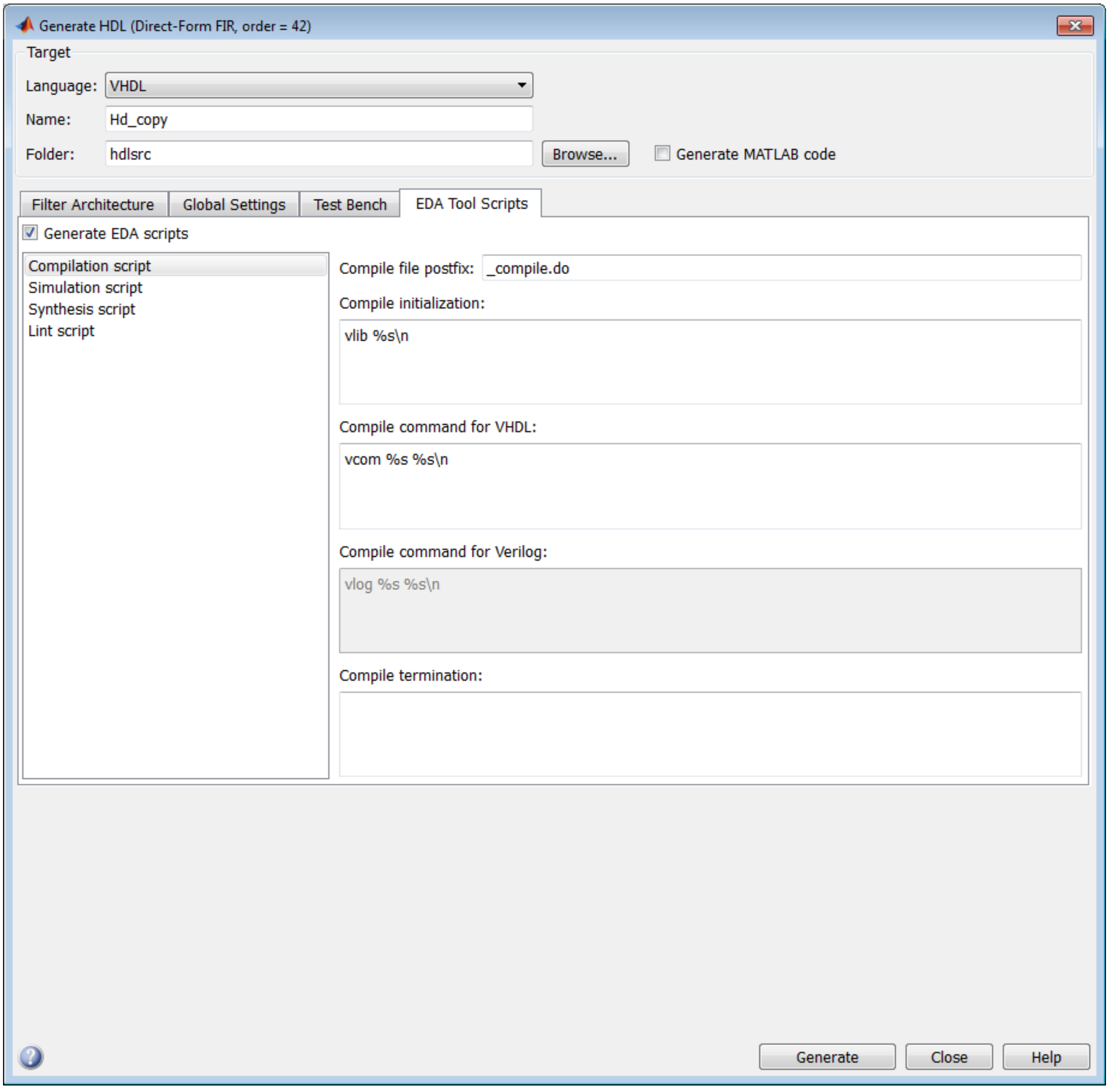
   If you want to disable script generation, clear this check box.
4. The list on the left of the dialog box lets you select from several categories. Select a category and set the options as desired. The categories are:

   - **Compilation script**: customize scripts for compilation of generated VHDL or Verilog code. See “Compilation Script Options” on page 6-38.
   - **Simulation script**: customize scripts for HDL simulators. See “Simulation Script Options” on page 6-41 .
   - **Synthesis script**: customizing scripts for synthesis tools. See “Automation Scripts for Third-Party Synthesis Tools” on page 7-2 .
5. The custom strings for each section are passed to `fprintf` to write each section of the selected script. You can use format strings supported by the `fprintf` function. Some of the strings 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 string, `''`, by default)<br>- File name of the current module |
| **Compile termination** | No implicit argument |
| **Compile initialization** | No implicit argument |
| **Simulation command** | - Library name<br>- 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.



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

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

☑ Generate EDA scripts

Compile file postfix: _compile.do

Compile initialization: vlib %s\n

Compile command for VHDL: vcom %s %s\n

Compile command for Verilog: vlog %s %s\n

Compile termination:

The coder generates a script called `Hd_copy_compile.do`:

```
vlib work
vcom  Hd_copy.vhd
```

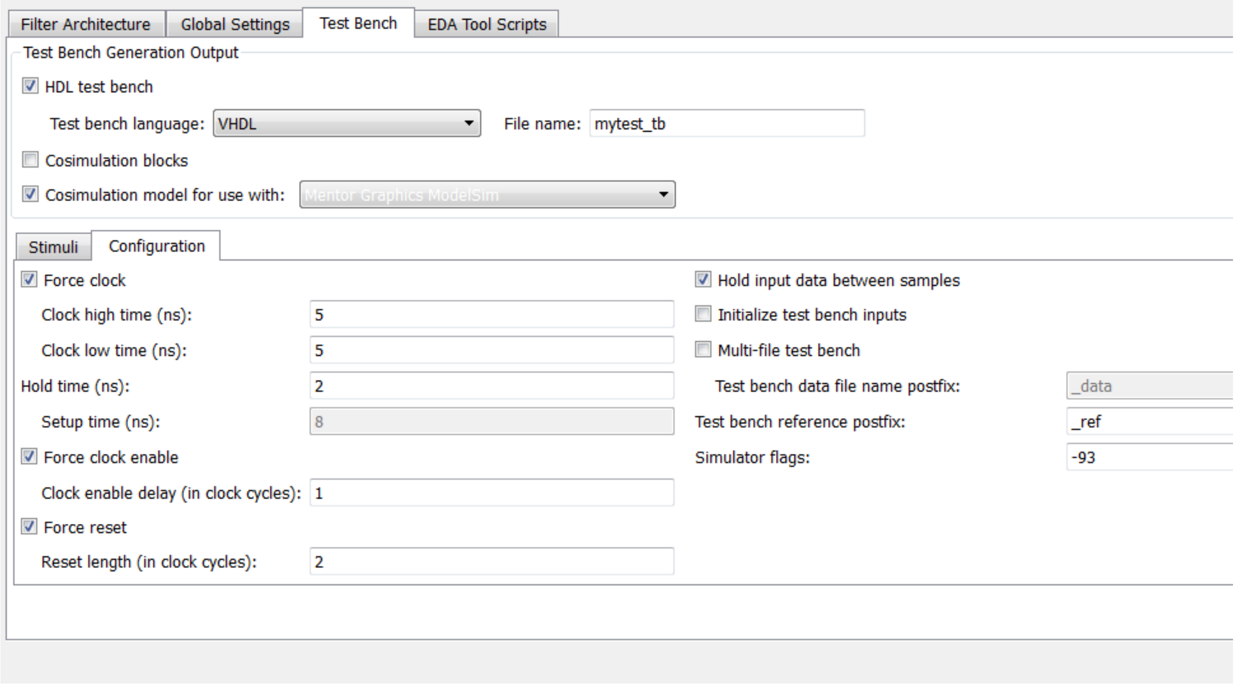
If you generate a test bench for your filter, the coder also generates a script called `Hd_copy_tb_compile.do`

```
vlib work
vcom  Hd_copy.vhd
vcom  Hd_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 dialog box.
2. Type the flags of interest in the **Simulator flags** field. In the figure, the dialog box specifies that the Mentor Graphics ModelSim simulator use the `-93` compiler option for compilation.



Test Bench Generation Output

☑ HDL test bench

Test bench language: VHDL ▾ File name: mytest_tb

☐ Cosimulation blocks

☑ Cosimulation model for use with: Mentor Graphics ModelSim ▾

☑ Force clock

Clock high time (ns): 5

Clock low time (ns): 5

Hold time (ns): 2

Setup time (ns): 8

☑ Force clock enable

Clock enable delay (in clock cycles): 1

☑ Force reset

Reset length (in clock cycles): 2

☑ Hold input data between samples

☐ Initialize test bench inputs

☐ Multi-file test bench

Test bench data file name postfix: _data

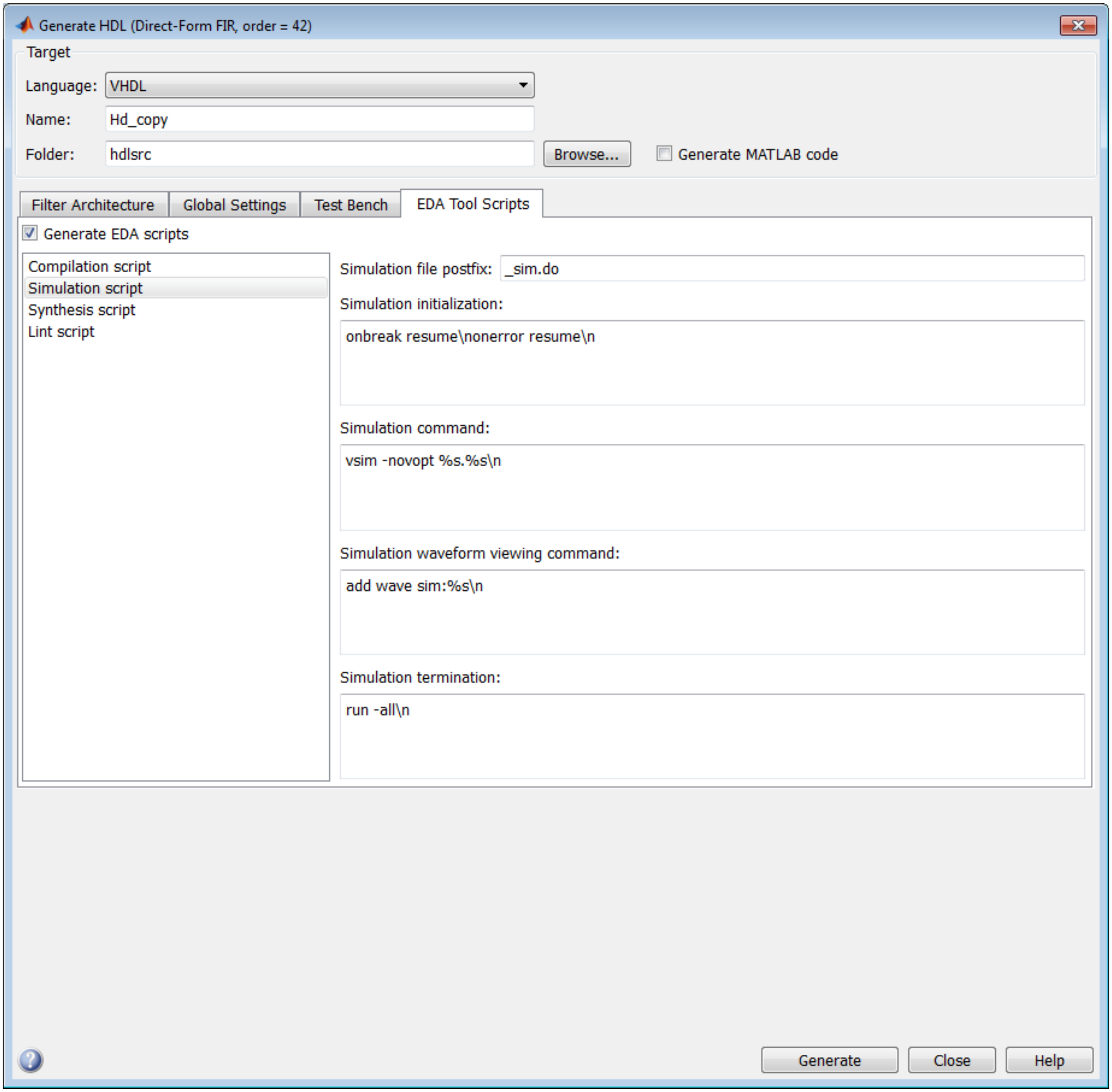
Test bench reference postfix: _ref

Simulator flags: -93

**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.



Language: VHDL

Name: Hd_copy

Folder: hdlsrc

□ Generate MATLAB code

☑ Generate EDA scripts

Simulation file postfix: _sim.do

Simulation initialization:

```
onbreak resume\nonerror resume\n
```

Simulation command:

```
vsim -novopt %s.%s\n
```

Simulation waveform viewing command:

```
add wave sim:%s\n
```

Simulation termination:

```
run -all\n
```

The coder generates a script called `Hd_copy_tb_sim.do`:

```
onbreak resume
onerror resume
vsim -novopt work.Hd_copy_tb
add wave sim:/Hd_copy_tb/u_Hd_copy/clk
add wave sim:/Hd_copy_tb/u_Hd_copy/clk_enable
add wave sim:/Hd_copy_tb/u_Hd_copy/reset
add wave sim:/Hd_copy_tb/u_Hd_copy/filter_in
add wave sim:/Hd_copy_tb/u_Hd_copy/filter_out
add wave sim:/Hd_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 dialog box, 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.

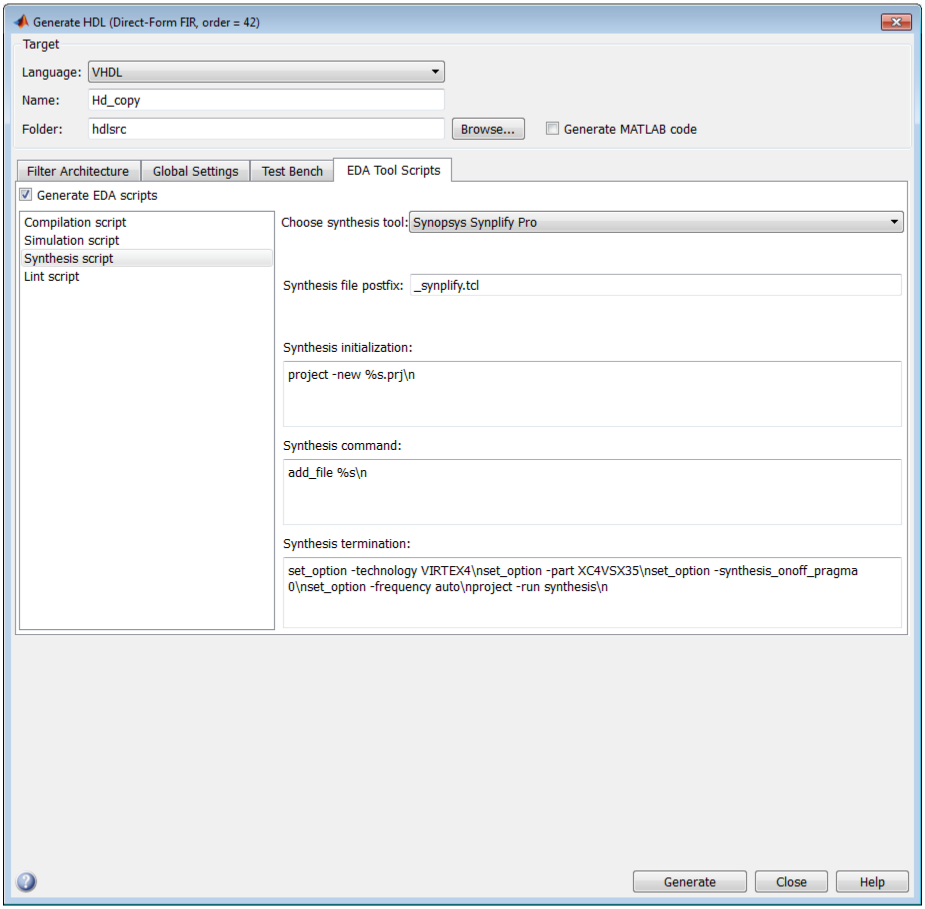
You can also select `'Custom'`, and set the **Synthesis initialization**, **Synthesis command**, and **Synthesis termination** Tcl code fields to generate a script that supports your tool.

## 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 dialog box, 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 dialog box. The script file is named according to the name of your module or entity combined with the string in **Synthesis file postfix**. The custom strings for each section are passed to `fprintf` to write each section of the synthesis script. You can use format strings 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.



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

Target

Language: VHDL

Name: Hd_copy

Folder: hdlsrc

☐ Generate MATLAB code

☑ Generate EDA scripts

Choose synthesis tool: Synopsys Synplify Pro

Synthesis file postfix: _synplify.tcl

Synthesis initialization:

```
project -new %s.prj\n
```

Synthesis command:

```
add_file %s\n
```

Synthesis termination:

```
set_option -technology VIRTEX4\nset_option -part XC4VSX35\nset_option -synthesis_onoff_pragma
0\nset_option -frequency auto\nproject -run synthesis\n
```

The coder generates a script called `Hd_copy_synplify.tcl`:

```
project -new Hd_copy.prj
add_file Hd_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 Automation Properties.

8

# Properties — Alphabetical List

# Fundamental Properties

Customize filter name and input data type, generation language, and target folder

## Description

Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options at the top of the Generate HDL dialog box.

### Target

**`Name` — File name for generated HDL code**
string

This name is also used for the VHDL entity or Verilog module for the filter. The coder creates the file in the location specified in the `TargetDirectory` property. The coder uses the file type extension defined by the `VerilogFileExtension` or `VHDLFileExtension` property.

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.

- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

**`TargetDirectory` — Folder location for generated output files**
`'hdlsrc'` (default) | string

Specify the subfolder under the current working folder into which generated files are written. Alternatively, the string can specify a complete path.

**`TargetLanguage` — HDL language to use for generated filter code**
`'VHDL'` (default) | `'Verilog'`

Specify which language to use to generate the HDL implementation of the filter. The coder uses the file type extension defined by the `VerilogFileExtension` or `VHDLFileExtension` property.

## Data Types

### `InputDataType` — Specify input data type for System objects

`numerictype`

When you call `generatehdl` on a System object, you must specify this property. Set the property to an object of the `numerictype` class. 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. See `numerictype`.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.22,1,60);
Hd = design(d,'equiripple','filterstructure','dfsymfir','SystemObject',true);
generatehdl(Hd,'InputDataType',numerictype(1,16,15))
```

## Language-Specific

### `VerilogFileExtension` — File type extension for generated Verilog files

`'.v'` (default) | string

The coder uses this extension for generated Verilog files.

### `VHDLFileExtension` — File type extension for generated VHDL files

`'.vhd'` (default) | string

The coder uses this extension for generated VHDL files.

### `VHDLArchitectureName` — Architecture name for generated VHDL code

`'rtl'` (default) | string

The coder creates this architecture in generated VHDL files.

### `VHDLLibraryName` — Library name used in initialization section of compilation script

`'work'` (default) | string

At script generation time, the coder substitutes this string into the HDLCompileInit string value. By default, the coder generates the library specification`'vlib work/n'`.

You can use `VHDLLibraryName` to avoid library name conflicts with your existing VHDL code. See “Integration with Third-Party EDA Tools” on page 6-36.

## See Also

`generatehdl`

## Related Examples

- “Selecting Target Language” on page 2-13
- “Generating HDL Code” on page 2-14

# Filter Configuration Properties

Configure coefficients, complex input ports, and optional ports for specific filter types

## Description

The coder provides options to customize your filter. The properties on this page configure specific types of filters, such as those with programmable coefficients or multiple rates. Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options in the Generate HDL dialog box. These options apply to specific types of filters and are found in various locations in the Generate HDL dialog box.

| Filter Type | Option | Location in Dialog Box |
|---|---|---|
| FIR or IIR filter with programmable coefficients | **Coefficient source** | **Filter Architecture** tab |
| FIR filter with serial architecture and programmable coefficients | **Coefficient memory** | **Global Settings** tab, when **Coefficient source** is set to `Processor Interface` |
| Multirate filters | **Clock inputs** | **Global Settings** tab |
| Filters with complex input data | **Input complexity** | **Global Settings** tab > **Ports** tab. |
| Single-rate Farrow filter | **Fractional delay port** | **Global Settings** tab > **Ports** tab |
| CIC filter | **Add rate port** | **Filter Architecture** tab |

For filter serialization and pipeline properties, see Optimization Properties.

## Coefficients

### `CoefficientSource` — Source for FIR or IIR filter coefficients

`'Internal'` (default) | `'ProcessorInterface'`

When you set this property to `'Internal'`, the filter coefficients are obtained from the filter object and hard-coded in the generated HDL code.

When you set this property to `'ProcessorInterface'`, the coder generates a memory interface for the filter coefficients. You can drive this interface with an external microprocessor. The generated entity or module definition for the filter includes these ports for the processor interface:

- `coeffs_in` — Input port for coefficient data
- `write_address` — The write address for coefficient memory
- `write_enable` — The write enable signal for coefficient memory
- `write_done` — Signal to indicate completion of coefficient write operation

The generated test bench also generates input stimulus for this interface. See TestBenchCoeffStimulus.

When you use a `'ProcessorInterface'` with a serial FIR filter, you can use the `CoefficientMemory` property to select the type of storage.

See:

- “Fully Parallel FIR Filter with Programmable Coefficients” on page 9-6
- “Programmable Filter Coefficients for FIR Filters” on page 3-30
- “Programmable Filter Coefficients for IIR Filters” on page 3-40

### `CoefficientMemory` — Type of memory for storing programmable coefficients for serial FIR filters

`'Registers'` (default) | `'DualPortRAMs'` | `'SinglePortRAMs'`

This property applies only to FIR filters that have a serial architecture: `Fully serial`, `Partly serial`, or `Cascade serial`.

The default setting, `'Registers'`, generates code that stores programmable coefficients in a register file. When you set this property to `'SinglePortRAMs'` or `'DualPortRAMs'`, the coder generates the respective RAM interface. See “Partly Serial FIR Filter with Programmable Coefficients” on page 9-6.

This property applies only when you set `CoefficientSource` to `'ProcessorInterface'`. The coder ignores `CoefficientMemory` unless it is generating an interface for programmable coefficients.

## Optional Ports

### `InputComplex` — Generate real and imaginary ports for complex input data

`'off'` (default) | `'on'`

Use this option when your filter design requires complex input data. To enable generation of ports and signal paths for the real and imaginary components of a complex signal, set `InputComplex` to `'on'`.

```
Hd = design(fdesign.lowpass,'equiripple','Filterstructure','dffir');
generatehdl(Hd,'InputComplex','on')
```

The generated VHDL entity includes ports for both complex data components.

```
ENTITY Hd IS
   PORT( clk                              :   IN    std_logic;
         clk_enable                       :   IN    std_logic;
         reset                            :   IN    std_logic;
         filter_in_re                     :   IN    real; -- double
         filter_in_im                     :   IN    real; -- double
         filter_out_re                    :   OUT   real; -- double
         filter_out_im                    :   OUT   real  -- double
         );
END Hd;
```

See “Using Complex Data and Coefficients” on page 5-34.

You can customize the names of the two ports by setting the ComplexRealPostfix and ComplexImagPostfix properties.

### `ClockInputs` — Generate single or multiple clock inputs for multirate filters

`'Single'` (default) | `'Multiple'`

When you set this property to `'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, 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. You can customize the name of the `ce_out` port using the ClockEnableOutputPort property. 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.

When you set this property to 'Multiple', the coder generates multiple clock inputs for a multirate filter. The module or entity declaration for the filter has separate clock inputs for each rate of a multirate filter. Each clock input has an associated clock enable input. The coder does not generate a clock enable output. 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. Multiple clock inputs are not supported for:

- Filters with a Partly serial architecture
- Multistage sample rate converters: dsp.FIRRateConverter, dsp.FarrowRateConverter, or multirate dsp.FilterCascade

This option provides more flexibility than a single clock input but assumes 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. See "Clock Ports for Multirate Filters" on page 9-14 and "Multirate Filters" on page 3-4.

### AddRatePort — Generate rate ports for variable-rate CIC filter

'off' (default) | 'on'

Generate rate and load_rate ports for variable-rate CIC filters. A variable-rate CIC filter has a programmable rate change factor. When the load_rate signal is asserted, the rate port loads in a rate factor. You can generate rate ports for a full-precision filter only.

The generated test bench also includes stimulus for the rate ports. See TestBenchRateStimulus.

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 rate-change factor. See "Variable Rate CIC Filters" on page 3-10.

### FracDelayPort — Name of fractional delay input port, for single-rate Farrow filters

'filter_fd' (default) | string

The fractional delay input of a single-rate Farrow filter enables the use of time-varying delays as the filter operates. The fractional delay input is a signal that ranges from 0 through 1.0. This property specifies the name of this port. For example, if you specify the string 'frac_delay' for filter entity Hd, the coder generates a port with that name.

```
ENTITY Hd IS
   PORT( clk              :  IN  std_logic;
         clk_enable       :  IN  std_logic;
         reset           :  IN  std_logic;
         filter__in      :  IN  std_logic_vector (15 DOWNTO 0);
         frac_delay      :  IN  std_logic_vector (5 DOWNTO 0);
         filter_out      :  OUT std_logic_vector (15 DOWNTO 0);
         );
END Hd;
```

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.
- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

See “Single-Rate Farrow Filters” on page 3-23.

The generated test bench also includes stimuli for the rate ports. See TestBenchFracDelayStimulus.

## See Also

`generatehdl`

# Optimization Properties

Optimize speed or area of generated HDL code

## Description

These properties configure the HDL architecture of your filter to improve speed or reduce area. You can customize a serial filter architecture, select alternative multiplier implementations, and add pipeline registers. Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options on the **Filter Architecture** tab of the Generate HDL dialog box.

### Speed Optimization

**`AddPipelineRegisters` — Optimize clock rate of generated filter code, by adding pipeline registers**
`'off'` (default) | `'on'`

When you set this property to `'on'`, the coder adds a pipeline register between stages of computation in a filter. For example, for a sixth-order IIR filter, the coder adds two pipeline registers, one between the first and second sections and one between the second and third sections. Although the registers add to the overall filter latency, they provide significant improvements to the clock rate. For FIR filters, the use of pipeline registers optimizes filter final summation. The coder forces a tree structure for non-transposed FIR filters. This setting overrides the setting of the `FIRAdderStyle` property.

| Filter Type | Location of Added Pipeline Register |
|---|---|
| FIR transposed | Between coefficient multipliers and adders |
| FIR, asymmetric FIR, and symmetric FIR | Between levels of a tree-based final adder |
| IIR | Between sections |

For details, see “Optimizing Final Summation for FIR Filters” on page 4-34.

**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. The difference occurs because adding pipeline registers forces the tree mode of final summation. In such cases, consider adjusting the generated test bench error margin with the ErrorMargin property.

---

You cannot use this property with a fully serial or cascade serial filter implementation.

### `FIRAdderStyle` — Final summation technique used for FIR filters

`'linear'` (default) | `'tree'`

By default, the coder generates linear adder summation logic. Set this property to `'tree'` to increase clock speed while using the same area. The tree architecture computes products in parallel, rather than sequentially, and it creates a final adder that performs pairwise addition on successive products.

Another option for FIR filter sum implementation is to set the `AddPipelineRegisters` property to `'on'`. The pipelined implementation produces results similar to tree mode, with the addition of a stage of pipeline registers after processing each level of the tree.

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

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

See “Optimizing Final Summation for FIR Filters” on page 4-34.

You cannot use this property with a fully serial or cascade serial filter implementation.

### `AddInputRegister` — Generate extra register on filter input in HDL code

`'on'` (default) | `'off'`

By default, the coder adds an extra input register to the generated HDL code for the filter. The code declares a signal named `input_register` and includes a `PROCESS` statement that controls the register. You can set other properties to control the names of the clock, clock enable, and reset signals, the polarity of the reset signal, and the coding style that checks for clock events. See Ports and Identifiers Properties.

```
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 ;
```

When you set this property to `'off'`, the coder omits the extra input register from the generated HDL code for the filter. Consider omitting the extra register if you are incorporating the filter into HDL code that has an existing register to drive the filter input. Also, omit the extra register if the latency it introduces to the filter is not tolerable.

### `AddOutputRegister` — Generate extra register for filter output in HDL code

`'on'` (default) | `'off'`

By default, the coder adds an extra output register to the generated HDL code for the filter. The code declares a signal named `output_register` and includes a `PROCESS` statement that controls the register. You can set other properties to control the names of the clock, clock enable, and reset signals, the polarity of the reset signal, and the coding style that checks for clock events. See Ports and Identifiers Properties.

```
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 ;
```

When you set this property to `'off'`, the coder omits the extra output register from the generated HDL code for the filter. Consider omitting the extra register if you are incorporating the filter into HDL code that has an existing output register. Also, omit the extra register if the latency it introduces to the filter is not tolerable.

### `MultiplierInputPipeline` — Number of pipeline stages at multiplier inputs for FIR filters

`0` (default) | nonnegative integer

For FIR filters, the coder generates this number of pipeline stages on each multiplier input. The number of multipliers must be an integer greater than or equal to zero. Multiplier pipelining can help you achieve significantly higher clock rates. The coder ignores this property if `CoeffMultipliers` is not set to `'multipliers'`.

### `MultiplierOutputPipeline` — Number of pipeline stages at multiplier outputs for FIR filters

`0` (default) | nonnegative integer

For FIR filters, the coder generates this number of pipeline stages on each multiplier output. The number of multipliers must be an integer greater than or equal to zero. Multiplier pipelining can help you achieve significantly higher clock rates. The coder ignores this property if `CoeffMultipliers` is not set to `'multipliers'`.

## Area Optimization

### `OptimizeForHDL` — Basic optimization of data types, quantization, and filter structure

`'off'` (default) | `'on'`

By default, the coder generates a fully parallel architecture with numerics that match the filter object exactly. However, the data types and quantization used in the software implementation are not necessarily optimal for HDL implementation. When you set this property to `'on'`, the coder generates HDL code that reduces area of the hardware implementation and optimizes data types and quantization effects. As a result of these optimizations, the coder can:

- Implement an adder-tree structure
- Make tradeoffs concerning data types
- Avoid excessive quantization
- Generate code that produces numeric results that differ from results produced by the original filter function

You can combine this option with the serial architecture and multiplier optimization properties.

### `CoeffMultipliers` — Implementation of coefficient multiplications in generated HDL code

`'multiplier'` (default) | `'csd'` | `'factored-csd'`

By default, the coder retains multiplier logic in the generated HDL code. To reduce the area of the filter implementation, you can choose to implement multiplication in either canonical signed digit (CSD) or factored CSD logic. The CSD technique replaces multipliers with shift and add logic.

A CSD architecture 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 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.

This option is not supported for multirate or serial architecture filters.

### `SerialPartition` — Number and size of partitions generated for serial filter architectures

`[p1 p2 p3...pN]`

By default, the coder generates a fully parallel architecture, which is equivalent to a vector of `FL` ones, where `FL` is the length of the filter.

To generate a fully serial architecture, set this property to the length of the filter, `FL`.

To generate a partly serial architecture, set this property to a vector of integers, `[p1 p2 p3...pN]`. This vector specifies the length of each of `N` partitions. The sum of the vector elements must be equal to the length of the filter, `FL`.

For a cascade of filters, set this property to `{[p1 p2 p3...pNa], [p1 p2 p3...pNb],...}`, where each vector in the cell array represents a serial partitioning of an individual filter within the cascade.

For further savings in area, you can optionally enable the `ReuseAccum` property to generate a cascade-serial architecture using the partitions you specified.

For a complete description of parallel and serial architectures and a list of filter types supported for each architecture, see “Speed vs. Area Tradeoffs” on page 4-2. For an example, see “Compare Serial Architectures for FIR Filter” on page 9-7

You can specify different `SerialPartition` values for each stage of a cascaded filter. See“Serial Partitions for Cascaded Filter” on page 9-8.

### `ReuseAccum` — Enable accumulator reuse, when generating cascade-serial architecture for FIR filters

`'off'` (default) | `'on'`

In a cascade-serial architecture, 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.

Set this property to `'on'` to enable accumulator reuse and generate a cascade-serial architecture. If the number and size of serial partitions is not specified in the `SerialPartition` property, the coder generates an optimal partition.

For a complete description of parallel and serial architectures and a list of filter types supported for each architecture, see "Speed vs. Area Tradeoffs" on page 4-2. For an example, see "Compare Serial Architectures for FIR Filter" on page 9-7.

### `DALUTPartition` — Number and size of lookup table (LUT) partitions for distributed arithmetic (DA) implementation

`[p1 p2...pN]`

Distributed arithmetic uses a lookup table to store the sums of partial products. The size of the LUT grows exponentially with the order of the filter. You can divide the LUT into several partitions, where each LUT partition operates on a different set of filter taps. This division reduces the total size of the LUT logic.

To divide the LUT into `N` partitions, set this property to a vector of `N` integers that 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 filter length.

To generate DA code for your filter design without LUT partitioning, specify a scalar, whose value is equal to the filter length.

```
fdes = fdesign.lowpass('N,Fc,Ap,Ast',4,0.4,0.05,0.03,'linear');
Hd = design(fdes);
Hd.arithmetic = 'fixed';
generatehdl(Hd,'DALUTPartition',5)
```

The filter length is calculated differently depending on the filter type.

| Filter Type | Filter Length (FL) Calculation |
|---|---|
| Direct form | `FL = length(find(Hd.numerator~= 0))` |
| Direct form symmetric<br>Direct form asymmetric | `FL = ceil(length(find(Hd.numerator~= 0))/2)` |

For supported multirate filters, you can specify the LUT partition as:

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

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

| LUT Partition Specification | Filter Length (FL) Calculation |
|---|---|
| Vector, whose elements sum to the overall filter length, FL. | `FL = size(polyphase(Hm), 2)` |
| Matrix, where each row specifies the partitions for one subfilter. The vector elements in each row must sum to the associated subfilter length, FLi. | `p = polyphase(Hm)`<br>`FLi = length(find(p(i,:))),` where i is the index to the ith row of the polyphase matrix of the filter. The ith row of the matrix p represents the ith subfilter. |

For more information about distributed arithmetic, see "Distributed Arithmetic for FIR Filters" on page 4-21.

For examples, see "Distributed Arithmetic for Single Rate Filters" on page 9-10 and "Distributed Arithmetic for Multirate Filters" on page 9-11.

You can specify different DALUTPartition values for each stage of a cascaded filter. See "Distributed Arithmetic for Cascaded Filters" on page 9-11.

### DARadix — Number of bits processed simultaneously in distributed arithmetic (DA) implementation

2 (default) | positive power of two

This property specifies a degree of parallelism in the DA architecture, which can improve clock speed at the expense of area. By default, the coder implements a fully serial DA architecture, that processes one bit at a time (DARadix = $2^1$). The value of this property, N, must be:

- A nonzero positive integer that is a power of two.
- Such that `mod(W,log2(N)) = 0`, where W is the input word size of the filter.
- Less than $2^W$, where W is the input word size of the filter. This maximum specifies a fully parallel DA architecture.

Values of N between $2^1$ and $2^W$ specify partly serial DA. For more information on distributed arithmetic, see "Distributed Arithmetic for FIR Filters" on page 4-21.

When setting a `DARadix` value for symmetrical (`dfilt.dfsymfir`) and asymmetrical (`dfilt.dfasymfir`) filters, see “Considerations for Symmetric and Asymmetric Filters” on page 4-24.

You can specify different `DARadix` values for each stage of a cascaded filter. See “Distributed Arithmetic for Cascaded Filters” on page 9-11

### `FoldingFactor` — Folding factor of a serial architecture for IIR SOS filter

filter length (default) | integer

Use this property to define a serial architecture for direct-form I or direct-form II SOS filters. Specify the number of clock cycles, `N`, taken for the computation of filter output. The generated HDL code shares multipliers to reduce area at the cost of latency. You can specify either `NumMultipliers` or `FoldingFactor`, but not both. If you do not specify either `NumMultipliers` or `FoldingFactor`, the coder generates HDL code for the filter with a fully parallel architecture. For a command-line example, see “Serial Architecture for IIR Filter” on page 9-9. For a GUI example, see “Specifying Serial Architectures for IIR SOS Filters” on page 4-15.

### `NumMultipliers` — Number of multipliers in a serial architecture for IIR SOS filter

integer greater than 1

Use this property to define a serial architecture for direct-form I or direct-form II SOS filters. You can specify either `NumMultipliers` or `FoldingFactor`, but not both. If you do not specify either `NumMultipliers` or `FoldingFactor`, the coder generates HDL code for the filter with a fully parallel architecture. For a command-line example, see “Serial Architecture for IIR Filter” on page 9-9. For a GUI example see “Specifying Serial Architectures for IIR SOS Filters” on page 4-15.

## See Also

`generatehdl`

# Ports and Identifiers Properties

Customize ports, clocks, resets, identifiers, and comments

## Description

Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options in the Generate HDL dialog box.

### Ports, Clocks, and Resets

**`ClockEnableInputPort` — Name of clock enable port in generated HDL code**
`'clk_enable'` (default) | string

For example, generate HDL code for filter object `Hd`, with a custom name for the clock enable signal.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'ClockEnableInputPort','filter_clk_
```

The generated entity declaration replaces the default port name with your string.

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

The clock enable signal is asserted active high (1). Thus, drive this port high to activate the registers in the filter.

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.
- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

### ClockEnableOutputPort — Name of clock enable output port in generated HDL code, for multirate filters with single input clock

'ce_out' (default) | string

This option is available only when you design a multirate filter and use a single input clock (the default behavior). For example, generate HDL code for filter object Hd, with a custom name for the clock enable output port.

```
Hm = dsp.CICDecimator(7,1,4);
generatehdl(Hm,'InputDataType',numerictype(1,14,13),'ClockEnableOutputPort','filter_clk
```

The generated entity declaration replaces the default port name with your string.

```
ENTITY cicdecimfilt IS
   PORT( clk                             :   IN    std_logic;
         clk_enable                      :   IN    std_logic;
         reset                           :   IN    std_logic;
         filter_in                       :   IN    std_logic_vector(13 DOWNTO 0); -- sfix14_En13
         filter_out                      :   OUT   std_logic_vector(25 DOWNTO 0); -- sfix26_En13
         filter_clk_out                  :   OUT   std_logic
         );
END cicdecimfilt;
```

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.

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.

- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

### ClockInputPort — Name of clock input port in generated HDL code

'clk' (default) | string

For example, generate HDL code for filter object Hd, with a custom name for the clock input port.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'ClockInputPort','filter_clk');
```

The generated entity declaration replaces the default port name with your string.

```
ENTITY Hd IS
```

```
   PORT( filter_clk        :  IN  std_logic;
         clk_enable        :  IN  std_logic;
         reset             :  IN  std_logic;
         filter_in         :  IN  std_logic_vector (15 DOWNTO 0);
         filter_out        :  OUT std_logic_vector (15 DOWNTO 0);
         );
END Hd;
```

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.

- The ReservedWordPostfix property.

See "Resolving HDL Reserved Word Conflicts" on page 5-11.

### `InputPort` — Name of filter input port in generated HDL code

`'filter_in'` (default) | string

For example, generate HDL code for filter object `Hd`, with a custom name for the input data port.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'InputPort','filter_data_in');
```

The generated entity declaration replaces the default port name with your string.

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

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.

- The ReservedWordPostfix property.

See "Resolving HDL Reserved Word Conflicts" on page 5-11.

### `InputType` — Data type of filter input port in generated HDL code

`'std_logic_vector'` (default) | `'signed/unsigned'` | `'wire'`

If your target language is VHDL, choose between `'std_logic_vector'` and `'signed/unsigned'`.

If your target language is Verilog, the input data type is `'wire'`.

### `OutputPort` — Name of filter output port in generated HDL code

`'filter_out'` (default) | string

For example, generate HDL code for filter object `Hd`, with a custom name for the output data port.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'OutputPort','filter_data_out');
```

The generated entity declaration replaces the default port name with your string.

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

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.
- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

### `OutputType` — Data type of filter output port in generated HDL code

`'Same as input data type'` (default) | `'std_logic_vector'` | `'signed/unsigned'` | `'wire'`

If your target language is VHDL, choose between `'Same as input data type'`, `'std_logic_vector'`, and `'signed/unsigned'`.

If your target language is Verilog, the output data type is `'wire'`.

### ResetInputPort — Name of filter reset port in generated HDL code

'reset' (default) | string

For example, generate HDL code for filter object Hd, with a custom name for the reset port.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'ResetInputPort','filter_reset');
```

The generated entity declaration replaces the default port name with your string.

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

To control whether the reset port is active high (drive 1 to reset registers) or active low (drive 0 to reset registers), set the ResetAssertedLevel property.

---

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.

- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

---

### RemoveResetFrom — Suppress generation of resets from shift registers

'none' (default) | 'ShiftRegister'

By default, the coder includes reset signals for shift registers in the generated HDL code. Omitting reset signals from shift register code can result in a more efficient FPGA implementation. To disable resets on shift registers, set this property to 'ShiftRegister'. See “Suppressing Generation of Reset Logic” on page 5-25.

### ResetAssertedLevel — Asserted (active) level of reset input signal

'active-high' (default) | 'active-low'

By default, the reset input signal must be driven high (1) to reset registers in the filter design. 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'));
```

When you set this property to `'active-low'`, the reset input signal must be driven low (0) to reset registers in the filter design. For example, this code fragment checks whether `reset` is active low before populating the `delay_pipeline` register.

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

### `ResetType` — Use asynchronous or synchronous reset style in the generated HDL code for registers

`'async'` (default) | `'sync'`

By default, the coder uses 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_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;
```

When you set this property to `'sync'`, the coder uses a synchronous reset style. In this case, the process block checks for the rising edge of the clock before performing a reset.

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

## Identifiers and Comments

### BlockGenerateLabel — String to append to block section labels in VHDL GENERATE statements

'_gen' (default) | string

The coder appends this string to the block section labels of VHDL GENERATE statements.

### InstanceGenerateLabel — String to append to instance section labels in VHDL GENERATE statements

'_gen' (default) | string

The coder appends this string to the instance section labels of VHDL GENERATE statements.

### OutputGenerateLabel — String to append to output assignment block labels in VHDL GENERATE statements

'outputgen' (default) | string

The coder appends this string to the output assignment block labels of VHDL GENERATE statements.

### ClockProcessPostfix — String to append to HDL clock process names

'_process' (default) | string

The coder uses process blocks to modify the content of the registers in the filter. The label for each of these blocks is derived from a register name and the postfix `_process`. For example, in the following block declaration, the coder derives the process label from the register name `delay_pipeline` and the default postfix string `_process`.

```
delay_pipeline_process : PROCESS (clk, reset)
BEGIN
```

### CoeffPrefix — Prefix for filter coefficient names

'coeff' (default) | string

| Filter Type | Coefficient Prefix String |
|---|---|
| FIR | Each coefficient number, starting with 1. For example, the default for the first coefficient is `coeff1`. |
| IIR | An underscore (_) and an `a` or `b` coefficient name (for example, `_a2`, `_b1`, or `_b2`) followed by the string `_section`*n*, where *n* is the section number. For example, |

| Filter Type | Coefficient Prefix String |
|---|---|
|  | the default for the first numerator coefficient of the third section is `coeff_b1_section3`. |

For example:

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

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.
- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

### `ComplexImagPostfix` — String to append to imaginary part of complex signal names

`'_im'` (default) | string

The coder appends this string to the imaginary part of complex signals in the generated HDL code. See “Using Complex Data and Coefficients” on page 5-34.

### `ComplexRealPostfix` — String to append to imaginary part of complex signal names

`'_re'` (default) | string

The coder appends this string to the real part of complex signals in the generated HDL code. See “Using Complex Data and Coefficients” on page 5-34.

### `EntityConflictPostfix` — String to append to duplicate VHDL entity or Verilog module names

`'block'` (default) | string

The coder uses 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` — String prefixed to component instance names in generated HDL code

`'u_'` (default) | string

The coder prefixes component instance names in the generated HDL code with this string.

### `PackagePostfix` — String to append to filter name to form name of VHDL package file

`'_pkg'` (default) | string

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

### `ReservedWordPostfix` — String to append to identifiers that are VHDL or Verilog reserved words

`'_rsvd'` (default) | string

For example, if you name your filter `mod`, the coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

### `SplitEntityArch` — Separate files for generated VHDL entity and architecture code

`'off'` (default) | `'on'`

By default, the coder writes the generated entity and architecture code to a single file.

When you set this property to `'on'`, the coder creates the filter VHDL entity and architecture in two separate files. The coder derives the names of the entity and architecture files from the filter name. Postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch`) are appended to the base file name. To override the default and specify your own postfix string, set the `SplitArchFilePostfix` and `SplitEntityFilePostfix` properties.

### `SplitArchFilePostfix` — String to append to filter name to form name of the VHDL architecture file

`'_arch'` (default) | string

By default, the coder names the architecture file *`filtername`*`_arch`. This option applies when you set the `SplitEntityArch` property to `'on'`.

### `SplitEntityFilePostfix` — String to append to filter name to form name of the VHDL entity file

`'_entity'` (default) | string

By default, the coder names the entity file *`filtername`*`_entity`. This option applies when you set the `SplitEntityArch` property to `'on'`.

### UserComment — Comment line in header of generated filter and test bench files

string

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

You can add your own comment lines to the header comment block by setting UserComment to the desired string value. The code generator adds leading comment characters that correspond to the target language. When you include new lines or line feeds in the string, the coder emits single-line comments for each new line.

For example, this generatehdl command adds two comment lines to the header in a generated VHDL file.

```
generatehdl(Hlp,'UserComment','This is a comment line.\nThis is a second line.')
```

The resulting header comment block for filter Hlp is:

```
-- -------------------------------------------------------------
--
-- Module: Hlp
--
-- Generated by MATLAB(R) 7.11 and the Filter Design HDL Coder 2.7.
--
-- Generated on: 2010-08-31 13:32:16
--
-- This is a comment line.
-- This is a second line.
--
-- -------------------------------------------------------------

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

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

-- -------------------------------------------------------------
-- HDL Implementation    : Fully parallel
-- Multipliers           : 43
-- Folding Factor        : 1
-- -------------------------------------------------------------
-- Filter Settings:
```

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

### VectorPrefix — String prefix for vector names in generated VHDL code

'vector_of_' (default) | 'string'

The coder prefixes the names of vector signals in the VHDL code with this string. This property has no effect on generated Verilog code.

## See Also

generatehdl

# HDL Constructs Properties

Customize generated HDL style

## Description

You can customize the style of the generated VHDL and Verilog code using the properties on this page. Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options on the **Global Settings** > **Advanced** tab in the Generate HDL dialog box.

### HDL Coding Style

**`CastBeforeSum` — Enable or disable type casting of input values of addition or subtraction operations**
`'on'` (default) | `'off'`

When you set this property to `'off'`, the generated code preserves the types of input values during addition and subtraction operations and then converts the result to the desired type.

When you set this property to `'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.

The `CastBeforeSum` property is related to the setting of the FDATool 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, `CastBeforeSum` is effectively off when HDL code is generated; it is not relevant to the filter. In the Generate HDL dialog box for these filters, **Cast before sum** is disabled.
- When the filter object does have the **Cast signals before sum** property, by default the coder sets `CastBeforeSum` following the **Cast signals before sum** setting in the filter object. This setting is visible in the Generate HDL dialog box. If you change the setting of **Cast signals before sum** in FDATool, the coder updates the setting of **Cast before sum** in Generate HDL.

- To override the **Cast signals before sum** setting passed in from FDATool, set **Cast before sum** explicitly in the Generate HDL dialog box, or set the `CastBeforeSum` property when you call `generatehdl`.

See “Specifying Input Type Treatment for Addition and Subtraction Operations” on page 5-32.

### `InlineConfigurations` — Enable or disable generation of inline VHDL configurations

`'on'` (default) | `'off'`

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

When you set this property to `'on'`, the coder includes VHDL configurations in files that instantiate a component.

When you set this property to `'off'`, the coder does not generate configurations, and requires user-supplied external configurations. Use this setting if you are creating your own VHDL configuration files.

### `LoopUnrolling` — Include or unroll and omit `FOR` and `GENERATE` loops in generated VHDL code

`'off'` (default) | `'on'`

When you set this property to `'on'`, the coder unrolls and omits `FOR` and `GENERATE` loops from the generated VHDL code. Use this option if your EDA tool does not support `GENERATE` loops.

When you set this property to `'off'`, the generated VHDL code can contain `FOR` and `GENERATE` loops.

### `SafeZeroConcat` — Type-safe syntax for concatenated zeros

`'on'` (default) | `'off'`

When you set this property to `'on'`, the coder uses the `'0' & '0'` syntax for concatenated zeros. This syntax is recommended because it is unambiguous.

When you set this property to `'off'`, the coder uses the `"000000..."` syntax for concatenated zeros. This syntax can be easier to read and is more compact, but can lead to ambiguous types.

### UseAggregatesForConst — Enable or disable aggregate declaration of constants smaller than 32 bits wide

'off' (default) | 'on'

When you set this property to 'on', the coder represents constants by aggregates, including constants that are less than 32 bits wide. These VHDL declarations show 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');
```

When you set this property to 'off', the coder represents constants less than 32 bits as scalars, and constants greater than or equal to 32 bits as aggregates. These VHDL declarations show 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
```

### UseRisingEdge — VHDL coding style used to check for rising clock edges

'off' (default) | 'on'

When you set this property to 'on', the generated code uses the VHDL rising_edge function when operating on registers.

```
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 you set this property to 'off', the generated code checks for clock events when operating on registers.

```
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 two coding styles have different simulation behavior when the clock transitions from `'X'` to `'1'`.

### `UseVerilogTimescale` — Allow or exclude use of compiler `` `timescale `` directives in generated Verilog code

`'on'` (default) | `'off'`

When you set this property to `'on'`, the coder uses compiler `` `timescale `` directives in generated Verilog code.

When you set this property to `'off'`, the coder does not include `` `timescale `` directives in generated Verilog code.

The `` `timescale `` directive provides a way of specifying different delay values for multiple modules in a Verilog file.

## See Also

`generatehdl`

## More About

- “HDL Constructs” on page 5-27

# Test Bench Properties

Enable and customize generated test bench

## Description

The coder can optionally generate an HDL test bench that applies generated input stimuli to the HDL code generated for the filter. The test bench compares the output of the HDL filter with saved result vectors from MATLAB simulation. You can configure clocks, resets, input stimuli, and other test bench options using the properties on this page. Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options on the **Test Bench** tab in the Generate HDL dialog box.

### General

**`GenerateHDLTestBench` — Enable generation of a test bench**
`'on'` (default) | `'off'`

When you set this property to `'on'`, the coder generates a test bench for your HDL filter code. The test bench applies generated input stimuli to the HDL code generated for the filter. The test bench compares the output of the HDL filter with saved result vectors from MATLAB simulation. Configure the clock and reset behavior, input stimulus, and other test bench features using the properties on this page.

**Note:** The `generatetb` function was removed in R2011a. Instead, call `generatehdl` and set the `GenerateHDLTestbench` property to `'on'`.

**`TestBenchName` — File name for the generated test bench**
*`filtername`*`_tb` (default) | string

This name is also used for the VHDL entity or Verilog module. The coder creates the file in the location specified in the `TargetDirectory` property. The coder uses the file type extension defined by the VerilogFileExtension or VHDLFileExtension property.

**Avoiding Reserved Words in Names** If you specify a string that is a reserved word in the selected language, the coder appends the string specified by either:

- The **Reserved word postfix** option on the **Global Settings** > **General** tab of the Generate HDL dialog box.
- The ReservedWordPostfix property.

See “Resolving HDL Reserved Word Conflicts” on page 5-11.

### `ErrorMargin` — Error margin for test bench comparison with reference signals

4 (default) | positive integer (bits)

Some HDL optimizations can generate test bench code that produces numeric results that differ from the results produced by the original filter function. Such optimizations include:

- CastBeforeSum
- OptimizeForHDL
- FIRAdderStyle set to `'Tree'`
- 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.

### `MultifileTestBench` — Divide generated test bench into helper functions, data, and HDL test bench code files

`'off'` (default) | `'on'`

When you set this property to `'on'`, the coder writes separate files for test bench code, helper functions, and test bench data. The file names are derived from the `TestBenchName` and `TestBenchDataPostFix` properties. For example, if the test bench name 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 uses the file type extension defined by the VerilogFileExtension or VHDLFileExtension property.

When you set this property to 'off', the coder writes a single test bench file containing HDL test bench code, helper functions, and test bench data.

### TestBenchDataPostfix — String appended to test bench data file name, when generating multifile test bench

'_data' (default) | string

This property applies when you set MultifileTestBench to 'on'. If the name of your test bench is test_fir_tb, the coder adds the postfix _data to form the test bench data file name test_fir_tb_data.

### TestBenchReferencePostfix — String appended to the names of reference signals in the generated test bench, when generating multifile test bench

'_ref' (default) | string

This property applies when you set MultifileTestBench to 'on'. The generated test bench represents reference signal data as arrays. The test bench stores the reference signal values in the _data file.

```
CONSTANT filter_out_expected : filter_in_data_log_type :=
    (
         -2.4228738523269194E-03,
         -2.0832449820793104E-03,
         6.7703446401186345E-03,...
```

Then the test bench accesses one array value at a time for comparison. This postfix applies to the output signal in the _tb file.

```
 SIGNAL filter_out_ref                    : real := 0.0; -- double
...
 filter_out_ref <= filter_out_expected(TO_INTEGER(filter_out_addr));
```

## Clocks and Resets

### ClockHighTime — Period during which the test bench drives clock input signals high (1)

5 (default) | positive integer or floating-point (ns)

You can specify an integer or a double-precision floating-point value (with a maximum of 6 significant digits after the decimal point). This option applies only if ForceClock is set to 'on'.

### ClockLowTime — Period during which the test bench drives clock input signals low (0)

5 (default) | positive integer or floating-point (ns)

You can specify an integer or a double-precision floating-point value (with a maximum of 6 significant digits after the decimal point). This option applies only if `ForceClock` is set to `'on'`.

### `ForceClock` — Enable or disable the test bench forcing the clock input signals

`'on'` (default) | `'off'`

When you set this property to `'on'`, the test bench forces the clock input signals. When this option is set, the values of the `ClockHighTime` and `ClockLowTime` properties control the clock waveform.

When you set this property to `'off'`, you must drive the clock input signals from a user-defined external source.

### `ForceClockEnable` — Enable or disable the test bench forcing the clock enable input signals

`'on'` (default) | `'off'`

When you set this property to `'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.

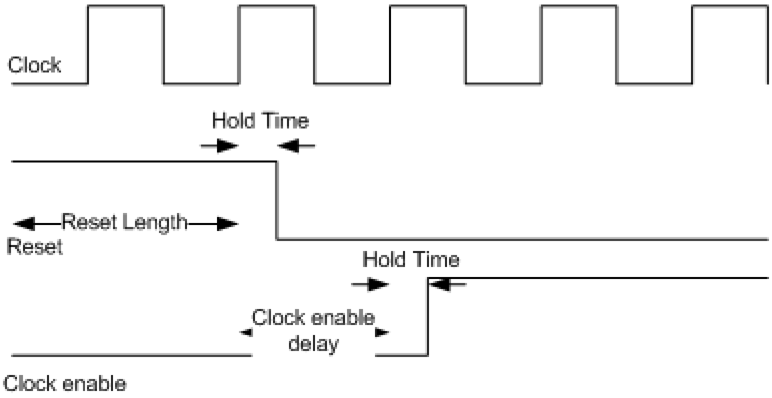
When you set this property to `'off'`, you must drive the clock enable input signals from a user-defined external source.

### `TestBenchClockEnableDelay` — Clock cycles between deassertion of reset and assertion of clock enable

1 (default) | positive integer (clock cycles)

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`.



### `ForceReset` — Enable or disable the test bench forcing the reset input signals

`'on'` (default) | `'off'`

When you set this property to `'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.

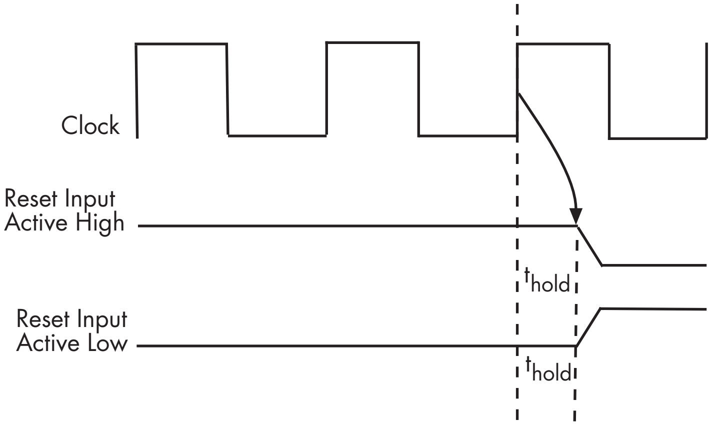
When you set this property to `'off'`, you must drive the reset input signals from a user-defined external source.

### `HoldTime` — Hold time for input data values and forced reset signals

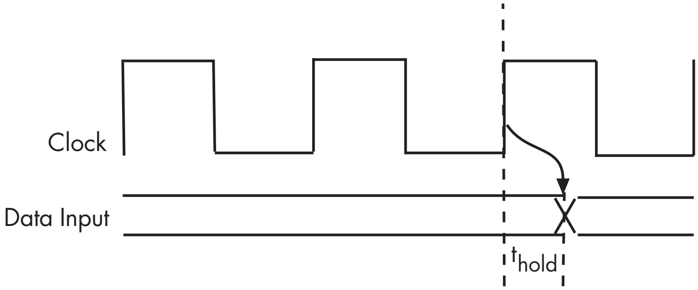
2 (default) | positive integer or floating-point (ns)

The test bench holds filter data input signals and forced reset input signals for this number of nanoseconds (ns) 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). This option applies to reset input signals only if `ForceReset` is set to `'on'`.

The hold time is the amount of time that reset input signals and input data are held past the clock rising edge. The following figures show the application of a hold time, $t_{hold}$, for reset and data input signals when the signals are forced to active high and active low. The `ResetLength` property is set to its default of 2 cycles, and the test bench asserts the reset signal for a total of 2 cycles plus $t_{hold}$.



**Hold Time for Reset Input Signals**

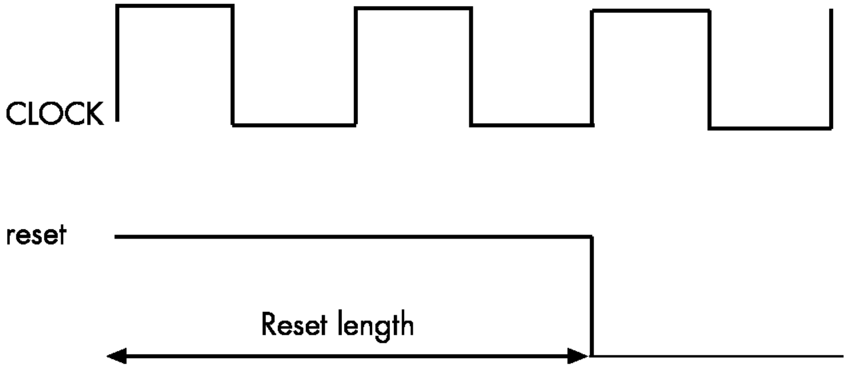


**Hold Time for Data Input Signals**

### `ResetLength` — Number of clock cycles that the test bench asserts the reset signal

2 (default) | positive integer (clock cycles)

The figure shows the default case. The test bench asserts an active-high reset signal for 2 clock cycles.



### `HoldInputDataBetweenSamples` — Determine how long the test bench holds input data of over-clocked filters in a valid state

`'off'` (default) | `'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 specifies the number of clock cycles that the test bench holds each input data value in a valid state.

- When you set this property to `'on'`, the generated test bench code holds input data values in a valid state across `N` clock cycles.
- When you set this property to `'off'`, the generated test bench code holds data values in a valid state for only 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.

### `InitializeTestBenchInputs` — Initial value driven on test bench inputs, before data starts

`'off'` (default) | `'on'`

When you set this property to `'on'`, the test bench drives zeros to the input ports at the start of the simulation.

When you set this property to `'off'`, the test bench drives an unknown state (expressed as `'X'`) to the input ports at the start of the simulation.

## Stimulus

### `TestBenchStimulus` — Input stimuli that generated test bench applies to your filter

`'impulse'` | `'step'` | `'ramp'` | `'chirp'` | `'noise'`

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. You can specify combinations of stimuli as strings in a cell array, in any order. For example:

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'GenerateHDLTestbench','on',...
           'TestBenchStimulus',{'ramp','impulse','noise'})
```

You can specify a custom input vector using the `TestBenchUserStimulus` property. When `TestBenchUserStimulus` is a nonempty vector, it takes priority over `TestBenchStimulus`.

### `TestBenchUserStimulus` — User-defined function that returns a vector of input data

`[]` (empty vector) (default) | function call

When this property is set to a non-empty vector, the generated test bench applies this input stimulus to your filter. Otherwise, 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 `generatehdl`.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'GenerateHDLTestbench','on',...
           'TestBenchUserStimulus',repmat([1 1 1 1 0 0 0 0],1,10))
```

### `TestBenchCoeffStimulus` — Stimulus for testing the coefficient memory interface for FIR or IIR filters

`[]` (empty vector) (default) | vector of coefficient values (FIR filters) | cell array of coefficient and scale values (IIR filters)

This property applies when you set CoefficientSource to `'ProcessorInterface'`.

- `[]` — The test bench loads the coefficients from the filter object and then 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 RAM as expected.

- FIR filter — Specify a vector of coefficient values. 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 coefficients specified by `TestBenchCoeffStimulus`, and processes the same input stimuli for a second time. In this case, the internal states of the filter, as set by the first run of the input stimulus, are retained. The test bench verifies that the interface writes two different sets of coefficients into the RAM. See “Programmable Filter Coefficients for FIR Filters” on page 3-30 and “Test Bench for FIR Filter with Programmable Coefficients” on page 9-13.
- IIR filter — Specify a cell array containing a column vector of scale values, and a second-order section (SOS) matrix for the filter. The test bench verifies that the interface writes two different sets of coefficients into the RAM, in the same way as an FIR filter. See “Programmable Filter Coefficients for IIR Filters” on page 3-40.

### `TestBenchFracDelayStimulus` — Input signal for the fractional delay port of a Farrow filter

constant obtained from the filter object (default) | `'RandSweep'` | `'RampSweep'` | vector or function call returning a vector

This property applies when generating a test bench for a single-rate Farrow filter. By default, the test bench drives the fractional delay input signal with a constant value obtained from the filter object. You can specify this input stimulus as a vector, a function returning a vector, or choose one of two predefined options:

`'RandSweep'` — 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'` — A vector of random values from 0 through 1. This stimulus signal has the same duration as the filter’s input signal, but changes at a slower rate. Each fractional delay value obtained from the vector is held for 10% of the total duration of the input signal.

See “Single-Rate Farrow Filters” on page 3-23.

### `TestBenchRateStimulus` — Rate input signal for CIC filter with optional rate port

maximum rate change factor (default) | integer

This property applies for variable-rate CIC filters, when you set AddRatePort to `'on'`. If you do not specify `TestBenchRateStimulus`, the coder uses the maximum rate-change factor specified in the filter object.

See “Variable Rate CIC Filters” on page 3-10.

## Cosimulation

### `GenerateCosimBlock` — Generate model containing HDL Cosimulation blocks for simulation of filter in Simulink

`'off'` (default) | `'on'`

When you set this property to `'on'`, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for each of Mentor Graphics ModelSim and Cadence Incisive. This feature requires an HDL Verifier license.

The coder configures the generated HDL Cosimulation blocks to conform to the port and data type interface of the filter selected for code generation. To cosimulate your design with the desired HDL simulator, copy the block corresponding to your HDL simulator into a Simulink model in place of the corresponding filter block.

### `GenerateCosimModel` — Generate model containing realized filter and HDL Cosimulation block for simulation of filter in Simulink

`'none'` (default) | `'ModelSim'` | `'Incisive'`

When you set this property to `'ModelSim'` or `'Incisive'`, the coder generates and opens a Simulink model that 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. This feature requires an HDL Verifier license.

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 6-29.

## See Also

`generatehdl`

## More About

- “Enabling Test Bench Generation” on page 6-9

# Compilation and Simulation Properties

Integrate third-party EDA tools into filter design workflow

## Description

The coder generates one script to compile your HDL files and one script to simulate the compiled HDL code. You can modify the commands that the coder prints to the script by setting the properties described on this page. The coder passes the property values to `fprintf` to create the script. You can use format strings supported by the `fprintf` function. For example, `'\n'` inserts a newline into the script file.

Specify these properties as `'Name',Value` arguments to the `generatehdl` function, or set the corresponding options in the Generate HDL dialog box.

To see these options in the Generate HDL dialog box, select the **EDA Tool Scripts** tab, and click **Compilation script** or **Simulation script** from the menu in the left column.

### Generate Scripts

**`EDAScriptGeneration` — Enable or disable generation of script files**
`'on'` (default) | `'off'`

Setting this property to `'off'` disables generation of compilation, simulation, synthesis, and lint scripts.

### Compilation

**`HDLCompileFilePostfix` — String appended to file name of generated compilation script**
`'_compile.do'` (default) | string

For example, if the name of the filter or test bench is `my_design`, the coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

**`HDLCompileInit` — Initialization section of compilation script**
`'vlib %s\n'` (default) | string

The implicit argument, `%s`, is the name of your library, VHDLLibraryName. By default, this string generates the library specification`'vlib work/n'`. If you compile your filter design with code from other libraries, use `VHDLLibraryName` to avoid library name conflicts.

### `HDLCompileVerilogCmd` — Command written to compilation script for each Verilog file

`'vlog %s %s\n'` (default) | string

This command adds your 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 two arguments are the contents of the `SimulatorFlags` property (an empty string, `''`, by default) and the file name of the current module.

### `HDLCompileVHDLCmd` — Command written to compilation script for each VHDL file

`'vcom %s %s\n'` (default) | string

This command adds your 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 two arguments are the contents of the `SimulatorFlags` property (an empty string, `''`, by default) and the file name of the current entity.

### `SimulatorFlags` — Simulator options written to compilation script

`''` (default) | string

Specify options that are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag `-93`. The coder adds the flags you specify with this option to the compilation command in generated EDA tool scripts. The compilation command string is specified by the `HDLCompileVHDLCmd` or `HDLCompileVerilogCmd` property.

### `HDLCompileTerm` — Termination section of compilation script

`''` (default) | string

The coder prints this string to the end of the compilation script. Add commands to this property to customize your script.

## Simulation

### `HDLSimFilePostfix` — String appended to file name of generated simulation script

`'_sim.do'` (default) | string

For example, if the name of the filter or test bench is `my_design`, the coder adds the postfix `_sim.do` to form the name `my_design_sim.do`.

### `HDLSimInit` — Initialization section of simulation script

`'onbreak resume\nonerror resume\n'` (default) | string

The coder appends this string to the beginning of the simulation script. Add commands to this property to customize your script.

### `HDLSimCmd` — Command written to simulation script

`'vsim -novopt %s.%s\n'` (default) | string

The two arguments are your library name and top-level module or entity name. If you are using VHDL, you can set the library name in the VHDLLibraryName property. If you are using Verilog, the library name is `'work'`. If you compile your filter design with code from other libraries, use `VHDLLibraryName` to avoid library name conflicts.

### `HDLSimViewWaveCmd` — Waveform viewing command written to simulation script

`'add wave sim:%s\n'` (default) | string

The implicit argument is a command that adds the signal paths for the DUT top-level input signals, output signals, and output reference signals.

### `HDLSimTerm` — Termination section of simulation script

`'run -all\n'` (default) | string

The coder appends this string to the end of the simulation script. Add commands to this property to customize your script.

## See Also

`generatehdl`

## More About

- “Integration with Third-Party EDA Tools” on page 6-36

# Synthesis Automation Properties

Control generation of script for HDL synthesis tool

## Description

When you specify an HDL synthesis tool, the coder generates a script to call that synthesis tool on your generated HDL code. You can modify the commands that the coder prints to the script using the properties on this page. The coder passes the property values to `fprintf` to create the script. You can use format strings 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, or set the corresponding options in the Generate HDL dialog box.

To see these options in the Generate HDL dialog box, select the **EDA Tool Scripts** tab, and click **Synthesis script** from the menu in the left column.

### Synthesis Automation

**`HDLSynthTool` — Synthesis tool for which the coder generates a script**
`'none'` (default) | `'ISE'` | `'Libero'` | `'Precision'` | `'Quartus'` | `'Synplify'` | `'Vivado'` | `'Custom'`

This property enables or disables generation of scripts for third-party synthesis tools. By default, the coder does not generate a synthesis script. To generate a script for one of the supported synthesis tools, set `HDLSynthTool` to one of the tool strings in the table. The coder uses tool-specific default values for the `HDLSynthCmd`, `HDLSynthInit`, and `HDLSynthTerm` properties. You can customize each of these properties according to your target device, constraints, and so on.

| HDLSynthTool Value | Synthesis Tool |
|---|---|
| `none` | N/A; script generation disabled |
| `'ISE'` | Xilinx® ISE |
| `'Vivado'` | Xilinx Vivado® |

| HDLSynthTool Value | Synthesis Tool |
|---|---|
| `'Libero'` | Microsemi® Libero® |
| `'Precision'` | Mentor Graphics Precision |
| `'Quartus'` | Altera® Quartus II |
| `'Synplify'` | Synopsys® Synplify Pro® |
| `'Custom'` | Varies; set the `HDLSynthCmd`, `HDLSynthInit`, and `HDLSynthTerm` properties to generate a script that supports your tool. |

### `HDLSynthFilePostfix` — String appended to file name for generated synthesis script

string

The default value of this property depends on your setting for `HDLSynthTool`.

For example, if the value of `HDLSynthTool` is `'Synplify'`, then `HDLSynthFilePostfix` defaults to the string `'_synplify.tcl'`. Then, if the name of the device under test is `my_design`, the coder adds the postfix `_synplify.tcl` to form the synthesis script file name `my_design_synplify.tcl`.

### `HDLSynthInit` — Initialization section of synthesis script

string

The coder prints this command at the beginning of the synthesis script. The default value of this property depends on your setting for `HDLSynthTool`. The implicit argument, `%s`, is the name of your top-level entity or module.

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
```

### `HDLSynthCmd` — Command written to synthesis script for each HDL file

string

This command adds your 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 your setting for `HDLSynthTool`. The implicit argument, `%s`, is the name of the HDL file.

For example, if you set `HDLSynthTool` to `'Quartus'`, this property defaults to `'set_global_assignment -name %s_FILE "$src_dir/%s"\n'`. The first implicit argument is the `TargetLanguage`, and the second is the name of the HDL file. The first argument is used only when your synthesis tool is set to `'Quartus'`.

### `HDLSynthTerm` — Termination section of synthesis script

string

The default value of this property depends on your setting for `HDLSynthTool`. This section of the script has no implicit argument.

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
```

## See Also

`generatehdl`

## Related Examples

- "Generate Default Altera Quartus II Synthesis Script" on page 9-16
- "Construct Customized Synthesis Script" on page 9-16

## More About

- "Automation Scripts for Third-Party Synthesis Tools" on page 7-2

9

# Function Reference

# fdhdltool

Open Generate HDL dialog box

## Syntax

```
fdhdltool(Hd,nt)
fdhdltool(Hd)
```

## Description

`fdhdltool(Hd,nt)` opens the Generate HDL dialog box to set options and generate HDL for a filter System object, `Hd`. `nt` is the numeric type of the input data to the filter.

When the Generate HDL 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. Changes made to the original filter after you call `fdhdltool` 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(Hd)` opens the Generate HDL dialog box to set options and generate HDL for a `dfilt` filter, `Hd`.

## Input Arguments

### `Hd` — Filter object

filter System object, or `dfilt` object

Filter object for which to generate HDL, for instance, as returned by the `design` function. If `Hd` is a System object, you must specify the input data type, `nt`.

### `nt` — Specify input data type for System objects

object of `numerictype` class

This argument is required when the input filter, `Hd`, is a System 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.

# Examples

### Open the Generate HDL Dialog Box for a Filter Design

Design a filter System object™.

```
Fs = 96e3;
filtSpecs = fdesign.lowpass(20e3,22.05e3,1,80,Fs);
FIRLowpass = design(filtSpecs,'equiripple','filterstructure','dfsymfir','SystemObject',
```

Choose 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)
```

# More About

- “Opening the Filter Design HDL Coder GUI Using the `fdhdltool` Command” on page 2-11

### Introduced in R2007a

# generatehdl

Generate HDL code for quantized filter

## Syntax

```
generatehdl(Hd,'InputDataType',nt)
generatehdl(Hd)
generatehdl(___,Name,Value)
```

## Description

`generatehdl(Hd,'InputDataType',nt)` generates HDL code for a filter System object, `Hd`, using the default settings. `nt` is the data type of the input signal, specified as a `numerictype` object.

- The function places generated files in a subfolder named `hdlsrc`, inside your current working folder.
- The function includes the entity declaration and architecture code in a single source file.

`generatehdl(Hd)` generates HDL code for a `dfilt` filter, `Hd`, using default settings.

`generatehdl(___,Name,Value)` generates HDL code with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Generate HDL Code for FIR Equiripple Filter

Call `fdesign` to pass the specifications for designing a minimum order lowpass filter.

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

Those specifications determine the following characteristics for this filter:

- Normalized passband frequency of 0.2
- Stopband frequency of 0.22
- Passband ripple of 1 dB
- Stopband attenuation of 60 dB

Call the `design` function to create a FIR equiripple filter, `Hd`. The function returns a `dsp.FIRFilter` object.

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

Call `generatehdl` to generate VHDL code for the FIR equiripple filter. When the filter is a System object, you must specify the input data type.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'Name','MyFilter')
```

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

The function names the file `MyFilter.vhd` and places it in the default target folder, `hdlsrc`.

## Generate HDL Code and Test Bench for Lowpass Filter

Call `fdesign` to pass the specifications for designing a minimum order lowpass filter.

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

Call the `design` function to create a FIR equiripple filter, `Hd`. The function returns a `dsp.FIRFilter` object.

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

Call `generatehdl` to generate VHDL code and VHDL test bench for the FIR equiripple filter. When the filter is a System object, you must specify the input data type.

```
generatehdl(Hd,'InputDataType',numerictype(1,16,15),'Name','MyFilter','GenerateHDLTestb
```

```
### Starting VHDL code generation process for filter: MyFilter
### Generating: H:\hdlsrc\MyFilter.vhd
### 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: H:\hdlsrc\MyFilterTB.vhd
### Creating stimulus vectors ...
### Done generating VHDL Test Bench.
```

The function names the files `MyFilter.vhd` and `MyFilterTB.vhd`, and places them in the default target folder, `hdlsrc`.

## Fully Parallel FIR Filter with Programmable Coefficients

Generate VHDL code for a direct-form symmetric FIR filter with fully parallel (default) architecture and programmable coefficients. The coder generates a processor interface for the coefficients.

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

The coder generates this VHDL entity for the filter object `Hd`.

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

END Hd;
```

## Partly Serial FIR Filter with Programmable Coefficients

Create an asymmetric filter, `Hd`, and generate VHDL code for the filter. Specify a partly serial architecture. The coder only reacts to `CoefficientMemory` when you also set `CoefficientSource` to `ProcessorInterface`. The generated code includes a dual-port RAM interface for the programmable coefficients.

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

## Compare Serial Architectures for FIR Filter

Create a direct-form FIR filter.

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

For comparison, generate a default fully parallel architecture.

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

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

Generate a fully serial architecture by setting the partition size to the total filter length. Notice that the system clock rate is nine times the input sample rate. Also, the HDL latency reported is one sample greater than the default parallel implementation.

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

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

Generate a partly serial architecture with three equal partitions. This architecture uses three multipliers. The clock rate is three times the input rate, and the latency is the same as the default parallel implementation.

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

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

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

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

You can also generate a cascade-serial architecture without specifying the partitions explicitly. The coder automatically selects partition sizes.

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

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

## Serial Partitions for Cascaded Filter

Create a two-stage cascade filter and define different serial partitions for each stage. Specify the partition vectors in a cell array.

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

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.

You can use the `hdlfilterserialinfo` function to display the effective filter length and partitioning options for each filter stage of a cascade. 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(Hc.stage(1))
```

```
   | 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]                  |
```

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.

```
sp1 = hdlfilterserialinfo(Hc.stage(1),'Multiplier',2)
sp2 = hdlfilterserialinfo(Hc.stage(2),'Multiplier',3)
generatehdl(Hc,'serialpartition',{sp1,sp2})
```

Each stage uses a different clock rate based on the number of multipliers. The coder generates a timing controller to derive these clocks.

## Serial Architecture for IIR Filter

Create a direct-form II SOS filter.

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

To find possible serial architecture specifications, use the helper function.

```
hdlfilterserialinfo(Hd)
```

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

   | Folding Factor | Multipliers |
   --------------------------------
   |        6       |      3      |
   |        9       |      2      |
   |       18       |      1      |
```

Call `generatehdl` and request one of the serial architectures by specifying either the `NumMultipliers` or `FoldingFactor` property, but not both.

```
generatehdl(Hd,'NumMultipliers',2)
```

Alternatively, specify the same architecture with the `FoldingFactor` property.

```
generatehdl(Hd,'FoldingFactor',9)
```

For either of these calls to `generatehdl`, the coder generates 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

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')
Hd = design(filtdes,'filterstructure','dffir')
Hd.arithmetic = 'fixed'
FL = length(find(Hd.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(Hd,'DALUTPartition',[8 8 8 7])
```

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')
Hd = design(filtdes,'filterstructure','dfsymfir')
Hd.arithmetic = 'fixed'
FL = ceil(length(find(Hd.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.

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

**Tip** Use the `hdlfilterdainfo` function to display the effective filter length, LUT partitioning options, and possible `DARadix` values for a filter.

## Distributed Arithmetic for Multirate Filters

Create a direct-form FIR polyphase decimator, and calculate the filter length, FL.

```
d = fdesign.decimator(4);
Hm = design(d,'Systemobject',true)
FL = size(polyphase(Hm),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(Hm,'InputDataType',numerictype(1,16,15),'DALUTPartition',[8 8 8 3])
```

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.

```
d = fdesign.decimator(4);
Hm = design(d,'Systemobject',true)
generatehdl(Hm,'InputDataType',numerictype(1,16,15),'DALUTPartition',[1 0 0 0; 8 8 8 2; 8 8 6 4; 8 8 8 2])
```

**Tip** Use the hdlfilterdainfo function to display the effective filter length, LUT partitioning options, and possible DARadix values for a filter.

## Distributed Arithmetic for Cascaded Filters

Create a two-stage cascade filter and define different LUT partitions for each stage. Specify the partition vectors in a cell array.

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

You can also specify different DARadix values for each filter in a cascade. Specify DARadix as a cell array.

```
generatehdl(Hc,'DALUTPartition',{[5 4],[3 3 3]}, 'DARadix',{2^8,2^4})
```

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.

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(Hc.stage(1))
```

```
   | Total Coefficients | Zeros | Effective |
   ------------------------------------------
   |          9         |   0   |     9     |

Effective filter length for SerialPartition value is 9.

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

   | Folding Factor | LUT-Sets Multiple | DARadix |
   ------------------------------------------------
   |        1       |        16         |   2^16  |
   |        2       |        8          |   2^8   |
   |        4       |        4          |   2^4   |
   |        8       |        2          |   2^2   |
   |       16       |        1          |   2^1   |

  Details of LUTs with corresponding 'DALUTPartition' values.

   | Max Address Width | Size(bits) |               LUT Details               | DALUTPar
   ------------------------------------------------------------------------------------
   |         9         |    9216    |1x512x18                                 |[9]
   |         8         |    4628    |1x256x18, 1x2x10                         |[8 1]
   |         7         |    2352    |1x128x18, 1x4x12                         |[7 2]
   |         6         |    1192    |1x64x17, 1x8x13                          |[6 3]
   |         5         |     800    |1x16x16, 1x32x17                         |[5 4]
   |         4         |     548    |1x16x16, 1x16x17, 1x2x10                 |[4 4 1]
   |         3         |     344    |2x8x13, 1x8x17                           |[3 3 3]
   |         2         |     252    |1x2x10, 1x4x12, 1x4x13, 1x4x16, 1x4x17   |[2 2 2 2

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

Select address widths and folding factors to obtain LUT partition vectors for each stage.

```
dp1 = hdlfilterdainfo(Hc.stage(1),'LUTInputs',5,'FoldingFactor',4)
dp2 = hdlfilterdainfo(Hc.stage(2),'LUTInputs',3,'FoldingFactor',4)
generatehdl(Hc,'DALUTPartition',{dp1,dp2})
```

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.

## Cascaded Filter with Multiple Architectures

You can specify a mix of serial, distributed arithmetic (DA), and parallel architectures depending upon your hardware constraints. Create a three-stage filter. Each stage is a different type.

```
h1 = dfilt.dffir([0.05 -.25 .88 0.9 .88 -.25 0.05])
h1.Arithmetic = 'fixed'
h2 = dfilt.dfasymfir([-0.008 0.06 -0.44 0.44 -0.06 0.008])
h2.Arithmetic = 'fixed'
h3 = dfilt.dfsymfir([-0.008 0.06 0.44 0.44 0.06 -0.008])
h3.Arithmetic = 'fixed'
Hd = cascade(h1,h2,h3)
```

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. Set the property values as cell arrays, where each cell applies to a stage. Use the default values —`-1` for the partitions and `2` for `DARadix`— to disable a property for a particular stage.

```
generatehdl(Hd,'SerialPartition',{-1,3,-1},...
               'DALUTPartition',{[4 3],-1,-1},...
               'DARadix',{2^8,2,2})
```

## Test Bench for FIR 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))
hd = dfilt.dfsymfir(b)
```

```
generatehdl(hd,'GenerateHDLTestbench','on','CoefficientSource','ProcessorInterface',...
'TestbenchCoeffStimulus',c)
```

## IIR Filter with Programmable Coefficients

Generate VHDL code for an SOS IIR Direct Form II filter with programmable coefficients.

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

The coder generates this VHDL entity for the filter object `Hd`.

```
ENTITY Hd IS
PORT( clk               :   IN    std_logic;
      clk_enable        :   IN    std_logic;
      reset             :   IN    std_logic;
      filter_in         :   IN    std_logic_vector(15 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_En12
      );

END Hd;
```

## Clock Ports for Multirate Filters

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.

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

```
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); -- s
         filter_out                      :   OUT   std_logic_vector(35 DOWNTO 0); -- st
         ce_in                           :   OUT   std_logic;
         ce_out                          :   OUT   std_logic
         );

END firrc;
```

You can provide custom names for the input clock enable and the output clock enable signals. You cannot rename the `ce_in` signal.

```
frac_cvrter = dsp.FIRRateConverter('InterpolationFactor',5,'DecimationFactor',3)
generatehdl(frac_cvrter,'InputDataType',numerictype(1,16,15),...
            'ClockEnableInputPort','clk_en1','ClockEnableOutputPort','clk_en2')
```

```
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); -- st
         filter_out                      :   OUT   std_logic_vector(35 DOWNTO 0); -- st
         ce_in                           :   OUT   std_logic;
         clk_en2                         :   OUT   std_logic
         );

END firrc;
```

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')
```

```
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); -- st
         clk1                            :   IN    std_logic;
         clk_enable1                     :   IN    std_logic;
         reset1                          :   IN    std_logic;
         filter_out                      :   OUT   std_logic_vector(27 DOWNTO 0)  -- st
         );

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);
Hd = design(lpf,'equiripple','FilterStructure','dfsymfir','Systemobject',true);
generatehdl(Hd,'InputDataType',numerictype(1,14,13),'HDLSynthTool','Quartus');
```

The coder generates a script file named `firfilt_quartus.tcl`, using the default script properties for the Altera Quartus II synthesis tool.

```
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

This example sets the script automation properties to dummy values to illustrate how the coder constructs the synthesis script from the properties.

```
lpf = fdesign.lowpass('fp,fst,ap,ast',0.45,0.55,1,60);
Hd = design(lpf,'equiripple','FilterStructure','dfsymfir','Systemobject',true);
generatehdl(Hd,'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');
```

The coder generates a script file named `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

**`Hd` — Filter object**
filter System object, or `dfilt` object

Filter object for which to generate HDL, for instance, the object returned by the `design` function. If `Hd` is a System object, you must specify `InputDataType`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'TargetLanguage','Verilog'`

## Data Types

**`'InputDataType'` — Specify input data type for System objects**
object of `numerictype` class

This argument is required when the input filter, `Hd`, is a System 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. For more information, see InputDataType.

## Language Selection

**`'TargetLanguage'` — Target language**
`'VHDL'` (default) | `'Verilog'`

For more information, see TargetLanguage.

## File Naming and Location

**`'Name'` — Specify file name for generated HDL code and for filter VHDL entity or Verilog module**
string

For more information, see Name.

**`'TargetDirectory'` — Output folder**
`'hdlsrc'` (default) | string

For more information, see TargetDirectory.

**`'VerilogFileExtension'` — Verilog file extension**
`'.v'` (default) | string

For more information, see VerilogFileExtension.

**`'VHDLFileExtension'` — VHDL file extension**
`'.vhd'` (default) | string

For more information, see VHDLFileExtension.

## Resets

**`'RemoveResetFrom'` — Suppress generation of resets from shift registers**
`'none'` (default) | `'ShiftRegister'`

For more information, see RemoveResetFrom.

**`'ResetAssertedLevel'` — Asserted (active) level of reset**
`'active-high'` (default) | `'active-low'`

For more information, see ResetAssertedLevel.

**`'ResetLength'` — Define length of time (in clock cycles) during which reset is asserted**
2 (default) | `N`

For more information, see ResetLength.

**`'ResetType'` — Reset type**
`'async'` (default) | `'sync'`

For more information, see ResetType.

## Header Comment and General Naming

**`'ClockProcessPostfix'` — Postfix for clock process names**
`'_process'` (default) | string

For more information, see ClockProcessPostfix.

### `'CoeffPrefix'` — Specify prefix (string) for filter coefficient names

`'coeff'` (default) | string

For more information, see CoeffPrefix.

### `'ComplexImagPostfix'` — Postfix for imaginary part of complex signal

`'_im'` (default) | string

For more information, see ComplexImagPostfix.

### `'ComplexRealPostfix'` — Postfix for imaginary part of complex signal names

`'_re'` (default) | string

For more information, see ComplexRealPostfix.

### `'EntityConflictPostfix'` — Postfix for duplicate VHDL entity or Verilog module names

`'_block'` (default) | string

For more information, see EntityConflictPostfix.

### `'InstancePrefix'` — Prefix for generated component instance names

`'u_'` (default) | string

For more information, see InstancePrefix.

### `'PackagePostfix'` — Postfix for package file name

`'_pkg'` (default) | string

For more information, see PackagePostfix.

### `'ReservedWordPostfix'` — Postfix for names conflicting with VHDL or Verilog reserved words

`'_rsvd'` (default) | string

For more information, see ReservedWordPostfix.

### `'SplitArchFilePostfix'` — Postfix for VHDL architecture file names

`'_arch'` (default) | string

For more information, see SplitArchFilePostfix.

**`'SplitEntityArch'` — Split VHDL entity and architecture into separate files**
`'off'` (default) | `'on'`

For more information, see SplitEntityArch.

**`'SplitEntityFilePostfix'` — Postfix for VHDL entity file names**
`'_entity'` (default) | string

For more information, see SplitEntityFilePostfix.

**`'UserComment'` — HDL file header comment**
string

For more information, see UserComment.

**`'VectorPrefix'` — Prefix for vector names**
`'vector_of_'` (default) | string

For more information, see VectorPrefix.

**`'BlockGenerateLabel'` — Block label for HDL `GENERATE` statements**
`_gen` (default) | string

For more information, see BlockGenerateLabel.

**`'InstanceGenerateLabel'` — Instance section label postfix for VHDL `GENERATE` statements**
`'_gen'` (default) | string

For more information, see InstanceGenerateLabel.

**`'OutputGenerateLabel'` — Output assignment label postfix for VHDL `GENERATE` statements**
`'outputgen'` (default) | string

For more information, see OutputGenerateLabel.

**`'VHDLArchitectureName'` — VHDL architecture name**
`'rtl'` (default) | string

For more information, see VHDLArchitectureName.

**`'VHDLLibraryName'` — VHDL library name**
`'work'` (default) | string

For more information, see VHDLLibraryName.

## Ports

### `'AddInputRegister'` — Extra register indicator to HDL code for filter input

`'on'` (default) | `'off'`

For more information, see AddInputRegister.

### `'AddOutputRegister'` — Generate extra register in HDL code for filter output

`'on'` (default) | `'off'`

For more information, see AddOutputRegister.

### `'ClockEnableInputPort'` — HDL port name for filter clock enable input signals

`clk_enable` (default) | string

For more information, see ClockEnableInputPort.

### `'ClockEnableOutputPort'` — Name of clock enable output port for multirate filters with a single clock

`ce_out` (default) | string

For more information, see ClockEnableOutputPort.

### `'ClockInputPort'` — HDL port name for filter clock input signals

`clk` (default) | string

For more information, see ClockInputPort.

### `'InputPort'` — Name HDL port for filter input signals

`'filter_in'` (default) | string

For more information, see InputPort.

### `'InputType'` — Specify HDL data type for filter input port

`'std_logic_vector'` | `'signed/unsigned'` | `'wire'` (Verilog)

For more information, see InputType.

### `'OutputPort'` — Name HDL port for filter output signals

`'filter_out'` (default) | string

For more information, see OutputPort.

**'OutputType' — Specify HDL data type for filter output port**
'Same as input data type' (VHDL default) | 'std_logic_vector' | 'signed/unsigned' | 'wire' (Verilog)

For more information, see OutputType.

**'ResetInputPort' — Name HDL port for filter reset input signals**
'reset' (default) | string

For more information, see ResetInputPort.

## Filter Configuration

**'CoefficientSource' — Specify source for FIR or IIR filter coefficients**
'Internal' (default) | 'ProcessorInterface'

For more information, see CoefficientSource.

**'CoefficientMemory' — Specify type of memory for storage of programmable coefficients for serial FIR filters settings**
'Registers' (default) | 'DualPortRAMs' | 'SinglePortRAMs'

For more information, see CoefficientMemory.

**'ClockInputs' — Generation of single or multiple clock inputs for multirate filters**
'Single' (default) | 'Multiple'

For more information, see ClockInputs.

**'FracDelayPort' — Name port for Farrow filter fractional delay input signal**
'filter_fd' (default) | string

For more information, see FracDelayPort.

**'InputComplex' — Enable generation ports and signal paths that correspond to filters with complex input data**
'off' (default) | 'on'

For more information, see InputComplex.

**`'AddRatePort'` — Generate rate ports for variable-rate CIC filter**
`'off'` (default) | `'on'`

For more information, see AddRatePort.

## Advanced Coding

**`'CastBeforeSum'` — Type casting of input values enable or disable for addition and subtraction operations**
`'off'` (default) | `'on'`

For more information, see CastBeforeSum.

**`'InlineConfigurations'` — Include VHDL configurations**
`'on'` (default) | `'off'`

For more information, see InlineConfigurations.

**`'LoopUnrolling'` — Unroll VHDL `FOR` and `GENERATE` loops**
`'off'` (default) | `'on'`

For more information, see LoopUnrolling.

**`'SafeZeroConcat'` — Type-safe syntax for concatenated zeros**
`'on'` (default) | `'off'`

For more information, see SafeZeroConcat.

**`'UseAggregatesForConst'` — Represent constant values with aggregates**
`'off'` (default) | `'on'`

For more information, see UseAggregatesForConst.

**`'UseRisingEdge'` — Use VHDL `rising_edge` function to clock registers**
`'off'` (default) | `'on'`

For more information, see UseRisingEdge.

**`'UseVerilogTimescale'` — Generate `` `timescale `` compiler directives**
`'on'` (default) | `'off'`

For more information, see UseVerilogTimescale.

## Optimizations

### `'AddPipelineRegisters'` — Add pipeline register indicator for optimizing filter code clock rate

`'off'` (default) | `'on'`

For more information, see AddPipelineRegisters.

### `'CoeffMultipliers'` — Specify technique used for processing coefficient multiplier operations

`'multiplier'` (default) | `'csd'` | `'factored-csd'`

For more information, see CoeffMultipliers.

### `'DALUTPartition'` — Specify number and size of LUT partitions for distributed arithmetic architecture

`[p1 p2...pN]`, a vector of N integers

For more information, see DALUTPartition.

### `'DARadix'` — Specify number of bits processed simultaneously in distributed arithmetic architecture

2 (default) | `N`, a nonzero positive integer that is a power of two

For more information, see DARadix.

### `'FIRAdderStyle'` — Specify final summation technique used for FIR filters

`'linear'` (default) | `'tree'`

For more information, see FIRAdderStyle.

### `'FoldingFactor'` — Specify folding factor for IIR SOS filter with serial architecture

integer greater than 1

For more information, see FoldingFactor.

### `'MultiplierInputPipeline'` — Specify number of pipeline stages at multiplier inputs for FIR filters

`0` (default) | integer

For more information, see MultiplierInputPipeline.

### `'MultiplierOutputPipeline'` — Specify number of pipeline stages at multiplier outputs for FIR filters

`0` (default) | integer

For more information, see MultiplierOutputPipeline.

### `'NumMultipliers'` — Specify multipliers for IIR SOS filter with serial architecture

integer greater than 1

For more information, see NumMultipliers.

### `'OptimizeForHDL'` — Specify whether generated HDL code is optimized for specific performance or space requirements

`'off'` (default) | `'on'`

For more information, see OptimizeForHDL.

### `'ReuseAccum'` — Enable accumulator reuse, generating cascade-serial architecture for FIR filters

`'off'` (default) | `'on'`

For more information, see ReuseAccum.

### `'SerialPartition'` — Specify number and size of partitions generated for serial filter architectures

`[p1 p2 p3...pN]`, a vector of N integers

For more information, see SerialPartition.

## Test Bench

### `'ClockHighTime'` — Period during which test bench drives clock input signals high (1)

`5` (default) | `ns`

For more information, see ClockHighTime.

### `'ClockLowTime'` — Specify period, in nanoseconds, during which test bench drives clock input signals low (0)

`5` (default) | `ns`

For more information, see ClockLowTime.

### `'ErrorMargin'` — Specify error margin for HDL language-based test benches

4 (bits) (default)

For more information, see ErrorMargin.

### `'ForceClock'` — Specify whether test bench forces clock input signals

`'on'` (default) | `'off'`

For more information, see ForceClock.

### `'ForceClockEnable'` — Specify whether test bench forces input signals for clock enable

`'on'` (default) | `'off'`

For more information, see ForceClockEnable.

### `'ForceReset'` — Specify whether test bench forces reset input signals

`'on'` (default) | `'off'`

For more information, see ForceReset.

### `'GenerateCoSimBlock'` — Generate HDL Cosimulation block

`'off'` (default) | `'on'`

For more information, see GenerateCosimBlock.

### `'GenerateCoSimModel'` — Generate HDL Cosimulation model

`'ModelSim'` (default) | `'Incisive'`

For more information, see GenerateCosimModel.

### `'GenerateHDLTestbench'` — Enable generation of a test bench

`'off'` (default) | `'on'`

For more information, see GenerateHDLTestBench.

### `'HoldInputDataBetweenSamples'` — Specify how long input data values are held in valid state

`'on'` (default) | `'off'`

For more information, see HoldInputDataBetweenSamples.

### `'HoldTime'` — Specify hold time for filter data input signals and forced reset input signals

5 (ns) (default)

For more information, see HoldTime.

**`'InitializeTestBenchInputs'` — Specify initial value driven on test bench inputs before data is asserted to filter**
`'off'` (default) | `'on'`

For more information, see InitializeTestBenchInputs.

**`'MultifileTestBench'` — Divide generated test bench into helper functions, data, and HDL test bench code files**
`'off'` (default) | `'on'`

For more information, see MultifileTestBench.

**`'TestBenchClockEnableDelay'` — Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable**
`1` (default) | integer number of clock cycles

For more information, see TestBenchClockEnableDelay.

**`'TestBenchCoeffStimulus'` — Specify testing options for coefficient memory interface for FIR or IIR filters**
empty vector (default) | vector

For more information, see TestBenchCoeffStimulus.

**`'TestBenchDataPostFix'` — Specify suffix added to test bench data file name when generating multifile test bench**
`'_data'` (default) | string

For more information, see TestBenchDataPostfix.

**`'TestBenchFracDelayStimulus'` — Specify input stimulus that test bench applies to Farrow filter fractional delay port**
constant (either `'RandSweep'` or `'RampSweep'`) (default) | vector or function returning a vector

For more information, see TestBenchFracDelayStimulus.

**`'TestBenchName'` — Name VHDL test bench entity or Verilog module and file that contains test bench code**
string

For more information, see TestBenchName.

**`'TestBenchRateStimulus'` — Specify rate stimulus for CIC filter with rate port**
integer

For more information, see TestBenchRateStimulus.

**`'TestBenchReferencePostFix'` — Specify string appended to names of reference signals generated in test bench code**
`'_ref'` (default) | string

For more information, see TestBenchReferencePostfix.

**`'TestBenchStimulus'` — Specify input stimuli that test bench applies to filter**
`'impulse'` | `'step'` | `'ramp'` | `'chirp'` | `'noise'`

For more information, see TestBenchStimulus.

**`'TestBenchUserStimulus'` — Specify user-defined function that returns vector of values that test bench applies to filter**
function call

For more information, see TestBenchUserStimulus.

## Script Generation

**`'EDAScriptGeneration'` — Enable or disable script generation for third-party tools**
`'on'` (default) | `'off'`

For more information, see EDAScriptGeneration.

**`'HDLCompileFilePostfix'` — String appended to file name of generated compilation script**
`'_compile.do'` (default) | string

For more information, see HDLCompileFilePostfix.

**`'HDLCompileInit'` — Initialization section of compilation script**
`'vlib work\n'` (default) | string

For more information, see HDLCompileInit.

**`'HDLCompileTerm'` — Termination section of compilation script**
`''` (default) | string

For more information, see HDLCompileTerm.

**`'HDLCompileVerilogCmd'` — Command written to compilation script for each Verilog file**
`'vlog %s %s\n'` (default) | string

For more information, see HDLCompileVerilogCmd.

**`'HDLCompileVHDLCmd'` — Command written to compilation script for each VHDL file**
`'vcom %s %s\n'` (default) | string

For more information, see HDLCompileVHDLCmd.

**`'HDLSimCmd'` — Command written to simulation script, between initialization and termination sections**
`'vsim -novopt %s.%s\n'` (default) | string

For more information, see HDLSimCmd.

**`'HDLSimFilePostfix'` — String appended to file name for generated simulation scripts**
`'_sim.do'` (default) | string

For more information, see HDLSimFilePostfix.

**`'HDLSimInit'` — Initialization section of simulation script**
`['onbreak resume\n',...`
`'onerror resume\n']` (default) | string

For more information, see HDLSimInit.

**`'HDLSimTerm'` — Termination section of simulation script**
`'run -all\n'` (default) | string

For more information, see HDLSimTerm.

**`'HDLSimViewWaveCmd'` — Waveform viewing command written to simulation script**
`'add wave sim:%s\n'` (default) | string

For more information, see HDLSimViewWaveCmd.

**`'HDLSynthCmd'` — Command written to synthesis script, between initialization and termination sections**
string

See HDLSynthCmd.

**`'HDLSynthFilePostfix'` — String appended to file name for generated synthesis script**
string

See HDLSynthFilePostfix.

**`'HDLSynthInit'` — Initialization section of synthesis script**
string

See HDLSynthInit.

**`'HDLSynthTerm'` — Termination section of synthesis script**
string

See HDLSynthTerm.

**`'HDLSynthTool'` — Synthesis tool for which the coder generates a script**
`'none'` (default) | `‘ISE’` | `‘Libero’` | `'Precision'` | `'Quartus'` | `'Synplify'` | `‘Vivado’` | `‘Custom’`

See HDLSynthTool.

**`'SimulatorFlags'` — Specify simulator flags applied to generated test bench**
string

For more information, see SimulatorFlags.

## See Also

`generatetbstimulus`

**Introduced before R2006a**

# generatetbstimulus

Generate and return HDL test bench stimulus

## Syntax

```
data_in = generatetbstimulus(Hd,'InputDataType',nt)
data_in = generatetbstimulus(Hd)
data_in = generatetbstimulus(___,Name,Value...)
```

## Description

`data_in = generatetbstimulus(Hd,'InputDataType',nt)` generates filter input stimulus for a filter System object `Hd`. The argument `nt` is a `numerictype` object specifying the input data type.

`data_in = generatetbstimulus(Hd)` generates filter input stimuli for a `dfilt` filter.

`data_in = generatetbstimulus(___,Name,Value...)` generates filter input stimuli for the filter `Hd`, based on the specified values of the properties. The stimulus is generated based on the setting of the properties TestBenchStimulus and TestBenchUserStimulus. The following choices of stimuli are available:

- `'impulse'`
- `'step'`
- `'ramp'`
- `'chirp'`
- `'noise'`

## Examples

### Generate Test Bench Stimulus for FIR Filter

Design a lowpass filter and construct a direct-form FIR filter System object™, Hd.

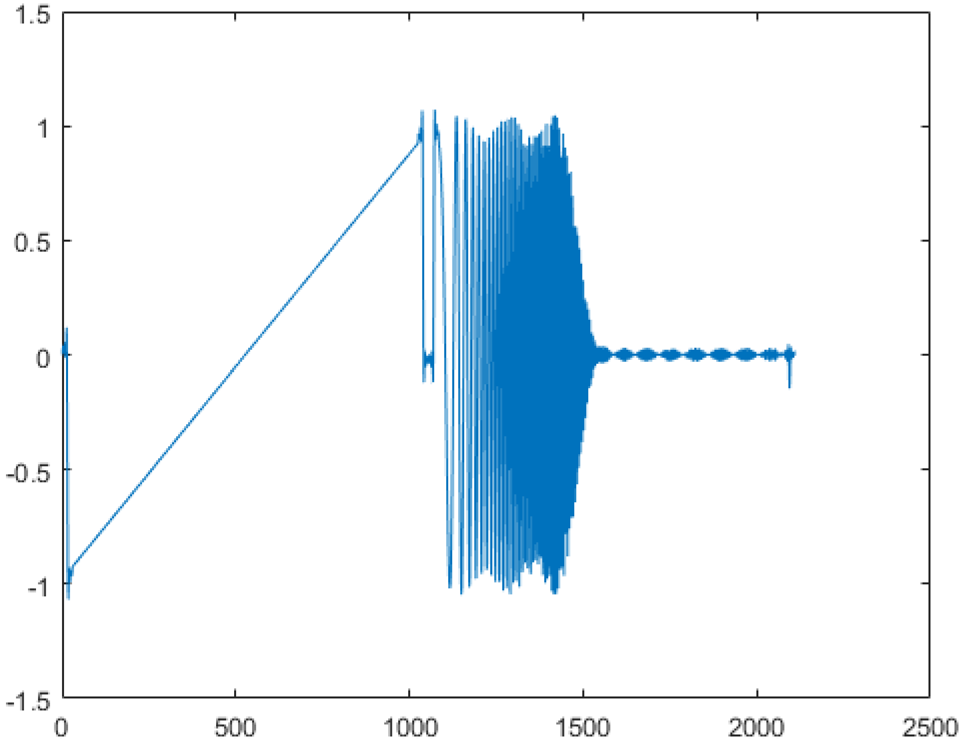
```
filtdes = fdesign.lowpass('N,Fc,Ap,Ast',30,0.4,0.05,0.03,'linear');
fir_lp = design(filtdes,'filterstructure','dffir','Systemobject',true);
```

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),'TestBenchStir
```

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'))
```



## See Also

`generatehdl`

**Introduced before R2006a**

# hdlfilterdainfo

Distributed arithmetic information for filter architectures

## Syntax

```
hdlfilterdainfo(Hd,'InputDataType',nt)
hdlfilterdainfo(Hd)
hdlfilterdainfo(___,'FoldingFactor',ff)
hdlfilterdainfo(___,'DARadix',dr)
hdlfilterdainfo(___,'LUTInputs',lutip)
hdlfilterdainfo(___,'DALUTPartition',dp)
[dp,dr,lutsize,ff] = hdlfilterdainfo(___)
```

## Description

`hdlfilterdainfo` is an informational function that helps you define optimal distributed arithmetic (DA) settings for a filter. For general information on distributed arithmetic architectures, see “Distributed Arithmetic for FIR Filters” on page 4-21.

`hdlfilterdainfo(Hd,'InputDataType',nt)` displays an exhaustive table of `DARadix` values for the filter System object, *Hd*, with the corresponding folding factor and number of LUT sets. This option also displays a table of `DALUTPartition` values with corresponding LUT inputs (maximum address width) and details of the LUT sets. The argument `nt` is a `numerictype` object specifying the input data type.

`hdlfilterdainfo(Hd)` displays an exhaustive table of `DARadix` and `DALUTPartition` values for the `dfilt` filter, *Hd*.

`hdlfilterdainfo(___,'FoldingFactor',ff)` displays a table of LUT input values, sizes, and dimensions, for the folding factor, *ff*. *ff* must be a nonzero positive integer, or `inf` (to indicate the maximum).

`hdlfilterdainfo(___,'DARadix',dr)` displays a table of LUT input values, sizes, and dimensions, for the `DARadix` value *dr*.

`hdlfilterdainfo(___,'LUTInputs',lutip)` displays a table of folding factor values for LUT inputs (maximum address width) of *lutip*. This option also displays LUT size and dimensions for each value of folding factor.

`hdlfilterdainfo(___,'DALUTPartition',dp)` displays a table of folding factor values for `DALUTPartition` of *`dp`*. This option also displays the LUT size and dimensions for each value of folding factor.

`[dp,dr,lutsize,ff] = hdlfilterdainfo(___)` returns the DA LUT partition, *`dp`*, DA radix, *`dr`*, folding factor, *`fold`*, and LUT size, *`lutsize`* to a cell array.

# Input Arguments

### `Hd` — Filter object

filter System object, or `dfilt` object

See "Distributed Arithmetic for FIR Filters" on page 4-21 for filter types that support distributed arithmetic. If `Hd` is a System object, you must specify the `InputDataType`.

## Parameter Name/Value Pairs

The following parameter name/value inputs are optional. These parameters do not have a default value; you must supply a valid value.

### `'InputDataType'` — Specify input data type for System objects

object of `numerictype` class

This argument is required when the input filter, `Hd`, is a System 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. For more information, see InputDataType.

### `'FoldingFactor'`

Hardware folding factor, an integer greater than `1` (or `inf`). If the folding factor is `inf`, the coder uses the maximum folding factor. Given the folding factor, the coder the corresponding LUT inputs.

### `'DARadix'`

Desired DA Radix value. Given the DA radix, of multipliers, the coder computes a table of values of LUT inputs and sizes.

**'LUTInputs'**

Displays an exhaustive table of values of folding factor corresponding to LUT inputs.

**'DALUTPartition'**

Displays an exhaustive table of values of folding factor for corresponding LUT partition.

## Output Arguments

### [dp,dr,lutsize,ff] — Filter architecture details

cell array

Cell array. hdlfilterdainfo returns, in order, the DA LUT partition, *dp*, DA radix, *dr*, folding factor, *fold*, and LUT size, *lutsize*.

## 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.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4),'Systemobject',true);
hdlfilterdainfo(Hd,'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             | DALUTParti
   ------------------------------------------------------------------------------------
   |         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.
```

## More About

- "Distributed Arithmetic for FIR Filters" on page 4-21

#### Introduced in R2011a

# hdlfilterserialinfo

Serial partition information for filter architectures

## Syntax

```
hdlfilterserialinfo(Hd,'InputDataType',nt)
hdlfilterserialinfo(Hd)
hdlfilterserialinfo(___,'FoldingFactor',ff)
hdlfilterserialinfo(___,'Multipliers',nmults)
hdlfilterserialinfo(___,'SerialPartition',[p1 p2 ... pN])
[sp,fold,nm] = hdlfilterserialinfo(___)
```

## Description

`hdlfilterserialinfo` is an informational function that helps you define an optimal serial partition for a filter. When you specify a serial architecture for a filter, you can define the serial partitioning in the following ways:

- Directly specify *serial partitions* as a vector of integers having *N* elements, where *N* is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition.
- Specify the desired hardware *folding factor*, *ff*, an integer greater than `1`. Given the folding factor, the coder computes the serial partition and the number of multipliers.
- Specify the desired number of multipliers, *nmults*, an integer greater than `1`. Given the number of multipliers, the coder computes the serial partition and the folding factor.

`hdlfilterserialinfo(Hd,'InputDataType',nt)` displays a table of serial partition values for the filter System object, *Hd*. The filter architecture is quantized based on the input data type, *nt*, specified as a `numerictype(`*s,w,f*`)` object. You can use a System object, along with the `InputDataType` `Name,Value` pair, with any other syntax.

`hdlfilterserialinfo(Hd)` displays a table of serial partition values, with corresponding values of folding factor and number of multipliers, for the `dfilt`, *Hd*.

`hdlfilterserialinfo(___,'FoldingFactor',ff)` displays only those serial partition values corresponding to the folding factor, *ff*.

`hdlfilterserialinfo(___,'Multipliers',nmults)` displays only those serial partition values corresponding to the number of multipliers, *`nmults`*.

`hdlfilterserialinfo(___,'SerialPartition',[p1 p2 ... pN])` displays the folding factor and number of multipliers corresponding to the serial partition vector [*p1p2...pN*].

`[sp,fold,nm] = hdlfilterserialinfo(___)` returns a cell array of serial partition values with their corresponding folding factors and numbers of multipliers. To narrow the set of returned serial partition values, you can also specify any of the properties in the previous syntaxes.

# Input Arguments

### `Hd` — Filter object

filter System object, or `dfilt` object

See "Speed vs. Area Tradeoffs" on page 4-2 for filter types that support serial architectures. If `Hd` is a System object, you must specify the `InputDataType`.

## Parameter Name/Value Pairs

The following parameter name/value inputs are optional. These parameters do not have a default value; you must supply a valid value.

### `'InputDataType'` — Specify input data type for System objects

object of `numerictype` class

This argument is required when the input filter, `Hd`, is a System 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. For more information, see InputDataType.

### `'FoldingFactor'` — Hardware folding factor

integer greater than 1, or `inf`

If the folding factor is `inf`, the coder uses the maximum folding factor. Given the folding factor, the coder computes the serial partition and the number of multipliers.

### `'Multipliers'` — Desired number of multipliers

integer greater than 1, or `inf`

If the number of multipliers is `inf`, the coder uses the maximum number of multipliers. Given the number of multipliers, the coder computes the serial partition and the folding factor.

**`'SerialPartition'` — Lengths of hardware partitions**
vector of N integers

Each element of the vector specifies the length of the corresponding partition. The vector length, `N`, is the number of serial partitions.

## Output Arguments

**`[sp,fold,nm]` — Filter architecture details**
cell array

`hdlfilterserialinfo` returns, in order, the serial partition, *`sp`*, folding factor, *`fold`*, and number of multipliers, *`nm`*.

## Examples

### Explore Serial Partition Options

To display valid serial partitions, pass the filter, with no other arguments, to `hdlfilterserialinfo`.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4),'Systemobject',true);
hdlfilterserialinfo(Hd,'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.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4),'Systemobject',true);
hdlfilterserialinfo(Hd,'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.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4),'Systemobject',true);
hdlfilterserialinfo(Hd,'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.

```
Hd = design(fdesign.lowpass('N,Fc',8,.4),'Systemobject',true);
[sp,ff,nm] = hdlfilterserialinfo(Hd,'InputDataType',numerictype(1,12,10))
```

```
sp =

    '[1 1 1 1 1 1 1 1 1]'
    '[2 2 2 2 1]'
    '[3 3 3]'
    '[4 4 1]'
    '[5 4]'
    '[6 3]'
    '[7 2]'
    '[8 1]'
    '[9]'


ff =

    '1'
    '2'
    '3'
    '4'
    '5'
    '6'
    '7'
    '8'
    '9'


nm =

    '1'
    '2'
    '3'
    '5'
    '9'
```

You can also use this syntax while specifying a number of multipliers or folding factor.

```
[sp_ff4,ff4,nm_ff4] = hdlfilterserialinfo(Hd,'InputDataType',numerictype(1,12,10),...
                                 'FoldingFactor',4)


sp_ff4 =

     4     4     1


ff4 =

     4


nm_ff4 =
     3
```

## More About

- “Speed vs. Area Tradeoffs” on page 4-2

**Introduced in R2010b**