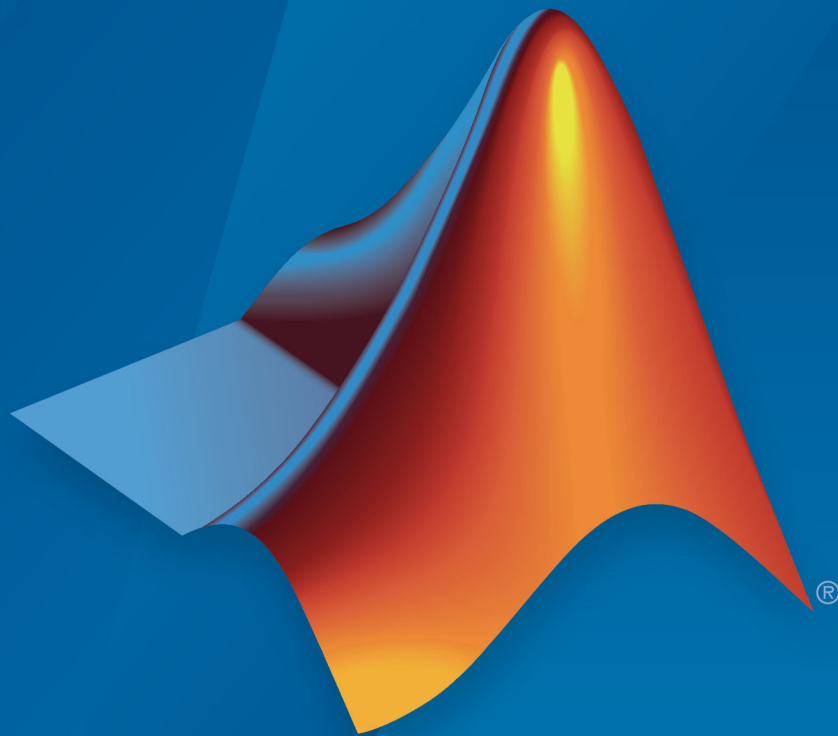


**HDL Coder™**

Reference



**MATLAB® & SIMULINK®**

R2019a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*HDL Coder™ Reference*

© COPYRIGHT 2013–2019 by The MathWorks, Inc.

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

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

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

March 2013	Online only	New for Version 3.2 (R2013a)
September 2013	Online only	Revised for Version 3.3 (R2013b)
March 2014	Online only	Revised for Version 3.4 (Release 2014a)
October 2014	Online only	Revised for Version 3.5 (Release 2014b)
March 2015	Online only	Revised for Version 3.6 (Release 2015a)
September 2015	Online only	Revised for Version 3.7 (Release 2015b)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (Release 2016a)
September 2016	Online only	Revised for Version 3.9 (Release 2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)
September 2017	Online only	Revised for Version 3.11 (Release 2017b)
March 2018	Online only	Revised for Version 3.12 (Release 2018a)
September 2018	Online only	Revised for Version 3.13 (Release 2018b)
March 2019	Online only	Revised for Version 3.14 (Release 2019a)





<b>1</b>	<b>Apps – Alphabetical List</b>
<b>2</b>	<b>Functions – Alphabetical List</b>
<b>3</b>	<b>Supported Blocks</b>
<b>4</b>	<b>Properties – Alphabetical List</b>
<b>5</b>	<b>Class reference for HDL code generation from Simulink</b>
<b>6</b>	<b>Function Reference for HDL Code Generation from MATLAB</b>
<b>7</b>	<b>Class Reference for HDL Code Generation from MATLAB</b>

# Shared Class and Function Reference for HDL Code Generation from MATLAB and Simulink

---

**8**





# Apps — Alphabetical List

---

## HDL Coder

Generate HDL code from MATLAB code

### Description

The **HDL Coder** app generates synthesizable HDL code from MATLAB® code that is supported for hardware. You can generate VHDL or Verilog HDL code that you can integrate into existing HDL applications outside of MATLAB.

The workflow-based user interface steps you through the code generation process. Using the app, you can:

- Create a project or open an existing HDL Coder project.
- Specify the MATLAB function and the MATLAB testbench for your project.
- Propose input data types or autodefine data types by specifying the MATLAB testbench file.
- Convert floating-point MATLAB code to fixed-point HDL code.
- Specify the target device and synthesis tool to deploy the generated HDL code on the target hardware.
- Access generated files and view code generation reports.
- Verify the numerical behavior of generated HDL code with HDL test bench, cosimulation, or FPGA-in-the loop.
- Synthesize, and place and route the generated HDL code for the specified hardware with the **Generic ASIC/FPGA** workflow.
- Integrate your generated HDL IP core with the embedded processor by using **IP Core Generation** workflow.
- Generate a programming file and download it to the target device with the **FPGA Turnkey** workflow.

### Open the HDL Coder App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the HDL Coder app icon.

- MATLAB command prompt: Enter `hdlcoder`.

## Examples

- “HDL Code Generation and FPGA Synthesis from a MATLAB Algorithm”

## Programmatic Use

`hdlcoder` opens the **HDL Coder** app.

## See Also

### Apps

**Fixed-Point Converter**

### Functions

`codegen`

### Topics

“HDL Code Generation and FPGA Synthesis from a MATLAB Algorithm”

“Guidelines for Efficient HDL Code”

“Create and Set Up Your Project”

**Introduced in R2012a**



# Functions — Alphabetical List

---

# checkhdl

Check subsystem or model for HDL code generation compatibility

## Syntax

```
checkhdl (bdroot)
checkhdl ('dut')
checkhdl (gcb)
output = checkhdl ('system')
```

## Description

checkhdl generates an HDL Code Generation Check Report, saves the report to the target folder, and displays the report in a new window. Before generating HDL code, use checkhdl to check your subsystems or models.

---

**Note** Running this command can activate the **Open at simulation start** setting for blocks such as the Scope block and therefore invoke the block.

---

The report lists compatibility errors with a link to each block or subsystem that caused a problem. To highlight and display incompatible blocks, click each link in the report while keeping the model open.

The report file name is `system_report.html`. *system* is the name of the subsystem or model passed in to checkhdl.

When a model or subsystem passes checkhdl, that does not imply code generation will complete. checkhdl does not verify all block parameters.

checkhdl (bdroot) examines the current model for HDL code generation compatibility.

checkhdl ('dut') examines the specified DUT model name, model reference name, or subsystem name with full hierarchical path.

checkhdl (gcb) examines the currently selected subsystem.

```
output = checkhdl('system')
```

does not generate a report. Instead, it returns a 1xN struct array with one entry for each error, warning, or message. *system* specifies a model or the full block path for a subsystem at any level of the model hierarchy.

checkhdl reports three levels of compatibility problems:

- *Errors*: cause the code generation process to terminate. The report must not contain errors to continue with HDL code generation.
- *Warnings*: indicate problems in the generated code, but allow HDL code generation to continue.
- *Messages*: indication that some data types have special treatment. For example, the HDL Coder software automatically converts single-precision floating-point data types to double-precision because VHDL® and Verilog® do not support single-precision data types.

## Examples

Check the subsystem `symmetric_fir` within the model `sfir_fixed` for HDL code generation compatibility and generate a compatibility report.

```
checkhdl('sfir_fixed/symmetric_fir')
```

Check the subsystem `symmetric_fir_err` within the model `sfir_fixed_err` for HDL code generation compatibility, and return information on problems encountered in the struct `output`.

```
output = checkhdl('sfir_fixed_err/symmetric_fir_err')
### Starting HDL Check.
...
### HDL Check Complete with 4 errors, warnings and messages.
```

The following MATLAB commands display the top-level structure of the struct `output`, and its first cell.

```
output =
1x4 struct array with fields:
    path
    type
    message
    level
```

```
output(1)
ans =
    path: 'sfir_fixed_err/symmetric_fir_err/Product'
    type: 'block'
    message: 'Unhandled mixed double and non-double datatypes at ports of block'
    level: 'Error'
```

## See Also

makehdl

## Topics

“Create Simulink Model for HDL Code Generation”

“Check Your Model for HDL Compatibility”

**Introduced in R2006b**



# hdladvisor

Display HDL Workflow Advisor

## Syntax

```
hdladvisor(gcb)  
hdladvisor(subsystem)  
hdladvisor(model, 'SystemSelector')
```

## Description

`hdladvisor(gcb)` starts the HDL Workflow Advisor, passing the currently selected subsystem within the current model as the DUT to be checked.

`hdladvisor(subsystem)` starts the HDL Workflow Advisor, passing in the path to a specified subsystem within the model.

`hdladvisor(model, 'SystemSelector')` opens a System Selector window that lets you select a subsystem to be opened into the HDL Workflow Advisor as the device under test (DUT) to be checked.

## Examples

Open the subsystem `symmetric_fir` within the model `sfir_fixed` into the HDL Workflow Advisor.

```
hdladvisor('sfir_fixed/symmetric_fir')
```

Open a System Selector window to select a subsystem within the current model. Then open the selected subsystem into the HDL Workflow Advisor.

```
hdladvisor(gcs, 'SystemSelector')
```

## **Alternatives**

You can also open the HDL Workflow Advisor from the your model window by selecting **Code > HDL Code > HDL Workflow Advisor**.

## **See Also**

“HDL Workflow Advisor Tasks” | “Getting Started with the HDL Workflow Advisor”

**Introduced in R2010a**

# hdlcoder.optimizeDesign

Automatic iterative HDL design optimization

## Syntax

```
hdlcoder.optimizeDesign(model, optimizationCfg)
hdlcoder.optimizeDesign(model, cpGuidanceFile)
```

## Description

`hdlcoder.optimizeDesign(model, optimizationCfg)` automatically optimizes your generated HDL code based on the optimization configuration you specify.

`hdlcoder.optimizeDesign(model, cpGuidanceFile)` regenerates the optimized HDL code without rerunning the iterative optimization, by using data from a previous run of `hdlcoder.optimizeDesign`.

## Examples

### Maximize clock frequency

Maximize the clock frequency for a model, `sfir_fixed`, by performing up to 10 optimization iterations.

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);
```

Set your synthesis tool and target device options.

```
hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
                  'SynthesisToolChipFamily', 'Zynq', ...
```

```
'SynthesisToolDeviceName', 'xc7z030', ...  
'SynthesisToolPackageName', 'fbg484', ...  
'SynthesisToolSpeedValue', '-3')
```

Enable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'on');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Set the iteration limit to 10.

```
oc.IterationLimit = 10;
```

Optimize the model.

```
hdlcoder.optimizeDesign(model, oc)
```

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');  
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');  
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');  
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');  
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');  
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
```

```
Iteration 0
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 16.26 (Constraint ns) 5.85 (Elapsed s) 143.66 Iteration 1
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 16.26 (Constraint ns) 5.85 (Elapsed s) 278.72 Iteration 2
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 10.25 (Constraint ns) 12.73 (Elapsed s) 427.22 Iteration 3
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 9.55 (Constraint ns) 9.73 (Elapsed s) 584.37 Iteration 4
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 9.55 (Constraint ns) 9.38 (Elapsed s) 741.04 Iteration 5
```

```

Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 9.55 ns           Elapsed : 741.04 s
Iteration 0: (CP ns) 16.26   (Constraint ns) 5.85   (Elapsed s) 143.66
Iteration 1: (CP ns) 16.26   (Constraint ns) 5.85   (Elapsed s) 278.72
Iteration 2: (CP ns) 10.25   (Constraint ns) 12.73   (Elapsed s) 427.22
Iteration 3: (CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-04-41
Validation model: gm_sfir_fixed_vnl

```

Then HDL Coder stops after five iterations because the fourth and fifth iterations had the same critical path, which indicates that the coder has found the minimum critical path. The design's maximum clock frequency after optimization is 1 / 9.55 ns, or 104.71 MHz.

### Optimize for specific clock frequency

Optimize a model, `sfir_fixed`, to a specific clock frequency, 50 MHz, by performing up to 10 optimization iterations, and do not generate an HDL test bench.

Open the model and specify the DUT subsystem.

```

model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);

```

Set your synthesis tool and target device options.

```

hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
    'SynthesisToolChipFamily', 'Zynq', ...
    'SynthesisToolDeviceName', 'xc7z030', ...
    'SynthesisToolPackageName', 'fbg484', ...
    'SynthesisToolSpeedValue', '-3')

```

Disable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'off');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to stop after it reaches a clock frequency of 50MHz, or 10 iterations, whichever comes first.

```
oc.ExplorationMode = ...  
    hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency;  
oc.TargetFrequency = 50;  
oc.IterationLimit = 10; =
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed','GenerateHDLTestBench','off');  
hdlset_param('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir');  
hdlset_param('sfir_fixed','SynthesisTool','Xilinx ISE');  
hdlset_param('sfir_fixed','SynthesisToolChipFamily','Zynq');  
hdlset_param('sfir_fixed','SynthesisToolDeviceName','xc7z030');  
hdlset_param('sfir_fixed','SynthesisToolPackageName','fbg484');  
hdlset_param('sfir_fixed','SynthesisToolSpeedValue','-3');
```

```
Iteration 0
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 16.26      (Constraint ns) 20.00      (Elapsed s) 134.02 Iteration 1
```

```
Generate and synthesize HDL code ...
```

```
Exiting because constraint (20.00 ns) has been met (16.26 ns).
```

```
Summary report: summary.html
```

```
Achieved Critical Path (CP) Latency : 16.26 ns      Elapsed : 134.02 s
```

```
Iteration 0: (CP ns) 16.26      (Constraint ns) 20.00      (Elapsed s) 134.02
```

```
Final results are saved in
```

```
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-14
```

```
Validation model: gm_sfir_fixed_vnl
```

Then HDL Coder stops after one iteration because it has achieved the target clock frequency. The critical path is 16.26 ns, a clock frequency of 61.50 GHz.

## Resume clock frequency optimization using saved data

Run additional optimization iterations for a model, `sfir_fixed`, using saved iteration data, because you terminated in the middle of a previous run.

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);
```

Set your synthesis tool and target device options to the same values as in the interrupted run.

```
hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
                'SynthesisToolChipFamily', 'Zynq', ...
                'SynthesisToolDeviceName', 'xc7z030', ...
                'SynthesisToolPackageName', 'fbg484', ...
                'SynthesisToolSpeedValue', '-3')
```

Enable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'on');
```

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to run using data from the first iteration of a previous run.

```
oc.ResumptionPoint = 'Iter5-07-Jan-2014-17-04-29';
```

Optimize the model.

```
hdlcoder.optimizeDesign(model, oc)

hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
```

```
Try to resume from resumption point: Iter5-07-Jan-2014-17-04-29
Iteration 5
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 9.55 ns           Elapsed : 741.04 s
Iteration 0: (CP ns) 16.26   (Constraint ns) 5.85   (Elapsed s) 143.66
Iteration 1: (CP ns) 16.26   (Constraint ns) 5.85   (Elapsed s) 278.72
Iteration 2: (CP ns) 10.25   (Constraint ns) 12.73   (Elapsed s) 427.22
Iteration 3: (CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-30
Validation model: gm_sfir_fixed_vnl
```

Then coder stops after one additional iteration because it has achieved the target clock frequency. The critical path is 9.55 ns, or a clock frequency of 104.71 MHz.

### Regenerate code using original design and saved optimization data

Regenerate HDL code using the original model, `sfir_fixed`, and saved data from the final iteration of a previous optimization run.

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);
```

Set your synthesis tool and target device options to the same values as in the original run.

```
hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
    'SynthesisToolChipFamily', 'Zynq', ...
    'SynthesisToolDeviceName', 'xc7z030', ...
    'SynthesisToolPackageName', 'fbg484', ...
    'SynthesisToolSpeedValue', '-3')
```

Regenerate HDL code using saved optimization data from `cpGuidance.mat`.

```
hdlcoder.optimizeDesign(model,
    'hdlsrc/sfir_fixed/hdlexpl/Final-19-Dec-2013-23-05-04/cpGuidance.mat')
```



Final results are saved in  
/tmp/hdlsrc/sfir\_fixed/hdlexpl/Final-07-Jan-2014-17-16-52  
Validation model: gm\_sfir\_fixed\_vnl

## Input Arguments

### **model** — Model name

character vector

Model name, specified as a character vector.

Example: 'sfir\_fixed'

### **optimizationCfg** — Optimization configuration

hdlcoder.OptimizationConfig

Optimization configuration, specified as an hdlcoder.OptimizationConfig object.

### **cpGuidanceFile** — File containing saved optimization data

' ' (default) | character vector

File that contains saved data from the final optimization iteration, including relative path, specified as a character vector. Use this file to regenerate optimized code without rerunning the iterative optimization.

The file name is cpGuidance.mat. You can find the file in the iteration folder name that starts with Final, which is a subfolder of hdlexpl.

Example: 'hdlexpl/Final-11-Dec-2013-23-17-10/cpGuidance.mat'

## See Also

### **Classes**

hdlcoder.OptimizationConfig

### **Functions**

hdlcoder.supportedDevices

## Topics

“Automatic Iterative Optimization”

“Tool and Device”

**Introduced in R2014a**

# importhdl

Import Verilog code and generate Simulink model

`importhdl` imports and parses the specified Verilog files to generate the corresponding Simulink® model.

## Syntax

```
importhdl(FileNames)
importhdl(FileNames,Name,Value)
```

## Description

`importhdl(FileNames)` imports the specified Verilog files and generates the corresponding Simulink model.

`importhdl(FileNames,Name,Value)` imports the specified Verilog files and generates the corresponding Simulink model with options specified by one or more name-value pair arguments.

## Examples

### Generate Simulink Model From Single Verilog File

This example shows how you can import a file containing Verilog code and generate the corresponding Simulink™ model.

### Specify Input Verilog File

Make sure that the input HDL file does not contain any syntax errors, is synthesizable, and uses constructs that are supported by HDL import. This example shows a Verilog code of a comparator.

```
edit('comparator.v')
```

```
// File Name: comparator.v
// This module implements a simple comparator module

`define value 12
module comparator (clk, rst, a, b);

input clk, rst;
input [1:0] a;
output reg [1:0] b;

parameter d = 2'b11;

always@(posedge clk) begin
    if (rst)
        b <= 0;
    else if (a < `value)
        b <= a + 1;
end

endmodule
```

### Import Verilog File

To import the HDL file and generate the Simulink™ model, pass the file name as a character vector to the `importhdl` function.

```
importhdl('comparator.v')

### Parsing <a href="matlab:edit('comparator.v')">comparator.v</a>.
### Top Module of the source: 'comparator'.
### Identified ClkName::clk.
### Identified RstName::rst.
### Hdl Import parsing done.
### Creating Target model comparator
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'comparator'.
### Laying out components.
### Working on hierarchy at ---> 'comparator/comparator'.
```

```

### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting the model parameters.
### Generated model as C:\Temp\examples\examples\hdlcoder-ex77699673\hdlimport\comparat
### HDL Import completed.

```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `comparator.slx`. The generated model uses the same name as the top module in the input Verilog file.

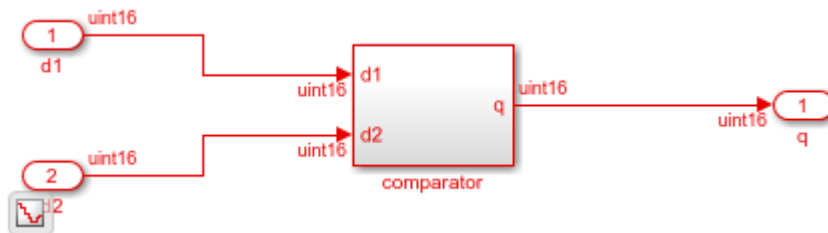
### Examine Generated Simulink™ Model

To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/comparator` path relative to the current folder. You can simulate the model and observe the simulation results.

```

addpath('hdlimport/comparator')
open_system('comparator.slx')
sim('comparator.slx')

```



### Generate Simulink Model From Multiple Verilog Files

This example shows how you can import multiple files containing Verilog code and generate the corresponding Simulink™ model.

#### Specify input Verilog File

Make sure that the input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three

Verilog files that use module instantiation to form a hierarchical design. One module `example1.v` implements a simple sequential circuit based on an if-else condition. The other module `example2.v` implements a simple combinational arithmetic expression.

```
edit('example1.v')  
edit('example2.v')
```

```
// File Name: example1.v
// This module implements a sequential circuit that
// adds two inputs or multiplies one of the inputs by a factor
// based on a conditional.

module example1(clk, cond, y, a, b);

input clk, cond;
input [7:0] a, b;
output reg [7:0] y;

parameter g = 8'd5;

always@(posedge clk)
    if (cond == 1'b1) y <= a + b;
    else y <= a * g;

endmodule

// File Name: example2.v
// This module implements a combinational arithmetic expression
module example2(c, d, e, f, y2);

input [7:0] c, d, e, f;
output [7:0] y2;

assign y2 = (c + d) * e / f;

endmodule
```

A top module contained in file `example.v` instantiates the two modules in `example1.v` and `example2.v`

```
edit('example.v')

// File Name: example.v
// This is the top-level module
module example(clk, cond, a, b, c, d, e, f, y, y2);

    input clk;
    input cond;
    input [7:0] a, b, c, d, e, f;
    output [7:0] y;
    output [7:0] y2;

    example1 example1(.clk(clk), .a(a), .cond(cond), .b(b), .y(y));

    example2 example2(.c(c), .d(d), .e(e), .f(f), .y2(y2));

endmodule
```

### Import Verilog Files

To import the HDL file and generate the Simulink™ model, pass the file names as a cell array of character vectors to the `importhdl` function. By default, HDL import identifies the top module and clock bundle when parsing the input file.

```
importhdl({'example.v','example1.v','example2.v'})

### Parsing <a href="matlab:edit('example.v')">example.v</a>.
### Parsing <a href="matlab:edit('example1.v')">example1.v</a>.
### Parsing <a href="matlab:edit('example2.v')">example2.v</a>.
### Top Module of the source: 'example'.
### Identified ClkName::clk.
### Hdl Import parsing done.
### Creating Target model example
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'example'.
### Laying out components.
### Working on hierarchy at ---> 'example/example'.
```



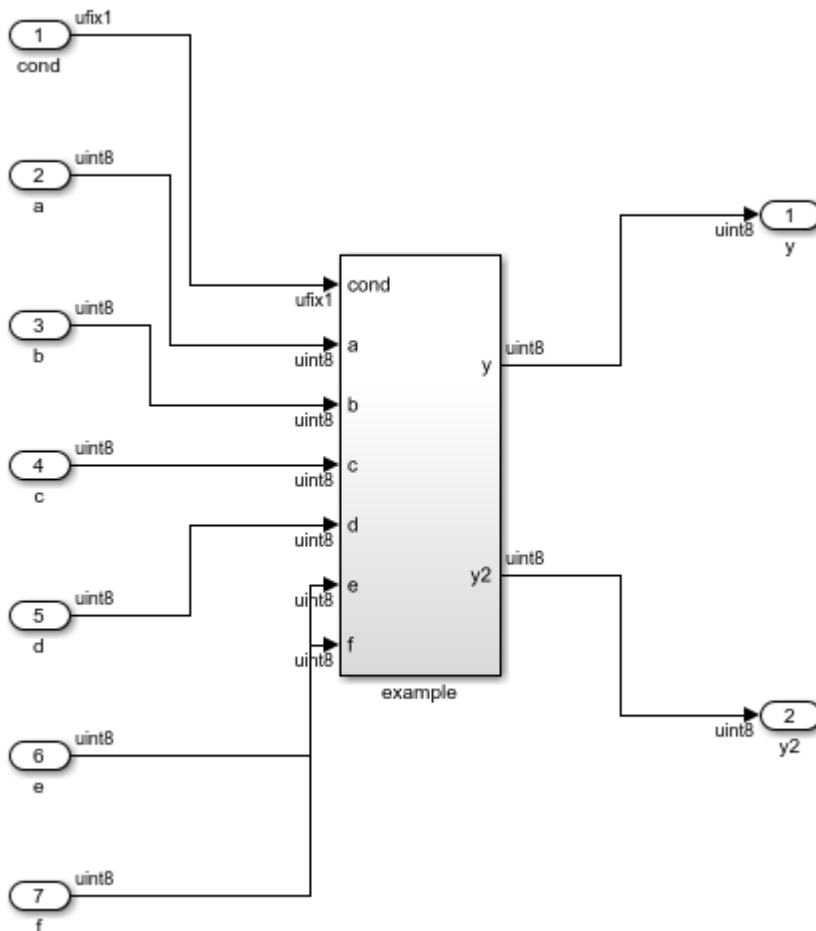
```
### Laying out components.  
### Working on hierarchy at ---> 'example/example/example1'.  
### Laying out components.  
### Applying Dot Layout...  
### Drawing block edges...  
### Working on hierarchy at ---> 'example/example/example2'.  
### Laying out components.  
### Applying Dot Layout...  
### Drawing block edges...  
### Applying Dot Layout...  
### Drawing block edges...  
### Applying Dot Layout...  
### Drawing block edges...  
### Setting the model parameters.  
### Generated model as C:\Temp\examples\examples\hdlcoder-ex56732899\hdlimport\example1.v  
### HDL Import completed.
```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `example.slx`. The generated model uses the same name as the top module that is contained in the input Verilog file `example1.v`.

### **Examine Generated Simulink™ Model**

To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/example` path relative to the current folder. You can simulate the model and observe the simulation results.

```
addpath('hdlimport/example')  
open_system('example.slx')
```

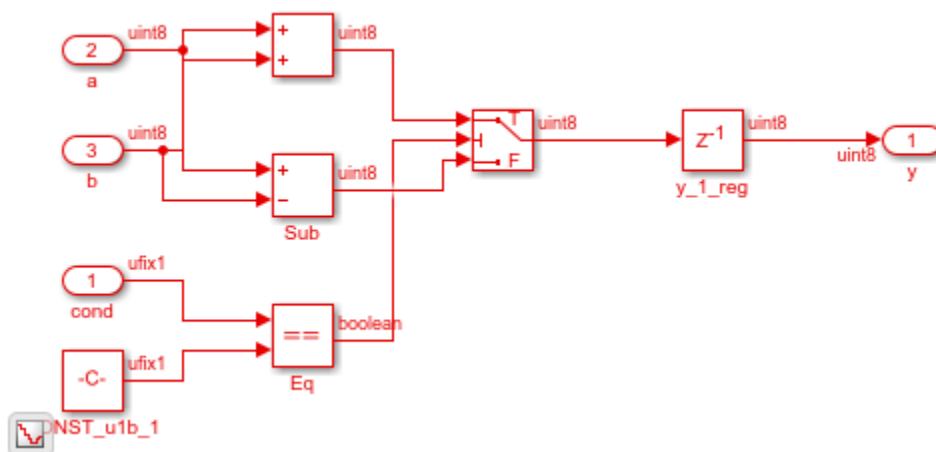


To avoid a division by zero, you can suppress the warning diagnostic before simulation.

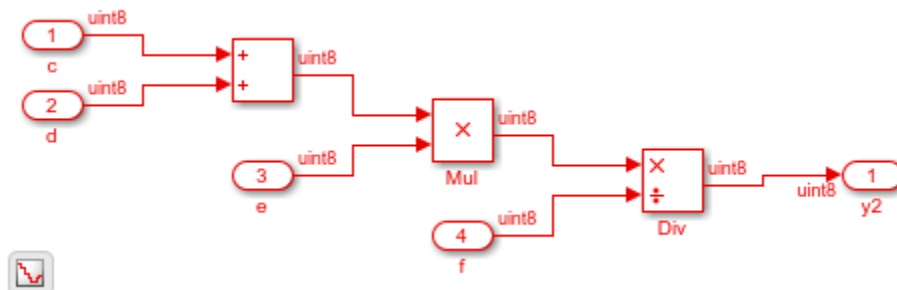
```
Simulink.suppressDiagnostic({'example/example/example2/Div'}, ...
                           'SimulinkFixedPoint:util:fxpDivisionByZero')
sim('example')
```

You can see the hierarchy of Subsystems that implement the Verilog code that uses module instantiation.

```
open_system('example/example/example1')
```



```
open_system('example/example/example2')
```



## Generate Simulink Model From Verilog Files with BlackBox Modules

This example shows how you can import multiple files containing Verilog code and generate the corresponding Simulink™ model. When you import multiple files, if you want to obfuscate the HDL code or if your files contain HDL code for vendor-specific IPs, you can import the HDL code as a BlackBox module using the `importhdl` function.

### Specify input Verilog Files

Make sure that the input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three

Verilog files that use module instantiation to form a hierarchical design. One module `sequentialexp.v` implements a simple sequential circuit based on an if-else condition. The other module `conditionalcomb.v` implements a simple combinational arithmetic expression.

```
edit('conditionalcomb.v')  
edit('sequentialexp.v')  
edit('intelip.v')
```

```
// File Name: conditionalcomb.v
// This module implements a sequential circuit that
// adds or subtracts two inputs based on a conditional.

module conditionalcomb(clk, cond, y, a, b);

input clk, cond;
input [7:0] a, b;
output reg [7:0] y;

always@(posedge clk)
    if (cond == 1'b1) y <= a + b;
    else y <= a - b;

endmodule

// File Name: sequentialexp.v
// This module implements a combinational arithmetic expression

module sequentialexp(a, b, c, d, e, f, y1, y2);

input a, b;
input [7:0] c, d, e, f;
output [7:0] y1;
output y2;

assign y1 = (c + d) * e / f;

//Instantiate Intel Vendor IP
intelip u_intelip(.dataa(a), .datab(b), .datac(y2));

endmodule
```

See that the `sequentialexp.v` module instantiates an Intel® IP that implements a single-precision floating-point adder.

```
// This module is the Intel IP that implements a
// single-precision floating-point adder.
module intelip(dataa, datab, datac);

input dataa, datab;
output datac;

assign datac = dataa + datab;

endmodule
```

A top module `top` contained in file `blackboxtop.v` instantiates the two modules in `conditionalcomb.v` and `sequentialexp.v`

```
edit('blackboxtop.v')
```

```

// File Name: blackboxtop.v
// This is the top-level module that instantiates
// modules example1 and example2.
module top(clk, cond, a, b, c, d, e, f, g, h, y, y1, y2);

input clk, cond, g, h;
input [7:0] a, b, c, d, e, f;
output [7:0] y, y1;
output y2;

conditionalcomb u_comb(.clk(clk),.a(a),.cond(cond),.b(b),.y(y));

sequentialexp u_seq(.a(g),.b(h),.c(c),.d(d),.e(e),.f(f),.y1(y1),.y2(y2));

endmodule

```

## Import Verilog Files

To import the HDL file and generate the Simulink™ model, pass the file names as a cell array of character vectors to the `importhdl` function. By default, HDL import identifies the top module and clock bundle when parsing the input file.

```

importhdl({'blackboxtop.v','conditionalcomb.v','sequentialexp.v','intelip.v'}, ...
          'topModule','top','blackBoxModule','intelip')

### Parsing <a href="matlab:edit('blackboxtop.v')">blackboxtop.v</a>.
### Parsing <a href="matlab:edit('conditionalcomb.v')">conditionalcomb.v</a>.
### Parsing <a href="matlab:edit('sequentialexp.v')">sequentialexp.v</a>.
### Parsing <a href="matlab:edit('intelip.v')">intelip.v</a>.
### Top Module of the source: 'top'.
### Identified ClkName::clk.
### Hdl Import parsing done.
### Creating Target model top
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'top'.
### Laying out components.
### Working on hierarchy at ---> 'top/top'.

```

```
### Laying out components.
### Working on hierarchy at ---> 'top/top/u_comb'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Working on hierarchy at ---> 'top/top/u_seq'.
### Laying out components.
### Working on hierarchy at ---> 'top/top/u_seq/u_intelip'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting the model parameters.
### Generated model as C:\Temp\examples\examples\hdlcoder-ex63017378\hdlimport\top\top
### HDL Import completed.
```

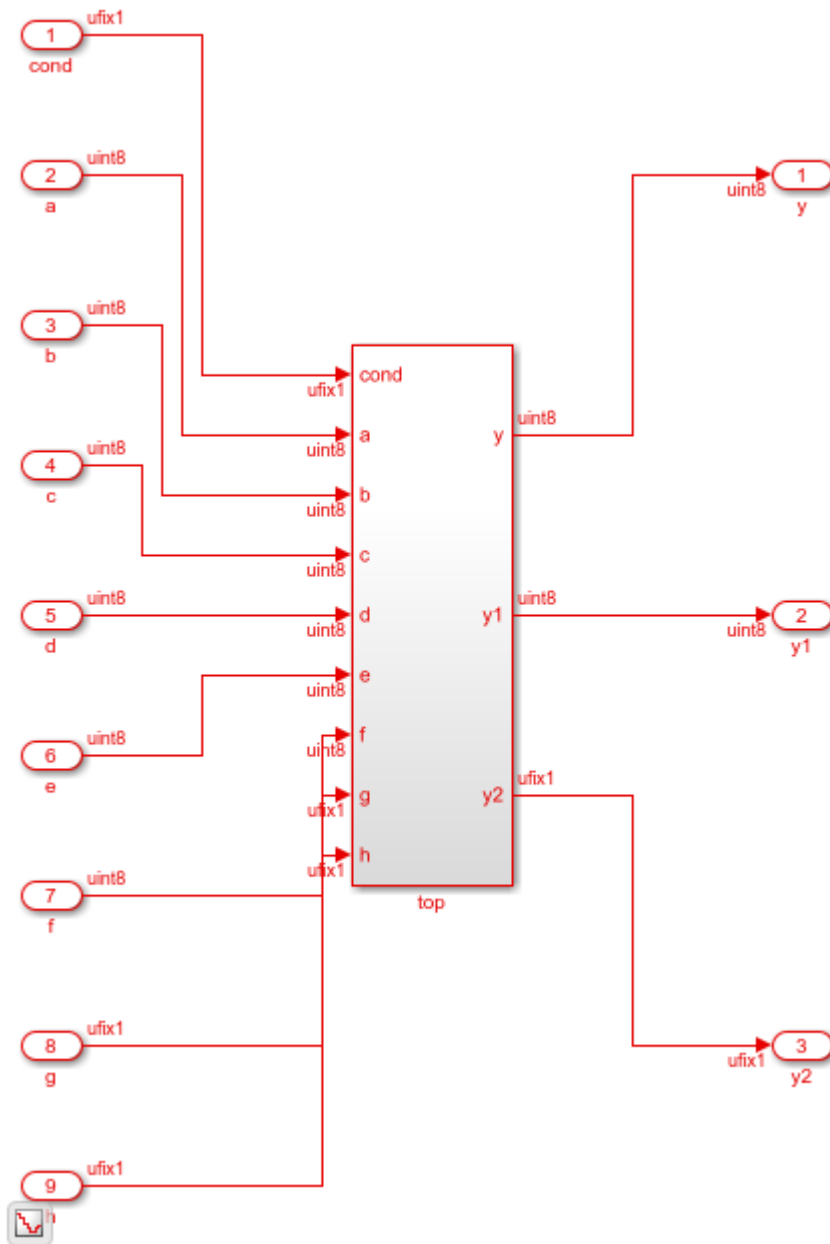
HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `top.slx`. The generated model uses the same name as the top module that is contained in the input Verilog file `conditionalcomb.v`.

### Examine Generated Simulink™ Model

To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/top` path relative to the current folder. You can simulate the model and observe the simulation results.

```
addpath('hdlimport/top')
open_system('top.slx')
set_param('top', 'SimulationCommand', 'update')
```



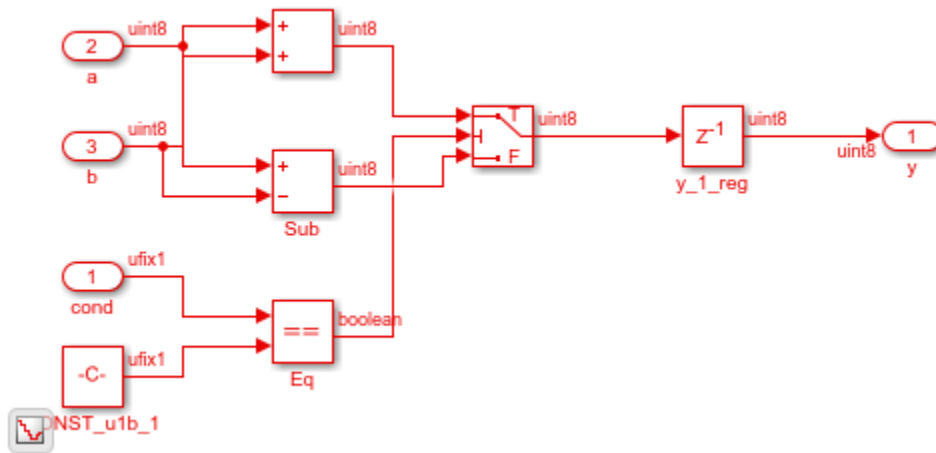


To avoid a division by zero, you can suppress the warning diagnostic before simulation.

```
Simulink.suppressDiagnostic({'top/top/u_seq/Div'}, ...  
                           'SimulinkFixedPoint:util:fxpDivisionByZero')  
sim('top')
```

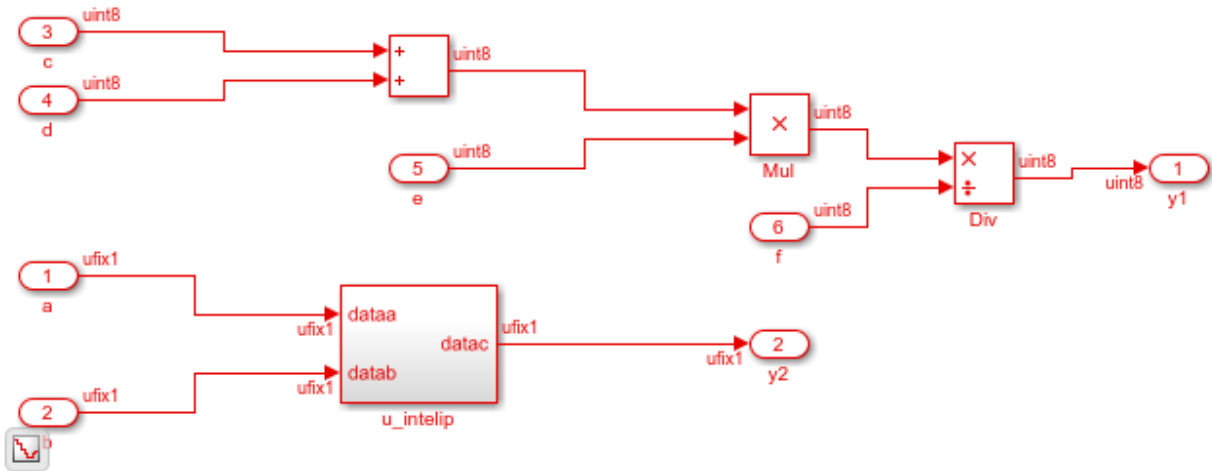
You can see the hierarchy of Subsystems that implement the Verilog code that uses module instantiation.

```
open_system('top/top/u_comb')
```

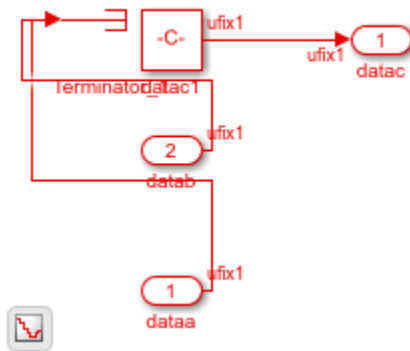


If you open the Subsystem that implements the sequential circuit, you can open the `u_intelip` Subsystem to see the blackbox implementation.

```
open_system('top/top/u_seq')
```



```
open_system('top/top/u_seq/u_intelip')
```



**Generate Simulink Model from Verilog Code for Various Operators**

This example shows how you can import Verilog code that contains these operators and generate the corresponding Simulink™ model:

- Arithmetic

- Logical
- XOR
- Bitwise
- Conditional
- Relational
- Concatenation

### **Specify Input Verilog File**

Make sure that the input HDL file does not contain any syntax errors, is synthesizable, and uses constructs for the various operators. For example, this Verilog code shows various operators.

```
edit('VerilogOperators.v')
```

```

`timescale 1 ns / 1 ns

module VerilogOperators (A, B, C, D, Y1, Y2, Y3, Y4, Y5, Y6);

    input    [7:0] A, B;
    input    C, D;
    output reg [7:0] Y1, Y2;
    output reg Y3, Y4;
    output reg [15:0] Y5;
    output [7:0] Y6;

    always @(A or B or C or D) begin

        Y1 = A % B;           // Arithmetic remainder operator
        Y2 = B >> 4;         // Logical shift right operator
        Y3 = C ^ D;          // Reduction XOR operator
        Y4 = A <= B;         // Relational operator
        Y5 = {A, D, {2{3'b011}}, 1'b0}; // Concatenation operator

    end

    assign Y6[7:0] = (C == 1'b1) ? A : B; // Conditional operator

endmodule

```

### Import Verilog File

To import the HDL file and generate the Simulink™ model, pass the file name as a character vector to the `importhdl` function.

```

importhdl('VerilogOperators.v')

### Parsing <a href="matlab:edit('VerilogOperators.v')">VerilogOperators.v</a>.
### Top Module of the source: 'VerilogOperators'.
### HdL Import parsing done.
### Creating Target model VerilogOperators
### Generating Dot Layout...

```

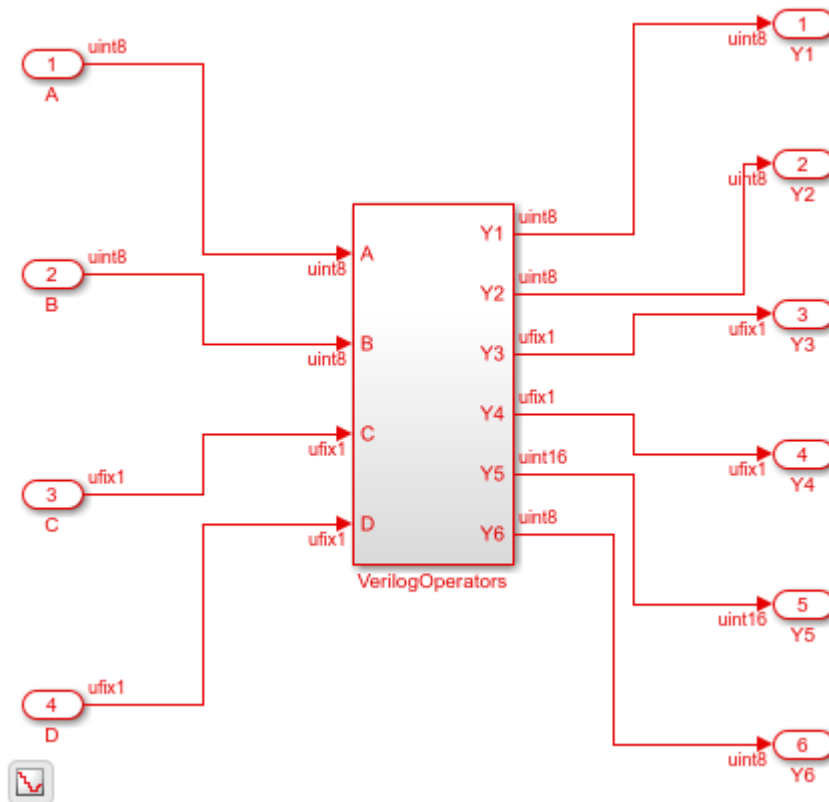
```
### Start Layout...
### Working on hierarchy at ---> 'VerilogOperators'.
### Laying out components.
### Working on hierarchy at ---> 'VerilogOperators/VerilogOperators'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Generated model as C:\Temp\examples\examples\hdlcoder-ex29847655\hdlimport\Verilog
### HDL Import completed.
```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `VerilogOperators.slx`. The generated model uses the same name as the top module in the input Verilog file.

### **Examine Generated Simulink™ Model**

To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/VerilogOperators` path relative to the current folder. You can simulate the model and observe the simulation results.

```
addpath('hdlimport/VerilogOperators')
open_system('VerilogOperators.slx')
sim('VerilogOperators.slx')
```



## Implicit Data Type Conversion when Importing Verilog Code

This example shows how you can import multiple files containing Verilog code that perform implicit data type conversions and generate the corresponding Simulink™ model. HDL import can perform implicit data type conversion such as in arithmetic operations, data type conversion, bit selection, and bit concatenation.

### Specify input Verilog File

Make sure that input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three

Verilog files that use module instantiation to form a hierarchical design. The modules `NG1_implicit.v` and `round_const.v` perform implicit data type conversion.

```
edit('NG1_implicit.v')
edit('round_constant.v')
```

```
module NG1(IN_A, IN_B, OUT_A);

    input [4:0] IN_A;
    input IN_B;
    output OUT_A;

    parameter IN_B_AND = 5'b10011;

    assign OUT_A = IN_A == IN_B_AND & IN_B;

endmodule
```

```
/* round constant */
module rconst(i,rc);

    input [23:0] i;
    output reg [63:0] rc;

    always@(i) begin
        rc = 0;
    end

endmodule
```

A top module contained in file `example.v` instantiates the two modules in `NG1_implicit.v` and `round_constant.v`.

```
edit('implicit_top.v')
```



```

// File Name: implicit_top.v
// This is the top-level module that instantiates
// modules NG1_implicit and round_constant.
module top(A, B, C, Y1, Y2);

input [4:0] A;
input B;
input [23:0] C;
output Y1, Y2;

NG1 NG1(A, B, Y1);

rconst rconst(C, Y2);

endmodule

```

## Import Verilog Files

To import the HDL file and generate the Simulink™ model, pass the file names as a cell array of character vectors to the `importhdl` function. By default, HDL import identifies the top module when parsing the input file.

```
importhdl({'implicit_top.v', 'NG1_implicit.v', 'round_constant.v'})
```

```

### Parsing <a href="matlab:edit('implicit_top.v')">implicit_top.v</a>.
### Parsing <a href="matlab:edit('NG1_implicit.v')">NG1_implicit.v</a>.
### Parsing <a href="matlab:edit('round_constant.v')">round_constant.v</a>.
### Top Module name: 'top'.

```

```
Warning: Unused input port 'i' in 'rconst' module.
```

```
### Hdl Import parsing done.
```

```
### Creating Target model top
```

```
### Generating Dot Layout...
```

```
### Start Layout...
```

```
### Working on hierarchy at ---> 'top'.
```

```
### Laying out components.
```

```
### Working on hierarchy at ---> 'top/top'.
```

```
### Laying out components.
```

```
### Working on hierarchy at ---> 'top/top/NG1'.
```

```
### Laying out components.
```

```
Configurable Subsystem block 'simulink/Ports & Subsystems/Configurable Subsystem' must
```

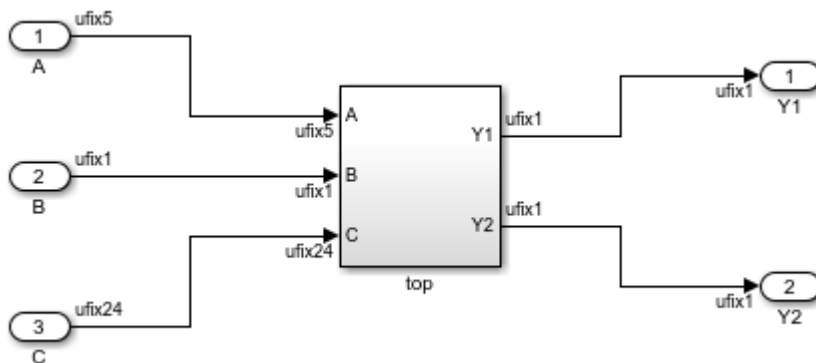
```
### Applying Dot Layout...
### Drawing block edges...
### Working on hierarchy at ---> 'top/top/rconst'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Generated model file C:\TEMP\Examples\hdlcoder-ex12503110\hdlimport\top\top.slx.
### Importhdl completed.
```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `implicit_top.slx`. The generated model uses the same name as the top module that is contained in the input Verilog file `implicit_top.v`.

### Examine Generated Simulink™ Model

To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/example` path relative to the current folder. You can simulate the model and observe the simulation results.

```
addpath('hdlimport/top');
open_system('top.slx')
```

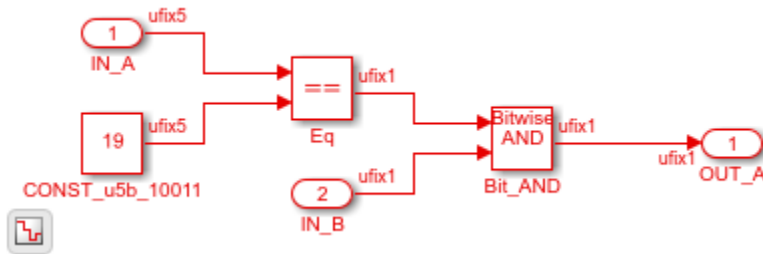


In the rconst Subsystem, one of the input ports is unconnected. It is good practice to avoid unterminated outputs by adding a Terminator block.

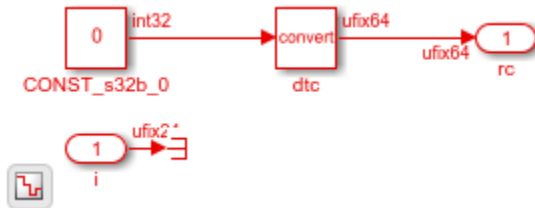
```
addterms('top');
sim('top');
```

You can see the hierarchy of Subsystems that implement the Verilog code that uses module instantiation.

```
open_system('top/top/NG1')
```



```
open_system('top/top/rconst')
```



### Generate Simulink Model from Verilog Code That Infers RAMs

This example shows how you can import a file containing Verilog code and infer RAM blocks in the Simulink™ model that gets generated. You can import Verilog code that infers any of the various RAMs in the HDL RAMs library including the hdl.RAM System-Object based blocks and Block RAMs.

### **Specify Input Verilog File**

Make sure that the input HDL file does not contain any syntax errors, is synthesizable, and uses constructs that are supported by HDL import. This example shows the Verilog code.

```
edit('simple_dual_port_ram.v')
```

```
//Inference of Multiple RAMs
`timescale 1 ns / 1 ns

module SimpleDualPortRAM (clk, enb,
                        wr_din, wr_addr, wr_en,
                        rd_addr, rd_dout1, rd_dout2);

    input  clk, enb;
    input  [15:0] wr_din;
    input  [7:0] wr_addr, rd_addr;
    input  wr_en;
    output [15:0] rd_dout;

    reg [7:0] raml[7:0], ram2[7:0];

    // This is the first RAM block
    always @(posedge clk) begin
        if (enb) begin
            if (wr_en) begin
                raml[wr_addr] <= wr_din;
            end
            rd_out1 <= raml[rd_addr];
        end
    end

    // This is the second RAM block
    always @(posedge clk) begin
        if (enb) begin
            if (wr_en) begin
                ram2[wr_addr] <= wr_din;
            end
            rd_out2 <= raml[rd_addr];
        end
    end

endmodule
```

### Import Verilog File

To import the HDL file and generate the Simulink™ model, pass the file name as a character vector to the `importhdl` function.

```
importhdl('simple_dual_port_ram.v')

### Parsing <a href="matlab:edit('simple_dual_port_ram.v')">simple_dual_port_ram.v</a>
### Top Module name: 'SimpleDualPortRAM'.
### Identified ClkName::clk.
### Hdl Import parsing done.
### Creating Target model SimpleDualPortRAM
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'SimpleDualPortRAM'.
### Laying out components.
### Working on hierarchy at ---> 'SimpleDualPortRAM/SimpleDualPortRAM'.
### Laying out components.
Configurable Subsystem block 'simulink/Ports & Subsystems/Configurable Subsystem' must

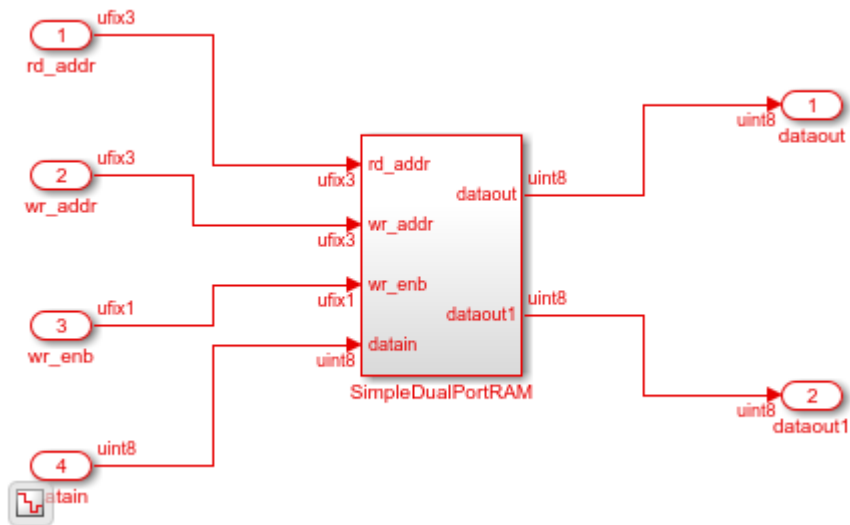
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting model parameters.
### Generated model file C:\TEMP\Examples\hdlcoder-ex67646187\hdlimport\SimpleDualPortRAM.slx
### Importhdl completed.
```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model `SimpleDualPortRAM.slx`. The generated model uses the same name as the top module in the input Verilog file.

### Examine Generated Simulink™ Model

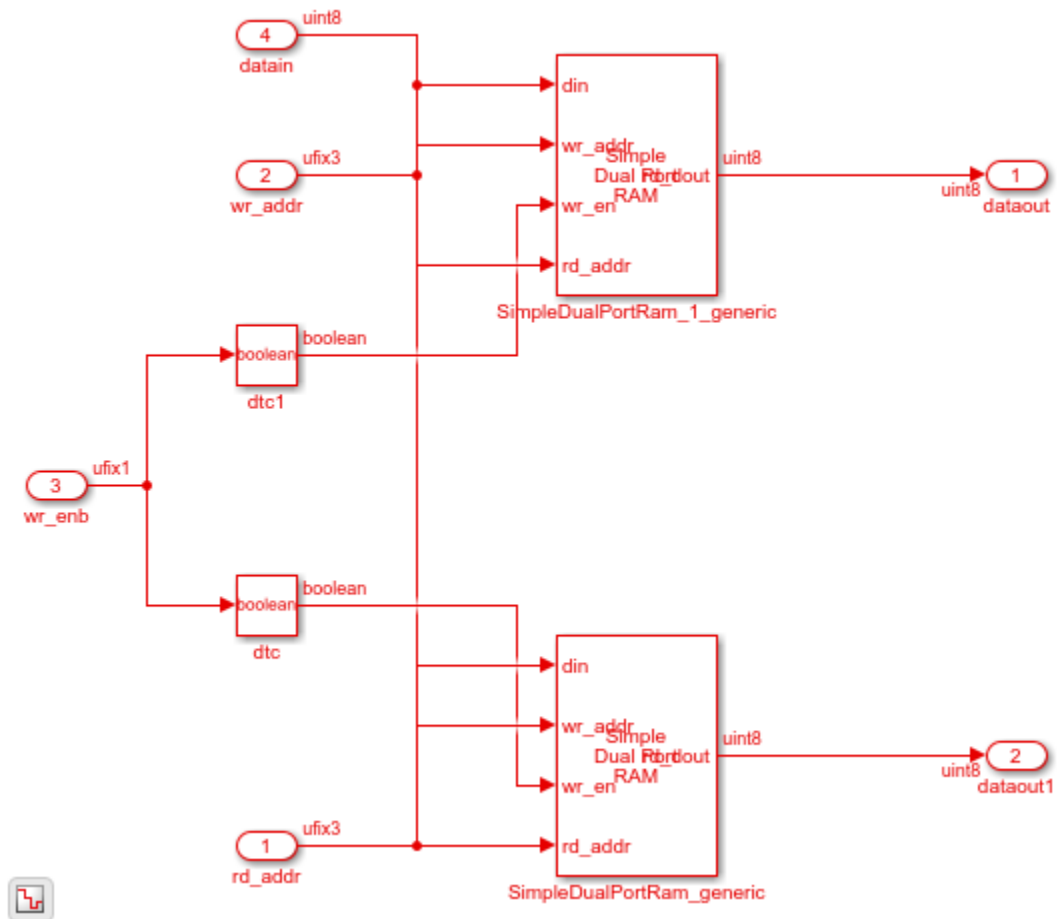
To open the generated Simulink™ model, select the link. The model is saved in the `hdlimport/SimpleDualPortRAM` path relative to the current folder. You can simulate the model and observe the simulation results.

```
addpath('hdlimport/SimpleDualPortRAM');
open_system('SimpleDualPortRAM.slx');
sim('SimpleDualPortRAM.slx');
```



If you navigate the model, you see the Simple Dual Port RAM block.

```
open_system('SimpleDualPortRAM/SimpleDualPortRAM')
```



## Input Arguments

### FileNames — Names of HDL files to import

'Filename' | {'Filename1','Filename2',..., 'FilenameN'} | 'Foldername'

Names of HDL files to import for generation of the Simulink model. By default, `importhdl` imports Verilog files. To import:



- One HDL file, specify the file name as a character vector.
- Multiple HDL files, specify the file names as a cell array of character vectors.
- All HDL files in a folder, specify the folder name as a character vector.
- Multiple folders and combinations of files and folders, specified as cell array of character vectors.

Example: `importhdl('example')` imports the specified Verilog file. If `example` is a subfolder in the current working folder, HDL import generates a Simulink model for all `.vhd` files in that folder.

Example: `importhdl({'top.v', 'subsystem1.v', 'subsystem2.v'})` imports the specified Verilog files and generates the corresponding Simulink model.

Example: `importhdl(pwd)` imports all Verilog files in the current folder and generates the corresponding Simulink model.

Example: `importhdl('root/example/hdlsrc')` imports all Verilog files on the specified path and generates the corresponding Simulink model. You can specify a relative or absolute path.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `importhdl('root/example/hdlsrc')` imports all Verilog files in the specified path and generates the corresponding Simulink model. You can specify a relative or absolute path.

### Language — Language of input HDL file

'Verilog' (default)

Language of input source file that contains the HDL code, specified as a character vector. If you specified a VHDL file, HDL import generates an error.

Example: `importhdl('fifo.v', 'Language', 'Verilog')` imports the Verilog file `fifo.v` and generates the corresponding Simulink model `fifo.slx`.

### topModule — Name of top module or entity

Identified by parsing input file (default) | character vector | string scalar

Top-level module name in the HDL code, specified as a character vector. This name becomes the name of the top-level Subsystem from which HDL import constructs the hierarchy of subsystems in the generated Simulink model. If the input HDL files contain more than one top module, specify the top-level module to use for generating the Simulink model by using the `TopModule` property.

Example: `importhdl('full_adder.v','TopModule','two_half_adders')` imports the Verilog file `full_adder.v` and generates the corresponding Simulink model `full_adder.slx` with `two_half_adders` as the top-level Subsystem.

### **clockBundle** — Clock bundle names

`{'clock', 'reset', 'enable'}` (default) | cell array of character vectors

Names of clock, reset, and clock enable signals for sequential circuits, specified as a cell array of character vector. Default names for the clock bundle signals are:

- Clock signal - `clk`, `clock`
- Reset signal - `rst`, `reset`
- Clock Enable signal - `clk_enb`, `clk_en`, `clk_enable`, `enb`, `enable`

If you do not specify the clock bundle information, HDL import uses the default values. When parsing the input file, if HDL import identifies a clock name that is different from the clock name specified by the `ClockBundle`, the import generates an error.

Example: `importhdl('example.v','clockBundle',{'clk','rst','clk_enb'})` imports the Verilog file `example.v` with the specified clock bundle information.

### **blackBoxModule** — BlackBox module names

`''` (default) | character vector | cell array of character vectors

Name or names of modules in the Verilog input files to be imported as BlackBox subsystems in the generated Simulink model. The Subsystem block that is imported as BlackBox uses the input and output ports that you provide to the module definition. Inside the Subsystem, the input ports are connected to Terminator blocks, Constant blocks with a value of zero are connected to the output ports. Use this capability to import vendor-specific IPs as BlackBox subsystems in your model.

Example:

`importhdl({'example.v','example1.v','example2.v','xilinxIP.v'},'topModule','top','blackBoxModule','xilinxIP')` imports the specified Verilog files with `xilinxIP` as a BlackBox module. The corresponding Subsystem in the Simulink model has the input ports connected to Terminator blocks and Constant blocks with constant value of zero connected to the output ports.

## See Also

checkhdl | makehdl

## Topics

“Verilog HDL Import: Import Verilog Code and Generate Simulink Model”

“Supported Verilog Constructs for HDL Import”

**Introduced in R2018b**

## hdlcoder.supportedDevices

Show supported target hardware and device details

### Syntax

```
hdlcoder.supportedDevices
```

### Description

`hdlcoder.supportedDevices` shows a link to a report that contains the device and device property names for target devices supported by your synthesis tool.

You can use the supported target device information to set `SynthesisToolChipFamily`, `SynthesisToolDeviceName`, `SynthesisToolPackageName`, and `SynthesisToolSpeedValue` for your model.

To see the report link, you must have a synthesis tool set up. If you have more than one synthesis tool available, you see a different report link for each synthesis tool.

### Examples

#### Set the target device for your model

In this example, you set the target device for a model, `sfir_fixed`. Two synthesis tools are available, Altera® Quartus II and Xilinx® ISE. The target device is a Xilinx Virtex-6 XC6VLX130T FPGA.

Show the supported target device reports.

```
hdlcoder.supportedDevices
```

```
Altera QUARTUS II Device List  
Xilinx ISE Device List
```

Click the [Xilinx ISE Device List](#) link to open the supported target device report and view details for your target device.

Open the model, `sfir_fixed`.

```
sfir_fixed
```

Set the `SynthesisToolChipFamily`, `SynthesisToolDeviceName`, `SynthesisToolPackageName`, and `SynthesisToolSpeedValue` model parameters based on details from the supported target device report.

```
hdlset_param('sfir_fixed',
             'SynthesisToolChipFamily', 'Virtex6',
             'SynthesisToolDeviceName', 'xc6vlx130t',
             'SynthesisToolPackageName', 'ff484',
             'SynthesisToolSpeedValue', '-1')
```

View the nondefault parameters for your model, including target device information.

```
hdldispmdlparams
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL CodeGen Parameters (non-default)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
SynthesisTool           : 'Xilinx ISE'
SynthesisToolChipFamily : 'Virtex6'
SynthesisToolDeviceName : 'xc6vlx130t'
SynthesisToolPackageName : 'ff484'
SynthesisToolSpeedValue : -1
```

## See Also

### Topics

“Synthesis Tool Path Setup”  
 “Tool and Device”

**Introduced in R2014a**

## hdldispblkparams

Display HDL block parameters with nondefault values

### Syntax

```
hdldispblkparams(path)  
hdldispblkparams(path, 'all')
```

### Description

`hdldispblkparams(path)` displays, for the specified block, the names and values of HDL parameters that have nondefault values.

`hdldispblkparams(path, 'all')` displays, for the specified block, the names and values of all HDL block parameters.

### Input Arguments

#### **path**

Path to a block or subsystem in the current model.

**Default:** None

#### **'all'**

If you specify `'all'`, `hdldispblkparams` displays the names and values of all HDL properties of the specified block.

### Examples

The following example displays nondefault HDL block parameter settings for a Sum of Elements block).

```

hdldispblkparams('simplevectorsum/vsum/Sum of Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture : Linear

Implementation Parameters

    InputPipeline : 1

```

The following example displays HDL block parameters and values for the currently selected block, (a Sum of Elements block).

```

hdldispblkparams(gcf,'all')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('simplevectorsum/vsum/Sum of
Elements')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture : Linear

Implementation Parameters

    InputPipeline : 0
    OutputPipeline : 0

```

## See Also

“Set and View HDL Block Parameters”

**Introduced in R2010b**

# hdldispmdlparams

Display HDL model parameters with nondefault values

## Syntax

```
hdldispmdlparams(model)  
hdldispmdlparams(model, 'all')
```

## Description

`hdldispmdlparams(model)` displays, for the specified model, the names and values of HDL parameters that have nondefault values.

`hdldispmdlparams(model, 'all')` displays the names and values of all HDL parameters for the specified model.

## Input Arguments

### **model**

Name of an open model.

**Default:** None

### **'all'**

If you pass in `'all'`, `hdldispmdlparams` displays the names and values of all HDL properties of the specified model.

## Examples

The following example displays HDL properties of the current model that have nondefault values.



```

hdldispmdblparams(bdroot)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL CodeGen Parameters (non-default)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
CodeGenerationOutput      : 'GenerateHDLCodeAndDisplayGeneratedModel'
HDLSubsystem              : 'simplevectorsum_2atomics/Subsystem'
OptimizationReport        : 'on'
ResetInputPort            : 'rst'
ResetType                  : 'Synchronous'

```

The following example displays HDL properties and values of the current model.

```

hdldispmdblparams(bdroot, 'all')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL CodeGen Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
AddPipelineRegisters      : 'off'
Backannotation            : 'on'
BlockGenerateLabel        : '_gen'
CheckHDL                  : 'off'
ClockEnableInputPort      : 'clk_enable'
.
.
.
VerilogFileExtension      : '.v'

```

## See Also

“View HDL Model Parameters”

**Introduced in R2010b**

# hdlget\_param

Return value of specified HDL block-level parameter for specified block

## Syntax

```
p = hdlget_param(block_path,prop)
```

## Description

`p = hdlget_param(block_path,prop)` gets the value of a specified HDL property of a block or subsystem, and returns the value to the output variable.

## Input Arguments

### **block\_path**

Path to a block or subsystem in the current model.

**Default:** None

### **prop**

A character vector that designates one of the following:

- The name of an HDL block property of the block or subsystem specified by `block_path`.
- `'all'` : If `prop` is set to `'all'`, `hdlget_param` returns `Name,Value` pairs for HDL properties of the specified block.

**Default:** None

## Output Arguments

**p**

**p** receives the value of the HDL block property specified by **prop**. The data type and dimensions of **p** depend on the data type and dimensions of the value returned. If **prop** is set to 'all', **p** is a cell array.

## Examples

In the following example `hdlget_param` returns the value of the HDL block parameter `OutputPipeline` to the variable **p**.

```
p = hdlget_param(gcb,'OutputPipeline')  
  
p =  
    3
```

In the following example `hdlget_param` returns HDL block parameters and values for the current block to the cell array **p**.

```
p = hdlget_param(gcb,'all')  
  
p =  
    'Architecture'    'Linear'    'InputPipeline'    [0]    'OutputPipeline'    [0]
```

## Tips

- Use `hdlget_param` only to obtain the value of HDL block parameters (see “HDL Block Properties: General” for a list of block implementation parameters). Use `hdldispmdlparams` to see the values of HDL model parameters. To obtain the value of general model parameters, use the `get_param` function.

## See Also

`hdlrestoreparams` | `hdlsaveparams` | `hdlset_param`

**Introduced in R2010b**

## hdllib

Display blocks that are compatible with HDL code generation

### Syntax

```
hdllib
hdllib('off')
hdllib('html')
hdllib('librarymodel')
```

### Description

`hdllib` displays the blocks that are supported for HDL code generation, and for which you have a license, in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this Library Browser view.

If you close and reopen the Library Browser in the same MATLAB session, the Library Browser continues to show only the blocks supported for HDL code generation. To show all blocks, regardless of HDL code generation compatibility, at the command prompt, enter `hdllib('off')`.

`hdllib('off')` displays all the blocks for which you have a license in the Library Browser, regardless of HDL code generation compatibility.

`hdllib('html')` creates a library of blocks that are compatible with HDL code generation. It generates two additional HTML reports: a categorized list of blocks (`hdlblklist.html`) and a table of blocks and their HDL code generation parameters (`hdlsupported.html`).

To run `hdllib('html')`, you must have an HDL Coder license.

`hdllib('librarymodel')` displays blocks that are compatible with HDL code generation in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this library.

The default library name is `hdl_supported`. After you generate the library, you can save it to a folder of your choice.

To keep the library current, you must regenerate it each time that you install a new software release.

To run `hdlLib('librarymodel')`, you must have an HDL Coder license.

## Examples

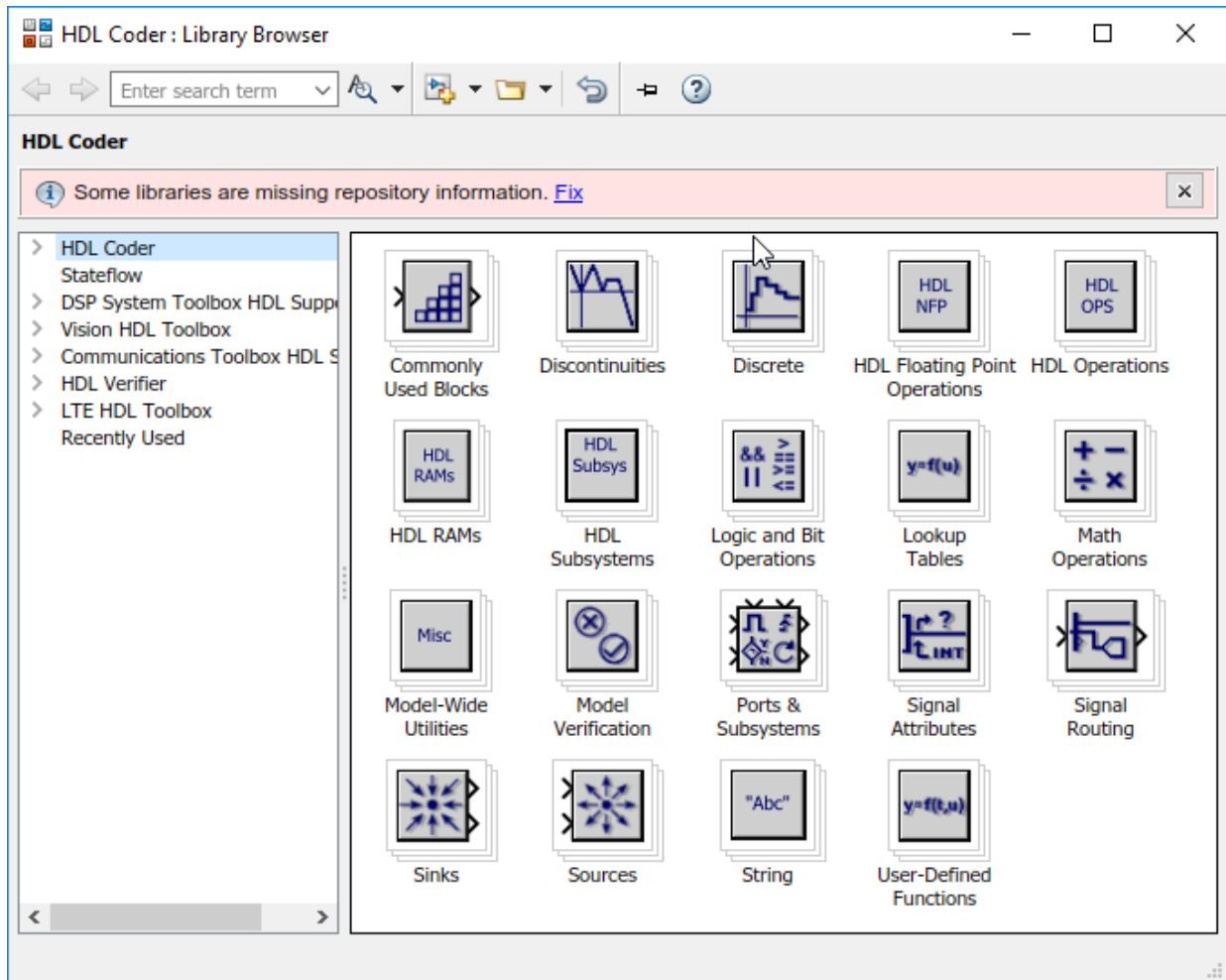
### Display Supported Blocks in the Library Browser

To display blocks that are compatible with HDL code generation in the Library Browser:

```
hdlLib
```

```
### Generating view of HDL Coder compatible blocks in Library Browser.
```

```
### To restore the Library Browser to the default Simulink view, enter "hdlLib off".
```

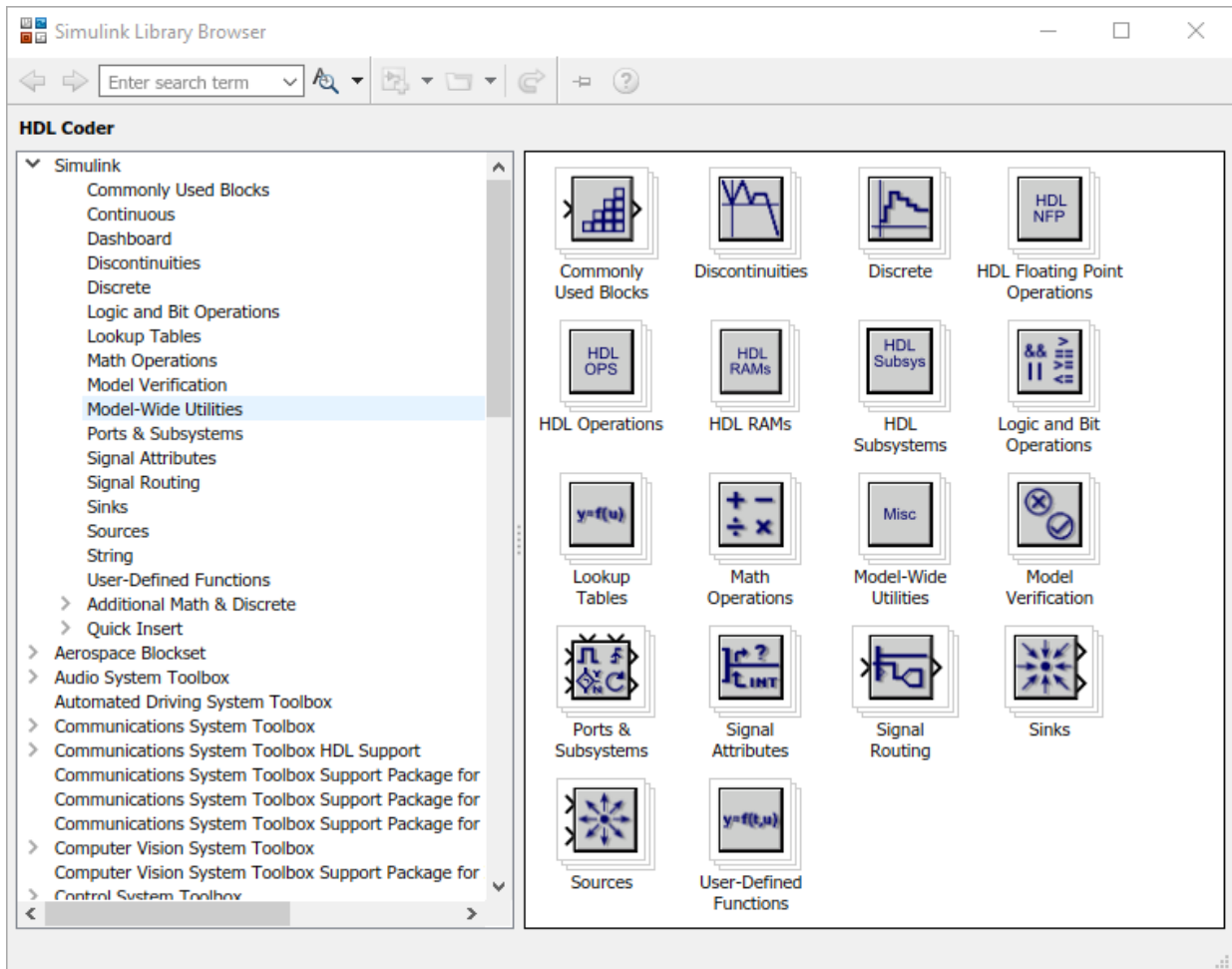


### Display All Blocks in the Library Browser

To display all blocks in the Library Browser, regardless of HDL code generation compatibility:

```
hdllib('off')
```

```
### Restoring Library Browser to default view; removing the HDL Coder compatibility filter
```



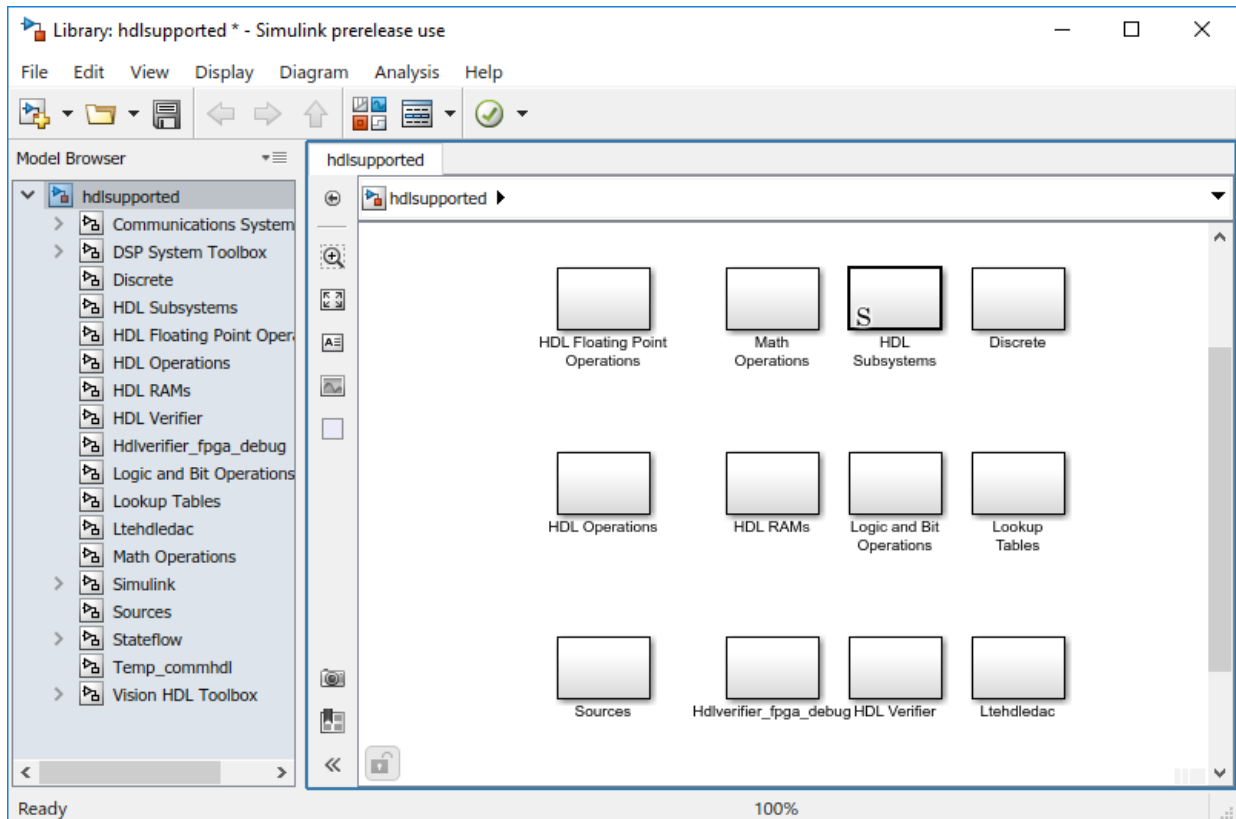
## Create a Supported Blocks Library and HTML Reports

To create a library and HTML reports showing the blocks that are compatible with HDL code generation:

```
hdlLib('html')
```

```
### HDL supported block list hdlblklist.html  
### HDL implementation list hdlsupported.html
```

The `hdlsupported` library opens. To view the reports, click the `hdlblklist.html` and `hdlsupported.html` links.



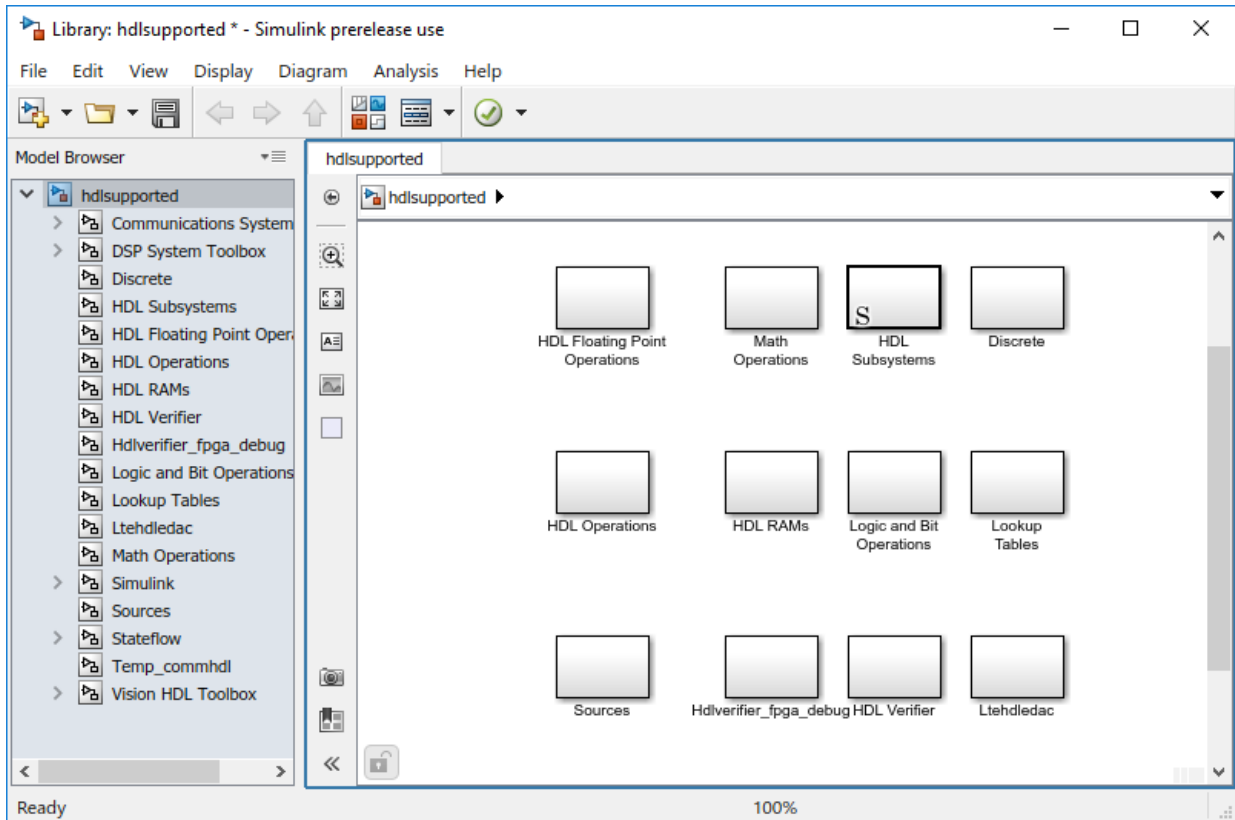
### Create a Supported Blocks Library

To create a library that contains blocks that are compatible with HDL code generation:

```
hdlLib('librarymodel')
```

The `hdlsupported` block library opens.





## See Also

“Supported Blocks”

## Topics

“Show Blocks Supported for HDL Code Generation”

“View HDL-Specific Block Documentation”

“Create Simulink Model for HDL Code Generation”

**Introduced in R2006b**

## hdlmodelchecker

Open HDL Model Checker

### Syntax

```
hdlmodelchecker(subsystem)  
hdlmodelchecker(model)
```

### Description

`hdlmodelchecker(subsystem)` opens the Model Checker for the subsystem within the model.

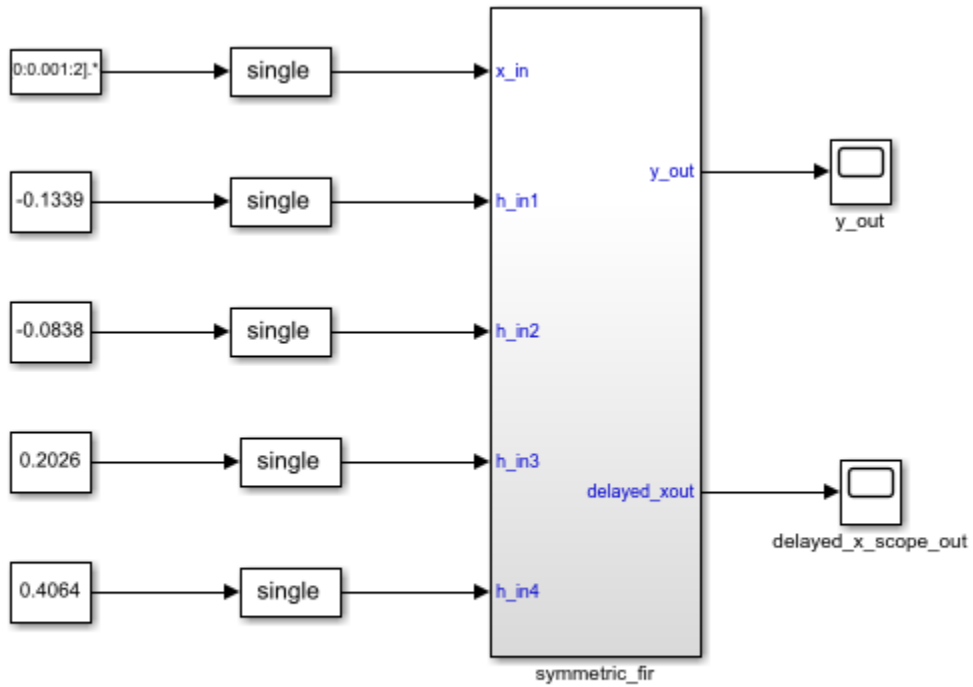
`hdlmodelchecker(model)` opens the Model Checker for the model.

### Examples

#### Open the HDL Model Checker For a Model

This example shows how to open the HDL Model Checker for the `sfir_single` model.

```
sfir_single
```



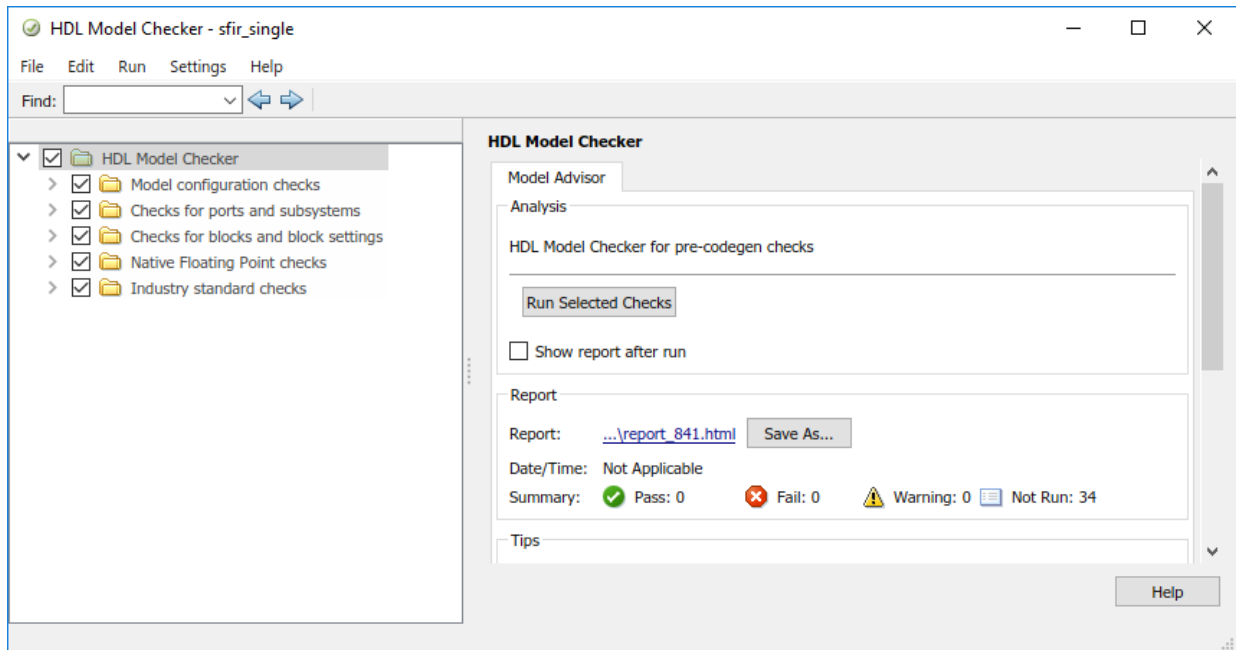
Copyright 2016-2017 The MathWorks, Inc.

To open the HDL Model Checker for the `sfir_single` model, enter:

```
hdlmodelchecker('sfir_single')
```

Updating Model Advisor cache...

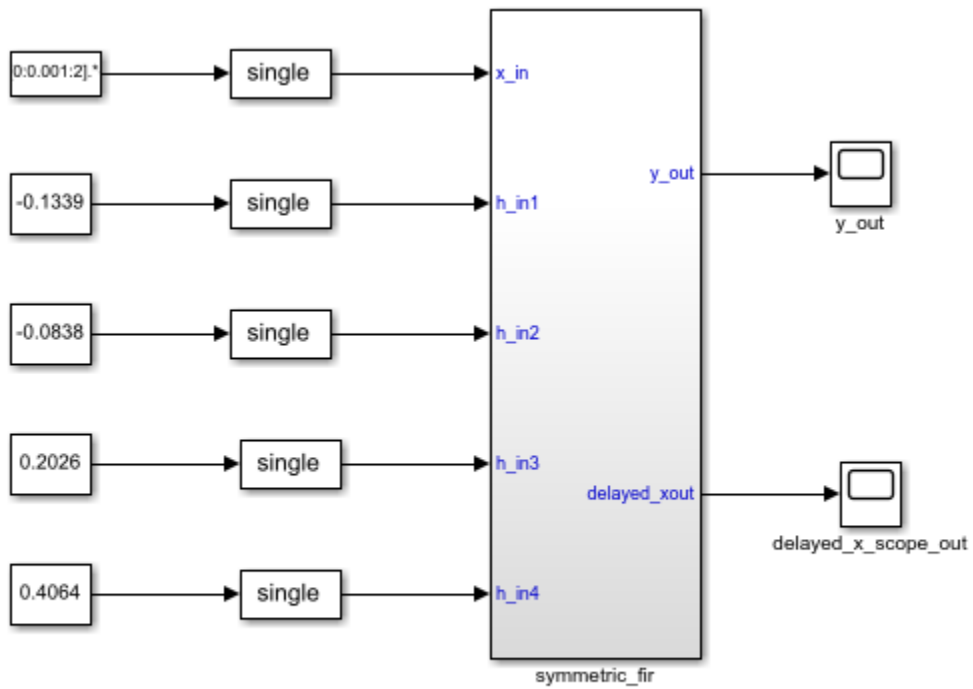
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor



### Open the HDL Model Checker For a Subsystem

This example shows how to open the HDL Model Checker for the `symmetric fir` subsystem within the `sfir_single` model.

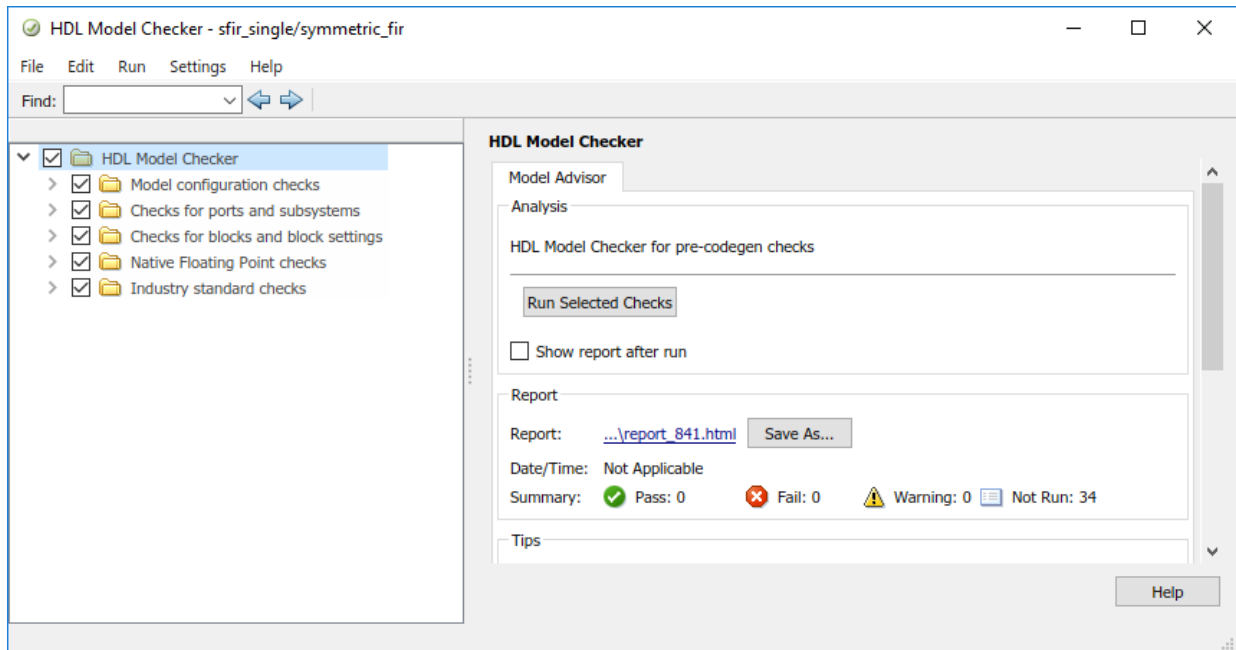
```
sfir_single
```



Copyright 2016-2017 The MathWorks, Inc.

To open the HDL Model Checker for the Symmetric fir subsystem, enter:

```
hdlmodelchecker('sfir_single/symmetric_fir')
```



## Input Arguments

### **subsystem** — Subsystem name

character vector

Subsystem name or handle, specified as a character vector.

Data Types: char

### **model** — Model name

character vector

Model name or handle, specified as a character vector.

Data Types: char

## **See Also**

### **Topics**

“Getting Started with the HDL Model Checker”

“Model Checks in HDL Coder”

**Introduced in R2017b**

# hdlrestoreparams

Restore block- and model-level HDL parameters to model

## Syntax

```
hdlrestoreparams(dut)  
hdlrestoreparams(dut,filename)
```

## Description

`hdlrestoreparams(dut)` restores to the specified model the default block- and model-level HDL settings.

`hdlrestoreparams(dut,filename)` restores to the specified model the block- and model-level HDL settings from a previously saved file.

## Examples

### Save and Restore HDL-Related Model Parameters

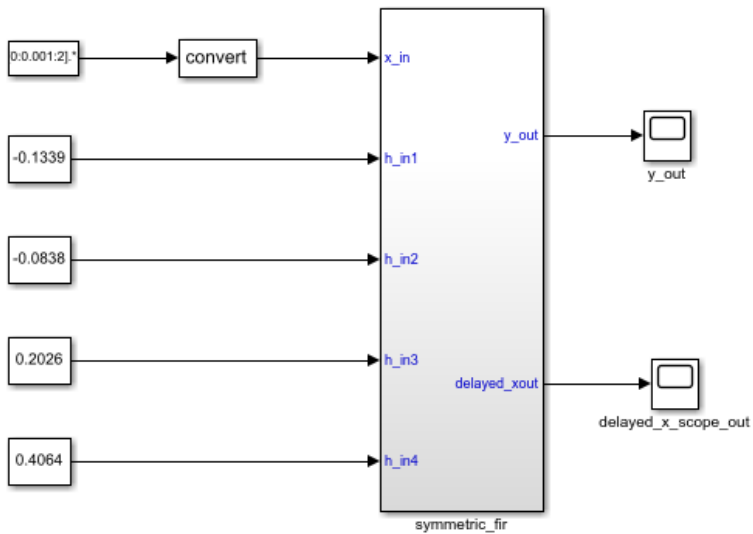
This example shows how to set HDL parameters on a model and save the parameters in a MATLAB® script.

#### Set Model HDL Parameters

Open the `sfir_fixed` model.

```
sfir_fixed
```





This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdltb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

Launch HDL Dialog

Run Demo

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
```

Set HDL-related model parameters for the `symmetric_fir` subsystem.

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3)
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5)
```

### Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');

% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

Save the model parameters to a MATLAB® script, `sfir_saved_params.m`.

```
hdlsaveparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

### Verify Saved Parameters

Reset HDL-related model parameters to default values.

```
hdlrestoreparams('sfir_fixed/symmetric_fir')
```

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

Restore the saved model parameters from `sfir_saved_params.m`.

```
hdlrestoreparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
```

```
% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
```

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

## Input Arguments

### **dut** — DUT subsystem name

character vector

DUT subsystem name, specified as a character vector, with full hierarchical path.

Example: 'modelName/subsysTarget'

Example: 'modelName/subsysA/subsysB/subsysTarget'

### **filename** — Name of file

character vector

Name of file containing previously saved HDL model parameters.

Example: 'mymodel\_saved\_params.m'

## See Also

hdlsaveparams

**Introduced in R2012b**

## hdlsaveparams

Save nondefault block- and model-level HDL parameters

### Syntax

```
hdlsaveparams(dut)
hdlsaveparams(dut,filename)
hdlsaveparams(dut,filename,force_overwrite)
varname = hdlsaveparams(dut)
```

### Description

`hdlsaveparams(dut)` displays nondefault block- and model-level HDL parameters.

`hdlsaveparams(dut,filename)` saves nondefault block- and model-level HDL parameters to a MATLAB script.

`hdlsaveparams(dut,filename,force_overwrite)` saves nondefault block- and model-level HDL parameters to a MATLAB script and specifies whether to overwrite the previously saved parameters MATLAB script.

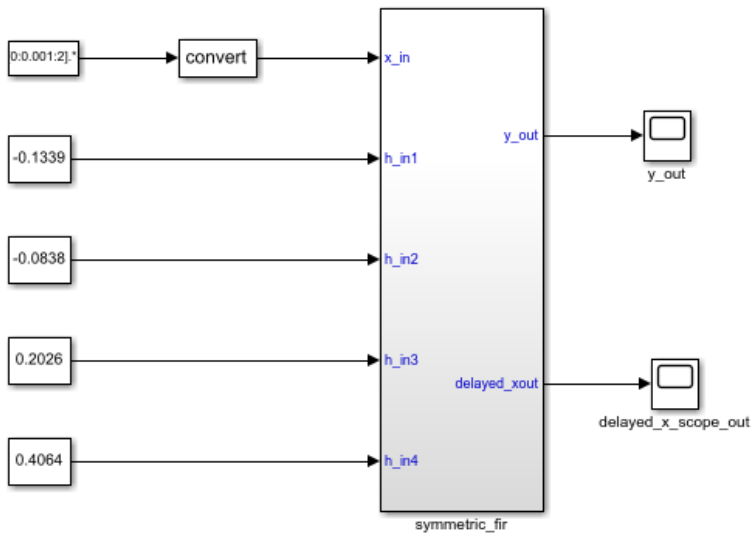
`varname = hdlsaveparams(dut)` saves the nondefault block- and model-level HDL parameters to a structure array, `varname`.

### Examples

#### Display HDL-Related Nondefault Model Parameters

Open the model.

```
sfir_fixed
```



This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdlb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

Launch HDL Dialog

Run Demo

Copyright 2007 The MathWorks, Inc.

Set HDL-related model parameters for the `symmetric_fir` subsystem.

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3)
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5)
```

Display HDL-related nondefault model parameters for the `symmetric_fir` subsystem.

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
```

```
% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

The output identifies the subsystem and displays its HDL-related parameter values.

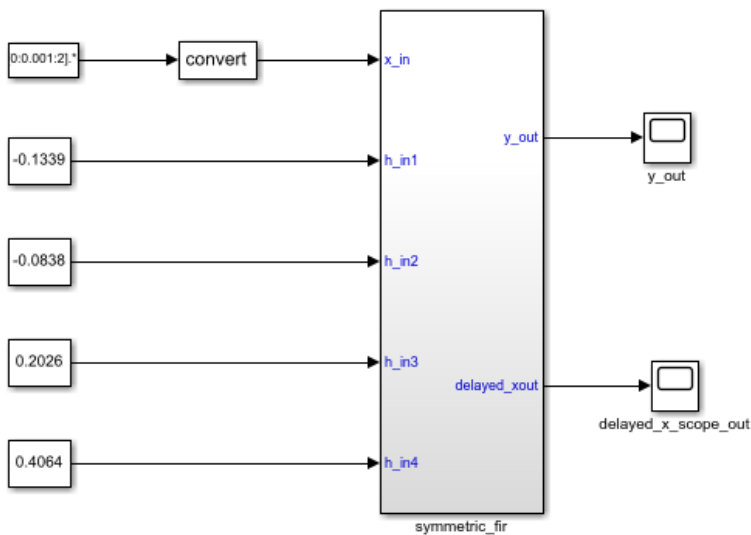
### Save and Restore HDL-Related Model Parameters

This example shows how to set HDL parameters on a model and save the parameters in a MATLAB® script.

#### Set Model HDL Parameters

Open the `sfir_fixed` model.

```
sfir_fixed
```



This example shows how to use HDL Code to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdltb('sfir_fixed/symmetric_fir')`  
Or double-click the blue button at the bottom to see the dialog.

**Launch HDL Dialog**

**Run Demo**

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
```

Set HDL-related model parameters for the `symmetric_fir` subsystem.

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3)  
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5)
```

### Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');  
  
% Set SubSystem HDL parameters  
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);  
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

Save the model parameters to a MATLAB® script, `sfir_saved_params.m`.

```
hdlsaveparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

### Verify Saved Parameters

Reset HDL-related model parameters to default values.

```
hdlrestoreparams('sfir_fixed/symmetric_fir')
```

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

Restore the saved model parameters from `sfir_saved_params.m`.

```
hdlrestoreparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')

%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');

% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

### **Overwrite Previously Saved HDL Parameters File**

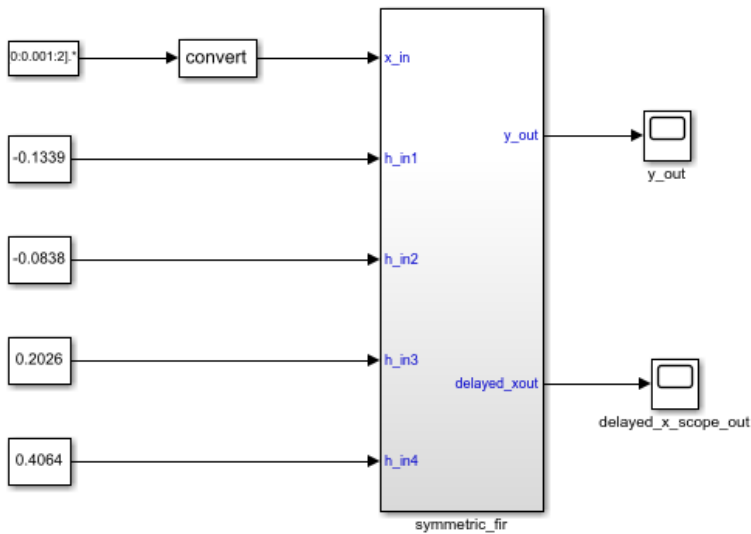
This example shows how to set HDL parameters on a model, save the parameters in a MATLAB® script, and then overwrite the saved parameters.

#### **Set Model HDL Parameters**

Open the `sfir_fixed` model.

```
sfir_fixed
```





This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdlb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

Launch HDL Dialog

Run Demo

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
```

Set HDL-related model parameters for the `symmetric_fir` subsystem.

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3)
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5)
```

### Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');  
  
% Set SubSystem HDL parameters  
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);  
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

Save the model parameters to a MATLAB® script, `sfir_saved_params.m`.

```
hdlsaveparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

### Verify Saved Parameters

Reset HDL-related model parameters to default values.

```
hdlrestoreparams('sfir_fixed/symmetric_fir')
```

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

Restore the saved model parameters from `sfir_saved_params.m`.

```
hdlrestoreparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')  
  
%% Set Model 'sfir_fixed' HDL parameters  
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');  
  
% Set SubSystem HDL parameters  
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);  
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 3);
```

### Modify Saved HDL Parameters

Modify HDL-related model parameters set for the `symmetric_fir` subsystem.

```
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 4)
hdlset_param('sfir_fixed/symmetric_fir', 'OutputPipeline', 2)
hdlset_param('sfir_fixed', 'ShareAdders', 'on')
```

### Overwrite Saved Parameters File

Set the `force_overwrite` flag to `true` to overwrite the parameters file `sfir_saved_parameters.m` with the new parameters. If you do not specify this flag, HDL Coder™ generates an error and doesn't overwrite the parameter values. When you run `hdlsaveparams` with the parameter set to `true`, HDL Coder™ generates a warning that it overwrites the file.

```
hdlsaveparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m', 'true')
```

```
Warning: HDL parameters file 'sfir_saved_params.m' already exists. By
overwriting it now, you will lose any parameter settings made earlier.
```

### Verify Resaved Parameters

Reset HDL-related model parameters to default values.

```
hdlrestoreparams('sfir_fixed/symmetric_fir')
```

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

Restore the saved model parameters from `sfir_saved_params.m`.

```
hdlrestoreparams('sfir_fixed/symmetric_fir', 'sfir_saved_params.m')
```

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')
```

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'ShareAdders', 'on');
```

```
% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
```

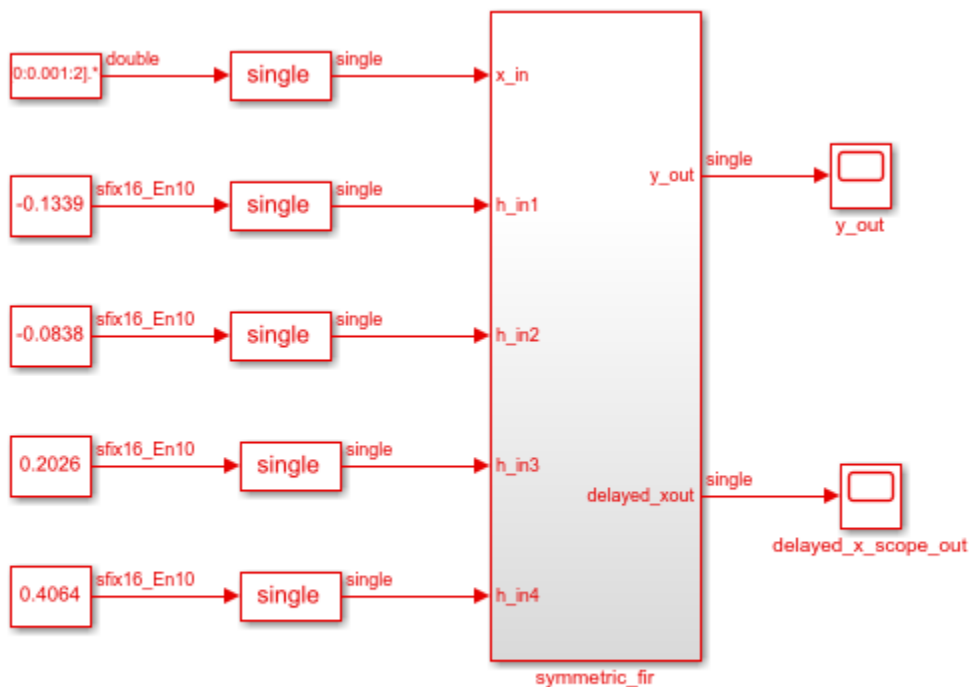
```
hdlset_param('sfir_fixed/symmetric_fir', 'OutputPipeline', 2);  
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 4);
```

### Save and Access Non-Default HDL Parameters in a Structure Array

This example shows how to save non-default HDL model and block parameters in a structure array and access individual parameters.

#### Open the model

```
sfir_single  
sim('sfir_single')
```



Copyright 2016-2017 The MathWorks, Inc.

## Save HDL Model and Block parameters

```

hparams = hdlsaveparams('sfir_single/symmetric_fir');

%% Set Model 'sfir_single' HDL parameters
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', hdlcoder.createFloatingPointTargetConfiguration, 'MantissaMultiplyStrategy', 'FullMultiplier') ...
);
hdlset_param('sfir_single', 'HDLSubsystem', 'sfir_single/symmetric_fir');

% Set SubSystem HDL parameters
hdlset_param('sfir_single/symmetric_fir', 'InputPipeline', 1);
hdlset_param('sfir_single/symmetric_fir', 'OutputPipeline', 1);

```

## View and Access Block Parameters

```
hparams
```

```
hparams =
```

```
1x4 struct array with fields:
```

```

    object
    parameter
    value

```

To see specific non-default parameters saved in the structure, you can access individual elements of the structure.

```
hparams(2)
```

```
ans =
```

```
struct with fields:
```

```

    object: 'sfir_single'
    parameter: 'HDLSubsystem'
    value: 'sfir_single/symmetric_fir'

```

To view the parameters and values specified for the model, in the MATLAB™ Workspace, double-click the `hparams` variable. You see the fields of the structure array and the corresponding values in the MATLAB Editor.

The screenshot shows a MATLAB Editor window titled 'hparams'. Below the title bar, it displays '1x4 struct with 3 fields'. A table below shows the structure's fields and values:

Fields	object	parameter	value
1	'sfir_single'	'FloatingPointTargetConfiguration'	1x1 FloatingPointTargetConfig
2	'sfir_single'	'HDLSubsystem'	'sfir_single/symmetric_fir'
3	'sfir_single/...	'InputPipeline'	1
4	'sfir_single/...	'OutputPipeline'	1
5			

## Input Arguments

### **dut** — DUT subsystem name

character vector

DUT subsystem name, specified as a character vector, with full hierarchical path.

Example: 'modelName/subsysTarget'

Example: 'modelName/subsysA/subsysB/subsysTarget'

### **filename** — Name of file

character vector

Name of file to which you are saving model parameters, specified as a character vector.

Example: 'mymodel\_saved\_params.m'

### **force\_overwrite** — Overwrite parameters file

boolean

Specify whether to overwrite the previously saved parameters file as a boolean.

Example: 'true'

## Output Arguments

**varname** — Name of variable containing saved parameters

struct

Specify the name of the variable that contains the saved model parameters. The variable are saved as a structure array.

Example: 'hparams'

## See Also

hdlrestoreparams

**Introduced in R2012b**

## hdlset\_param

Set HDL-related parameters at model or block level

### Syntax

```
hdlset_param(path,Name,Value)
```

### Description

`hdlset_param(path,Name,Value)` sets HDL-related parameters in the block or model referenced by `path`. The parameters to be set, and their values, are specified by one or more `Name,Value` pair arguments. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Input Arguments

#### **path**

Path to the model or block for which `hdlset_param` is to set one or more parameter values.

**Default:** None

#### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments, where `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`.

#### **Name**

`Name` is a character vector that specifies one of the following:



- A model-level HDL-related property. See Properties — Alphabetical List for a list of model-level properties, their data types and their default values.
- An HDL block property, such as an implementation name or an implementation parameter. See “HDL Block Properties: General” for a list of block implementation parameters.

**Default:** None

### Value

Value is a value to be applied to the corresponding property in a Name, Value argument.

**Default:** Default value is dependent on the property.

## Examples

The following example uses the `sfir_fixed` model to demonstrate how to locate a group of blocks in a subsystem and specify the same output pipeline depth for each of the blocks.

```
open sfir_fixed;
prodblocks = find_system('sfir_fixed/symmetric_fir', 'BlockType', 'Product');
for ii=1:length(prodblocks), hdlset_param(prodblocks{ii}, 'OutputPipeline', 2), end;
```

## Tips

- When you set multiple parameters on the same model or block, use a single `hdlset_param` command with multiple pairs of arguments, rather than multiple `hdlset_param` commands. This technique is more efficient because using a single call requires evaluating parameters only once.
- To set HDL block parameters for multiple blocks, use the `find_system` function to locate the blocks of interest. Then, use a loop to iterate over the blocks and call `hdlset_param` to set the desired parameters.

## See Also

`hdlget_param` | `hdlrestoreparams` | `hdlsaveparams`

## **Topics**

“Set and View HDL Block Parameters”

“Set HDL Block Parameters for Multiple Blocks”

**Introduced in R2010b**

# hdlsetup

Set up model parameters for HDL code generation

## Syntax

```
hdlsetup('modelname')
```

## Description

`hdlsetup('modelname')` sets the parameters of the model specified by *modelname* to common default values for HDL code generation. After using `hdlsetup`, you can use `set_param` to modify these default settings.

Open the model before you invoke the `hdlsetup` command.

To see which model parameters are affected by `hdlsetup`, open `hdlsetup.m`.

## How hdlsetup Configures Solver Options

`hdlsetup` configures the **Solver** options that are recommended or required by HDL Coder. These are:

- **Type:** Fixed-step. (HDL Coder currently supports variable-step solvers under limited conditions. See `hdlsetup`)
- **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually the best one for simulating discrete systems.
- **Tasking mode:** SingleTasking. HDL Coder does not currently support models that execute in multitasking mode.

Do not set **Tasking mode** to Auto.

`hdlsetup` also configures the model start and stop times and fixed-step size as follows:

- **Start Time:** 0.0 s

- **Stop Time:** 10 s
- **Fixed step size (fundamental periodic sample time) :** auto

If **Fixed step size** is set to auto the step size is chosen automatically, based on the sample times specified in the model. In the example model, only the Signal From Workspace block specifies an explicit sample time (1 s); the other blocks inherit this sample time.

The model start and stop times determine the total simulation time. This in turn determines the size of data arrays that are generated to provide stimulus and output data for generated test benches. For the example model, computation of 10 seconds of test data does not take a significant amount of time. Computation of sample values for more complex models can be time consuming. In such cases, you may want to decrease the total simulation time.

The remaining parameters set by `hdlsetup` control error severity levels, data logging, and model display options. If you want to view the complete set of model parameters affected by `hdlsetup`, open `hdlsetup.m` in the MATLAB Editor.

The model parameter settings provided by are intended as useful defaults, but they may not be optimal for your application. For example, `hdlsetup` sets a default **Simulation stop time** of 10 s. A total simulation time of 1000 s would be more realistic for a test of the `sfir_fixed` example model. If you would like to change the simulation time, enter the desired value into the **Simulation stop time** field of the Simulink window.

See the “Model Parameters” table in the “Model and Block Parameters” section of the Simulink documentation for a summary of model parameters.

**Introduced in R2006b**

# hdlsetuptoolpath

Set up system environment to access FPGA synthesis software

## Syntax

```
hdlsetuptoolpath('ToolName', TOOLNAME, 'ToolPath', TOOLPATH)
```

## Description

`hdlsetuptoolpath('ToolName', TOOLNAME, 'ToolPath', TOOLPATH)` adds a third-party FPGA synthesis tool to your system path. It sets up the system environment variables for the synthesis tool. To configure one or more supported third-party FPGA synthesis tools to use with HDL Coder, use the `hdlsetuptoolpath` function.

Before opening the HDL Workflow Advisor, add the tool to your system path. If you already have the HDL Workflow Advisor open, see “Add Synthesis Tool for Current HDL Workflow Advisor Session”.

## Examples

### Set Up Intel Quartus Prime

The following command sets the synthesis tool path to point to an installed Intel® Quartus Prime Standard Edition 17.1 executable file. You must have already installed Altera Quartus II.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', 'ToolPath', ...  
    'C:\intel\17.1\quartus\bin\quartus.exe');
```

### Set Up Xilinx ISE

The following command sets the synthesis tool path to point to an installed Xilinx ISE 14.7 executable file. You must have already installed Xilinx ISE.

```
hdlsetuptoolpath('ToolName', 'Xilinx ISE', 'ToolPath', ...  
'C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\ise.exe');
```

### Set Up Xilinx Vivado

The following command sets the synthesis tool path to point to an installed Vivado® Design Suite 2018.2 batch file. You must have already installed Xilinx Vivado.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', ...  
'C:\Xilinx\Vivado\2018.2\bin\vivado.bat');
```

### Set Up Microsemi Libero SoC

The following command sets the synthesis tool path to point to an installed Microsemi® Libero® Design Suite batch file. You must have already installed Microsemi Libero SoC.

```
hdlsetuptoolpath('ToolName', 'Microsemi Libero SoC', 'ToolPath', ...  
'C:\Microsemi\Libero_SoC_v11.8\Designer\bin');
```

## Input Arguments

### **TOOLNAME** — Synthesis tool name

character vector

Synthesis tool name, specified as a character vector.

Example: 'Xilinx Vivado'

### **TOOLPATH** — Full path to the synthesis tool executable or batch file

character vector

Full path to the synthesis tool executable or batch file, specified as a character vector.

Example: 'C:\Xilinx\Vivado\2018.2\bin\vivado.bat'

## Tips

- If you have an icon for the tool on your Windows® desktop, you can find the full path to the synthesis tool.

- 1 Right-click the icon and select **Properties**.
  - 2 Click the **Shortcut** tab.
- The `hdlsetuptoolpath` function changes the system path and system environment variables for only the current MATLAB session. To execute `hdlsetuptoolpath` programmatically when MATLAB starts, add `hdlsetuptoolpath` to your `startup.m` script.

## See Also

`setenv` | `startup`

## Topics

“Supported Third-Party Tools and Hardware”

“Tool Setup”

“Add Synthesis Tool for Current HDL Workflow Advisor Session”

**Introduced in R2011a**

# makehdl

Generate HDL RTL code from model, subsystem, or model reference

## Syntax

```
makehdl (dut)  
makehdl (dut, Name, Value)
```

## Description

`makehdl (dut)` generates HDL code from the specified DUT model, subsystem, or model reference.

---

**Note** Running this command can activate the **Open at simulation start** setting for blocks such as the Scope block and therefore invoke the block.

---

`makehdl (dut, Name, Value)` generates HDL code from the specified DUT model, subsystem, or model reference with options specified by one or more name-value pair arguments.

## Examples

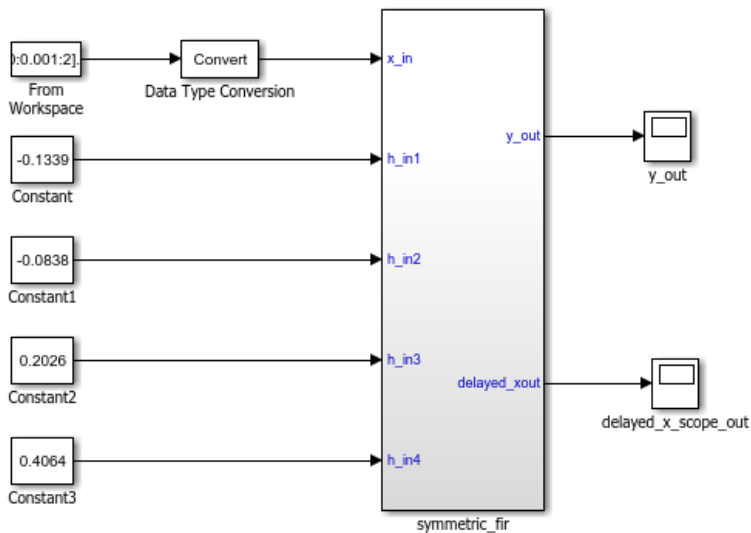
### Generate VHDL for the Current Model

This example shows how to generate VHDL for the symmetric FIR model.

Open the `sfir_fixed` model.

```
sfir_fixed
```





This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdltb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

**Launch HDL Dialog**

**Run Demo**

Copyright 2007 The MathWorks, Inc.

Generate HDL code for the current model with code generation options set to default values.

```
makehdl('sfir_fixed/symmetric_fir', 'TargetDirectory', 'C:\GenVHDL\hdlsrc')
```

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\GenVHDL\hdlsrc\sfir_fixed\symmetric_fir.v
### Creating HDL Code Generation Check Report file://C:\GenVHDL\hdlsrc\sfir_fixed\symme
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

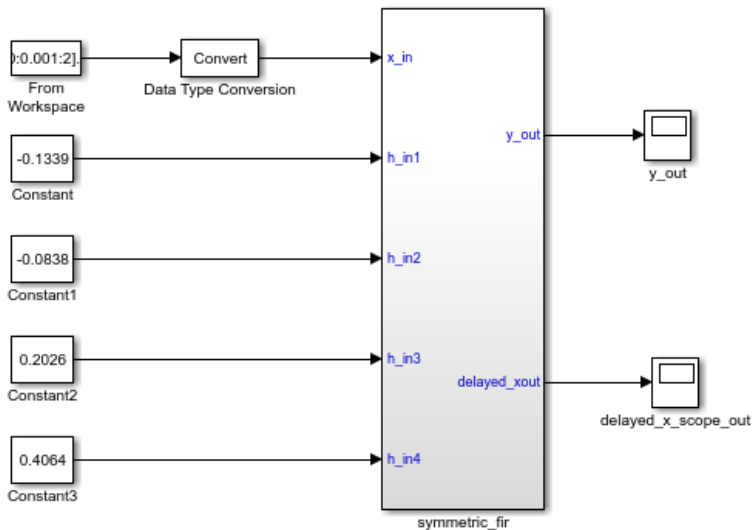
The generated VHDL code is saved in the hdlsrc folder.

### Generate Verilog for a Subsystem Within a Model

Generate Verilog® for the subsystem `symmetric_fir` within the model `sfir_fixed`.

Open the `sfir_fixed` model.

```
sfir_fixed;
```



This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:

```
checkhdl('sfir_fixed/symmetric_fir')
makehdl('sfir_fixed/symmetric_fir')
makehdltb('sfir_fixed/symmetric_fir')
```

Or double-click the blue button at the bottom to see the dialog.

Launch HDL Dialog

Run Demo

Copyright 2007 The MathWorks, Inc.

The model opens in a new Simulink® window.

Generate Verilog for the `symmetric_fir` subsystem.

```
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog', ...
        'TargetDirectory', 'C:/Generate_Verilog/hdlsrc')
```

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
```

```
### Begin Verilog Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\Generate_Verilog\hdlsrc\sfir_fixed\symmet
### Creating HDL Code Generation Check Report file://C:\Generate_Verilog\hdlsrc\sfir_f
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated Verilog code for the `symmetric_fir` subsystem is saved in `hdlsrc\sfir_fixed\symmetric_fir.v`.

Close the model.

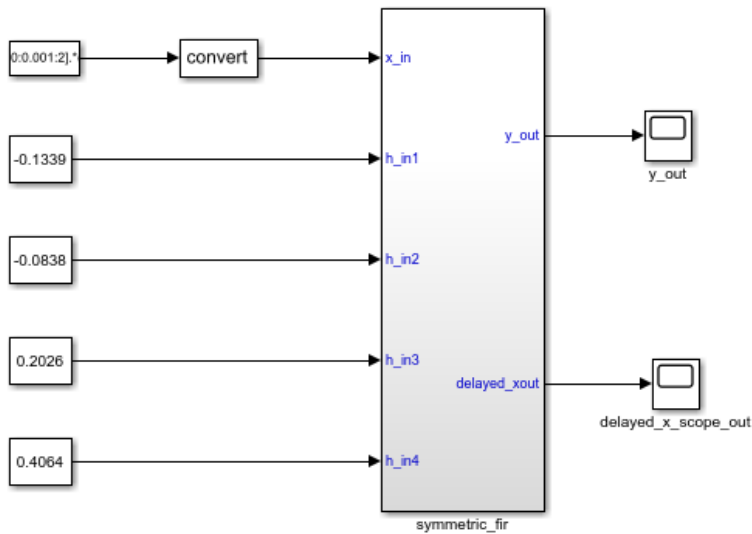
```
bdclose('sfir_fixed');
```

### **Check Subsystem for Compatibility with HDL Code Generation**

Check that the subsystem `symmetric_fir` is compatible with HDL code generation, then generate HDL.

Open the `sfir_fixed` model.

```
sfir_fixed
```



This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdlb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

**Launch HDL Dialog**

**Run Demo**

Copyright 2007 The MathWorks, Inc.

The model opens in a new Simulink® window.

Use the `checkhdl` function to check whether the `symmetric_fir` subsystem is compatible with HDL code generation.

```
hdlset_param('sfir_fixed', 'TargetDirectory', 'C:/HDL_Checks/hdlsrc');
checkhdl('sfir_fixed/symmetric_fir')
```

```
### Starting HDL check.
```

```
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\sfir_fixed\sy
```

```
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
```

`checkhdl` completed successfully, which means that the model is compatible for HDL code generation. To generate code, use `makehdl`

```
makehdl('sfir_fixed/symmetric_fir')
```

```

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\HDL_Checks\hdlsrc\sfir_fixed\symmetric_f
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\sfir_fixed\sy
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

```

The generated VHDL® code for the `symmetric_fir` subsystem is saved in `hdlsrc\sfir_fixed\symmetric_fir.vhd`.

Close the model.

```
bdclose('sfir_fixed');
```

## Input Arguments

### **dut** — DUT model or subsystem name

character vector

Specified as subsystem name, top-level model name, or model reference name with full hierarchical path.

Example: `'top_level_name'`

Example: `'top_level_name/subsysA/subsysB/codegen_subsys_name'`

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'TargetLanguage', 'Verilog'`

### Target Options

#### **HDLSubsystem** — DUT Subsystem

character vector

Specify the Subsystem in your model to generate HDL code for. For more information, see the **Generate HDL for** section in “Target”.

### **TargetLanguage — Target language**

'VHDL' (default) | 'Verilog'

Specify whether to generate VHDL or Verilog code. For more information, see the **Language** section in “Target”.

### **TargetDirectory — Output directory**

'hdlsrc' (default) | character vector

Specify a path to write the generated files and HDL code into. For more information, see the **Folder** section in “Target”.

### **SplitEntityArch — Split VHDL entity and architecture into separate files**

'off' (default) | 'on'

For more information, see **Split entity and architecture** in “Split entity and architecture”.

### **UseSingleLibrary — Generate VHDL code for model references into a single library**

'off' (default) | 'on'

For more information, see “Generate VHDL code for model references into a single library”.

### **Code Generation Output Options**

#### **CodeGenerationOutput — Generation of HDL code and display of generated model**

'GenerateHDLCode' (default) | 'GenerateHDLCodeAndDisplayGeneratedModel' | 'DisplayGeneratedModelOnly'

Specify whether you want to generate HDL code, or only display the generated model, or generate HDL code and display the generated model. For more information, see the **Generate HDL code** section in “Code Generation Output”.

#### **GenerateHDLCode — Generate HDL code**

'on' (default) | 'off'

Specify whether to generate HDL code for the model. For more information, see the **Generate HDL code** section in “Code Generation Output”.

### **GenerateValidationModel — Generate validation model**

'off' (default) | 'on'

Specify whether to generate the validation model with HDL code. For more information, see the **Generate validation model** section in “Code Generation Output”.

### **Code Generation Report Options**

#### **HDLCodingStandard — Specify HDL coding standard**

character vector

Specify whether the generated HDL code must conform to the Industry coding standard guidelines. For more information, see “Choose Coding Standard and Report Options”.

#### **HDLCodingStandardCustomizations — Specify HDL coding standard customization object**

hdlcoder.CodingStandard object

Specify the coding standards customization object to use with the Industry coding standard when generating HDL code. For more information, see `hdlcoder.CodingStandard`.

#### **Traceability — Generate report with mapping links between HDL and model**

'off' (default) | 'on'

Specify whether to generate a traceability report that has hyperlinks for navigating from code-to-model and from model-to-code. For more information, see “Generate traceability report”.

#### **ResourceReport — Resource utilization report generation**

'off' (default) | 'on'

Specify whether to generate a resource utilization report that displays the number of hardware resources that the generated HDL code uses. For more information, see “Generate resource utilization report”.

#### **OptimizationReport — Optimization report generation**

'off' (default) | 'on'

Specify whether to generate an optimization report that displays the effect of optimizations such as streaming, sharing, and distributed pipelining. For more information, see “Generate optimization report”.

### **HDLGenerateWebview — Include model Web view**

'on' (default) | 'off'

Specify whether to generate a web view of the model in the Code Generation report to easily navigate between the code and model. For more information, see “Generate model Web view”.

### **Speed and Area Optimization**

#### **BalanceDelays — Delay balancing**

'on' (default) | 'off'

Specify whether to enable delay balancing on the model. For more information, see “Balance delays”.

#### **DistributedPipeliningPriority — Specify priority for distributed pipelining algorithm**

'NumericalIntegrity' (default) | 'Performance'

Specify whether to prioritize the distributed pipelining optimization for numerical integrity or performance. For more information, see the **Distributed pipelining priority** section in “Distributed Pipelining”.

#### **HierarchicalDistPipelining — Hierarchical distributed pipelining**

'off' (default) | 'on'

Specify whether to apply the hierarchical distributed pipelining optimization on the model. For more information, see “Distributed Pipelining”.

#### **PreserveDesignDelays — Prevent distributed pipelining from moving design delays**

'off' (default) | 'on'

Specify whether you want the code generator to distribute design delays in your model. For more information, see “Preserve design delays”.

#### **ClockRatePipelining — Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths**

'on' (default) | 'off'



Specify whether to insert pipeline registers at the clock rate or the data rate. For more information, see “Clock Rate Pipelining”.

### **MinimizeClockEnables — Omit clock enable logic for single-rate designs**

'off' (default) | 'on'

For more information, see “Minimize Clock Enables and Reset Signals”.

### **RAMMappingThreshold — Minimum RAM size for mapping to RAMs instead of registers**

256 (default) | positive integer

Specify, in bits, the minimum RAM size required for mapping to RAMs instead of registers. For more information, see the **RAM mapping threshold (bits)** section in “RAM Mapping”.

### **MapPipelineDelaysToRAM — Map pipeline registers in the generated HDL code to RAM**

'off' (default) | 'on'

Specify whether to map pipeline registers in the generated HDL code to block RAMs on the FPGA. For more information, see the **Map pipeline delays to RAM** section in “RAM Mapping”.

### **HighlightFeedbackLoops — Highlight feedback loops inhibiting delay balancing and optimizations**

'off' (default) | 'on'

Specify whether to highlight feedback loops in your design. For more information, see “Diagnostics for Optimizations”.

### **Coding Style**

#### **UserComment — HDL file header comment**

character vector

Specify comment lines in header of generated HDL and test bench files. For more information, see “Comment in header”.

#### **UseAggregatesForConst — Represent constant values with aggregates**

'off' (default) | 'on'

For more information, see **Represent constant values by aggregates** in “RTL Customizations for Constants and MATLAB Function Blocks”.

### **UseRisingEdge — Use VHDL rising\_edge or falling\_edge function to detect clock transitions**

'off' (default) | 'on'

For more information, see **Use "rising\_edge/falling\_edge" style for registers** in “RTL Style”.

### **LoopUnrolling — Unroll VHDL FOR and GENERATE loops**

'off' (default) | 'on'

For more information, see **Loop unrolling** in “RTL Style”.

### **UseVerilogTimescale — Generate 'timescale compiler directives**

'on' (default) | 'off'

For more information, see **Use Verilog 'timescale directives** in “RTL Annotations”.

### **InlineConfigurations — Include VHDL configurations**

'on' (default) | 'off'

For more information, see **Inline VHDL configuration** in “RTL Annotations”.

### **SafeZeroConcat — Type-safe syntax for concatenated zeros**

'on' (default) | 'off'

For more information, see **Concatenate type safe zeros** in “RTL Annotations”.

### **DateComment — Include time stamp in header**

'on' (default) | 'off'

For more information, see **Emit time/date stamp in header** in “RTL Annotations”.

### **ScalarizePorts — Flatten vector ports into scalar ports**

'off' (default) | 'on'

For more information, see **Scalarize vector ports** in “RTL Style”.

### **MinimizeIntermediateSignals — Minimize intermediate signals**

'off' (default) | 'on'

For more information, see **Minimize intermediate signals** in “RTL Style”.

### **RequirementComments — Link from code generation reports to requirement documents**

'on' (default) | 'off'

For more information, see **Include requirements in block comments** in “RTL Annotations”.

### **InLineMATLABBlockCode — Inline HDL code for MATLAB Function blocks**

'off' (default) | 'on'

For more information, see **Inline MATLAB Function block code** “RTL Customizations for Constants and MATLAB Function Blocks”.

### **MaskParameterAsGeneric — Reusable code generation for subsystems with identical mask parameters**

'off' (default) | 'on'

For more information, see **Generate parameterized HDL code from masked subsystem** in “RTL Style”.

### **InitializeBlockRAM — Initial signal value generation for RAM blocks**

'on' (default) | 'off'

For more information, see **Initialize all RAM blocks** in “RTL Customizations for RAMs”.

### **RAMArchitecture — RAM architecture**

'WithClockEnable' (default) | 'WithoutClockEnable'

For more information, see **RAM Architecture** in “RTL Customizations for RAMs”.

## **Clocks and Reset**

### **ClockEdge — Active clock edge**

'Rising' (default) | 'Falling'

Specify the active clock edge for the generated HDL code. For more information, see the **Clock edge** section in “Clock Settings and Timing Controller Postfix”.

### **ClockInputs — Single or multiple clock inputs**

'Single' (default) | 'Multiple'

Specify whether to generate single or multiple clock inputs in the HDL code. For more information, see the **Clock inputs** section in “Clock Settings and Timing Controller Postfix”.

### **Oversampling — Oversampling factor for global clock**

1 (default) | integer greater than or equal to 0

Frequency of global oversampling clock, specified as an integer multiple of the model’s base rate. For more information, see “Oversampling factor”.

### **ResetAssertedLevel — Asserted (active) level of reset**

'active-high' (default) | 'active-low'

Specify whether to use an active-high or active-low asserted level for the reset input signal. For more information, see the **Reset asserted level** section in “Reset Settings”.

### **ResetType — Reset type**

'async' (default) | 'sync'

Specify whether to use synchronous or asynchronous reset in the generated HDL code. For more information, see “Reset Settings”.

### **TriggerAsClock — Use trigger signal as clock in triggered subsystems**

'off' (default) | 'on'

For more information, see “Use trigger signal as clock”.

### **TimingControllerArch — Generate reset for timing controller**

'default' (default) | 'resettable'

For more information, see **Timing controller architecture** in “Timing Controller Settings”.

### **Test Bench**

#### **GenerateCoSimBlock — Generate HDL Cosimulation block**

'off' (default) | 'on'

When you use this property with `makehdl`, HDL Coder does not a Cosimulation block. To generate a Cosimulation block, use `makehdltb`. With the Cosimulation block, you can simulate the DUT in Simulink with an HDL simulator.

For more information, see `GenerateCoSimBlock`.

**GenerateCoSimModel — Generate HDL Cosimulation Model**`'ModelSim' (default) | 'Incisive' | 'None'`

When you use this property with makehdl, HDL Coder does not a Cosimulation model. To generate a Cosimulation model, use makehdl**tb**. The model contains a Cosimulation block for the HDL simulator that you specify.

For more information, see `GenerateCoSimModel`.

**SimulatorFlags — Options for generated compilation scripts**`character vector`

For more information, see `SimulatorFlags`.

**TestBenchReferencePostFix — Suffix for test bench reference signals**`'_ref' (default) | character vector`

For more information, see `TestBenchReferencePostFix`.

**Script Generation****EDAScriptGeneration — Enable or disable script generation for third-party tools**`'on' (default) | 'off'`

For more information, see `EDAScriptGeneration`.

**HDLCompileInit — Compilation script initialization text**`'vlib %s\n' (default) | character vector`

For more information, see `HDLCompileInit`.

**HDLCompileTerm — Compilation script termination text**`' ' (default) | character vector`

For more information, see `HDLCompileTerm`.

**HDLCompileFilePostfix — Postfix for compilation script file name**`'_compile.do' (default) | character vector`

For more information, see `HDLCompileFilePostfix`.

**HDLCompileVerilogCmd — Verilog compilation command**`'vlog %s %s\n' (default) | character vector`

Verilog compilation command, specified as a character vector. The `SimulatorFlags` name-value pair specifies the first argument, and the module name specifies the second argument.

For more information, see `HDLCompileVerilogCmd`.

### **HDLCompileVHDLCmd — VHDL compilation command**

`'vcom %s %s\n'` (default) | character vector

VHDL compilation command, specified as a character vector. The `SimulatorFlags` name-value pair specifies the first argument, and the entity name specifies the second argument.

For more information, see `HDLCompileVHDLCmd`.

### **HDLLintTool — HDL lint tool**

`'None'` (default) | `'AscentLint'` | `'Leda'` | `'SpyGlass'` | `'Custom'`

For more information, see `HDLLintTool`.

### **HDLLintInit — HDL lint initialization name**

character vector

HDL lint initialization name, specified as a character vector. The default is derived from the `HDLLintTool` name-value pair.

For more information, see `HDLLintInit`.

### **HDLLintCmd — HDL lint command**

character vector

HDL lint command, specified as a character vector. The default is derived from the `HDLLintTool` name-value pair.

For more information, see `HDLLintCmd`.

### **HDLLintTerm — HDL lint termination name**

character vector

HDL lint termination, specified as a character vector. The default is derived from the `HDLLintTool` name-value pair.

For more information, see `HDLLintTerm`.

**HDLSynthTool — Synthesis tool**

'None' (default) | 'ISE' | 'Libero' | 'Precision' | 'Quartus' | 'Synplify' | 'Vivado' | 'Custom'

For more information, see HDLSynthTool.

**HDLSynthCmd — HDL synthesis command**

character vector

HDL synthesis command, specified as a character vector. The default is derived from the HDLSynthTool name-value pair.

For more information, see HDLSynthCmd.

**HDLSynthFilePostfix — Postfix for synthesis script file name**

character vector

HDL synthesis script file name postfix, specified as a character vector. The default is derived from the HDLSynthTool name-value pair.

For more information, see HDLSynthFilePostfix.

**HDLSynthInit — Synthesis script initialization name**

character vector

Initialization for the HDL synthesis script, specified as a character vector. The default is derived from the HDLSynthTool name-value pair.

For more information, see HDLSynthInit.

**HDLSynthTerm — Synthesis script termination name**

character vector

Termination name for the HDL synthesis script. The default is derived from the HDLSynthTool name-value pair.

For more information, see HDLSynthTerm.

**Generated Model****GeneratedModelNamePrefix — Prefix for generated model name**

'gm\_' (default) | character vector

For more information, see “Prefix for generated model name”.

### **Synthesis**

#### **SynthesisTool — Synthesis tool**

' ' (default) | 'Altera Quartus II' | 'Xilinx ISE' | 'Xilinx Vivado'

Specify the synthesis tool for targeting the generated HDL code as a character vector. For more information, see “Tool and Device”.

#### **SynthesisToolChipFamily — Synthesis tool chip family**

' ' (default) | character vector

Specify the synthesis tool chip family for the target device as a character vector. For more information, see the **Family** section in “Tool and Device”.

#### **SynthesisToolDeviceName — Synthesis tool device name**

' ' (default) | character vector

Specify the synthesis tool device name for the target device as a character vector. For more information, see the **Device** section in “Tool and Device”.

#### **SynthesisToolPackageName — Synthesis tool package name**

' ' (default) | character vector

Specify the synthesis tool package name for the target device as a character vector. For more information, see the **Package** section in “Tool and Device”.

#### **SynthesisToolSpeedValue — Synthesis tool speed value**

' ' (default) | character vector

Specify the synthesis tool speed value for the target device as a character vector. For more information, see the **Speed** section in “Tool and Device”.

#### **SynthesisToolSpeedValue — Synthesis tool speed value**

' ' (default) | character vector

Specify the synthesis tool speed value for the target device as a character vector. For more information, see the **Speed** section in “Tool and Device”.

#### **TargetFrequency — Target frequency in MHz**

' ' (default) | character vector



Specify the target frequency in MHz as a character vector. For more information, see “Target Frequency”.

### **MulticyclePathInfo — Multicycle path constraint file generation**

'off' (default) | 'on'

Specify whether to generate a multicycle path constraints text file. For more information, see “Multicycle Path Constraints”.

### **MulticyclePathConstraints — Enable-based multicycle path constraint file generation**

'off' (default) | 'on'

Specify whether to generate an enable-based multicycle path constraints file. For more information, see **Enable-based constraints** in “Multicycle Path Constraints”.

### **Port Names and Types**

#### **ClockEnableInputPort — Clock enable input port name**

'clk\_enable' (default) | character vector

Specify the clock enable input port name as a character vector. For more information, see “Clock Enable Settings”.

#### **ClockEnableOutputPort — Clock enable output port name**

'ce\_out' (default) | character vector

Clock enable output port name, specified as a character vector.

For more information, see “Clock Enable output port”.

#### **ClockInputPort — Clock input port name**

'clk' (default) | character vector

Specify the clock input port name as a character vector. For more information, see “Clock Settings and Timing Controller Postfix”.

#### **InputType — HDL data type for input ports**

'wire' or 'std\_logic\_vector' (default) | 'signed/unsigned'

VHDL inputs can have 'std\_logic\_vector' or 'signed/unsigned' data type. Verilog inputs must be 'wire'.

For more information, see **Input data type** in “Input and Output Port Data Types”.

### **OutputType — HDL data type for output ports**

'Same as input data type' (default) | 'std\_logic\_vector' | 'signed/unsigned' | 'wire'

VHDL output can be 'Same as input data type', 'std\_logic\_vector' or 'signed/unsigned'. Verilog output must be 'wire'.

For more information, see **Output data type** in “Input and Output Port Data Types”.

### **ResetInputPort — Reset input port name**

'reset' (default) | character vector

Reset input port name, specified as a character vector.

For more information, see the **Reset input port** section in “Reset Settings”.

### **File and Variable Names**

#### **VerilogFileExtension — Verilog file extension**

' .v ' (default) | character vector

Specify the file name extension for generated Verilog files. For more information, see “Language-Specific Identifiers and File Extensions”.

#### **VHDLFileExtension — VHDL file extension**

' .vhd ' (default) | character vector

Specify the file name extension for generated VHDL files. For more information, see the **VHDLFileExtension** section in “Language-Specific Identifiers and File Extensions”.

#### **VHDLArchitectureName — VHDL architecture name**

' rtl ' (default) | character vector

For more information, see **VHDL architecture name** in “VHDL Architecture and Library Name”.

#### **VHDLLibraryName — VHDL library name**

' work ' (default) | character vector

For more information, see **VHDL library name** in “VHDL Architecture and Library Name”.

**SplitEntityFilePostfix — Postfix for VHDL entity file names**

'\_entity' (default) | character vector

For more information, see **Split entity file postfix** in “Split entity and architecture”.

**SplitArchFilePostfix — Postfix for VHDL architecture file names**

'\_arch' (default) | character vector

For more information, see **Split arch file postfix** in “Split entity and architecture”.

**PackagePostfix — Postfix for package file name**

'\_pkg' (default) | character vector

Specify the postfix for the package file name as a character vector. For more information, see the **Package Postfix** section in “Language-Specific Identifiers and File Extensions”.

**HDLMapFilePostfix — Postfix for mapping file**

'\_map.txt' (default) | character vector

For more information, see “Map file postfix”.

**BlockGenerateLabel — Block label postfix for VHDL GENERATE statements**

'\_gen' (default) | character vector

For more information, see **Block generate label** in “Generate Statement Labels”.

**ClockProcessPostfix — Postfix for clock process names**

'\_process' (default) | character vector

Specify the postfix for clocked process names as a character vector. For more information, see the **Clocked process postfix** section in “Clock Settings and Timing Controller Postfix”.

**ComplexImagPostfix — Postfix for imaginary part of complex signal**

'\_im' (default) | character vector

For more information, see **Complex imaginary part postfix** in “Complex Signals Postfix”.

**ComplexRealPostfix — Postfix for imaginary part of complex signal names**

'\_re' (default) | character vector

For more information, see **Complex real part postfix** in “Complex Signals Postfix”.

### **EntityConflictPostfix** — Postfix for duplicate VHDL entity or Verilog module names

'\_block' (default) | character vector

Specify the postfix as a character vector that resolves duplicate entity or module names. For more information, see the **Entity conflict postfix** section in “Language-Specific Identifiers and File Extensions”.

### **InstanceGenerateLabel** — Instance section label postfix for VHDL GENERATE statements

'\_gen' (default) | character vector

For more information, see **Instance generate label** in “Generate Statement Labels”.

### **InstancePostfix** — Postfix for generated component instance names

' ' (default) | character vector

For more information, see **Instance postfix** in “Vector and Component Instances Labels”.

### **InstancePrefix** — Prefix for generated component instance names

'u\_' (default) | character vector

For more information, see **Instance prefix** in “Vector and Component Instances Labels”..

### **OutputGenerateLabel** — Output assignment label postfix for VHDL GENERATE statements

'outputgen' (default) | character vector

For more information, see **Output generate label** in “Generate Statement Labels”.

### **PipelinePostfix** — Postfix for input and output pipeline register names

'\_pipe' (default) | character vector

For more information, see “Pipeline postfix”.

### **ReservedWordPostfix** — Postfix for names conflicting with VHDL or Verilog reserved words

'\_rsvd' (default) | character vector

For more information, see **Reserved word postfix** in “Language-Specific Identifiers and File Extensions”.

**TimingControllerPostfix — Postfix for timing controller name**

'\_tc' (default) | character vector

For more information, see **Timing controller postfix** in “Clock Settings and Timing Controller Postfix”.

**VectorPrefix — Prefix for vector names**

'vector\_of\_' (default) | character vector

For more information, see **Vector prefix** in “Vector and Component Instances Labels”.

**EnablePrefix — Prefix for internal enable signals**

'enb' (default) | character vector

Prefix for internal clock enable and control flow enable signals, specified as a character vector. For more information, see “Clock Enable Settings”.

**ModulePrefix — Prefix for modules or entity names**

' ' (default) | character vector

Specify a prefix for every module or entity name in the generated HDL code. HDL Coder also applies this prefix to generated script file names

For more information, see **ModulePrefix** in “Language-Specific Identifiers and File Extensions”.

**See Also**

checkhdl | makehdltb

**Introduced in R2006b**

# makehdltb

Generate HDL test bench from model or subsystem

## Syntax

```
makehdltb(dut)  
makehdltb(dut,Name,Value)
```

## Description

`makehdltb(dut)` generates an HDL test bench from the specified subsystem or model reference.

---

**Note** If you have not previously executed `makehdl` within the current MATLAB session, `makehdltb` calls `makehdl` to generate model code before generating the test bench code. Properties passed in to `makehdl` persist after `makehdl` executes, and (unless explicitly overridden) are passed to subsequent `makehdl` calls during the same MATLAB session.

---

`makehdltb(dut,Name,Value)` generates an HDL test bench from the specified subsystem or model reference with options specified by one or more name-value pair arguments.

## Examples

### Generate VHDL Test Bench

Generate VHDL DUT and test bench for a subsystem.

Use `makehdl` to generate VHDL code for the subsystem `symmetric_fir`.

```
makehdl('sfir_fixed/symmetric_fir')
```

```

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings,
and 0 messages.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as
   hdlsrc\sfir_fixed\symmetric_fir.vhd
### HDL code generation complete.

```

After makehdl is complete, use makehdltb to generate a VHDL test bench for the same subsystem.

```
makehdltb('sfir_fixed/symmetric_fir')
```

```

### Begin TestBench generation.
### Generating HDL TestBench for 'sfir_fixed/symmetric_fir'.
### Begin simulation of the model 'gm_sfir_fixed'...
### Collecting data...
### Generating test bench: hdlsrc\sfir_fixed\symmetric_fir_tb.vhd
### Creating stimulus vectors...
### HDL TestBench generation complete.

```

The generated VHDL test bench code is saved in the hdlsrc folder.

## Generate Verilog Test Bench

Generate Verilog DUT and test bench for a subsystem.

Use makehdl to generate Verilog code for the subsystem symmetric\_fir.

```
makehdl('sfir_fixed/symmetric_fir', 'TargetLanguage', 'Verilog')
```

```

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings,
and 0 messages.
### Begin Verilog Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as
   hdlsrc\sfir_fixed\symmetric_fir.v
### HDL code generation complete.

```

After makehdl is complete, use makehdltb to generate a Verilog test bench for the same subsystem.

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog')  
  
### Begin TestBench generation.  
### Generating HDL TestBench for 'sfir_fixed/symmetric_fir'.  
### Begin simulation of the model 'gm_sfir_fixed'...  
### Collecting data...  
### Generating test bench:hdlsrc\sfir_fixed\symmetric_fir_tb.v  
### Creating stimulus vectors...  
### HDL TestBench generation complete.
```

The generated Verilog test bench code is saved in the `hdlsrc\sfir_fixed` folder.

### Generate a SystemVerilog DPI Test Bench

Generate SystemVerilog DPI test bench for a subsystem.

Consider this option if generation or simulation of the default HDL test bench takes a long time. Generation of a DPI test bench can be faster than the default version because it does not run a Simulink simulation to create the test bench data. Simulation of a DPI test bench with a large data set is faster than the default version because it does not store the input or expected data in a separate file. For requirements to use this feature, see the `GenerateSVDPI TestBench` property.

Use `makehdl` to generate Verilog code for the subsystem `symmetric_fir`.

```
makehdl('sfir_fixed/symmetric_fir','TargetLanguage','Verilog')  
  
### Generating HDL for 'sfir_fixed/symmetric_fir'.  
### Starting HDL check.  
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings,  
    and 0 messages.  
### Begin Verilog Code Generation for 'sfir_fixed'.  
### Working on sfir_fixed/symmetric_fir as  
    hdlsrc\sfir_fixed\symmetric_fir.v  
### HDL code generation complete.
```

After the code is generated, use `makehdltb` to generate a test bench for the same subsystem. Specify your HDL simulator so that the coder can generate scripts to build and run the generated SystemVerilog and C code. Disable generation of the default test bench.

```
makehdltb('sfir_fixed/symmetric_fir','TargetLanguage','Verilog',...  
    'GenerateSVDPI TestBench','ModelSim','GenerateHDLTestBench','off')
```



```
### Start checking model compatibility with SystemVerilog DPI testbench
### Finished checking model compatibility with SystemVerilog DPI testbench
### Preparing generated model for SystemVerilog DPI component generation
### Generating SystemVerilog DPI component
### Starting build procedure for model: gm_sfir_fixed_ref
### Starting SystemVerilog DPI Component Generation
### Generating DPI H Wrapper gm_sfir_fixed_ref_dpi.h
### Generating DPI C Wrapper gm_sfir_fixed_ref_dpi.c
### Generating SystemVerilog module gm_sfir_fixed_ref_dpi.sv using template C:\matlab\
### Generating makefiles for: gm_sfir_fixed_ref_dpi
### Invoking make to build the DPI Shared Library
### Successful completion of build procedure for model: gm_sfir_fixed_ref
### Working on symmetric_fir_dpi_tb as hdlsrc\sfir_fixed\symmetric_fir_dpi_tb.sv.
### Generating SystemVerilog DPI testbench simulation script for ModelSim/QuestaSim hdl

### HDL TestBench generation complete.
```

The generated SystemVerilog and C test bench files, and the build scripts, are saved in the `hdlsrc\sfir_fixed` folder.

## Input Arguments

### **dut** — DUT subsystem or model reference name

character vector

DUT subsystem or model reference name, specified as a character vector, with full hierarchical path.

Example: `'modelName/subsysTarget'`

Example: `'modelName/subsysA/subsysB/subsysTarget'`

### **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'TargetLanguage', 'Verilog'`

### Basic Options

#### TargetLanguage — Target language

'VHDL' (default) | 'Verilog'

Specify whether to generate VHDL or Verilog code. For more information, see the **Language** section in “Target”.

#### TargetDirectory — Output directory

'hdlsrc' (default) | character vector

Specify a path to write the generated files and HDL code into. For more information, see the **Folder** section in “Target”.

#### SplitEntityArch — Split VHDL entity and architecture into separate files

'off' (default) | 'on'

For more information, see **Split entity and architecture** in “Split entity and architecture”.

### Test Bench Generation

#### GenerateHDLTestBench — Generate HDL test bench

'on' (default) | 'off'

The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT. For more information, see **HDL test bench** “Test Bench Generation Output”.

#### GenerateSVPITestBench — Generate SystemVerilog DPI test bench

'none' (default) | 'ModelSim' | 'Incisive' | 'VCS' | 'Vivado Simulator'

When you set this property, the coder generates a direct programming interface (DPI) component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder™. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

Consider using this option if the default HDL test bench takes a long time to generate or simulate. Generation of a DPI test bench is sometimes faster than the default version because it does not run a full Simulink simulation to create the test bench data.

Simulation of a DPI test bench with a large data set is faster than the default version because it does not store the input or expected data in a separate file. For an example, see “Generate a SystemVerilog DPI Test Bench” on page 2-116.

To use this feature, you must have HDL Verifier™ and Simulink Coder licenses. To run the SystemVerilog testbench with generated VHDL code, you must have a mixed-language simulation license for your HDL simulator.

---

**Limitations** This test bench is not supported when you generate HDL code for the top-level Simulink model. Your DUT subsystem must meet the following conditions:

- Input and output data types of the DUT cannot be larger than 64 bits.
  - Input and output ports of the DUT cannot use enumerated data types.
  - Input and output ports cannot be single-precision or double-precision data types.
  - The DUT cannot have multiple clocks. You must set the **Clock inputs** code generation option to `Single`.
  - **Use trigger signal as clock** must not be selected.
  - If the DUT uses vector ports, you must use **Scalarize vector ports** to flatten the interface.
- 

### **GenerateCoSimBlock — Generate HDL Cosimulation block**

'off' (default) | 'on'

Generate an HDL Cosimulation block so you can simulate the DUT in Simulink with an HDL simulator.

For more information, see `GenerateCoSimBlock`.

### **GenerateCoSimModel — Generate HDL Cosimulation model**

'ModelSim' (default) | 'Incisive' | 'None'

Generate a model containing an HDL Cosimulation block for the specified HDL simulator.

For more information, see `GenerateCoSimModel`.

### **HDLCodeCoverage — Enable code coverage on the generated test bench**

'off' (default) | 'on'

Include code coverage switches in the generated build-and-run scripts. These switches turn on code coverage for the generated test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

### **SimulationTool — HDL simulator where you will run the generated test bench**

'ModelSim' (default) | 'Incisive' | 'VCS' | 'Vivado' | 'Custom'

This property applies to generated test benches. 'VCS' and 'Vivado' are supported only for SystemVerilog DPI test benches. When you select 'Custom', the tool uses the custom script settings. See the “Script Generation” properties.

### **Test Bench Configuration**

#### **ForceClock — Force clock input**

'on' (default) | 'off'

Specify that the generated test bench drives the clock enable input based on `ClockLowTime` and `ClockHighTime`.

For more information, see `ForceClock`.

#### **ClockHighTime — Clock high time**

5 (default) | positive integer

Clock high time during a clock period, specified in nanoseconds.

For more information, see `ClockHighTime`.

#### **ClockLowTime — Clock low time**

5 (default) | positive integer

Clock low time during a clock period, specified in nanoseconds.

For more information, see `ClockLowTime`.

#### **ForceClockEnable — Force clock enable input**

'on' (default) | 'off'

Specify that the generated test bench drives the clock enable input.

For more information, see `ForceClockEnable`.

**ClockInputs — Single or multiple clock inputs**`'Single' (default) | 'Multiple'`

Specify whether to generate single or multiple clock inputs in the HDL code. For more information, see the **Clock inputs** section in “Clock Settings and Timing Controller Postfix”.

**ForceReset — Force reset input**`'on' (default) | 'off'`

Specify that the generated test bench drives the reset input.

For more information, see ForceReset.

**ResetLength — Reset asserted time length**`2 (default) | integer greater than or equal to 0`

Length of time that reset is asserted, specified as the number of clock cycles.

For more information, see .

**ResetAssertedLevel — Asserted (active) level of reset**`'active-high' (default) | 'active-low'`

Specify whether to use an active-high or active-low asserted level for the reset input signal. For more information, see the **Reset asserted level** section in “Reset Settings”.

**HoldInputDataBetweenSamples — Hold valid data for signals clocked at slower rate**`'on' (default) | 'off'`

For more information, see HoldInputDataBetweenSamples.

**HoldTime — Hold time for inputs and forced reset**`2 (default) | positive integer`

Hold time for inputs and forced reset, specified in nanoseconds.

For more information, see HoldTime.

**IgnoreDataChecking — Time to wait after clock enable before checking output data**`0 (default) | positive integer`

Time after clock enable is asserted before starting output data checks, specified in number of samples.

For more information, see `IgnoreDataChecking`.

### **InitializeTestBenchInputs — Initialize test bench inputs to 0**

'off' (default) | 'on'

For more information, see `InitializeTestBenchInputs`.

### **MultifileTestBench — Divide generated test bench into helper functions, data, and HDL test bench files**

'off' (default) | 'on'

For more information, see `MultifileTestBench`.

### **UseFileI0InTestBench — Use file I/O to read/write test bench data**

'on' (default) | 'off'

For more information, see `UseFileI0InTestBench`.

### **TestBenchClockEnableDelay — Number of clock cycles between deassertion of reset and assertion of clock enable**

1 (default) | positive integer

For more information, see `TestBenchClockEnableDelay`.

### **TestBenchDataPostFix — Postfix for test bench data file name**

'\_data' (default) | character vector

For more information, see `TestBenchDataPostFix`.

### **TestBenchPostFix — Suffix for test bench name**

'\_tb' (default) | character vector

For more information, see `TestBenchPostFix`.

### **Coding Style**

### **UseVerilogTimescale — Generate 'timescale compiler directives**

'on' (default) | 'off'

For more information, see **Use Verilog 'timescale directives** in “RTL Annotations”.

**DateComment — Include time stamp in header**`'on'` (default) | `'off'`

For more information, see **Emit time/date stamp in header** in “RTL Annotations”.

**InlineConfigurations — Include VHDL configurations**`'on'` (default) | `'off'`

For more information, see **Inline VHDL configuration** in “RTL Annotations”.

**ScalarizePorts — Flatten vector ports into scalar ports**`'off'` (default) | `'on'`

For more information, see **Scalarize vector ports** in “RTL Style”.

**Script Generation****HDLCompileInit — Compilation script initialization text**`'vlib %s\n'` (default) | character vector

For more information, see HDLCompileInit.

**HDLCompileTerm — Compilation script termination text**`' '` (default) | character vector

For more information, see HDLCompileTerm.

**HDLCompileFilePostfix — Postfix for compilation script file name**`'_compile.do'` (default) | character vector

For more information, see HDLCompileFilePostfix.

**HDLCompileVerilogCmd — Verilog compilation command**`'vlog %s %s\n'` (default) | character vector

Verilog compilation command, specified as a character vector. The SimulatorFlags name-value pair specifies the first argument, and the module name specifies the second argument.

For more information, see HDLCompileVerilogCmd.

**HDLCompileVHDLCmd — VHDL compilation command**`'vcom %s %s\n'` (default) | character vector

VHDL compilation command, specified as a character vector. The `SimulatorFlags` name-value pair specifies the first argument, and the entity name specifies the second argument.

For more information, see `HDLCompileVHDLCmd`.

### **HDLSimCmd — HDL simulation command**

`'vsim -novopt %s.%s\n'` (default) | character vector

The HDL simulation command, specified as a character vector.

For more information, see `HDLSimCmd`.

### **HDLSimInit — HDL simulation script initialization name**

`['onbreak resume\n', 'onerror resume\n']` (default) | character vector

Initialization for the HDL simulation script, specified as a character vector.

For more information, see `HDLSimInit`.

### **HDLSimTerm — HDL simulation script termination name**

`'run -all'` (default) | character vector

The termination name for the HDL simulation command, specified as a character vector.

For more information, see `HDLSimTerm`.

### **HDLSimFilePostfix — Postscript for HDL simulation script**

`'_sim.do'` (default) | character vector

For more information, see `HDLSimFilePostfix`.

### **HDLSimViewWaveCmd — HDL simulation waveform viewing command**

`'add wave sim:%s\n'` (default) | character vector

Waveform viewing command, specified as a character vector. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.

For more information, see `HDLSimViewWaveCmd`.

## **Port Names and Types**

### **ClockEnableInputPort — Clock enable input port name**

`'clk_enable'` (default) | character vector



Specify the clock enable input port name as a character vector. For more information, see “Clock Enable Settings”.

**ClockEnableOutputPort — Clock enable output port name**

'ce\_out' (default) | character vector

Clock enable output port name, specified as a character vector.

For more information, see “Clock Enable output port”.

**ClockInputPort — Clock input port name**

'clk' (default) | character vector

Specify the clock input port name as a character vector. For more information, see “Clock Settings and Timing Controller Postfix”.

**ResetInputPort — Reset input port name**

'reset' (default) | character vector

Reset input port name, specified as a character vector.

For more information, see the **Reset input port** section in “Reset Settings”.

**File and Variable Names****VerilogFileExtension — Verilog file extension**

'.v' (default) | character vector

Specify the file name extension for generated Verilog files. For more information, see “Language-Specific Identifiers and File Extensions”.

**VHDLFileExtension — VHDL file extension**

'.vhd' (default) | character vector

Specify the file name extension for generated VHDL files. For more information, see the **VHDLFileExtension** section in “Language-Specific Identifiers and File Extensions”.

**VHDLArchitectureName — VHDL architecture name**

'rtl' (default) | character vector

For more information, see **VHDL architecture name** in “VHDL Architecture and Library Name”.

### **VHDLLibraryName — VHDL library name**

'work' (default) | character vector

For more information, see **VHDL library name** in “VHDL Architecture and Library Name”.

### **SplitEntityFilePostfix — Postfix for VHDL entity file names**

'\_entity' (default) | character vector

For more information, see **Split entity file postfix** in “Split entity and architecture”.

### **SplitArchFilePostfix — Postfix for VHDL architecture file names**

'\_arch' (default) | character vector

For more information, see **Split arch file postfix** in “Split entity and architecture”.

### **PackagePostfix — Postfix for package file name**

'\_pkg' (default) | character vector

Specify the postfix for the package file name as a character vector. For more information, see the **Package Postfix** section in “Language-Specific Identifiers and File Extensions”.

### **ComplexImagPostfix — Postfix for imaginary part of complex signal**

'\_im' (default) | character vector

For more information, see **Complex imaginary part postfix** in “Complex Signals Postfix”.

### **ComplexRealPostfix — Postfix for imaginary part of complex signal names**

'\_re' (default) | character vector

For more information, see **Complex real part postfix** in “Complex Signals Postfix”.

### **EnablePrefix — Prefix for internal enable signals**

'enb' (default) | character vector

Prefix for internal clock enable and control flow enable signals, specified as a character vector. For more information, see “Clock Enable Settings”.

## **See Also**

makehdl

**Introduced in R2006b**

## Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

### Syntax

```
Simulink.ModelReference.protect(model)
Simulink.ModelReference.protect(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.protect(model, '
Harness',true)
[~,neededVars] = Simulink.ModelReference.protect(model)
```

### Description

`Simulink.ModelReference.protect(model)` creates a protected model from the specified `model`. It places the protected model in the current working folder. The protected model has the same name as the source model. It has the extension `.slxp`.

`Simulink.ModelReference.protect(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[harnessHandle] = Simulink.ModelReference.protect(model, 'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~,neededVars] = Simulink.ModelReference.protect(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

### Examples

#### Protect Referenced Model

Protect a referenced model and place the protected model in the current working folder.

```
sldemo_mdref_bus;
model= 'sldemo_mdref_counter_bus'
```

```
Simulink.ModelReference.protect(model);
```

A protected model named `sldemo_mdref_counter_bus.slxp` is created. The protected model file is placed in the current working folder.

### Place Protected Model in Specified Folder

Protect a referenced model and place the protected model in a specified folder.

```
sldemo_mdref_bus;
model= 'sldemo_mdref_counter_bus'
```

```
Simulink.ModelReference.protect(model, 'Path', 'C:\Work');
```

A protected model named `sldemo_mdref_counter_bus.slxp` is created. The protected model file is placed in `C:\Work`.

### Generate Code for Protected Model

Protect a referenced model, generate code for it in normal mode, and obfuscate the code.

```
sldemo_mdref_bus;
model= 'sldemo_mdref_counter_bus'
```

```
Simulink.ModelReference.protect(model, 'Path', 'C:\Work', 'Mode', 'CodeGeneration', ...
'ObfuscateCode', true);
```

A protected model named `sldemo_mdref_counter_bus.slxp` is created. The protected model file is placed in the `C:\Work` folder. The protected model runs as a child of the parent model. The code generated for the protected model is obfuscated by the software.

### Generate HDL Code for Protected Model

Protect a referenced model, and generate HDL code for it in normal mode.

```
parent_model= 'hdlcoder_protected_model_parent_harness';
```

```
reference_model_to_protect = 'hdlcoder_referenced_model_gain';  
Simulink.ModelReference.protect(reference_model_to_protect, ...  
    'Mode', 'HDLCodeGeneration')
```

A protected model named `hdlcoder_referenced_model_gain.slxp` is created. The protected model file is placed in the same folder as the parent model and the referenced model. The protected model runs as a child of the parent model.

Set the **hdl** option to **true** with **Mode** set to **CodeGeneration** to enable both C code generation and HDL code generation support for a protected model that you create.

```
parent_model= 'hdlcoder_protected_model_parent_harness';  
reference_model_to_protect = 'hdlcoder_referenced_model_gain';  
Simulink.ModelReference.protect(reference_model_to_protect, ...  
    'Mode', 'CodeGeneration', 'hdl', true)
```

### Control Code Visibility for Protected Model

Control code visibility by allowing users to view only binary files and headers in the code generated for a protected model.

```
sldemo_mdhref_bus;  
model= 'sldemo_mdhref_counter_bus'  
  
Simulink.ModelReference.protect(model, 'Mode', 'CodeGeneration', 'OutputFormat', ...  
    'CompiledBinaries');
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created. The protected model file is placed in the current working folder. Users can view only binary files and headers in the code generated for the protected model.

### Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
sldemo_mdhref_bus;  
modelPath= 'sldemo_mdhref_bus/CounterA'  
  
[harnessHandle] = Simulink.ModelReference.protect(modelPath, 'Path', 'C:\Work', ...  
    'Harness', true, 'Report', true);
```

A protected model named `sldemo_mdhref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The

folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

## Input Arguments

### **model** — Model name

string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the model to be protected.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
'Mode', 'CodeGeneration', 'OutputFormat', 'Binaries', 'ObfuscateCode', true
```

ue specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

### **Harness** — Option to create a harness model

false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness', true`

### **Mode** — Model protection mode

'Normal' (default) | 'Accelerator' | 'CodeGeneration' | 'HDLCodeGeneration' | 'ViewOnly'

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.

- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation. (Requires HDL Coder license)
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode', 'Accelerator'`

### **CodeInterface — Interface through which generated code is accessed by Model block**

`'Model reference'` (default) | `'Top model'`

Applies only if the system target file (`SystemTargetFile`) is set to an ERT-based system target file (for example, `ert.tlc`). Requires Embedded Coder® license.

Specify one of the following values:

- `'Model reference'`: Code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL/PIL simulations with the protected model.
- `'Top model'`: Code access through the standalone interface. Users of the protected model can run Model block SIL/PIL simulations with the protected model.

Example: `'CodeInterface', 'Top model'`

### **ObfuscateCode — Option to obfuscate generated code**

`true` (default) | `false`

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation during protection is enabled. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode', true`

### **Path — Folder for protected model**

current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.



Example: 'Path', 'C:\Work'

### **Report — Option to generate a report**

false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: 'Report', true

### **hdl — Option to generate HDL code**

false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: 'hdl', true

### **OutputFormat — Protected code visibility**

'CompiledBinaries' (default) | 'MinimalCode' | 'AllReferencedHeaders'

---

**Note** This argument affects the output only when you specify **Mode** as `'Accelerator'` or `'CodeGeneration'`. When you specify **Mode** as `'Normal'`, only a MEX-file is part of the output package.

---

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. All code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. All code in the build folder is visible. All headers referenced by the code are also visible.

Example: `'OutputFormat', 'AllReferencedHeaders'`

### **Webview — Option to include a Web view**

`false` (default) | `true`

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdhref_counter`, you can call:

```
Simulink.ProtectedModel.open('sldemo_mdhref_counter', 'webview');
```

- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview', true`

### **Encrypt — Option to encrypt protected model**

`false` (default) | `true`

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model:  
`Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`

- Password for HDL code generation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: `'Encrypt',true`

### **CustomPostProcessingHook — Option to add postprocessing function for protected model files**

function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example:

```
'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)
```

### **Modifiable — Option to create a modifiable protected model**

false (default) | true

Option to create a modifiable protected model, specified as a Boolean value. To use this option:

- Add a password for modification using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. If a password has not been added at the time that you create the modifiable protected model, you are prompted to create one.
- Modify the options of your protected model by first providing the modification password using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. Then use the `Simulink.ModelReference.modifyProtectedModel` function to make your option changes.

Example: `'Modifiable',true`

### **Callbacks — Option to specify protected model callbacks**

cell array

Option to specify callbacks for a protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

## **Output Arguments**

### **harnessHandle — Handle of the harness model**

double

Handle of the harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is `0`.

### **neededVars — Names of base workspace variables**

cell array

Names of base workspace variables used by the model being protected, returned as a cell array.

The cell array can also include variables that the protected model does not use.

## **Alternatives**

“Protect Models to Conceal Contents” (Simulink Coder)

## **See Also**

`Simulink.ModelReference.ProtectedModel.clearPasswords` |  
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel` |  
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration` |  
`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGenerati`  
`on` | `Simulink.ModelReference.ProtectedModel.setPasswordForModify` |  
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation` |

Simulink.ModelReference.ProtectedModel.setPasswordForView |  
Simulink.ModelReference.modifyProtectedModel

## Topics

- “Protect Models to Conceal Contents” (Simulink Coder)
- “Protected Models for Model Reference” (Simulink)
- “Test Protected Models” (Simulink Coder)
- “Package and Share Protected Models” (Simulink Coder)
- “Specify Custom Obfuscators for Protected Models” (Simulink Coder)
- “Configure and Run SIL Simulation” (Embedded Coder)
- “Define Callbacks for Protected Models” (Simulink Coder)
- “Reference Protected Models from Third Parties” (Simulink)
- “Code Interfaces for SIL and PIL” (Embedded Coder)

## Introduced in R2012b

## Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

### Syntax

```
Simulink.ModelReference.modifyProtectedModel(model)
Simulink.ModelReference.modifyProtectedModel(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(
model,'Harness',true)
[~,neededVars] = Simulink.ModelReference.modifyProtectedModel(
model)
```

### Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified model. If `Name,Value` pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

`[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(model,'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(model) returns a cell array that includes the names of base workspace variables used by the protected model.

## Examples

### Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Create a modifiable protected model with support for code generation and Web view. Optionally, if you want to add support for HDL code generation, set 'hdl' to true.

```
Simulink.ModelReference.protect('sldemo_mdref_counter', 'Mode', ...  
'CodeGeneration', 'Modifiable', true, 'Report', true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...  
'sldemo_mdref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

### Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdref_counter', 'Mode', ...  
'CodeGeneration', 'Webview', true, 'Modifiable', true, 'Report', true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...  
'sldemo_mdref_counter', 'Mode', 'CodeGeneration', 'Report', true);
```

### **Change Encryption Password for Code Generation**

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
'sldemo_mdref_counter', 'cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdref_counter', 'Mode', ...  
'CodeGeneration', 'Encrypt', true, 'Modifiable', true, 'Report', true);
```

Provide the password to modify the protected model.



```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdhref_counter', 'password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(
'sldemo_mdhref_counter', 'Mode', 'CodeGeneration', 'Encrypt', true, ...
'Report', true, 'ChangeSimulationPassword', ...
{'cgpassword', 'new_password'});
```

### Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdhref_counter', 'password');
```

Create a modifiable protected model with a report and support for code generation with encryption. Optionally, if you want to add support for HDL code generation, set 'hdl' to true.

```
Simulink.ModelReference.protect('sldemo_mdhref_counter', 'Mode', ...
'CodeGeneration', 'Modifiable', true, 'Report', true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdhref_counter', 'password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdhref_counter', 'Mode', 'CodeGeneration', 'Report', true, ...
'Harness', true);
```

## Input Arguments

### **model** — Model name

string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example:

```
'Mode', 'CodeGeneration', 'OutputFormat', 'Binaries', 'ObfuscateCode', true
```

specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

#### General

##### Path — Folder for protected model

current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

Example: `'Path', 'C:\Work'`

##### Report — Option to generate a report

false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the report option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report', true`

##### hdl — Option to generate HDL code

false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

### **Harness — Option to create a harness model**

`false` (default) | `true`

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

### **CustomPostProcessingHook — Option to add postprocessing function for protected model files**

function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. The object also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example:

```
'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)
```

### **Functionality**

#### **Mode — Model protection mode**

`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode', 'Accelerator'`

### **OutputFormat — Protected code visibility**

`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

---

**Note** This argument affects the output only when you specify `Mode` as `'Accelerator'` or `'CodeGeneration'`. When you specify `Mode` as `'Normal'`, only a MEX-file is part of the output package.

---

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. Code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. Code in the build folder is visible. Header files referenced by the code are also visible.

Example: `'OutputFormat', 'AllReferencedHeaders'`

### **ObfuscateCode — Option to obfuscate generated code**

`true` (default) | `false`

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation is enabled for the protected model. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

### **Webview — Option to include a Web view**

false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdhref_counter`, you can call:

```
Simulink.ProtectedModel.open('sldemo_mdhref_counter', 'webview');
```

- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

### **Encryption**

#### **ChangeSimulationPassword — Option to change the encryption password for simulation**

cell array of two character vectors

Option to change the encryption password for simulation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeSimulationPassword',{'old_password','new_password'}`

#### **ChangeViewPassword — Option to change the encryption password for read-only view**

cell array of two character vectors

Option to change the encryption password for read-only view, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

### **ChangeCodeGenerationPassword — Option to change the encryption password for code generation**

cell array of two character vectors

Option to change the encryption password for code generation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: 'ChangeCodeGenerationPassword',  
{'old\_password', 'new\_password'}

### **Encrypt — Option to encrypt protected model**

false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model:  
`Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`
- Password for HDL code generation:  
`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: 'Encrypt', true

## **Output Arguments**

### **harnessHandle — Handle of the harness model**

double

Handle of the harness model, returned as a double or 0, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is 0.

**neededVars — Names of base workspace variables**

cell array

Names of base workspace variables used by the protected model, returned as a cell array.

The cell array can also include variables that the protected model does not use.

**See Also**

`Simulink.ModelReference.ProtectedModel.setPasswordForModify` |  
`Simulink.ModelReference.protect`

**Introduced in R2014b**

## Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration

Add or provide encryption password for HDL code generation from protected model

### Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(model,password)
```

### Description

`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(model,password)` adds an encryption password for HDL code generation if you create a protected model. If you use a protected model, the function provides the required password to generate code from the model.

### Examples

#### Create a Protected Model with Encryption for HDL Code Generation

Create a protected model with encryption for HDL code generation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(...  
'hdlcoder_referenced_model_gain','password');  
Simulink.ModelReference.protect('sldemo_mdref_counter',...  
'Mode','HDLCodeGeneration','Encrypt',true,'Report',true);
```

A protected model named `hdlcoder_referenced_model_gain.slxp` is created that requires an encryption password for HDL code generation.

#### Generate HDL Code from an Encrypted Protected Model

Use a protected model with encryption for HDL code generation.



Provide the encryption password required for HDL code generation from the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(...  
'hdlcoder_referenced_model_gain', 'password');
```

After you have provided the encryption password, you can generate code from the protected model.

## Input Arguments

### **model** — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

### **password** — Password for protected model code generation

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for code generation, the password is required.

## See Also

[Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration](#) |  
[Simulink.ModelReference.ProtectedModel.setPasswordForSimulation](#) |  
[Simulink.ModelReference.ProtectedModel.setPasswordForView](#)

## Topics

“Create Protected Models to Conceal Contents and Generate HDL Code”

**Introduced in R2019a**

# Simulink.ModelReference.ProtectedModel.setPasswordForModify

Add or provide password for modifying protected model

## Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(model, password)
```

## Description

`Simulink.ModelReference.ProtectedModel.setPasswordForModify(model, password)` adds a password for a modifiable protected model. After the password has been created, the function provides the password for modifying the protected model.

## Examples

### Add Functionality to Protected Model

Create a modifiable protected model with support for code generation, then modify it to add Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdref_counter', 'password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdref_counter', 'Mode', ...  
'CodeGeneration', 'Modifiable', true, 'Report', true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...  
'sldemo_mdhref_counter', 'password');
```

Add support for Web view to the protected model that you created. Optionally, if you want to add support for HDL code generation, set 'hdl' to true.

```
Simulink.ModelReference.modifyProtectedModel(...  
'sldemo_mdhref_counter', 'Mode', 'CodeGeneration', 'Webview', true, ...  
'Report', true);
```

## Input Arguments

### **model** — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model to be modified.

### **password** — Password to modify protected model

string or character vector

Password, specified as a string or character vector. The password is required for modification of the protected model.

## See Also

Simulink.ModelReference.modifyProtectedModel |  
Simulink.ModelReference.protect

**Introduced in R2014b**

# Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Add or provide encryption password for simulation of protected model

## Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,password)
```

## Description

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,password)` adds an encryption password for simulation if you create a protected model. If you use a protected model, the function provides the required password to simulate the model.

## Examples

### Create a Protected Model with Encryption

Create a protected model with encryption for simulation.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
'sldemo_mdref_counter','password');  
Simulink.ModelReference.protect('sldemo_mdref_counter',...  
'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdref_counter.slxp` is created that requires an encryption password for simulation.

## Simulate an Encrypted Protected Model

Use a protected model with encryption for simulation.

Provide the encryption password required for simulation of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...  
'sldemo_mdref_counter', 'password');
```

After you have provided the encryption password, you can simulate the protected model.

## Input Arguments

### **model** — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

### **password** — Password for protected model simulation

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for simulation, the password is required.

## See Also

Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |  
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGenerati  
on | Simulink.ModelReference.ProtectedModel.setPasswordForView |  
Simulink.ModelReference.protect

**Introduced in R2014b**

## Simulink.ModelReference.ProtectedModel.setPasswordForView

Add or provide encryption password for read-only view of protected model

### Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(model,  
password)
```

### Description

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model, password)` adds an encryption password for read-only view if you create a protected model. If you use a protected model, the function provides the required password for a read-only view of the model.

### Examples

#### Create a Protected Model with Encryption

Create a protected model with encryption for read-only view.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...  
'sldemo_mdref_counter', 'password');  
Simulink.ModelReference.protect('sldemo_mdref_counter', ...  
'Webview', true, 'Encrypt', true, 'Report', true);
```

A protected model named `sldemo_mdref_counter.slxp` is created that requires an encryption password for read-only view.

## View an Encrypted Protected Model

Use a protected model with encryption for read-only view.

Provide the encryption password required for the read-only view of the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(...  
'sldemo_mdref_counter', 'password');
```

After you have provided the encryption password, you have access to the read-only view of the protected model.

## Input Arguments

### **model** — Model name

string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

### **password** — Password for read-only view of protected model

string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for read-only view, the password is required.

## See Also

Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |  
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGenerati  
on | Simulink.ModelReference.ProtectedModel.setPasswordForSimulation |  
Simulink.ModelReference.protect

**Introduced in R2014b**

## Simulink.ModelReference.ProtectedModel.clearPasswords

Clear cached passwords for protected models

### Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

### Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears protected model passwords that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

### Examples

#### Clear cached passwords for protected models

After using protected models, clear passwords cached for the models during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswords()
```

### See Also

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel
```

### Topics

“Protect Models to Conceal Contents” (Simulink Coder)



**Introduced in R2014b**

## Simulink.ModelReference.ProtectedModel.-clearPasswordsForModel

Clear cached passwords for a protected model

### Syntax

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

### Description

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)` clears protected model passwords for `model` that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

### Examples

#### Clear cached passwords for a protected model

After using a protected model, clear passwords cached for the model during the MATLAB session.

```
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)
```

### Input Arguments

#### **model** — Protected model name

string or character vector

Model name specified as a string or character vector

Example: 'rtwdemo\_counter'

Data Types: char

## **See Also**

`Simulink.ModelReference.ProtectedModel.clearPasswords`

## **Topics**

“Protect Models to Conceal Contents” (Simulink Coder)

**Introduced in R2014b**

# Simulink.ProtectedModel.open

Open protected model

## Syntax

```
Simulink.ProtectedModel.open(model)  
Simulink.ProtectedModel.open(model, type)
```

## Description

`Simulink.ProtectedModel.open(model)` opens a protected model. If you do not specify how to view the protected model, the software first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

`Simulink.ProtectedModel.open(model, type)` opens a protected model using the specified viewing method. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report. If the method that you specify is not enabled, the software reports an error. The protected model is not opened.

## Examples

### Open a Protected Model

Open a protected model without a specified method.

Load the model and save a local copy.

```
sldemo_mdhref_counter  
save_system('sldemo_mdhref_counter', 'mdhref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdlref_counter', 'Mode', ...  
'CodeGeneration', 'Report', true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdlref_counter')
```

The protected model does not have Web view enabled, so the protected model report is opened.

### Open a Protected Model Web View

Open a protected model, specifying the Web view.

Load the model and save a local copy.

```
sldemo_mdlref_counter  
save_system('sldemo_mdlref_counter', 'mdlref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdlref_counter', 'Mode', ...  
'CodeGeneration', 'Webview', true, 'Report', true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdlref_counter', 'webview')
```

The protected model Web view is opened.

## Input Arguments

### **model** — Model name

string or character vector

Protected model name, specified as a string or character vector.

### **type** — Open method

'webview' | 'report'

Method for viewing the protected model. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report.

### **See Also**

`Simulink.ModelReference.protect`

**Introduced in R2015a**

# sschdladvisor

Open Simscape HDL Workflow Advisor

## Syntax

```
sschdladvisor(subsystem)  
sschdladvisor(model)
```

## Description

`sschdladvisor(subsystem)` opens the Simscape HDL Workflow Advisor for the subsystem within the model.

`sschdladvisor(model)` opens the Simscape HDL Workflow Advisor for the model.

## Examples

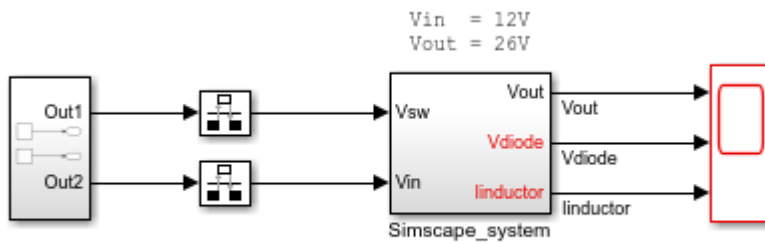
### Open Simscape HDL Workflow Advisor

This example shows how to open advisor for the model and a subsystem inside a model.

### Open Simscape HDL Advisor for a Model

For example: To open the advisor for the Boost Converter model, enter:

```
Modelname = 'sschdlexBoostConverterExample';  
open_system(Modelname)  
sschdladvisor(Modelname)
```



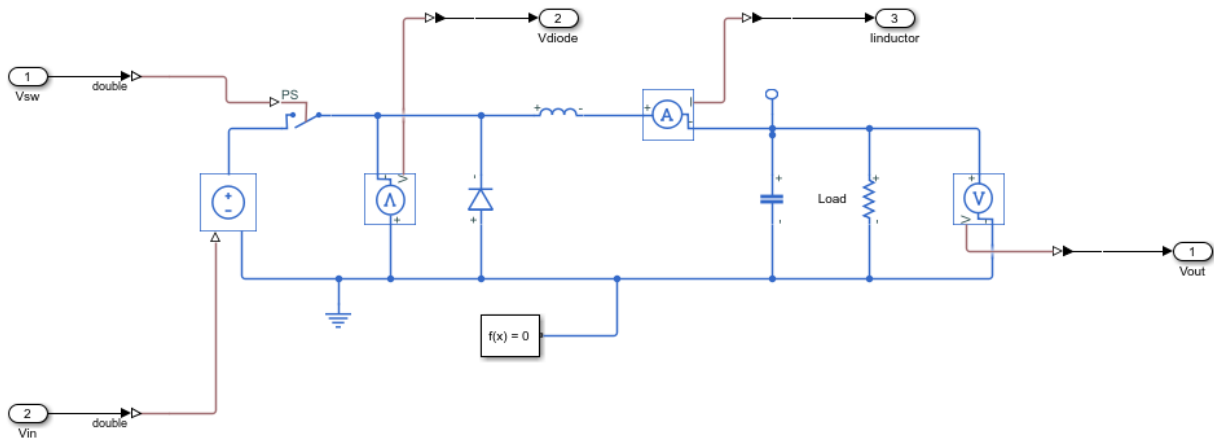
Copyright 2018 The MathWorks, Inc.

### Open Simscape HDL Advisor for a Subsystem

For example: To open the advisor for the `Simscape_system` block inside the Buck Converter model, enter:

```
Modelname = 'sschdlexBuckConverterExample';  
Subsysname = 'sschdlexBuckConverterExample/Simscape_system';  
load_system(Modelname)  
open_system(Subsysname)  
sschdladvisor(Subsysname)
```





## Input Arguments

### **subsystem** — Subsystem name

character vector

Subsystem name or handle, specified as a character vector.

Data Types: char

### **model** — Model name

character vector

Model name or handle, specified as a character vector.

Data Types: char

## See Also

`simscape.findNonlinearBlocks`

## Topics

“Generate HDL Code from Simscape Models”

**Introduced in R2018b**

# Supported Blocks

---

# 1-D Lookup Table

Approximate one-dimensional function (HDL Coder)

## Description

The 1-D Lookup Table block is a one-dimensional version of the n-D Lookup Table block. For HDL code generation information, see n-D Lookup Table.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## 2-D Lookup Table

Approximate two-dimensional function (HDL Coder)

### Description

The 2-D Lookup Table block is a two-dimensional version of the n-D Lookup Table block. For HDL code generation information, see n-D Lookup Table.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Abs

Output absolute value of input (HDL Coder)

## Description

The Abs block is available with Simulink.

For information about the simulation behavior and block parameters, see Abs.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Native Floating Point**

### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

## **Complex Data Support**

This block does not support code generation for complex signals. To calculate the magnitude of a complex number, use the Complex to Magnitude-Angle HDL Optimized block instead.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Add

Add inputs (HDL Coder)

## Description

The Add block is available with Simulink.

For information about the simulation behavior and block parameters, see Add.

## HDL Architecture

The default Linear architecture generates a chain of N operations (adders) for N inputs.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## Native Floating Point

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

## Complex Data Support

The default Linear implementation supports complex data.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Assertion

Check whether signal is zero (HDL Coder)

## Description

The Assertion block is available with Simulink.

For information about the simulation behavior and block parameters, see Assertion.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Assignment

Assign values to specified elements of signal (HDL Coder)

## Description

The Assignment block is available with Simulink.

For information about the simulation behavior and block parameters, see Assignment.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

### **Restrictions**

- 3-dimensional matrix inputs are not supported. You can use 1-D vectors and 2-D matrices with the block.
- Variable-size signals are not supported for code generation.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Atomic Subsystem

Represent system within another system (HDL Coder)

## Description

The Atomic Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see Atomic Subsystem.

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### **AdaptivePipelining**

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### **BalanceDelays**

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### **ClockRatePipelining**

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

#### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

#### **FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the **IP Core Generation** workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

**ProcessorFPGASynchronization**

Processor/FPGA synchronization mode, specified as a character vector.

To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: Free running (default) | Coprocessing - blocking

Example: 'Free running'

**TestPointMapping**

To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'TestPoint','AXI4-Lite','x"108"'}}

### TunableParameterMapping

To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: ' ' (default) | cell array of character vectors

Example: '{{'myParam', 'AXI4-Lite', 'x"108"'}}'

### AXI4RegisterReadback

To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'off' (default) | 'on'

### GenerateDefaultAXI4Slave

To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'on' (default) | 'off'

### IPCoreAdditionalFiles

Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).

You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.

Values: ' ' (default) | character vector

Example: 'C:\myprojfiles\led\_blinking\_file1.vhd;C:\myprojfiles\led\_blinking\_file2.vhd;'

### IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: ' ' (default) | character vector



Example: 'my\_model\_name'

### **IPCoreVersion**

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: '' (default) | character vector

Example: '1.3'

## **Restrictions**

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“External Component Interfaces”

“Generate Black Box Interface for Subsystem”

**Introduced in R2014a**

# Backlash

Model behavior of system with play (HDL Coder)

## Description

The Backlash block is available with Simulink.

For information about the simulation behavior and block parameters, see Backlash.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

The **Deadband width** and **Initial output** parameters support only scalar values.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Bias

Add bias to input (HDL Coder)

## Description

The Bias block is available with Simulink.

For information about the simulation behavior and block parameters, see Bias.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Bilateral Filter

2-D bilateral filtering (HDL Coder)

## Description

The Bilateral Filter block is available with Vision HDL Toolbox™.

For information about the simulation behavior and block parameters, see [Bilateral Filter](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**



# Biquad Filter

Model biquadratic IIR (SOS) filters (HDL Coder)

## Description

The Biquad Filter block is available with DSP System Toolbox™.

For information about the simulation behavior and block parameters, see Biquad Filter.

## Programmable Filter Support

HDL Coder supports programmable filters for Biquad Filter blocks.

- 1 On the filter block mask, set **Coefficient source** to **Input port(s)**.
- 2 Connect vector signals to the Num and Den coefficient ports.

The following limitations apply to the HDL optimizations for a programmable Biquad Filter block:

- Fully serial and partly serial architectures are not supported. **Architecture** must be set to **Fully parallel**.
- Canonical signed digit (CSD) multiplier optimization is not supported. **CoeffMultipliers** must be set to **multiplier**.

## Multichannel Filter Support

HDL Coder supports the use of vector inputs to Biquad Filter blocks.

- 1 Connect a vector signal to the Biquad Filter block input port.
- 2 Specify **Input processing** as **Elements as channels (sample based)**.
- 3 To reduce area by sharing the filter kernel between channels, set the **StreamingFactor** parameter of the subsystem to the number of channels. See the Streaming section of “Subsystem Optimizations for Filters”.

## HDL Architecture

### Block Optimizations

#### Serial Architectures

To use block-level optimizations to reduce hardware resources, select a serial **Architecture**. Then set either **NumMultipliers** or **Folding Factor**. See “HDL Filter Properties” on page 3-24.

When you select a serial architecture, set **Filter structure** to **Direct form I** or **Direct form II**. The direct form transposed structures are not supported with serial architectures.

#### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

Filter Structure	Pipeline Register Placement	Latency (Clock Cycles)
Any	Pipeline registers are added between the filter sections.	NS - 1, where NS is number of sections.

### Subsystem Optimizations

This block can participate in subsystem-level optimizations such as sharing, streaming, and pipelining. For the block to participate in subsystem-level optimizations, set **Architecture** to **Fully parallel**. See “Subsystem Optimizations for Filters”.

## HDL Filter Properties

#### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also **AddPipelineRegisters**.

#### CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully

parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See also `CoeffMultipliers`.

### **FoldingFactor**

Specify a serial implementation of an IIR SOS filter by the number of cycles it takes to generate the result. See also `FoldingFactor`.

### **NumMultipliers**

Specify a serial implementation of an IIR SOS filter by the number of hardware multipliers that are generated. See also `NumMultipliers`.

For HDL filter property descriptions, see “HDL Filter Block Properties”.

## **HDL Block Properties**

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “`ConstrainedOutputPipeline`”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “`InputPipeline`”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “`OutputPipeline`”.

## **Restrictions**

- Frame input is not supported for HDL code generation.
- You must set **Initial conditions** to 0. HDL code generation is not supported for nonzero initial states.
- You must select **Optimize unity scale values**.

- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Birds-Eye View

Transform front-facing camera image into top-down view (HDL Coder)

## Description

The Birds-Eye View block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see [Birds-Eye View](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# Bit Clear

Set specified bit of stored integer to zero (HDL Coder)

## Description

The Bit Clear block is available with Simulink.

For information about the simulation behavior and block parameters, see Bit Clear.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

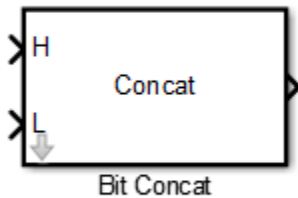
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



## Bit Concat

Concatenates up to 128 input words into single output



## Library

HDL Coder / Logic and Bit Operations

## Description

The Bit Concat block concatenates up to 128 input words into a single output. The input port labeled L designates the lowest-order input word. The port labeled H designates the highest-order input word. The right-to-left ordering of words in the output follows the low-to-high ordering of input signals.

How the block operates depends on the number and dimensions of the inputs, as follows:

- Single input: The input is a scalar or a vector. When the input is a vector, the coder concatenates the individual vector elements.
- Two inputs: Inputs are any combination of scalar and vector. When one input is scalar and the other is a vector, the coder performs scalar expansion. Each vector element is concatenated with the scalar, and the output has the same dimension as the vector. When both inputs are vectors, they must have the same size.
- Three or more inputs (up to a maximum of 128 inputs): Inputs are uniformly scalar or vector. All vector inputs must have the same size.

### Parameters

**Number of Inputs:** Enter an integer specifying the number of input signals. The number of block input ports updates when you change **Number of Inputs**.

- Default: 2
- Minimum: 1
- Maximum: 128

---

**Caution** Make sure that the **Number of Inputs** is equal to the number of signals you connect to the block. If the block has unconnected inputs, an error occurs at code generation time.

---

### Ports

The block has up to 128 input ports, with H representing the highest-order input word, and L representing the lowest-order input word. The maximum concatenated output word size is 128 bits.

### Supported Data Types

- Input: Fixed-point, integer (signed or unsigned), Boolean
- Output: Unsigned fixed-point or integer

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

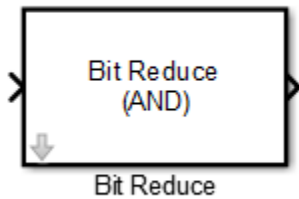
### See Also

Bit Reduce | Bit Rotate | Bit Shift | Bit Slice

**Introduced in R2014a**

# Bit Reduce

AND, OR, or XOR bit reduction on all input signal bits to single bit



## Library

HDL Coder / Logic and Bit Operations

## Description

The Bit Reduce block performs a selected bit-reduction operation (AND, OR, or XOR) on all the bits of the input signal, for a single-bit result.

## Parameters

### Reduction Mode

Specifies the reduction operation:

- AND (default): Perform a bitwise AND reduction of the input signal.
- OR: Perform a bitwise OR reduction of the input signal.
- XOR: Perform a bitwise XOR reduction of the input signal.

## Ports

The block has the following ports:

### Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

### Output

Supported data type: `ufix1`

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

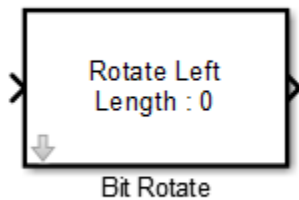
### **See Also**

Bit Concat | Bit Rotate | Bit Shift | Bit Slice

**Introduced in R2014a**

## Bit Rotate

Rotate input signal by bit positions



## Library

HDL Coder / Logic and Bit Operations

## Description

The Bit Rotate block rotates the input signal left or right by the specified number of bit positions.

## Parameters

**Rotate Mode:** Specifies direction of rotation, left or right. The default is `Rotate Left`.

**Rotate Length:** Specifies the number of bits to rotate. Specify a value greater than or equal to zero. The default is 0.

## Ports

The block has the following ports:

### Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

### Output

Has the same data type as the input signal.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.



## **See Also**

Bit Concat | Bit Reduce | Bit Shift | Bit Slice

**Introduced in R2014a**

# Bit Set

Set specified bit of stored integer to one (HDL Coder)

## Description

The Bit Set block is available with Simulink.

For information about the simulation behavior and block parameters, see Bit Set.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

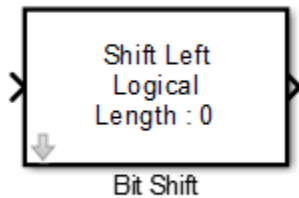
### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Bit Shift

Logical or arithmetic shift of input signal



## Library

HDL Coder / Logic and Bit Operations

## Description

The Bit Shift block performs a logical or arithmetic shift on the input signal.

## Parameters

### Shift Mode

Default: Shift Left Logical

Specifies the type and direction of shift:

- Shift Left Logical (default)
- Shift Right Logical
- Shift Right Arithmetic

### **Shift Length**

Specifies the number of bits to be shifted. Specify a value greater than or equal to zero. The default is 0.

## **Ports**

The block has the following ports:

### **Input**

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Minimum bit width: 2
- Maximum bit width: 128

### **Output**

Has the same data type and bit width as the input signal.

## **HDL Architecture**

This block has a single, default HDL architecture.

## **HDL Block Properties**

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

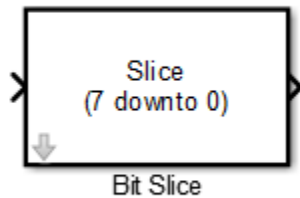
### **See Also**

Bit Concat | Bit Reduce | Bit Rotate | Bit Slice

**Introduced in R2014a**

## Bit Slice

Return field of consecutive bits from input signal



## Library

HDL Coder / HDL Operations

## Description

The Bit Slice block returns a field of consecutive bits from the input signal. Specify the lower and upper boundaries of the bit field by using zero-based indices in the **LSB Position** and **MSB Position** parameters.

## Parameters

### MSB Position

Specifies the bit position (zero-based) of the most significant bit (MSB) of the field to extract. The default is 7.

For an input word size **WS**, **LSB Position** and **MSB Position** must satisfy the following constraints:

```
WS > MSB Position >= LSB Position >= 0;
```

The word length of the output is computed as (MSB Position - LSB Position) + 1.

### **LSB Position**

Specifies the bit position (zero-based) of the least significant bit (LSB) of the field to extract. The default is 0.

## **Ports**

The block has the following ports:

Input

- Supported data types: Fixed-point, integer (signed or unsigned), Boolean
- Maximum bit width: 128

Output

Supported data types: unsigned fixed-point or unsigned integer.

## **HDL Architecture**

This block has a single, default HDL architecture.

## **HDL Block Properties**

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.



**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Bit Concat | Bit Reduce | Bit Rotate | Bit Shift

**Introduced in R2014a**

# Bitwise Operator

Specified bitwise operation on inputs (HDL Coder)

## Description

The Bitwise Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Bitwise Operator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# BPSK Demodulator Baseband

Demodulate BPSK-modulated data (HDL Coder)

## Description

The BPSK Demodulator Baseband block is available with Communications Toolbox™.

For information about the simulation behavior and block parameters, see BPSK Demodulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# BPSK Modulator Baseband

Modulate using binary phase shift keying method (HDL Coder)

## Description

The BPSK Modulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see BPSK Modulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Bus Assignment

Replace specified bus elements (HDL Coder)

## Description

The Bus Assignment block is available with Simulink.

For information about the simulation behavior and block parameters, see [Bus Assignment](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“Buses”

**Introduced in R2014b**

# Bus Creator

Create signal bus (HDL Coder)

## Description

The Bus Creator block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus Creator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

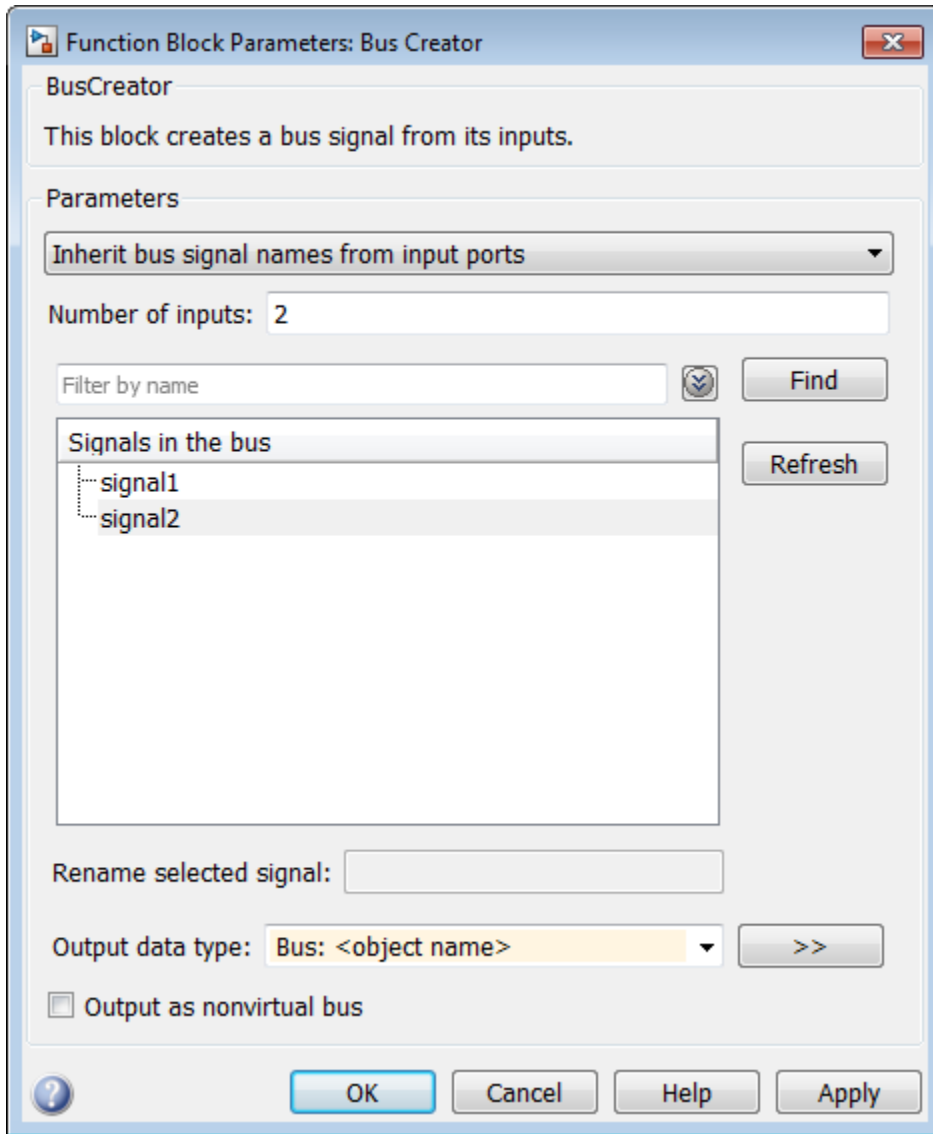
### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

### Setup

For **Output data type**, specify a bus object.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“Buses”

**Introduced in R2014a**

## Bus Selector

Select signals from incoming bus (HDL Coder)

### Description

The Bus Selector block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus Selector.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

Inputs must be bus signals. Non-bus inputs are not supported for code generation.

## **See Also**

### **Topics**

“Buses”

**Introduced in R2014a**

# Bus to Vector

Convert virtual bus to vector (HDL Coder)

## Description

The Bus to Vector block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus to Vector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“Buses”

**Introduced in R2016a**

# Channelizer HDL Optimized

Polyphase filter bank and fast Fourier transform—optimized for HDL code generation (HDL Coder)

## Description

The Channelizer HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Channelizer HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017a**

# Chart

Implement control logic with finite state machine (HDL Coder)

## Description

The Chart block is available with Stateflow®.

For information about the simulation behavior and block parameters, see Chart.

## Tunable Parameters

You can use a tunable parameter in a Stateflow Chart intended for HDL code generation.

For more information, see “Generate DUT Ports for Tunable Parameters”.

## HDL Architecture

This block has a single, default HDL architecture.

## Active State Output

To generate an output port in the HDL code that shows the active state, select **Create output port for monitoring** in the Properties window of the chart. The output is an enumerated data type. See “Simplify Stateflow Charts by Incorporating Active State Output” (Stateflow).

## Registered Output

If you want to insert an output register that delays the chart output by a simulation cycle, use the OutputPipeline block property.

---

## HDL Block Properties

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **InstantiateFunctions**

Generate a VHDL entity or Verilog module for each function. The default is `off`. See also “InstantiateFunctions”.

### **LoopOptimization**

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

### **MapPersistentVarsToRAM**

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

### VariablesToPipeline

---

**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

---

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

### Location of Charts in the Model

A chart intended for HDL code generation must be part of a Simulink subsystem. If the chart for which you want to generate code is at the root level of your model, embed the chart in a subsystem. Connect the relevant signals to the subsystem inputs and outputs.

### Data Types

The current release supports a subset of MATLAB data types in charts intended for use in HDL code generation. Supported data types are

- Signed and unsigned integer

- Double and single

---

**Note** Some results obtained from HDL code generated for models using double or single data types are not bit-true to results from simulation of the original model.

---

- Fixed point
- Boolean
- Enumeration

---

**Note** Except for data types assigned to ports, multidimensional arrays of these types are supported. Port data types must be either scalar or vector.

---

## Chart Initialization

You must enable the chart property **Execute (enter) Chart at Initialization**. This option executes the update chart function immediately following chart initialization. The option is required for HDL because outputs must be available at time 0 (hardware reset). “Execution of a Chart at Initialization” (Stateflow) describes existing restrictions under this property.

The reset action must not entail the delay of combinatorial logic. Therefore, do not perform arithmetic in initialization actions.

To generate HDL code that is more readable and has better synthesis results, enable the **Initialize Outputs Every Time Chart Wakes Up** chart property. If you use a Moore state machine, HDL Coder generates an error if you disable the chart property.

If you disable **Initialize Outputs Every Time Chart Wakes Up**, the chart output is persistent, so the generated HDL code must internally register the output values.

## Imported Code

A chart intended for HDL code generation must be entirely self-contained. The following restrictions apply:

- Do not call MATLAB functions other than `min` or `max`.
- Do not use MATLAB System Objects in a Chart block.

- Do not use MATLAB workspace data.
- Do not call C math functions. HDL does not have a counterpart to the C math library.
- If the **Enable bit operations** property is disabled, do not use the exponentiation operator (^). The exponentiation operator is implemented with the C Math Library function `pow`.
- Do not include custom code. Information entered on the **Simulation Target > Custom Code** pane in the Configuration Parameters dialog box is ignored.
- Do not share data (via Data Store Memory blocks) between charts. HDL Coder does not map such global data to HDL because HDL does not support global data.

### Vector of Tunable Parameters

Vector of Tunable Parameters as data types for Chart blocks are not supported.

### Input and Output Events

HDL Coder supports the use of input and output events with Stateflow charts, subject to the following constraints:

- You can define and use only one input event per Stateflow chart. (There is no restriction on the number of output events that you can use.)
- The coder does not support HDL code generation for charts that have a single input event, and which also have nonzero initial values on the chart's output ports.
- All input and output events must be edge-triggered.

For detailed information on input and output events, see “Activate a Stateflow Chart by Sending Input Events” (Stateflow) and “Activate a Simulink Block by Sending Output Events” (Stateflow).

### Messages

Stateflow messages are not supported for HDL code generation.

### Loops

Other than `for` loops, do not explicitly use loops in a chart intended for HDL code generation. Observe the following restrictions on `for` loops:



- The data type of the loop counter variable must be `int32`.
- HDL Coder supports only constant-bounded loops.

The `for` loop example, `sf_for`, shows a design pattern for a `for` loop using a graphical function.

## Other Restrictions

HDL Coder imposes additional restrictions on the use of classic chart features. These limitations exist because HDL does not support some features of general-purpose sequential programming languages.

- Do not define local events in a chart from which HDL code is generated.

Do not use the following implicit events:

- `enter`
- `exit`
- `change`

You can use the following implicit events:

- `wakeup`
- `tick`

You can use temporal logic if the base events are limited to these types of implicit events.

---

**Note** Absolute-time temporal logic is not supported for HDL code generation.

---

- Do not use recursion through graphical functions. HDL Coder does not currently support recursion.
- Avoid unstructured code. Although charts allow unstructured code (through transition flow diagrams and graphical functions), this usage results in `goto` statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements. Therefore, do not use unstructured flow diagrams.
- If you have not selected the **Initialize Outputs Every Time Chart Wakes Up** chart option, do not read from output ports.

- Do not use Data Store Memory objects.
- Do not use pointer (&) or indirection (\*) operators. See “Pointer and Address Operations” (Stateflow).
- If a chart gets a run-time overflow error during simulation, it is possible to disable data range error checking and generate HDL code for the chart. However, in such cases, some results obtained from the generated HDL code might not be bit-true to results from the simulation. The recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

## See Also

Sequence Viewer | State Transition Table | Truth Table

## Topics

“Generate HDL for Mealy and Moore Finite State Machines”

“Design Patterns Using Advanced Chart Features”

“Hardware Realization of Stateflow Semantics”

**Introduced in R2014a**

# Check Discrete Gradient

Check that absolute value of difference between successive samples of discrete signal is less than upper bound (HDL Coder)

## Description

The Check Discrete Gradient block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Discrete Gradient.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Dynamic Gap

Check that gap of possibly varying width occurs in range of signal's amplitudes (HDL Coder)

## Description

The Check Dynamic Gap block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Gap.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Dynamic Lower Bound

Check that one signal is always less than another signal (HDL Coder)

## Description

The Check Dynamic Lower Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Lower Bound.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Dynamic Range

Check that signal falls inside range of amplitudes that varies from time step to time step (HDL Coder)

## Description

The Check Dynamic Range block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Range.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Dynamic Upper Bound

Check that one signal is always greater than another signal (HDL Coder)

## Description

The Check Dynamic Upper Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Upper Bound.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Check Input Resolution

Check that input signal has specified resolution (HDL Coder)

## Description

The Check Input Resolution block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Input Resolution.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Static Gap

Check that gap exists in signal's range of amplitudes (HDL Coder)

## Description

The Check Static Gap block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Gap.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Static Lower Bound

Check that signal is greater than (or optionally equal to) static lower bound (HDL Coder)

## Description

The Check Static Lower Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Lower Bound.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Static Range

Check that signal falls inside fixed range of amplitudes (HDL Coder)

## Description

The Check Static Range block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Range.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Check Static Upper Bound

Check that signal is less than (or optionally equal to) static upper bound (HDL Coder)

## Description

The Check Static Upper Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Upper Bound.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Chroma Resampler

Downsample or upsample chrominance component (HDL Coder)

## Description

The Chroma Resampler block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Chroma Resampler.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

## CIC Decimation

Decimate signal using Cascaded Integrator-Comb filter (HDL Coder)

### Description

The CIC Decimation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see [CIC Decimation](#).

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

### HDL Architecture

#### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

Pipeline Register Placement	Latency (clock cycles)
A pipeline register is added between the comb stages of the differentiators.	NS - 1, where NS is number of sections (at the output side).

### HDL Filter Properties

#### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also [AddPipelineRegisters](#).



## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option Zero-latency decimator is not supported for HDL code generation. From the **Filter Structure** drop-down list, select Decimator.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# CIC Interpolation

Interpolate signal using Cascaded Integrator-Comb filter (HDL Coder)

## Description

The CIC Interpolation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see [CIC Interpolation](#).

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

## HDL Architecture

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

Pipeline Register Placement	Latency (clock cycles)
A pipeline register is added between the comb stages of the differentiators.	NS, the number of sections (at the input side).

## HDL Filter Properties

### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also [AddPipelineRegisters](#).

# HDL Block Properties

## ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

## InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

## OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

# Restrictions

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option Zero-latency interpolator is not supported for HDL code generation. From the **Filter Structure** drop-down list, select Interpolator.
- When you use **AddPipelineRegisters**, delays in parallel paths are not automatically balanced. Manually add delays where required by your design.

## Introduced in R2014a

---

# Closing

Morphological close of binary pixel data (HDL Coder)

## Description

The Closing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Closing.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Color Space Converter

Convert color information between color spaces (HDL Coder)

## Description

The Color Space Converter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Color Space Converter.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**



# Compare To Constant

Determine how signal compares to specified constant (HDL Coder)

## Description

The Compare To Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Compare To Constant.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Compare To Zero

Determine how signal compares to zero (HDL Coder)

## Description

The Compare To Zero block is available with Simulink.

For information about the simulation behavior and block parameters, see Compare To Zero.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Complex to Magnitude-Angle HDL Optimized

Compute magnitude and/or phase angle of complex signal—optimized for HDL code generation using the CORDIC algorithm (HDL Coder)

## Description

The Complex to Magnitude-Angle HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Complex to Magnitude-Angle HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Complex to Real-Imag

Output real and imaginary parts of complex input signal (HDL Coder)

## Description

The Complex to Real-Imag block is available with Simulink.

For information about the simulation behavior and block parameters, see [Complex to Real-Imag](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Constant

Generate constant value (HDL Coder)

## Description

The Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Constant.

## Tunable Parameters

You can use a tunable parameter in a Constant block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

## HDL Architecture

Architecture	Parameters	Description
default Constant	None	This implementation emits the value of the Constant block.
Logic Value	None	By default, this implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.
	{'Value', 'Z'}	If the signal is in a high-impedance state, use this parameter value. This implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.

Architecture	Parameters	Description
	{'Value', 'X'}	If the signal is in an unknown state, use this parameter value. This implementation emits the character 'X' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'XXXX'.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

- The `Logic Value` implementation does not support the `double` data type. If you specify this implementation for a constant value of type `double`, a code generation error occurs.
- For **Sample time**, enter -1. Delay balancing does not support an `inf` sample time.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Constellation Diagram

Display constellation diagram for input signals (HDL Coder)

## Description

The Constellation Diagram block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Constellation Diagram.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Convert 1-D to 2-D

Reshape 1-D or 2-D input to 2-D matrix with specified dimensions (HDL Coder)

## Description

The Convert 1-D to 2-D block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Convert 1-D to 2-D.

## HDL Architecture

This block has a pass-through implementation.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Convolutional Deinterleaver

Restore ordering of symbols that were permuted using shift registers (HDL Coder)

## Description

The Convolutional Deinterleaver block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Deinterleaver.

## HDL Architecture

- “Shift Register Based Implementation” on page 3-109
- “RAM Based Implementation” on page 3-109

### Shift Register Based Implementation

The default implementation for the Convolutional Deinterleaver block is shift register-based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to 'none'.

When you set `ResetType` to 'none', reset is not applied to the shift registers. When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

### RAM Based Implementation

When you select the RAM implementation for a Convolutional Deinterleaver block, HDL Coder uses RAM resources instead of shift registers.

# HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

# Restrictions

When you select the RAM implementation:

- Double or single data types are not supported for either input or output signals.
- You must set **Initial conditions** for the block to zero.
- At least two rows of interleaving are required.

# Extended Capabilities

## C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.



## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Convolutional Encoder

Create convolutional code from binary data (HDL Coder)

## Description

The Convolutional Encoder block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Encoder.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- Input data requirements:
  - Must be sample-based,
  - Must have a `boolean` or `ufix1` data type.
- HDL Coder supports only the following coding rates:
  - $\frac{1}{2}$  to  $\frac{1}{7}$
  - $\frac{2}{3}$
- The coder supports only constraint lengths for 3 to 9.
- Specify **Trellis structure** by the `poly2trellis` function.
- The coder supports the following **Operation mode** settings:
  - Continuous
  - Reset on nonzero input via port

If you select this mode, you must select the **Delay reset action to next time step** option. When you select this option, the Convolutional Encoder block finishes its current computation before executing a reset.

- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Convolutional Interleaver

Permute input symbols using set of shift registers (HDL Coder)

## Description

The Convolutional Interleaver block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Interleaver.

## HDL Architecture

- “Shift Register Based Implementation” on page 3-114
- “RAM Based Implementation” on page 3-114

## Shift Register Based Implementation

The default implementation for the Convolutional Interleaver block is shift register-based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to 'none'.

When you set `ResetType` to 'none', reset is not applied to the shift registers. When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

## RAM Based Implementation

When you select the RAM implementation for a Convolutional Interleaver block, HDL Coder uses RAM resources instead of shift registers.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## Restrictions

When you select the RAM implementation:

- Double or single data types are not supported for either input or output signals.
- You must set **Initial conditions** for the block to zero.
- At least two rows of interleaving are required.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Cosine

Implement fixed-point cosine wave using lookup table approach that exploits quarter wave symmetry (HDL Coder)

## Description

The Cosine block is available with Simulink.

For information about the simulation behavior and block parameters, see [Sine, Cosine](#).

## HDL Architecture

The HDL code implements Cosine using the quarter-wave lookup table that you specify in the Simulink block parameters.

To avoid generating a division operator ( $/$ ) in the HDL code, for **Number of data points for lookup table**, enter  $(2^n)+1$ .  $n$  is an integer.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Limitations

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### See Also

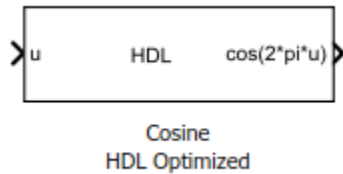
Cosine HDL Optimized | Sine | Sine HDL Optimized

**Introduced in R2014a**



# Cosine HDL Optimized

Implement fixed-point cosine wave optimized for HDL code generation



## Library

HDL Coder / Lookup Tables

## Description

The Cosine HDL Optimized block implements a fixed-point cosine wave by using a lookup table method that exploits quarter-wave symmetry.

For the most efficient HDL implementation, configure the block with an exact power of two as the number of elements. In the Block Parameters dialog box, for **Number of data points**, specify an integer that is an exact power of two. That is, specify the lookup table data points to be  $(2^n)$ , where  $n$  is an integer. By default, the **Number of data points** is 64.

When you specify a power of two for the **Number of data points**, the lookup tables precede a register without reset after HDL code generation. The combination of the lookup table block and register without reset maps efficiently to RAM on the target device.

Depending on your selection of the **Output formula** parameter, the blocks can output these functions of the input signal:

- $\sin(2\pi u)$
- $\cos(2\pi u)$

- $\exp(i2\pi u)$
- $\sin(2\pi u)$  and  $\cos(2\pi u)$

Use the **Table data type** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

## Data Type Support

The Cosine HDL Optimized block accepts signals of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” (Simulink) in the Simulink documentation.

## Parameters

### Output formula


Select the signal(s) to output.

### Number of data points

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be  $(2^n)$ , where  $n$  is an integer.

### Table data type

Specify the table data type. You can specify an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

### Show data type assistant

Select the mode of data type specification. If you select **Expression**, enter an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

If you select **Fixed point**, you can use the options in the **Data Type Assistant** to specify the fixed-point data type. In the **Fixed point** mode, you can choose binary point scaling, and specify the signedness, word length, fraction length, and the data type override setting.

## Characteristics

Data Types	Double   Single   Boolean   Base Integer   Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

## HDL Architecture

The HDL code implements the Cosine HDL Optimized block by using the quarter-wave lookup table that you specify in the Simulink block parameters.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Sine HDL Optimized | Sine, Cosine | Trigonometric Function

### **Introduced in R2016b**

# Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere (HDL Coder)

## Description

The Coulomb and Viscous Friction block is available with Simulink.

For information about the simulation behavior and block parameters, see Coulomb and Viscous Friction.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

HDL code generation does not support complex input.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Counter Free-Running

Count up and overflow back to zero after reaching maximum value for specified number of bits (HDL Coder)

## Description

The Counter Free-Running block is available with Simulink.

For information about the simulation behavior and block parameters, see Counter Free-Running.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Counter Limited

Count up and wrap back to zero after outputting specified upper limit (HDL Coder)

## Description

The Counter Limited block is available with Simulink.

For information about the simulation behavior and block parameters, see Counter Limited.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Data Type Conversion

Convert input signal to specified data type (HDL Coder)

## Description

The Data Type Conversion block is available with Simulink.

For information about the simulation behavior and block parameters, see Data Type Conversion.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**Note** If you use `double` data types in your model, use this block for conversion between `double` and `single` data types. You cannot use the block to convert between `double` and fixed-point data types.

---

### Native Floating Point

With the HDL Model Checker, you can replace Data Type Conversion blocks that use the `Stored Integer (SI)` mode and convert between floating-point and fixed-point data types with Float Typecast blocks.

#### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

#### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

If you configure a Data Type Conversion block for `double` to fixed-point conversion or fixed-point to `double` conversion, a warning is displayed during code generation.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Data Type Duplicate

Force all inputs to same data type (HDL Coder)

## Description

The Data Type Duplicate block is available with Simulink.

For information about the simulation behavior and block parameters, see [Data Type Duplicate](#).

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals (HDL Coder)

## Description

The Data Type Propagation block is available with Simulink.

For information about the simulation behavior and block parameters, see Data Type Propagation.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# DC Blocker

Block DC component (HDL Coder)

## Description

The DC Blocker block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see DC Blocker.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Dead Zone

Provide region of zero output (HDL Coder)

## Description

The Dead Zone block is available with Simulink.

For information about the simulation behavior and block parameters, see Dead Zone.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Dead Zone Dynamic

Set inputs within bounds to zero (HDL Coder)

## Description

The Dead Zone Dynamic block is available with Simulink.

For information about the simulation behavior and block parameters, see Dead Zone Dynamic.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# Decrement Real World

Decrease real world value of signal by one (HDL Coder)

## Description

The Decrement Real World block is available with Simulink.

For information about the simulation behavior and block parameters, see Decrement Real World.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Decrement Stored Integer

Decrease stored integer value of signal by one (HDL Coder)

## Description

The Decrement Stored Integer block is available with Simulink.

For information about the simulation behavior and block parameters, see Decrement Stored Integer.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Delay

Delay input signal by fixed or variable sample periods (HDL Coder)

### Description

The Delay block is available with Simulink. For information about simulation behavior and block parameters, see Delay.

Block Parameter Setting	Description
Set <b>External reset</b> to Level .	Generates a reset port in the HDL code.
Select <b>Show enable port</b> .	Generates an enable port in the HDL code.
For <b>Initial condition</b> , set <b>Source</b> to Dialog and enter the value.	Specifies an initial condition for the block.
Set <b>Input processing</b> to Columns as channels (frame based).	Expects vector input data, where each element of the vector represents a sample in time.

### Additional Settings When Using State Control Block

If you use a State Control block with the Delay block inside a subsystem in your Simulink model, use these additional settings.

Block Parameter Setting	Description
Set <b>External reset</b> to Level hold for Synchronous mode and Level for Classic mode of the State Control block.	Generates a reset port in the HDL code.
Set <b>Delay length</b> to zero for a Delay block with an external enable port.	Treated as a wire in only Synchronous mode of the State Control block.
Set <b>Delay length</b> to zero for a Delay block with an external reset port.	Treated as a wire in Synchronous and Classic modes of the State Control block.

For more information about the State Control block, see State Control.

---

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### UseRAM

Map delays to RAM instead of registers. The default is `off`. See also “UseRAM”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

For **Initial condition** and **Delay length**, **Source** set to `Input port` is not supported for HDL code generation.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Demosaic Interpolator

Construct RGB pixel data from Bayer pattern pixels (HDL Coder)

## Description

The Demosaic Interpolator block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Demosaic Interpolator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Demux

Extract and output elements of vector signal (HDL Coder)

## Description

The Demux block is available with Simulink.

For information about the simulation behavior and block parameters, see Demux.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Depuncturer

Reverse puncturing scheme to prepare for decoding (HDL Coder)

## Description

The Depuncturer block is available with LTE HDL Toolbox™.

For information about the simulation behavior and block parameters, see Depuncturer.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

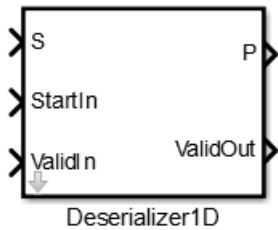
### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

## Deserializer1D

Convert scalar stream or smaller vectors to vector signal



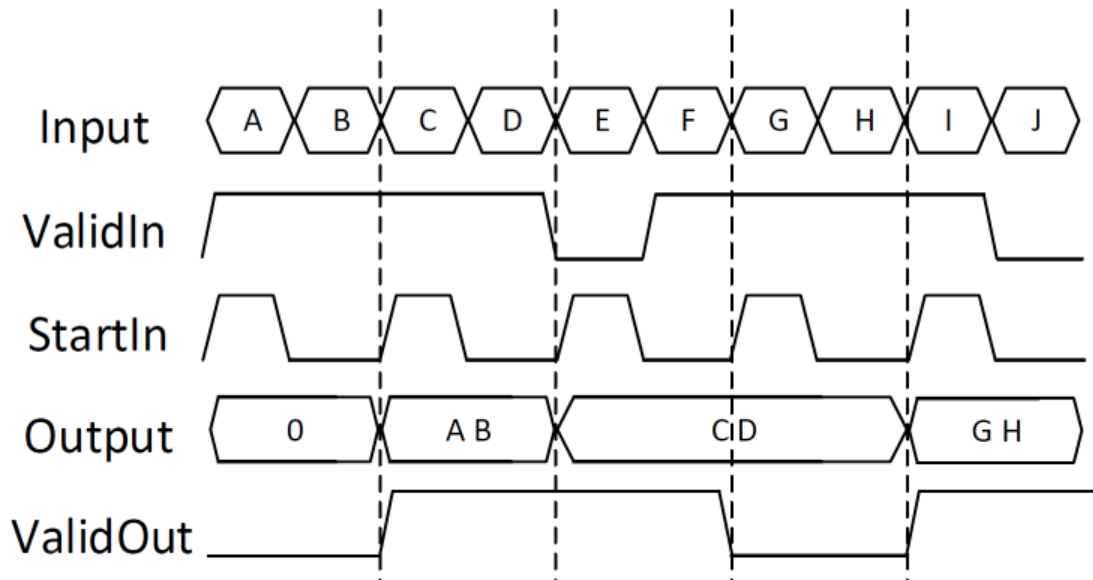
## Library

HDL Coder / HDL Operations

## Description

The Deserializer1D block buffers a faster, scalar stream or vector signals into a larger, slower vector signal. The faster input signal is converted to a slower signal based on the **Ratio** and **Idle Cycle** values, the conversion changes sample time. Also, the output signal is delayed one slow signal cycle because the serialized data needs to be collected before it can be output as a vector. See the examples below for more details.

You can configure the deserialization to depend on a valid input signal ValidIn and a start signal StartIn. If the **ValidIn** and **StartIn** block parameters are both selected, data collection starts only if both ValidIn and StartIn signals are true. Consider this example:

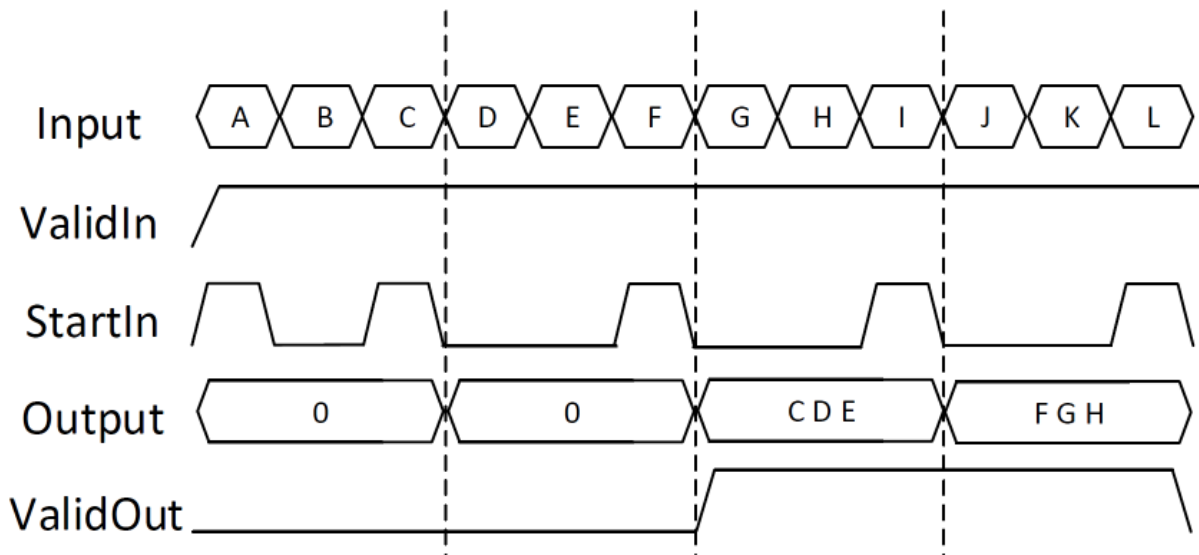


- **Ratio** is 2 and **Idle Cycles** is 0, so each output cycle is two input signals long with all data points considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection begins for A and B. The block outputs the deserialized vector in the next valid cycle, so the AB vector is output in the next cycle. This is also true in the second cycle for C and D.

In the third cycle, starting at E, StartIn is true, but ValidIn is not. E is dropped. At F, ValidIn is true, but StartIn is not, so F is also dropped. Since it cannot collect data for E or F, Deserializer1D outputs the previous cycle vector, CD, but ValidOut changes to false.

Another scenario to consider is when the StartIn signal arrives too early. If the length between two StartIn signals is not long enough to collect a full ratio cycle, the insufficient signal data is dropped. Consider this example:



- **Ratio** is 3, so each cycle is two sections long.
- **Idle Cycles** is 0, so all data inputs are considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection can begin for A and B. However, at C another StartIn signal arrives before three signals can be collected. Because the StartIn arrived early, A and B are dropped and no valid vector is collected during the first cycle. Therefore, the output of the second cycle is still zero. Deserialization begins at the StartIn at C, for C, D, and E. This vector is output at the next valid cycle, which is cycle 3. Similarly, deserialization starts again at the StartIn at F, and outputs the FGH vector in the fourth cycle.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

## Parameters

### Ratio

Enter the deserialization ratio. Default is 1.

The ratio is the output vector size, divided by the input vector size. The ratio must be divisible by the input vector size.

### Idle Cycles

Enter the number of idle cycles added to the end of each serialized input. Default is 0.

The value of **Idle Cycles** affects the deserialized output rate. For example, if **Ratio** is 2 and the input signal is A, B, B, C, D, D, . . ., without idle cycles the output would be AB, BC, DD . . . However for the same input and ratio with **Idle Cycles** set to 1, the output is AB, CD . . . The idle cycles, B and D, are dropped.

The Deserializer1D behavior changes if **Idle Cycles** is not zero, and **ValidIn** or **StartIn** are on. The idle cycles value affects only the output rate, while **ValidIn** and **StartIn** control what input data is deserialized.

### Initial condition

Specify the initial output of the simulation. Default is 0.

### StartIn

Select to activate the StartIn port. Default is off.

### ValidIn

Select to activate the ValidIn port. Default is off.

### ValidOut

Select to activate ValidOut port. Default is off.

### Input data port dimensions (-1 for inherited)

Enter the size of the input data signal. The input size must be divisible by the ratio plus the number of idle cycles. By default, the block inherits size based on context within the model.

### Input sample time (-1 for inherited)

Enter the time interval between sample time hits or specify another appropriate sample time such as continuous. By default, the block inherits its sample time based on context within the model. For more information, see “Sample Time” (Simulink).

**Input signal type**

Specify the input signal type of the block as `auto`, `real`, or `complex`.

**Ports****S**

Input signal to deserialize. Bus data types are not supported.

**ValidIn**

Indicates valid input signal. Use with the Serializer1D block. This port is available when you select the **ValidIn** check box.

Data type: Boolean

**StartOut**

Indicates where to start deserialization. Use with the Serializer1D block. This port is available when you select the **StartOut** check box.

Data type: Boolean

**P**

Deserialized output signal. Bus data types are not supported.

**ValidOut**

Indicates valid output signal. This port is available when you select the **ValidOut** check box.

Data type: Boolean

**HDL Architecture**

---

**Note** For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

---

This block has a single, default HDL architecture.

## **HDL Block Properties**

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Serializer1D

**Introduced in R2014b**



# Detect Change

Detect change in signal value (HDL Coder)

## Description

The Detect Change block determines whether there is a change in the input signal from its previous value. The Detect Change block is available with Simulink. For information about the simulation behavior and block parameters, see Detect Change.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

# Detect Decrease

Detect decrease in signal value (HDL Coder)

## Description

The Detect Decrease block determines whether the input signal is less than its previous value. When the input signal is less than the previous value, the output is true or equal to one. When the input is greater than or equal to the previous value, the output is false or equal to zero.

The Detect Decrease block is available with Simulink. For information about the simulation behavior and block parameters, see Detect Decrease.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

# Detect Increase

Detect increase in signal value (HDL Coder)

## Description

The Detect Increase block determines whether the input signal is greater than its previous value. When the input signal is greater than the previous value, the output is true or equal to one. When the input is less than or equal to the previous value, the output is false or equal to zero.

The Detect Increase block is available with Simulink. For information about the simulation behavior and block parameters, see Detect Increase.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

## Digital Filter (Obsolete)

Filter each channel of input over time using static or time-varying digital filter implementations (HDL Coder)

### Description

The Digital Filter block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Digital Filter.

---

**Note** Use of Digital Filter block in future releases is not recommended. Existing instances will continue to operate, but certain functionality will be disabled. See “Functionality being removed or replaced for blocks and System objects” (DSP System Toolbox). We strongly recommend using Discrete FIR Filter or Biquad Filter in new designs.

---

### HDL Architecture

When you specify `SerialPartition` and `ReuseAccum` for a Digital Filter block, observe the following constraints.

- If you specify **Dialog parameters** as the Coefficient source:
  - Set **Transfer function type** to FIR (all zeros).
  - Select **Filter structure** as one of: Direct form, Direct form symmetric, or Direct form asymmetric.

### Distributed Arithmetic Support

Distributed Arithmetic properties **DALUTPartition** and **DARadix** are supported for the following filter structures.

Architecture	Supported FIR Structures
default	FIR, Asymmetric FIR, and Symmetric FIR

## AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

Architecture	Pipeline Register Placement	Latency (clock cycles)
FIR, Asymmetric FIR, and Symmetric FIR filters	A pipeline register is added between levels of a tree-based adder.	$\text{ceil}(\log_2(FL))$ . FL is the filter length.
FIR Transposed	A pipeline register is added after the products.	1
IIR SOS	Pipeline registers are added between the filter sections.	NS - 1. NS is the number of sections.

## HDL Filter Properties

### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also **AddPipelineRegisters**.

### CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See also **CoeffMultipliers**.

### DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value



equal to the filter length to generate DA code without LUT partitions. See also `DALUTPartition`.

**MultiplierInputPipeline**

Specify the number of pipeline stages to add at filter multiplier inputs. See also `MultiplierInputPipeline`.

**MultiplierOutputPipeline**

Specify the number of pipeline stages to add at filter multiplier outputs. See also `MultiplierOutputPipeline`.

**ReuseAccum**

Enable or disable accumulator reuse in a serial filter implementation. Set **ReuseAccum** to on to use a cascade-serial implementation. See also `ReuseAccum`.

## HDL Block Properties

**ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “`ConstrainedOutputPipeline`”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “`InputPipeline`”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “`OutputPipeline`”.

## Complex Coefficients and Data Support

Except for decimator and interpolator filter structures, HDL Coder supports use of complex coefficients and complex input signals for all filter structures of the Digital Filter block.

### Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- HDL Coder does not support the Digital Filter block **Input port(s)** option for HDL code generation.

**Introduced in R2015a**

# Dilation

Morphological dilate of binary pixel data (HDL Coder)

## Description

The Dilation block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see [Dilation](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Direct Lookup Table (n-D)

Index into N-dimensional table to retrieve element, column, or 2-D matrix (HDL Coder)

## Description

The Direct Lookup Table (n-D) block is available with Simulink.

For information about the simulation behavior and block parameters, see Direct Lookup Table (n-D).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

# Restrictions

## MAX 10 Device Settings

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

## Required Block Settings

- **Number of table dimensions:** HDL Coder supports a maximum dimension of 2.
- **Inputs select this object from table:** Select Element.
- **Make table an input:** Clear this check box.
- **Diagnostic for out-of-range input:** Select Error. If you select other options, the coder displays a warning.

## Table Data Typing and Sizing

- It is good practice to size each dimension in the table to be a power of two. If the length of a dimension (*except* the innermost dimension) is not a power of two, HDL Coder issues a warning. By following this practice, you can avoid multiplications during table indexing operations and realize a more efficient table in hardware.
- Table data must resolve to a nonfloating-point data type. The coder examines the output port to verify that its data type meets this requirement.
- All ports on the block require scalar values.

# Extended Capabilities

## C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Discrete FIR Filter

Model finite impulse response filter (HDL Coder)

## Description

The Discrete FIR Filter block is available with Simulink, but a DSP System Toolbox license is required to use a filter structure other than direct form.

For information about the simulation behavior and block parameters, see Discrete FIR Filter.

For hardware-friendly valid and reset control signals, and to model exact hardware latency behavior in Simulink, use the Discrete FIR Filter HDL Optimized block instead.

## Multichannel Filter Support

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents an independent channel.

- 1 Connect a vector signal to the Discrete FIR Filter block input port.
- 2 Specify **Input processing** as `Elements as channels (sample based)`.
- 3 To reduce area by sharing the filter kernel between channels, set the **ChannelSharing** property to the number of channels.

## Programmable Filter Support

HDL Coder supports programmable filters for Discrete FIR Filter blocks.

- 1 On the filter block mask, set **Coefficient source** to **Input port**.
- 2 Connect a vector signal to the Num coefficient port.

## Frame-Based Input Support

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents a sample in time. You can use an input vector of up to



512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types. You can use real input signals with real coefficients, complex input signals with real coefficients, or real input signals with complex coefficients. You can also use frame-based input with programmable coefficients.

- 1 Connect a vector signal to the Discrete FIR Filter block input port.
- 2 Specify **Input processing** as Columns as channels (frame based).
- 3 Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to Frame Based. The block implements a parallel HDL architecture. See “Frame-Based Architecture”.

## Control Ports

You can generate HDL code for filters with or without the optional enable port, and with or without the optional reset port.

## HDL Architecture

To reduce area or increase speed, the Discrete FIR Filter block supports either block-level optimizations or subsystem-level optimizations. When you enable block optimizations, the block cannot participate in subsystem optimizations. Use block optimizations when your design is a single one-channel filter. Use subsystem optimizations to share resources across multiple channels or multiple filters.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

## Block Optimizations

### Serial Architectures

To use block-level optimizations to reduce hardware resources, set **Architecture** to one of the serial options. See “HDL Filter Architectures”.

When you specify **SerialPartition** and **ReuseAccum** for a Discrete FIR Filter block, set **Filter structure** to Direct form, Direct form symmetric, or Direct form asymmetric. The Direct form transposed structure is not supported with serial architectures.

### Distributed Arithmetic

To minimize multipliers by replacing them with LUTs and shift registers, use a distributed arithmetic (DA) filter implementation. See “Distributed Arithmetic for HDL Filters”.

When you select the Distributed Arithmetic (DA) architecture and use the **DALUTPartition** and **DARadix** distributed arithmetic properties, set **Filter structure** to Direct form, Direct form symmetric, or Direct form asymmetric. The Direct form transposed structure is not supported with distributed arithmetic.

### Multichannel Area Reduction

To share logic between channels, you can use the block-level **ChannelSharing** option or the subsystem-level **StreamingFactor** option. Set either property to the number of channels. Using **ChannelSharing** excludes the filter from other optimizations.

**StreamingFactor** operates over all eligible logic in a subsystem, rather than on a single block. It also enables the filter to participate in other subsystem optimizations. See the Streaming section of “Subsystem Optimizations for Filters”.

### Pipelining

To improve clock speed, use **AddPipelineRegisters** to use a pipelined adder tree rather than the default linear adder. You can also specify the number of pipeline stages before and after the multipliers. See “HDL Filter Architectures”.

## Subsystem Optimizations

This block can participate in subsystem-level optimizations such as sharing, streaming, and pipelining. For the block to participate in subsystem-level optimizations, set the **Architecture** to Fully parallel. See “Subsystem Optimizations for Filters”.

## HDL Filter Properties

### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

### ChannelSharing

For a multichannel filter, generate a single filter implementation to be shared between channels. See also ChannelSharing.

**CoeffMultipliers**

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See also `CoeffMultipliers`.

**DALUTPartition**

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also `DALUTPartition`.

**DARadix**

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ( $2^3$ ) generates a DA implementation that computes three sums at a time. The default value is  $2^1$ , which generates a fully serial DA implementation. See also `DARadix`.

**MultiplierInputPipeline**

Specify the number of pipeline stages to add at filter multiplier inputs. See also `MultiplierInputPipeline`.

**MultiplierOutputPipeline**

Specify the number of pipeline stages to add at filter multiplier outputs. See also `MultiplierOutputPipeline`.

**ReuseAccum**

Enable or disable accumulator reuse in a serial filter implementation. Set **ReuseAccum** to `on` to use a cascade-serial implementation. See also `ReuseAccum`.

**SerialPartition**

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also `SerialPartition`.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- HDL code generation is not supported for:
  - Unsigned input data.
  - Nonzero initial states. You must set **Initial states** to 0.
  - **Filter Structure**: Lattice MA.
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers** is hidden from the HDL Block Properties dialog box.

Programmable filters are not supported for:

- Architectures for which you specify the coefficients by dialog box parameters (for example, complex input and coefficients with serial architecture)
- distributed arithmetic (DA)
- **CoeffMultipliers** set to `csd` or `factored-csd`

Frame-based input filters are not supported for:

- Optional block-level reset and enable control signals

- Resettable and enabled subsystems
- Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals.
- Multichannel input
- Sharing and streaming optimizations

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

### Topics

Generate HDL Code for FIR Programmable Filter

**Introduced in R2014a**

# Discrete FIR Filter HDL Optimized

Model finite impulse response filter — HDL optimized (HDL Coder)

## Description

The Discrete FIR Filter HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Discrete FIR Filter HDL Optimized.

For FIR filters with complex coefficients, or with multichannel or frame-based inputs, use the Discrete FIR Filter block instead.

## HDL Architecture

The block provides three filter structures. The direct form systolic architecture provides a fully parallel implementation that makes efficient use of Intel and Xilinx DSP blocks. The direct form transposed architecture is a fully parallel implementation and is suitable for FPGA and ASIC applications. The partly serial systolic architecture provides a configurable serial implementation that also makes efficient use of FPGA DSP blocks. For a filter implementation that matches multipliers, pipeline registers, and pre-adders to the DSP configuration of your FPGA vendor, specify your target device when you generate HDL code.

All three structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters. The parallel implementations also remove the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

You can set block parameters to make tradeoffs between throughput and resource utilization.

- For highest throughput, choose a fully parallel systolic or transposed architecture. The generated code can accept input data and provides filtered output data on every cycle.
- For reduced area, choose partly serial systolic architecture. Then specify a rule that the block uses to serialize the filter based on either input timing or resource usage. To specify a serial filter using an input timing rule, set **Specify serialization factor as to**

Minimum number of cycles between valid input samples, and choose **Number of cycles** to be greater than or equal to 2. In this case, the filter accepts only input samples that are at least **Number of cycles** cycles apart. To specify a serial filter using a resource rule, set **Specify serialization factor as** to Maximum number of multipliers, and set **Number of multipliers** to be less than the number of filter coefficients. In this case, the filter accepts input samples that are at least NumCoeffs/NumMults apart.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- The Discrete FIR Filter HDL Optimized block does not support:
  - HDL code generation for floating-point input data types.
  - Complex coefficients.
  - Vector inputs. The block is sample based, accepting one scalar at a time.
  - Resource sharing optimization through HDL Coder. Instead, set the **Filter structure** to **Partly serial systolic**, and configure a serialization factor based on either input timing or resource usage.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017a**



# Discrete PID Controller

Simulate discrete-time PID controllers (HDL Coder)

## Description

The Discrete PID Controller block is available with Simulink.

For information about the simulation behavior and block parameters, see Discrete PID Controller.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

HDL code generation does not support the following settings:

- **Continuous-time.**
- **Filter method > Backward Euler or Trapezoidal.**
- **Source > external.**
- **External reset > rising, falling, either, or level.**
- If inputs are of type Double, **Anti-windup method > clamping.**

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Discrete Transfer Fcn

Implement discrete transfer function (HDL Coder)

## Description

The Discrete Transfer Fcn block is available with Simulink.

For information about the simulation behavior and block parameters, see Discrete Transfer Fcn.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## Restrictions

- You must use the **Inherit: Inherit via internal rule** option for data type propagation only if the input data type is double.
- Frame, matrix, and vector input data types are not supported.
- The leading denominator coefficient (a0) must be 1 or -1.

The Discrete Transfer Fcn block is excluded from the following optimizations:

- Resource sharing
- Distributed pipelining

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal (HDL Coder)

## Description

The Discrete-Time Integrator block is available with Simulink.

For information about the simulation behavior and block parameters, see Discrete-Time Integrator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## Restrictions

- State ports are not supported for HDL code generation. Clear the **Show state port** option.
- External initial conditions are not supported for HDL code generation. Set **Initial condition source** to `Internal`.
- **External Reset** must be set to `none`, `rising`, or `falling`.
- Width of input and output signals must not exceed 32 bits.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Display

Show value of input (HDL Coder)

## Description

The Display block is available with Simulink.

For information about the simulation behavior and block parameters, see [Display](#).

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Divide

Divide one input by another (HDL Coder)

### Description

The Divide block is available with Simulink. For information about the simulation behavior and block parameters, see Divide.

---

**Note** When you deploy the generated HDL code onto the target hardware, make sure that you set the **signed integer division rounds to** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to **Zero** or **Simplest**.

---

### HDL Architecture

To perform an HDL-optimized divide operation, connect a Product block to a Divide block in reciprocal mode. For information about the Divide block in reciprocal mode, see “Reciprocal Mode” on page 3-192.

### Default Mode

In default mode, the Divide block supports only integer data types for HDL code generation.

Architecture	Parameters	Description
default Linear	None	Generate a divide (/) operator in the HDL code.

### Reciprocal Mode

When **Number of Inputs** is set to /, the Divide block is in reciprocal mode.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

In reciprocal mode, the Divide block has the HDL block implementations described in the following table.

Architectures	Parameters	Additional cycles of latency	Description
default Linear	None	0	When you compute a reciprocal, use the HDL divide (/) operator to implement the division.
ReciprocalRsqrBasedNewton	Iterations	Signed input: Iterations + 5  Unsigned input: Iterations + 3	Use the iterative Newton method. Select this option to optimize area.  The default value for Iterations is 3.  The recommended value for Iterations is between 2 and 10. If Iterations is outside the recommended range, HDL Coder displays a message.

Architectures	Parameters	Additional cycles of latency	Description
ReciprocalRsqrBasedNewtonSingleRate	Iterations	Signed input: (Iterations * 4) + 8  Unsigned input: (Iterations * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.  The default value for Iterations is 3.  The recommended value for Iterations is between 2 and 10. If Iterations is outside the recommended range, the coder displays a message.

The Newton-Raphson iterative method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i(1.5 - 0.5ax_i^2)$$

ReciprocalRsqrBasedNewton and ReciprocalRsqrBasedNewtonSingleRate implement the Newton-Raphson method with:

$$f(x) = \frac{1}{x^2} - 1$$

---

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

#### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

#### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### **NFPCustomLatency**

To specify a value, set **LatencyStrategy** to Custom. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### **MantissaMultiplyStrategy**

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

### **DivisionAlgorithm**

Specify whether to use the Radix-2 or Radix-4 algorithm to perform the floating-point division. The Radix-2 mode offers a trade-off between latency and frequency. The Radix-4 mode offers a trade-off between latency and resource usage. For more information, see “DivisionAlgorithm”.

## **Complex Data Support**

This block does not support code generation for division with complex signals.

## **Restrictions**

When you use the Divide block in reciprocal mode, the following restrictions apply:

- The input must be scalar and must have integer or fixed-point (signed or unsigned) data type.
- The output must be scalar and have integer or fixed-point (signed or unsigned) data type.
- Only the Zero rounding mode is supported.
- You must select the **Saturate on integer overflow** option on the block.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## DocBlock

Create text that documents model and save text with model (HDL Coder)

### Description

The DocBlock block is available with Simulink.

For information about the simulation behavior and block parameters, see DocBlock.

### HDL Architecture

Architecture	Description
Annotation (default)	Insert text as comment in the generated code.
HDLText	Integrate text as custom HDL code.
No HDL	Do not generate HDL code for this block.

### HDL Block Properties

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



**TargetLanguage**

Language of the text, either Verilog or VHDL. The default is VHDL.

When **Architecture** is HDLText, this property is available. To learn more, see “Integrate Custom HDL Code Using DocBlock”.

**Restrictions**

- **Document type** must be Text.

HDL Coder does not support the HTML or RTF options.

- You can have a maximum of two DocBlock blocks with **Architecture** set to HDLText in the same subsystem.

If you have two DocBlock blocks, one must have **TargetLanguage** set to VHDL, and the other must have **TargetLanguage** set to Verilog. When generating code, HDL Coder only integrates the custom code from the DocBlock that matches the target language for code generation.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**See Also****Topics**

“Generate Code with Annotations or Comments”

“Integrate Custom HDL Code Using DocBlock”

**Introduced in R2014a**

# Dot Product

Generate dot product of two vectors (HDL Coder)

## Description

The Dot Product block is available with Simulink.

For information about the simulation behavior and block parameters, see Dot Product.

## HDL Architecture

Architecture	Description
Linear (default)	Generates a linear chain of adders to compute the sum of products.
Tree	Generates a tree structure of adders to compute the sum of products.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Downsample

Resample input at lower rate by deleting samples (HDL Coder)

## Description

The Downsample block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Downsample.

## Best Practices

It is good practice to follow the Downsample block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation”.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

- **Input processing** set to Columns as channels (frame based) is not supported.
- For **Input processing** set to Elements as channels (sample based), select Allow multirate processing. With this setting, if **Sample offset** is set to 0, **Initial conditions** has no effect on generated code.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

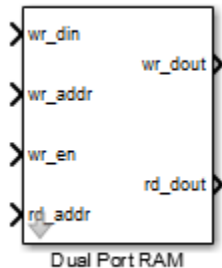
### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Dual Port RAM

Dual port RAM with two output ports



## Library

HDL Coder / HDL Operations

## Description

The Dual Port RAM block models a RAM that supports simultaneous read and write operations, and has both a read data output port and write data output port. You can use this block to generate HDL code that maps to RAM in most FPGAs.

If you do not need to use the write output data, `wr_dout`, you can achieve better RAM inference with synthesis tools by using the Simple Dual Port RAM block.

## Read-During-Write Behavior

During a write, new data appears at the output of the write port (`wr_dout`) of the Dual Port RAM block. If a read operation occurs simultaneously at the same address as a write operation, old data appears at the read output port (`rd_dout`).

## Parameters

### Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

## Ports

The block has the following ports:

### wr\_din

Write data input. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

### wr\_addr

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

### wr\_en

Write enable.

Data type: Boolean

### rd\_addr

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

### wr\_dout

Output data from write address, `wr_addr`.

### rd\_dout

Output data from read address, `rd_addr`.



## HDL Architecture

This block has a single, default HDL architecture.

HDL code generated for RAM blocks has:

- A latency of one clock cycle for read data output.
- No reset signal, because some synthesis tools do not infer a RAM from HDL code if it includes a reset.

Code generation for a RAM block creates a separate file, *blockname.ext*. *blockname* is derived from the name of the RAM block. *ext* is the target language file name extension.

## RAM Initialization

Code generated to initialize a RAM is intended for simulation only. Synthesis tools can ignore this code.

## Implement RAM With or Without Clock Enable

The HDL block property, `RAMArchitecture`, enables or suppresses generation of clock enable logic for all RAM blocks in a subsystem. You can set `RAMArchitecture` to the following values:

- `WithClockEnable` (default): Generates RAMs using HDL templates that include a clock enable signal, and an empty RAM wrapper.
- `WithoutClockEnable`: Generates RAMs without clock enables, and a RAM wrapper that implements the clock enable logic.

Some synthesis tools do not infer RAMs with a clock enable. If your synthesis tool does not support RAM structures with a clock enable, and cannot map your generated HDL code to FPGA RAM resources, set `RAMArchitecture` to `'WithoutClockEnable'`. To learn how to generate RAMs without clock enables for your design, see the Getting Started with RAM and ROM example. To open the example, at the command prompt, enter:

```
hdlcoderramrom
```

## RAM Inference Limitations

If you use RAM blocks to perform concurrent read and write operations, verify the read-during-write behavior in hardware. The read-during-write behavior of the RAM blocks in Simulink matches that of the generated behavioral HDL code. However, if a synthesis tool does not follow the same behavior during RAM inference, it causes the read-during-write behavior in hardware to differ from the behavior of the Simulink model or generated HDL code.

Your synthesis tool might not map the generated code to RAM for the following reasons:

- Small RAM size: your synthesis tool uses registers to implement a small RAM for better performance.
- A clock enable signal is present. You can suppress generation of a clock enable signal in RAM blocks, as described in “Implement RAM With or Without Clock Enable” on page 3-207.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

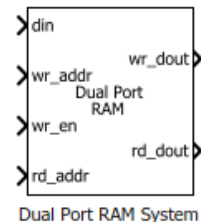
Dual Rate Dual Port RAM | Simple Dual Port RAM | Single Port RAM

**Introduced in R2014a**

## Dual Port RAM System

Dual Port RAM block based on the `hdl.RAM` system object with ability to provide initial value

**Library:** HDL Coder / HDL RAMs



## Description

The blocks are MATLAB System blocks that use the `hdl.RAM` System object™. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Dual Port RAM System block behaves similar to the Dual Port RAM, the Single Port RAM System behaves similar to the Single Port RAM, and so on. With the MATLAB System blocks, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.
- Obtain higher performance and support for large data memories.

## Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.
- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

## Ports

### Input

#### **din** — Write data input

Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be double, single, integer, or a fixed-point (`fi`) object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

#### **addr** — Write or Read address

Scalar (default) | Vector

Address that you write the data into when `wrEn` is true. The RAM reads the value in memory location **addr** when `wrEn` is false. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Single port`.

Data Types: `uint8` | `uint16` | `fixed point`

#### **wr\_addr** — Write address

Scalar (default) | Vector

RAM address that you write the data into. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

#### **wr\_en** — Write enable

Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location **addr** when `wrEn` is false.

Data Types: Boolean

### **rd\_addr — Read address**

Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either fixed-point (fi) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

Data Types: uint8 | uint16 | fixed point

## **Output**

### **dout — Output data**

Scalar (default) | Vector

Output data that the RAM reads from the memory location `addr` when `wrEn` is false.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Single port.

### **rd\_dout — Read data**

Scalar (default) | Vector

Old output data that the RAM reads from the memory location `rd_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

### **wr\_dout — Write data output**

Scalar (default) | Vector

New or old output data that the RAM reads from the memory location `wr_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Dual port.

## Parameters

### Specify the type of RAM — RAM type

Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- `Single port` — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- `Simple dual port` — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- `Dual port` — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

### Specify the output data for a write operation — Write output behavior

New data (default) | Old data

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `'Old data'` — Send out old data at the address to the output.

### Specify the RAM initial value — Initial simulation output of RAM

'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAMSystem` object.

#### HDL Block Properties

##### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

##### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

##### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### Complex Data Support

This block supports code generation for complex signals.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.



## See Also

### System Objects

hdl.RAM

### Blocks

Simple Dual Port RAM System | Single Port RAM System

### Topics

“”

“”

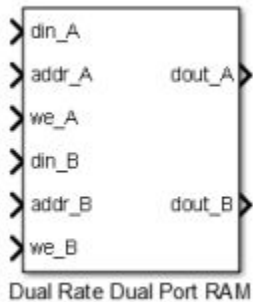
“Implement RAM Using MATLAB Code”

“HDL Code Generation for System Objects”

**Introduced in R2017b**

## Dual Rate Dual Port RAM

Dual Port RAM that supports two rates



## Library

HDL Coder / HDL Operations

## Description

The Dual Rate Dual Port RAM block models a RAM that supports simultaneous read and write operations to different addresses at two clock rates. Port A of the RAM can run at one rate, and port B can run at a different rate.

In high-performance hardware applications, you can use this block to access the RAM twice per clock cycle. If you generate HDL code, this block maps to a dual-clock dual-port RAM in most FPGAs.

## Simultaneous Access

You can access different addresses from ports A and B simultaneously. You can also read the same address from ports A and B simultaneously.

However, do not access an address from one RAM port while it is being written from the other RAM port. During simulation, if you access an address from one RAM port at the same time as you write that address from the other RAM port, the software reports an error.

## Read-During-Write Behavior

The RAM has write-first behavior. When you write to the RAM, the new write data is immediately available at the output port.

## Parameters

### Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 28. The default value is 8.

## Ports

The block has the following ports:

### `din_A`

Write data input for RAM port A. The data can be any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

### `addr_A`

Write address for RAM port A.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

### `we_A`

Write enable for RAM port A. Set `we_A` to `true` for a write operation, or `false` for a read operation.

Data type: Boolean

din\_B

Write data input for RAM port B. The data can be of any width, and inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

addr\_B

Write address for RAM port B.

Data type: scalar unsigned integer (uintN) or unsigned fixed point (ufixN) with a fraction length of 0

we\_B

Write enable for RAM port B. Set we\_B to true for a write operation, or false for a read operation.

Data type: Boolean

dout\_A

Output data from RAM port A address, addr\_A.

dout\_B

Output data from RAM port B address, addr\_B.

## HDL Architecture

---

**Note** For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

---

This block has a single, default HDL architecture.

HDL code generated for RAM blocks has:

- A latency of one clock cycle for read data output.
- No reset signal, because some synthesis tools do not infer a RAM from HDL code if it includes a reset.

Code generation for a RAM block creates a separate file, *blockname.ext*. *blockname* is derived from the name of the RAM block. *ext* is the target language file name extension.

## RAM Initialization

Code generated to initialize a RAM is intended for simulation only. Synthesis tools can ignore this code.

## Implement RAM With or Without Clock Enable

The HDL block property, `RAMArchitecture`, enables or suppresses generation of clock enable logic for all RAM blocks in a subsystem. You can set `RAMArchitecture` to the following values:

- `WithClockEnable` (default): Generates RAM using HDL templates that include a clock enable signal, and an empty RAM wrapper.
- `WithoutClockEnable`: Generates RAM without clock enables, and a RAM wrapper that implements the clock enable logic.

Some synthesis tools do not infer RAM with a clock enable. If your synthesis tool does not support RAM structures with a clock enable, and cannot map your generated HDL code to FPGA RAM resources, set `RAMArchitecture` to `WithoutClockEnable`.

## RAM Inference Limitations

If you use RAM blocks to perform concurrent read and write operations, verify the read-during-write behavior in hardware. The read-during-write behavior of the RAM blocks in Simulink matches that of the generated behavioral HDL code. However, if a synthesis tool does not follow the same behavior during RAM inference, it causes the read-during-write behavior in hardware to differ from the behavior of the Simulink model or generated HDL code.

Your synthesis tool might not map the generated code to RAM for the following reasons:

- Small RAM size: your synthesis tool uses registers to implement a small RAM for better performance.
- A clock enable signal is present. You can suppress generation of a clock enable signal in RAM blocks, as described in “Implement RAM With or Without Clock Enable” on page 3-219.

# HDL Block Properties

## ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

## InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

## OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

# Complex Data Support

This block supports code generation for complex signals.

# Extended Capabilities

## HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

Dual Port RAM | HDL FIFO | Simple Dual Port RAM | Single Port RAM

**Introduced in R2014a**

# Edge Detector

Find edges of objects in image (HDL Coder)

## Description

The Edge Detector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Edge Detector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**



# Enable

Add enabling port to system (HDL Coder)

## Description

The Enable block is available with Simulink.

For information about the simulation behavior and block parameters, see [Enable](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “[ConstrainedOutputPipeline](#)”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[InputPipeline](#)”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[OutputPipeline](#)”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Enabled Subsystem

**Introduced in R2014a**

# Enabled Subsystem

Represent subsystem whose execution is enabled by external input (HDL Coder)

## Description

An enabled subsystem is a subsystem that receives a control signal via an Enable block. The enabled subsystem executes at each simulation step where the control signal has a positive value.

For detailed information on how to construct and configure enabled subsystems, see “Using Enabled Subsystems” (Simulink).

## Best Practices

When using enabled subsystems in models targeted for HDL code generation, it is good practice to consider the following:

- For synthesis results to match Simulink results, the Enable port must be driven by registered logic (with a synchronous clock) on the FPGA.
- Put unit delays on Enabled Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- Enabled subsystems can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of enabled subsystem instances and the number of output ports per subsystem.

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.

Architecture	Description
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

#### DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

**FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

**Restrictions**

HDL Coder supports HDL code generation for enabled subsystems that meet the following conditions:

- The enabled subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The enable signal is a scalar.
- The data type of the enable signal is either `boolean` or `ufix1`.
- Outputs of the enabled subsystem have an initial value of 0.

- All inputs and outputs of the enabled subsystem (including the enable signal) run at the same rate.
- The **Show output port** parameter of the Enable block is set to Off.
- The **States when enabling** parameter of the Enable block is set to held (i.e., the Enable block does not reset states when enabled).
- The **Output when disabled** parameter for the enabled subsystem output ports is set to held (i.e., the enabled subsystem does not reset output values when disabled).
- If the DUT contains the following blocks, RAMArchitecture is set to WithClockEnable:
  - Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- The enabled subsystem does not contain the following blocks:
  - CIC Decimation
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation
  - Downsample
  - Upsample
  - HDL Cosimulation blocks for HDL Verifier
  - Rate Transition

### Example

The Automatic Gain Controller example shows how you can use enabled subsystems in HDL code generation. To open the example, enter:

```
hdlcoder_agc
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Enable | Subsystem

**Introduced in R2014a**

## Enabled Synchronous Subsystem

Represent enabled subsystem that has synchronous reset and enable behavior



## Library

HDL Coder / HDL Subsystems

## Description

An Enabled Synchronous Subsystem is an Enabled Subsystem that uses the Synchronous mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create an Enabled Synchronous Subsystem block, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to Synchronous inside an Enabled subsystem.

For more information, see State Control and “Using Enabled Subsystems” (Simulink).

## Data Type Support

See Inport for information on the data types accepted by a subsystem's input ports. See Outport for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” (Simulink) in the Simulink documentation.



## Parameters

### Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

#### Settings

**Default:** FromPortIcon

none

Does not display port labels on the subsystem block.

FromPortIcon

If the corresponding port icon displays a signal name, display the signal name on the subsystem block. Otherwise, display the port block's name.

FromPortBlockName

Display the name of the corresponding port block on the subsystem block.

SignalName

If a name exists, display the name of the signal connected to the port on the subsystem block; otherwise, the name of the corresponding port block.

#### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Read/Write permissions

Control user access to the contents of the subsystem.

#### Settings

**Default:** ReadWrite

ReadWrite

Enables opening and modification of subsystem contents.

### ReadOnly

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

### NoReadOrWrite

Disables opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Name of error callback function

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

### Settings

**Default:** ' '

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

### Settings

**Default:** All

**All**

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

**ExplicitOnly**

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

**None**

Do not resolve any workspace variable names.

**Command-Line Information**

See “Block-Specific Parameters” (Simulink) for the command-line information.

**Treat as atomic unit**

Causes Simulink software to treat the subsystem as a unit when determining the execution order of block methods.

**Settings**

**Default:** Off

On

Cause Simulink software to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink software to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

### Dependencies

This parameter enables:

- **Minimize algebraic loop occurrences**
- **Sample time**
- **Function packaging** (requires a Simulink Coder license)

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

### Settings

**Default:** On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

### Dependency

**Treat as grouped when propagating variant conditions** enables this parameter.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

### Settings

**Default:** Auto

#### Auto

Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

#### Inline

Simulink Coder software inlines the subsystem unconditionally.

#### Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

#### Reusable function

Simulink Coder software generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Characteristics

Data Types	Double   Single   Boolean   Base Integer   Fixed-Point   Enumerated   Bus
------------	---

Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.  The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

**ClockRatePipelining**

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

**ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

**DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

**FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the **IP Core Generation** workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

### **ProcessorFPGASynchronization**

Processor/FPGA synchronization mode, specified as a character vector.

To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: Free running (default) | Coprocessing - blocking

Example: 'Free running'

### **TestPointMapping**

To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'TestPoint', 'AXI4-Lite', 'x"108"'}}

### **TunableParameterMapping**

To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'myParam', 'AXI4-Lite', 'x"108"'}}

### **AXI4RegisterReadback**

To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'off' (default) | 'on'



### **GenerateDefaultAXI4Slave**

To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'on' (default) | 'off'

### **IPCoreAdditionalFiles**

Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).

You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.

Values: '' (default) | character vector

Example: 'C:\myprojfiles\led\_blinking\_file1.vhd;C:\myprojfiles\led\_blinking\_file2.vhd;'

### **IPCoreName**

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: '' (default) | character vector

Example: 'my\_model\_name'

### **IPCoreVersion**

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: '' (default) | character vector

Example: '1.3'

### Restrictions

- Your DUT cannot be an Enabled Synchronous Subsystem.
- You cannot have a Delay block with an external reset port inside the subsystem.
- You cannot generate HDL for the Turbo Decoder block inside an Enabled Synchronous Subsystem.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

Enable | Resettable Synchronous Subsystem | State Control | Synchronous Subsystem

### Topics

“”

“”

“Synchronous Subsystem Behavior with the State Control Block”

**Introduced in R2016a**

# Enumerated Constant

Generate enumerated constant value (HDL Coder)

## Description

The Enumerated Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Enumerated Constant.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Erosion

Morphological erode of binary pixel data (HDL Coder)

## Description

The Erosion block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Erosion.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Error Rate Calculation

Compute bit error rate or symbol error rate of input data (HDL Coder)

## Description

The Error Rate Calculation block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Error Rate Calculation.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Extract Bits

Output selection of contiguous bits from input signal (HDL Coder)

## Description

The Extract Bits block is available with Simulink.

For information about the simulation behavior and block parameters, see Extract Bits.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Eye Diagram

Display multiple traces of modulated signal (HDL Coder)

## Description

The Eye Diagram block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Eye Diagram.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

# Extended Capabilities

## C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# FFT HDL Optimized

Fast Fourier transform—optimized for HDL code generation (HDL Coder)

## Description

The FFT HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see FFT HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

- If you use the FFT HDL Optimized block with the State Control block inside an Enabled Subsystem, the optional reset port is not supported. If you enable the reset port on the FFT HDL Optimized block in such a subsystem, the model will error on Update Diagram.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# FIR Decimation

Filter and downsample input signals (HDL Coder)

## Description

The FIR Decimation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see FIR Decimation.

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**. Programmable coefficients are not supported.

## Frame-Based Input Support

HDL Coder supports the use of vector inputs to FIR Decimation blocks, where each element of the vector represents a sample in time. You can use an input vector of up to 512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types. The output is a column vector of reduced size, corresponding to your decimation factor. You can use real input signals with real coefficients, complex input signals with real coefficients, or real input signals with complex coefficients.

- 1 Connect a column vector signal to the FIR Decimation block input port.
- 2 Specify **Input processing** as Columns as channels (frame based).
- 3 Set **Rate options** to Enforce single-rate processing.
- 4 Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to Frame Based. The block implements a parallel HDL architecture. See “Frame-Based Architecture”.

## HDL Architecture

To reduce area or increase speed, the FIR Decimator block supports block-level optimizations.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

## Block Optimizations

### Serial Architectures

To use block-level optimizations to reduce hardware resources, set **Architecture** to **Fully Serial** or **Partly Serial**. See “HDL Filter Architectures”.

When you specify **SerialPartition** for a FIR Decimator block, set **Filter structure** to **Direct** form. The **Direct** form transposed structure is not supported with serial architectures. Accumulator reuse is not supported for FIR Decimation filters.

### Distributed Arithmetic

To minimize multipliers by replacing them with LUTs and shift registers, use a distributed arithmetic (DA) filter implementation. See “Distributed Arithmetic for HDL Filters”.

When you select the **Distributed Arithmetic (DA)** architecture and use the **DALUTPartition** and **DARadix** distributed arithmetic properties, set **Filter structure** to **Direct** form. The **Direct** form transposed structure is not supported with distributed arithmetic.

### Pipelining

To improve clock speed, use **AddPipelineRegisters** to use a pipelined adder tree rather than the default linear adder. This option is supported for **Direct** form architecture. You can also specify the number of pipeline stages before and after the multipliers. See “HDL Filter Architectures”.

## HDL Filter Properties

### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also `AddPipelineRegisters`.

### CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully

parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See also `CoeffMultipliers`.

### **DALUTPartition**

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also `DALUTPartition`.

### **DARadix**

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ( $2^3$ ) generates a DA implementation that computes three sums at a time. The default value is  $2^1$ , which generates a fully serial DA implementation. See also `DARadix`.

### **MultiplierInputPipeline**

Specify the number of pipeline stages to add at filter multiplier inputs. See also `MultiplierInputPipeline`.

### **MultiplierOutputPipeline**

Specify the number of pipeline stages to add at filter multiplier outputs. See also `MultiplierOutputPipeline`.

### **SerialPartition**

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also `SerialPartition`.

## **HDL Block Properties**

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “`ConstrainedOutputPipeline`”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “`InputPipeline`”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - Slope and Bias scaling
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers** is hidden from the HDL Block Properties dialog box.
- Frame-based input filters are not supported for:
  - Resettable and enabled subsystems
  - Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals.
  - Sharing and streaming optimizations

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.



## **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## FIR Interpolation

Upsample and filter input signals (HDL Coder)

### Description

The FIR Interpolation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see FIR Interpolation.

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**.

### HDL Architecture

When you select Fully Serial architecture, the `SerialPartition` property is set on the FIR Interpolation Block.

### Distributed Arithmetic Support

Distributed Arithmetic properties **DALUTPartition** and **DARadix** are supported for the following filter structures.

Architecture	Supported FIR Structures
Distributed Arithmetic (DA)	default

### AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on the filter structure. The pipeline register placement determines the latency.

<b>Pipeline Register Placement</b>	<b>Latency (clock cycles)</b>
A pipeline register is added between levels of a tree-based adder.	$\text{ceil}(\log_2(PL)) - 1$ . PL is polyphase filter length.

## HDL Filter Properties

### AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

### CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See also CoeffMultipliers.

### DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition.

### DARadix

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ( $2^3$ ) generates a DA implementation that computes three sums at a time. The default value is  $2^1$ , which generates a fully serial DA implementation. See also DARadix.

### MultiplierInputPipeline

Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline.

### MultiplierOutputPipeline

Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline.

### SerialPartition

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - **Coefficients**: Slope and Bias scaling
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers** is hidden from the HDL Block Properties dialog box.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# FIR Rate Conversion HDL Optimized

Upsample, filter, and downsample input signals—optimized for HDL code generation (HDL Coder)

## Description

The FIR Rate Conversion HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see FIR Rate Conversion HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015b**

# Floating Scope

Display signals generated during simulation (HDL Coder)

## Description

The Floating Scope block is available with Simulink.

For information about the simulation behavior and block parameters, see Floating Scope.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

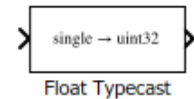
**Introduced in R2014a**



# Float Typecast

Typecast a floating-point type to an unsigned integer or vice versa

**Library:** HDL Coder / HDL Floating Point Operations

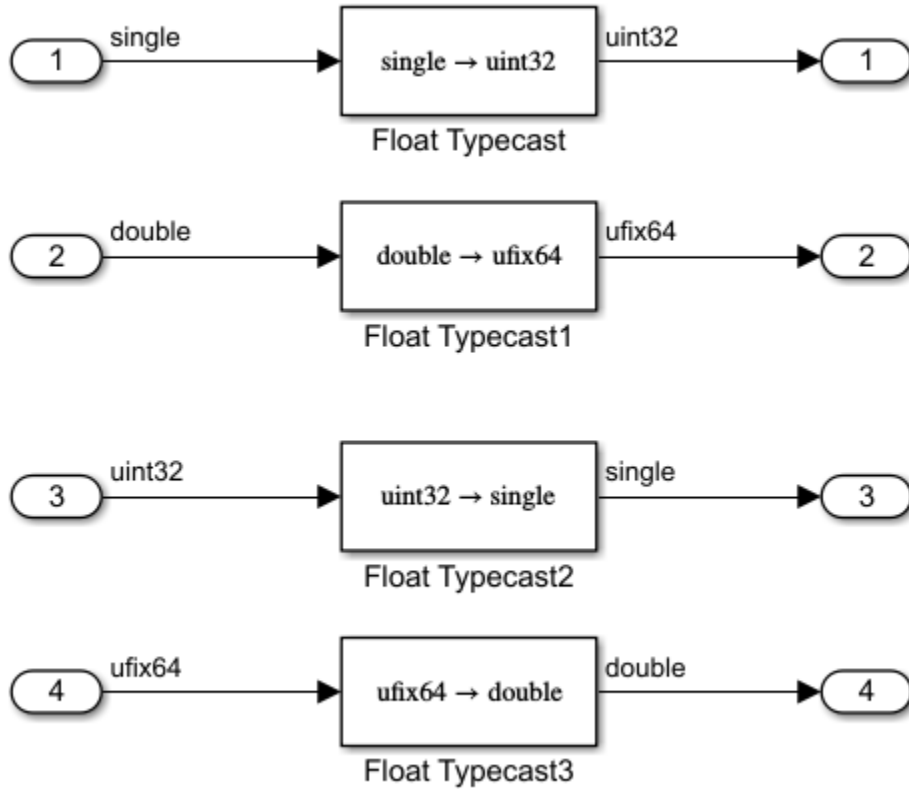


## Description

The block casts the underlying bits of the input to the corresponding fixed-point or floating point representation. The input and output of the block contain the same number of bits.

Input Data Type	Output Data Type
single	uint32
double	ufix64
uint32	single
ufix64	double

This figure shows how the block mask, behavior, and output data type changes dynamically depending on the input data type that you specify.



## Ports

### Input

**Port\_1(u) – Input signal**

scalar | vector

Port to provide input to the block.

Data Types: single | double | uint32 | fixed point

## Output

### Port\_1(y) — Output signal

scalar | vector

Port to obtain calculated output from the block.

Data Types: single | double | uint32 | fixed point

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

The block supports HDL code generation in the **Native Floating Point** mode. To use this mode, specify `single` or `uint32` data types as input to the block. With the HDL Model Checker, you can replace Data Type Conversion blocks that use the **Stored Integer (SI)** mode and convert between floating-point and fixed-point data types.

The block supports code generation for complex signals.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

## See Also

### Functions

typecast

## **Topics**

“Getting Started with HDL Coder Native Floating-Point Support”

**Introduced in R2017b**

## For Each Subsystem

Repeatedly perform algorithm on each element or subarray of input signal and concatenate results (HDL Coder)

### Description

To repeat the same algorithm for each element or subarray of the input signals, use the For Each Subsystem block. The block reduces simulation time because it processes individual elements or subarrays of the input signals simultaneously. For information about the simulation behavior and block parameters, see For Each Subsystem.

By using the For Each block inside the For Each Subsystem, you can specify how to partition elements of the input signals. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which to slice the input signal and the width of each slice respectively. To partition a row vector, specify the **Partition Dimension** as 2. To partition a column vector, specify the **Partition Dimension** as 1. To learn more about the block parameters, see For Each.

When you generate HDL code for the For Each Subsystem, the code generator uses a for-generate loop that iterates through elements of the input and output signals. The for-generate loop improves readability and reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.

### Limitations

- You cannot use the For Each Subsystem block as the DUT.
- You cannot partition mask parameters of the For Each Subsystem for HDL code generation.

### HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.

Architecture	Description
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

**DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

**FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

**Complex Data Support**

The block does not support complex data signals for HDL code generation. To input complex signals, you can convert this signal to an array of signals, and then input to the block. To learn more, see “Generate HDL Code for Blocks Inside For Each Subsystem”

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017a**



# Frame Conversion

Specify sampling mode of output signal (HDL Coder)

## Description

The Frame Conversion block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see [Frame Conversion](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “[ConstrainedOutputPipeline](#)”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[InputPipeline](#)”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[OutputPipeline](#)”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## From

Accept input from Goto block (HDL Coder)

## Description

The From block is available with Simulink.

For information about the simulation behavior and block parameters, see From.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Gain

Multiply input by constant (HDL Coder)

### Description

The Gain block is available with Simulink.

For information about the simulation behavior and block parameters, see Gain.

### Tunable Parameters

You can use a tunable parameter in a Gain block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

### HDL Architecture

ConstMultiplierOptimization	Description
none( <i>Default</i> )	By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
csd	When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations.  CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.

<b>ConstMultiplierOptimization</b>	<b>Description</b>
<code>fcsd</code>	This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction. You can achieve a greater area reduction with FCSD at the cost of decreasing clock speed.
<code>auto</code>	When you specify this option, the coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. When you specify <code>auto</code> , the coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).

## HDL Block Properties

### General

#### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

---

**Note** For certain values of the **Gain** parameter, native floating point implements the algorithm differently instead of using multipliers. For example, if you set the **Gain** parameter to 1, the generated model uses a wire to pass the input to the output. If you set the **Gain** parameter to -1, the generated model shows a Unary Minus block that inverts the polarity of the input signal. This implementation reduces the latency and resource usage on the target platform.

---

You can specify these settings in the **Native Floating Point** tab for the Gain block.

### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Gamma Corrector

Apply or remove gamma correction (HDL Coder)

## Description

The Gamma Corrector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Gamma Corrector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# General CRC Generator HDL Optimized

Generate CRC code bits and append to input data, optimized for HDL code generation (HDL Coder)

## Description

The General CRC Generator HDL Optimized block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see General CRC Generator HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# General CRC Syndrome Detector HDL Optimized

Detect errors in input data using CRC (HDL Coder)

## Description

The General CRC Syndrome Detector HDL Optimized block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see General CRC Syndrome Detector HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# General Multiplexed Deinterleaver

Restore ordering of symbols using specified-delay shift registers (HDL Coder)

## Description

The General Multiplexed Deinterleaver block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see General Multiplexed Deinterleaver.

## HDL Architecture

The implementation for the General Multiplexed Deinterleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to `none`.

When you set `ResetType` to `none`, reset is not applied to the shift registers. When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# General Multiplexed Interleaver

Permute input symbols using set of shift registers with specified delays (HDL Coder)

## Description

The General Multiplexed Interleaver block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see General Multiplexed Interleaver.

## HDL Architecture

The implementation for the General Multiplexed Interleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to 'none'.

When you set `ResetType` to 'none', reset is not applied to the shift registers. Mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface.)

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also "InputPipeline".

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Gold Sequence Generator

Generate Gold sequence (HDL Coder)

## Description

The Gold Sequence Generator block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Gold Sequence Generator.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018a**

## Goto

Pass block input to From blocks (HDL Coder)

### Description

The Goto block is available with Simulink.

For information about the simulation behavior and block parameters, see Goto.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Grayscale Closing

Morphological close of grayscale pixel data (HDL Coder)

## Description

The Grayscale Closing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Closing.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016a**



# Grayscale Dilation

Morphological dilate of grayscale pixel data (HDL Coder)

## Description

The Grayscale Dilation block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Dilation.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016a**

# Grayscale Erosion

Morphological erode of grayscale pixel data (HDL Coder)

## Description

The Grayscale Erosion block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Erosion.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016a**

# Grayscale Opening

Morphological open of grayscale pixel data (HDL Coder)

## Description

The Grayscale Opening block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Opening.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016a**

# Ground

Ground unconnected input port (HDL Coder)

## Description

The Ground block is available with Simulink.

For information about the simulation behavior and block parameters, see [Ground](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “[ConstrainedOutputPipeline](#)”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[OutputPipeline](#)”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# HDL Cosimulation

Cosimulate hardware component by communicating with HDL module instance executing in HDL simulator (HDL Coder)

## Description

The HDL Cosimulation block is available with HDL Verifier.

For information about the simulation behavior and block parameters, see HDL Cosimulation.

HDL Coder supports HDL code generation for the following HDL Cosimulation blocks:

- HDL Verifier for use with Mentor Graphics® ModelSim®
- HDL Verifier for use with Cadence Incisive®

Each of the HDL Cosimulation blocks cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model that executes under an HDL simulator.

For information about timing, latency, data typing, frame-based processing, and other issues when setting up an HDL cosimulation, see “Define HDL Cosimulation Block Interface” (HDL Verifier).

You can use an HDL Cosimulation block with HDL Coder to generate an interface to your manually written or legacy HDL code. When an HDL Cosimulation block is included in a model, the coder generates a VHDL or Verilog interface, depending on the selected target language.

When the target language is VHDL, the generated interface includes:

- An entity definition. The entity defines ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. Clock enable and reset ports are also declared.
- An RTL architecture including a component declaration, a component configuration declaring signals corresponding to signals connected to the HDL Cosimulation ports, and a component instantiation.

- Port assignment statements as required by the model.

When the target language is Verilog, the generated interface includes:

- A module defining ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. The module also defines clock enable and reset ports, and wire declarations corresponding to signals connected to the HDL Cosimulation ports.
- A module instance.
- Port assignment statements as required by the model.

Before initiating code generation, to check the requirements for using the HDL Cosimulation block for code generation, select **Simulation > Update Diagram**.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

For implementation parameter descriptions, see “Customize Black Box or HDL Cosimulation Interface”.

## See Also

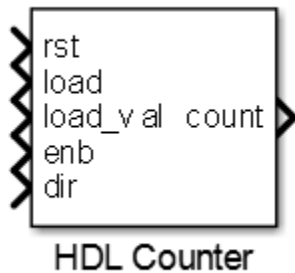
### Topics

“Generate a Cosimulation Model”

**Introduced in R2014a**

## HDL Counter

Free-running or count-limited hardware counter



## Library

HDL Coder / HDL Operations

## Description

The HDL Counter block models a free-running or count-limited hardware counter that supports signed and unsigned integer and fixed-point data types.

The counter emits its value for the current sample time.

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the `Simulink.restoreDiagnostic` reference page. The block does report errors due to wrap on overflow.

## Control Ports

By default, the counter does not have input ports. Optionally, you can add control ports that enable, disable, load, reset or set the direction of the counter.

The table shows the priority of the control signals and how the counter value is updated in relation to the control signals.

Local reset, rst	Load trigger, load	Count enable, enb	Count direction, dir	Next Counter Value
1	-	-	-	initial value
0	1	-	-	load_val value
0	0	0	-	current value
0	0	1	1	current value + step value
0	0	1	0	current value - step value

## Count direction

The **Step value** parameter and optional count direction port, `dir`, interact to determine the actual count direction.

dir Signal Value	Step Value Sign	Actual Count Direction
1	+ (positive)	Up
1	- (negative)	Down
0	+ (positive)	Down
0	- (negative)	Up

## Parameters

### Counter type

Counter behavior.

- **Free running** (default): The counter continues to increment or decrement by the **Step value** until reset.
- **Count limited**: The counter increments or decrements by the **Step value** until it is exactly equal to the **Count to value**.

### Initial value

Counter value after reset. The default is 0.

**Step value**

Value added to counter at each sample time. The default is 1.

**Count to value**

When the count is exactly equal to **Count to value**, the count restarts at the **Initial value**. This option is available when **Counter type** is set to `Count limited`. The default is 100.

**Count from**

Specifies the parameter that sets the start value after rollover. When set to `Specify`, the **Count from value** parameter is the start value after rollover. The default is `Initial value`.

**Count from value**

Counter value after rollover when **Count from** is set to `Specify`. The default is 0.

**Local reset port**

When selected, creates a local reset port, `rst`.

**Load ports**

When selected, creates a load data port, `load_val`, and load trigger port, `load`.

**Count enable port**

When selected, creates a count enable port, `enb`.

**Count direction port**

When selected, creates a count direction port, `dir`.

**Counter output data is**

Output data type signedness. The default is `Unsigned`.

**Word length**

Bit width, including sign bit, for an integer counter; word length for a fixed-point data type counter. The minimum value if Output data type is `Unsigned` is 1, 2 if `Signed`. The maximum value is 125. The default is 8.

**Fraction length**

Fixed-point data type fraction length. The default is 0.

**Sample time**

Sample time. The default is 1.

This parameter is not available, and the block inherits its sample time from the input ports when any of these parameters is selected:

- **Local reset port**
- **Load ports**
- **Count enable port**
- **Count direction port**

## Ports

The block has the following ports:

`rst`

Resets the counter value. Active-high.

This port is available when you select **Local reset port**.

Data type: Boolean

`load`

Sets the counter to the load value, `load_val`. Active-high.

This port is available when you select **Load ports**.

Data type: Boolean

`load_val`

Data value to load.

This port is available when you select **Load ports**.

Data type: Same as count.

`enb`

Enables counter operation. Active-high.

This port is available when you select **Count enable port**.

Data type: Boolean

`dir`

Count direction. This port interacts with **Step value** to determine count direction.

- **1: Step value** is added to the current counter value to compute the next value.
- **0: Step value** is subtracted from the current counter value to compute the next value.

This port is available when you select **Count direction port**.

Data type: Boolean

count

Counter value.

Data type: Determined automatically based on **Counter output data is**, **Word length**, and **Fraction length**.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

If the bitwidth of the input signal to a HDL Counter exceeds the data type limit, the generated HDL code can produce incorrect simulation results. To accommodate the larger bit width, use a larger data type.

### **Extended Capabilities**

#### **HDL Code Generation**

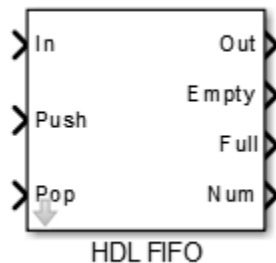
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



## HDL FIFO

Stores sequence of input samples in first in, first out (FIFO) register



## Library

HDL Coder / HDL Operations

## Description

The HDL FIFO block stores a sequence of input samples in a first in, first out (FIFO) register.

## Parameters

### Register size

Specify the number of entries that the FIFO register can hold. The minimum is 4. The default is 10.

### The ratio of output sample time to input sample time

Inputs (In, Push) and outputs (Out, Pop) can run at different sample times. Enter the ratio of output sample time to input sample time. Use a positive integer or  $1/N$ , where  $N$  is a positive integer. The default is 1.

For example:

- If you enter 2, the output sample time is twice the input sample time, meaning the outputs run slower.
- If you enter 1/2, the output sample time is half the input sample time, meaning the outputs run faster.

The Full, Empty, and Num signals run at the faster rate.

### **Push onto full register**

Response (Ignore, Error, or Warning) to a trigger received at the Push port when the register is full. The default is Warning.

### **Pop empty register**

Response (Ignore, Error, or Warning) to a trigger received at the Pop port when the register is empty. The default is Warning.

### **Show empty register indicator port (Empty)**

Enable the Empty output port, which is high (1) when the FIFO register is empty and low (0) otherwise.

### **Show full register indicator port (Full)**

Enable the Full output port, which is high (1) when the FIFO register is full and low (0) otherwise.

### **Show number of register entries port (Num)**

Enable the Num output port, which tracks the number of entries currently in the queue.

## **Ports**

The block has the following ports:

In

Data input signal.

Push

Control signal. When this port receives a value of 1, the block pushes the input at the In port onto the end of the FIFO register.

**Pop**

Control signal. When this port receives a value of 1, the block pops the first element off the FIFO register and holds the Out port at that value.

**Out**

Data output signal.

**Empty**

The block asserts this signal when the FIFO register is empty. This port is optional.

**Full**

The block asserts this signal when the FIFO register is full. This port is optional.

**Num**

Current number of data values in the FIFO register. This port is optional.

If two or more of the control input ports are triggered in the same time step, the operations execute in the following order:

- 1 Pop
- 2 Push

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Dual Rate Dual Port RAM

**Introduced in R2014a**

# HDL Minimum Resource FFT

FFT— optimized for HDL code generation using minimum hardware resources (HDL Coder)

## Description

The HDL Minimum Resource FFT block is available with DSP System Toolbox.

For information about the DSP System Toolbox simulation behavior and block parameters, see HDL Minimum Resource FFT.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

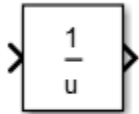
### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Introduced in R2014b

## HDL Reciprocal

Calculate reciprocal with Newton-Raphson approximation method



HDL Reciprocal

## Library

HDL Coder / HDL Operations

## Description

The HDL Reciprocal block uses the Newton-Raphson iterative method to compute the reciprocal of the block input. The Newton-Raphson method uses linear approximation to successively find better approximations to the roots of a real-valued function.

The reciprocal of a real number  $a$  is defined as a zero of the function:

$$f(x) = \frac{1}{x} - a$$

HDL Coder chooses an initial estimate in the range  $0 < x_0 < \frac{2}{a}$  as this is the domain of convergence for the function.

To successively compute the roots of the function, specify the **Number of iterations** parameter in the Block Parameters dialog box. The process is repeated as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + (x_i - ax_i^2) = x_i \cdot (2 - ax_i)$$

$f'(x)$  is the derivative of the function  $f(x)$ .

Following table shows comparison of simulation behavior of HDL Reciprocal with Math Reciprocal block:

<b>Math Reciprocal</b>	<b>HDL Reciprocal</b>
Computes the reciprocal as $1/N$ by using the HDL divide operator (/) to implement the division.	Uses the Newton-Raphson iterative method. The block computes an approximate value of reciprocal of the block input and can yield different simulation results compared to the Math Reciprocal block.  To match the simulation results with the Math Reciprocal block, increase the number of iterations for the HDL Reciprocal block.

## Parameters

### Number of iterations

Number of Newton-Raphson iterations. The default is 3.

## Ports

The block has the following ports:

### Input

- Supported data types: Fixed-point, integer (signed or unsigned), double, single
- Minimum bit width: 2
- Maximum bit width: 128

### Output

<b>Input data type</b>	<b>Output data type</b>
double	double
single	single
built-in integer	built-in integer
built-in fixed-point	built-in fixed-point

<b>Input data type</b>	<b>Output data type</b>
<i>fi (value, 0, word_length, fraction_length)</i>	<i>fi (value, 0, word_length, word_length-fraction_length-1)</i>
<i>fi (value, 1, word_length, fraction_length)</i>	<i>fi (value, 1, word_length, word_length-fraction_length-2)</i>

## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

<b>Architecture</b>	<b>Additional cycles of latency</b>	<b>Description</b>
ReciprocalNewton (default)	Iterations + 1	<p>Use the multirate implementation of the iterative Newton method. Select this option to optimize area.</p> <p>The default value for <b>Iterations</b> is 3.</p> <p>The recommended value for <b>Iterations</b> is from 2 through 10. If <b>Iterations</b> is outside the recommended range, HDL Coder displays a message.</p>



Architecture	Additional cycles of latency	Description
ReciprocalNewtonSingleRate	$(\text{Iterations} * 2) + 1$	<p>Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.</p> <p>The default value for <code>Iterations</code> is 3.</p> <p>The recommended value for <code>Iterations</code> is between 2 and 10. If <code>Iterations</code> is outside the recommended range, the coder displays a message.</p>

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Divide | Math Function

**Introduced in R2014b**

# HDL Streaming FFT

Radix-2 FFT with decimation-in-frequency (DIF) — optimized for HDL code generation (HDL Coder)

## Description

The HDL Streaming FFT block will be removed in a future release. Use the FFT HDL Optimized block instead.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Introduced in R2014b**

# Histogram

Frequency distribution (HDL Coder)

## Description

The Histogram block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see [Histogram](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Hit Crossing

Detect crossing point (HDL Coder)

## Description

The Hit Crossing block is available with Simulink.

For information about the simulation behavior and block parameters, see Hit Crossing.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restriction

The Hit crossing direction can only be *rising* or *falling*.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# HV Counter

Count active dimensions of a pixel stream (HDL Coder)

## Description

The HV Counter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see HV Counter.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Introduced in R2019a



# IFFT HDL Optimized

Inverse fast Fourier transform—optimized for HDL code generation (HDL Coder)

## Description

The IFFT HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see IFFT HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

- If you use the IFFT HDL Optimized block with the State Control block inside an Enabled Subsystem, the optional reset port is not supported. If you enable the reset port on the IFFT HDL Optimized block in such a subsystem, the model will error on Update Diagram.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Image Filter

2-D FIR filtering (HDL Coder)

## Description

The Image Filter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Image Filter.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Image Statistics

Mean, variance, and standard deviation (HDL Coder)

## Description

The Image Statistics block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Image Statistics.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Increment Real World

Increase real world value of signal by one (HDL Coder)

## Description

The Increment Real World block is available with Simulink.

For information about the simulation behavior and block parameters, see Increment Real World.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Increment Stored Integer

Increase stored integer value of signal by one (HDL Coder)

## Description

The Increment Stored Integer block is available with Simulink.

For information about the simulation behavior and block parameters, see Increment Stored Integer.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Index Vector

Switch output between different inputs based on value of first input (HDL Coder)

## Description

The Index Vector block is a Multiport Switch block with **Number of data ports** set to 1. For HDL code generation information, see Multiport Switch.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Inport

Create input port for subsystem or external input (HDL Coder)

### Description

The Inport block is available with Simulink.

For information about the simulation behavior and block parameters, see Inport.

### HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### BidirectionalPort

<b>BidirectionalPort Setting</b>	<b>Description</b>
on	Specify the port as bidirectional.  The following requirements apply: <ul style="list-style-type: none"><li>• The port must be in a Subsystem block with black box implementation.</li><li>• There must also be no logic between the bidirectional port and the corresponding top-level DUT subsystem port.</li></ul> For more information, see “Specify Bidirectional Ports”.

BidirectionalPort Setting	Description
off (default)	Do not specify the port as bidirectional.

## Target Specification

### IOInterface

Target platform interface type for DUT ports, specified as a character vector. The IOInterface block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterface settings, use the HDL Workflow Advisor:

- 1 In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
- 2 Save the model.

The IOInterface value is saved as an HDL block property of the port.

For example, to view the IOInterface value, if the full path to your DUT port is hdlcoder\_led\_blinking/led\_counter/LED, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
```

### IOInterfaceMapping

Target platform interface port mapping for DUT ports, specified as a character vector. The IOInterfaceMapping block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterfaceMapping settings, use the HDL Workflow Advisor:

- 1 In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
- 2 In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.
- 3 Save the model.

The IOInterfaceMapping value is saved as an HDL block property of the port.

For example, to view the `IOInterfaceMapping` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED',...  
            'IOInterfaceMapping')
```

## See Also

### Topics

""

**Introduced in R2014a**

# Integer-Input RS Encoder HDL Optimized

Encode data using a Reed-Solomon encoder (HDL Coder)

## Description

The Integer-Input RS Encoder HDL Optimized block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Integer-Input RS Encoder HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Integer-Output RS Decoder HDL Optimized

Decode data using a Reed-Solomon decoder (HDL Coder)

## Description

The Integer-Output RS Decoder HDL Optimized block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Integer-Output RS Decoder HDL Optimized.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Line Buffer

Store video lines and return neighborhood pixels (HDL Coder)

## Description

The Line Buffer block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see [Line Buffer](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# LMS Filter

Compute output, error, and weights using LMS adaptive algorithm (HDL Coder)

## Description

The LMS Filter block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see LMS Filter.

## HDL Architecture

By default, the LMS Filter implementation uses a linear sum for the FIR section of the filter.

The LMS Filter implements a tree summation (which has a shorter critical path) under the following conditions:

- The LMS Filter is used with real data.
- The word length of the Accumulator  $W'u$  data type is at least  $\text{ceil}(\log_2(\text{filter length}))$  bits wider than the word length of the Product  $W'u$  data type.
- The Accumulator  $W'u$  data type has the same fraction length as the Product  $W'u$  data type.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Restrictions**

- HDL Coder does not support the Normalized LMS algorithm of the LMS Filter.
- The Reset port supports only Boolean and unsigned inputs.
- The Adapt port supports only Boolean inputs.
- **Filter length** must be greater than or equal to 2.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Logical Operator

Perform specified logical operation on input (HDL Coder)

## Description

The Logical Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Logical Operator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Lookup Table

Map input pixel to output pixel using custom rule (HDL Coder)

## Description

The Lookup Table block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Lookup Table.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# LTE Convolutional Decoder

Decode convolutional-encoded samples using Viterbi algorithm (HDL Coder)

## Description

The LTE Convolutional Decoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Decoder .

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# LTE Convolutional Encoder

Encode binary samples using tailbiting convolutional algorithm (HDL Coder)

## Description

The LTE Convolutional Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Encoder.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# LTE CRC Decoder

Detect errors in input samples using checksum (HDL Coder)

## Description

The LTE CRC Decoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see CRC Decoder.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**



# LTE CRC Encoder

Generate checksum and append to input sample stream (HDL Coder)

## Description

The LTE CRC Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see CRC Encoder.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# LTE Turbo Decoder

Decode turbo-encoded samples (HDL Coder)

## Description

The LTE Turbo Decoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Turbo Decoder .

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem or an Enabled Synchronous Subsystem.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# LTE OFDM Demodulator

Demodulate samples using orthogonal frequency-division (HDL Coder)

## Description

The LTE OFDM Demodulator block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see OFDM Demodulator.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018a**

# LTE OFDM Modulator

Modulate samples using orthogonal frequency-division (HDL Coder)

## Description

The LTE OFDM Modulator block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see OFDM Modulator.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2019a**



# LTE Symbol Modulator

Modulate data bits according to LTE standard (HDL Coder)

## Description

The LTE Symbol Modulator block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see LTE Symbol Modulator.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Introduced in R2019a**

# NR Symbol Modulator

Modulate data bits according to 5G NR standard (HDL Coder)

## Description

The NR Symbol Modulator block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see NR Symbol Modulator.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Introduced in R2019a**

# LTE Turbo Encoder

Encode binary samples using turbo algorithm (HDL Coder)

## Description

The LTE Turbo Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Turbo Encoder.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017b**

# M-PSK Demodulator Baseband

Demodulate PSK-modulated data (HDL Coder)

## Description

The M-PSK Demodulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see M-PSK Demodulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# M-PSK Modulator Baseband

Modulate using M-ary phase shift keying method (HDL Coder)

## Description

The M-PSK Modulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see M-PSK Modulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



```
<softwaremeta type="block" version="5.0-variant tmwbook5.0" xml:base="../../
shreddoc/prod_softwaremeta/hdlcoder/
block_magnitudeangletocomplex_softwaremeta.xml">
<librarypath> HDL Coder / HDL Floating Point Operations </librarypath>
<extendedcapabilities codegen="yes" hdl="yes"></extendedcapabilities>
</softwaremeta>
```

## Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal (HDL Coder)

### Description

The Magnitude-Angle to Complex block is available with Simulink.

For information about the simulation behavior and block parameters, see Magnitude-Angle to Complex.

### HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Block configuration with additional latency	Number of additional cycles
Approximation method is CORDIC	Number of iterations + 1

### HDL Block Properties

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

The Magnitude-Angle to Complex block supports HDL code generation when you set **Approximation method** to CORDIC.

**Introduced in R2014a**

# Math Function

Perform mathematical function (HDL Coder)

## Description

The Math Function block is available with Simulink.

For information about the simulation behavior and block parameters, see Math Function.

## HDL Architecture

### conj

Architecture	Description
ComplexConjugate	Compute complex conjugate. See Math Function in the Simulink documentation.

### hermitian

Architecture	Description
Hermitian	Compute hermitian. See Math Function in the Simulink documentation.

### reciprocal

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Parameters	Additional cycles of latency	Description
Math (default) Reciprocal	None	0	Compute reciprocal as $1/N$ , using the HDL divide (/) operator to implement the division.
ReciprocalRsqrBasedNewton	Iterations	Signed input: Iterations + 5  Unsigned input: Iterations + 3	Use the iterative Newton method. Select this option to optimize area.  The default value for Iterations is 3.  The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, HDL Coder generates a message.
ReciprocalRsqrBasedNewtonSingleRate	Iterations	Signed input: (Iterations * 4) + 8  Unsigned input: (Iterations * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.  The default value for Iterations is 3.

Architecture	Parameters	Additional cycles of latency	Description
			The recommended value for <b>Iterations</b> is from 2 through 10. If <b>Iterations</b> is outside the recommended range, the coder generates a message.

The Newton-Raphson iterative method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i(1.5 - 0.5ax_i^2)$$

`ReciprocalRsqrtBasedNewton` and `ReciprocalRsqrtBasedNewtonSingleRate` implement the Newton-Raphson method with:

$$f(x) = \frac{1}{x^2} - 1$$

## transpose

Architecture	Description
Transpose	Compute array transpose. See Math Function in the Simulink documentation.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Native Floating Point

#### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

#### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

#### **NFPCustomLatency**

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

## Complex Data Support

The `conj`, `hermitian`, and `transpose` functions support complex data.

## Restrictions

When you use a reciprocal implementation:

- Input must be scalar and must have integer or fixed-point (signed or unsigned) data type.
- The output must be scalar and have integer or fixed-point (signed or unsigned) data type.
- Only the Zero rounding mode is supported.
- The **Saturate on integer overflow** option on the block must be selected.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# MATLAB Function

Include MATLAB code in models that generate embeddable C code (HDL Coder)

## Description

The MATLAB Function block is available with Simulink.

For information about the simulation behavior and block parameters, see MATLAB Function in Simulink documentation.

## Best Practices

- “Design Guidelines for the MATLAB Function Block”
- “Generate Instantiable Code for Functions”
- “Optimize MATLAB Loops”
- “Pipeline MATLAB Expressions”

## HDL Block Properties

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining”.



**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**InstantiateFunctions**

Generate a VHDL entity or Verilog module for each function. The default is `off`. See also “InstantiateFunctions”.

**LoopOptimization**

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

**MapPersistentVarsToRAM**

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**UseMatrixTypesInHDL**

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

**VariablesToPipeline**

---

**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

---

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

### Complex Data Support

This block supports code generation for complex signals.

See also “Complex Data Type Support”.

### Tunable Parameter Support

HDL Coder supports both tunable and non-tunable parameters with the following data types:

- Scalar
- Vector
- Complex
- Structure
- Enumeration

When using tunable parameters with the MATLAB Function block:

- The tunable parameter should be a Simulink.Parameter object with the StorageClass set to ExportedGlobal.

```
x = Simulink.Parameter
x.Value = 1
x.CoderInfo.StorageClass = 'ExportedGlobal'
```

- In the Ports and Data Manager dialog box, select the **tunable** check box.

For details, see “Generate DUT Ports for Tunable Parameters”.

### Restrictions

- If the block contains a System object, block inputs cannot have non-discrete (constant or Inf) sample time.
- HDL Coder does not support a MATLAB Function that contains the same variable as the input and output of the function. For example, this MATLAB code is not supported.

```
function y = myFun(y)
%#codegen
```

```
y = 3 * y;
```

For the MATLAB language subset supported for HDL code generation from a MATLAB Function block, see:

- “Supported MATLAB Data Types, Operators, and Control Flow Statements”
- “Persistent Variables and Persistent Array Variables”
- “HDL Code Generation for System Objects”
- “Complex Data Type Support”
- “Fixed-Point Bitwise Functions”
- “Fixed-Point Run-Time Library Functions”

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

### Topics

- “Code Generation from a MATLAB Function Block”
- “MATLAB Function Block Design Patterns for HDL”
- “Distributed Pipeline Insertion for MATLAB Function Blocks”
- “Generate DUT Ports for Tunable Parameters”
- “HDL Applications for the MATLAB Function Block”

**Introduced in R2014a**

# MATLAB System

Include System object in model (HDL Coder)

## Description

You can define a System object and use it in a MATLAB System block for HDL code generation.

The MATLAB System block is available with Simulink.

For information about the Simulink behavior and block parameters, see MATLAB System.

## Tunable Parameter Support

HDL Coder supports tunable parameters with the following data types:

- Numeric
- Fixed point
- Character
- Logical

When using tunable parameters with the MATLAB System block, the tunable parameter should be a Simulink.Parameter object with the StorageClass set to ExportedGlobal.

```
x = Simulink.Parameter
x.Value = 1
x.CoderInfo.StorageClass = 'ExportedGlobal'
```

For details, see “Generate DUT Ports for Tunable Parameters”.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

If you use a predefined System object, the HDL block properties available are the same as the properties available for the corresponding block.

By default, the following HDL block properties are available.

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **LoopOptimization**

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

### **MapPersistentVarsToRAM**

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### VariablesToPipeline

---

**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

---

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

## Restrictions

- The DUT subsystem must be single-rate.
- Inputs cannot have non-discrete (constant or `Inf`) sample time.
- The following predefined System objects are supported for code generation when you use them in the MATLAB System block:
  - `hdl.RAM`
  - `comm.HDLCRCDetector`
  - `comm.HDLCRCGenerator`
  - `comm.HDLRSDecoder`
  - `comm.HDLRSEncoder`
  - `dsp.DCBlocker`
  - `dsp.HDLComplexToMagnitudeAngle`
  - `dsp.HDLFFT`
  - `dsp.HDLIFFT`
  - `dsp.HDLNCO`
- If you use a user-defined System object, it must support HDL code generation. For information about user-defined System objects and requirements for HDL code generation, see “HDL Code Generation for System Objects”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“Generate Code for User-Defined System Objects”

“HDL Code Generation for System Objects”

**Introduced in R2014a**

# Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal (HDL Coder)

## Description

The Matrix Concatenate block is the Vector Concatenate block with **Mode** set to Multidimensional array. For HDL code generation information, see Vector Concatenate.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Matrix Multiply

Concatenate input signals of same data type to create contiguous output signal (HDL Coder)

## Description

The Matrix Multiply block is the Product block with **Multiplication** parameter set to `Matrix(*)`.

To learn about the block parameters and simulation behavior, see Product.

## HDL Architecture

This block has a single, default `Matrix Multiply` as the HDL Architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### DotProductStrategy

Implement the matrix multiplication by using a tree of adders and multipliers, or use the Multiply-Accumulate block implementation. The default is `Fully Parallel`. For more information, see “DotProductStrategy”.

### DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

The block supports code generation for complex signals.

## **Example**

For an example of how to use the Matrix Multiply block, see “Design Considerations for Matrices and Vectors”.

## **Restrictions**

HDL code generation does not support more than two inputs at the ports of the Matrix Multiply block.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018a**

# Matrix Viewer

Display matrices as color images (HDL Coder)

## Description

The Matrix Viewer block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Matrix Viewer.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## Maximum

Find maximum values in input or sequence of inputs (HDL Coder)

## Description

The Maximum block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Maximum.

## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
default Tree	0	Generates a tree structure of comparators.
Cascade	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**InstantiateStages**

Generate a VHDL entity or Verilog module for each cascade stage. The default is off. See also “InstantiateStages”.

**SerialPartition**

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Measure Timing

Measure timing of pixel control bus input (HDL Coder)

## Description

The Measure Timing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Measure Timing.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016b**

# Median Filter

2-D median filtering (HDL Coder)

## Description

The Median Filter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Median Filter.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot use the Median Filter block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**



# Memory

Output input from previous time step (HDL Coder)

## Description

The Memory block is available with Simulink.

For information about the simulation behavior and block parameters, see Memory.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Sequence Viewer

Display message or events between blocks during simulation (HDL Coder)

## Description

The Sequence Viewer block is available with Stateflow.

For information about the simulation behavior and block parameters, see [Sequence Viewer](#).

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## See Also

[Chart](#) | [State Transition Table](#) | [Truth Table](#)

**Introduced in R2015b**

## Minimum

Find minimum values in input or sequence of inputs (HDL Coder)

## Description

The Minimum block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Minimum.

## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
default Tree	0	Generates a tree structure of comparators.
Cascade	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**InstantiateStages**

Generate a VHDL entity or Verilog module for each cascade stage. The default is off. See also “InstantiateStages”.

**SerialPartition**

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## MinMax

Output minimum or maximum input value (HDL Coder)

### Description

The MinMax block is available with Simulink.

For information about the simulation behavior and block parameters, see MinMax.

### HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
default Tree	0	Generates a tree structure of comparators.
Cascade	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**InstantiateStages**

Generate a VHDL entity or Verilog module for each cascade stage. The default is off. See also “InstantiateStages”.

**SerialPartition**

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

**Native Floating Point**

---

**Note** To enable the **LatencyStrategy** setting for the MinMax block, you must specify Tree as the **HDL Architecture**.

---

**LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



## Model

Include model as block in another model (HDL Coder)

## Description

The Model block is available with Simulink. For information about the simulation behavior and block parameters, see Model.

## Generate Comments

If you enter text in the Model Block Properties dialog box **Description** field, HDL Coder generates a comment in the HDL code.

## Generate Code For Model Arguments

To generate a single Verilog module or VHDL entity for instances of a referenced model with different model argument values, see “Generate Parameterized Code for Referenced Models”.

## HDL Architecture

Architecture	Description
ModelReference (default)	When you want to generate code from a referenced model and any nested models, use the ModelReference implementation. For more information, see “How To Generate Code for a Referenced Model”.

Architecture	Description
BlackBox	<p>Use the BlackBox implementation to instantiate an HDL wrapper, or black box interface, for legacy or external HDL code. If you specify a black box interface, HDL Coder does not attempt to generate HDL code for the referenced model.</p> <p>For more information, see “Generate Black Box Interface for Referenced Model”.</p>

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**ReferenceModelPrefix**

Prefix of the referenced model to insert in the generated code. The code generator applies this prefix to submodel file names and HDL identifiers. The default prefix is `modelName_` where `modelName` is the name of the referenced model.

---

**Note**

- If you specify an empty prefix, the code generator does not add a prefix to submodel file names. This can cause HDL compilation errors due to naming collisions between the models.
  - If you use the referenced model as the DUT, the code generator ignores the prefix that you specify.
- 

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Restrictions**

- Model block must have default values for the Block parameters.
- Model block cannot be a masked subsystem.
- Multiple model references that refer to the same model must have the same HDL block properties.
- Referenced models cannot be protected models.
- Hierarchical distributed pipelining must be disabled.

HDL Coder cannot move registers across a model reference. Therefore, referenced models can inhibit these optimizations:

- Distributed pipelining
- Constrained output pipelining
- Streaming

When you have model references and generate HDL code, the generated model, validation model, and cosimulation model can fail to compile or simulate. To fix compilation or simulation errors, make sure that the referenced models are loaded or are on the search path.

The coder can apply the resource sharing optimization to share referenced model instances. However, you can apply this optimization only when all model references that point to the same referenced model have the same rate after optimizations and rate propagation. The model reference final rate may differ from the original rate, but all model references that point to the same referenced model must have the same final rate.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

### Topics

“Model Referencing for HDL Code Generation”

“Generate Black Box Interface for Referenced Model”

“Generate Parameterized Code for Referenced Models”

**Introduced in R2014a**

# Model Info

Display model properties and text in model (HDL Coder)

## Description

The Model Info block is available with Simulink.

For information about the simulation behavior and block parameters, see Model Info.

## Best Practices

When using Model Info blocks in models targeted for HDL code generation, consider using only ASCII characters in the text that you enter to display on the Model Info block. If you have non-ASCII characters in the generated HDL code, RTL simulation and synthesis tools can fail to compile the code.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

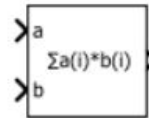
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Multiply-Accumulate

Perform a multiply-accumulate operation on the inputs

**Library:** HDL Coder / HDL Operations



Multiply-Accumulate

## Description

The Multiply-Accumulate block performs this operation on inputs *a* and *b*, and bias *c*, to compute result *dataOut*.

$$\text{dataOut} = \text{sum}(\mathbf{a}.* \mathbf{b}) + c$$

By default, the block operates in the vector mode. The inputs *a* and *b* can be scalars or vectors. The default bias value, *c*, is equal to zero, and the block computes the dot product of inputs *a* and *b*. You can specify a nonzero value for *c* using **Dialog** or **Input port** as the **Source**. The block adds this bias to the dot product of *a* and *b*. The multiplication operation is full precision irrespective of the **Output data type** setting. The **Output data type** and **Integer rounding mode** settings apply to the addition operation.

By using the **Operation Mode** setting, you can specify streaming modes of operation for the Multiply-Accumulate block. For HDL code generation, when you use the streaming operation mode, input scalar values to the block. The block has two streaming modes: **Streaming - using Start and End ports** and **Streaming - using Number of Samples**. When you select these streaming modes, you can specify the control signals to use with the mode. The control signals specify when to start and end accumulation, and when the output is valid.

## Limitations for HDL Code Generation

- Scalar inputs are not supported for HDL code generation. To generate code for the block, use vector inputs. With scalar inputs, use the Multiply-Add block.



- Matrix data types are not supported at the block port interfaces. If you have matrix type signals, use the Matrix Multiply block.
- Streaming modes of operation for the block are not supported inside a Resettable Subsystem block for HDL code generation.

## Ports

### Input

#### **a — Input signal**

vector | matrix | array | bus

Port to provide input to the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

#### **b — Input signal**

scalar | vector | matrix | array | bus

Port to provide input to the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

#### **c — Bias signal**

scalar | vector | matrix | array | bus

Port to provide the bias signal to the block. The block adds this bias to the inputs. Make sure that the bias signal data type matches that of the dot product of the inputs.

### Dependencies

To enable this port, set **Source** to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

#### **startIn — Start of accumulation control signal**

scalar | vector | matrix | array | bus

Port to provide the control signal to start accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the `dataOut` signal, both `startIn` and `validIn` signals must be high. The `dataOut` signal produces the accumulated result from the next clock cycle.

### Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### **validIn** — Valid input control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate that the input signal is valid for accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the `dataOut` signal, both `validIn` and `startIn` signals must be high. The `dataOut` signal produces the accumulated result from the next clock cycle. The `validIn` signal has higher priority than `startIn` and `endIn` signals.

### Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports or Streaming - using Number of Samples.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### **endIn** — End of accumulation control signal

`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate end of accumulation. You can use the `startIn` and `endIn` signals with the `validIn` signal to indicate a frame that contains the accumulated output.

### Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports and then select **End input and output ports**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

## Output

### **dataOut** — Output signal

scalar | vector | matrix | array | bus

Port that generates the output data from the multiply-accumulate operation. By default, the block uses the **Vector** mode of operation and computes the dot product of the input signals, and adds the bias to produce the result. If you specify a streaming mode of operation as **Operation Mode**, the value of the **dataOut** signal depends on the control signals that you provide. The data type of the output signal is same as that of the accumulator.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### **startOut** — Start of accumulation output control signal

scalar | vector | matrix | array | bus

Port that generates output control signal to indicate the start of accumulation. When both **validIn** and **startIn** are high, the **startOut** signal becomes high in the next clock cycle. The clock cycle at which **startOut** becomes high indicates the start of a frame and that the **dataOut** signal has started producing valid accumulated output.

#### **Dependencies**

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports and then select **Start output port**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### **validOut** — Valid output control signal

scalar | vector | matrix | array | bus

Port that generates the output control signal to indicate that the **dataOut** signal is valid. When the **validIn** signal becomes high, the **validOut** signal becomes high in the next clock cycle and indicates that the **dataOut** is valid.

#### **Dependencies**

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports and then select **Valid output port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### **endOut — End of accumulation output control signal**

`scalar` | `vector` | `matrix` | `array` | `bus`

Port that generates the output control signal to indicate the end of accumulation. You can use the clock cycles between when the **startOut** signal becomes high and when the **endOut** signal becomes high to indicate a valid frame that contains the accumulated output.

#### **Dependencies**

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **End input and output ports**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

### **countOut — Count output control signal**

`scalar` | `vector` | `matrix` | `array` | `bus`

Port that generates the output control signal to indicate number of samples to accumulate. The value of this signal increases from 1 to the value that you specify for **Number of Samples**. As long as the **validIn** signal is high, the **countOut** increments by 1 every clock cycle.

#### **Dependencies**

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples` and then select **Count output port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

## **Parameters**

### **Operation Mode — Mode of accumulation of inputs**

`'Vector'` (default) | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`

You can specify the **Operation Mode** as:

- **Vector:** You can use scalars or vectors as inputs. The block performs the dot product of the inputs `u1` and `u2` and adds bias `k` to produce the result.
- **Streaming - using Start and End Ports:** Use scalar inputs for HDL code generation. In this mode, you can use the **startIn** and **endIn** control signals to determine when to start and stop accumulation. The output data is valid when **validIn** is high.
- **Streaming - using Number of Samples:** Use scalar inputs for HDL code generation. In this mode, you can specify the **Number of Samples** and use the **countIn** control signal to determine when to start and stop accumulation. The output data is valid when **validIn** is high.

#### Programmatic Use

**Block parameter:** `opMode`

**Type:** character vector

**Value:** `'Vector'` | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`

**Default:** `'Vector'`

#### Bias — Offset to add to the input dot product

`{'0.0'}` (default)

You can specify the bias with:

- **Source** as `Dialog`. Then, specify the **Value**.
- **Source** as `Input port`. This setting creates an external input port `c` to input the bias signal to the block.

#### Programmatic Use

**Block parameter:** `initValueSetting`

**Type:** character vector

**Value:** `'Dialog'` | `'Input port'`

**Default:** `'Dialog'`

If you set **Source** as `Dialog`, you can specify the initial value by using the `initValue2` setting.

**Block parameter:** `initValue2`

**Type:** character vector

**Value:** An integer greater than or equal to zero

**Default:** `'0.0'`

#### **Number of Samples — Number of samples of valid accumulated output signal** { '2' } (default)

You can specify the **Number of Samples** to specify a frame containing the number of samples of valid accumulated output **dataOut**.

#### **Dependencies**

To enable this port, set **Operation Mode** to Streaming - using Number of Samples.

#### **Programmatic Use**

**Block parameter:** num\_samples

**Type:** character vector

**Value:** An integer greater than or equal to zero

**Default:** '2'

#### **Output data type — Data type of the block output**

Inherit: Inherit via back propagation (default)

Set the output data type to:

- A rule that inherits a data type, such as Inherit: Same as first input.
- A built-in data type, such as single or int16.
- The name of a data type object. for instance, a Simulink.NumericType object.
- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

The streaming modes do not support Inherit: Inherit via internal rule. When you set the **Output data type**, you can use the **Data Type Assistant**. To display the

assistant, click the **Show data type assistant** .

#### **Programmatic Use**

**Block parameter:** OutDataTypeStr

**Type:** character vector

**Default:** {'Inherit: Inherit via internal rule'}

To see possible values that you can specify for this parameter, see “Block-Specific Parameters” (Simulink).

#### **Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding action as:

#### Ceiling

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

#### Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer™ `convergent` function.

#### Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

#### Nearest

Rounds the number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

#### Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

#### Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

#### Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

#### **Programmatic Use**

**Block parameter:** `RndMeth`

**Type:** character vector

**Default:** `{'Floor'}`

To see possible values that you can specify for this parameter, see “Block-Specific Parameters” (Simulink).

#### **Valid output port — Control generation of validOut output port**

`off` (default) | `on`

Control generation of the **validOut** output port. This port indicates whether **dataOut** is valid.

off

Does not display the **validOut** output port.

on

Display the **validOut** output port.

#### Dependencies

To enable this port, set **Operation Mode** to Streaming - using Number of Samples or Streaming - using Start and End Ports.

#### Programmatic Use

**Block parameter:** validOut

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

#### End input and output ports — Control generation of endIn input port and endOut output port

off (default) | on

Control generation of the **endIn** input port and the **endOut** output port. The ports indicate the end of a frame containing valid accumulation output.

off

Does not display the **endIn** input port and the **endOut** output port.

on

Display the **endIn** input port and the **endOut** output port.

#### Dependencies

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports.

#### Programmatic Use

**Block parameter:** endInandOut

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'



**Start output port — Control generation of startOut output port**

off (default) | on

Control generation of the **startOut** output port. This port generates the **startOut** signal that indicates the start of a frame containing valid accumulated output.

 off

Does not display the **startOut** output port.

 on

Display the **startOut** output port.

**Dependencies**

To enable this port, set **Operation Mode** to Streaming - using Start and End Ports.

**Programmatic Use****Block parameter:** startOut**Type:** character vector**Values:** 'off' | 'on'**Default:** 'off'**Count output port — Control generation of countOut output port**

off (default) | on

Control generation of the **countOut** output port. This port generates the counter that indicates a frame containing valid samples.

 off

Does not display the **countOut** output port.

 on

Display the **countOut** output port.

**Dependencies**

To enable this port, set **Operation Mode** to Streaming - using Number of Samples.

**Programmatic Use**

**Block parameter:** countOut

**Type:** character vector

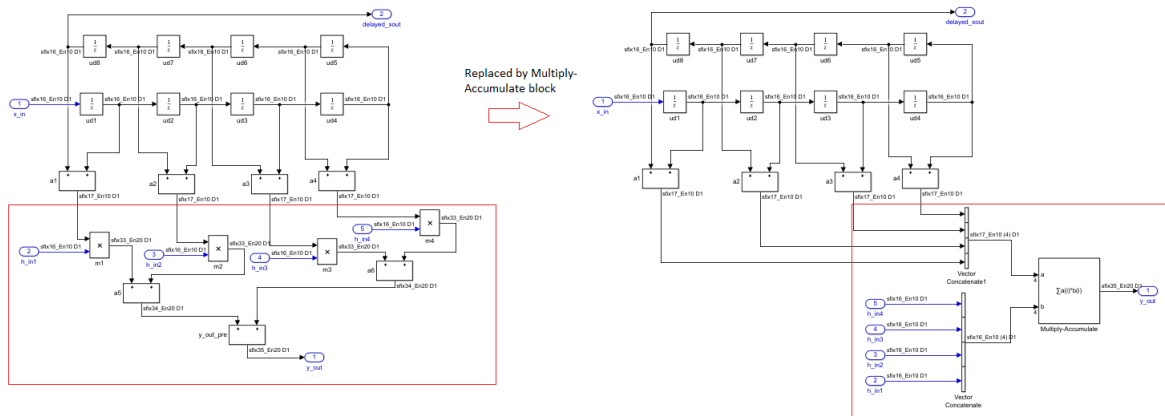
**Values:** 'off' | 'on'

**Default:** 'off'

**Benefits**

With the Multiply-Accumulate block, you can:

- Perform matrix multiplication operations. For example, if you have two matrix inputs with dimensions N-by-M and M-by-P, you can compute the result by using N-by-P multiply-accumulate operations in parallel.
- Replace a sequence of multiplication and addition operations, such as in filter blocks, and improve the performance on hardware by mapping to DSP slices on the FPGA. This figure shows how you can use the Multiply-Accumulate block with the `sfir_fixed` model.



# Algorithms

## Streaming Mode Using Start and End Ports

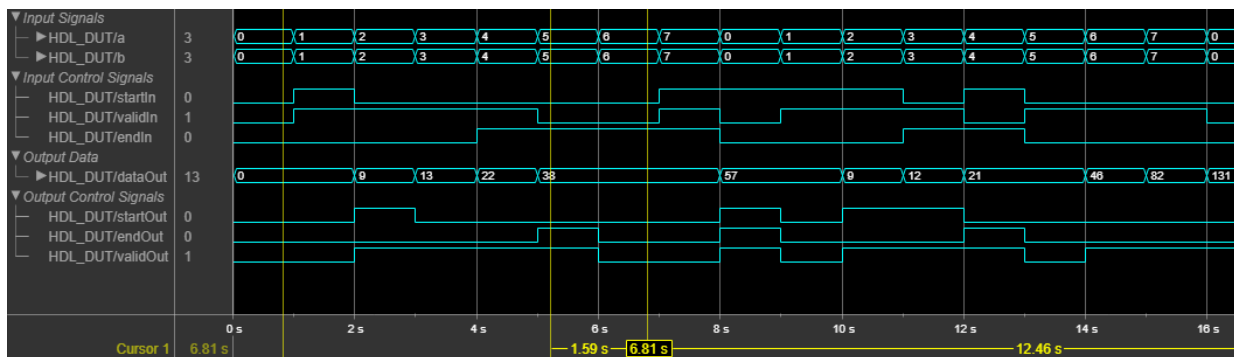
You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select Streaming - using Start and End Ports, you see three additional settings enabled by default. The settings include:

- **Valid output port**
- **End input and output ports**
- **Start output port**

It is recommended that you leave these settings enabled. When you apply the settings, three additional input ports and three additional output ports appear:

Input Ports	Output Ports
<b>startIn</b>	<b>startOut</b>
<b>validIn</b>	<b>validOut</b>
<b>endIn</b>	<b>endOut</b>

This figure illustrates the streaming mode of operation using the start and end ports. In this example, the bias value is 8.



Initially, when **validIn** is low, **dataOut** is zero. At time 1s, both **startIn** and **validIn** become high. Therefore, **validOut** becomes high in the next clock cycle and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the

values of **a** and **b** from the previous clock cycle. For example, at time  $t = 2s$ , **dataOut** =  $1*1 + 8 = 9$ .

To continue accumulation, make **startIn** low at the next clock cycle and keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. At each time step, **dataOut** computes the product of the inputs from the previous clock cycle and sums the result with the **dataOut** value from the previous clock cycle. For example, at time  $t = 3s$ , **dataOut** =  $2*2 + 9 = 13$ .

When **validIn** becomes low, **dataOut** holds the output value as seen at time  $t = 5s$ . At  $t = 5s$ , **endIn** and **validIn** are high. Therefore, **endOut** becomes high in the next clock cycle, which indicates end of frame. Therefore the frame between  $t = 2s$  (when **startOut** is high) and  $t = 6s$  (when **endOut** is high) indicates a frame containing valid output.

If **startIn**, **validIn**, and **endIn** are both high at the same time, only the **dataOut** corresponding to those inputs are accumulated as seen at  $t = 8s$ . If **startIn** is high for multiple clock cycles, and if **validIn** is high, the accumulator is reset at each clock cycle as seen at  $t = 10s$  and  $t = 11s$ . The accumulation continues at  $t = 12s$ .

## Streaming Mode Using Number of Samples

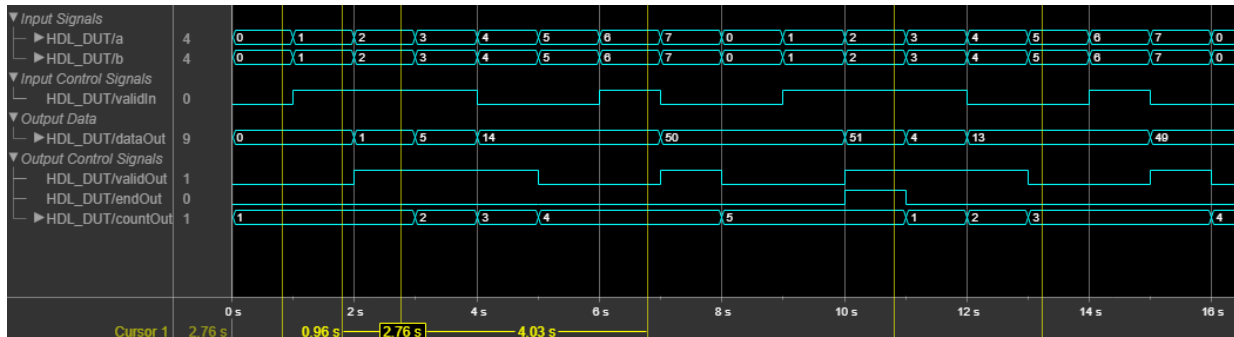
You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select Streaming - using Number of Samples, you see two additional settings enabled by default. The settings include:

- **Valid output port**
- **Count output port**

It is recommended that you leave these settings enabled. When you apply the settings, you have an additional input port **validIn** and three additional output ports appear:

- **endOut**
- **validOut**
- **countOut**

This figure illustrates the streaming mode of operation using the number of samples. In this example, the bias value is 8 and the **Number of Samples** is 5.



Initially, when **validIn** is low, **dataOut** is 0 and **countOut** is 1. At time 1s, **validIn** becomes high. Therefore, **validOut** becomes high in the next clock cycle, and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the values of **a** and **b** from the previous clock cycle. For example, at time  $t = 2s$ , **dataOut** =  $1 * 1 = 1$ . **countOut** increments by 1 at the next clock cycle, that is, at  $t = 3s$ , **countOut** becomes 2.

To continue accumulation, keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. When five valid outputs are obtained from **dataOut**, **countOut** becomes 5 and **endOut** becomes high, which indicates the end of the frame. Therefore, the time between when **countOut** is 1 and when **countOut** is five indicates a frame containing valid output.

The accumulator counter is now reset and **countOut** starts from 1. When **validIn** becomes high again, **dataOut** starts accumulating a new set of values and **countOut** starts incrementing for each valid **dataOut**.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### HDL Architecture

HDL Architecture Setting	Description
Auto (Default)	<p>This mode selects the <code>Serial</code> architecture by default. When the block is inside a feedback loop, the code generator cannot use the <code>Serial</code> architecture if the block is not part of a clock-rate pipelining region and does not have a <code>Delay</code> at the block output. This error occurs because the <code>Serial</code> architecture introduces additional latency which cannot be delay balanced inside the feedback loop. When you use the <code>Auto</code> mode, the code generator switches to the <code>Parallel</code> architecture automatically.</p>
Parallel	<p>For input vectors of size <code>N</code>, this mode uses <code>N</code> <code>Multiply-Add</code> blocks in series to compute the result. This mode uses a combinatorial implementation and does not introduce any latency. If you specify the <b>Synthesis tool</b> and <b>Target frequency</b>, since the adaptive pipelining optimization is enabled, the code generator inserts pipeline registers for the <code>Multiply-Add</code> blocks. When you synthesize your design, depending on the input bit widths, this architecture maps up to <code>N</code> DSP slices on the FPGA.</p>

HDL Architecture Setting	Description
Serial	<p>For input vectors of size N, this mode uses a streaming algorithm to implement the multiply-accumulate operation. This architecture has two implementation modes:</p> <ul style="list-style-type: none"> <li>• The default mode uses a local multirate implementation. This implementation overclocks the shared resources by N and multiplexes the input vectors with a Multiply-Add block. This implementation introduces an additional latency of one at the data rate.</li> <li>• If you have clock-rate pipelining enabled on the model or subsystem that contains the Multiply-Accumulate block, this architecture uses a single-rate implementation. This implementation runs the shared resources at the clock-rate and multiplexes the input vectors with a Multiply-Add block. This implementation introduces an additional latency of N at the clock rate.</li> </ul> <p>When you synthesize your design, depending on the input bit widths, this architecture maps to one DSP slice on the FPGA.</p>

### HDL Block Properties

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Complex Data Support**

When you use complex signals, this block can generate HDL code, but does not map to DSP slices.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

### **See Also**

Dot Product | Multiply-Add

### **Topics**

“Adaptive Pipelining”

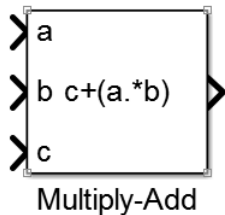
“Clock-Rate Pipelining”

**Introduced in R2017b**



## Multiply-Add

Multiply-add combined operation



## Library

HDL Coder / HDL Operations

## Description

The Multiply-Add block computes the product of the first two inputs, a and b, and adds the result to the third input, c. The inputs can be vectors or scalars.

The multiplication operation is full precision, regardless of the output type. The **Integer rounding mode**, **Output data type**, and **Saturate on integer overflow** settings apply only to the addition operation.

Use the Multiply-Add block to map a combined multiply-add or a multiply-subtract operation to a DSP unit in your target hardware. You can select the **Function** setting in the Block Parameters dialog box for the Multiply-Add block.

To map to a DSP unit, specify the `SynthesisTool` property for your model. When you generate HDL code for your model, HDL Coder configures the multiply-add operation so that your synthesis tool can map to a DSP unit.

---

**Note** Some DSP units do not have the multiply-add capability. To see if your hardware has the multiply-add capability, refer to the documentation for the hardware.

---

# Data Type Support

The Multiply-Add block accepts and outputs signals of any numeric data type that Simulink supports, including fixed-point data types.

For more information, see “Data Types Supported by Simulink” (Simulink).

## Parameters

### Function

Specify the function to perform a combined multiply and add or a multiply and subtract operation.

#### Settings

**Default:** `c+(a.*b)`

You can set the function to:

- `c+(a.*b)`
- `c-(a.*b)`
- `(a.*b)-c`

### Output data type

Specify the output data type.

#### Settings

**Default:** `Inherit: Inherit via internal rule`

Set the output data type to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button  to display the Data Type Assistant dialog box, which helps you to set the **Output data type** parameter.

For more information, see “Control Signal Data Types” (Simulink) in *Simulink User's Guide* (Simulink) .

## Integer rounding mode

Specify the rounding mode for fixed-point operations.

### Settings

#### Default: Floor

##### Ceiling

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

##### Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

##### Floor

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

##### Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

##### Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

##### Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

##### Zero

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### See Also

For more information, see “Rounding” (Fixed-Point Designer).

## Saturate on integer overflow

Specify whether overflows saturate.

### Settings

**Default:** Off

On

Overflows saturate to either the minimum or maximum value that the data type can represent.

For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent.

For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

### Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors” (Simulink).

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

### Command-Line Information

**Parameter:** SaturateOnIntegerOverflow

**Type:** character vector

**Value:** 'off' | 'on'

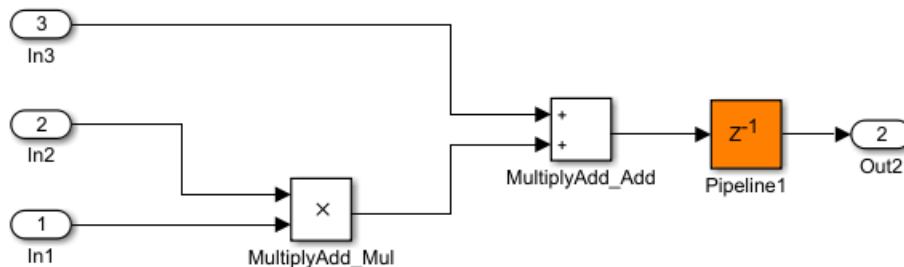
**Default:** 'off'

## Pipeline Depth

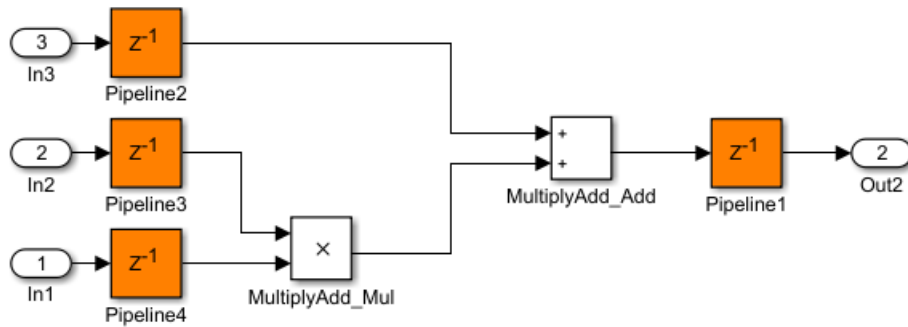
If you have fixed-point inputs to a Multiply-Add block, you can set the **PipelineDepth** for the block. For floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert pipeline registers.

The following diagrams show different configurations of pipeline registers for different synthesis tools and **PipelineDepth** settings. When you specify the **PipelineDepth** setting, HDL Coder inserts pipeline registers so that the configuration maps efficiently to DSP units.

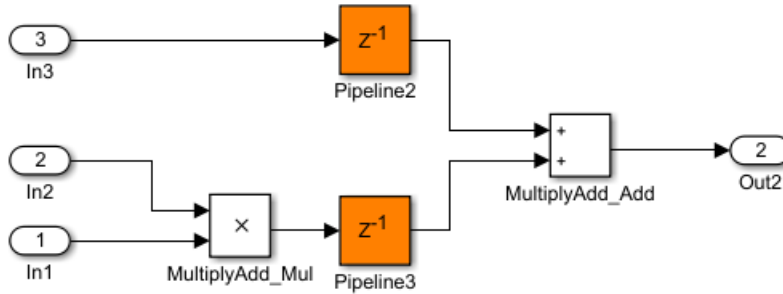
### Altera Hardware with PipelineDepth = 1



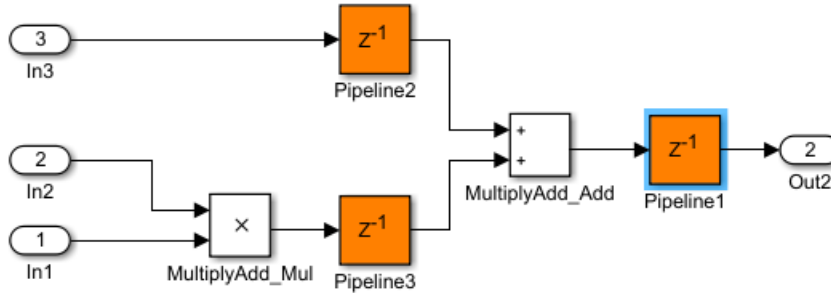
### Altera Hardware with PipelineDepth = 2



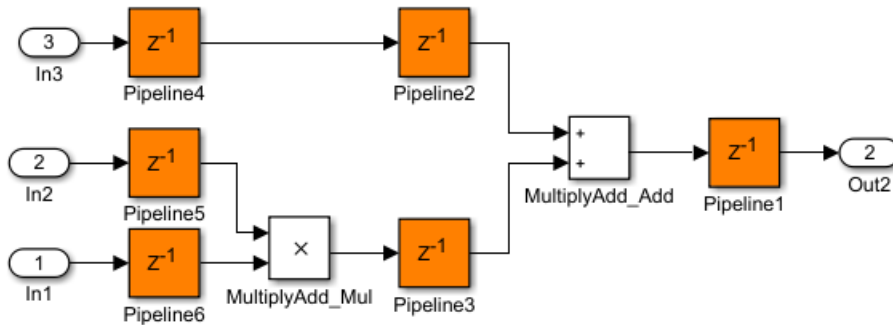
### Xilinx Hardware with PipelineDepth = 1



## Xilinx Hardware with PipelineDepth = 2



## Xilinx Hardware with PipelineDepth = 3



## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **PipelineDepth**

Number of pipeline stages. The default is `auto` which means that the coder determines the number of pipeline stages based on your synthesis tool.

You can enter an integer between 0 and 3. For Altera hardware targets, the maximum pipeline depth is 2.

### Native Floating Point

#### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

#### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

#### **MantissaMultiplyStrategy**

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

- When the block has floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert pipeline registers.



- If the block is in a feedback loop and you do not have sufficient delays at the block output, the coder reduces the **PipelineDepth** to prevent delay balancing failure. For sufficient delays, add Delay blocks at the output of the Multiply-Add block.
- To map the combined multiply-add operation to a DSP unit, the width of the third input *c* has to be less than 64 bits for Altera and 48 bits for Xilinx respectively.
- The subtraction operation in the **Function** setting ( $a \cdot b - c$ ) does not map to a DSP unit in Altera FPGA libraries.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

Dot Product | Multiply-Accumulate

### Topics

“Adaptive Pipelining”

“Clock-Rate Pipelining”

**Introduced in R2015b**

# Multiport Selector

Distribute arbitrary subsets of input rows or columns to multiple output ports (HDL Coder)

## Description

The Multiport Selector block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Multiport Selector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Multiport Switch

Choose between multiple block inputs (HDL Coder)

## Description

The Multiport Switch block is available with Simulink.

For information about the simulation behavior and block parameters, see Multiport Switch.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

`<varlistentry version="5.0-variant tmwbook5.0" xml:base="../../shreddoc/hdl/ref/hdlblkprops/block_property_latencystrategy.xml">LatencyStrategy`

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also "LatencyStrategy".

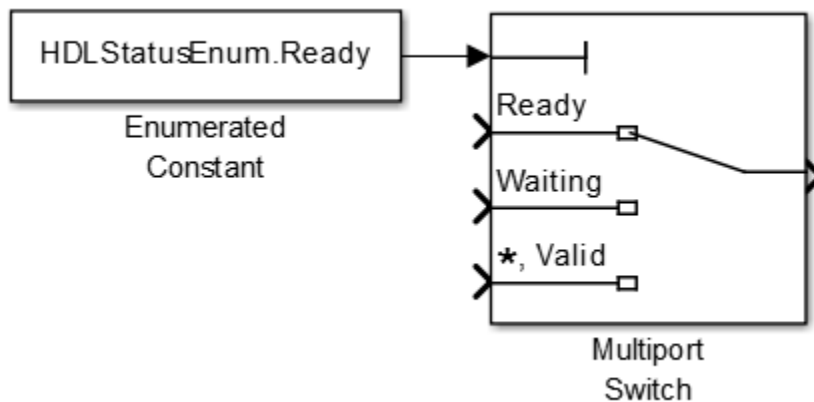
`</varlistentry>`

## Complex Data Support

This block supports code generation for complex signals.

## Example

You can set **Data port order** to **Specify indices**, and enter enumeration values for the **Data port indices**. For example, you can connect the Enumerated Constant block to the Multiport Switch control port and use the enumerated types as data port indices.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Mux

Combine several input signals into vector (HDL Coder)

## Description

The Mux block is available with Simulink.

For information about the simulation behavior and block parameters, see Mux.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Restrictions**

Buses are not supported for HDL code generation.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# n-D Lookup Table

Approximate N-dimensional function (HDL Coder)

## Description

The n-D Lookup Table block is available with Simulink.

For information about the simulation behavior and block parameters, see n-D Lookup Table.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## Restrictions

### MAX 10 Device Settings

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

### Required Block Settings

Block Setting	HDL Coder support
<b>Number of table dimensions</b>	Specify upto a maximum dimension of 2.
<b>Breakpoints specification</b>	Select either Explicit values or Even spacing.
<b>Index search method</b>	Select Evenly spaced points.
<b>Extrapolation method</b>	Select Clip. The code generator does not support extrapolation beyond even bounds.
<b>Interpolation method</b>	Select Flat or Linear point-slope.
<b>Diagnostic for out-of-range input</b>	Select Error. If you specify other options, HDL Coder generates a warning.
<b>Use last table value for inputs at or above last breakpoint</b>	Select this check box.
<b>Require all inputs to have the same data type</b>	Select this check box.
<b>Fraction</b>	Select Inherit: Inherit via internal rule.
<b>Integer rounding mode</b>	Select Zero, Floor, or Simplest.

## Avoid Generation of Divide Operator

If HDL Coder encounters conditions under which a division operation is required to match the model simulation behavior, a warning is displayed. The conditions described cause this block to emit a divide operator. When you use this block for HDL code generation, avoid the following conditions:

- If the block is configured to use interpolation, a division operator is required. To avoid this requirement, set **Interpolation method** : to `Flat`.
- Uneven table spacing. HDL code generation requires the block to use the "Evenly Spaced Points" algorithm. The block mapping from the input data type to the zero-based table index in general requires a division. When the breakpoint spacing is an exact power of 2, this divide is implemented as a shift instead of as a divide. To adjust the breakpoint spacing, adjust the number of breakpoints in the table, or the difference between the left and right bounds of the breakpoint range.

## Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. If the breakpoint spacing does not meet this condition, HDL Coder issues a warning. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- Table data must resolve to a nonfloating-point data type.
- All ports on the block require scalar values.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# NCO

Generate real or complex sinusoidal signals (HDL Coder)

## Description

HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## NCO HDL Optimized

Generate real or complex sinusoidal signals—optimized for HDL code generation (HDL Coder)

### Description

The NCO HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see NCO HDL Optimized.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **LUTRegisterResetType**

The reset type of the lookup table output register. Select `none` to synthesize the lookup table to a ROM when your target is an FPGA. See also “LUTRegisterResetType”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

- When you set **Dither source** to `Property`, the block adds random dither every cycle. If you generate a validation model with these settings, a warning is displayed. Random generation of the internal dither can cause mismatches between the models. You can increase the error margin for the validation comparison to account for the difference. You can also disable dither or set **Dither source** to `Input port` to avoid this issue.
- You cannot use the NCO HDL Optimized block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Opening

Morphological open of binary pixel data (HDL Coder)

## Description

The Opening block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see [Opening](#).

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

## Output

Create output port for subsystem or external output (HDL Coder)

## Description

The Output block is available with Simulink.

For information about the simulation behavior and block parameters, see Output.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### BidirectionalPort

<b>BidirectionalPort Setting</b>	<b>Description</b>
on	<p>Specify the port as bidirectional.</p> <p>The following requirements apply:</p> <ul style="list-style-type: none"><li>• The port must be in a Subsystem block with black box implementation.</li><li>• There must also be no logic between the bidirectional port and the corresponding top-level DUT subsystem port.</li></ul> <p>For more information, see “Specify Bidirectional Ports”.</p>

BidirectionalPort Setting	Description
off (default)	Do not specify the port as bidirectional.

## Target Specification

### IOInterface

Target platform interface type for DUT ports, specified as a character vector. The IOInterface block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterface settings, use the HDL Workflow Advisor:

- 1 In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
- 2 Save the model.

The IOInterface value is saved as an HDL block property of the port.

For example, to view the IOInterface value, if the full path to your DUT port is hdlcoder\_led\_blinking/led\_counter/LED, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
```

### IOInterfaceMapping

Target platform interface port mapping for DUT ports, specified as a character vector. The IOInterfaceMapping block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterfaceMapping settings, use the HDL Workflow Advisor:

- 1 In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
- 2 In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.
- 3 Save the model.

The IOInterfaceMapping value is saved as an HDL block property of the port.

For example, to view the `IOInterfaceMapping` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED',...  
            'IOInterfaceMapping')
```

## See Also

### Topics

""

**Introduced in R2014a**

# Pixel Stream Aligner

Align two streams of pixel data (HDL Coder)

## Description

The Pixel Stream Aligner block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Pixel Stream Aligner.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2017a**

# Pixel Stream FIFO

Rebuffer input stream to make each image line contiguous valid pixels (HDL Coder)

## Description

The Pixel Stream FIFO block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Pixel Stream FIFO.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem, or a Triggered Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018a**



# PN Sequence Generator

Generate pseudonoise sequence (HDL Coder)

## Description

The PN Sequence Generator block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see PN Sequence Generator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

- You can select `Input` port as the **Output mask source** on the block. However, in this case, the `Mask` input signal must be a vector of data type `ufix1`.
- If you select **Reset on nonzero input**, the input to the `Rst` port must have data type `Boolean`.
- Outputs of type `double` are not supported for HDL code generation. All other output types (including bit packed outputs) are supported.
- You cannot generate HDL for this block inside a `Resettable Synchronous Subsystem`.
- You cannot generate HDL for this block inside a `Triggered Subsystem`, if the **Use trigger signal as clock** option is selected.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Prelookup

Compute index and fraction for Interpolation Using Prelookup block (HDL Coder)

## Description

The Prelookup block is available with Simulink.

For information about the simulation behavior and block parameters, see Prelookup.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

- “Required Block Settings” on page 3-466

- “Table Data Typing and Sizing” on page 3-466

### Required Block Settings

- **Breakpoint data:** For **Source**, select Dialog.
- **Specification:** You can select either Explicit values or Even spacing.
- **Index search method:** Select Evenly spaced points.
- **Extrapolation method:** Select Clip.
- **Diagnostic for out-of-range input:** Select Error.
- **Use last breakpoint for input at or above upper limit:** Select this check box.
- **Breakpoint:** For **Data Type**, select Inherit: Same as input.
- **Integer rounding mode:** Select Zero, Floor, or Simplest.

### Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. If the breakpoint spacing does not meet this condition, HDL Coder issues a warning. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- All ports on the block require scalar values.
- The coder permits floating-point data for breakpoints.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Probe

Output signal attributes, including width, dimensionality, sample time, and complex signal flag (HDL Coder)

## Description

The Probe block outputs selected information about the input signal to the block. Use the Block Parameters of the Probe block to output these input signal attributes:

- Width
- Dimensions
- Sample time and offset
- Flag that indicates whether the signal is complex-valued

The block has one input port. The number of output ports depends on the input signal attributes that you select for probing. The Probe block is available with Simulink. For information about the simulation behavior and block parameters, see Probe.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

# Product

Multiply and divide scalars and nonscalars or multiply and invert matrices (HDL Coder)

## Description

The Product block is available with Simulink.

For information about the simulation behavior and block parameters, see Product.

## Divide or Reciprocal

For block implementations of the Product block in divide mode or reciprocal mode, see Divide.

---

**Note** In divide mode, **Number of Inputs** is set to  $*/$ .

In reciprocal mode, **Number of Inputs** is set to  $/$ .

---

## HDL Architecture

The default Linear implementation generates a chain of N operations (multipliers) for N inputs.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Native Floating Point**

### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### **NFPCustomLatency**

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### **MantissaMultiplyStrategy**

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## **Complex Data Support**

The default (linear) implementation supports complex data.



Complex division is not supported. For block implementations of the Product block in divide mode or reciprocal mode, see Divide.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Product of Elements

Copy or invert one scalar input, or collapse one nonscalar input (HDL Coder)

### Description

The Product of Elements block is available with Simulink.

For information about the simulation behavior and block parameters, see Product of Elements.

### HDL Architecture

HDL Coder supports Tree and Cascade architectures for Product or Product of Elements blocks that have a single vector input with multiple elements.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
Linear (default)	0	Generates a linear chain of adders to compute the sum of products.
Tree	0	Generates a tree structure of adders to compute the sum of products.
Cascade	1, when block has a single vector input port.	This implementation optimizes latency * area and is faster than the Tree implementation. It computes partial products and cascades multipliers.  See “Cascade Architecture Best Practices”.

---

**Note** The Product of Element block does not support HDL code generation with double data types in the Native Floating Point mode.

---

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### **NFPCustomLatency**

To specify a value, set **LatencyStrategy** to Custom. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### **MantissaMultiplyStrategy**

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## **Complex Data Support**

The default (linear) implementation supports complex data.

Complex division is not supported. For block implementations of the Product block in divide mode or reciprocal mode, see Divide.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# QPSK Demodulator Baseband

Demodulate QPSK-modulated data (HDL Coder)

## Description

The QPSK Demodulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see QPSK Demodulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# QPSK Modulator Baseband

Modulate using quaternary phase shift keying method (HDL Coder)

## Description

The QPSK Modulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see QPSK Modulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Raised Cosine Receive Filter

Apply pulse shaping by downsampling signal using raised cosine FIR filter (HDL Coder)

## Description

The Raised Cosine Receive Filter is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Raised Cosine Receive Filter.

This block is a subsystem that contains a FIR Decimation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See Subsystem and FIR Decimation for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Receive Filter block and execute this command.

```
set_param(gcf, 'LinkStatus', 'inactive')
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Raised Cosine Transmit Filter

Apply pulse shaping by upsampling signal using raised cosine FIR filter (HDL Coder)

## Description

The Raised Cosine Transmit Filter is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Raised Cosine Transmit Filter.

This block is a subsystem that contains a FIR Interpolation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See Subsystem and FIR Interpolation for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Transmit Filter block and execute this command.

```
set_param(gcf, 'LinkStatus', 'inactive')
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2015a**

# Rate Transition

Handle transfer of data between blocks operating at different rates (HDL Coder)

## Description

The Rate Transition block is available with Simulink.

For information about the simulation behavior and block parameters, see Rate Transition.

## Best Practices

When the Rate Transition block is operating at a faster input rate and slower output rate, it is good practice to follow the Rate Transition block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation”.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

- Sample rate cannot be 0 or Inf for block input or output ports.
- **Ensure data integrity during data transfer** must be enabled.
- **Ensure deterministic data transfer (maximum delay)** must be enabled.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal (HDL Coder)

## Description

The Real-Imag to Complex block is available with Simulink.

For information about the simulation behavior and block parameters, see Real-Imag to Complex.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Reciprocal Sqrt

Calculate square root, signed square root, or reciprocal of square root (HDL Coder)

## Description

The Reciprocal Sqrt block is available with Simulink.

For information about the simulation behavior and block parameters, see Reciprocal Sqrt.

## HDL Code Generation Support

For the Sqrt block with **Function** set to `rSqrt`, the code generator supports various architectures and data types. The `sqrtfunction` architecture supports code generation in native floating-point mode. For this architecture, you can specify the **HandleDenormals** and **LatencyStrategy** settings from the **Native Floating Point** tab in the HDL Block Properties dialog box.

Architecture	Fixed-Point	Native Floating-Point	HandleDenormals	LatencyStrategy
<code>sqrtfunction</code>	—	✓	✓	✓
<code>recipsqrtnewton</code>	✓	—	—	—
<code>recipsqrtnewtonsinglerate</code>	✓	—	—	—

## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
SqrtFunction (default)	0	Use a bitset shift/addition algorithm.  The SqrtFunction architecture is equivalent to the SqrtBitset architecture with UseMultiplier set to off.
RecipSqrtNewton	Iterations + 2	Use the iterative Newton method. Select this option to optimize area.
RecipSqrtNewtonSingleRate	(Iterations * 4) + 5	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### Iterations

Number of iterations for Newton method. The default is 3.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## Native Floating Point

---

**Note** The Product of Element block does not support HDL code generation with double data types in the Native Floating Point mode.

---

### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “MantissaMultiplyStrategy”.

## Restrictions

- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Rectangular QAM Demodulator Baseband

Demodulate rectangular-QAM-modulated data (HDL Coder)

## Description

The Rectangular QAM Demodulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Rectangular QAM Demodulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

- The block does not support single or double data types for HDL code generation.
- HDL Coder supports the following **Output type** options:
  - Integer
  - Bit is supported only if the **Decision Type** that you select is Hard decision.
- The coder requires that you set **Normalization Method** to Minimum Distance Between Symbols, with a **Minimum distance** of 2.
- The coder requires that you set **Phase offset (rad)** to a value that is a multiple of  $\pi/4$ .

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Rectangular QAM Modulator Baseband

Modulate using rectangular quadrature amplitude modulation (HDL Coder)

## Description

The Rectangular QAM Modulator Baseband block is available with Communications Toolbox.

For information about the simulation behavior and block parameters, see Rectangular QAM Modulator Baseband.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Restrictions

- The block does not support single or double data types for HDL code generation.
- When **Input Type** is set to `Bit`, the block does not support HDL code generation for input types other than `boolean` or `ufix1`.

When the input type is set to `Bit`, but the block input is actually multibit (`uint16`, for example), the Rectangular QAM Modulator Baseband block does not support HDL code generation.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Relational Operator

Perform specified relational operation on inputs (HDL Coder)

## Description

The Relational Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Relational Operator.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Native Floating Point

#### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

#### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### Complex Data Support

The `~=` and `==` operators are supported for code generation.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Relay

Switch output between two constants (HDL Coder)

## Description

The Relay block is available with Simulink.

For information about the simulation behavior and block parameters, see Relay.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Repeat

Resample input at higher rate by repeating values (HDL Coder)

## Description

The Repeat block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Repeat.

## Best Practices

The Repeat block uses fewer hardware resources than the Upsample block. If your algorithm does not require zero-padding upsampling, use the Repeat block.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Restrictions**

**Input processing** set to Columns as channels (frame based) is not supported.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

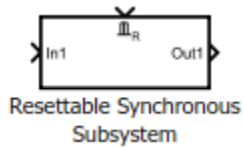
### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## Resetable Synchronous Subsystem

Represent resettable subsystem that has synchronous reset and enable behavior

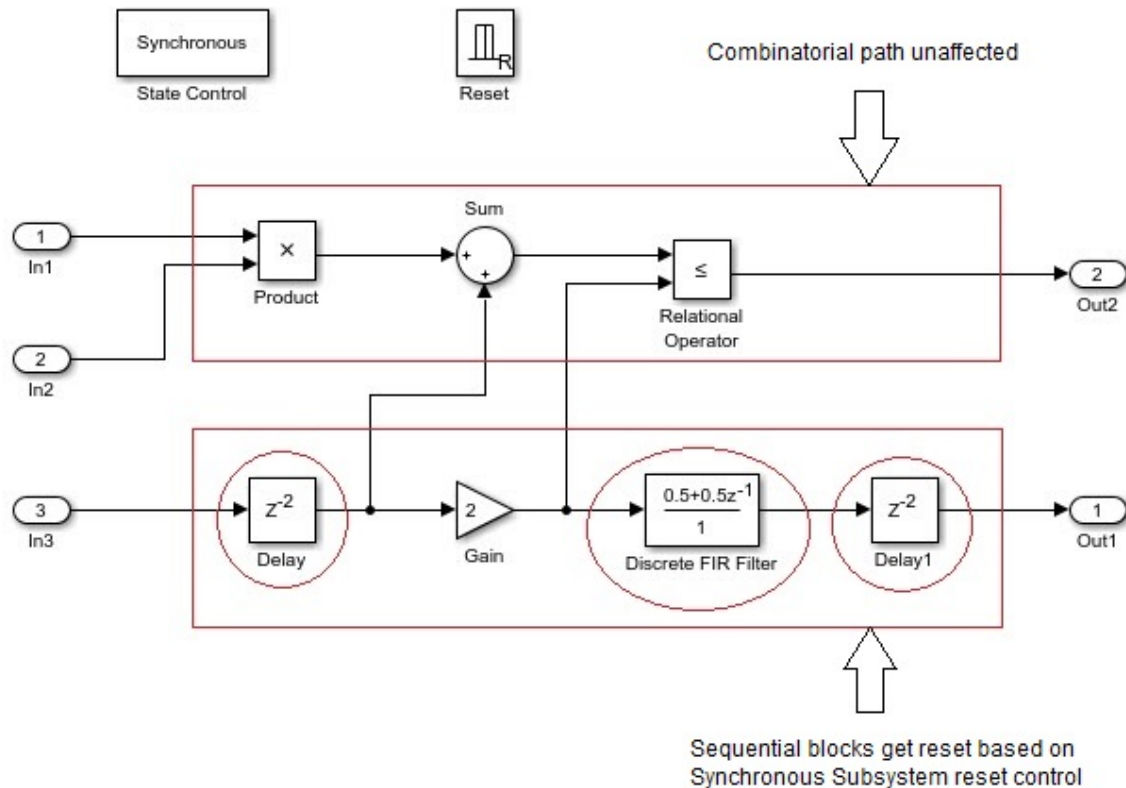


## Library

HDL Coder / HDL Subsystems

## Description

The Resetable Synchronous Subsystem uses the State Control block in **Synchronous** mode with the Resetable Subsystem block. For subsystem blocks with state, the State Control block in **Synchronous** mode provides efficient reset and enable simulation behavior on hardware.



The reset port in the Resettable Synchronous Subsystem block adds reset capability to blocks inside the subsystem that have state. This includes blocks that need not have an external reset port capability, such as filters, Stateflow Chart, and MATLAB Function blocks. For HDL code generation, the **Reset trigger type** of the Reset port is set to level hold by default.

## Data Type Support

See Inport for information on the data types accepted by a subsystem's input ports. See Outport for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” (Simulink) in the Simulink documentation.

## Parameters

### Show port labels

Display subsystem port labels on the subsystem block.

#### Settings

**Default:** FromPortIcon

none

Does not display port labels on the subsystem block.

FromPortIcon

If the corresponding port icon displays a signal name, the parameter displays the signal name on the subsystem block. Otherwise, it displays the port block name.

FromPortBlockName

Display the name of the corresponding port block on the subsystem block.

SignalName

If the signal connected to the subsystem block port is named, this parameter displays the name. Otherwise, it displays the name of the corresponding port block.

#### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Read/Write permissions

Control user access to the contents of the subsystem.

#### Settings

**Default:** ReadWrite

### ReadWrite

Enables opening and modification of subsystem contents.

### ReadOnly

Enables the opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem, and create and modify local copies of the subsystem. You cannot change the permissions or modify the contents of the original library instance.

### NoReadOrWrite

Disables the opening or modification of subsystem. If the subsystem resides in a block library, you can create links to the subsystem in a model. You cannot open, modify, change permissions, or create local copies of the subsystem.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Name of error callback function

Enter the name of the function to be called if an error occurs while Simulink software is executing the subsystem.

### Settings

**Default:** ' '

Simulink passes two arguments to the function: the subsystem handle and a character vector that specifies the error type. If no function is specified, you get a generic error message.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

### Settings

**Default:** All



**All**

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

**ExplicitOnly**

Resolve the names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked by using the signal resolution icon.

**None**

Do not resolve any workspace variable names.

**Command-Line Information**

See “Block-Specific Parameters” (Simulink) for the command-line information.

**Treat as atomic unit**

Causes Simulink to treat the subsystem as a unit when determining the execution order of block methods.

**Settings**

**Default:** Off

On

Cause Simulink to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause the execution of block methods in the subsystem to be interleaved with the execution of block methods outside the subsystem.

### Dependencies

This parameter enables:

- **Minimize algebraic loop occurrences**
- **Sample time**
- **Function packaging** (requires a Simulink Coder license)

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

### Settings

**Default:** On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

## Settings

### Default: Auto

#### Auto

Simulink Coder chooses the optimal format based on the type and number of subsystem instances in the model.

#### Inline

Simulink Coder inlines the subsystem unconditionally.

#### Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

#### Reusable function

Simulink Coder generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

This option generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy. In this case, the subsystem must be in a library.

## Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Characteristics

Data Types	Double   Single   Boolean   Base Integer   Fixed-Point   Enumerated   Bus
Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.  The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

**ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

**DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

**FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the **IP Core Generation** workflow, these target specification block property values are saved with the model. If you specify these target

specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

#### **ProcessorFPGASynchronization**

Processor/FPGA synchronization mode, specified as a character vector.

To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: Free running (default) | Coprocessing - blocking

Example: 'Free running'

#### **TestPointMapping**

To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'TestPoint','AXI4-Lite','x"108"'}}

#### **TunableParameterMapping**

To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'myParam','AXI4-Lite','x"108"'}}

#### **AXI4RegisterReadback**

To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'off' (default) | 'on'

#### **GenerateDefaultAXI4Slave**

To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'on' (default) | 'off'

### **IPCoreAdditionalFiles**

Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).

You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.

Values: '' (default) | character vector

Example: 'C:\myprojfiles\led\_blinking\_file1.vhd;C:\myprojfiles\led\_blinking\_file2.vhd;'

### **IPCoreName**

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: '' (default) | character vector

Example: 'my\_model\_name'

### **IPCoreVersion**

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: '' (default) | character vector

Example: '1.3'

## **Restrictions**

- You cannot use the State Control block in **Classic** mode or remove the State Control block from the Resettable Synchronous Subsystem block.
- The **Reset trigger type** of the Reset port inside the subsystem must be set to level hold.

- A Delay block with nonvirtual bus input signals inside a Resettable Synchronous Subsystem is not supported if you enable optimizations on the subsystem.
- HDL Coder does not support these blocks inside a Resettable Synchronous Subsystem:
  - All RAM blocks or blocks that infer a RAM in the generated HDL code. The RAM blocks include:
    - Single Port RAM
    - Simple Dual Port RAM
    - Dual Port RAM
    - Dual Rate Dual Port RAM
    - HDL FIFO
    - `hdl.RAM` system object

### **DSP System Toolbox**

- Biquad Filter
- NCO HDL Optimized

### **Communications Toolbox**

- Convolutional Encoder
- Viterbi Decoder
- PN Sequence Generator
- Integer-Output RS Decoder HDL Optimized

### **Vision HDL Toolbox**

- Demosaic Interpolator
- Edge Detector
- Histogram
- Image Filter, Median Filter, Bilateral Filter
- Line Memory
- Binary and Grayscale Morphology blocks
- Pixel Stream FIFO



**LTE HDL Toolbox**

- Turbo Decoder
- Turbo Encoder
- Convolutional Encoder
- OFDM Demodulator

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

[Enable](#) | [Enabled Synchronous Subsystem](#) | [State Control](#) | [Synchronous Subsystem](#)

### Topics

""

""

“Synchronous Subsystem Behavior with the State Control Block”

**Introduced in R2016b**

# Reshape

Change dimensionality of signal (HDL Coder)

## Description

The Reshape block is available with Simulink.

For information about the simulation behavior and block parameters, see Reshape.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# ROI Selector

Select a region of interest (ROI) from a pixel stream (HDL Coder)

## Description

The ROI Selector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see ROI Selector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2016a**

# Rounding Function

Apply rounding function to signal (HDL Coder)

## Description

The Rounding Function block is available with Simulink. For information about the simulation behavior and block parameters, see Rounding Function.

To generate HDL code, use single data types as inputs to the block, and specify the native floating point mode. In the Configuration Parameters dialog box, on the **HDL Code Generation** > **Floating Point** pane, for **Library**, select **Native Floating Point**.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “`LatencyStrategy`”.

### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “`NFPCustomLatency`”.

## Complex Data Support

This block supports code generation for complex signals.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

### Topics

“Getting Started with HDL Coder Native Floating-Point Support”

“Generate Target-Independent HDL Code with Native Floating-Point”

**Introduced in R2017a**

# Sample and Hold

Sample and hold input signal (HDL Coder)

## Description

The Sample and Hold block is available with DSP System Toolbox.

For information about the DSP System Toolbox simulation behavior and block parameters, see Sample and Hold.

HDL code for the Sample and Hold block is generated as a Triggered Subsystem. Similar restrictions apply to both blocks.

## HDL Block Properties

For HDL block property descriptions, see “HDL Block Properties: General”.

## Best Practices

When using the Sample and Hold block in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put a unit delay on the output signal. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems, such as the Sample and Hold block, can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances.



## Restrictions

The Sample and Hold block must meet the following conditions:

- The DUT (i.e., the top-level subsystem for which code is generated) must not be the Sample and Hold block.
- The trigger signal must be a scalar.
- The data type of the trigger signal must be either `boolean` or `ufix1`.
- The output of the Sample and Hold block must have an initial value of 0.
- The input, output, and trigger signal of the Sample and Hold block must run at the same rate. If one of the input or the trigger signals is an output of a Signal Builder block, see “Using the Signal Builder Block” on page 3-618 for how to match rates.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014b**

# Saturation

Limit range of signal (HDL Coder)

## Description

The Saturation block is available with Simulink.

For information about the simulation behavior and block parameters, see Saturation.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Saturation Dynamic

Bound range of input (HDL Coder)

## Description

The Saturation Dynamic block is available with Simulink.

For information about the simulation behavior and block parameters, see Saturation Dynamic.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Scope

Display signals generated during simulation (HDL Coder)

## Description

The Scope block is available with Simulink.

For information about the simulation behavior and block parameters, see Scope.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

# Extended Capabilities

## HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Selector

Select input elements from vector, matrix, or multidimensional signal (HDL Coder)

## Description

The Selector block is available with Simulink.

For information about the simulation behavior and block parameters, see Selector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Native Floating Point**

#### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### **Complex Data Support**

This block supports code generation for complex signals.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

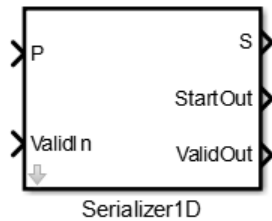
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Serializer1D

Convert vector signal to scalar or smaller vectors



## Library

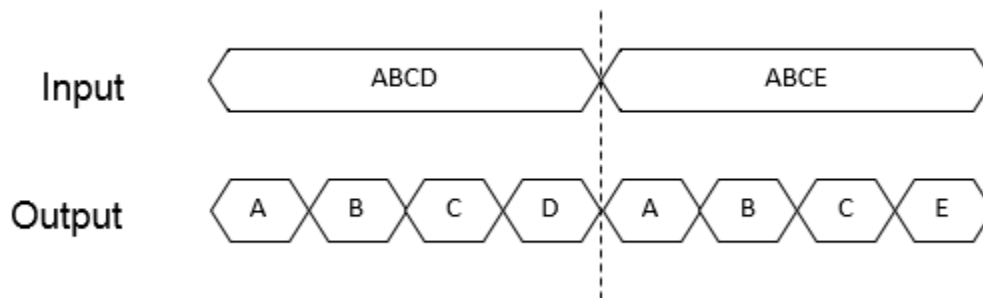
HDL Coder / HDL Operations

## Description

The Serializer1D block converts a slower vector signal into a faster stream of scalar signals or smaller size vector signals based on the **Ratio** and **Idle Cycle** values. To match the faster serialized output, the sample time changes according to this equation:

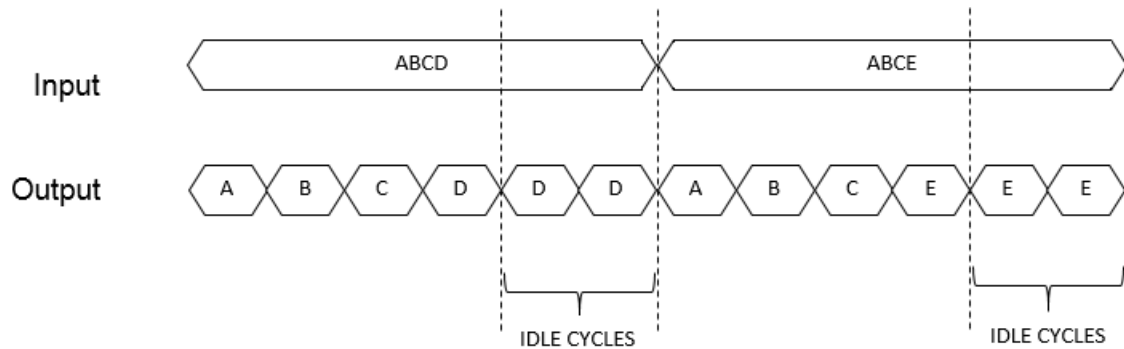
$$\text{Output Sample Time} = \text{Input Sample Time} / (\text{Ratio} + \text{Idle Cycles})$$

Consider this example where the input data is a vector of size 4 and the **Ratio** is set to 4.



The output data serializes each of the vector signals into four scalar signals. The sample time at the output is:  $Output\ Sample\ Time = Input\ Sample\ Time/4$ .

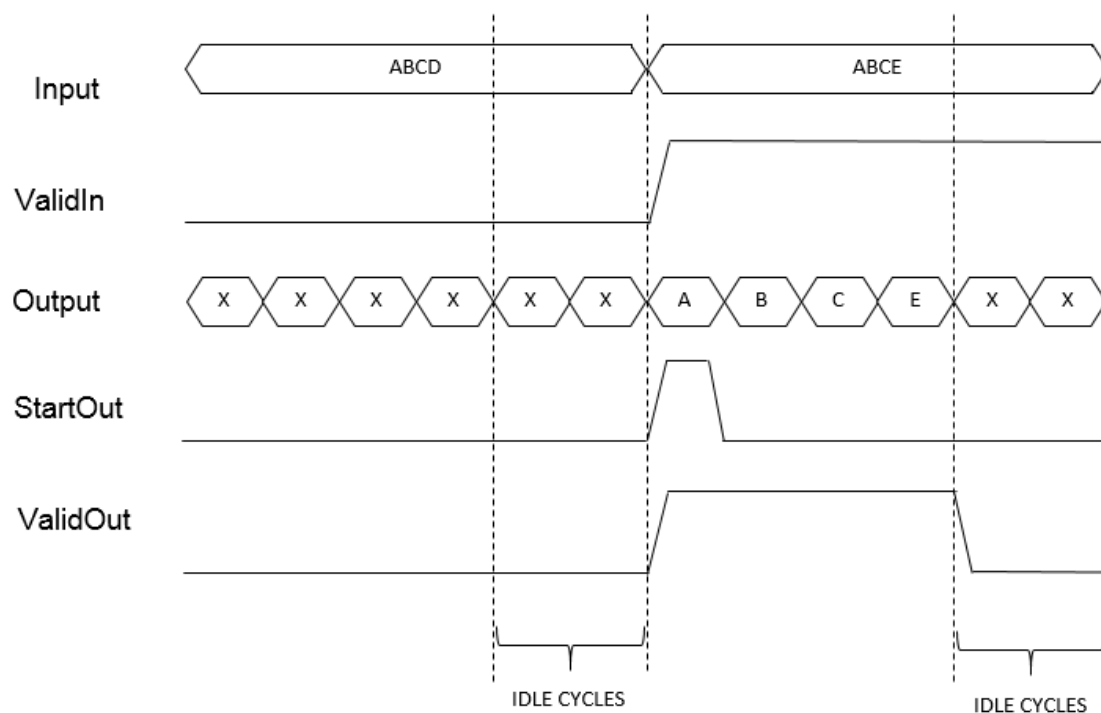
To add idle cycles at the end of each output, for **Idle Cycles**, specify an integer greater than zero. Consider this example with **Ratio** set to 4 and **Idle Cycles** set to 2.



For each slow vector signal, the output has six fast cycles consisting of the four serialized scalar signals and two idle cycles. The sample time at the output is .

The Serializer1D block provides three control signals: **ValidIn**, **ValidOut**, and **StartOut**. You can use **ValidIn** to control **ValidOut** and **StartOut**. The serialized output does not depend on **ValidIn**. To determine whether the output serialized data is valid, use **ValidIn** and **ValidOut**. If you give a high input to **ValidIn**, and if there are no idle cycles, **ValidOut** gives a high output, which indicates that the output serialized data is valid.

Consider an example that has input data as a vector of size 4, **Ratio** set to 4, **Idle Cycles** set to 2, and uses all three control signals.



For the first input vector, ABCD, **ValidIn** is false. **StartOut** and **ValidOut** become false. This means that the output data values are not valid. In the waveform, the data values are represented as X, which correspond to *don't care* values.

For the second input vector, ABCE, **ValidIn** is true. The output data serializes the vector into four scalar signals. The control signal **StartOut** becomes true at output A to indicate the start of deserialization. In the next cycle, the **StartOut** signal becomes false. **ValidOut** is true for all four output signals indicating valid output data for the four cycles. **ValidOut** becomes false for the idle cycles, and the output data values are *don't care* values.

# Parameters

### Ratio

Serialization factor, specified as a positive scalar. Default is 1.

The ratio is equal to the size of the input vector divided by the size of the output vector. Input vector size must be divisible by the ratio.

### Idle Cycles

Number of idle cycles to add at the end of each output. Default is 0.

### ValidIn

Activates the **ValidIn** port. Default is `off`.

### StartOut

Activates the **StartOut** port. Default is `off`.

### ValidOut

Activates the **ValidOut** port. Default is `off`.

### Input data port dimensions (-1 for inherited)

Size of the input data signal. Input vector size must be divisible by the ratio. By default, the block inherits size based on the context within the model.

### Input sample time (-1 for inherited)

Time interval between sample time hits, or another appropriate sample time such as continuous. By default, the block inherits sample time based on context within the model. For more information, see "Sample Time" (Simulink).

### Input signal type

Input signal type of the block, specified as `auto`, `real`, or `complex`. Default is `auto`.

# Ports

### P

Input signal to serialize. Bus data types are not supported.

### ValidIn

Input control signal. This port is available when you select the **ValidIn** check box.

Data type: Boolean

## S

Serialized output signal. Bus data types are not supported.

### StartOut

Output control signal that indicates where to start deserialization. You can use this signal as the **StartIn** input to the Deserializer1D block. To use this port, select the **StartOut** check box.

Data type: Boolean

### ValidOut

Output control signal that indicates valid output signal. You can use this signal as the **ValidIn** input to the Deserializer1D block. This port is available when you select the **ValidOut** check box.

Data type: Boolean

## HDL Architecture

---

**Note** For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

---

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

Deserializer1D

**Introduced in R2014b**

# Shift Arithmetic

Shift bits or binary point of signal (HDL Coder)

## Description

The Shift Arithmetic block is available with Simulink.

For information about the simulation behavior and block parameters, see Shift Arithmetic.

You can generate HDL code when **Bits to shift: Source** is **Dialog** or **Input port**.

## HDL Architecture

The generated VHDL code uses the `shift_right` function and `sll` operator.

The generated Verilog code uses the `>>>` and `<<<` shift operators.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Complex Data Support**

This block supports code generation for complex signals.

### **Restrictions**

When **Bits to shift: Source** is **Input port**, binary point shifting is not supported.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Sign

Indicate sign of input (HDL Coder)

## Description

The Sign block is available with Simulink.

For information about the simulation behavior and block parameters, see Sign.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Signal Conversion

Convert signal to new type without altering signal values (HDL Coder)

## Description

The Signal Conversion block is available with Simulink.

For information about the simulation behavior and block parameters, see [Signal Conversion](#).

## HDL Architecture

This block has a pass-through implementation.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “[ConstrainedOutputPipeline](#)”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[InputPipeline](#)”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “[OutputPipeline](#)”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal (HDL Coder)

## Description

The Signal Specification block is available with Simulink.

For information about the simulation behavior and block parameters, see Signal Specification.

## HDL Architecture

This block has a pass-through implementation.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

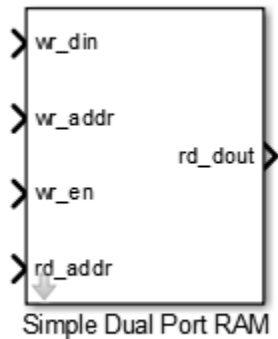
### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Simple Dual Port RAM

Dual port RAM with single output port



## Library

HDL Coder / HDL Operations

## Description

The Simple Dual Port RAM block models RAM that supports simultaneous read and write operations, and has a single output port for read data. You can use this block to generate HDL code that maps to RAM in most FPGAs.

The Simple Dual Port RAM is similar to the Dual Port RAM, but the Dual Port RAM has both a write data output port and a read data output port.

## Read-During-Write Behavior

During a write operation, if a read operation occurs at the same address, old data appears at the output.

# Parameters

### Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

# Ports

The block has the following ports:

`wr_din`

Write data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`wr_addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`wr_en`

Write enable.

Data type: Boolean

`rd_addr`

Read address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`rd_dout`

Output data from read address, `rd_addr`.

# HDL Architecture

This block has a single, default HDL architecture.



HDL code generated for RAM blocks has:

- A latency of one clock cycle for read data output.
- No reset signal, because some synthesis tools do not infer a RAM from HDL code if it includes a reset.

Code generation for a RAM block creates a separate file, *blockname.ext*. *blockname* is derived from the name of the RAM block. *ext* is the target language file name extension.

## RAM Initialization

Code generated to initialize a RAM is intended for simulation only. Synthesis tools can ignore this code.

## Implement RAM With or Without Clock Enable

The HDL block property, `RAMArchitecture`, enables or suppresses generation of clock enable logic for all RAM blocks in a subsystem. You can set `RAMArchitecture` to the following values:

- `WithClockEnable` (default): Generates RAM using HDL templates that include a clock enable signal, and an empty RAM wrapper.
- `WithoutClockEnable`: Generates RAM without clock enables, and a RAM wrapper that implements the clock enable logic.

Some synthesis tools do not infer RAM with a clock enable. If your synthesis tool does not support RAM structures with a clock enable, and cannot map your generated HDL code to FPGA RAM resources, set `RAMArchitecture` to `WithoutClockEnable`.

To learn how to generate RAM without clock enables for your design, see the Getting Started with RAM and ROM example. To open the example, at the command prompt, enter:

```
hdlcoderramrom
```

## RAM Inference Limitations

If you use RAM blocks to perform concurrent read and write operations, verify the read-during-write behavior in hardware. The read-during-write behavior of the RAM blocks in Simulink matches that of the generated behavioral HDL code. However, if a synthesis tool

does not follow the same behavior during RAM inference, it causes the read-during-write behavior in hardware to differ from the behavior of the Simulink model or generated HDL code.

Your synthesis tool might not map the generated code to RAM for the following reasons:

- Small RAM size: your synthesis tool uses registers to implement a small RAM for better performance.
- A clock enable signal is present. You can suppress generation of a clock enable signal in RAM blocks, as described in “Implement RAM With or Without Clock Enable” on page 3-543.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

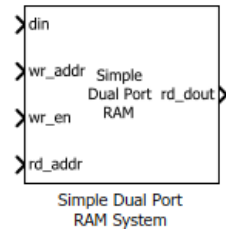
Dual Port RAM | Dual Rate Dual Port RAM | Single Port RAM

**Introduced in R2014a**

## Simple Dual Port RAM System

Simple Dual Port RAM block based on the `hdl.RAM` system object with ability to provide initial value

**Library:** HDL Coder / HDL RAMs



## Description

The blocks are MATLAB System blocks that use the `hdl.RAM` System object. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Dual Port RAM System block behaves similar to the Dual Port RAM, the Single Port RAM System behaves similar to the Single Port RAM, and so on. With the MATLAB System blocks, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.
- Obtain higher performance and support for large data memories.

## Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.
- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

## Ports

### Input

#### **din** — Write data input

Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be double, single, integer, or a fixed-point (`fi`) object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

#### **addr** — Write or Read address

Scalar (default) | Vector

Address that you write the data into when `wrEn` is true. The RAM reads the value in memory location **addr** when `wrEn` is false. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Single port`.

Data Types: `uint8` | `uint16` | `fixed point`

#### **wr\_addr** — Write address

Scalar (default) | Vector

RAM address that you write the data into. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

#### **wr\_en** — Write enable

Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location **addr** when `wrEn` is false.

Data Types: Boolean

### **rd\_addr — Read address**

Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either fixed-point (fi) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

Data Types: uint8 | uint16 | fixed point

## **Output**

### **dout — Output data**

Scalar (default) | Vector

Output data that the RAM reads from the memory location `addr` when `wrEn` is false.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Single port.

### **rd\_dout — Read data**

Scalar (default) | Vector

Old output data that the RAM reads from the memory location `rd_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

### **wr\_dout — Write data output**

Scalar (default) | Vector

New or old output data that the RAM reads from the memory location `wr_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Dual port.

## Parameters

### Specify the type of RAM — RAM type

Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- **Single port** — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- **Simple dual port** — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- **Dual port** — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

### Specify the output data for a write operation — Write output behavior

New data (default) | Old data

Behavior for Write output, specified as either:

- **'New data'** — Send out new data at the address to the output.
- **'Old data'** — Send out old data at the address to the output.

### Specify the RAM initial value — Initial simulation output of RAM

'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### HDL Architecture

The block has a MATLABSystem architecture which indicates that the block implementation uses the `hdl.RAM System` object.

#### HDL Block Properties

##### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

##### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

##### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### Complex Data Support

This block supports code generation for complex signals.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.



## See Also

### System Objects

hdl.RAM

### Blocks

Dual Port RAM System | Single Port RAM System

### Topics

“”

“”

“Implement RAM Using MATLAB Code”

“HDL Code Generation for System Objects”

**Introduced in R2017b**

# Sine

Implement fixed-point sine wave using lookup table approach that exploits quarter wave symmetry (HDL Coder)

## Description

The Sine block is available with Simulink.

For information about the simulation behavior and block parameters, see Sine, Cosine.

## HDL Architecture

The HDL code implements Sine using the quarter-wave lookup table you specify in the Simulink block parameters.

To avoid generating a division operator ( $/$ ) in the HDL code, for **Number of data points for lookup table**, enter  $(2^n)+1$ .  $n$  is an integer.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Limitations

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

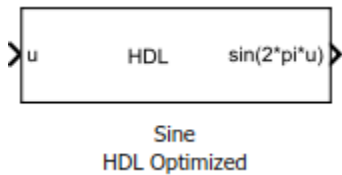
### See Also

Cosine | Cosine HDL Optimized | Sine HDL Optimized

**Introduced in R2014a**

## Sine HDL Optimized

Implement fixed-point sine wave by using lookup table approach optimized for HDL code generation



## Library

HDL Coder / Lookup Tables

## Description

The Sine HDL Optimized block implements a fixed-point sine wave by using a lookup table method that exploits quarter-wave symmetry.

For the most efficient HDL implementation, configure the block with an exact power of two as the number of elements. In the Block Parameters dialog box, for **Number of data points**, specify an integer that is an exact power of two. That is, specify the lookup table data points to be  $(2^n)$ , where  $n$  is an integer. By default, the **Number of data points** is 64.

When you specify a power of two for the **Number of data points**, the lookup tables precede a register without reset after HDL code generation. The combination of the lookup table block and register without reset maps efficiently to RAM on the target device.

Depending on your selection of the **Output formula** parameter, the blocks can output these functions of the input signal:

- $\sin(2\pi u)$
- $\cos(2\pi u)$

- $\exp(i2\pi u)$
- $\sin(2\pi u)$  and  $\cos(2\pi u)$

Use the **Table data type** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

## Data Type Support

The Sine HDL Optimized block accepts signals of these data types:

- Floating point
- Built-in integer
- Fixed point
- Boolean

The output of the block is a fixed-point data type.

For more information, see “Data Types Supported by Simulink” (Simulink) in the Simulink documentation.

## Parameters

### Output formula


Select the signal(s) to output.

### Number of data points

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be  $(2^n)$ , where  $n$  is an integer.

### Table data type

Specify the table data type. You can specify an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the table data type.

#### Show data type assistant

Display the **Data Type Assistant**. In the **Data Type Assistant**, you can select the mode to specify the data type.

##### Mode

Select the mode of data type specification. If you select **Expression**, enter an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

If you select **Fixed point**, you can use the options in the **Data Type Assistant** to specify the fixed-point data type. In the **Fixed point** mode, you can choose binary point scaling, and specify the signedness, word length, fraction length, and the data type override setting.

## Characteristics

Data Types	Double   Single   Boolean   Base Integer   Fixed-Point
Sample Time	Inherited from driving block
Direct Feedthrough	Yes
Multidimensional Signals	No
Variable-Size Signals	No
Zero-Crossing Detection	No
Code Generation	Yes

## HDL Architecture

The HDL code implements the Sine HDL Optimized block by using the quarter-wave lookup table that you specify in the Simulink block parameters.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

Cosine HDL Optimized | Sine, Cosine | Trigonometric Function

**Introduced in R2016b**

# Sine Wave

Generate continuous or discrete sine wave (HDL Coder)

## Description

The Sine Wave block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see [Sine Wave](#).

## HDL Architecture

This block has a single, default HDL architecture.

## Restrictions

For HDL code generation, you must select the following Sine Wave block settings:

- **Computation method:** Table lookup
- **Sample mode:** Discrete

Output:

- The output port cannot have data types `single` or `double`.

## Complex Data Support

This block supports code generation for complex signals.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

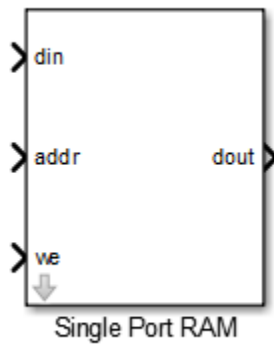
### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## Single Port RAM

Single port RAM



## Library

HDL Coder / HDL Operations

## Description

The Single Port RAM block models RAM that supports sequential read and write operations.

If you want to model RAM that supports simultaneous read and write operations, use the Dual Port RAM or Simple Dual Port RAM.

## Parameters

### Address port width

Address bit width. Minimum bit width is 2, and maximum bit width is 29. The default is 8.

### Output data during write

Controls the output data, `dout`, during a write access.

- `New data` (default): During a write, new data appears at the output port, `dout`.
- `Old data`: During a write, old data appears at the output port, `dout`.

## Ports

The block has the following ports:

`din`

Data input. The data can have any width. It inherits the width and data type from the input signal.

Data type: scalar fixed point, integer, or complex

`addr`

Write address.

Data type: scalar unsigned integer (`uintN`) or unsigned fixed point (`ufixN`) with a fraction length of 0

`we`

Write enable.

Data type: Boolean

`dout`

Output data from address, `addr`.

## HDL Architecture

This block has a single, default HDL architecture.

HDL code generated for RAM blocks has:

- A latency of one clock cycle for read data output.
- No reset signal, because some synthesis tools do not infer a RAM from HDL code if it includes a reset.

Code generation for a RAM block creates a separate file, *blockname.ext*. *blockname* is derived from the name of the RAM block. *ext* is the target language file name extension.

### RAM Initialization

Code generated to initialize a RAM is intended for simulation only. Synthesis tools can ignore this code.

### Implement RAM With or Without Clock Enable

The HDL block property, `RAMArchitecture`, enables or suppresses generation of clock enable logic for all RAM blocks in a subsystem. You can set `RAMArchitecture` to the following values:

- `WithClockEnable` (default): Generates RAM using HDL templates that include a clock enable signal, and an empty RAM wrapper.
- `WithoutClockEnable`: Generates RAM without clock enables, and a RAM wrapper that implements the clock enable logic.

Some synthesis tools do not infer RAM with a clock enable. If your synthesis tool does not support RAM structures with a clock enable, and cannot map your generated HDL code to FPGA RAM resources, set `RAMArchitecture` to `WithoutClockEnable`.

To learn how to generate RAM without clock enables for your design, see the Getting Started with RAM and ROM example. To open the example, at the command prompt, enter:

```
hdlcoderramrom
```

### RAM Inference Limitations

Depending on your synthesis tool and target device, the setting of **Output data during write** can affect RAM inference.

If you use RAM blocks to perform concurrent read and write operations, verify the read-during-write behavior in hardware. The read-during-write behavior of the RAM blocks in Simulink matches that of the generated behavioral HDL code. However, if a synthesis tool does not follow the same behavior during RAM inference, it causes the read-during-write behavior in hardware to differ from the behavior of the Simulink model or generated HDL code.

Your synthesis tool might not map the generated code to RAM for the following reasons:

- Small RAM size: your synthesis tool uses registers to implement a small RAM for better performance.
- A clock enable signal is present. You can suppress generation of a clock enable signal in RAM blocks, as described in “Implement RAM With or Without Clock Enable” on page 3-562.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

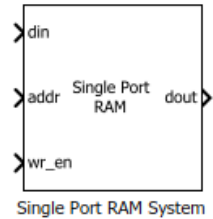
Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM

**Introduced in R2014a**

# Single Port RAM System

Single Port RAM block based on hdl.RAM system object with ability to provide initial value

**Library:** HDL Coder / HDL RAMs



## Description

The blocks are MATLAB System blocks that use the `hdl.RAM System` object. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Single Port RAM System block behaves similar to the Single Port RAM.

By using the MATLAB System block implementation, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM System` object functionality.
- Obtain higher performance and support for large data memories.

## Limitations

- The block does not support boolean inputs. Cast any boolean types to `ufix1` for input to the block.
- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

# Ports

## Input

### **din** — Write data input

Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be double, single, integer, or a fixed-point (`fi`) object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

### **addr** — Write or Read address

Scalar (default) | Vector

Address that you write the data into when `wrEn` is true. The RAM reads the value in memory location **addr** when `wrEn` is false. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Single port`.

Data Types: `uint8` | `uint16` | `fixed point`

### **wr\_addr** — Write address

Scalar (default) | Vector

RAM address that you write the data into. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

### **wr\_en** — Write enable

Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location **addr** when `wrEn` is false.



Data Types: Boolean

### **rd\_addr — Read address**

Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either fixed-point (fi) or integer, and must be real and unsigned.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

Data Types: uint8 | uint16 | fixed point

## **Output**

### **dout — Output data**

Scalar (default) | Vector

Output data that the RAM reads from the memory location `addr` when `wrEn` is false.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Single port.

### **rd\_dout — Read data**

Scalar (default) | Vector

Old output data that the RAM reads from the memory location `rd_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Simple dual port or Dual port.

### **wr\_dout — Write data output**

Scalar (default) | Vector

New or old output data that the RAM reads from the memory location `wr_addr`.

#### **Dependencies**

To enable this port, set the **Specify the type of RAM** parameter to Dual port.

## Parameters

### **Specify the type of RAM — RAM type**

Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- `Single port` — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- `Simple dual port` — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- `Dual port` — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

### **Specify the output data for a write operation — Write output behavior**

New data (default) | Old data

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `'Old data'` — Send out old data at the address to the output.

### **Specify the RAM initial value — Initial simulation output of RAM**

'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM System` object.

#### HDL Block Properties

##### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

##### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

##### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### Complex Data Support

This block supports code generation for complex signals.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

## See Also

### System Objects

hdl.RAM

### Blocks

Dual Port RAM System | Simple Dual Port RAM

## Topics

“”

“”

“Implement RAM Using MATLAB Code”

“HDL Code Generation for System Objects”

**Introduced in R2017b**

# Spectrum Analyzer

Display frequency spectrum of time-domain signals (HDL Coder)

## Description

The Spectrum Analyzer block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Spectrum Analyzer.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## Sqrt

Calculate square root, signed square root, or reciprocal of square root (HDL Coder)

### Description

The Sqrt block is available with Simulink.

For information about the simulation behavior and block parameters, see Sqrt.

### HDL Code Generation Support

For the Sqrt block with **Function** set to `sqrt`, the code generator supports various architectures and data types. The `sqrtfunction` architecture supports code generation in native floating-point mode. For this architecture, you can specify the **HandleDenormals** and **LatencyStrategy** settings from the **Native Floating Point** tab in the HDL Block Properties dialog box.

Architecture	Fixed-Point	Native Floating-Point	HandleDenormals	LatencyStrategy
<code>sqrtfunction</code>	✓	✓	✓	✓
<code>sqrtnewton</code>	✓	—	—	—
<code>sqrtnewtonsingle rate</code>	✓	—	—	—
<code>sqrtbitset</code>	✓	—	—	—

### HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Parameter	Additional cycles of latency	Description
SqrtFunction (default)	None	0	Use a bitset shift/addition algorithm.  The SqrtFunction architecture is equivalent to the SqrtBitset architecture with UseMultiplier set to off.
SqrtBitset	UseMultiplier	0	Algorithm depends on the UseMultiplier setting: <ul style="list-style-type: none"> <li>off (default): Use a bitset shift/addition algorithm.</li> <li>on: Use a multiply/add algorithm.</li> </ul>
SqrtNewton	Iterations	Iterations + 3	Use the iterative Newton method. Select this option to optimize area.  The default value for Iterations is 3.  The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, HDL Coder generates a message.
SqrtNewtonSingleRate	Iterations	(Iterations * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.  The default value for Iterations is 3.  The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, the coder generates a message.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **Iterations**

Number of iterations for SqrtNewton or SqrtNewtonSingleRate implementation.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### **UseMultiplier**

Select algorithm for SqrtBitset implementation. The default is `off`.

## Native Floating Point

---

**Note** The Sqrt block does not support HDL code generation with `double` data types in the Native Floating Point mode.

---

#### **HandleDenormals**

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.



**LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “`LatencyStrategy`”.

**NFPCustomLatency**

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “`NFPCustomLatency`”.

## Restrictions

- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

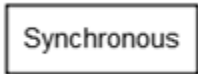
### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# State Control

Specify synchronous reset and enable behavior for blocks with state



## Library

HDL Coder / HDL Subsystems

## Description

Use the State Control block to toggle subsystem behavior between the default Simulink simulation behavior and the synchronous hardware simulation behavior.

- For default Simulink simulation behavior, set **State control** to **Classic**. The simulation behavior in **Classic** mode is the same as when you do not use the State Control block inside the subsystem.
- For synchronous hardware simulation behavior, set **State control** to **Synchronous**. The State Control block in **Synchronous** mode improves the HDL simulation behavior of blocks with state, or blocks that have reset or enable ports. When use the **Synchronous** mode of the block, the generated HDL code uses fewer hardware resources and the Simulink simulation behavior closely matches that of the digital hardware.

See “Synchronous Subsystem Behavior with the State Control Block”.

## Parameters

### State control

Specify whether to use synchronous or classic semantics. The default is Synchronous.

## HDL Architecture

This block has a single, default HDL architecture. HDL Coder does not generate HDL code specific to the State Control block. How you set the State Control block affects other blocks inside the subsystem that have state.

## Limitations

### Subsystem-level Limitations

- Conditional subsystems using classic semantics cannot have subsystems with synchronous semantics inside them.
- You cannot flatten a synchronous subsystem up into a classic system.
- Conditional subsystems must be single rate when you use the State Control block in synchronous mode.
- Synchronous Enabled Subsystem cannot contain reset subsystems or a reset parameter port. For example, you cannot have a Delay block with an external reset port inside the subsystem.
- All action subsystems connected to If and Switch Case blocks must have the same semantics, either classic or synchronous.
- These blocks are not supported in synchronous mode:
  - For Iterator Subsystem
  - While Iterator Subsystem
  - Function-Call Subsystem
  - Triggered Subsystem

### Model-Level Limitations

- Variable-size signals are not supported with synchronous semantics.
- Synchronous semantics do not propagate across model boundaries. If your parent model has synchronous semantics, any referenced model must have synchronous semantics explicitly specified. At the root level of each referenced model, add a State Control block with the **State control** parameter set to Synchronous.

### Supported Block Modes

The following restrictions apply to blocks in synchronous mode:

- Delay block: When you have an external reset port, set the **External reset** to Level hold.
- The method `ssSetStateSemanticsClassicAndSynchronous` must be set to `true`.
- Stateflow Chart: Set the **State Machine Type** to Moore.
- MATLAB Function block:
  - You cannot have System Objects inside the MATLAB Function block.
  - If you use nondirect feedthrough in a MATLAB Function block, do not program the outputs to rely on inputs or updated persistent variables. The MATLAB Function block must drive the outputs from persistent variables.

To use nondirect feedthrough, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box. See “Use Nondirect Feedthrough in a MATLAB Function Block” (Simulink).

### Unsupported Blocks

The following blocks are not allowed in synchronous mode:

- The set of unit delay blocks in the **Additional Math & Discrete > Additional Discrete** sublibrary in Simulink, such as the Unit Delay Resettable and Unit Delay External IC blocks
- Simulink blocks with **Input processing** set to Columns as channels (frame based), where this parameter applies.
- Continuous time blocks and blocks with continuous rate
- Discrete-Time Integrator with reset port
- From Workspace
- Trigger
- LMS Filter
- HDL Minimum Resource FFT
- DC Blocker
- PN Sequence Generator

- Convolutional Interleaver and Convolutional Deinterleaver
- General Multiplexed Interleaver and General Multiplexed Deinterleaver
- Convolutional Encoder and Viterbi Decoder
- Sample and Hold

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

[Enable](#) | [Enabled Subsystem](#) | [Enabled Synchronous Subsystem](#) | [Resettable Synchronous Subsystem](#)

### Topics

“”

“”

“Synchronous Subsystem Behavior with the State Control Block”

**Introduced in R2016a**

# State Transition Table

Represent modal logic in tabular format (HDL Coder)

## Description

The State Transition Table block is available with Stateflow.

For information about the simulation behavior and block parameters, see State Transition Table.

## Tunable Parameters

You can use a tunable parameter in a State Transition Table intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

## HDL Architecture

This block has a single, default HDL architecture.

## Active State Output

To generate an output port in the HDL code that shows the active state, select **Create output port for monitoring** in the Properties window of the chart. The output is an enumerated data type. See “Simplify Stateflow Charts by Incorporating Active State Output” (Stateflow).

## HDL Block Properties

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

**ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

**DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

**InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

**InstantiateFunctions**

Generate a VHDL entity or Verilog module for each function. The default is `off`. See also “InstantiateFunctions”.

**LoopOptimization**

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

**MapPersistentVarsToRAM**

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**UseMatrixTypesInHDL**

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

### VariablesToPipeline

**Warning** VariablesToPipeline is not recommended. Use `coder.hdl.pipeline` instead.

---

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

### See Also

Chart | Sequence Viewer | Truth Table

**Introduced in R2014a**



# Stop Simulation

Stop simulation when input is nonzero (HDL Coder)

## Description

The Stop Simulation block is available with Simulink.

For information about the simulation behavior and block parameters, see Stop Simulation.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Subsystem

Represent system within another system (HDL Coder)

## Description

The Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see [Subsystem](#).

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See [“Customize Black Box or HDL Cosimulation Interface”](#).

---

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

#### DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

#### FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

## Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the **IP Core Generation** workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

### ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: Free running (default) | Coprocessing - blocking

Example: 'Free running'

### TestPointMapping

To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'TestPoint','AXI4-Lite','x"108"'}}

### TunableParameterMapping

To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: ' ' (default) | cell array of character vectors

Example: '{{'myParam', 'AXI4-Lite', 'x"108"'}}'

### AXI4RegisterReadback

To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'off' (default) | 'on'

### GenerateDefaultAXI4Slave

To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'on' (default) | 'off'

### IPCoreAdditionalFiles

Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).

You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.

Values: ' ' (default) | character vector

Example: 'C:\myprojfiles\led\_blinking\_file1.vhd;C:\myprojfiles\led\_blinking\_file2.vhd;'

### IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: ' ' (default) | character vector

Example: 'my\_model\_name'

### **IPCoreVersion**

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: ' ' (default) | character vector

Example: '1.3'

## **Restrictions**

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## **See Also**

### **Topics**

“External Component Interfaces”

“Generate Black Box Interface for Subsystem”

**Introduced in R2014a**

# Subtract

Add or subtract inputs (HDL Coder)

## Description

The Subtract block is available with Simulink.

For information about the simulation behavior and block parameters, see Subtract.

## HDL Architecture

The default `Linear` implementation generates a chain of `N` operations (adders) for `N` inputs.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



## Native Floating Point

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

## Complex Data Support

The default Linear implementation supports complex data.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Sum

Add or subtract inputs (HDL Coder)

## Description

The Sum block is available with Simulink.

For information about the simulation behavior and block parameters, see Sum.

## HDL Architecture

The default `Linear` implementation generates a chain of `N` operations (adders) for `N` inputs.

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

### LatencyStrategy

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, `Zero`, or `Custom` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

### NFPCustomLatency

To specify a value, set **LatencyStrategy** to `Custom`. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

## Complex Data Support

The default Linear implementation supports complex data.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Sum of Elements

Add or subtract inputs (HDL Coder)

### Description

The Sum of Elements block is available with Simulink.

For information about the simulation behavior and block parameters, see Sum of Elements.

### HDL Architecture

HDL Coder supports Tree and Cascade architectures for Sum of Elements blocks that have a single vector input with multiple elements.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
Linear	0	Generates a linear chain of adders to compute the sum of products.
Tree	0	Generates a tree structure of adders to compute the sum of products.
Cascade	1, when block has a single vector input port.	This implementation optimizes latency * area and is faster than the Tree implementation. It computes partial sums and cascades adders.  See “Cascade Architecture Best Practices”.

---

**Note** To use the **LatencyStrategy** setting in the **Native Floating Point** tab of the HDL Block Properties dialog box, specify **Linear** or **Tree** as the HDL Architecture.

---

## HDL Block Properties

### General

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Native Floating Point

---

**Note** The Sum of Elements block does not support HDL code generation with **double** data types in the **Native Floating Point** mode.

---

### **LatencyStrategy**

Specify whether to map the blocks in your design to **inherit**, **Max**, **Min**, or **Zero** for the floating-point operator. The default is **inherit**. See also “LatencyStrategy”.

### **NFPCustomLatency**

To specify a value, set **LatencyStrategy** to **Custom**. HDL Coder adds latency equal to the value that you specify for the **NFPCustomLatency** setting. See also “NFPCustomLatency”.

### Complex Data Support

The `Linear` implementation supports complex data.

The `Tree` implementation supports complex data with `+` for the **List of signs** block parameter. With native floating point support, the `Tree` implementation supports complex data with both `+` and `-` for **List of signs**.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Switch

Switch output between first input and third input based on value of second input (HDL Coder)

## Description

The Switch block is available with Simulink.

For information about the simulation behavior and block parameters, see Switch.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior



## Library

HDL Coder / HDL Subsystems

## Description

A Synchronous Subsystem is a subsystem that uses the Synchronous mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create a Synchronous Subsystem, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to Synchronous inside a subsystem.

## Data Type Support

See Inport for information on the data types accepted by a subsystem's input ports. See Outport for information on the data types output by a subsystem's output ports.

For more information, see “Data Types Supported by Simulink” (Simulink) in the Simulink documentation.

## Parameters

### Show port labels

Cause Simulink software to display labels for the subsystem's ports on the subsystem's icon.

#### Settings

**Default:** FromPortIcon

none

Does not display port labels on the subsystem block.

FromPortIcon

If the corresponding port icon displays a signal name, display the signal name on the subsystem block. Otherwise, display the port block's name.

FromPortBlockName

Display the name of the corresponding port block on the subsystem block.

SignalName

If a name exists, display the name of the signal connected to the port on the subsystem block; otherwise, the name of the corresponding port block.

#### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

### Read/Write permissions

Control user access to the contents of the subsystem.

#### Settings

**Default:** ReadWrite

ReadWrite

Enables opening and modification of subsystem contents.

### ReadOnly

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

### NoReadOrWrite

Disables opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Name of error callback function

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

### Settings

**Default:** ' '

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Permit hierarchical resolution

Specify whether to resolve names of workspace variables referenced by this subsystem.

### Settings

**Default:** All

### All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

### ExplicitOnly

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

### None

Do not resolve any workspace variable names.

## Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Treat as atomic unit

Causes Simulink software to treat the subsystem as a unit when determining the execution order of block methods.

## Settings

**Default:** Off

On

Cause Simulink software to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink software invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

Off

Cause Simulink software to treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

## Dependencies

This parameter enables:

- **Minimize algebraic loop occurrences**
- **Sample time**
- **Function packaging** (requires a Simulink Coder license)

## Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Treat as grouped when propagating variant conditions

Causes Simulink software to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

## Settings

**Default:** On

On

Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

Off

Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

## Dependency

**Treat as grouped when propagating variant conditions** enables this parameter.

## Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Function packaging

Specify the code format to be generated for an atomic (nonvirtual) subsystem.

### Settings

**Default:** Auto

#### Auto

Simulink Coder software chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

#### Inline

Simulink Coder software inlines the subsystem unconditionally.

#### Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

#### Reusable function

Simulink Coder software generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

### Command-Line Information

See “Block-Specific Parameters” (Simulink) for the command-line information.

## Characteristics

Data Types	Double   Single   Boolean   Base Integer   Fixed-Point   Enumerated   Bus
------------	---

Multidimensional Signals	Yes
Variable-Size Signals	Yes
HDL Code Generation	Yes

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

### **ClockRatePipelining**

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### **FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### **StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

## **Target Specification**

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.



In the HDL Workflow Advisor, if you use the **IP Core Generation** workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

### **ProcessorFPGASynchronization**

Processor/FPGA synchronization mode, specified as a character vector.

To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: Free running (default) | Coprocessing - blocking

Example: 'Free running'

### **TestPointMapping**

To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'TestPoint', 'AXI4-Lite', 'x"108"'}}

### **TunableParameterMapping**

To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow.

Values: '' (default) | cell array of character vectors

Example: '{{'myParam', 'AXI4-Lite', 'x"108"'}}

### **AXI4RegisterReadback**

To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'off' (default) | 'on'

### **GenerateDefaultAXI4Slave**

To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow.

Values: 'on' (default) | 'off'

### **IPCoreAdditionalFiles**

Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).

You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.

Values: '' (default) | character vector

Example: 'C:\myprojfiles\led\_blinking\_file1.vhd;C:\myprojfiles\led\_blinking\_file2.vhd;'

### **IPCoreName**

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: '' (default) | character vector

Example: 'my\_model\_name'

### **IPCoreVersion**

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: '' (default) | character vector

Example: '1.3'

## Restrictions

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

[Enable](#) | [Enabled Synchronous Subsystem](#) | [Resettable Synchronous Subsystem](#) | [State Control](#)

## Topics

“”  
“”

“Synchronous Subsystem Behavior with the State Control Block”

**Introduced in R2016a**

# Tapped Delay

Delay scalar signal multiple sample periods and output the delayed versions (HDL Coder)

## Description

The Tapped Delay block is available with Simulink.

For information about the simulation behavior and block parameters, see Tapped Delay.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Terminator

Terminate unconnected output port (HDL Coder)

## Description

The Terminator block is available with Simulink.

For information about the simulation behavior and block parameters, see Terminator.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Time Scope

Display time-domain signals (HDL Coder)

## Description

The Time Scope block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Time Scope.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# To File

Write data to file (HDL Coder)

## Description

The To File block is available with Simulink.

For information about the simulation behavior and block parameters, see To File.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



## To VCD File

Generate value change dump (VCD) file (HDL Coder)

### Description

The To VCD File block is available with HDL Verifier.

For information about the simulation behavior and block parameters, see To VCD File.

### HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

**Introduced in R2014a**

# To Workspace

Write data to MATLAB workspace (HDL Coder)

## Description

The To Workspace block is available with Simulink.

For information about the simulation behavior and block parameters, see [To Workspace](#).

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

# Extended Capabilities

## HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Trigger

Add trigger port to model or subsystem (HDL Coder)

## Description

The Trigger block is available with Simulink.

For information about the simulation behavior and block parameters, see [Trigger](#).

## HDL Architecture

This block has a single, default HDL architecture.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

Triggered Subsystem

**Introduced in R2014a**

# Triggered Subsystem

Represent subsystem whose execution is triggered by external input (HDL Coder)

## Description

A triggered subsystem is a subsystem that receives a control signal via a Trigger block. The triggered subsystem executes for one cycle each time a trigger event occurs. For detailed information on how to define trigger events and configure triggered subsystems, see “Using Triggered Subsystems” (Simulink).

## Best Practices

When using triggered subsystems in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put unit delays on Triggered Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances and the number of output ports per subsystem.

## Using the Signal Builder Block

When you connect outputs from a Signal Builder block to a triggered subsystem, you might need to use a Rate Transition block. To run all triggered subsystem ports at the same rate:

- If the trigger source is a Signal Builder block, but the other triggered subsystem inputs come from other sources, insert a Rate Transition block into the signal path before the trigger input.

- If all inputs (including the trigger) come from a Signal Builder block, they have the same rate, so special action is not required.

## Using the Trigger as Clock

You can generate code that uses the trigger signal as a clock with the `TriggerAsClock` property. See “Use Trigger As Clock in Triggered Subsystems”.

## HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

## Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

### **BalanceDelays**

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

### **ClockRatePipelining**

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### **FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### **StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

## Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

## Restrictions

HDL Coder supports HDL code generation for triggered subsystems that meet the following conditions:

- The triggered subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The trigger signal is a scalar.
- The data type of the trigger signal is either `boolean` or `ufix1`.
- Outputs of the triggered subsystem have an initial value of 0.
- All inputs and outputs of the triggered subsystem (including the trigger signal) run at the same rate. (See “Using the Signal Builder Block” on page 3-618 For information about a special case.)
- The **Show output port** parameter of the Trigger block is set to `Off`.
- If the DUT contains the following blocks, `RAMArchitecture` is set to `WithClockEnable`:
  - Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- The triggered subsystem does not contain the following blocks:
  - Discrete-Time Integrator
  - CIC Decimation
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation
  - Downsample
  - Upsample

- HDL Cosimulation blocks for HDL Verifier
- Rate Transition
- Pixel Stream FIFO (Vision HDL Toolbox)
- PN Sequence Generator, if the **Use trigger signal as clock** option is selected.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### See Also

Subsystem | Trigger

**Introduced in R2014a**



## Triggered To Workspace

Write input sample to MATLAB workspace when triggered (HDL Coder)

### Description

The Triggered To Workspace block is available with DSP System Toolbox. For information about the simulation behavior and block parameters, see Triggered To Workspace.

### HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem.
BlackBox	<p>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.</p> <p>The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

### Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

## HDL Block Properties

### General

#### **AdaptivePipelining**

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

#### **BalanceDelays**

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

#### **ClockRatePipelining**

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

#### **DSPStyle**

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

#### **FlattenHierarchy**

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

**StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

**Target Specification**

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**See Also**

Subsystem | Trigger

**Introduced in R2014a**

# Trigonometric Function

Specified trigonometric function on input (HDL Coder)

## Description

The Trigonometric Function block is available with Simulink.

For information about the simulation behavior and block parameters, see Trigonometric Function.

## HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

The Trigonometric Function block supports HDL code generation for the functions in this table.

Architecture	Function	Approximation Method	Use Pipelined Kernel Setting	Additional cycles of latency
SinCosCordic	sin	CORDIC	On	Number of iterations + 1
			Off	0
	cos	CORDIC	On	Number of iterations + 1
			Off	0
	cos + jsin	CORDIC	On	Number of iterations + 1
			Off	0
	sin cos	CORDIC	On	Number of iterations + 1
			Off	0

For an HDL implementation of the atan2 function, use the Complex to Magnitude-Angle HDL Optimized block, from the Math Operations library in DSP System Toolbox.

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### UsePipelinedKernel

Whether to use a pipelined implementation of the CORDIC algorithm in the generated code. The default is On.

Setting	Description
On (default)	Use a pipelined implementation of the CORDIC algorithm. The pipelined implementation adds latency.
Off	Use a combinatorial implementation of the CORDIC algorithm. The combinatorial implementation does not add latency. If the block is in a feedback loop, use this implementation.

## Native Floating Point

#### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

### **LatencyStrategy**

Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also “LatencyStrategy”.

## **Restrictions**

- For the `sin` and `cos` functions, only signed fixed-point data types are supported for CORDIC approximations.
- HDL Coder displays an error when you select the **SinCosCordic** architecture, **UsePipelinedKernel** is `On`, and the block is in a feedback loop.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **See Also**

`cordiccos` | `cordicsin` | `cordicsincos`

**Introduced in R2014a**

# Truth Table

Represent logical decision-making behavior with conditions, decisions, and actions (HDL Coder)

## Description

The Truth Table block is available with Stateflow.

For information about the simulation behavior and block parameters, see Truth Table.

## Tunable Parameters

You can use a tunable parameter in a Truth Table intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### **ConstMultiplierOptimization**

Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization”.

### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### **DistributedPipelining**

Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining”.

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **InstantiateFunctions**

Generate a VHDL entity or Verilog module for each function. The default is `off`. See also “InstantiateFunctions”.

### **LoopOptimization**

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

### **MapPersistentVarsToRAM**

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### **SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### **UseMatrixTypesInHDL**

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

### **VariablesToPipeline**

---

**Warning** `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

---

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.



## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

### **See Also**

[Chart](#) | [Sequence Viewer](#) | [State Transition Table](#)

**Introduced in R2014a**

## Unary Minus

Negate input (HDL Coder)

### Description

The Unary Minus block is available with Simulink.

For information about the simulation behavior and block parameters, see Unary Minus.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### **ConstrainedOutputPipeline**

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Unit Delay

Delay signal one sample period (HDL Coder)

## Description

The Unit Delay block is available with Simulink.

For information about the simulation behavior and block parameters, see Unit Delay.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

## Complex Data Support

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Unit Delay Enabled (Obsolete)

Delay signal one sample period, if external enable signal is on (HDL Coder)

---

**Note** The Unit Delay Enabled block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Unit Delay Enabled Synchronous block instead. Existing models that contain the Unit Delay Enabled block continue to work for backward compatibility.

---

## Description

The Unit Delay Enabled block delays a signal by one sample period when the external enable signal is on. While the enable is off, the block is disabled. It holds the current state at the same value and outputs that value.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also "ResetType".

## **Complex Data Support**

This block supports code generation for complex signals.

**Introduced in R2007b**

# Unit Delay Enabled Resettable (Obsolete)

Delay signal one sample period, if external enable signal is on, with external Boolean reset (HDL Coder)

---

**Note** The Unit Delay Enabled Resettable block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Unit Delay Enabled Resettable Synchronous block instead. Existing models that contain the Unit Delay Enabled Resettable block continue to work for backward compatibility.

---

## Description

The Unit Delay Enabled Resettable block can delay the signal one sample period, if external enable signal is on, with external reset as off. If the enable signal is off, the block is disabled.

When the enable and reset signals are on, the block output resets the current state.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.



### **SoftReset**

Specify on to generate reset logic for the block that is more efficient for synthesis, but does not match the Simulink behavior. The default is off. See “SoftReset”.

**Introduced in R2010a**

# Unit Delay Resettable (Obsolete)

Delay signal one sample period, with external Boolean reset (HDL Coder)

---

**Note** The Unit Delay Resettable block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Unit Delay Resettable Synchronous block instead. Existing models that contain the Unit Delay Enabled Resettable block continue to work for backward compatibility.

---

## Description

The Unit Delay Resettable block delays the signal one sample period, with external reset. The block can reset both its state and output based on an external reset signal. The block has two input ports. One input port is for the input signal and the other input port is for the external reset signal.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**SoftReset**

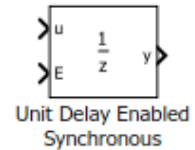
Specify on to generate reset logic for the block that is more efficient for synthesis, but does not match the Simulink behavior. The default is off. See “SoftReset”.

**Introduced in R2010a**

## Unit Delay Enabled Synchronous

Delay input signal by one sample period when external Enable signal is true

**Library:** HDL Coder / Discrete



### Description

The Unit Delay Enabled Synchronous block delays the input signal  $u$  by one sample period when the external Enable signal is true. When the Enable signal is false, the state and output signal hold the previous value. The Enable signal is true when  $E$  is not zero and false when  $E$  is zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in Synchronous mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the State Control block.

### Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use Classic semantics. The Subsystem must use Synchronous semantics.

## Ports

### Input

#### **u — Input signal**

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Enabled Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

### Input

#### **E — Enable signal**

Scalar

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

### Output

#### **y — Output signal**

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

## Parameters

#### **Initial condition — Initial output of simulation**

0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

**Programmatic Use**

**Block parameter:** InitialCondition

**Type:** character vector

**Value:** '0' | '[n]' | '[m n]'

**Default:** '0'

**Sample time — Time interval between samples**

-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

**Programmatic Use**

**Block parameter:** SampleTime

**Type:** character vector

**Value:** '-1' | '[n]' | '[m n]'

**Default:** '-1'

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generate C and C++ code using Simulink Coder.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### HDL Architecture

This block has a single, default HDL architecture.

## **HDL Block Properties**

### **InputPipeline**

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### **OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **ResetType**

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### **Complex Data Support**

This block supports code generation for complex signals.

## **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Convert floating-point algorithms to fixed point using Fixed-Point Designer.

## **See Also**

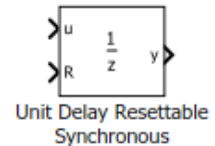
State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled Resettable Synchronous

### **Introduced in R2017b**

## Unit Delay Resettable Synchronous

Delay input signal by one sample period when external Reset signal is false

**Library:** HDL Coder / Discrete



### Description

The Unit Delay Resettable Synchronous block delays the input signal  $u$  by one sample period when the external Reset signal is false. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Reset signal is true when  $R$  is not zero and false when  $R$  is zero.

The Unit Delay Resettable Synchronous block implementation consists of a Synchronous Subsystem that contains a Resettable Delay block with a **Delay length** of one and a State Control block in Synchronous mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the State Control block.

### Limitations

- The block does not support vector inputs on the Reset port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use Classic semantics. The Subsystem must use Synchronous semantics.



## Ports

### Input

**u — Input signal**

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

### Input

**R — Reset signal**

Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

### Output

**y — Output signal**

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

## Parameters

**Initial condition — Initial output of simulation**

0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

**Programmatic Use**

**Block parameter:** InitialCondition

**Type:** character vector

**Value:** '0' | '[n]' | '[m n]'

**Default:** '0'

**Sample time — Time interval between samples**

-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

**Programmatic Use**

**Block parameter:** SampleTime

**Type:** character vector

**Value:** '-1' | '[n]' | '[m n]'

**Default:** '-1'

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generate C and C++ code using Simulink Coder.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

#### HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

### Complex Data Support

This block supports code generation for complex signals.

## Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Convert floating-point algorithms to fixed point using Fixed-Point Designer.

## See Also

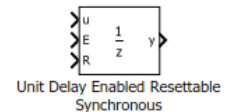
State Control | Unit Delay | Unit Delay Enabled Resettable Synchronous | Unit Delay Enabled Synchronous

### Introduced in R2017b

## Unit Delay Enabled Resettable Synchronous

Delay input signal by one sample period when external Enable signal is true and external Reset signal is false

**Library:** HDL Coder / Discrete



### Description

The Unit Delay Enabled Resettable Synchronous block combines the functionality of the Unit Delay Enabled Synchronous block and the Unit Delay Resettable Synchronous block.

The Unit Delay Enabled Resettable Synchronous block delays the input signal  $u$  by one sample period when the external Enable signal is true and when the external Reset signal is false. When the Enable signal is false, the state and output signal hold the previous value. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Enable and Reset signals are true when  $E$  and  $R$  are nonzero and false when  $E$  and  $R$  equal zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in Synchronous mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the State Control block.

### Limitations

- The block does not support vector inputs on the Reset and Enable ports.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use Synchronous semantics.

## Ports

### Input

#### **u** — Input signal

Scalar | Vector | Matrix | Array | Bus

The Unit Delay Enabled Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

### Input

#### **E** — Enable signal

Scalar

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

#### **R** — Reset signal

Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see “Data Types Supported by Simulink” (Simulink).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

### Output

#### **y** — Output signal

Scalar | Vector | Matrix | Array | Bus

Output data type always matches input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

## Parameters

### **Initial condition — Initial output of simulation**

0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

#### **Programmatic Use**

**Block parameter:** InitialCondition

**Type:** character vector

**Value:** '0' | '[n]' | '[m n]'

**Default:** '0'

### **Sample time — Time interval between samples**

-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

#### **Programmatic Use**

**Block parameter:** SampleTime

**Type:** character vector

**Value:** '-1' | '[n]' | '[m n]'

**Default:** '-1'

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generate C and C++ code using Simulink Coder.

## HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

#### ResetType

Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType”.

### Complex Data Support

This block supports code generation for complex signals.

## Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

Convert floating-point algorithms to fixed point using Fixed-Point Designer.

## See Also

State Control | Unit Delay | Unit Delay Resettable Synchronous | Unit Delay Enabled Synchronous

### Introduced in R2017b

# Upsample

Resample input at higher rate by inserting zeros (HDL Coder)

## Description

The Upsample block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Upsample.

## Best Practices

Consider whether your model can use the Repeat block instead of the Upsample block. The Repeat block uses fewer hardware resources, so it is a best practice to use Upsample only when your algorithm requires zero-padding upsampling.

See also “Multirate Model Requirements for HDL Code Generation”.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.



**OutputPipeline**

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

**Restrictions**

**Input processing** set to Columns as channels (frame based) is not supported.

**Complex Data Support**

This block supports code generation for complex signals.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

# Variable Selector

Select subset of rows or columns from input (HDL Coder)

## Description

The Variable Selector block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Variable Selector.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### **Fixed-Point Conversion**

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**

## Variant Subsystem

Represent a subsystem with multiple subsystems (HDL Coder)

### Description

The Variant Subsystem block is available with Simulink. For information about the simulation behavior and block parameters, see Variant Subsystem.

### HDL Architecture

Architecture	Description
Module (default)	Generate code for the subsystem and the blocks within the subsystem. HDL Coder generates code for only the active variant.
BlackBox	<p>Generate a black-box interface. That is, the generated HDL code includes only the input/output port definitions for the subsystem. In this way, you can use a subsystem in your model to generate an interface to existing manually written HDL code.</p> <p>The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals.</p>
No HDL	Remove the subsystem from the generated code. You can use the subsystem in simulation but treat it as a “no-op” in the HDL code.

### Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

# HDL Block Properties

## General

### AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also “AdaptivePipelining”.

### BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

### ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also “ClockRatePipelining”.

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

### DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

### FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **SharingFactor**

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

### **StreamingFactor**

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

## **Target Specification**

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

## **Restrictions**

- The DUT cannot be a Variant Subsystem.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Vector Concatenate

Concatenate input signals of same data type to create contiguous output signal (HDL Coder)

## Description

The Vector Concatenate block is available with Simulink.

For information about the simulation behavior and block parameters, see Vector Concatenate.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Restrictions**

HDL code generation does not support matrices at the input or output ports of the block .

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**



# Viterbi Decoder

Decode convolutionally encoded data using Viterbi algorithm (HDL Coder)

## Description

The Viterbi Decoder block is available with Communications Toolbox.

---

**Note** For decoding data encoded with truncated or terminated modes, or punctured codes, use the Viterbi Decoder block from LTE HDL Toolbox.

---

For information about the simulation behavior and block parameters, see Viterbi Decoder.

HDL Coder supports the following features of the Viterbi Decoder block:

- Non-recursive encoder/decoder with feed-forward trellis and simple shift register generation configuration
- Continuous mode
- Sample-based input
- Decoder rates from 1/2 to 1/7
- Constraint length from 3 to 9

## HDL Architecture

The Viterbi Decoder block decodes every bit by tracing back through a traceback depth that you define for the block. The block implements a complete traceback for each decision bit, using registers to store the minimum state index and branch decision in the traceback decoding unit. There are two methods to optimize the traceback logic: a pipelined register-based implementation or a RAM-based architecture. See the “HDL Code Generation for Viterbi Decoder” (Communications Toolbox) example.

## Register-Based Traceback

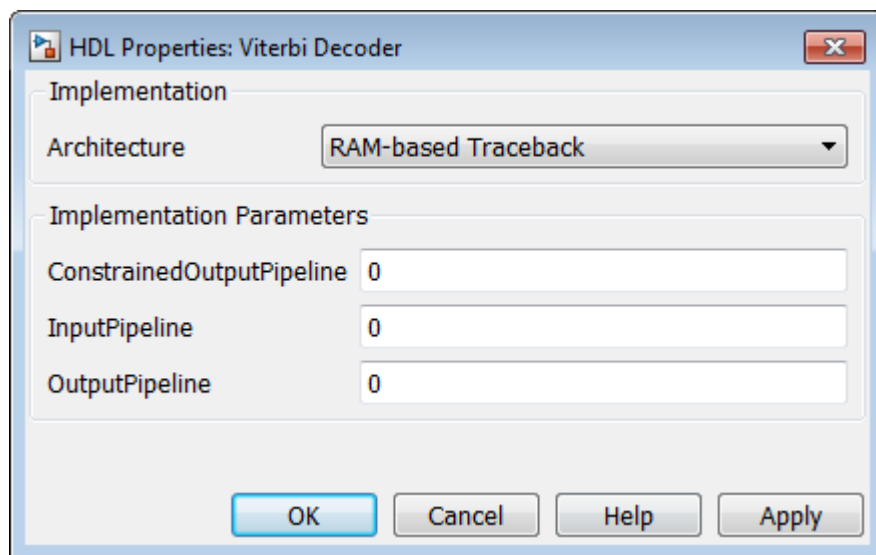
You can specify that the traceback decoding unit be pipelined to improve the speed of the generated circuit. You can add pipeline registers to the traceback unit by specifying the number of traceback stages per pipeline register.

Using the `TracebackStagesPerPipeline` implementation parameter, you can balance the circuit performance based on system requirements. A smaller parameter value indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the parameter value results in fewer registers along with a decrease in the circuit speed.

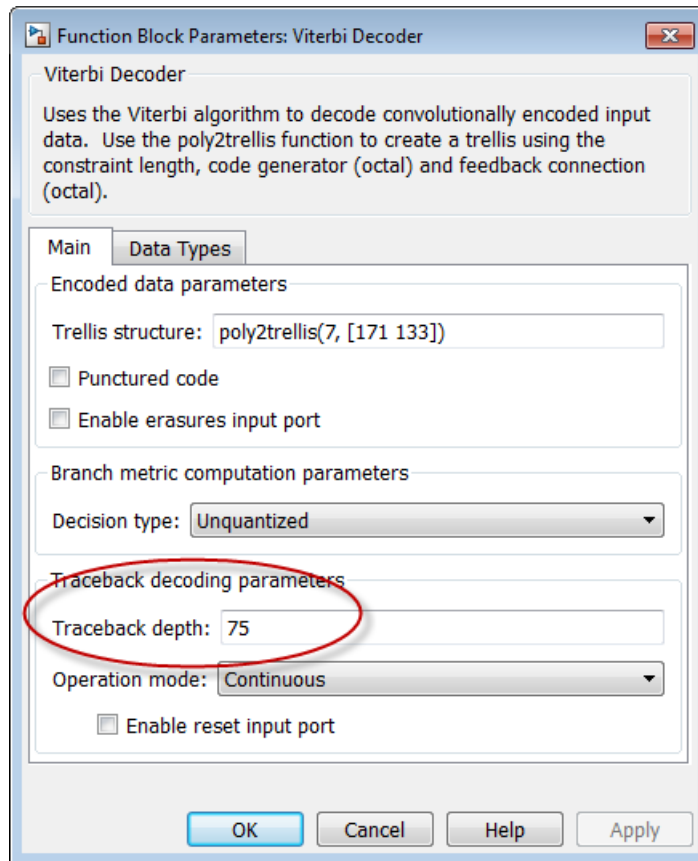
## RAM-Based Traceback

Instead of using registers, you can choose to use RAMs to save the survivor branch information. The coder does not support **Enable reset input port** when using RAM-based traceback.

- 1 Set the **Architecture** property of the Viterbi Decoder block to RAM-based Traceback.



- 2 Set the traceback depth on the Viterbi Decoder block mask.



RAM-based traceback and register-based traceback differ in the following ways:

- The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data. The register-based implementation combines the traceback and decode operations into one step. It uses the best state found from the minimum operation as the decoding initial state.
- RAM-based implementation traces back through M samples, decodes the previous M bits in reverse order, and releases one bit in order at each clock cycle. The register-based implementation decodes one bit after a complete traceback.

Because of the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback. A

longer traceback depth, for example, 10 times the constraint length, is recommended in the RAM-based traceback. This depth achieves a similar bit error rate (BER) as the register-based implementation. The size of RAM required for the implementation depends on the trellis and the traceback depth.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### TracebackStagesPerPipeline

See “Register-Based Traceback” on page 3-664.

## Restrictions

- **Punctured code:** Do not select this option. Punctured code requires frame-based input, which HDL Coder does not support.
- **Decision type:** The coder does not support the Unquantized decision type.
- **Error if quantized input values are out of range:** The coder does not support this option.
- **Operation mode:** The coder supports only the Continuous mode.
- **Enable reset input port:** When you enable both **Enable reset input port** and **Delay reset action to next time step**, HDL support is provided. You must select Continuous operation mode, and use register-based traceback.

- You cannot use the Viterbi Decoder block inside a Resettable Synchronous Subsystem.

## Input and Output Data Types

- When **Decision type** is set to `Soft decision`, the HDL implementation of the Viterbi Decoder block supports fixed-point inputs and output. For input, the fixed-point data type must be `ufixN`. `N` is the number of soft-decision bits. Signed built-in data types (`int8`, `int16`, `int32`) are not supported. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- When **Decision type** is set to `Hard decision`, the block supports input with data types `ufix1` and `Boolean`. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- The HDL implementation of the Viterbi Decoder block does not support double and single input data types. The block does not support floating point output for fixed-point inputs.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

## See Also

### Topics

“HDL Code Generation for Viterbi Decoder” (Communications Toolbox)

**Introduced in R2014a**

# Viterbi Decoder

Decode convolutionally encoded data using Viterbi algorithm (HDL Coder)

## Description

The Viterbi Decoder block is available with LTE HDL Toolbox. This block supports continuous, truncated, and terminated modes and accepts an optional erasure signal.

For information about the simulation behavior and block parameters, see Viterbi Decoder.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2018b**

# Waterfall

View vectors of data over time (HDL Coder)

## Description

The Waterfall block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see [Waterfall](#).

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

**Introduced in R2014a**



# Wrap To Zero

Set output to zero if input is above threshold (HDL Coder)

## Description

The Wrap To Zero block is available with Simulink.

For information about the simulation behavior and block parameters, see Wrap To Zero.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### **Restrictions**

The input signal and **Threshold** parameter must have equal size. For example, if the input is a two-dimensional vector, **Threshold** must also be a two-dimensional vector.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

#### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014b**

# XY Graph

Display X-Y plot of signals using MATLAB figure window (HDL Coder)

## Description

The XY Graph block is available with Simulink.

For information about the simulation behavior and block parameters, see XY Graph.

## HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

## Extended Capabilities

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

# Zero-Order Hold

Implement zero-order hold of one sample period (HDL Coder)

## Description

The Zero-Order Hold block is available with Simulink.

For information about the simulation behavior and block parameters, see Zero-Order Hold.

## HDL Architecture

This block has a single, default HDL architecture.

## HDL Block Properties

### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

## **Complex Data Support**

This block supports code generation for complex signals.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Introduced in R2014a**

## Signed Sqrt

Calculate signed square root (HDL Coder)

**Library:** HDL Coder / HDL Floating Point Operations



### Description

The Sqrt block calculates the square root, signed square root, or reciprocal of the square root on the input signal. From the **Function** parameter list, select one of the functions listed in this table.

Function	Description	Mathematical Expression	MATLAB Equivalent
sqrt	Square root of the input	$u^{0.5}$	sqrt
signedSqrt	Square root of the absolute value of the input, multiplied by the sign of the input	$\text{sign}(u) *  u ^{0.5}$	—
rSqrt	Reciprocal of the square root of the input	$u^{-0.5}$	—

The block icon changes to match the function.

### HDL Code Generation Support

For the Sqrt block with **Function** set to signedSqrt, the code generator supports SqrtFunction architecture and various data types. The SqrtFunction architecture supports code generation in native floating-point mode. For this architecture, you can specify the **HandleDenormals** and **LatencyStrategy** settings from the **Native Floating Point** tab in the HDL Block Properties dialog box.

Architecture	Fixed-Point	Native Floating-Point	HandleDenormals	LatencyStrategy
SqrtFunction	N/A	✓	✓	✓

## HDL Architecture

This block has multicycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

Architecture	Parameter	Additional cycles of latency	Description
SqrtFunction (default)	None	34 (For output data type other than single)	Use a bitset shift/addition algorithm.
		28 (For output data type single)	

## HDL Block Properties

### General

#### ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

#### InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

#### OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

### Native Floating Point

#### HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “HandleDenormals”.

#### LatencyStrategy

Specify whether to map the blocks in your design to minimum or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

## Restrictions

- Input must be a floating point.

## Ports

### Input

#### Port\_1 — Input signal

scalar | vector

Input signal to the block to calculate the square root, signed square root, or reciprocal of square root. The `sqrt` function accepts real or complex inputs, except for complex fixed-point signals. `signedSqrt` and `rSqrt` do not accept complex inputs.

Data Types: `single`

### Output

#### Port\_1 — Output signal

scalar | vector

Output signal that is the signed square root of the input signal.

Data Types: `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`






## Parameters

### Main

#### Function — Function the block performs

sqrt (default) | signedSqrt | rSqrt

Specify the mathematical function that the block calculates. The block icon changes to match the function you select.

Function	Block Icon
sqrt	
signedSqrt	
rSqrt	

#### Programmatic Use

**Block Parameter:** Operator

**Type:** character vector

**Values:** 'sqrt' | 'signedSqrt' | 'rSqrt'

**Default:** 'sqrt'

#### Output signal type — Output signal type

auto (default) | real | complex

Specify the output signal type of the block.

Function	Input Signal Type	Output Signal Type		
		Auto	Real	Complex
sqrt	real	real for nonnegative inputs NaN for negative inputs	real for nonnegative inputs NaN for negative inputs	complex
	complex	complex	error	complex
signedSqrt	real	real	real	error
	complex	error	error	error
rSqrt	real	real	real	error
	complex	error	error	error

#### Programmatic Use

**Block Parameter:** OutputSignalType

**Type:** character vector


**Values:** 'auto' | 'real' | 'complex'

**Default:** 'auto'

## Algorithm

The **Algorithm** tab contains the **Method** and the **Number of iterations** fields. These fields are available only if you select the `rsqrt` option in the **Function** field of the **Main** tab. For more information, see [Reciprocal Sqrt](#).

## Data Types

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant” (Simulink).

#### Intermediate results data type — Data type of intermediate results

Inherit:Inherit via internal rule (default)

#### Programmatic Use

**Block Parameter:** IntermediateResultsDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via internal rule'

**Default:** 'Inherit: Inherit via internal rule'

### Output — Output data type

Inherit: Same as first input (default) | Inherit: Inherit via internal rule | Inherit: Inherit via back propagation | double | single | int8 | int32 | uint32 | fixdt(1,16,2^0,0) | <data type expression> | ...

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

#### Programmatic Use

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'Inherit: Same as first input' | 'double' | 'single', 'int8', 'uint8', int16, 'uint16', 'int32', 'uint32', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0). '<data type expression>'

**Default:** 'Inherit: Same as first input'

### Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

off (default) | on

Select this option to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

#### Programmatic Use

**Block Parameter:** LockScale

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

### Saturate on integer overflow — Choose the behavior when integer overflow occurs

on (default) | boolean

Action	Rationale	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Clear this check box.	<p>You want to optimize efficiency of your generated code.</p> <p>You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors” (Simulink).</p>	Overflows wrap to the appropriate value that is representable by the data type.	The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code> , which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code> , is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use****Block Parameter:** SaturateOnIntegerOverflow**Type:** character vector**Value:** 'off' | 'on'**Default:** 'on'

## Block Characteristics

<b>Data Types</b>	single   base integer   fixed point
<b>Multidimensional Signals</b>	Yes
<b>Variable-Size Signals</b>	Yes

## See Also

Math Function | Reciprocal Sqrt | Sqrt

**Introduced in R2018b**



# Properties – Alphabetical List

---

## **ClockHighTime**

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

### **Settings**

ns

Default: 5

The clock high time is expressed as a positive integer.

The `ClockHighTime` and `ClockLowTime` properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

### **Usage Notes**

HDL Coder ignores this property if `ForceClock` is set to `off`.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### **See Also**

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



## ClockLowTime

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

### Settings

Default: 5

The clock low time is expressed as a positive integer.

The `ClockHighTime` and `ClockLowTime` properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

### Usage Notes

HDL Coder ignores this property if `ForceClock` is set to `off`.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

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

## **EDAScriptGeneration**

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

### **Settings**

'on' (default)

Enable generation of script files.

'off'

Disable generation of script files.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### **See Also**

“Generate Scripts for Compilation, Simulation, and Synthesis”

## ForceClock

Specify whether test bench forces clock input signals

### Settings

'on' (default)

**Selected** (default)

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

'off'

**Cleared**

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

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

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

## ForceClockEnable

Specify whether test bench forces clock enable input signals

### Settings

'on' (default)

**Selected** (default)

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

'off'

**Cleared**

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

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

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

# ForceReset

Specify whether test bench forces reset input signals

## Settings

'on' (default)

**Selected** (default)

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

'off'

**Cleared**

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

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

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

## FPToleranceStrategy

Specify whether to check for floating-point tolerance based on relative error or ULP error

### Settings

Use this setting to specify the tolerance strategy for checking the numerical accuracy in the generated test bench. Based on the tolerance strategy that you specify, you can enter a custom tolerance value.

'relative' (default)

When you verify the generated code, HDL Coder checks for the floating-point tolerance based on the relative error.

'ulp'

When you verify the generated code, HDL Coder checks for the floating-point tolerance based on the ULP error.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### Example

To specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');
```

```
% When using ULP error, optionally enter tolerance value greater than or equal to 0
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

## See Also

- FPToleranceValue
- “Floating point tolerance check based on”
- “Tolerance Value”
- “Getting Started with HDL Coder Native Floating-Point Support”

# FPToleranceValue

Enter the tolerance value based on floating-point tolerance check setting

## Settings

*N*

Default: 1e-07

The value of *N* depends on the floating-point tolerance check setting that you specify. Use this setting to specify a custom tolerance value for checking the numerical accuracy in the generated test bench. When you set the **Floating point tolerance check based on** to:

- `relative error`, the default is a tolerance value of 1e-07. When you use this floating-point tolerance check setting, specify the tolerance value as a double data type.
- `ulp error`, the default is a **Tolerance Value** of 0. When you use this floating-point tolerance check setting, specify the tolerance value as an integer. You can specify a **Tolerance Value**, *N*, that is greater than or equal to 0.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Example

To specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');
```



```
% When using ULP error, optionally enter tolerance value greater than or equal to 0  
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

## See Also

- “Tolerance Value”
- FPToleranceStrategy
- “Floating point tolerance check based on”

## GenerateCoSimBlock

Generate HDL Cosimulation blocks for use in testing DUT

### Settings

'on'

If your installation includes one or more of the following HDL simulation features, HDL Coder generates an HDL Cosimulation block for each:

- HDL Verifier for use with Mentor Graphics ModelSim
- HDL Verifier for use with Cadence Incisive

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

The coder appends the character vector that the `CosimLibPostfix` property specifies to the names of the generated HDL Cosimulation blocks.

'off' (default)

Do not generate HDL Cosimulation blocks.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# GenerateCoSimModel

Generate model containing HDL Cosimulation block for use in testing DUT

## Settings

'ModelSim' (default)

If your installation includes HDL Verifier, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for Mentor Graphics ModelSim.

'Incisive'

If your installation includes HDL Verifier, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for Cadence Incisive.

'None'

Do not create a cosimulation model.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

“Generate a Cosimulation Model”

## GenerateSVDPItestbench

Generate SystemVerilog DPI test bench

### Settings

'ModelSim' (default)

Generate SystemVerilog DPI test bench, and build-and-run scripts, for the Mentor Graphics ModelSim simulator.

'Incisive'

Generate SystemVerilog DPI test bench, and build-and-run scripts, for the Cadence Incisive simulator.

'VCS' (default)

Generate SystemVerilog DPI test bench, and build-and-run scripts, for the Synopsys® VCS® simulator.

'Vivado'

Generate SystemVerilog DPI test bench, and build-and-run scripts, for the Xilinx Vivado simulator.

When you set this property, the coder generates a direct programming interface (DPI) component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

Consider using this option if the default HDL test bench takes a long time to generate or simulate. Generation of a DPI test bench is sometimes faster than the default version because it does not run a full Simulink simulation to create the test bench data. Simulation of a DPI test bench with a large data set is faster than the default version because it does not store the input or expected data in a separate file.

To use this feature, you must have HDL Verifier and Simulink Coder licenses. To run the SystemVerilog testbench with generated VHDL code, you must have a mixed-language simulation license for your HDL simulator.

---

**Limitations** This test bench is not supported when you generate HDL code for the top-level Simulink model. Your DUT subsystem must meet the following conditions:

- Input and output data types of the DUT cannot be larger than 64 bits.
  - Input and output ports of the DUT cannot use enumerated data types.
  - Input and output ports cannot be single-precision or double-precision data types.
  - The DUT cannot have multiple clocks. You must set the **Clock inputs** code generation option to `Single`.
  - **Use trigger signal as clock** must not be selected.
  - If the DUT uses vector ports, you must use **Scalarize vector ports** to flatten the interface.
- 

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

“Generate a SystemVerilog DPI Test Bench” on page 2-116.

“”

## **HDLCodeCoverage**

Include HDL code coverage switches in generated test bench scripts

### **Settings**

'on'

Generated script includes code coverage switches. When you run the HDL simulation, code coverage is collected for your generated test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

'off' (default)

Generated script does not include code coverage switches, and does not collect code coverage.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### **See Also**

`SimulationTool`

# HDLCompileInit

Specify text written to initialization section of compilation script

## Settings

*'Initialization text'*

Default: `'vlib %s\n'`.

Specify text written to initialization section of compilation script as a character vector. If your `TargetLanguage` is VHDL, the implicit argument, `%s`, is the contents of the `VHDLLibraryName` property. If your `TargetLanguage` is Verilog, the implicit argument is `work`.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

### Topics

“Generate Scripts for Compilation, Simulation, and Synthesis”

## **HDLCompileTerm**

Specify text written to termination section of compilation script

### **Settings**

*'Termination text'*

Specify text written to termination section of compilation script as a character vector. The default is ''.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### **See Also**

“Generate Scripts for Compilation, Simulation, and Synthesis”



## HDLCompileFilePostfix

Specify postfix appended to file name for generated Mentor Graphics ModelSim compilation scripts

### Settings

*'Compilation file postfix'*

Default: `'_compile.do'`.

Specify the postfix as a character vector. HDL Coder appends the postfix to the file name for generated Mentor Graphics ModelSim compilation scripts.

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## HDLCompileVerilogCmd

Specify command written to compilation script for Verilog files

### Settings

*'Compilation command'*

Default: `'vlog %s %s\n'`.

Specify command written to compilation script for Verilog files as a character vector. The two arguments are the contents of the `SimulatorFlags` property and the file name of the current module. To omit the flags, set `SimulatorFlags` to `' '` (the default).

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLCompileVHDLCmd

Specify command written to compilation script for VHDL files

### Settings

*'Compilation command'*

Default: 'vcom %s %s\n'.

Specify command written to compilation script for VHDL files as a character vector. The two arguments are the contents of the `SimulatorFlags` property and the file name of the current entity. To omit the flags, set `SimulatorFlags` to '' (the default).

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLSimCmd

Specify command written to simulation script

### Settings

*'Simulation command'*

Default: `'vsim -novopt %s.%s\n'`.

Specify the command written to simulation script as a character vector. If your `TargetLanguage` is `'VHDL'`, the first implicit argument is the value of `VHDLLibraryName`. If your `TargetLanguage` is `'Verilog'`, the first implicit argument is `'work'`.

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

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLsimInit

Specify text written to initialization section of simulation script

### Settings

*'Simulation initialization'*

Specify text written to initialization section of simulation script as a character vector. The default is

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

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLSimFilePostfix

Specify postfix appended to file name for generated Mentor Graphics ModelSim simulation scripts

### Settings

*'Simulation file postfix'*

Default: `_sim.do`.

Specify the postfix as a character vector. HDL Coder appends the postfix to the file name for generated Mentor Graphics ModelSim simulation scripts.

For example, if the name of your test bench file is `my_design`, HDL Coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## HDLsimTerm

Specify text written to termination section of simulation script

### Settings

*'Termination text'*

Specify text written to termination section of simulation script as a character vector. Default is 'run -all\n'.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLSimViewWaveCmd

Specify waveform viewing command written to simulation script

### Settings

*'Waveform view command'*

Default: 'add wave sim:%s\n'

Specify waveform viewing command as a character vector. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”



## HDLLintCmd

Specify command written to HDL lint script

### Settings

*'Script command'*

Default: ''

Specify the command written to the HDL lint Tcl script as a character vector. The command must contain %s, which is a placeholder for the HDL file name.

### Dependencies

If HDLLintCmd is set to the default value, '', and you set HDLLintCmd to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default command string in the Tcl script.

### Usage

If you set HDLLintTool to Custom, you must use %s as a placeholder for the HDL file name in the generated Tcl script. Specify HDLLintCmd using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

### Set or View This Property

To set this property, use hdlset\_param or makehdl. To view the property value, use hdlget\_param.

## **See Also**

HDLLintTool, HDLLintInit, HDLLintTerm, “Generate an HDL Lint Tool Script”

## HDLLintInit

Specify HDL lint script initialization name

### Settings

*'Initialization name'*

Default: ''

Specify the HDL lint script initialization name as a character vector.

### Dependencies

If `HDLLintInit` is set to the default value, '', and you set `HDLLintCmd` to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default initialization string in the Tcl script.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

`HDLLintTool`, `HDLLintCmd`, `HDLLintTerm`, “Generate an HDL Lint Tool Script”

## HDLLintTerm

Specify HDL lint script termination name

### Settings

*'Script termination name'*

Default: ''

Specify the HDL lint script termination name as a character vector.

### Dependencies

If HDLLintTerm is set to the default value, '', and you set HDLLintCmd to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default termination string in the Tcl script.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

HDLLintTool, HDLLintCmd, HDLLintInit, “Generate an HDL Lint Tool Script”

## HDLLintTool

Select HDL lint tool for which HDL Coder generates scripts

### Settings

*'Lint Tool'*

Default: *'None'*.

HDLLintTool enables or disables generation of scripts for third-party HDL lint tools. By default, HDL Coder does not generate a lint script.

To generate a script for one of the supported lint tools, set HDLLintTool to one of the following:

HDLLintTool Option	Lint Tool
'None'	None. Lint script generation is disabled.
'AscentLint'	Real Intent Ascent Lint
'Leda'	Synopsys Leda
'SpyGlass'	Atrenta SpyGlass
'Custom'	A custom lint tool.

### Dependencies

If you set HDLLintTool to one of the supported third-party tools, you can generate a Tcl script without setting HDLLintInit, HDLLintCmd, and HDLLintTerm to nondefault values. If the HDLLintInit, HDLLintCmd, and HDLLintTerm have default values, HDL Coder automatically writes tool-specific default initialization, command, and termination strings to the Tcl script.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

#### Properties

`HDLLintCmd` | `HDLLintInit` | `HDLLintTerm`

#### Topics

“Generate an HDL Lint Tool Script”

# HDLSynthCmd

Specify command written to synthesis script

## Settings

*'Synthesis command'*

Default: none.

Specify command written to synthesis script as a character vector. Your choice of synthesis tool (see HDLSynthTool) sets the synthesis command string. The default is a formatted text string passed to `fprintf` to write the command section of the synthesis script. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

HDLSynthTool, HDLSynthInit, HDLSynthTerm, HDLSynthFilePostfix, “Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLSynthFilePostfix

Specify postfix appended to file name for generated synthesis scripts

### Settings

*'file name postfix'*

Specify HDLSynthTool as a character vector.

Default: The value of HDLSynthFilePostfix normally defaults to a string that corresponds to the synthesis tool that HDLSynthTool specifies.

For example, if the value of HDLSynthTool is 'Synplify', HDLSynthFilePostfix defaults to '\_synplify.tcl'. Then, if the name of the device under test is my\_design, HDL Coder adds the postfix \_synplify.tcl to form the synthesis script file name my\_design\_synplify.tcl.

### Set or View This Property

To set this property, use hdlset\_param or makehdl. To view the property value, use hdlget\_param.

### See Also

HDLSynthTool, HDLSynthCmd, HDLSynthInit, HDLSynthTerm, “Generate Scripts for Compilation, Simulation, and Synthesis”



# HDLSynthInit

Specify text written to initialization section of synthesis script

## Settings

*'Initialization text'*

Default: none

Specify the text written to the synthesis script initialization as a character vector. Your choice of synthesis tool (see HDLSynthTool) sets the synthesis script initialization string. The default is a formatted text passed to `fprintf` to write the initialization section of the synthesis script. The default is a synthesis project creation command. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## See Also

HDLSynthTool, HDLSynthCmd, HDLSynthTerm, HDLSynthFilePostfix, “Generate Scripts for Compilation, Simulation, and Synthesis”

## HDLSynthTerm

Specify text written to termination section of synthesis script

### Settings

*'Termination text'*

Default: none

Specify the synthesis script termination text as a character vector. Your choice of synthesis tool (see HDLSynthTool) sets the synthesis termination string. The default is a formatted text passed to `fprintf` to write the termination and clean up section of the synthesis script. This section does not take arguments. The content of the string is specific to the selected synthesis tool.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

HDLSynthTool, HDLSynthCmd, HDLSynthInit, HDLSynthFilePostfix, “Generate Scripts for Compilation, Simulation, and Synthesis”

# HDLSynthTool

Select synthesis tool for which HDL Coder generates scripts

## Settings

'*Synthesis tool*'

Default: 'None'.

Specify the synthesis tool as a character vector. HDLSynthTool enables or disables generation of scripts for third-party synthesis tools. By default, HDL Coder does not generate a synthesis script. To generate a script for one of the supported synthesis tools, set HDLSynthTool to one of the following:

---

**Tip** The value of HDLSynthTool also sets the postfix (HDLSynthFilePostfix) that the coder appends to generated synthesis script file names.

---

Choice of HDLSynthTool Value...	Generates Script For...	Sets HDLSynthFilePostfix To...
'None'	N/A; script generation disabled	N/A
'ISE'	Xilinx ISE	'_ise.tcl'
'Libero'	Microsemi Libero	'_libero.tcl'
'Precision'	Mentor Graphics Precision	'_precision.tcl'
'Quartus'	Altera Quartus II	'_quartus.tcl'
'Synplify'	Synopsys Synplify Pro®	'_synplify.tcl'
'Vivado'	Xilinx Vivado	'_vivado.tcl'
'Custom'	A custom synthesis tool	'_custom.tcl'

## **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## **See Also**

`HDLSynthCmd`, `HDLSynthInit`, `HDLSynthTerm`, `HDLSynthFilePostfix`, “Generate Scripts for Compilation, Simulation, and Synthesis”

# HoldInputDataBetweenSamples

Specify how long subrate signal values are held in valid state

## Settings

'on' (default)

Data values for subrate signals are held in a valid state across N base-rate clock cycles, where N is the number of base-rate clock cycles that elapse per subrate sample period and  $N \geq 2$ .

'off'

Data values for subrate signals are held in a valid state for only one base-rate clock cycle. For the subsequent base-rate cycles, data is in an unknown state (expressed as 'X') until leading edge of the next subrate sample period.

## Usage Notes

In most cases, the default ('on') is the best setting for this property. This setting matches the behavior of a Simulink simulation, in which subrate signals are held valid through each base-rate clock period.

In some cases (for example modeling memory or memory interfaces), it is desirable to set `HoldInputDataBetweenSamples` to 'off'. In this way, you can obtain diagnostic information about when data is in an invalid ('X') state.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## **See Also**

HoldTime, “Code Generation from Multirate Models”

## HoldTime

Specify hold time for input signals and forced reset input signals

### Settings

ns

Default: 2

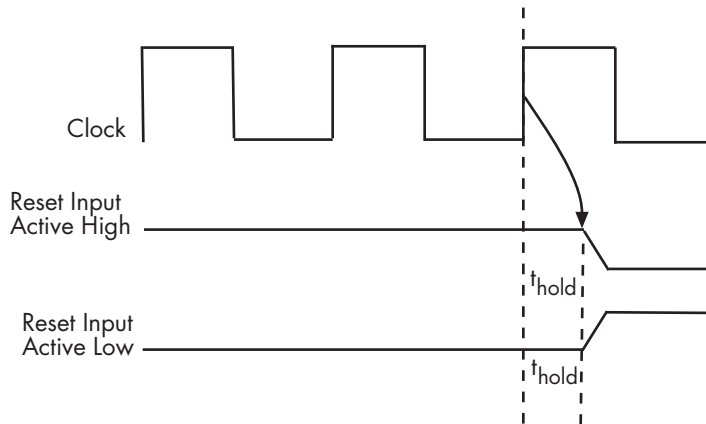
Specify the number of nanoseconds during which the model's data input signals and forced reset input signals are held past the clock rising edge.

The hold time is expressed as a positive integer.

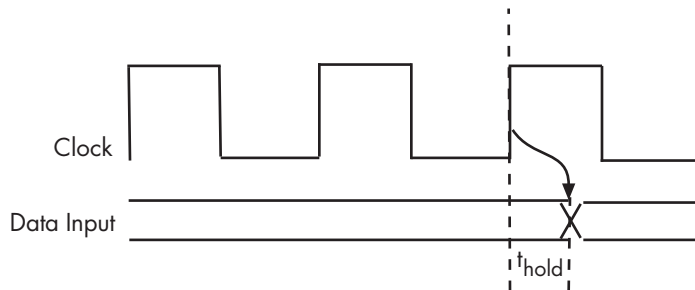
This option applies to reset input signals only if forced resets are enabled.

### Usage Notes

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



### Hold Time for Reset Input Signals



### Hold Time for Data Input Signals

---

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

---

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.



## **See Also**

ClockHighTime, ClockLowTime, ForceClock

## IgnoreDataChecking

Specify number of samples during which output data checking is suppressed

### Settings

$N$

Default: 0.

$N$  must be a positive integer.

When  $N > 0$ , the test bench suppresses output data checking for the first  $N$  output samples after the clock enable output (`ce_out`) is asserted.

### Usage Notes

When using pipelined block implementations, output data may be in an invalid state for some number of samples. To avoid spurious test bench errors, determine this number and set `IgnoreDataChecking` accordingly.

Be careful to specify  $N$  as a number of samples, not as a number of clock cycles. For a single-rate model, these are equivalent, but they are not equivalent for a multirate model.

You should use `IgnoreDataChecking` in cases where there is a state (register) initial condition in the HDL code that does not match the Simulink state, including the following specific cases:

- When you set the `DistributedPipelining` parameter to 'on' for the MATLAB Function block (see “Distributed Pipeline Insertion for MATLAB Function Blocks”).
- When you set the `ResetType` parameter to 'None' (see “ResetType”) for the following block types:
  - `commcnvintrlv2/Convolutional Deinterleaver`
  - `commcnvintrlv2/Convolutional Interleaver`
  - `commcnvintrlv2/General Multiplexed Deinterleaver`

- commcnvintrlv2/General Multiplexed Interleaver
  - dspsigops/Delay
  - simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled
  - simulink/Commonly Used Blocks/Unit Delay
  - simulink/Discrete/Delay
  - simulink/Discrete/Memory
  - simulink/Discrete/Tapped Delay
  - simulink/User-Defined Functions/MATLAB Function
  - sflib/Chart
  - sflib/Truth Table
- When generating a black box interface to existing manually-written HDL code.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## **InitializeTestBenchInputs**

Specify initial value driven on test bench inputs before data is asserted to DUT

### **Settings**

'on'

Initial value driven on test bench inputs is '0'.

'off' (default)

Initial value driven on test bench inputs is 'X' (unknown).

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

# MultifileTestBench

Divide generated test bench into helper functions, data, and HDL test bench code files

## Description

You can use this property to specify how you want to divide files that contain the test bench code, data, and helper functions.

The file names are derived from the name of the DUT, the **Test bench name postfix** property, and the **Test bench data file name postfix** property as:  
*DUTname\_TestBenchPostfix\_TestBenchDataPostfix*

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

- `symmetric_fir_tb.vhd`: test bench code
- `symmetric_fir_tb_pkg.vhd`: helper functions package
- `symmetric_fir_tb_data.vhd`: data package

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

- `symmetric_fir_tb.v`: test bench code
- `symmetric_fir_tb_pkg.v`: helper functions package
- `symmetric_fir_tb_data.v`: test bench data

## Settings

'on'

Write three separate HDL files. There is a separate file for test bench code, helper functions, and test bench data.

'off' (default)

Write two separate HDL files. One file contains the HDL test bench code. The other file contains the helper functions package and test bench data.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### **See Also**

`TestBenchPostFix`, `TestBenchDataPostFix`

# SimulationLibPath

Specify the path to the compiled Altera or Xilinx simulation libraries

## Settings

*'Simulation library path'*

Default: ''

Specify the path to the compiled Altera or Xilinx simulation libraries. Altera provides the simulation model files in `\quartus\eda\sim_lib` folder.

## Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## Usage Example

If you want to set the path to the compiled Xilinx Simulation library, enter:

```
myDUT = gcb;  
libpath = '/apps/Xilinx_ISE/XilinxISE-13.4/Linux/ISE_DS/ISE/vhdl/  
    mti_se/6.6a/lin64/xilinxcorelib';  
hdlset_param (myDUT, 'SimulationLibPath', libpath);
```

## See Also

### Topics

“Simulation library path”

## **SimulationTool**

Simulator for which the tool generates build-and-run scripts for the test bench and optional code coverage

### **Settings**

'Mentor Graphics ModelSim' | 'Cadence Incisive'|'Custom'

Default: 'Mentor Graphics ModelSim'

When you select 'Custom', the tool uses the custom script properties.

### **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.



## SimulatorFlags

Specify simulator flags to apply to generated compilation scripts

### Settings

*'compilation simulator flags'*

Default: ''

Specify simulator flags to apply to generated compilation scripts as a character vector. The simulator flags are specific to your application and the simulator you are using. For example, if you must use the 1076-1993 VHDL compiler, specify the flag `-93`.

### Usage Notes

The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command is specified by the `HDLCompileVHDL` or `HDLCompileVerilog` properties.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## SynthesisProjectAdditionalFiles

Include additional HDL or constraint files in synthesis project

### Settings

' ' (default)

Additional project files, such as HDL source files (.v, .vhd) or constraint files (.ucf), that you want to include in your synthesis project, specified as a character vector. Separate file names with a semicolon (;).

You cannot use `SynthesisProjectAdditionalFiles` to include Tcl files. To specify synthesis project Tcl files, use the `AdditionalProjectCreationTclFiles` property of the `hdlcoder.WorkflowConfig` object.

### Usage

To include a source file, `src_file.vhd`, and a constraint file, `constraint_file.ucf`, in the synthesis project for a DUT subsystem, `myDUT`:

```
hdlset_param (myDUT, 'SynthesisProjectAdditionalFiles', ...  
              'L:\src_file.vhd;L:\constraint_file.ucf;')
```

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

`hdlcoder.WorkflowConfig`

## TestBenchClockEnableDelay

Define elapsed time in clock cycles between deassertion of reset and assertion of clock enable

### Settings

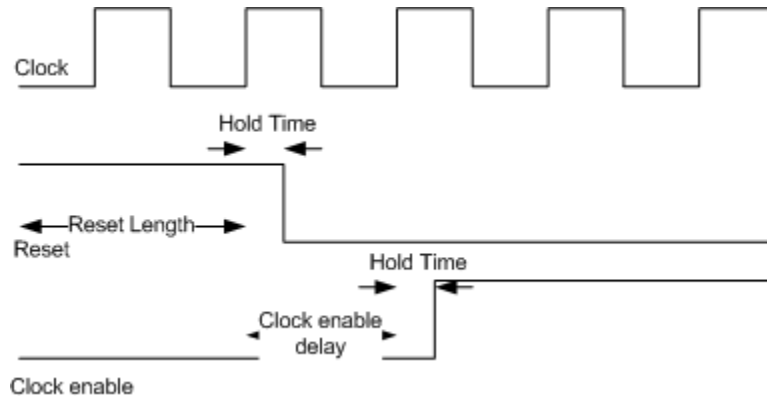
$N$  (integer number of clock cycles)

Default: 1

The `TestBenchClockEnableDelay` property specifies a delay time  $N$ , expressed in base-rate clock cycles ( the default value is 1) elapsed between the time the reset signal is deasserted and the time the clock enable signal is first asserted.

`TestBenchClockEnableDelay` works in conjunction with the `HoldTime` property; after deassertion of reset, the clock enable goes high after a delay of  $N$  base-rate clock cycles plus the delay specified by `HoldTime`.

In the figure below, the reset signal (active-high) deasserts after the interval labelled `Hold Time`. The clock enable asserts after a further interval labelled `Clock enable delay`.



## **Set or View This Property**

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## **See Also**

`HoldTime`,

## TestBenchDataPostFix

Specify suffix added to test bench data file name when generating multifile test bench

### Settings

*'Data postfix'*

Default: `'_data'`.

Specify the postfix as a character vector. HDL Coder applies `TestBenchDataPostFix` only when generating a multi-file test bench (i.e. when `MultifileTestBench` is `'on'`).

For example, if the name of your DUT is `my_test`, and `TestBenchPostFix` has the default value `_tb`, the coder adds the postfix `_data` to form the test bench data file name `my_test_tb_data`.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

`MultifileTestBench`, `TestBenchPostFix`

## TestBenchPostFix

Specify suffix to test bench name

### Settings

*'testbench suffix'*

Default: *'\_tb'*.

Specify the suffix to testbench name as a character vector.

For example, if the name of your DUT is `my_test`, HDL Coder adds the postfix `_tb` to form the name `my_test_tb`.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

### See Also

`MultifileTestBench`, `TestBenchDataPostFix`

## TestBenchReferencePostFix

Specify text appended to names of reference signals generated in test bench code

### Settings

*'Signal name postfix'*

Default: `'_ref'`.

Reference signal data is represented as arrays in the generated test bench code. HDL Coder appends the character vector that `TestBenchReferencePostFix` specifies to the generated signal names.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

## UseFileIOInTestBench

Specify whether to use data files for reading and writing test bench stimulus and reference data

### Settings

'on' (default)

**Selected** (default)

Create and use data files for reading and writing test bench stimulus and reference data.

'off'

**Cleared**

Generated test bench contains stimulus and reference data as constants.

### Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.



# **Class reference for HDL code generation from Simulink**

---

## hdlcoder.FloatingPointTargetConfig class

**Package:** hdlcoder

Specify floating-point target configuration for floating-point library

### Description

The `hdlcoder.FloatingPointTargetConfig` object sets options for HDL Coder to generate synthesizable floating-point code. To create an `hdlcoder.FloatingPointTargetConfig` object for a floating-point library, use the `hdlcoder.createFloatingPointTargetConfig` function. You can create a floating-point configuration object for these floating-point libraries:

- Native Floating Point
- Altera Megafunctions (ALTERA FP Functions)
- Altera Megafunctions (ALTFP)
- Xilinx LogiCORE®

### Construction

`fpconfig = hdlcoder.createFloatingPointConfig(library)` creates an `hdlcoder.FloatingPointTargetConfig` object for a floating-point library.

`fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)` creates an `hdlcoder.FloatingPointTargetConfig` object with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name on page 5-3 and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The name-value pair arguments that you can specify depend on the library that you select for creating the floating-point configuration.

## Input Arguments

### Library — Floating point library name

None (default) | NATIVEFLOATINGPOINT | ALTERAFPFUNCTIONS | ALTFP | XILINXLOGICORE

Floating-point library name, specified as a character vector

Example: 'ALTERAFPFUNCTIONS'

## Properties

### Native Floating Point

#### HandleDenormals — Specify whether to handle denormal numbers in your design

'off' (default) | 'on'

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a character vector. Denormal numbers are nonzero numbers that are smaller than the smallest normal number.

#### LatencyStrategy — Specify whether to use maximum or minimum latency for the native floating-point operator

'MAX' (default) | 'MIN' | 'ZERO'

Specify whether you want HDL Coder to use maximum or minimum latency setting for the floating-operators that your design uses. Specify this property as a character vector.

#### MantissaMultiplyStrategy — Specify how you want HDL Coder to implement the mantissa multiplication operation when your design uses floating-point multipliers

'FullMultiplier' (default) | 'PartMultiplierPartAddShift' | 'NoMultiplierFullAddShift'

Specify how you want HDL Coder to implement the mantissa multiplication process for floating-point multipliers in your design. With this option, you can control the DSP usage on the target platform for your design. To learn more, see “Mantissa Multiplier Strategy”.

### Altera FP Functions

#### **InitializeIPipelinesToZero** — Specify whether to initialize pipeline registers in the Altera Megafunction IP to zero

true (default) | false

Specify whether you want HDL Coder to initialize pipeline registers in the Altera Megafunction IP to zero. Specify this property as a `logical`. To avoid potential numerical mismatches in the HDL simulation, leave `InitializeIPipelinesToZero` set to `true`.

### ALTFP and Xilinx LogiCORE

#### **LatencyStrategy** — Specify whether to use maximum or minimum latency when mapping your design to FPGA floating-point target libraries

'MIN' (default) | 'MAX'

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

#### **Objective** — Specify whether to optimize the design for speed or area when mapping your design to FPGA floating-point target libraries

'SPEED' (default) | 'AREA'

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

## Methods

`createFloatingPointTargetConfig`

Create floating-point target configuration for floating-point library that you specify

## Examples

## Create Floating-Point Configuration with Native Floating Point and Generate Code

This example shows how to create a floating-point target configuration with the native floating-point support in HDL Coder, and then generate code.

### Create a Floating-Point Target Configuration

To create a floating-point configuration, use `hdlcoder.createFloatingPointTargetConfig`.

```
load_system('sfir_single');  
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT')
```

```
fpconfig =
```

```
  FloatingPointTargetConfig with properties:
```

```
      Library: 'NativeFloatingPoint'  
LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]  
      IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

### Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.HandleDenormals = 'off';  
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';  
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'NoMultiplierFullAddShift';  
fpconfig.LibrarySettings
```

```
ans =
```

```
  NFPLatencyDrivenMode with properties:
```

```
      LatencyStrategy: 'MIN'  
      HandleDenormals: 'off'  
MantissaMultiplyStrategy: 'NoMultiplierFullAddShift'  
      Version: '1.0.0'
```

## View Latency of Native Floating Point Operators

The IPConfig object displays the maximum and minimum latency values of the floating-point operators.

`fpconfig.IPConfig`

`ans =`

Name	DataType	MaxLatency	MinLatency
'ABS'	'SINGLE'	0	0
'ADDSUB'	'SINGLE'	12	7
'ATAN'	'SINGLE'	36	36
'ATAN2'	'SINGLE'	42	42
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6
'COS'	'SINGLE'	27	27
'DIV'	'SINGLE'	32	32
'EXP'	'SINGLE'	23	23
'FIX'	'SINGLE'	3	3
'LOG'	'SINGLE'	20	20
'MINMAX'	'SINGLE'	3	3
'MOD'	'SINGLE'	0	0
'MUL'	'SINGLE'	8	8
'POW2'	'SINGLE'	2	2
'RECIPI'	'SINGLE'	19	19
'RELOP'	'SINGLE'	3	3
'REM'	'SINGLE'	0	0
'ROUNDING'	'SINGLE'	5	5
'RSQRT'	'SINGLE'	17	17
'SIGNUM'	'SINGLE'	0	0
'SIN'	'SINGLE'	27	27
'SINCOS'	'SINGLE'	27	27
'SQRT'	'SINGLE'	28	28
'UMINUS'	'SINGLE'	0	0

## Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/NativeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences this latency.
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_comp as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_nfp_add_comp.vhdl
### Working on sfir_single/symmetric_fir/nfp_mul_comp as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_nfp_mul_comp.vhdl
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir.vhdl
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhdl
### Creating HDL Code Generation Check Report file://C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_codegen_report.html
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated VHDL code is saved in the `hdlsrc` folder.

## See Also

`hdlcoder.FloatingPointTargetConfig.IPConfig` |  
`hdlcoder.FloatingPointTargetConfig.IPConfig.customize`

## Topics

""  
""

“Share Floating-Point IPs”

“Generate HDL Code for FPGA Floating-Point Target Libraries”

“Customize Floating-Point IP Configuration”

“Generate Target-Independent HDL Code with Native Floating-Point”

## Introduced in R2016b

## createFloatingPointTargetConfig

**Class:** hdlcoder.FloatingPointTargetConfig

**Package:** hdlcoder

Create floating-point target configuration for floating-point library that you specify

### Syntax

```
fpconfig = hdlcoder.createFloatingPointConfig(library)
fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)
```

### Description

To create a floating-point target configuration object for a floating-point library, use the `hdlcoder.createFloatingPointTargetConfig` function. You can create a floating-point configuration object for these libraries:

- Native Floating Point
- Altera Megafunctions (ALTERA FP Functions)
- Altera Megafunctions (ALTFP)
- Xilinx LogiCORE

`fpconfig = hdlcoder.createFloatingPointConfig(library)` creates an `hdlcoder.FloatingPointTargetConfig` object for a given floating-point library.

`fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)` creates an `hdlcoder.FloatingPointTargetConfig` object with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.



## Input Arguments

### **Library — Floating point library name**

None (default) | NATIVEFLOATINGPOINT | ALTERAFPFUNCTIONS | ALTFP | XILINXLOGICORE

Floating-point library name, specified as a character vector

Example: 'ALTERAFPFUNCTIONS'

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

The name-value pair arguments that you can specify depend on the library that you select for creating the floating-point configuration.

### **Native Floating Point**

#### **HandleDenormals — Specify whether to handle denormal numbers in your design**

'off' (default) | 'on'

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a character vector. Denormal numbers are nonzero numbers that are smaller than the smallest normal number. To specify this property, for `Library`, select `NATIVEFLOATINGPOINT`.

#### **LatencyStrategy — Specify whether to use maximum or minimum latency for the native floating-point operator**

'MAX' (default) | 'MIN' | 'ZERO'

Specify whether you want HDL Coder to use maximum or minimum latency setting for the floating-operators that your design uses. Specify this property as a character vector. To specify this property, for `Library`, select `NATIVEFLOATINGPOINT`

### **MantissaMultiplyStrategy — Specify how you want HDL Coder to implement the mantissa multiplication operation when your design uses floating-point multipliers**

'FullMultiplier' (default) | 'PartMultiplierPartAddShift' |  
'NoMultiplierFullAddShift'

Specify how you want HDL Coder to implement the mantissa multiplication process for floating-point multipliers in your design. With this option, you can control the DSP usage on the target platform for your design. To learn more, see “Mantissa Multiplier Strategy”.

### **Altera FP Functions**

### **InitializeIPPipelinesToZero — Specify whether to initialize pipeline registers in the Altera Megafunction IP to zero**

true (default) | false

Specify whether you want HDL Coder to initialize pipeline registers in the Altera Megafunction IP to zero. Specify this property as a logical. To avoid potential numerical mismatches in the HDL simulation, leave `InitializeIPPipelinesToZero` set to true. To specify this property, for Library, select ALTERAFPFUNCTIONS.

### **ALTFP and Xilinx LogiCORE**

### **LatencyStrategy — Specify whether to use maximum or minimum latency when mapping your design to FPGA floating-point target libraries**

'MIN' (default) | 'MAX'

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

### **Objective — Specify whether to optimize the design for speed or area when mapping your design to FPGA floating-point target libraries**

'SPEED' (default) | 'AREA'

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

## **Examples**

## Create Floating-Point Configuration with Native Floating Point and Generate Code

This example shows how to create a floating-point target configuration with the native floating-point support in HDL Coder, and then generate code.

### Create a Floating-Point Target Configuration

To create a floating-point configuration, use `hdlcoder.createFloatingPointTargetConfig`.

```
load_system('sfir_single');
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT')
```

```
fpconfig =
```

```
  FloatingPointTargetConfig with properties:
```

```
      Library: 'NativeFloatingPoint'
LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]
      IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

### Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.HandleDenormals = 'off';
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'NoMultiplierFullAddShift';
fpconfig.LibrarySettings
```

```
ans =
```

```
  NFPLatencyDrivenMode with properties:
```

```
      LatencyStrategy: 'MIN'
      HandleDenormals: 'off'
MantissaMultiplyStrategy: 'NoMultiplierFullAddShift'
      Version: '1.0.0'
```

## View Latency of Native Floating Point Operators

The IPConfig object displays the maximum and minimum latency values of the floating-point operators.

fpconfig.IPConfig

ans =

Name	DataType	MaxLatency	MinLatency
'ABS'	'SINGLE'	0	0
'ADDSUB'	'SINGLE'	12	7
'ATAN'	'SINGLE'	36	36
'ATAN2'	'SINGLE'	42	42
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6
'COS'	'SINGLE'	27	27
'DIV'	'SINGLE'	32	32
'EXP'	'SINGLE'	23	23
'FIX'	'SINGLE'	3	3
'LOG'	'SINGLE'	20	20
'MINMAX'	'SINGLE'	3	3
'MOD'	'SINGLE'	0	0
'MUL'	'SINGLE'	8	8
'POW2'	'SINGLE'	2	2
'RECIPI'	'SINGLE'	19	19
'RELOP'	'SINGLE'	3	3
'REM'	'SINGLE'	0	0
'ROUNDING'	'SINGLE'	5	5
'RSQRT'	'SINGLE'	17	17
'SIGNUM'	'SINGLE'	0	0
'SIN'	'SINGLE'	27	27
'SINCOS'	'SINGLE'	27	27
'SQRT'	'SINGLE'	28	28
'UMINUS'	'SINGLE'	0	0

## Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/NativeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additional delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences this latency.
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_comp as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir\nfp_add_comp.vhdl
### Working on sfir_single/symmetric_fir/nfp_mul_comp as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir\nfp_mul_comp.vhdl
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir.vhdl
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhdl
### Creating HDL Code Generation Check Report file://C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_codegen_report.html
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated VHDL code is saved in the `hdlsrc` folder.

## See Also

`hdlcoder.FloatingPointTargetConfig.IPConfig` |  
`hdlcoder.FloatingPointTargetConfig.IPConfig.customize`

## Topics

""  
""

“Share Floating-Point IPs”  
“Generate HDL Code for FPGA Floating-Point Target Libraries”  
“Customize Floating-Point IP Configuration”  
“Generate Target-Independent HDL Code with Native Floating-Point”

## hdlcoder.FloatingPointTargetConfig.IPConfig class

**Package:** hdlcoder

Specify IP settings for selected floating-point configuration

### Description

Use the `hdlcoder.FloatingPointTargetConfig.IPConfig` object to see the list of supported IP blocks for a floating-point library. The IP configuration depends on the library settings. The library settings are specific to the floating-point library that you choose.

- 1 Create a floating-point target configuration object for the library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

- 2 To see the IP settings, use the `IPConfig` object.

```
fpconfig.IPConfig
```

Optionally, to customize the IP configuration, use the `customize` method of the floating-point configuration object.

### Construction

`fpconfig.IPConfig` shows the IP settings for the `fpconfig` floating-point target configuration that you create for the floating-point library.

### Methods

`customize` Customize IP configuration for specified floating-point library

## Examples

### Create and Customize Floating Point Configuration and Generate Code

This example shows how to create a floating-point target configuration with Altera® Megafunctions (ALTFP) in HDL Coder, and then generate code.

#### Create a Floating-Point Target Configuration

To create a floating-point configuration, use `hdlcoder.createFloatingPointTargetConfig`. Before creating a configuration, set up the path to your synthesis tool.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', ...
    'ToolPath', 'C:/Altera/16.0/quartus/bin64/quartus.exe');
load_system('sfir_single')
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP')
```

Prepending following Altera Quartus II path(s) to the system path:  
C:\Altera\16.0\quartus\bin64

```
fpconfig =
```

```
    FloatingPointTargetConfig with properties:
```

```
        Library: 'ALTFP'
    LibrarySettings: [1x1 fpconfig.LatencyDrivenMode]
        IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

#### Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
fpconfig.LibrarySettings.Objective = 'AREA';
fpconfig.LibrarySettings
```

```
ans =
```

```
    LatencyDrivenMode with properties:
```

```
        LatencyStrategy: 'MAX'
```

Objective: 'AREA'

### View Latency of Floating-Point IPs

The IPConfig object displays the maximum and minimum latency values of the floating-point operators.

fpconfig.IPConfig

ans =

Name	DataType	MinLatency	MaxLatency	Latency	Ext
'ABS'	'DOUBLE'	1	1	-1	''
'ABS'	'SINGLE'	1	1	-1	''
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	-1	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6	-1	''
'COS'	'SINGLE'	35	35	-1	''
'DIV'	'DOUBLE'	10	61	-1	''
'DIV'	'SINGLE'	6	33	-1	''
'EXP'	'DOUBLE'	25	25	-1	''
'EXP'	'SINGLE'	17	17	-1	''
'LOG'	'DOUBLE'	34	34	-1	''
'LOG'	'SINGLE'	21	21	-1	''
'MUL'	'DOUBLE'	11	11	-1	''
'MUL'	'SINGLE'	11	11	-1	''
'RECIP'	'DOUBLE'	27	27	-1	''
'RECIP'	'SINGLE'	20	20	-1	''
'RELOP'	'DOUBLE'	1	3	-1	''
'RELOP'	'SINGLE'	1	3	-1	''
'RSQRT'	'DOUBLE'	36	36	-1	''
'RSQRT'	'SINGLE'	26	26	-1	''
'SIN'	'SINGLE'	36	36	-1	''
'SQRT'	'DOUBLE'	30	57	-1	''
'SQRT'	'SINGLE'	16	28	-1	''



## Customize Latency of ADDSUB IP

Using the customize method of the IPConfig object, you can customize the latency of the floating-point IP and specify any additional arguments.

```
fpconfig.IPConfig.customize('ADDSUB', 'Single', 'Latency', 6);
fpconfig.IPConfig
```

ans =

Name	DataType	MinLatency	MaxLatency	Latency	Ext
'ABS'	'DOUBLE'	1	1	-1	''
'ABS'	'SINGLE'	1	1	-1	''
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	6	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6	-1	''
'COS'	'SINGLE'	35	35	-1	''
'DIV'	'DOUBLE'	10	61	-1	''
'DIV'	'SINGLE'	6	33	-1	''
'EXP'	'DOUBLE'	25	25	-1	''
'EXP'	'SINGLE'	17	17	-1	''
'LOG'	'DOUBLE'	34	34	-1	''
'LOG'	'SINGLE'	21	21	-1	''
'MUL'	'DOUBLE'	11	11	-1	''
'MUL'	'SINGLE'	11	11	-1	''
'RECIP'	'DOUBLE'	27	27	-1	''
'RECIP'	'SINGLE'	20	20	-1	''
'RELOP'	'DOUBLE'	1	3	-1	''
'RELOP'	'SINGLE'	1	3	-1	''
'RSQRT'	'DOUBLE'	36	36	-1	''
'RSQRT'	'SINGLE'	26	26	-1	''
'SIN'	'SINGLE'	36	36	-1	''
'SQRT'	'DOUBLE'	30	57	-1	''
'SQRT'	'SINGLE'	16	28	-1	''

### Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/FloatingPoint/hdlsrc','SynthesisToolChipFamily','Arria10')

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### Using C:\Altera\16.0\quartus\bin64\qmegawiz for the selected floating point IP library.
### The code generation and optimization options you have chosen have introduced additional delays.
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences the following delays:
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Generating Altera(R) megafunction: altfp_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_add.vhd)
### Done.
### Generating Altera(R) megafunction: altfp_mul_single for latency of 11.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_mul.vhd)
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir.vhd
### Generating package file C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\FloatingPoint\hdlsrc\sfir_single\sfir_single_report.html
### HDL check for 'sfir_single' complete with 0 errors, 7 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

The generated VHDL code is saved in the `hdlsrc` folder.

### See Also

`hdlcoder.FloatingPointTargetConfiguration`

### Topics

""  
""

“Share Floating-Point IPs”

“Generate HDL Code for FPGA Floating-Point Target Libraries”

“Customize Floating-Point IP Configuration”

“Generate Target-Independent HDL Code with Native Floating-Point”

**Introduced in R2016b**

## customize

**Class:** hdlcoder.FloatingPointTargetConfig.IPConfig

**Package:** hdlcoder

Customize IP configuration for specified floating-point library

## Syntax

```
fpconfig.IPConfig.customize(Name,DataType,Name,Value)
```

## Description

`fpconfig.IPConfig.customize(Name,DataType,Name,Value)` customizes the `fpconfig` floating-point configuration with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **Name — Name of the floating-point IP**

' ' (default) | character vector

Name of the floating-point IP to customize, specified as a character vector.

Example: 'ADDSUB'

### **DataType — Data type of the floating-point IP**

' ' (default) | character vector

Data type of the floating-point IP to customize, specified as a character vector.

Example: 'SINGLE'

## 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 quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### Latency — Latency of the floating-point IP

-1 (default) | positive integer

Specify a custom latency value for the floating-point IP as an integer.

Example: `fpconfig.IPConfig.customize('ADDSUB', 'Double', 'Latency', 6)` specifies a custom latency value of 6 for the ADDSUB IP.

### ExtraArgs — Specify any additional arguments of the floating-point IP

' ' (default) | character vector

Specify any additional arguments of the floating-point IP as a character vector.

Example:

`fpconfig.IPConfig.customize('ADDSUB', 'Double', 'Latency', 6, 'ExtraArgs', 'CSET_c_mult_usage=Full_Usage')` specifies that you want to use DSP blocks on the target device.

## Examples

### Create and Customize Floating Point Configuration and Generate Code

This example shows how to create a floating-point target configuration with Altera® Megafunctions (ALTFP) in HDL Coder, and then generate code.

#### Create a Floating-Point Target Configuration

To create a floating-point configuration, use `hdlcoder.createFloatingPointTargetConfig`. Before creating a configuration, set up the path to your synthesis tool.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', ...
    'ToolPath', 'C:/Altera/16.0/quartus/bin64/quartus.exe');
load_system('sfir_single')
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP')
```

Prepending following Altera Quartus II path(s) to the system path:  
C:\Altera\16.0\quartus\bin64

```

fpconfig =

FloatingPointTargetConfig with properties:

    Library: 'ALTFP'
LibrarySettings: [1x1 fpconfig.LatencyDrivenMode]
    IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
    
```

### Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```

fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
fpconfig.LibrarySettings.Objective = 'AREA';
fpconfig.LibrarySettings
    
```

```

ans =

LatencyDrivenMode with properties:

    LatencyStrategy: 'MAX'
    Objective: 'AREA'
    
```

### View Latency of Floating-Point IPs

The IPConfig object displays the maximum and minimum latency values of the floating-point operators.

```
fpconfig.IPConfig
```

```
ans =
```

Name	DataType	MinLatency	MaxLatency	Latency	Ext
'ABS'	'DOUBLE'	1	1	-1	''
'ABS'	'SINGLE'	1	1	-1	''
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	-1	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''

'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6	-1	''
'COS'	'SINGLE'	35	35	-1	''
'DIV'	'DOUBLE'	10	61	-1	''
'DIV'	'SINGLE'	6	33	-1	''
'EXP'	'DOUBLE'	25	25	-1	''
'EXP'	'SINGLE'	17	17	-1	''
'LOG'	'DOUBLE'	34	34	-1	''
'LOG'	'SINGLE'	21	21	-1	''
'MUL'	'DOUBLE'	11	11	-1	''
'MUL'	'SINGLE'	11	11	-1	''
'RECIP'	'DOUBLE'	27	27	-1	''
'RECIP'	'SINGLE'	20	20	-1	''
'RELOP'	'DOUBLE'	1	3	-1	''
'RELOP'	'SINGLE'	1	3	-1	''
'RSQRT'	'DOUBLE'	36	36	-1	''
'RSQRT'	'SINGLE'	26	26	-1	''
'SIN'	'SINGLE'	36	36	-1	''
'SQRT'	'DOUBLE'	30	57	-1	''
'SQRT'	'SINGLE'	16	28	-1	''

### Customize Latency of ADDSUB IP

Using the customize method of the IPConfig object, you can customize the latency of the floating-point IP and specify any additional arguments.

```
fpconfig.IPConfig.customize('ADDSUB', 'Single', 'Latency', 6);
fpconfig.IPConfig
```

ans =

Name	DataType	MinLatency	MaxLatency	Latency	Ext
'ABS'	'DOUBLE'	1	1	-1	''
'ABS'	'SINGLE'	1	1	-1	''
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	6	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6	-1	''
'COS'	'SINGLE'	35	35	-1	''

'DIV'	'DOUBLE'	10	61	-1	''
'DIV'	'SINGLE'	6	33	-1	''
'EXP'	'DOUBLE'	25	25	-1	''
'EXP'	'SINGLE'	17	17	-1	''
'LOG'	'DOUBLE'	34	34	-1	''
'LOG'	'SINGLE'	21	21	-1	''
'MUL'	'DOUBLE'	11	11	-1	''
'MUL'	'SINGLE'	11	11	-1	''
'RECIP'	'DOUBLE'	27	27	-1	''
'RECIP'	'SINGLE'	20	20	-1	''
'RELOP'	'DOUBLE'	1	3	-1	''
'RELOP'	'SINGLE'	1	3	-1	''
'RSQRT'	'DOUBLE'	36	36	-1	''
'RSQRT'	'SINGLE'	26	26	-1	''
'SIN'	'SINGLE'	36	36	-1	''
'SQRT'	'DOUBLE'	30	57	-1	''
'SQRT'	'SINGLE'	16	28	-1	''

### Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/FloatingPoint/hdlsrc','SynthesisToolChipFamily','Arria10

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### Using C:\Altera\16.0\quartus\bin64\qmegawiz for the selected floating point IP lib
### The code generation and optimization options you have chosen have introduced addit
### The delay balancing feature has automatically inserted matching delays for compens
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Generating Altera(R) megafunction: altfp_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\s
### Done.
### Generating Altera(R) megafunction: altfp_mul_single for latency of 11.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\s
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\FloatingPoint\hdlsrc\sfir_single\symmetr
### Generating package file C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\FloatingPoint\hdlsrc\sfir_sing
### HDL check for 'sfir_single' complete with 0 errors, 7 warnings, and 0 messages.
### HDL code generation complete.
```



The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

The generated VHDL code is saved in the `hdlsrc` folder.

## Tips

Before using this function, create a floating-point target configuration object for the floating-point library that you specify. Select library as **Altera Megafunctions (ALTERA FP FUNCTIONS)**, **Altera Megafunctions (ALTFP)**, or **Xilinx LogiCORE**.

This example creates a floating-point target configuration for the **Altera Megafunctions (ALTFP)** library.

```
fpcfg = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

## See Also

`hdlcoder.FloatingPointTargetConfig`

## Topics

“”

“Share Floating-Point IPs”

“Generate HDL Code for FPGA Floating-Point Target Libraries”

“Customize Floating-Point IP Configuration”

**Introduced in R2016b**

## hdlcoder.WorkflowConfig class

**Package:** hdlcoder

Configure HDL code generation and deployment workflows

### Description

Use the `hdlcoder.WorkflowConfig` object to set HDL workflow options for the `hdlcoder.runWorkflow` function. You can customize the `hdlcoder.WorkflowConfig` object for these workflows:

- Generic ASIC/FPGA
- FPGA-in-the-Loop (requires HDL Verifier)
- FPGA Turnkey
- IP Core Generation
- Simulink Real-Time FPGA I/O (requires Simulink Real-Time™)

A best practice is to use the HDL Workflow Advisor to configure the workflow, and then export a workflow script. The commands in the workflow script create and configure the `hdlcoder.WorkflowConfig` object. See “Run HDL Workflow with a Script”.

### Construction

`hdlcoder.WorkflowConfig(Name, Value)` creates a workflow configuration object for you to specify your HDL code generation and deployment workflows, with additional options specified by one or more `Name, Value` pair arguments.

### 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

**SynthesisTool — Synthesis tool name**

'Xilinx Vivado' (default) | 'Altera QUARTUS II' | 'Xilinx ISE'

Name of the synthesis tool, specified as a character vector.

Example: 'SynthesisTool', 'Altera QUARTUS II' creates a workflow configuration object with 'Altera QUARTUS II' as the synthesis tool and 'Generic ASIC/FPGA' as the target workflow.

**TargetWorkflow — Specify the target workflow**

'Generic ASIC/FPGA' (default) | 'FPGA Turnkey' | 'IP Core Generation' | 'FPGA-in-the-Loop' | 'Simulink Real-Time FPGA I/O'

Target workflow for HDL code generation, specified as a character vector.

Example: 'TargetWorkflow', 'IP Core Generation' creates a workflow configuration object with 'Xilinx Vivado' as the synthesis tool and 'IP Core Generation' as the target workflow.

## Properties

**Generic ASIC/FPGA Workflow****ProjectFolder — Folder for generated project files**

' ' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project\_file\_folder'

**Objective — Synthesis tool objective**

hdlcoder.Objective.None (default) | hdlcoder.Objective.SpeedOptimized | hdlcoder.Objective.AreaOptimized | hdlcoder.Objective.CompileOptimized

High-level synthesis tool objective, specified as one of these values.

hdlcoder.Objective.None (default)	Do not generate additional Tcl commands.
hdlcoder.Objective.SpeedOptimized	Generate synthesis tool Tcl commands to optimize for speed.

<code>hdlcoder.Objective.AreaOptimized</code>	Generate synthesis tool Tcl commands to optimize for area.
<code>hdlcoder.Objective.CompileOptimized</code>	Generate synthesis tool Tcl commands to optimize for compilation time.

If your synthesis tool is Xilinx ISE and your target workflow is Generic ASIC/FPGA or FPGA Turnkey, set the `Objective` to `hdlcoder.Objective.None`.

For the tool-specific Tcl commands that are added to the synthesis project creation Tcl script, see “Synthesis Objective to Tcl Command Mapping”.

**RunTaskGenerateRTLCodeAndTestbench — Enable task to generate code and test bench**

`true` (default) | `false`

Enable or disable workflow task to generate code and test bench, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**RunTaskVerifyWithHDLCosimulation — Enable task to verify generated code with HDL cosimulation**

`true` (default) | `false`

Enable or disable task to verify the generated code with HDL cosimulation, specified as a `logical`. This option takes effect only when `GenerateCosimulationModel` is `true`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Verify with HDL Cosimulation** task.

**RunTaskCreateProject — Enable task to create synthesis tool project**

`true` (default) | `false`

Enable or disable task to create a synthesis tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

**RunTaskPerformLogicSynthesis — Enable task to launch synthesis tool and run logic synthesis**

`true` (default) | `false`

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

**RunTaskPerformMapping — Enable task to map synthesized logic to target device**

`true (default) | false`

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

**RunTaskPerformPlaceAndRoute — Enable task to run place and route process**

`true (default) | false`

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

**RunTaskRunSynthesis — Enable task to launch Xilinx Vivado and run synthesis**

`true (default) | false`

Enable or disable task to launch Xilinx Vivado and run synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Synthesis** task.

**RunTaskRunImplementation — Enable task to launch Xilinx Vivado and run implementation**

`true (default) | false`

Enable or disable task to launch Xilinx Vivado and run the implementation step, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Implementation** task.

**RunTaskAnnotateModelWithSynthesisResult — Enable task to analyze timing information and highlight critical paths**

true (default) | false

Enable or disable task to analyze pre- or post-routing timing information and highlight critical paths in your model, specified as a `logical`. This task is available only when the target workflow is `Generic ASIC/FPGA`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

**GenerateRTLCode — Generate HDL code**

true (default) | false

Option to generate HDL code in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**GenerateTestbench — Generate HDL test bench**

false (default) | true

Option to generate an HDL test bench in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**GenerateValidationModel — Generate validation model**

false (default) | true

Generate a validation model, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**AdditionalProjectCreationTclFiles — Additional project creation Tcl files to include in your synthesis project**

'' (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: 'L:\file1.tcl;L:\file2.tcl;'

### **SkipPreRouteTimingAnalysis — Skip pre-route timing analysis logical**

false (default) | true

Skip pre-route timing analysis, specified as a logical. If your tool does not support early timing estimation, set to true.

When you enable this option, `CriticalPathSource` is set to 'post-route'

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

### **IgnorePlaceAndRouteErrors — Ignore place and route errors**

false (default) | true

Ignore place and route errors, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

### **CriticalPathSource — Critical path source**

'pre-route' (default) | 'post-route'

Critical path source, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

### **CriticalPathNumber — Number of critical paths to annotate**

1 (default) | 2 | 3

Number of critical paths to annotate, specified as a positive integer from 1 to 3.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

### **ShowAllPaths — Show all critical paths**

false (default) | true

Show all critical paths, including duplicate paths, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

### **ShowDelayData — Annotate cumulative timing delay on each critical path**

true (default) | false

Annotate the cumulative timing delay on each critical path, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

### **ShowUniquePaths — Show only the first instance of a critical path**

false (default) | true

Show only the first instance of a critical path that is duplicated, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

### **ShowEndsOnly — Show only endpoints of each critical path**

false (default) | true

Show the endpoints of each critical path, omitting connecting signal lines, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

### **FPGA-in-the-Loop**

#### **ProjectFolder — Folder for generated project files**

' ' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project\_file\_folder'



**RunTaskGenerateRTLCodeAndTestbench — Enable task to generate code and test bench**`true (default) | false`

Enable or disable workflow task to generate code and test bench, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**RunTaskVerifyWithHDLCosimulation — Enable task to verify generated code with HDL cosimulation**`true (default) | false`

Enable or disable task to verify the generated code with HDL cosimulation, specified as a `logical`. This option takes effect only when `GenerateCosimulationModel` is `true`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Verify with HDL Cosimulation** task.

**RunTaskBuildFPGAInTheLoop — Enable task to generate a model that contains a FIL block and a testbench around the FIL block**`true (default) | false`

Enable or disable task to generate a model that contains a FIL block and a testbench around the FIL block specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Build FPGA-in-the-Loop** task.

**GenerateRTLCode — Generate HDL code**`true (default) | false`

Option to generate HDL code in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

**GenerateTestbench — Generate HDL test bench**`false (default) | true`

Option to generate an HDL test bench in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

### **GenerateValidationModel** — Generate validation model

false (default) | true

Generate a validation model, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

### **IPAddress** — IP address of FPGA board

'192.168.0.2' (default) | character vector

IP address of the FPGA board, specified as a character vector. Default address is '192.168.0.2'.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

### **MACAddress** — MAC address of FPGA board

'00-0A-35-02-21-8A' (default) | character vector

MAC address of the FPGA board, specified as a character vector, for example '00-0A-35-02-21-8A'. In most cases, you do not have to change the Board MAC address. If you want to connect more than one FPGA board to a single computer, specify a unique MAC address for each additional board.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

### **SourceFiles** — Additional HDL source files for verification

' ' (default) | character vector

Additional source files for the HDL design that you want to verify on the FPGA board, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

### **Connection** — JTAG or Ethernet connection

'JTAG' (default) | 'Ethernet'

Ethernet or JTAG connection type to the FPGA development board, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

### **RunExternalBuild — Run build process externally**

true (default) | false

Option to run build process in parallel with MATLAB, specified as a logical. If this option is disabled, you cannot use MATLAB until the build is finished.

### **FPGA Turnkey Workflow**

#### **ProjectFolder — Folder for generated project files**

' ' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project\_file\_folder'

#### **Objective — Synthesis tool objective**

hdlcoder.Objective.None (default) | hdlcoder.Objective.SpeedOptimized | hdlcoder.Objective.AreaOptimized | hdlcoder.Objective.CompileOptimized

High-level synthesis tool objective, specified as one of these values.

hdlcoder.Objective.None (default)	Do not generate additional Tcl commands.
hdlcoder.Objective.SpeedOptimized	Generate synthesis tool Tcl commands to optimize for speed.
hdlcoder.Objective.AreaOptimized	Generate synthesis tool Tcl commands to optimize for area.
hdlcoder.Objective.CompileOptimized	Generate synthesis tool Tcl commands to optimize for compilation time.

If your synthesis tool is Xilinx ISE and your target workflow is Generic ASIC/FPGA or FPGA Turnkey, set the `Objective` to `hdlcoder.Objective.None`.

For the tool-specific Tcl commands that are added to the synthesis project creation Tcl script, see “Synthesis Objective to Tcl Command Mapping”.

**RunTaskGenerateRTLCode — Enable task to generate RTL code and HDL top-level wrapper**

true (default) | false

Enable or disable workflow task to generate RTL code and an HDL top-level wrapper, specified as a `logical`. When enabled, this task also generates a constraint file that contains pin mapping information and clock constraints.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code** task.

**RunTaskCreateProject — Enable task to create synthesis tool project**

true (default) | false

Enable or disable task to create a synthesis tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

**RunTaskPerformLogicSynthesis — Enable task to launch synthesis tool and run logic synthesis**

true (default) | false

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

**RunTaskPerformMapping — Enable task to map synthesized logic to target device**

true (default) | false

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

**RunTaskPerformPlaceAndRoute — Enable task to run place and route process**

true (default) | false

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

#### **RunTaskRunSynthesis — Enable task to launch Xilinx Vivado and run synthesis**

`true (default) | false`

Enable or disable task to launch Xilinx Vivado and run synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Synthesis** task.

#### **RunTaskRunImplementation — Enable task to launch Xilinx Vivado and run implementation**

`true (default) | false`

Enable or disable task to launch Xilinx Vivado and run the implementation step, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Implementation** task.

#### **RunTaskGenerateProgrammingFile — Enable task to generate FPGA programming file**

`true (default) | false`

Enable or disable task to generate an FPGA programming file, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Programming File** task.

#### **RunTaskProgramTargetDevice — Enable task to program target device**

`true (default) | false`

Enable or disable task to download the FPGA programming file to the target device, specified as a `logical`. This task is available only when the target workflow is FPGA Turnkey.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Program Target Device** task.

### **AdditionalProjectCreationTclFiles** — Additional project creation Tcl files to include in your synthesis project

' ' (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: 'L:\file1.tcl;L:\file2.tcl;'

### **SkipPreRouteTimingAnalysis** — Skip pre-route timing analysis logical

false (default) | true

Skip pre-route timing analysis, specified as a logical. If your tool does not support early timing estimation, set to true.

When this option is enabled, `CriticalPathSource` is set to 'post-route'

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

### **IgnorePlaceAndRouteErrors** — Ignore place and route errors

false (default) | true

Ignore place and route errors, specified as a logical.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

### **IP Core Generation Workflow**

#### **ProjectFolder** — Folder for generated project files

' ' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project\_file\_folder'

#### **ReferenceDesignToolVersion** — Current reference design tool version

character vector

Current reference design tool version, specified as a character vector, for example '2017.4'. By default, the code generator selects a reference design tool version that is compatible with the current supported tool version. It is change this default reference design tool version, HDL Coder generates an error.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

### **IgnoreToolVersionMismatch — Ignore mismatch in reference design tool version**

false (default) | true

Whether you want the code generator to ignore a mismatch between the reference design tool version and the supported tool version, specified as a logical. By default, if there is a tool version mismatch, HDL Coder generates an error. If you set this option to true, HDL Coder generates a warning instead.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

### **RunTaskGenerateRTLCodeAndIPCore — Enable task to generate code and IP core**

true (default) | false

Enable or disable workflow task to generate code and IP core for embedded system, specified as a logical.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and IP Core** task.

### **RunTaskCreateProject — Enable task to create embedded system tool project**

true (default) | false

Enable or disable workflow task to create an embedded system tool project, specified as a logical.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Create Project** task.

### **RunTaskGenerateSoftwareInterfaceModel — Enable task to generate software interface model**

true (default) | false

Enable or disable workflow task to generate a software interface model with IP core driver blocks for embedded C code generation, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Generate Software Interface Model** task.

### **RunTaskBuildFPGABitstream — Enable task to generate bitstream for embedded system**

`true` (default) | `false`

Enable or disable workflow task to generate a bitstream for the embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Build FPGA Bitstream** task.

### **RunTaskProgramTargetDevice — Enable task to program connected target device**

`false` (default) | `true`

Enable or disable workflow task to program the connected target device, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Program Target Device** task.

### **IPCoreRepository — IP core repository folder path**

`' '` (default) | character vector

Full path to an IP core repository folder, specified as a character vector. The coder copies the generated IP core into the IP repository folder.

Example: `'L:\sandbox\work\IPfolder'`

### **GenerateIPCoreReport — Generate HTML documentation for the IP core**

`true` (default) | `false`

Option to generate HTML documentation for the IP core, specified as a `logical`. For details, see “Custom IP Core Report”.

### **Objective — Synthesis tool objective**

`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`



High-level synthesis tool objective, specified as one of these values.

<code>hdlcoder.Objective.None</code> (default)	Do not generate additional Tcl commands.
<code>hdlcoder.Objective.SpeedOptimized</code>	Generate synthesis tool Tcl commands to optimize for speed.
<code>hdlcoder.Objective.AreaOptimized</code>	Generate synthesis tool Tcl commands to optimize for area.
<code>hdlcoder.Objective.CompileOptimized</code>	Generate synthesis tool Tcl commands to optimize for compilation time.

If your synthesis tool is Xilinx ISE and your target workflow is Generic ASIC/FPGA or FPGA Turnkey, set the `Objective` to `hdlcoder.Objective.None`.

For the tool-specific Tcl commands that are added to the synthesis project creation Tcl script, see “Synthesis Objective to Tcl Command Mapping”.

### **EnableIPCaching — Create IP cache to reduce reference design synthesis time**

`false` (default) | `true`

Enable or disable IP caching, specified as a `logical`. When you enable IP caching, the code generator creates an IP cache. You can reuse this cache in subsequent project runs, which reduces reference design synthesis time.

In the HDL Workflow Advisor, you can specify this setting in the **Create Project** task.

### **OperatingSystem — Operating system**

`' '` (default) | character vector

Operating system for embedded processor, specified as a character vector. The operating system is board-specific.

### **AddLinuxDeviceDriver — Add IP core device driver**

`false` (default) | `true`

Option to insert the IP core node into the operating system device tree on the SD card on your board, specified as a `logical`. This option also restarts the operating system and adds the IP core driver as a loadable kernel module.

To use this option, your board must be connected.

**RunExternalBuild — Run build process externally**

true (default) | false

Option to run build process in parallel with MATLAB, specified as a logical. If this option is disabled, you cannot use MATLAB until the build is finished.

**ReportTimingFailure — Report timing failures as warnings or errors**

hdlcoder.ReportTimingFailure.Error (default) |  
hdlcoder.ReportTimingFailure.Warning

Select whether to report timing failures when generating the FPGA bitstream, specified as one of these values:

hdlcoder.ReportTimingFailure.Error (default)	Report timing failures as errors by default.
hdlcoder.ReportTimingFailure.Warning	Report timing failures as errors instead of warnings. Use this option if you have implemented the custom logic to resolve timing violations in your design.

**TclFileForSynthesisBuild — Use custom or default synthesis tool build script**

hdlcoder.BuildOption.Default (default) | hdlcoder.BuildOption.Custom

Select whether to use a custom or default synthesis tool build script, specified as one of these values:

hdlcoder.BuildOption.Default (default)	Use the default build script.
hdlcoder.BuildOption.Custom	Use a custom build script instead of the default build script.

**CustomBuildTclFile — Custom synthesis tool build script file**

' ' (default) | character vector

Full path to a custom synthesis tool build Tcl script file, specified as a character vector. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project. If `TclFileForSynthesisBuild` is set to `hdlcoder.BuildOption.Custom`, you must specify a file.

If you want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, this

Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

Example: 'C:\Temp\work\build.tcl'

### **Simulink Real-Time FPGA I/O**

#### **ProjectFolder — Folder for generated project files**

' ' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project\_file\_folder'

#### **ReferenceDesignToolVersion — Current reference design tool version**

character vector

Current reference design tool version, specified as a character vector, for example '2017.4'. By default, the code generator selects a reference design tool version that is compatible with the current supported tool version. If you change this default reference design tool version, HDL Coder generates an error.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

#### **IgnoreToolVersionMismatch — Ignore mismatch in reference design tool version**

false (default) | true

Whether you want the code generator to ignore a mismatch between the reference design tool version and the supported tool version, specified as a logical. By default, if there is a tool version mismatch, HDL Coder generates an error. If you set this option to true, HDL Coder generates a warning instead.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

#### **RunTaskGenerateRTLCodeAndIPCore — Enable task to generate code and IP core**

true (default) | false

Enable or disable workflow task to generate code and IP core for embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and IP Core** task.

### **RunTaskGenerateRTLCode – Enable task to generate RTL code and HDL top-level wrapper**

`true` (default) | `false`

Enable or disable workflow task to generate RTL code and an HDL top-level wrapper, specified as a `logical`. When enabled, this task also generates a constraint file that contains pin mapping information and clock constraints.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code** task.

### **RunTaskCreateProject – Enable task to create embedded system tool project**

`true` (default) | `false`

Enable or disable workflow task to create an embedded system tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Create Project** task.

### **RunTaskPerformLogicSynthesis – Enable task to launch synthesis tool and run logic synthesis**

`true` (default) | `false`

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

### **RunTaskPerformMapping – Enable task to map synthesized logic to target device**

`true` (default) | `false`

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

**RunTaskPerformPlaceAndRoute — Enable task to run place and route process**

`true` (default) | `false`

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

**RunTaskGenerateProgrammingFile — Enable task to generate FPGA programming file**

`true` (default) | `false`

Enable or disable task to generate an FPGA programming file, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Programming File** task.

**RunTaskGenerateSimulinkRealTimeInterface — Enable task to generate a model that contains an interface subsystem that you can plug into a Simulink Real-Time model**

`true` (default) | `false`

Enable or disable task to generate a Simulink Real-Time model that contains an interface subsystem, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Simulink Real-Time Interface** task.

**Objective — Synthesis tool objective**

`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`

High-level synthesis tool objective, specified as one of these values.

<code>hdlcoder.Objective.None</code> (default)	Do not generate additional Tcl commands.
<code>hdlcoder.Objective.SpeedOptimized</code>	Generate synthesis tool Tcl commands to optimize for speed.
<code>hdlcoder.Objective.AreaOptimized</code>	Generate synthesis tool Tcl commands to optimize for area.
<code>hdlcoder.Objective.CompileOptimized</code>	Generate synthesis tool Tcl commands to optimize for compilation time.

If your synthesis tool is Xilinx ISE and your target workflow is Generic ASIC/FPGA or FPGA Turnkey, set the `Objective` to `hdlcoder.Objective.None`.

For the tool-specific Tcl commands that are added to the synthesis project creation Tcl script, see “Synthesis Objective to Tcl Command Mapping”.

**AdditionalProjectCreationTclFiles** — Additional project creation Tcl files to include in your synthesis project

' ' (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: `'L:\file1.tcl;L:\file2.tcl;'`

**SkipPreRouteTimingAnalysis** — Skip pre-route timing analysis logical

false (default) | true

Skip pre-route timing analysis, specified as a `logical`. If your tool does not support early timing estimation, set to `true`.

When you enable this option, `CriticalPathSource` is set to `'post-route'`

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

**IgnorePlaceAndRouteErrors** — Ignore place and route errors

false (default) | true

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

### **RunTaskBuildFPGABitstream — Enable task to generate bitstream for embedded system**

true (default) | false

Enable or disable workflow task to generate a bitstream for the embedded system, specified as a logical.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Build FPGA Bitstream** task.

### **ReportTimingFailure — Report timing failures as warnings or errors**

hdlcoder.ReportTimingFailure.Error (default) |  
hdlcoder.ReportTimingFailure.Warning

Select whether to report timing failures when generating the FPGA bitstream, specified as one of these values:

hdlcoder.ReportTimingFailure.Error (default)	Report timing failures as errors by default.
hdlcoder.ReportTimingFailure.Warning	Report timing failures as errors instead of warnings. Use this option if you have implemented the custom logic to resolve timing violations in your design.

## **Methods**

export	Generate MATLAB script that recreates the workflow configuration
setAllTasks	Enable all tasks in workflow
clearAllTasks	Disable all tasks in workflow
validate	Check property values in HDL Workflow CLI configuration object

## **Examples**

## Configure and Run Generic ASIC/FPGA Workflow with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex® 7 device and uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 14:42:37 on 29/03/2018
% This script was generated using the following parameter values:
%   Filename   : 'S:\generic_workflow_example.m'
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('sfir_fixed', 'GenerateHDLTestBench', 'off');
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffgl761');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');

%% Workflow Configuration Settings
```



```
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Generi

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
hWC.CriticalPathSource = 'pre-route';
hWC.CriticalPathNumber = 1;
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `generic_workflow_example.m`, at the command line, enter:

```
generic_workflow_example.m
```

### Configure and Run FPGA-in-the-Loop with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA-in-the-Loop workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
%-----  
% HDL Workflow Script  
% Generated with MATLAB 9.5 (R2018b Prerelease) at 15:11:23 on 04/05/2018  
% This script was generated using the following parameter values:  
%   Filename   : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\h  
%   Overwrite  : true  
%   Comments   : true  
%   Headers    : true  
%   DUT        : 'sfir_fixed/symmetric_fir'  
% To view changes after modifying the workflow, run the following command:  
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');  
%-----  
  
%% Load the Model  
load_system('sfir_fixed');  
  
%% Restore the Model to default HDL parameters  
%hdlrestoreparams('sfir_fixed/symmetric_fir');  
  
%% Model HDL Parameters  
%% Set Model 'sfir_fixed' HDL parameters
```

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 25);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Xilinx Kintex-7 KC705 development board');
hdlset_param('sfir_fixed', 'Workflow', 'FPGA-in-the-Loop');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','FPGA-in-the-Loop');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = false;
hWC.RunTaskBuildFPGAInTheLoop = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskBuildFPGAInTheLoop' Task
hWC.IPAddress = '192.168.0.2';
hWC.MACAddress = '00-0A-35-02-21-8A';
hWC.SourceFiles = '';
hWC.Connection = 'Ethernet';
hWC.RunExternalBuild = true;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hwc`.

Run the HDL workflow script.

For example, if the script file name is `FIL_workflow_example.m`, at the command line, enter:

```
fil_workflow_example.m
```

### Configure and Run FPGA Turnkey Workflow with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA Turnkey workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:24:32 on 08/07/2015
% Parameter Values:
%   Filename   : 'S:\turnkey_workflow_example.m'
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA'

%% Load the Model
load_system('hdlcoderUARTServoControllerExample');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'HDLSubsystem', 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoderUARTServoControllerExample', ...
```

```

    'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoderUARTServoControllerExample', 'Workflow', 'FPGA Turnkey');

% Set Inport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterface', 'RS-232 Serial Port Rx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterface', 'RS-232 Serial Port Tx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_txd', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterface', 'LEDs General Purpose [0:7]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/version', ...
    'IOInterfaceMapping', '[0:3]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/pwm_output', ...
    'IOInterfaceMapping', '[0]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug1', ...
    'IOInterfaceMapping', '[1]');

% Set Output HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterface', 'Expansion Headers J6 Pin 2-64 [0:31]');

```

```
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/servo_debug2', ...
    'IOInterfaceMapping', '[2]');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE', ...
    'TargetWorkflow','FPGA Turnkey');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Perform Mapping Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set Properties related to Perform Place and Route Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `turnkey_workflow_example.m`, at the command line, enter:

```
turnkey_workflow_example.m
```

## Configure and Run IP Core Generation Workflow with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an IP core generation workflow script that targets the Altera Cyclone V SoC development kit and uses the Altera Quartus II synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:42:16 on 08/07/2015
% Parameter Values:
%   Filename   : 'S:\ip_core_gen_workflow_example.m'
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'hdlcoder_led_blinking/led_counter'

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoder_led_blinking', ...
    'HDLSubsystem', 'hdlcoder_led_blinking/led_counter');
hdlset_param('hdlcoder_led_blinking', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_led_blinking', ...
    'ReferenceDesign', 'Default system (Qsys 14.0)');
hdlset_param('hdlcoder_led_blinking', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Altera QUARTUS II');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Cyclone V');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', '5CSXFC6D6F31C6');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
```

```
hdlset_param('hdlcoder_led_blinking', ...
    'TargetPlatform', 'Altera Cyclone V SoC development kit - Rev.D');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', ...
    'ProcessorFPGASynchronization', 'Free running');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceMapping', 'x"100"');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceMapping', 'x"104"');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', 'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', ...
    'IOInterfaceMapping', 'x"108"');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Altera QUARTUS II', ...
    'TargetWorkflow','IP Core Generation');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterfaceModel = false;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskProgramTargetDevice = false;
```



```
% Set Properties related to Generate RTL Code And IP Core Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.AreaOptimized;

% Set Properties related to Generate Software Interface Model Task
hWC.OperatingSystem = '';
hWC.AddLinuxDeviceDriver = false;

% Set Properties related to Build FPGA Bitstream Task
hWC.RunExternalBuild = true;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `ip_core_workflow_example.m`, at the command line, enter:

```
ip_core_gen_workflow_example.m
```

### **Configure and Run Simulink Real-Time FPGA I/O Workflow for ISE-Based Boards with a Script**

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a Simulink Real-Time FPGA I/O workflow script that targets the Speedgoat I0331 board that uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```

%-----
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:14 on 08/05/2018
% This script was generated using the following parameter values:
%   Filename   : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\h
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Spartan6');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc6slx150');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fgg676');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 75);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat I0331');
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE','TargetWorkflow','Simulink F

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '';
hWC.IgnoreToolVersionMismatch = false;

```

```
% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCode = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskPerformMapping' Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set properties related to 'RunTaskPerformPlaceAndRoute' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

```
slrt_workflow_example.m
```

### **Configure and Run Simulink Real-Time FPGA I/O Workflow for Vivado-Based Boards with a Script**

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a Simulink Real-Time FPGA I/O workflow script that targets the Speedgoat I0333-325K board that uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```

%-----
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:33 on 08/05/2018
% This script was generated using the following parameter values:
%   Filename   : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\h
%   Overwrite  : true
%   Comments   : true
%   Headers    : true
%   DUT        : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-----

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 100);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat I0333-325K');
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');

%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

```

```
hWC.ReferenceDesignToolVersion = '2017.4';
hWC.IgnoreToolVersionMismatch = false;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;
hWC.GenerateIPCoreTestbench = false;
hWC.CustomIPTopHDLFile = '';
hWC.AXI4RegisterReadback = false;
hWC.IPDataCaptureBufferSize = '128';

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.ReportTimingFailure = hdlcoder.ReportTiming.Error;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

slrt\_workflow\_example.m

## **See Also**

### **Functions**

hdlcoder.runWorkflow

### **Topics**

“Run HDL Workflow with a Script”

**Introduced in R2015b**

## export

**Class:** hdlcoder.WorkflowConfig

**Package:** hdlcoder

Generate MATLAB script that recreates the workflow configuration

## Syntax

`export(Name,Value)`

## Description

`export(Name,Value)` generates MATLAB commands that can recreate the current workflow configuration, with additional options specified by one or more `Name,Value` pair arguments.

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **Filename — Full path to exported script file**

`''` (default) | character vector

Full path to the exported MATLAB script file, specified as a character vector. If the path is empty, the MATLAB commands are displayed in the Command Window, but not saved in a file.

Example: `'L:\sandbox\work\hdlworkflow.m'`

### **Overwrite — Overwrite existing file**

`false` (default) | `true`

Specify whether to overwrite the existing file as a `logical`.

### **Comments — Include comments**

`true` (default) | `false`

Specify whether to include comments in the command list or script as a `logical`.

### **Headers — Include headers**

`true` (default) | `false`

Specify whether to include a header in the command list or script as a `logical`.

### **DUT — Full path to DUT**

`''` (default) | character vector

Full path to the DUT, specified as a character vector.

Example: `'hdlcoder_led_blinking/led_counter'`

## **See Also**

### **Classes**

`hdlcoder.WorkflowConfig`

### **Topics**

“Run HDL Workflow with a Script”

**Introduced in R2015b**



# setAllTasks

**Class:** hdlcoder.WorkflowConfig

**Package:** hdlcoder

Enable all tasks in workflow

## Syntax

```
setAllTasks
```

## Description

setAllTasks enables all workflow tasks in the hdlcoder.WorkflowConfig object.

If you do not want to enable each task individually, use this method. For example, if you want to run all tasks but one, you can run hdlcoder.WorkflowConfig.setAllTasks, then disable the task that you want to skip.

## See Also

### Functions

hdlcoder.WorkflowConfig.clearAllTasks

### Classes

hdlcoder.WorkflowConfig

## Topics

“Run HDL Workflow with a Script”

**Introduced in R2015b**

## clearAllTasks

**Class:** hdlcoder.WorkflowConfig

**Package:** hdlcoder

Disable all tasks in workflow

## Syntax

`clearAllTasks`

## Description

`clearAllTasks` disables all workflow tasks in the `hdlcoder.WorkflowConfig` object.

If you do not want to disable each task individually, use this method. For example, if you want to run a single task, you can run `hdlcoder.WorkflowConfig.clearAllTasks`, then enable the task that you want to run.

## See Also

### Functions

`hdlcoder.WorkflowConfig.setAllTasks`

### Classes

`hdlcoder.WorkflowConfig`

## Topics

“Run HDL Workflow with a Script”

**Introduced in R2015b**

# validate

**Class:** hdlcoder.WorkflowConfig

**Package:** hdlcoder

Check property values in HDL Workflow CLI configuration object

## Syntax

validate

## Description

validate verifies that the `hdlcoder.WorkflowConfig` object has acceptable values for all required properties, and that property values have valid data types. If validation fails, you get an error message.

## See Also

`hdlcoder.WorkflowConfig`

## Topics

“Run HDL Workflow with a Script”

**Introduced in R2015b**

## hdlcoder.runWorkflow

Run HDL code generation and deployment workflow

### Syntax

```
hdlcoder.runWorkflow(DUT)  
hdlcoder.runWorkflow(DUT,workflow_config)
```

### Description

`hdlcoder.runWorkflow(DUT)` runs the HDL code generation and deployment workflow with default workflow configuration settings.

`hdlcoder.runWorkflow(DUT,workflow_config)` runs the HDL code generation and deployment workflow according to the specified workflow configuration, `workflow_config`.

A best practice is to use the HDL Workflow Advisor to configure the workflow, then export a workflow script. The commands in the workflow script create and configure a workflow configuration object that matches the settings in the HDL Workflow Advisor. The script includes the `hdlcoder.runWorkflow` command. To learn more, see “Run HDL Workflow with a Script”.

### Examples

#### Run Workflow with Configuration Object

This example is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex-7 device. It uses the Xilinx Vivado synthesis tool. The example generates HDL code for the `sfir_fixed` model, and performs FPGA synthesis and analysis.

## Before running the Workflow

Before running the workflow, you must have the synthesis tool installed. Use `hdlsetuptoolpath` to specify the path to your synthesis tool.

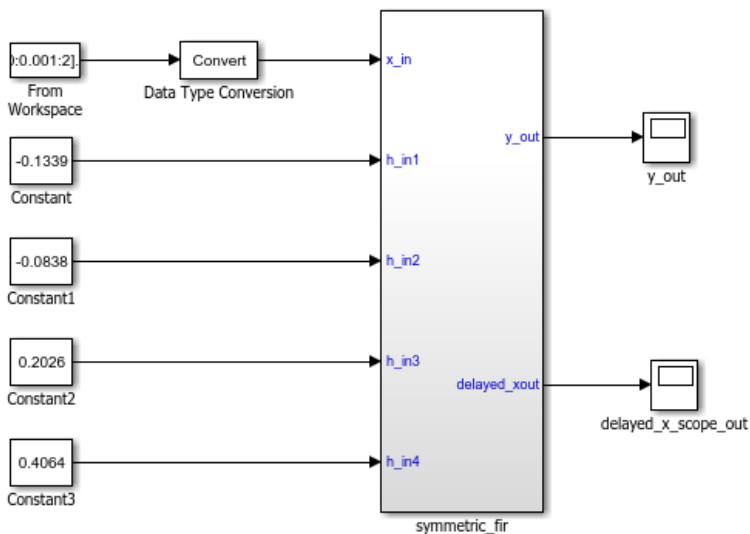
```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', ...
'L:\Xilinx\Vivado\2016.2\bin\vivado.bat');
```

Prepending following Xilinx Vivado path(s) to the system path:  
L:\Xilinx\Vivado\2016.2\bin

## Specify the model for running the workflow

To run the HDL workflow with default settings for a DUT subsystem, `modelName/DUT`, at the command line, enter:

```
open_system('sfir_fixed');
```



This example shows how to use HDL Coder to check, generate, and verify HDL for a fixed-point symmetric FIR filter. In MATLAB, type the following:  
`checkhdl('sfir_fixed/symmetric_fir')`  
`makehdl('sfir_fixed/symmetric_fir')`  
`makehdltb('sfir_fixed/symmetric_fir')`  
 Or double-click the blue button at the bottom to see the dialog.

Launch HDL Dialog

Run Demo

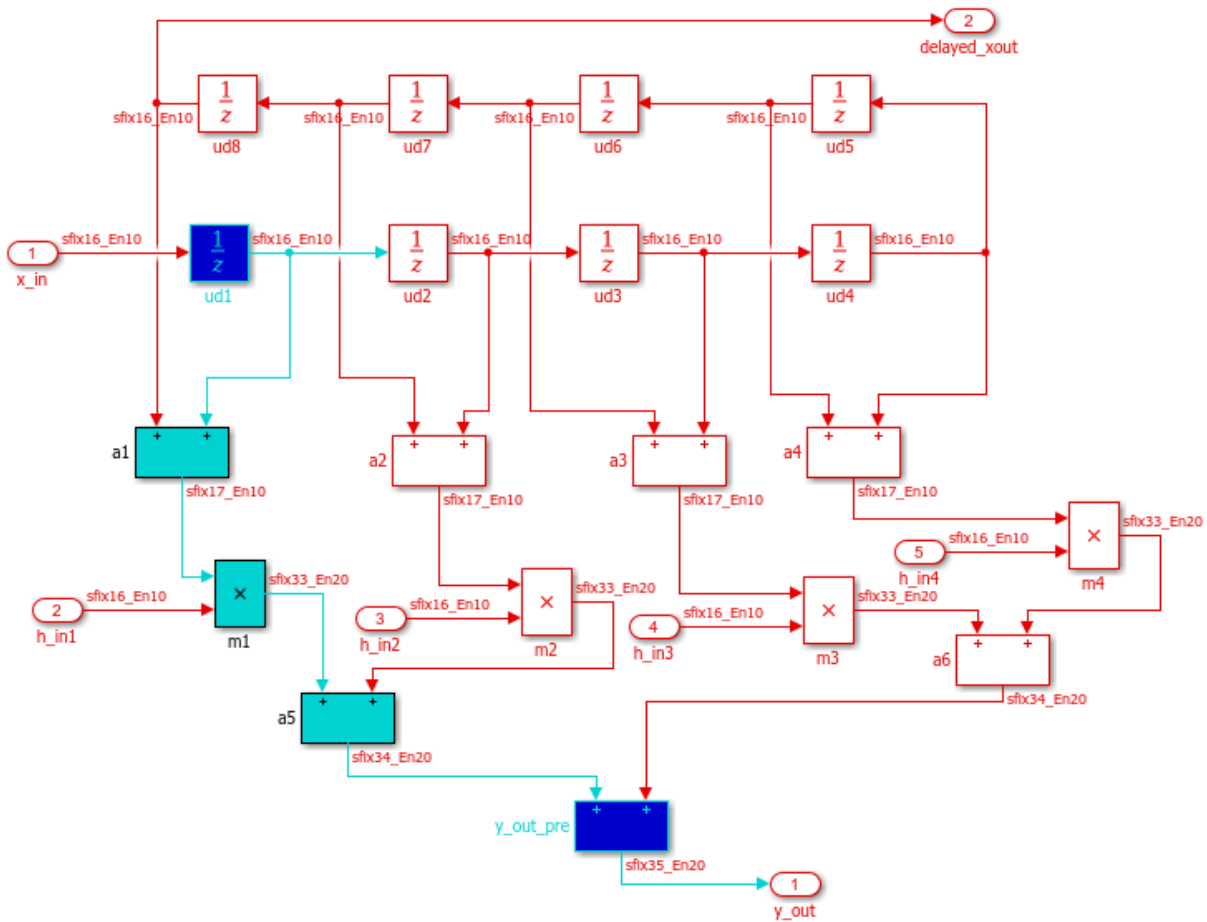
Copyright 2007 The MathWorks, Inc.



```

### Highlighting CP 1 from 'sfir_fixed/symmetric_fir/ud1_out1' to 'sfir_fixed/symmetri
### Click <a href="matlab:hdlannotatpath('reset')">here</a> to reset highlighting.
### Workflow complete.

```



## Input Arguments

**DUT — Full path to DUT**

' ' (default) | character vector

Full path to the DUT, specified as a character vector.

Example: 'hdlcoder\_led\_blinking/led\_counter'

### **workflow\_config** — Workflow configuration

`hdlcoder.WorkflowConfig`

HDL code generation and deployment workflow configuration, specified as an `hdlcoder.WorkflowConfig` object.

## See Also

### Functions

`hdlcoder.WorkflowConfig.clearAllTasks` |  
`hdlcoder.WorkflowConfig.setAllTasks`

### Classes

`hdlcoder.WorkflowConfig`

### Topics

“Run HDL Workflow with a Script”

**Introduced in R2015b**



# hdlcoder.OptimizationConfig class

**Package:** hdlcoder

hdlcoder.optimizeDesign configuration object

## Description

Use the `hdlcoder.OptimizationConfig` object to set options for the `hdlcoder.optimizeDesign` function.

## Maximum Clock Frequency Configuration

To configure `hdlcoder.optimizeDesign` to maximize the clock frequency of your design:

- Set `ExplorationMode` to `hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`.
- Set `ResumptionPoint` to the default, `' '`.

You can optionally set `IterationLimit` and `TestbenchGeneration` to nondefault values. HDL Coder ignores the `TargetFrequency` setting.

## Target Clock Frequency Configuration

To configure `hdlcoder.optimizeDesign` to meet a target clock frequency:

- Set `ExplorationMode` to `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`.
- Set `TargetFrequency` to your target clock frequency.
- Set `ResumptionPoint` to the default, `' '`.

You can optionally set `IterationLimit` and `TestbenchGeneration` to nondefault values.

## Resume From Interruption Configuration

To configure `hdlcoder.optimizeDesign` to resume after an interruption, specify `ResumptionPoint`.

When you set `ResumptionPoint` to a nondefault value, the other properties are ignored.

## Construction

`optimcfg = hdlcoder.OptimizationConfig` creates an `hdlcoder.OptimizationConfig` object for automatic iterative HDL design optimization.

## Properties

### ExplorationMode — Optimization target mode

`hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency` (default) | `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`

Optimization target mode, specified as one of these values:

<code>hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency</code>	Optimizes the design to try to achieve the maximum clock frequency
	<code>hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency</code> is the default.
<code>hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency</code>	Optimizes the design to try to achieve the specified target clock frequency

### IterationLimit — Maximum number of iterations

1 (default) | positive integer

Maximum number of optimization iterations before exiting, specified as a positive integer.

If `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`, HDL Coder runs this number of iterations.

If `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`, HDL Coder runs the number of iterations needed to meet the target frequency. Otherwise, the coder runs the maximum number of iterations.

### **ResumptionPoint — Folder containing optimization data from earlier iteration**

' ' (default) | character vector

Name of folder that contains previously-generated optimization iteration data, specified as a character vector. The folder is a subfolder of `hdlexpl`, and the folder name begins with the character vector, `Iter`.

When you set `ResumptionPoint` to a nondefault value, `hdlcoder.optimizeDesign` ignores the other configuration object properties.

Example: `'Iter1-26-Sep-2013-10-19-13'`

### **TargetFrequency — Target clock frequency**

Inf (default) | double

Target clock frequency, specified as a double in MHz. Specify when `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`.

## **Examples**

### **Configure `hdlcoder.optimizeDesign` for maximum clock frequency**

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);
```

Set your synthesis tool and target device options.

```
hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
    'SynthesisToolChipFamily', 'Zynq', ...
    'SynthesisToolDeviceName', 'xc7z030', ...
    'SynthesisToolPackageName', 'fbg484', ...
    'SynthesisToolSpeedValue', '-3')
```

Enable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'on');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Set the iteration limit to 10.

```
oc.IterationLimit = 10;
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
```

```
Iteration 0
```

```
Generate and synthesize HDL code ...
(CP ns) 16.26 (Constraint ns) 5.85 (Elapsed s) 143.66 Iteration 1
Generate and synthesize HDL code ...
(CP ns) 16.26 (Constraint ns) 5.85 (Elapsed s) 278.72 Iteration 2
Generate and synthesize HDL code ...
(CP ns) 10.25 (Constraint ns) 12.73 (Elapsed s) 427.22 Iteration 3
Generate and synthesize HDL code ...
(CP ns) 9.55 (Constraint ns) 9.73 (Elapsed s) 584.37 Iteration 4
Generate and synthesize HDL code ...
(CP ns) 9.55 (Constraint ns) 9.38 (Elapsed s) 741.04 Iteration 5
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 9.55 ns Elapsed : 741.04 s
```

```

Iteration 0: (CP ns) 16.26      (Constraint ns) 5.85      (Elapsed s) 143.66
Iteration 1: (CP ns) 16.26      (Constraint ns) 5.85      (Elapsed s) 278.72
Iteration 2: (CP ns) 10.25      (Constraint ns) 12.73     (Elapsed s) 427.22
Iteration 3: (CP ns) 9.55       (Constraint ns) 9.73      (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55       (Constraint ns) 9.38      (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-04-41
Validation model: gm_sfir_fixed_vnl

```

Then HDL Coder stops after five iterations because the fourth and fifth iterations had the same critical path, which indicates that the coder has found the minimum critical path. The design's maximum clock frequency after optimization is 1 / 9.55 ns, or 104.71 MHz.

### Configure `hdlcoder.optimizeDesign` for target clock frequency

Open the model and specify the DUT subsystem.

```

model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);

```

Set your synthesis tool and target device options.

```

hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
    'SynthesisToolChipFamily', 'Zynq', ...
    'SynthesisToolDeviceName', 'xc7z030', ...
    'SynthesisToolPackageName', 'fbg484', ...
    'SynthesisToolSpeedValue', '-3')

```

Disable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'off');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to stop after it reaches a clock frequency of 50MHz, or 10 iterations, whichever comes first.

```
oc.ExplorationMode = ...  
    hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency;  
oc.TargetFrequency = 50;  
oc.IterationLimit = 10; =
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed','GenerateHDLTestBench','off');  
hdlset_param('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir');  
hdlset_param('sfir_fixed','SynthesisTool','Xilinx ISE');  
hdlset_param('sfir_fixed','SynthesisToolChipFamily','Zynq');  
hdlset_param('sfir_fixed','SynthesisToolDeviceName','xc7z030');  
hdlset_param('sfir_fixed','SynthesisToolPackageName','fbg484');  
hdlset_param('sfir_fixed','SynthesisToolSpeedValue','-3');
```

```
Iteration 0
```

```
Generate and synthesize HDL code ...
```

```
(CP ns) 16.26      (Constraint ns) 20.00      (Elapsed s) 134.02 Iteration 1
```

```
Generate and synthesize HDL code ...
```

```
Exiting because constraint (20.00 ns) has been met (16.26 ns).
```

```
Summary report: summary.html
```

```
Achieved Critical Path (CP) Latency : 16.26 ns      Elapsed : 134.02 s
```

```
Iteration 0: (CP ns) 16.26      (Constraint ns) 20.00      (Elapsed s) 134.02
```

```
Final results are saved in
```

```
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-14
```

```
Validation model: gm_sfir_fixed_vnl
```

Then HDL Coder stops after one iteration because it has achieved the target clock frequency. The critical path is 16.26 ns, a clock frequency of 61.50 GHz.

### Configure `hdlcoder.optimizeDesign` to resume from interruption

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';  
dutSubsys = 'symmetric_fir';
```

```
open_system(model);
hdlset_param(model, 'HDLSubsystem', [model, '/', dutSubsys]);
```

Set your synthesis tool and target device options to the same values as in the interrupted run.

```
hdlset_param(model, 'SynthesisTool', 'Xilinx ISE', ...
                 'SynthesisToolChipFamily', 'Zynq', ...
                 'SynthesisToolDeviceName', 'xc7z030', ...
                 'SynthesisToolPackageName', 'fbg484', ...
                 'SynthesisToolSpeedValue', '-3')
```

Enable HDL test bench generation.

```
hdlset_param(model, 'GenerateHDLTestBench', 'on');
```

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to run using data from the first iteration of a previous run.

```
oc.ResumptionPoint = 'Iter5-07-Jan-2014-17-04-29';
```

Optimize the model.

```
hdlcoder.optimizeDesign(model, oc)
```

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
```

Try to resume from resumption point: Iter5-07-Jan-2014-17-04-29

Iteration 5

Generate and synthesize HDL code ...

Exiting because critical path cannot be further improved.

Summary report: summary.html

Achieved Critical Path (CP) Latency : 9.55 ns	Elapsed : 741.04 s
Iteration 0: (CP ns) 16.26 (Constraint ns) 5.85	(Elapsed s) 143.66
Iteration 1: (CP ns) 16.26 (Constraint ns) 5.85	(Elapsed s) 278.72
Iteration 2: (CP ns) 10.25 (Constraint ns) 12.73	(Elapsed s) 427.22

```
Iteration 3: (CP ns) 9.55      (Constraint ns) 9.73      (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55      (Constraint ns) 9.38      (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-30
Validation model: gm_sfir_fixed_vnl
```

Then coder stops after one additional iteration because it has achieved the target clock frequency. The critical path is 9.55 ns, or a clock frequency of 104.71 MHz.

### See Also

`hdlcoder.optimizeDesign`



# Function Reference for HDL Code Generation from MATLAB

---

## codegen

Generate HDL code from MATLAB code

### Syntax

```
codegen -confighdlcfg matlab_design_name  
codegen -confighdlcfg -float2fixed fixptcfg matlab_design_name
```

### Description

`codegen -confighdlcfg matlab_design_name` generates HDL code from MATLAB code.

`codegen -confighdlcfg -float2fixed fixptcfg matlab_design_name` converts floating-point MATLAB code to fixed-point code, then generates HDL code.

### Examples

#### Generate Verilog Code from MATLAB Code

Create a `coder.HdlConfig` object, `hdlcfg`.

```
hdlcfg = coder.config('hdl'); % Create a default 'hdl' config
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set the target language to Verilog.

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL code from your MATLAB design. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -config hdlcfg mlhdlc_dti
```

## Generate HDL Code from Floating-Point MATLAB Code

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Create a `coder.HdlConfig` object, `hdlcfg`, with default settings.

```
hdlcfg = coder.config('hdl');
```

Convert your floating-point MATLAB design to fixed-point, and generate HDL code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

## Input Arguments

### **hdlcfg** — HDL code generation configuration

`coder.HdlConfig`

HDL code generation configuration options, specified as a `coder.HdlConfig` object.

Create a `coder.HdlConfig` object using the HDL `coder.config` function.

### **matlab\_design\_name** — MATLAB design function name

character vector

Name of top-level MATLAB function for which you want to generate HDL code.

### **fixptcfg** — Floating-point to fixed-point conversion configuration

`coder.FixptConfig`

Floating-point to fixed-point conversion configuration options, specified as a `coder.FixptConfig` object.

Use `fixptcfg` when generating HDL code from floating-point MATLAB code. Create a `coder.FixptConfig` object using the HDL `coder.config` function.

### See Also

`coder.FixptConfig` | `coder.HdlConfig` | `coder.config`

### Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”

**Introduced in R2013a**

# coder.approximation

Create function replacement configuration object

## Syntax

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

## Description

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the `codegen` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

## Examples

### Replace Log Function with Default Lookup Table

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```

### Replace Log Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...  
'FunctionNamePrefix','log_replace');
```

### Replace Log Function with Optimized Lookup Table

Create a function replacement configuration object using the 'OptimizeLUTSize' option to specify to replace the log function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...  
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...  
'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

### Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, saturateExp, with an optimized lookup table.

Create a custom function, saturateExp.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the saturateExp function with an optimized lookup table. Because the saturateExp function is not listed as a function for which coder.approximation can generate an approximation automatically, you must specify the CandidateFunction property.

```
saturateExp = @(x) 1/(1+exp(-x));  
custAppx = coder.approximation('Function','saturateExp',...
```

```
'CandidateFunction', saturateExp,...
'NumberOfPoints',50,'InputRange',[0,10]);
```

## Input Arguments

### **function\_name** — Name of the function to replace

```
'acos' | 'acosd' | 'acosh' | 'acoth' | 'asin' | 'asind' | 'asinh' | 'atan' |
'atand' | 'atanh' | 'cos' | 'cosd' | 'cosh' | 'erf' | 'erfc' | 'exp' | 'log' |
'normcdf' | 'reallog' | 'realsqrt' | 'reciprocal' | 'rsqrt' | 'sin' | 'sinc' |
'sind' | 'sinh' | 'sqrt' | 'tan' | 'tand'
```

Name of function to replace, specified as a string. The function must be one of the listed functions.

Example: 'sqrt'

Data Types: char

## 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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Function', 'log'

### **Architecture** — Architecture of lookup table approximation

```
'LookupTable' (default) | 'Flat'
```

Architecture of the lookup table approximation, specified as the comma-separated pair consisting of 'Architecture' and a string. Use this argument when you want to specify the architecture for the lookup table. The `Flat` architecture does not use interpolation.

Data Types: char

### **CandidateFunction** — Function handle of the replacement function

```
function handle | string
```

Function handle of the replacement function, specified as the comma-separated pair consisting of 'CandidateFunction' and a function handle or string referring to a function handle. Use this argument when the function that you want to replace is not

listed under `function_name`. Specify the function handle or string referring to a function handle of the function that you want to replace. You can define the function in a file or as an anonymous function.

If you do not specify a candidate function, then the function you chose to replace using the `Function` property is set as the `CandidateFunction`.

Example: `'CandidateFunction', @(x) (1./(1+x))`

Data Types: `function_handle` | `char`

### **ErrorThreshold — Error threshold value used to calculate optimal lookup table size**

0.001 (default) | nonnegative scalar

Error threshold value used to calculate optimal lookup table size, specified as the comma-separated pair consisting of `'ErrorThreshold'` and a nonnegative scalar. If `'OptimizeLUTSize'` is true, this argument is required.

### **Function — Name of function to replace with a lookup table approximation**

`function_name`

Name of function to replace with a lookup table approximation, specified as the comma-separated pair consisting of `'Function'` and a string. The function must be continuous and stateless. If you specify one of the functions that is listed under `function_name`, the conversion process automatically provides a replacement function. Otherwise, you must also specify the `'CandidateFunction'` argument for the function that you want to replace.

Example: `'Function','log'`

Example: `'Function','my_log','CandidateFunction',@my_log`

Data Types: `char`

### **FunctionNamePrefix — Prefix for generated fixed-point function names**

`'replacement_'` (default) | string

Prefix for generated fixed-point function names, specified as the comma-separated pair consisting of `'FunctionNamePrefix'` and a string. The name of a generated function consists of this prefix, followed by the original MATLAB function name.

Example: `'log_replace_'`



**InputRange — Range over which to replace the function**

[ ] (default) | 2x1 row vector | 2xN matrix

Range over which to replace the function, specified as the comma-separated pair consisting of 'InputRange' and a 2-by-1 row vector or a 2-by-N matrix.

Example: [-1 1]

**InterpolationDegree — Interpolation degree**

1 (default) | 0 | 2 | 3

Interpolation degree, specified as the comma-separated pair consisting of 'InterpolationDegree' and 1 (linear), 0 (none), 2 (quadratic), or 3 (cubic).

**NumberOfPoints — Number of points in lookup table**

1000 (default) | positive integer

Number of points in lookup table, specified as the comma-separated pair consisting of 'NumberOfPoints' and a positive integer.

**OptimizeIterations — Number of iterations**

25 (default) | positive integer

Number of iterations to run when optimizing the size of the lookup table, specified as the comma-separated pair consisting of 'OptimizeIterations' and a positive integer.

**OptimizeLUTSize — Optimize lookup table size**

false (default) | true

Optimize lookup table size, specified as the comma-separated pair consisting of 'OptimizeLUTSize' and a logical value. Setting this property to true generates an area-optimal lookup table, that is, the lookup table with the minimum possible number of points. This lookup table is optimized for size, but might not be speed efficient.

**PipelinedArchitecture — Option to enable pipelining**

false (default) | true

Option to enable pipelining, specified as the comma-separated pair consisting of 'PipelinedArchitecture' and a logical value.

## Output Arguments

**q** — Function replacement configuration object, returned as a `coder.mathfcngenerator.LookupTable` or a `coder.mathfcngenerator.Flat` configuration object

`coder.mathfcngenerator.LookupTable` configuration object |  
`coder.mathfcngenerator.Flat` configuration object

Function replacement configuration object. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `codegen` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point code.

Property	Default Value
Auto-replace function	' '
InputRange	[]
FunctionNamePrefix	'replacement_'
Architecture	LookupTable (read only)
NumberOfPoints	1000
InterpolationDegree	1
ErrorThreshold	0.001
OptimizeLUTSize	false
OptimizeIterations	25

## See Also

### Classes

`coder.FixptConfig`

### Functions

`codegen`

### Topics

“Replace the exp Function with a Lookup Table”

“Replace a Custom Function with a Lookup Table”

“Replacing Functions Using Lookup Table Approximations”

**Introduced in R2014b**

## **coder.config**

Create HDL Coder code generation configuration objects

### **Syntax**

```
config_obj = coder.config('hdl')  
config_obj = coder.config('fixpt')
```

### **Description**

`config_obj = coder.config('hdl')` creates a `coder.HdlConfig` configuration object for use with the HDL codegen function when generating HDL code from MATLAB code.

`config_obj = coder.config('fixpt')` creates a `coder.FixptConfig` configuration object for use with the HDL codegen function when generating HDL code from floating-point MATLAB code. The `coder.FixptConfig` object configures the floating-point to fixed-point conversion.

### **Examples**

#### **Generate HDL Code from Floating-Point MATLAB Code**

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Create a `coder.HdlConfig` object, `hdlcfg`, with default settings.

```
hdlcfg = coder.config('hdl');
```

Convert your floating-point MATLAB design to fixed-point, and generate HDL code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

## See Also

`codegen` | `coder.FixptConfig` | `coder.HdlConfig`

## Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”

**Introduced in R2013a**

## addDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Add design range specification to parameter

### Syntax

```
addDesignRangeSpecification(fcnName,paramName,designMin, designMax)
```

### Description

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)` specifies the minimum and maximum values allowed for the parameter, `paramName`, in function, `fcnName`. The fixed-point conversion process uses this design range information to derive ranges for downstream variables in the code.

### Input Arguments

**fcnName** — Function name

string

Function name, specified as a string.

Data Types: char

**paramName** — Parameter name

string

Parameter name, specified as a string.

Data Types: char

**designMin** — Minimum value allowed for this parameter

scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: double

**designMax** — Maximum value allowed for this parameter

scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: double

## Examples

## See Also

# addFunctionReplacement

**Class:** coder.FixptConfig

**Package:** coder

Replace floating-point function with fixed-point function during fixed-point conversion

## Syntax

```
addFunctionReplacement(floatFn, fixedFn)
```

## Description

`addFunctionReplacement(floatFn, fixedFn)` specifies a function replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion in the HDL code generation workflow, the conversion process replaces the specified floating-point function with the specified fixed-point function. The fixed-point function must be in the same folder as the floating-point function or on the MATLAB path.

## Input Arguments

**floatFn** — Name of floating-point function

' ' (default) | string

Name of floating-point function, specified as a string.

**fixedFn** — Name of fixed-point function

' ' (default) | string

Name of fixed-point function, specified as a string.

## Examples



## Specify Function Replacement in Fixed-Point Conversion Configuration Object

Create a fixed-point code configuration object, `fixpCfg`, with a test bench, `myTestbenchName`.

```
fixpCfg = coder.config('fixpt');  
fixpCfg.TestBenchName = 'myTestbenchName';  
fixpCfg.addFunctionReplacement('min', 'fi_min');  
codegen -float2fixed fixpCfg designName
```

Specify that the floating-point function, `min`, should be replaced with the fixed-point function, `fi_min`.

```
fixpCfg.addFunctionReplacement('min', 'fi_min');
```

When you generate code, the code generator replaces instances of `min` with `fi_min` during floating-point to fixed-point conversion.

## Alternatives

You can specify function replacements in the HDL Workflow Advisor. See “Function Replacements”.

## See Also

`codegen` | `coder.FixptConfig` | `coder.config`

# clearDesignRangeSpecifications

**Class:** coder.FixptConfig

**Package:** coder

Clear all design range specifications

## Syntax

```
clearDesignRangeSpecifications()
```

## Description

`clearDesignRangeSpecifications()` clears all design range specifications.

## Examples

### Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

## See Also

# getDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Get design range specifications for parameter

## Syntax

```
[designMin, designMax] = getDesignRangeSpecification(fcnName,  
paramName)
```

## Description

[designMin, designMax] = getDesignRangeSpecification(fcnName, paramName) gets the minimum and maximum values specified for the parameter, paramName, in function, fcnName.

## Input Arguments

**fcnName — Function name**

string

Function name, specified as a string.

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Data Types: char

## Output Arguments

### **designMin** — Minimum value allowed for this parameter

scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: double

### **designMax** — Maximum value allowed for this parameter

scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: double

## Examples

### Get Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')

designMin =

    -1

designMax =

     1
```

## See Also

# hasDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Determine whether parameter has design range

## Syntax

```
hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)
```

## Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)`  
returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

## Input Arguments

**fcnName — Name of function**

string

Function name, specified as a string.

Example: 'dti'

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Example: 'dti'

Data Types: char

## Output Arguments

**hasDesignRange** — Parameter has design range

true | false

Parameter has design range, returned as a boolean.

Data Types: logical

## Examples

### Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')

hasDesignRanges =

     1
```

## See Also

# removeDesignRangeSpecification

**Class:** coder.FixptConfig

**Package:** coder

Remove design range specification from parameter

## Syntax

```
removeDesignRangeSpecification(fcnName,paramName)
```

## Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

## Input Arguments

**fcnName — Name of function**

string

Function name, specified as a string.

Data Types: char

**paramName — Parameter name**

string

Parameter name, specified as a string.

Data Types: char

## Examples

### Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

### See Also



# **Class Reference for HDL Code Generation from MATLAB**

---

## **coder.FixptConfig class**

**Package:** coder

Floating-point to fixed-point conversion configuration object

### **Description**

A `coder.FixptConfig` object contains the configuration parameters that the HDL codegen function requires to convert floating-point MATLAB code to fixed-point MATLAB code during HDL code generation. Use the `-float2fixed` option to pass this object to the codegen function.

### **Construction**

`fixptcfg = coder.config('fixpt')` creates a `coder.FixptConfig` object for floating-point to fixed-point conversion.

### **Properties**

#### **ComputeDerivedRanges**

Enable derived range analysis.

Values: `true`|`false` (default)

#### **ComputeSimulationRanges**

Enable collection and reporting of simulation range data. If you need to run a long simulation to cover the complete dynamic range of your design, consider disabling simulation range collection and running derived range analysis instead.

Values: `true` (default)|`false`

#### **DefaultFractionLength**

Default fixed-point fraction length.

Values: 4 (default) | positive integer

### **DefaultSignedness**

Default signedness of variables in the generated code.

Values: 'Automatic' (default) | 'Signed' | 'Unsigned'

### **DefaultWordLength**

Default fixed-point word length.

Values: 14 (default) | positive integer

### **DetectFixptOverflows**

Enable detection of overflows using scaled doubles.

Values: true | false (default)

### **fimath**

fimath properties to use for conversion.

Values: fimath('RoundingMethod', 'Floor', 'OverflowAction', 'Wrap', 'ProductMode', 'FullPrecision', 'SumMode', 'FullPrecision') (default) | string

### **FixPtFileNameSuffix**

Suffix for fixed-point file names.

Values: '\_fixpt' | string

### **LaunchNumericTypesReport**

View the numeric types report after the software has proposed fixed-point types.

Values: true (default) | false

### **LogI0ForComparisonPlotting**

Enable simulation data logging to plot the data differences introduced by fixed-point conversion.

Values: `true` (default) | `false`

### **OptimizeWholeNumber**

Optimize the word lengths of variables whose simulation min/max logs indicate that they are always whole numbers.

Values: `true` (default) | `false`

### **PlotFunction**

Name of function to use for comparison plots.

`LogIOForComparisonPlotting` must be set to `true` to enable comparison plotting. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

Values: `' '` (default) | string

### **PlotWithSimulationDataInspector**

Use Simulation Data Inspector for comparison plots.

`LogIOForComparisonPlotting` must be set to `true` to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

Values: `true` | `false` (default)

### **ProposeFractionLengthsForDefaultWordLength**

Propose fixed-point types based on `DefaultWordLength`.

Values: `true` (default) | `false`

### **ProposeTargetContainerTypes**

By default (`false`), propose data types with the minimum word length needed to represent the value. When set to `true`, propose data type with the smallest word length that can

represent the range and is suitable for C code generation ( 8,16,32, 64 ... ). For example, for a variable with range [0..7], propose a word length of 8 rather than 3.

Values: true| false (default)

### **ProposeWordLengthsForDefaultFractionLength**

Propose fixed-point types based on DefaultFractionLength.

Values: false (default) | true

### **ProposeTypesUsing**

Propose data types based on simulation range data, derived ranges, or both.

Values: 'BothSimulationAndDerivedRanges' (default) |  
'SimulationRanges'|'DerivedRanges'

### **SafetyMargin**

Safety margin percentage by which to increase the simulation range when proposing fixed-point types. The specified safety margin must be a real number greater than -100.

Values: 0 (default) | double

### **StaticAnalysisQuickMode**

Perform faster static analysis.

Values: true | false (default)

### **StaticAnalysisTimeoutMinutes**

Abort analysis if timeout is reached.

Values: '' (default) | positive integer

### **TestBenchName**

Test bench function name or names, specified as a string or cell array of strings. You must specify at least one test bench.

If you do not explicitly specify input parameter data types, the conversion uses the first test bench function to infer these data types.

Values: '' (default) | string | cell array of strings

### **TestNumerics**

Enable numerics testing.

Values: true| false (default)

## **Methods**

<code>addDesignRangeSpecification</code>	Add design range specification to parameter
<code>addFunctionReplacement</code>	Replace floating-point function with fixed-point function during fixed-point conversion
<code>clearDesignRangeSpecifications</code>	Clear all design range specifications
<code>getDesignRangeSpecification</code>	Get design range specifications for parameter
<code>hasDesignRangeSpecification</code>	Determine whether parameter has design range
<code>removeDesignRangeSpecification</code>	Remove design range specification from parameter

## **Examples**

### **Generate HDL Code from Floating-Point MATLAB Code**

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Create a `coder.HdlConfig` object, `hdlcfg`, with default settings.

```
hdlcfg = coder.config('hdl');
```

Convert your floating-point MATLAB design to fixed-point, and generate HDL code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

## Alternatives

You can also generate HDL code from MATLAB code using the HDL Workflow Advisor. For more information, see “HDL Code Generation and FPGA Synthesis from a MATLAB Algorithm”.

## See Also

`codegen` | `coder.HdlConfig` | `coder.config`

## Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”

## **coder.HdlConfig class**

**Package:** coder

HDL codegen configuration object

### **Description**

A `coder.HdlConfig` object contains the configuration parameters that the HDL codegen function requires to generate HDL code. Use the `-config` option to pass this object to the codegen function.

### **Construction**

`hdlcfg = coder.config('hdl')` creates a `coder.HdlConfig` object for HDL code generation.

### **Properties**

#### **Basic**

#### **AdderSharingMinimumBitwidth**

Minimum bit width for shared adders, specified as a positive integer.

If `ShareAdders` is `true` and `ResourceSharing` is greater than 1, share adders only if adder bit width is greater than or equal to `AdderSharingMinimumBitwidth`.

Values: integer greater than or equal to 2

#### **ClockEdge**

Specify active clock edge.

Values: 'Rising' (default) | 'Falling'



**DistributedPipeliningPriority**

Priority for distributed pipelining algorithm.

<b>DistributedPipeliningPriority Value</b>	<b>Description</b>
NumericalIntegrity (default)	<p>Prioritize numerical integrity when distributing pipeline registers.</p> <p>This option uses a conservative retiming algorithm that does not move registers across a component if the functional equivalence to the original design is unknown.</p>
Performance	<p>Prioritize performance over numerical integrity.</p> <p>Use this option if your design requires a higher clock frequency and the MATLAB behavior does not need to strictly match the generated code behavior.</p> <p>This option uses a more aggressive retiming algorithm that moves registers across a component even if the modified design's functional equivalence to the original design is unknown.</p>

Values: 'NumericalIntegrity' (default) | 'Performance'

**GenerateHDLTestBench**

Generate an HDL test bench, specified as a logical.

Values: false (default) | true

**HDLCodingStandard**

HDL coding standard to follow and check when generating code. Generates a compliance report showing errors, warnings, and messages.

Values: 'None' (default) | 'Industry'

**HDLCodingStandardCustomizations**

HDL coding standard rules and report customizations, specified using HDL coding standard customization. If you want to customize the coding standard rules and report, you must set HDLCodingStandard to 'Industry'.

Value: HDL coding standard customization object

**HDLLintTool**

HDL lint tool script to generate.

Values: 'None' (default) | 'AscentLint' | 'Leda' | 'SpyGlass' | 'Custom'

**HDLLintInit**

HDL lint script initialization name, specified as a character vector.

**HDLLintCmd**

HDL lint script command.

If you set HDLLintTool to Custom, you must use %s as a placeholder for the HDL file name in the generated Tcl script. Specify HDLLintCmd as a character vector using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

**HDLLintTerm**

HDL lint script termination name, specified as a character vector.

**InitializeBlockRAM**

Specify whether to initialize all block RAM to '0' for simulation.

Values: true (default) | false

**InlineConfigurations**

Specify whether to include inline configurations in generated VHDL code.

When true, include VHDL configurations in files that instantiate a component.

When `false`, suppress the generation of configurations and require user-supplied external configurations. Set to `false` if you are creating your own VHDL configuration files.

Values: `true` (default) | `false`

### **LoopOptimization**

Loop optimization in generated code. See “Optimize MATLAB Loops”.

<b>LoopOptimization Value</b>	<b>Description</b>
LoopNone (default)	Do not optimize loops in generated code.
StreamLoops	Stream loops.
UnrollLoops	Unroll Loops.

### **MinimizeClockEnables**

Specify whether to omit generation of clock enable logic.

When `true`, omit generation of clock enable logic wherever possible.

When `false` (default), generate clock enable logic.

### **MultiplierPartitioningThreshold**

Specify maximum input bit width for hardware multipliers. If a multiplier input bit width is greater than this threshold, HDL Coder splits the multiplier into smaller multipliers.

To improve your hardware mapping results, set this threshold to the input bit width of the DSP or multiplier hardware on your target device.

Values: integer greater than or equal to 2

### **MultiplierSharingMinimumBitwidth**

Minimum bit width for shared multipliers, specified as a positive integer.

If `ShareMultipliers` is `true` and `ResourceSharing` is greater than 1, share multipliers only if multiplier bit width is greater than or equal to `MultiplierSharingMinimumBitwidth`.

Values: integer greater than or equal to 2

### **InstantiateFunctions**

Generate instantiable HDL code modules from functions.

---

**Note** If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

---

Values: `false` (default) | `true`

### **PreserveDesignDelays**

Prevent distributed pipelining from moving design delays or allow distributed pipelining to move design delays, specified as a `logical`.

Persistent variables and `dsp.Delay System` objects are design delays.

Values: `false` (default) | `true`

### **ShareAdders**

Share adders, specified as a `logical`.

If `true`, share adders when `ResourceSharing` is greater than 1 and adder bit width is greater than or equal to `AdderSharingMinimumBitwidth`.

Values: `false` (default) | `true`

### **ShareMultipliers**

Share multipliers, specified as a `logical`.

If `true`, share multipliers when `ResourceSharing` is greater than 1, and multiplier bit width is greater than or equal to `MultiplierSharingMinimumBitwidth`.

Values: `true` (default) | `false`

### **SimulateGeneratedCode**

Simulate generated code, specified as a `logical`.

Values: `false` (default) | `true`

**SimulationIterationLimit**

Maximum number of simulation iterations during test bench generation, specified as an integer. This property affects only test bench generation, not simulation during fixed-point conversion.

Values: unlimited (default) | positive integer

**SimulationTool**

Simulation tool name.

Values: 'ModelSim' (default) | 'ISIM'

**SynthesisTool**

Synthesis tool name.

Values: 'Xilinx ISE' (default) | 'Altera Quartus II' | 'Xilinx Vivado'

**SynthesisToolChipFamily**

Synthesis target chip family name, specified as a character vector.

Values: 'Virtex4' (default) | 'Family name'

**SynthesisToolDeviceName**

Synthesis target device name, specified as a character vector.

Values: 'xc4vsx35' (default) | 'Device name'

**SynthesisToolPackageName**

Synthesis target package name, specified as a character vector.

Values: 'ff668' (default) | 'Package name'

**SynthesisToolSpeedValue**

Synthesis target speed, specified as a character vector.

Values: '-10' (default) | 'Speed value'

**SynthesizeGeneratedCode**

Synthesize generated code or not, specified as a `logical`.

Values: `false` (default) | `true`

**TargetLanguage**

Target language of the generated code.

Values: `'VHDL'` (default) | `'Verilog'`

**TestBenchName**

Test bench function name, specified as a character vector. You must specify a test bench.

Values: `' '` (default) | `'Testbench name'`

**TimingControllerArch**

Timing controller architecture.

<b>TimingControllerArch Value</b>	<b>Description</b>
<code>default</code> (default)	Do not generate a reset for the timing controller.
<code>resettable</code>	Generate a reset for the timing controller.

**TimingControllerPostfix**

Postfix to append to design name to form name of timing controller, specified as a character vector.

Values: `'_tc'` (default) | `'Postfix'`

**UseFileIOInTestBench**

Create and use data files for reading and writing test bench input and output data.

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

**UseMatrixTypesInHDL**

Generate 2-D matrix types in HDL code for MATLAB matrices, specified as a `logical`.

<b>UseMatrixTypesInHDL Value</b>	<b>Description</b>
false (default)	Generate HDL vectors with index computation logic for MATLAB matrices. This option can use more area in the synthesized hardware.
true	<p>Generate HDL matrices for MATLAB matrices. This option can save area in the synthesized hardware.</p> <p>The following requirements apply:</p> <ul style="list-style-type: none"> <li>Matrix elements cannot be complex or <code>struct</code> data types.</li> <li>You cannot use linear indexing to specify matrix elements. For example, if you have a 3x3 matrix, <code>A</code>, you cannot use <code>A(4)</code>. Instead, use <code>A(2,1)</code>.</li> </ul> <p>You can also use a colon operator in either the row or column subscript, but not both. For example, you can use <code>A(3,1:3)</code> and <code>A(2:3,1)</code>, but not <code>A(2:3,1:3)</code>.</p> <ul style="list-style-type: none"> <li>If you enable <code>InstantiateFunctions</code>, <code>UseMatrixTypesInHDL</code> has no effect.</li> </ul>

### **VHDLLibraryName**

Target library name for generated VHDL code, specified as a character vector.

Values: 'work' (default) | 'Library name'

### **Cosimulation**

#### **GenerateCosimTestBench**

Generate a cosimulation test bench or not, specified as a `logical`.

Values: false (default) | true

#### **SimulateCosimTestBench**

Simulate generated cosimulation test bench, specified as a `logical`. This option is ignored if `GenerateCosimTestBench` is false.

Values: `false` (default) | `true`

### **CosimClockEnableDelay**

Time (in clock cycles) between deassertion of reset and assertion of clock enable.

Values: 0 (default)

### **CosimClockHighTime**

The number of nanoseconds the clock is high.

Values: 5 (default)

### **CosimClockLowTime**

The number of nanoseconds the clock is low.

Values: 5 (default)

### **CosimHoldTime**

The hold time for input signals and forced reset signals, specified in nanoseconds.

Values: 2 (default)

### **CosimLogOutputs**

Log and plot outputs of the reference design function and HDL simulator.

Values: `false` (default) | `true`

### **CosimResetLength**

Specify time (in clock cycles) between assertion and deassertion of reset.

Values: 2 (default)

### **CosimRunMode**

HDL simulator run mode during simulation. When in Batch mode, you do not see the HDL simulator GUI, and the HDL simulator automatically shuts down after simulation.

Values: `Batch` (default) | `GUI`



**CosimTool**

HDL simulator for the generated cosim test bench.

Values: ModelSim (default) | Incisive

**FPGA-in-the-loop****GenerateFILTestBench**

Generate a FIL test bench or not, specified as a `logical`.

Values: `false` (default) | `true`

**SimulateFILTestBench**

Simulate generated cosimulation test bench, specified as a `logical`. This option is ignored if `GenerateCosimTestBench` is `false`.

Values: `false` (default) | `true`

**FILBoardName**

FPGA board name, specified as a character vector. You must override the default value and specify a valid board name.

Values: 'Choose a board' (default) | 'A board name'

**FILBoardIPAddress**

IP address of the FPGA board, specified as a character vector. You must enter a valid IP address.

Values: 192.168.0.2 (default)

**FILBoardMACAddress**

MAC address of the FPGA board, specified as a character vector. You must enter a valid MAC address.

Values: 00-0A-35-02-21-8A (default)

### **FILAdditionalFiles**

List of additional source files to include, specified as a character vector. Separate file names with a semi-colon (";").

Values: '' (default) | 'Additional source files'

### **FILLogOutputs**

Log and plot outputs of the reference design function and FPGA.

Values: false (default) | true

## **Examples**

### **Generate Verilog Code from MATLAB Code**

Create a `coder.HdlConfig` object, `hdlcfg`.

```
hdlcfg = coder.config('hdl'); % Create a default 'hdl' config
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set the target language to Verilog.

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL code from your MATLAB design. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -config hdlcfg mlhdlc_dti
```

### **Generate Cosim and FIL Test Benches**

Create a `coder.FixptConfig` object with default settings and provide test bench name.

```
fixptcfg = coder.config('fixpt');  
fixptcfg.TestBenchName = 'mlhdlc_sfir_tb';
```

Create a `coder.HdlConfig` object with default settings and set enable rate.

```
hdlcfg = coder.config('hdl'); % Create a default 'hdl' config  
hdlcfg.EnableRate = 'DUTBaseRate';
```

Instruct MATLAB to generate a cosim test bench and a FIL test bench. Specify FPGA board name.

```
hdlcfg.GenerateCosimTestBench = true;  
hdlcfg.FILBoardName = 'Xilinx Virtex-5 XUPV5-LX110T development board';  
hdlcfg.GenerateFILTestBench = true;
```

Perform code generation, Cosim test bench generation, and FIL test bench generation.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_sfir
```

## Alternatives

You can also generate HDL code from MATLAB code using the HDL Workflow Advisor. For more information, see “HDL Code Generation and FPGA Synthesis from a MATLAB Algorithm”.

## See Also

### Functions

`codegen` | `coder.config` | `hdlcoder.CodingStandard`

### Classes

`coder.FixptConfig`

### Properties

HDL Coding Standard Customization

## Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”



# **Shared Class and Function Reference for HDL Code Generation from MATLAB and Simulink**

---

## hdlcoder.CodingStandard

Create HDL coding standard customization object

### Syntax

```
cso = hdlcoder.CodingStandard(standardName)
```

### Description

`cso = hdlcoder.CodingStandard(standardName)` creates an HDL coding standard customization object that you can use to customize the rules and the appearance of the coding standard report.

If you do not want to customize the rules or appearance of the coding standard report, you do not need to create an HDL coding standard customization object.

### Examples

#### Customize coding standard rules for MATLAB to HDL workflow

Create an HDL coding standard customization object, *cso*.

```
cso = hdlcoder.CodingStandard('Industry');
```

Customize the coding standard options as follows:

- Do not show passing rules in the coding standard report.
- Set the maximum if-else nesting depth to 2.
- Disable the check for line length.

```
cso.ShowPassingRules.enable = false;  
cso.IfElseNesting.depth = 2;  
cso.LineLength.enable = false;
```

Create an HDL codegen configuration object.

```
hdlcfg = coder.config('hdl');
```

Specify the coding standard and coding standard customization object.

```
hdlcfg.HDLCodingStandard = 'Industry';
hdlcfg.HDLCodingStandardCustomizations = cso;
```

Specify your test bench function name. In this example, the test bench function is *mlhdlc\_dti\_tb*.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Generate HDL code for the design and check the code according to the customized HDL coding standard rules. In this example, the design function is *mlhdlc\_dti*.

```
codegen -config hdlcfg mlhdlc_dti
```

## Customize coding standard rules for Simulink to HDL workflow

### Create an HDL coding standard customization object

- Load the *sfir\_fixed* model
- Create a coding standard customization object *cso*

```
load_system('sfir_fixed')
cso = hdlcoder.CodingStandard('Industry');
```

### Customize the coding standard options

- Do not show passing rules in the report.
- Set maximum line length to 80 characters.
- Check that module, instance, and entity names are between 5 and 50 characters long.

```
cso.ShowPassingRules.enable = false;
cso.LineLength.length = 80;
cso.ModuleInstanceEntityNameLength.length = [5 50];
```

### Generate HDL code for your design

Generate HDL code and check it according to the customized HDL coding standard rules. The DUT subsystem is *symmetric\_fir*.

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...
        'HDLCodingStandardCustomizations', cso, 'TargetDirectory', 'C:/coding_standard/

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\coding_standard\hdlsrc\sfir_fixed\symmetr
### Industry Compliance report with 4 errors, 77 warnings, 6 messages.
### Generating Industry Compliance Report <a href="matlab:web('C:\coding_standard\hdlsrc\sfir_fixed\symmetric_fir_report.html')">matlab:web('C:\coding_standard\hdlsrc\sfir_fixed\symmetric_fir_report.html')</a>
### Creating HDL Code Generation Check Report file://C:\coding_standard\hdlsrc\sfir_fixed\symmetric_fir_report.html
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

## Input Arguments

### **standardName** — HDL coding standard name

'Industry'

Specify the HDL coding standard to customize. The `standardName` value must match the `HDLCodingStandard` property value.

Example: 'Industry'

## Output Arguments

### **cso** — HDL coding standard customizations

HDL coding standard customization object

HDL coding standard customizations, returned as an HDL coding standard customization object.

## See Also

### **Properties**

HDL Coding Standard Customization

### **Topics**

“Generate an HDL Coding Standard Report from Simulink”



“Generate an HDL Coding Standard Report from MATLAB”  
“Choose Coding Standard and Report Options”  
“HDL Coding Standard Report”

**Introduced in R2014b**

# HDL Coding Standard Customization Properties

Customize HDL coding standard

## Description

HDL coding standard customization properties control how HDL Coder generates and checks code according to a specified coding standard. By changing property values, you can customize the rules and the appearance of the coding standard report.

Use dot notation to refer to a particular object and property:

```
cso = hdlcoder.CodingStandard('Industry');  
len = cso.SignalPortParamNameLength.length;  
cso.ShowPassingRules.enable = false;
```

The generated code follows the customized coding standard rules as much as possible. However, if following a coding standard rule could cause the HDL code to be uncompileable or unsynthesizable, the coder does not follow the rule.

## Properties

### Coding Standard Report

#### **ShowPassingRules** — Show passing rules in coding standard report

structure

Show or do not show passing rules in coding standard report, specified as a structure with the following field.

Field	Description
enable	Set to <code>true</code> to show passing rules in coding standard report.  Set to <code>false</code> to show only rules with errors or warnings.  The default is <code>true</code> .

### Basic Coding Rules

#### HDLKeywords — Check for HDL keywords in design names

structure

Check for HDL keywords in design names (rule CGSL-1.A.A.3), specified as a structure with the following field.

Field	Description
enable	Set to <code>true</code> to check for HDL keywords in design names.  Set to <code>false</code> if you do not want to check for HDL keywords in design names.  The default is <code>true</code> .

#### DetectDuplicateNamesCheck — Check for duplicate names

structure

Check for duplicate names in the design (rule CGSL-1.A.A.5), specified as a structure with the following field.

Field	Description
enable	Set to <code>true</code> to check for duplicate names in the design.  Set to <code>false</code> if you do not want to check for duplicate names in the design.  The default is <code>true</code> .

**ModuleInstanceEntityNameLength — Check module, instance, and entity name length**

structure

Check for module, instance, and entity name lengths (rule CGSL-1.A.B.1), specified as a structure with the following fields.

Field	Description
enable	<p>Set to <code>true</code> to check the length of module, instance, and entity names.</p> <p>Set to <code>false</code> if you do not want to check the length of module, instance, and entity names.</p> <p>The default is <code>true</code>.</p>
length	<p>Minimum and maximum length of module, instance, and entity name names, specified as a 2-element array of positive integers.</p> <p>The first element is the minimum length, and the second element is the maximum length. The default is <code>[2 32]</code>.</p>

**SignalPortParamNameLength — Check signal, port, and parameter name length**

structure

Check for signal, port, and parameter name lengths (rule CGSL-1.A.C.3), specified as a structure with the following fields.

Field	Description
enable	<p>Set to <code>true</code> to check the length of signal, port, and parameter names.</p> <p>Set to <code>false</code> if you do not want to check the length of signal, port, and parameter names.</p> <p>The default is <code>true</code>.</p>

Field	Description
length	<p>Minimum and maximum length of signal, port, and parameter names, specified as a 2-element array of positive integers.</p> <p>The first element is the minimum length, and the second element is the maximum length. The default is [2 40].</p>

### RTL Description Rules

#### MinimizeClockEnableCheck — Check for clock enable signals

structure

Check for clock enable signals in the generated code (rule CGSL-2.C.C.4), specified as a structure with the following field.

Field	Description
enable	<p>Set to <code>true</code> to minimize clock enables in the generated code and check for clock enable signals after code generation.</p> <p>Set to <code>false</code> if you do not want to check for clock enable signals in the generated code.</p> <p>The default is <code>false</code>.</p>

#### RemoveResetCheck — Check for reset signals

structure

Check for reset signals in the design (rule CGSL-2.C.C.5), specified as a structure with the following field.

Field	Description
enable	<p>Set to <code>true</code> to minimize reset signals in the generated code and check for reset signals after code generation.</p> <p>Set to <code>false</code> if you do not want to check for reset signals in the design.</p> <p>The default is <code>false</code>.</p>

**AsynchronousResetCheck – Check for asynchronous reset signals in the generated code**

structure

Check for asynchronous reset signals in the generated code (CGSL-2.C.C.6), specified as a structure with the following field.

Field	Description
enable	<p>Set to <code>true</code> to check for asynchronous reset signals in the generated code.</p> <p>Set to <code>false</code> if you do not want to check for asynchronous reset signals in the generated code.</p> <p>The default is <code>true</code>.</p>

**MinimizeVariableUsage – Minimize use of variables**

structure

Minimize use of variables (rule CGSL-2.G), specified as a structure with the following field.

Field	Description
enable	<p>Set to <code>true</code> to minimize use of variables.</p> <p>Set to <code>false</code> if you do not want to minimize use of variables.</p> <p>The default is <code>false</code>.</p>

### ConditionalRegionCheck – Check for length of conditional statements in a process or always block

structure

Check for length of conditional statements (if-else, case, and loops) which are described separately in a process block or an always block (rule CGSL-2.F.B.1), specified as a structure with the following fields.

Field	Description
enable	<p>Set to <code>true</code> to check length of conditional statements.</p> <p>Set to <code>false</code> if you do not want to check the length of conditional statements.</p> <p>The default is <code>true</code>.</p>
length	<p>Number of conditional statements which are described separately within a process block (VHDL) or an always block (Verilog).</p> <p>The default is 1.</p>

### IfElseNesting – Check if-else statement nesting depth

structure

Check for if-else statement nesting depth (rule CGSL-2.G.C.1a), specified as a structure with the following fields.

Field	Description
enable	<p>Set to <code>true</code> to check if-else statement nesting depth.</p> <p>Set to <code>false</code> if you do not want to check if-else statement nesting depth.</p> <p>The default is <code>true</code>.</p>

Field	Description
depth	Maximum if-else statement nesting depth, specified as a positive integer.  The default is 3.

**IfElseChain – Check if-else statement chain length**

structure

Check for if-else statement chain length (rule CGSL-2.G.C.1c), specified as a structure with the following fields.

Field	Description
enable	Set to <code>true</code> to check if-else statement chain length.  Set to <code>false</code> if you do not want to check if-else statement chain length.  The default is <code>true</code> .
length	Maximum length of if-else statement chain, specified as a positive integer.  The default is 7.

**MultiplierBitWidth – Check multiplier bit width**

structure

Check for multiplier bit width (rule CGSL-2.J.F.5), specified as a structure with the following fields.

Field	Description
enable	Set to <code>true</code> to check multiplier bit width.  Set to <code>false</code> if you do not want to check multiplier bit width.  The default is <code>true</code> .



Field	Description
width	Maximum multiplier bit width, specified as a positive integer.  The default is 16.

**RTL Design Rules****LineLength — Check generated code line length**

structure

Check for generated code line length (rule CGSL-3.A.D.5), specified as a structure with the following fields.

Field	Description
enable	Set to <code>true</code> to check line lengths in generated code.  Set to <code>false</code> if you do not want to check line lengths in generated code.  The default is <code>true</code> .
length	Maximum number of characters per line in generated code, specified as a positive integer.  The default is 110.

**NonIntegerTypes — Check for non-integer constants**

structure

Check for non-integer constants (rule CGSL-3.B.D.1), specified as a structure with the following field.

<b>Field</b>	<b>Description</b>
enable	Set to <code>true</code> to check for non-integer constants.  Set to <code>false</code> if you do not want to check for non-integer constants.  The default is <code>true</code> .

## See Also

`hdlcoder.CodingStandard`

## Topics

“Generate an HDL Coding Standard Report from MATLAB”

“Generate an HDL Coding Standard Report from Simulink”

“HDL Coding Standard Report”

“Basic Coding Practices”

“RTL Description Techniques”

“RTL Design Methodology Guidelines”

# hdl.BlackBox

**Package:** hdl

Black box for including custom HDL code

## Description

`hdl.BlackBox` provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`B = hdl.BlackBox` creates a black box System object for HDL code generation.

## Properties

**AddClockEnablePort** — Add clock enable port

'on' (default) | 'off'

If 'on', add a clock enable input port to the interface generated for the black box System object. The name of the port is specified by `ClockEnableInputPort`.

### **AddClockPort — Add clock port**

'on' (default) | 'off'

If 'on', add a clock input port to the interface generated for the black box System object. The name of the port is specified by `ClockInputPort`.

### **AddResetPort — Add reset port**

'on' (default) | 'off'

If 'on', add a reset input port to the interface generated for the black box System object. The name of the port is specified by `ResetInputPort`.

### **AllowDistributedPipelining — Register placement for distributed pipelining**

'off' (default) | 'on'

If 'on', allow HDL Coder to move registers across the black box System object, from input to output or output to input.

### **ClockEnableInputPort — Clock enable input port name**

'clk\_enable' (default) | character vector

HDL name for clock enable input port, specified as a character vector.

### **ClockInputPort — Clock input port name**

'clk' (default) | character vector

HDL name for clock input port, specified as a character vector.

### **EntityName — Module or entity name**

System object instance name (default) | character vector

VHDL entity or Verilog module name generated for the black box System object, specified as a character vector.

Example: 'myBlackBoxName'

### **ImplementationLatency — Latency in clock cycles**

-1 (default) | integer

Latency of black box System object in clock cycles, specified as an integer.

If 0 or greater, this value is used for delay balancing.

If -1, latency is unknown. This disables delay balancing.

### **InlineConfigurations — Generate VHDL configuration**

InlineConfigurations global property value (default) | 'on' | 'off'

When 'on', generate a VHDL configuration.

When 'off', do not generate a VHDL configuration and require a user-supplied external configuration. Set to 'off' if you are creating your own VHDL configuration.

### **InputPipeline — Input pipeline stages**

0 (default) | positive integer

Number of input pipeline stages, or pipeline depth, to insert in the generated code.

### **OutputPipeline — Output pipeline stages**

0 (default) | positive integer

Number of output pipeline stages, or output pipeline depth, to insert in the generated code.

### **ResetInputPort — Reset port name**

'reset' (default) | character vector

HDL name for reset input port, specified as a character vector.

### **VHDLArchitectureName — VHDL architecture name**

'rtl' (default) | character vector

VHDL architecture name, specified as a character vector. The coder generates the architecture name only if InlineConfigurations is 'on'.

### **VHDLComponentLibrary — VHDL component library name**

'work' (default) | character vector

Library from which to load the VHDL component, specified as a character vector.

### **NumInputs — Number of custom input ports**

1 (default) | positive integer

Number of additional input ports in the custom HDL code, specified as a positive integer.

**NumOutputs — Number of custom output ports**

1 (default) | positive integer

Number of additional output ports in the custom HDL code, specified as a positive integer.

**See Also**

`coder.HdlConfig`

**Topics**

“Integrate Custom HDL Code Into MATLAB Design”

“Generate Board-Independent IP Core from MATLAB Algorithm”

“Generate Black Box Interface for Subsystem”

**Introduced in R2015a**

# hdl.RAM

**Package:** hdl

Single, simple dual, or dual-port RAM for memory read/write access

## Description

hdl.RAM reads from and writes to memory locations for a single, simple dual, or dual-port RAM. The output data is delayed one step. If your input data is scalar, the address and write enable inputs must be scalar, and HDL Coder infers a single RAM block. If your data is a vector, HDL Coder infers an array of parallel RAM banks. With vector data input, the address and write enable inputs can be both scalars or vectors. When you specify scalar inputs for the write enable and address ports, the system object applies the same operation to each RAM bank.

The hdl.RAM System object can have  $2^{31}$  bytes of internal storage. The RAM size takes into account the address width, the number of bytes that are used to store each word, and the number of RAM banks.

To read from or write to memory locations in the RAM:

- 1 Create the hdl.RAM object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects? \(MATLAB\)](#).

## Creation

### Syntax

```
ram = hdl.RAM  
ram = hdl.RAM(Name, Value)
```

## Description

`ram =hdl.RAM` returns a single port RAM System object that you can write to or read from a memory location.

`ram =hdl.RAM(Name,Value)` returns a single, simple dual, or dual port RAM System object with properties set using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

### RAMType — Type of RAM

'Single port' (default) | 'Simple dual port' | 'Dual port'

Type of RAM, specified as either:

- 'Single port' — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- 'Simple dual port' — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- 'Dual port' — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

### WriteOutputValue — Behavior for Write output

'New data' (default) | 'Old data'

Behavior for Write output, specified as either:



- 'New data' — Send out new data at the address to the output.
- 'Old data' — Send out old data at the address to the output.

### Dependencies

Specify this property when you set **RamType** to 'Single port' or 'Dual port'. This property does not apply for Simple Dual Port RAM object.

### RAMInitialValue — Initial output of RAM

'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

## Usage

---

**Note** For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Syntax

```
dataOut = ram(wrData, rwAddress, wrEn)
```

```
rdDataOut = ram(wrData, rwAddress, wrEn, rdAddress)
```

```
[wrDataOut, rdDataOut] = ram(wrData, rwAddress, wrEn, rdAddress)
```

## Description

`dataOut = ram(wrData, rwAddress, wrEn)` reads the value in memory location `rwAddress` when `wrEn` is false. When `wrEn` is true, you write the value `wrData` into the memory location `rwAddress`. `dataOut` is the new or old data at `rwAddress`. Use this syntax when you create a single port RAM System object.

`rdDataOut = ram(wrData,wrAddress,wrEn,rdAddress)` writes the value `wrData` into memory location `wrAddress` when `wrEn` is true. `rdDataOut` is the old data at the address location `rdAddress`. Use this syntax when you create a simple dual port RAM System object.

`[wrDataOut,rdDataOut] = ram(wrData,wrAddress,wrEn,rdAddress)` writes the value `wrData` into the memory location `wrAddress` when `wrEn` is true. `wrDataOut` is the new or old data at memory location `wrAddress`. `rdDataOut` is the old data at the address location `rdAddress`. Use this syntax when you create a dual port RAM System object.

## Input Arguments

### **wrData — Write data**

Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be double, single, integer, or a fixed-point (`fi`) object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fi`

### **rwAddress — Write or Read address**

Scalar (default) | Vector

Address that you write the `wrData` into when `wrEn` is true. The System object reads the value in memory location `rwAddress` when `wrEn` is false. This value can be either fixed-point (`fi`) or integer, and must be real and unsigned. Specify this address when you create a single port RAM object.

Data Types: `uint8` | `uint16` | `fi`

### **wrEn — Write enable**

Scalar (default) | Vector

When `wrEn` is true, you write the `wrData` into the RAM memory location. If you create a single port RAM, the System object reads the value in the memory location when `wrEn` is false. This value must be logical.

Data Types: `logical`

### **rdAddress — Read address**

Scalar (default) | Vector

Address that you read the data from when you create a simple dual port RAM or dual port RAM System object. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Data Types: `uint8` | `uint16` | `fi`

#### **wrAddress — Write address**

Scalar (default) | Vector

Address that you write the data into when you create a simple dual port RAM or dual port RAM System object. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Data Types: `uint8` | `uint16` | `fi`

## **Output Arguments**

#### **dataOut — Output data**

Scalar (default) | Vector

Output data that the System object reads from the memory location `rwAddress` a single port RAM object when `wrEn` is false.

#### **rdDataOut — Data from Read address**

Scalar (default) | Vector

Old output data that the System object reads from the memory location `rdAddress` of a simple dual port RAM or dual port RAM System object.

#### **wrDataOut — Data from Write address**

Scalar (default) | Vector

New or old output data that the System object reads from the memory location `wrAddress` of a simple dual port RAM or dual port RAM System object.

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## Examples

### Observe Previous Data at Write Time

Construct System object to read from or write to a memory location in RAM. Set `WriteOutputValue` to `Old data` to return the previous value stored at the write address.

The output data port corresponds to the read/write address passed in. During a write operation, the old data at the write address is sent out as the output.

**Note:** This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_1p = hdl.RAM('RAMType','Single port',...  
                'WriteOutputValue','Old data')
```

```
ram_1p =  
    hdl.RAM with properties:  
        RAMType: 'Single port'  
    WriteOutputValue: 'Old data'  
    RAMInitialValue: 0
```

```
dataLength = 10;  
dataIn = 1:10;  
dataOut = zeros(1,dataLength);
```

Write a count pattern to the memory. Previous values on the first writes are all zero.

```
for ii = 1:dataLength  
    addressIn = uint8(ii-1);
```

```

        writeEnable = true;
        dataOut(ii) = ram_lp(dataIn(ii),addressIn,writeEnable);
    end
dataOut

dataOut = 1×10

     0     0     0     0     0     0     0     0     0     0

```

Read the data back.

```

    for ii = 1:dataLength
        addressIn = uint8(ii-1);
        writeEnable = false;
        dataOut(ii) = ram_lp(dataIn(ii),addressIn,writeEnable);
    end
dataOut

dataOut = 1×10

     0     1     2     3     4     5     6     7     8     9

```

Now, write the count in reverse order. The previous values are the original count.

```

    for ii = 1:dataLength
        addressIn = uint8(ii-1);
        writeEnable = true;
        dataOut(ii) = ram_lp(dataIn(dataLength-ii+1),addressIn,writeEnable);
    end
dataOut

dataOut = 1×10

    10     1     2     3     4     5     6     7     8     9

```

### Read/Write Single-Port RAM

Create System object that writes to a single port RAM and reads the newly written value.

**Note:** This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Construct single-port RAM System object. When you write a location, the object returns the new value. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

```
ram_1p = hdl.RAM('RAMType','Single port','WriteOutputValue','New data');
dataLength = 16;
[dataIn,dataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and then read data back out again.

```
for ii = 1:dataLength
    dataIn(ii) = randi([0 63],1,1,'uint8');
    addressIn = fi((ii-1),0,4,0);
    writeEnable = true;
    dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
```

```
dataOut
```

```
dataOut = 1x16 uint8 row vector
```

```
    0    52    57     8    58    40     6    17    35    61    61    10    62    61    31    51
```

```
for ii = 1:dataLength
    addressIn = fi((ii-1),0,4,0);
    writeEnable = false;
    dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
```

```
dataOut
```

```
dataOut = 1x16 uint8 row vector
```

```
     9    52    57     8    58    40     6    17    35    61    61    10    62    61    31    51
```

### Create Simple Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

The output data port corresponds to the read address. If a read operation is performed at the same address as the write operation, old data at that address is read out as the output. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

**Note:** This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_2p = hdl.RAM('RAMType','Simple dual port');
dataLength = 16;
[dataIn,dataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and read the old data from the same address.

```
for ii = 1:dataLength
    dataIn(ii) = randi([0 63],1,1,'uint8');
    wrAddr = fi((ii-1),0,4,0);
    writeEnable = true;
    ataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);
end
dataOut
```

*dataOut = 1x16 uint8 row vector*

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Write and read from different addresses. The object returns the read result after one cycle delay.

```
for ii = 1:dataLength
    wrAddr = fi((ii-1),0,4,0);
    rdAddr = fi(dataLength-ii+1,0,4,0);
    writeEnable = true;
    dataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
dataOut
```

*dataOut = 1x16 uint8 row vector*

0 9 9 51 31 61 62 10 61 61 35 17 6 40 58 8

### Create Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

There are two output ports: a write output data port and a read output data port. The write output data port sends out the new data at the write address. The read output data port sends out the old data at the read address. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

**Note:** This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');  
dataLength = 16;  
[dataIn,wrDataOut,rdDataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and read the old data from the same address.

```
for ii = 1:dataLength  
    dataIn(ii) = randi([0 63],1,1,'uint8');  
    wrAddr = fi((ii-1),0,4,0);  
    writeEnable = true;  
    [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);  
end  
wrDataOut
```

```
wrDataOut = 1x16 uint8 row vector
```

```
    0    52    57     8    58    40     6    17    35    61    61    10    62    61    31    51
```

```
rdDataOut
```

```
rdDataOut = 1x16 uint8 row vector
```

```
    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

Write and read from different addresses. The object returns the read result after one cycle delay.



```

for ii = 1:dataLength
    wrAddr = fi((ii-1),0,4,0);
    rdAddr = fi(dataLength-ii+1,0,4,0);
    writeEnable = true;
    [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
wrDataOut

wrDataOut = 1x16 uint8 row vector

    9  52  57  8  58  40  6  17  35  61  61  10  62  61  31  51

rdDataOut

rdDataOut = 1x16 uint8 row vector

    0  9  9  51  31  61  62  10  61  61  35  17  6  40  58  8

```

### Create Dual-Port RAM with Multiple Banks

Create a System object that can write vector data to a dual-port RAM and read vector data out. Each element of the vector corresponds to a separate bank of RAM. This example creates 4 16-bit banks. Each bank has eight entries.

**Note:** This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Construct dual-port RAM System object.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
```

Create vector write data and addresses. Use a 3-bit address (for 8 locations), and write 16-bit data. Read and write addresses are independent. Allocate memory for the output data.

```
ramDataIn = fi(randi((2^16)-1,1,4),0,16,0);
ramReadAddr = fi([1,1,1,1],0,3,0);
ramWriteAddr = fi([1,1,1,1],0,3,0);
[wrOut,rdOut] = deal(fi(zeros(1,4),0,16,0));
```

First, write locations in bank 1 and 4, then read all banks. The write data is echoed in the `wrOut` output argument. The object returns read results after one cycle delay.

```
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[true,false,false,true],ramReadAddr);  
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr);  
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr)
```

```
wrOut =  
    53393         0         0    59859  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness:  Unsigned  
    WordLength:   16  
    FractionLength: 0  
  
rdOut =  
    53393         0         0    59859  
  
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness:  Unsigned  
    WordLength:   16  
    FractionLength: 0
```

## Algorithms

In your Simulink model, you can use the `hdl.RAM` inside a MATLAB System or a MATLAB Function block. If you log the output of a MATLAB System block, the output data has at least three dimensions because the MATLAB System block has at least two dimensions, and the time data adds a third dimension. For example, if you input scalar data to the block, the logged output data has the dimension  $1 \times 1 \times N$ , where  $N$  is the number of time steps. To obtain an output dimension that is same as the input dimension, add a Reshape block at the output with **Output dimensionality** set to `Derive` from reference input port.

## RAM Inference with Scalar Data

If your data is scalar, the RAM size, or number of locations, is inferred from the data type of the address variable.

Data type of address variable	RAM address size (bits)
single or double	16
uint <i>N</i>	<i>N</i>
embedded.fi	WordLength

The maximum RAM address size is 32 bits.

## RAM Inference with Vector Data

If your data is a vector, HDL Coder generates an array of parallel RAM banks. The number of elements in the vector determines the number of RAM banks. The size of each RAM bank is inferred from the data type of the address variable.

Data type of address variable	RAM address size (bits)
single or double	16
uint <i>N</i>	<i>N</i>
embedded.fi	WordLength

The maximum RAM bank address size is 32 bits.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

## See Also

### Blocks

Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM | Single Port RAM

### Topics

""

""

"Implement RAM Using MATLAB Code"

"HDL Code Generation for System Objects"

**Introduced in R2015a**

# coder.hdl.loopspec

Unroll or stream loops in generated HDL code

## Syntax

```
coder.hdl.loopspec('unroll')  
coder.hdl.loopspec('unroll',unroll_factor)  
coder.hdl.loopspec('stream')  
coder.hdl.loopspec('stream',stream_factor)
```

## Description

`coder.hdl.loopspec('unroll')` fully unrolls a loop in the generated HDL code. Instead of a loop statement, the generated code contains multiple instances of the loop body, with one loop body instance per loop iteration.

The `coder.hdl.loopspec` pragma does not affect MATLAB simulation behavior.

---

**Note** If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

---

`coder.hdl.loopspec('unroll',unroll_factor)` unrolls a loop by the specified unrolling factor, `unroll_factor`, in the generated HDL code.

The generated HDL code is a loop statement that contains `unroll_factor` instances of the original loop body. The number of loop iterations in the generated code is  $(original\_loop\_iterations / unroll\_factor)$ . If  $(original\_loop\_iterations / unroll\_factor)$  has a remainder, the remaining iterations are fully unrolled as loop body instances outside the loop.

This pragma does not affect MATLAB simulation behavior.

---

**Note** If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

---

`coder.hdl.loopspec('stream')` generates a single instance of the loop body in the HDL code. Instead of using a loop statement, the generated code implements local oversampling and added logic to match the functionality of the original loop.

You can specify this pragma for loops at the top level of your MATLAB design.

This pragma does not affect MATLAB simulation behavior.

---

**Note** If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

---

`coder.hdl.loopspec('stream',stream_factor)` unrolls the loop with `unroll_factor` set to *original\_loop\_iterations / stream\_factor* rounded down to the nearest integer, and also oversamples the loop. If (*original\_loop\_iterations / stream\_factor*) has a remainder, the remainder loop body instances outside the loop are not oversampled, and run at the original rate.

You can specify this pragma for loops at the top level of your MATLAB design.

This pragma does not affect MATLAB simulation behavior.

---

**Note** If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

---

## Examples

### Completely unroll MATLAB loop in generated HDL code

Unroll loop in generated code.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('unroll');
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
```

```

    end
end

```

### Partially unroll MATLAB loop in generated HDL code

Generate a loop statement in the HDL code that has two iterations and contains five instances of the original loop body.

```

function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('unroll', 5);
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end

```

### Completely stream MATLAB loop in generated HDL code

In the generated code, implement the 10-iteration MATLAB loop as a single instance of the original loop body that is oversampled by a factor of 10.

```

function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('stream');
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end

```

### Partially stream MATLAB loop in generated HDL code

In the generated code, implement the 10-iteration MATLAB loop as five instances of the original loop body that are oversampled by a factor of 2.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('stream', 2);
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end
```

## Input Arguments

### **stream\_factor** — Loop streaming factor

positive integer

Loop streaming factor, specified as a positive integer.

Setting `stream_factor` to the number of original loop iterations is equivalent to fully streaming the loop, or using `coder.hdl.loopspec('stream')`.

Example: 4

### **unroll\_factor** — Loop unrolling factor

positive integer

Number of loop body instances, specified as a positive integer.

Setting `unroll_factor` to the number of original loop iterations is equivalent to fully unrolling the loop, or using `coder.hdl.loopspec('unroll')`.

Example: 10

## See Also

### Topics

“Optimize MATLAB Loops”

**Introduced in R2015a**



# coder.hdl.pipeline

Insert pipeline registers at output of MATLAB expression

## Syntax

```
out = coder.hdl.pipeline(expr)
out = coder.hdl.pipeline(expr,num)
```

## Description

`out = coder.hdl.pipeline(expr)` inserts one pipeline register at the output of `expr` in the generated HDL code. This pragma does not affect MATLAB simulation behavior.

Use this pragma to specify exactly where to insert pipeline registers. For example, in a MATLAB assignment statement, you can specify the `coder.hdl.pipeline` pragma:

- On the entire right side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see “Port Registers”.

`out = coder.hdl.pipeline(expr,num)` inserts `num` pipeline registers at the output of `expr` in the generated HDL code. This pragma does not affect MATLAB simulation behavior.

Use this pragma to specify exactly where to insert pipeline registers. For example, in a MATLAB assignment statement, you can specify the `coder.hdl.pipeline` pragma:

- On the entire right side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;  
t = u + pvar;  
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see “Port Registers”.

## Examples

### Insert one pipeline register at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert a single pipeline register.

```
y = coder.hdl.pipeline(a + b * c);
```

### Insert multiple pipeline registers at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert three pipeline registers.

```
y = coder.hdl.pipeline(a + b * c, 3);
```

### Insert pipeline registers at intermediate stage of MATLAB expression

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers.

```
y = a + coder.hdl.pipeline(b * c, 5);
```

### **Insert pipeline registers at intermediate stage and at output of MATLAB expression**

At an intermediate stage and at the output of a MATLAB expression, use nested `coder.hdl.pipeline` pragmas to insert pipeline registers.

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers, and insert two pipeline registers at the output of the whole expression.

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5), 2);
```

## **Input Arguments**

### **expr — MATLAB expression to pipeline**

MATLAB expression

MATLAB expression to pipeline. At the output of this expression in the generated HDL code, insert pipeline registers.

Example: `a + b`

### **num — Number of registers**

MATLAB expression

Number of pipeline registers to insert at the output of `expr` in the generated HDL code, specified as a positive integer.

Example: `3`

## **See Also**

### **Topics**

“Pipeline MATLAB Expressions”

“Pipelining MATLAB Code”

**Introduced in R2015a**

# hdlcoder.Board class

**Package:** hdlcoder

Board registration object that describes SoC custom board

## Description

`board = hdlcoder.Board` creates a board object that you use to register a custom board for an SoC platform.

To specify the characteristics of your board, set the properties of the board object.

## Construction

`board = hdlcoder.Board` creates an `hdlcoder.Board` object that you can use to register a custom board for an SoC platform.

## Properties

### BoardName — Board name

' ' (default) | character vector

Board name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Target platform** dropdown list.

Example: 'Enclustra Mars ZX3 with PM3 base board'

### FPGAVendor — Vendor name

' ' (default) | 'Altera' | 'Xilinx'

FPGA vendor name, specified as a character vector.

Example: 'Xilinx'

### FPGAFamily — FPGA family name

' ' (default) | character vector

FPGA family name, specified as a character vector.

Example: 'Zynq'

### **FPGADevice — FPGA device identifier**

' ' (default) | character vector

FPGA device identifier, specified as a character vector.

Example: 'xc7z020'

### **FPGAPackage — FPGA package identifier for Xilinx devices**

' ' (default) | character vector

FPGA package identifier for Xilinx devices, specified as a character vector.

For Altera devices, this property is ignored.

Example: 'clg484'

### **FPGASpeed — FPGA speed for Xilinx devices**

' ' (default) | character vector

FPGA speed for Xilinx devices, specified as a character vector.

For Altera devices, this property is ignored.

Example: '-1'

### **SupportedTool — Supported synthesis tool**

' ' (default) | cell array of character vectors

Synthesis tool or tools that support this board, specified as a cell array of character vectors. In the HDL Workflow Advisor, the **Synthesis tool** dropdown list shows the values in this cell array.

Example: {'Altera Quartus II'}

Example: {'Xilinx Vivado'}

Example: {'Xilinx Vivado', 'Xilinx ISE'}

### **JTAGChainPosition — Optional JTAG chain position number**

2 (default) | positive integer

JTAG chain position number, specified as a positive integer. The JTAG chain position number is used when programming the FPGA via JTAG.

This property is optional.

Example: 3

## Methods

<code>addExternalIOInterface</code>	Define external IO interface for board object
<code>addExternalPortInterface</code>	Define external port interface for board object
<code>validateBoard</code>	Check property values in board object

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2015a**

## addExternalIOInterface

**Class:** hdlcoder.Board

**Package:** hdlcoder

Define external IO interface for board object

### Syntax

```
addExternalIOInterface('InterfaceID',interfacename,'InterfaceType',  
interfacetype,'PortName',portname,'PortWidth',portwidth,'FPGAPin',  
pins,'IOPadConstraint',constraints)
```

### Description

`addExternalIOInterface('InterfaceID',interfacename,'InterfaceType',interfacetype,'PortName',portname,'PortWidth',portwidth,'FPGAPin',pins,'IOPadConstraint',constraints)` adds an external IO interface to an `hdlcoder.Board` object. You can add multiple external IO interfaces to your board object.

Use this method if your board has more than one external interface, or if you want to be able to predefine FPGA pin names for mapping from the HDL Workflow Advisor.

### Input Arguments

**interfacename** — Interface name

character vector

Interface name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Target Platform Interfaces** dropdown list.

Example: 'LEDs General Purpose'

**interfacetype** — Interface direction

'IN' | 'OUT'



Interface direction, specified as a character vector. In the HDL Workflow Advisor, when you specify a target interface for each of your DUT ports, this external IO interface is available only for ports with a matching direction.

For example, if you set `interfacetype` to `'OUT'`, this external IO interface is available only for **Output** DUT ports.

Example: `'OUT'`

**portname — Port name**

character vector

Board top-level port name, specified as a character vector.

Example: `'GPLEDs'`

**portwidth — Port bit width**

positive integer

Port bit width, specified as a positive integer.

Example: `4`

**pins — Pin names**

cell array of character vectors

FPGA pin names, specified as a cell array of character vectors.

Example: `{'H18', 'AA14', 'AA13', 'AB15'}`

**constraints — IO pad constraints**

`{}` (default) | cell array of character vectors

IO pad constraints, specified as a cell array of character vectors.

Example: `{'IOSTANDARD = LVCMOS25'}`

Example: `{'IOSTANDARD = LVCMOS25', 'SLEW = SLOW'}`

## Tips

- For details about the external IO interface ports, pins, and constraints for your board, view the board documentation.

## See Also

`hdlcoder.Board` | `hdlcoder.Board.addExternalPortInterface`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Board and Reference Design Registration System”

**Introduced in R2015a**

# addExternalPortInterface

**Class:** hdlcoder.Board

**Package:** hdlcoder

Define external port interface for board object

## Syntax

```
addExternalPortInterface('IOPadConstraint', constraints)
```

## Description

`addExternalPortInterface('IOPadConstraint', constraints)` adds a generic external port interface to an `hdlcoder.Board` object. You can add at most one external port interface to your board object.

Use this method if you want the `External Port` option to be available in the HDL Workflow Advisor **Target Platform Interface** dropdown list. If you use this method to add an external port, in the HDL Workflow Advisor, you must manually specify pin names in the **Bit Range / Address / FPGA Pin** field.

## Input Arguments

**constraints — IO pad constraints**

{ } (default) | cell array of character vectors

IO pad constraints, specified as a cell array of character vectors.

Example: {'IOSTANDARD = LVCMOS25'}

Example: {'IOSTANDARD = LVCMOS25', 'SLEW = SLOW'}

### Tips

- To get IO constraint names for your board, view the board documentation.

### Alternatives

If you know the details of the external interface, and want to make them available as UI dropdown list options in the HDL Workflow advisor, use the `hdlcoder.Board.addExternalIOInterface` method instead. For example, using `hdlcoder.Board.addExternalIOInterface`, you can predefine characteristics of the interface such as the name, port bit width, signal direction, and valid pin names.

### See Also

`hdlcoder.Board` | `hdlcoder.Board.addExternalIOInterface`

### Topics

Define and Register Custom Board and Reference Design for SoC Workflow  
“Register a Custom Board”  
“Board and Reference Design Registration System”

**Introduced in R2015a**

# validateBoard

**Class:** hdlcoder.Board

**Package:** hdlcoder

Check property values in board object

## Syntax

```
validateBoard
```

## Description

validateBoard checks that the `hdlcoder.Board` object has nondefault values for all required properties, and that property values have valid data types. This method does not check the correctness of property values for the target board. If validation fails, the software displays an error message.

## See Also

`hdlcoder.Board`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Board and Reference Design Registration System”

**Introduced in R2015a**

## hdlcoder.ReferenceDesign class

**Package:** hdlcoder

Reference design registration object that describes SoC reference design

### Description

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool', toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

To specify the characteristics of your reference design, set the properties of the reference design object.

Use a reference design tool version that is compatible with the supported tool version. If you choose a different tool version, it is possible that HDL Coder is unable to create the reference design project for IP core integration.

### Construction

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool', toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

### Input Arguments

**toolname** — Synthesis tool name

Xilinx Vivado (default) | Altera Quartus II | Xilinx ISE | Xilinx Vivado

Synthesis tool name, specified as a character vector.

Example: 'Altera Quartus II'

## Properties

### ReferenceDesignName — Reference design name

' ' (default) | character vector

Reference design name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Reference design** dropdown list.

Example: 'Default system (Vivado 2015.4)'

### BoardName — Board name

' ' (default) | character vector

Board associated with this reference design, specified as a character vector.

Example: 'Enclustra Mars ZX3 with PM3 base board'

### SupportedToolVersion — Supported tool version

{ } (default) | cell array of character vectors

One or more tool versions that work with this reference design, specified as a cell array of character vectors.

Example: { '2015.4' }

Example: { '13.7', '14.0' }

### CustomConstraints — Design constraint file (optional)

{ } (default) | cell array of character vectors

One or more design constraint files, specified as a cell array of character vectors. This property is optional.

Example: { 'MarsZX3\_PM3.xdc' }

Example: { 'MyDesign.qsf' }

### CustomFiles — Relative path to required file or folder (optional)

{ } (default) | cell array of character vectors

One or more relative paths to files or folders that the reference design requires, specified as a cell array of character vectors. This property is optional.

Examples of required files or folders:

- Existing IP core used in the reference design.

For example, if the IP core, *my\_ip\_core*, is in the reference design folder, set `CustomFiles` to `{'my_ip_core'}`

- PS7 definition XML file.

For example, to include a PS7 definition XML file, *ps7\_system\_prj.xml*, in a folder, *data*, set `CustomFiles` to `{fullfile('data', 'ps7_system_prj.xml')}`

- Folder containing existing IP cores used in the reference design. HDL Coder only supports a specific IP core folder name for each synthesis tool:
  - For Altera Qsys, IP core files must be in a folder named `ip`. Set `CustomFiles` to `{'ip'}`.
  - For Xilinx Vivado, IP core files, or a zip file containing the IP core files, must be in a folder named `ipcore`. Set `CustomFiles` to `{'ipcore'}`.
  - For Xilinx EDK, IP core files must be in a folder named `pcores`. Set `CustomFiles` to `{'pcores'}`.

---

**Note** To add IP modules to the reference design, it is recommended to create an IP repository folder that contains these IP modules, and then use the `addIPRepository` on page 8-76 method.

---

Example: `{'my_ip_core'}`

Example: `{fullfile('data', 'ps7_system_prj.xml')}`

Example: `{'ip'}`

Example: `{'ipcore'}`

Example: `{'pcores'}`

### **IPCacheZipFile** — IP cache file to include in the project

`''` (default) | `'ipcache.zip'` | character vector

Specify the IP cache zip file to include in your project. When you run the IP Core Generation workflow in the HDL Workflow Advisor, the code generator extracts this file in the **Create Project** task. The **Build FPGA Bitstream** task reuses the IP cache, which accelerates reference design synthesis.



This property is optional.

Example: 'ipcache.zip'

**ReportTimingFailure — Report timing failures as warnings or errors**

'hdlcoder.ReportTiming.Warning' (default) | 'hdlcoder.ReportTiming.Error'

Specify whether you want the code generator to report timing failures in the **Build FPGA Bitstream** task as warnings or errors. When you run the **IP Core Generation** workflow in the HDL Workflow Advisor, by default, the code generator reports any timing failures as error. If you have implemented the custom logic to resolve timing failures, you can specify these failures to be reported as warning instead of error. To learn more, see “Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows”.

This property is optional.

Example: 'hdlcoder.ReportTiming.Warning'

## Methods

<code>addAXI4MasterInterface</code>	Add and define AXI4 Master interface
<code>addAXI4SlaveInterface</code>	Add and define AXI4 slave interface
<code>addInternalIOInterface</code>	Add and define internal IO interface between generated IP core and existing IP cores
<code>addClockInterface</code>	Add clock and reset interface
<code>addCustomEDKDesign</code>	Specify Xilinx EDK MHS project file
<code>addCustomQsysDesign</code>	Specify Altera Qsys project file
<code>addCustomVivadoDesign</code>	Specify Xilinx Vivado exported block design Tcl file
<code>addIPRepository</code>	Include IP modules from your IP repository folder in your custom reference design
<code>addParameter</code>	Add and define custom parameters for your reference design
<code>CallbackCustomProgrammingMethod</code>	Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor
<code>EmbeddedCoderSupportPackage</code>	Specify whether to use an Embedded Coder support package
<code>PostBuildBitstreamFcn</code>	Function handle for callback function that gets executed after Build FPGA Bitstream task in the HDL Workflow Advisor
<code>PostCreateProjectFcn</code>	Function handle for callback function that gets executed after Create Project task in the HDL Workflow Advisor
<code>PostSWInterfaceFcn</code>	Function handle for custom callback function that gets executed after Generate Software Interface Model task in the HDL Workflow Advisor
<code>PostTargetInterfaceFcn</code>	Function handle for callback function that gets executed after Set Target Interface task in the HDL Workflow Advisor
<code>PostTargetReferenceDesignFcn</code>	Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor
<code>validateReferenceDesign</code>	Check property values in reference design object

## See Also

hdlcoder.Board

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2015a**

## addAXI4MasterInterface

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Add and define AXI4 Master interface

### Syntax

```
addAXI4MasterInterface('InterfaceID',  
Interface_ID,'InterfaceConnection',Interface_Connection)  
addAXI4MasterInterface('InterfaceID',  
Interface_ID,'InterfaceConnection',  
Interface_Connection,'TargetAddressSegments',  
Target_Address_Segments)  
addAXI4MasterInterface('InterfaceID',  
Interface_ID,'InterfaceConnection',Interface_Connection, Name,Value)  
addAXI4MasterInterface('InterfaceID',  
Interface_ID,'InterfaceConnection',  
Interface_Connection,'TargetAddressSegments',  
Target_Address_Segments, Name,Value)
```

### Description

`addAXI4MasterInterface('InterfaceID', Interface_ID, 'InterfaceConnection', Interface_Connection)` adds and defines an AXI4 Master interface for an Intel Qsys reference design.

`addAXI4MasterInterface('InterfaceID', Interface_ID, 'InterfaceConnection', Interface_Connection, 'TargetAddressSegments', Target_Address_Segments)` adds and defines an AXI4 Master interface for a Xilinx Vivado reference design.

`addAXI4MasterInterface('InterfaceID', Interface_ID, 'InterfaceConnection', Interface_Connection, Name, Value)`

adds and defines an AXI4 Master interface for an Intel Qsys reference design, with additional options specified by one or more `Name, Value` pair arguments.

`addAXI4MasterInterface('InterfaceID', Interface_ID, 'InterfaceConnection', Interface_Connection, 'TargetAddressSegments', Target_Address_Segments, Name, Value)` adds and defines an AXI4 Master interface for a Xilinx Vivado reference design, with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **Interface\_ID — AXI4 Master interface name**

'AXI4 Master' (default) | character vector

Name of the AXI4 Master interface that you add to the reference design, specified as a character vector. If you create multiple AXI4 Master interfaces, make sure that you use unique names for each interface.

Example: 'AXI4 Master 1'

### **Interface\_Connection — Reference design port name**

' ' (default) | character vector

Name of the reference design port that is connected to the AXI4 Master interface, specified as a character vector.

Example: 'axi\_interconnect\_1/S01\_AXI'

### **Target\_Address\_Segments — Reference design address segments**

' ' (default) | character vector

Target address segment of the Xilinx Vivado reference design, specified as a character vector.

Example: '{{'mig\_7series\_0/memmap/memaddr', hex2dec('40000000'), hex2dec('40000000')}}'

## 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 quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example:

### **ReadSupport — AXI4 Master read interface support**

`'true'` (default) | `'false'`

Specify whether you want the AXI4 Master interface to support a read channel as a Boolean.

Example: `'ReadSupport', 'true'` specifies support for an AXI4 Master read interface connection.

### **WriteSupport — AXI4 Master write interface support**

`'true'` (default) | `'false'`

Specify whether you want the AXI4 Master interface to support a write channel as a Boolean.

Example: `'WriteSupport', 'true'` specifies support for an AXI4 Master write interface connection.

### **MaxDataWidth — Maximum data width**

128 (default) | Integer

Maximum width for the `Data` signal that is transferred across the AXI4 Master interface, specified as an integer.

Example: `'MaxDataWidth', 32` specifies maximum data width of 32 bits.

### **AddrWidth — Address width**

32 (default) | Integer

Width of the AXI4 Master interface read and write addresses, specified as an integer.

Example: `'AddrWidth', 32` specifies an address size of 32 bits.

### **DefaultReadBaseAddr — Starting read address**

0 (default) | Integer

Default starting address of the AXI4 Master read interface, specified as an integer.

Example: `'DefaultReadBaseAddr', hex2dec('40000000')` specifies `hex2dec('40000000')` as the starting read address.

**DefaultWriteBaseAddr — Starting write address**

0 (default) | Integer

Default starting address of the AXI4 Master write interface, specified as an integer.

Example: 'DefaultReadBaseAddr', `hex2dec('41000000')` specifies `hex2dec('41000000')` as the starting write address.

**See Also**

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addClockInterface`

**Topics**

“Model Design for AXI4 Master Interface Generation”

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2017b**

## addAXI4SlaveInterface

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Add and define AXI4 slave interface

### Syntax

```
addAXI4SlaveInterface('InterfaceConnection',  
    ref_design_port,'BaseAddress',base_addr)  
addAXI4SlaveInterface('InterfaceConnection',  
    ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace',  
    master_addr_space)  
addAXI4SlaveInterface('InterfaceConnection',  
    ref_design_port,'BaseAddress',base_addr,Name,Value)  
addAXI4SlaveInterface('InterfaceConnection',  
    ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace',  
    master_addr_space,Name,Value)
```

### Description

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port, 'BaseAddress', base_addr)` adds and defines an AXI4 interface for an Altera reference design, or an AXI4 or AXI4-Lite interface for a Xilinx ISE reference design.

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port, 'BaseAddress', base_addr, 'MasterAddressSpace', master_addr_space)` adds and defines an AXI4 or AXI4-Lite interface for Xilinx Vivado reference designs.

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port, 'BaseAddress', base_addr, Name, Value)` adds and defines an AXI4 interface for an Altera reference design, or an AXI4 or AXI4-Lite interface for a Xilinx ISE reference design, with additional options specified by one or more `Name, Value` pair arguments.



`addAXI4SlaveInterface('InterfaceConnection', ref_design_port, 'BaseAddress', base_addr, 'MasterAddressSpace', master_addr_space, Name, Value)` adds and defines an AXI4 or AXI4-Lite interface for Xilinx Vivado reference designs, with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ref\_design\_port** — Reference design port name

' ' (default) | character vector

Reference design port that is connected to the AXI4 or AXI4-Lite interface, specified as a character vector. For reference designs based on Intel Qsys, when you want to connect multiple AXI Master IPs to the AXI4 or AXI4-Lite interface, specify each of the AXI Master instance names and the corresponding port names as a cell array of character vectors.

Example: `'axi_interconnect_0/M00_AXI', {'hps_0.h2f_axi_master', 'master_0.master'}, ...`

### **base\_addr** — Base address

' ' (default) | character vector

Base address for AXI4 or AXI4-Lite slave interface, specified as a character vector.

Example: `'0x40010000'`

### **master\_addr\_space** — Master interface address space (Vivado only)

' ' (default) | character vector

Address space of the master interface connected to this slave interface, specified as a character vector. For Vivado reference designs only. When you want to connect more than one AXI Master IP, specify each of the AXI Master instance names and the corresponding address spaces.

Example: `'processing_system7_0/Data', {'processing_system7_0/Data', 'hdlverifier_axi_master_0/axi4m'}`

## 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 quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **InterfaceType — Interface type**

`{'AXI4-Lite', 'AXI4'} (default) | 'AXI4' | 'AXI4-Lite'`

Type of interface connection, specified as a character vector or a cell array of character vectors.

Example: `'InterfaceType', 'AXI4-Lite'` specifies an `'AXI4-Lite'` interface type connection.

### **InterfaceID — Interface name**

`{'AXI4-Lite', 'AXI4'} (default) | character vector`

Name of the interface, specified as a character vector. When you provide the `InterfaceID`, `InterfaceType` must be set to either `'AXI4'` or `'AXI4-Lite'`.

Example: `'InterfaceID', 'MyAXI4', 'InterfaceType', 'AXI4'` specifies interface name as `'MyAXI4'` and interface type as `'AXI4'`.

### **IDWidth — Width of ID signals**

`12 (default) | positive integer`

Width of all ID signals, such as `AWID`, `WID`, `ARID`, and `RID`, specified as a positive integer. This property enables you to specify the number of AXI Master interfaces that you want the AXI4 slave interface in the HDL DUT IP core to connect to. The default value is 12, which enables you to connect the HDL IP core to one AXI Master interface. To connect the IP core to multiple AXI Master interfaces, increase the `IDWidth`. The ID width is tool-specific.

Example: `'IDWidth', '13'` may indicate that you want the IP core to connect to two AXI Master interfaces in the reference design.

## **Tips**

- Before running this method, you must run the `hdlcoder.ReferenceDesign.addClockInterface` method.
- The `addAXI4SlaveInterface` method is optional. You can define your own custom reference design without the AXI4 slave interface.

- To connect the HDL IP core for your DUT to multiple AXI Master interfaces in the reference design, use the `IDWidth` property of this method. To learn more, see “Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface”.

## See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addClockInterface`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface”

“Board and Reference Design Registration System”

## Introduced in R2015a

## addInternalIOInterface

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Add and define internal IO interface between generated IP core and existing IP cores

### Syntax

```
addInternalIOInterface('InterfaceID',interface_name,'InterfaceType',  
interface_type,'PortName',port_name,'PortWidth',  
port_width,'InterfaceConnection',interface_connection)
```

### Description

`addInternalIOInterface('InterfaceID',interface_name,'InterfaceType',interface_type,'PortName',port_name,'PortWidth',port_width,'InterfaceConnection',interface_connection)` adds and defines an internal IO interface between the generated IP core and other IP cores in the reference design.

In the HDL Workflow Advisor, if you target a custom reference design that has an internal IO interface, you must map a DUT port to the internal IO interface. In the Target Platform Interface Table, you cannot leave the internal IO interface unassigned.

### Input Arguments

**interface\_name** — Custom internal IO interface name

' ' (default) | character vector

Custom internal IO interface name, specified as a character vector. In the HDL Workflow Advisor, when you select the custom reference design, this name appears as an option in the Target Platform Interface Table.

Example: 'MyCustomInternalInterface'

**interface\_type — Interface direction**

'IN' (default) | 'OUT'

Interface direction relative to the generated IP core, specified as a character vector.

For example, if the interface is an input to the generated IP core, set `interface_type` to 'IN'.

**port\_name — Port name**

' ' (default) | character vector

Name of generated IP core port in the HDL code, specified as a character vector.

Example: 'MyIPCoreInternalIOInterfacePort'

**port\_width — Port bit width**

8 (default) | integer

Bit width of generated IP core port, specified as an integer.

**interface\_connection — Internal IO interface connection**

' ' (default) | character vector

Internal IO interface port to connect with generated IP core port, specified as a character vector. The internal IO interface port is an existing port in the reference design. Its port bit width must match `port_width`.

Different synthesis tools have different formats for the internal IO interface port.

Synthesis Tool	Format Example
Altera Quartus II	'internal_ip_0.In0'
Xilinx Vivado	'internal_ip_0/In0'
Xilinx ISE	'internal_In0'

Example: 'internal\_ip\_0.In0'

Example: 'internal\_ip\_0/In0'

Example: 'internal\_In0'

## See Also

`hdlcoder.ReferenceDesign`

## Topics

“”

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2015b**

# addClockInterface

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Add clock and reset interface

## Syntax

```
addClockInterface('ClockConnection',clock_port,'ResetConnection',  
reset_port)  
addClockInterface('ClockConnection',clock_port,'ResetConnection',  
reset_port,Name,Value)
```

## Description

`addClockInterface('ClockConnection',clock_port,'ResetConnection',reset_port)` adds a clock and reset interface to an `hdlcoder.ReferenceDesign` object.

`addClockInterface('ClockConnection',clock_port,'ResetConnection',reset_port,Name,Value)` adds a clock and reset interface to the `hdlcoder.ReferenceDesign` object with additional options specified by one or more `Name,Value` pair arguments. When you specify these arguments, in the HDL Workflow Advisor, HDL Coder adds a **Set Target Frequency** task. To modify the output clock frequency setting in the reference design clock wizard, in this task, specify the **Target Frequency (MHz)**.

## Input Arguments

**clock\_port** — Clock port name

' ' (default) | character vector

Reference design port that is connected to the IP core clock port, specified as a character vector.

Example: 'processing\_system7\_1/FCLK\_CLK0'

### **reset\_port** — Reset port name

' ' (default) | character vector

Reference design port that is connected to the IP core reset port, specified as a character vector.

Example: 'proc\_sys\_reset/peripheral\_aresetn'

## 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 quotes. You can specify several name and value pair arguments in any order as **Name1, Value1, ..., NameN, ValueN**.

### **DefaultFrequencyMHz** — The default frequency in MHz

0 (default) | integer

The default clock frequency in MHz of the clock module IP in the reference design, specified as an integer. When you open the HDL Workflow Advisor, HDL Coder populates this information for **Default (MHz)** in the **Set Target Frequency** task.

Example: 'DefaultFrequencyMHz', 50 specifies the default frequency as 50 MHz.

### **MinFrequencyMHz** — The minimum frequency in MHz

0 (default) | integer

The minimum clock frequency in MHz of the clock module IP in the reference design, specified as an integer.

Example: 'MinFrequencyMHz', 5 specifies the minimum clock frequency as 5 MHz.

### **MaxFrequencyMHz** — The maximum frequency in MHz

0 (default) | integer

The maximum clock frequency in MHz of the clock module IP in the reference design, specified as an integer. In the HDL Workflow Advisor, HDL Coder sets the **Frequency Range (MHz)** in the **Set Target Frequency** task based on the **MinFrequencyMHz** and **MaxFrequencyMHz** values that you specify.



Example: 'MaxFrequencyMHz', 500 specifies the maximum clock frequency as 500 MHz.

### **ClockNumber — Clock output port number**

1 (default) | integer

Port number of the clock output from the clock module IP in the reference design, specified as an integer.

Example: 'ClockNumber', 2 specifies to use the second output port in the clock module IP as the clock port.

### **ClockModuleInstance — Clock module name**

'clk\_wiz\_0' (default) | character vector

The name of the clock module IP in the reference design, specified as a character vector.

Example: 'ClockModuleInstance', 'clk\_wiz\_1' specifies clk\_wiz\_1 as the name of the clock module IP.

## **See Also**

hdlcoder.ReferenceDesign |  
hdlcoder.ReferenceDesign.addAXI4SlaveInterface

## **Topics**

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

## **Introduced in R2015a**

## addCustomEDKDesign

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Specify Xilinx EDK MHS project file

### Syntax

```
addCustomEDKDesign('CustomEDKMHS',mhs_project_file)
```

### Description

`addCustomEDKDesign('CustomEDKMHS',mhs_project_file)` specifies the MHS project file that contains the Xilinx EDK embedded system design. Use this method if your synthesis tool is Xilinx ISE.

### Input Arguments

**mhs\_project\_file** — MHS project file

character vector

MHS project file for Xilinx EDK embedded system design, specified as a character vector.

Example: 'system.mhs'

### Tips

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Altera Quartus II, use the `addCustomQsysDesign` method.

## See Also

hdlcoder.ReferenceDesign |  
hdlcoder.ReferenceDesign.addCustomQsysDesign |  
hdlcoder.ReferenceDesign.addCustomVivadoDesign

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow  
“Register a Custom Board”  
“Register a Custom Reference Design”  
“Board and Reference Design Registration System”

**Introduced in R2015a**

## addCustomQsysDesign

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Specify Altera Qsys project file

### Syntax

```
addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file)
```

### Description

`addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file)` specifies the Qsys project file that contains the Altera Qsys embedded system design. Use this method if your synthesis tool is Altera Quartus II.

### Input Arguments

**qsys\_project\_file** — Qsys project file

character vector

Qsys project file for Altera Qsys embedded system design, specified as a character vector.

Example: 'system\_soc.qsys'

### Tips

- If you have more than one AXI Master IP, in the custom qsys project file, you must make sure that the AXI Master IPs connect to the same AXI Interconnect IP. The AXI4 slave interfaces in the HDL IP core also connect to this Interconnect.

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Xilinx ISE, use the `addCustomEDKDesign` method.

## See Also

`hdlcoder.ReferenceDesign.addCustomEDKDesign`  
`hdlcoder.ReferenceDesign.addCustomVivadoDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2015a**

## addCustomVivadoDesign

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Specify Xilinx Vivado exported block design Tcl file

### Syntax

```
addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)
```

### Description

`addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)` specifies the exported block design Tcl file that contains the Xilinx Vivado embedded system design. Use this method if your synthesis tool is Xilinx Vivado.

### Input Arguments

**bd\_tcl\_file** — Block design Tcl file

character vector

Block design Tcl file that you exported from your Xilinx Vivado embedded system design project, specified as a character vector. The Tcl file name must be the same as the Vivado block diagram name.

Example: 'system\_top.tcl'

### Tips

- If you have more than one AXI Master IP, in the custom block design Tcl file, you must make sure that the AXI Master IPs connect to the same AXI Interconnect IP. The AXI4 slave interfaces in the HDL IP core also connect to this Interconnect.

- If your synthesis tool is Xilinx ISE, use the `hdlcoder.ReferenceDesign.addCustomEDKDesign` method.
- If your synthesis tool is Altera Quartus II, use the `hdlcoder.ReferenceDesign.addCustomQsysDesign` method.

## See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addCustomEDKDesign`  
| `hdlcoder.ReferenceDesign.addCustomQsysDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow  
“Register a Custom Board”  
“Register a Custom Reference Design”  
“Board and Reference Design Registration System”

**Introduced in R2015a**

## addIPRepository

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Include IP modules from your IP repository folder in your custom reference design

### Syntax

```
addIPRepository('IPListFunction',IP_list_function)
addIPRepository('IPListFunction',IP_list_function,Name,Value)
```

### Description

`addIPRepository('IPListFunction',IP_list_function)` adds IP modules that are in the IP repository folder to your reference design project.

`addIPRepository('IPListFunction',IP_list_function,Name,Value)` adds IP modules that are in the IP repository folder to your reference design project with additional options specified by one or more `Name,Value` pair arguments.

Before you use this method, define the IP list function that points to the IP modules in the repository folder. To learn more, see “Define and Add IP Repository to Custom Reference Design”.

### Input Arguments

**IP\_list\_function** — Name and path to the function that points to the IP repository

' ' (default) | character vector

Name and path to the function that points to IP modules in the IP repository folder to add to the reference design project, specified as a character vector.

Example: 'adi.hdmi.vivado.hdlcoder\_video\_iplist'

Example: 'mathworks.hdlcoder.vivado.hdlcoder\_video\_iplist'



## 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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **NotExistMessage** — Error message to display if IP function is not found

`''` (default) | character vector

Error message that you create to be displayed if IP list function is not found on the MATLAB path, specified as a character vector.

Example: `'IP repository cannot be found'`

## See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Define and Add IP Repository to Custom Reference Design”

“Board and Reference Design Registration System”

“Register a Custom Board”

“Register a Custom Reference Design”

### **Introduced in R2017a**

## addParameter

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Add and define custom parameters for your reference design

### Syntax

```
addParameter('ParameterID',parameter_name,'DisplayName',  
display_name,'DefaultValue',default_value)  
addParameter('ParameterID',parameter_name,'DisplayName',  
display_name,'DefaultValue',default_value,Name,Value)
```

### Description

`addParameter('ParameterID',parameter_name,'DisplayName',display_name,'DefaultValue',default_value)` adds and defines a custom parameter for your reference design with a text box that displays the default value of the parameter.

`addParameter('ParameterID',parameter_name,'DisplayName',display_name,'DefaultValue',default_value,Name,Value)` adds and defines a custom parameter for your reference design with additional options specified by one or more `Name,Value` pair arguments.

The custom parameters are optional. In the HDL Workflow Advisor **Set Target Reference Design** task, HDL Coder populates the Reference design parameters section with the custom parameters and the options that you specify.

### Input Arguments

**parameter\_name** — Custom parameter name

' ' (default) | character vector

Custom parameter name, specified as a character vector.

Example: 'DUTPath'

Example: 'ChannelMapping'

### **display\_name — Custom parameter display name**

' ' (default) | character vector

Name that you want to display for the custom parameter in the HDL Workflow Advisor, specified as a character vector. This name appears in the **Reference design parameters** section in the **Set Target Reference Design** task.

Example: 'DUT Path'

Example: 'Channel Mapping'

### **default\_value — Custom parameter default value**

' ' (default) | character vector

Default value to set for the custom parameter, specified as a character vector. In the **Set Target Reference Design** task in the HDL Workflow Advisor, HDL Coder displays the default value of the custom parameter inside a text box.

Example: '1'

## **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 quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **ParameterType — Parameter widget**

`hdlcoder.ParameterType.Edit` (default) | `hdlcoder.ParameterType.DropDown` | `hdlcoder.ParameterType.Edit`

Specify the widget type to use for the parameter values. By default, the `ParameterType` is a text box. If you specify the drop-down list for `ParameterType`, use the `Choice` property to list the parameter values as a cell array of character vectors.

Example: 'ParameterType', `hdlcoder.ParameterType.DropDown` specifies a drop-down list with the values that the parameter can take.

### **Choice — Choice of parameter values**

' ' (default) | cell array of character vectors

The list of choices that you can specify for the custom parameter, specified as a cell array of character vectors. To specify this list, set `ParameterType` to `hdlcoder.ParameterType.Dropdown`.

Example: `'ParameterType',hdlcoder.ParameterType.Dropdown,'Choice', {'Rx','Tx'}` specifies a drop-down list with Rx and Tx as the drop-down values.

## **See Also**

`hdlcoder.ReferenceDesign`

## **Topics**

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

## **Introduced in R2016b**

# CallbackCustomProgrammingMethod

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor

## Syntax

CallbackCustomProgrammingMethod

## Description

CallbackCustomProgrammingMethod registers a function handle for the callback function that gets executed when running the **Program Target Device** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, then use this syntax to register the function handle:

```
hRD.CallbackCustomProgrammingMethod = @my_reference_design.callback_CustomProgrammingMethod
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is callback\_PostBuildBitstream, located in the reference design package folder, +my\_reference\_design.

With this callback function, you can specify a custom programming method to program the target device. This example code shows how to create the callback function.

```
function [status, log] = callback_CustomProgrammingMethod(infoStruct)
% Reference design callback function for custom programming method
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.BitstreamPath: the path to the generated FPGA bitstream file
```

```
% infoStruct.ToolProjectFolder: the path to synthesis tool project folder
% infoStruct.ToolProjectName: the synthesis tool project name
% infoStruct.ToolCommandString: the command for running a tcl file
%
% status: process run status
%     status == true means process run successfully
%     status == false means process run failed
% log:    output log string
status = true;
log = sprintf('Run custom programming method callback...\n');

% Enter your commands for custom programming here
% ...
% ...

end
```

In the HDL Workflow Advisor, HDL Coder selects the custom programming method to program the target SoC device. If you do not specify the custom programming method, HDL Coder provides JTAG and Download as the options to program the target device.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

## Introduced in R2016a

# EmbeddedCoderSupportPackage

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Specify whether to use an Embedded Coder support package

## Syntax

EmbeddedCoderSupportPackage

## Description

EmbeddedCoderSupportPackage specifies if you want to use an Embedded Coder support package for your reference design. Use this parameter if you are targeting a standalone FPGA board or an SoC device such as the Xilinx Zynq<sup>®</sup>-7000 platform.

If you are targeting a standalone FPGA board, the reference designs do not require an Embedded Coder support package. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, then use this syntax:

```
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.None;
```

When you are not using the support package, HDL Coder does not have the **Generate Software Interface Model** task in the HDL Workflow Advisor.

If you are targeting SoC devices, use this syntax depending on whether you are using an Altera SoC or a Xilinx Zynq platform.

```
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.Zynq;  
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.AlteraSoC;
```

## See Also

hdlcoder.ReferenceDesign

## **Topics**

“”

“IP Core Generation Workflow for Standalone FPGA Devices”

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2016b**



# PostBuildBitstreamFcn

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for callback function that gets executed after Build FPGA Bitstream task in the HDL Workflow Advisor

## Syntax

PostBuildBitstreamFcn

## Description

PostBuildBitstreamFcn registers a function handle for the callback function that gets called at the end of the **Build FPGA Bitstream** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, then use this syntax to register the function handle:

```
hRD.PostBuildBitstreamFcn = @my_reference_design.callback_PostBuildBitstream;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostBuildBitstream`, located in the reference design package folder `+my_reference_design`.

With this callback function, you can specify custom settings when HDL Coder runs the build process and generates the bitstream. This example code shows how to create the callback function. The function displays the status after running the task, and the board and reference design information.

```
function [status, log] = callback_PostBuildBitstream(infoStruct)
% Reference design callback run at the end of the task Build FPGA Bitstream
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
```

```
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.BitstreamPath: the path to generated FPGA bitstream file
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string

status = false;
log = sprintf('Run post build bitstream callback\n%s\n%s\n', infoStruct.HDLModelDutPath, infoStruct.BitstreamPath);

% Exporting the InfoStruct Contents
% ...
% ...

end
```

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2016b**

# PostCreateProjectFcn

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for callback function that gets executed after Create Project task in the HDL Workflow Advisor

## Syntax

PostCreateProjectFcn

## Description

PostCreateProjectFcn registers a function handle for the callback function that gets called at the end of the **Create Project** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax to register the function handle.

```
hRD.PostCreateProjectFcn = @my_reference_design.callback_PostCreateProject;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostCreateProject`, and is located in the reference design package folder `+my_reference_design`.

With this callback function, you can specify custom settings for reference design project creation. This example code shows how to create the callback function. The function exports the contents of the board and reference design object to a `PostCreateProjectInfo.txt` file, and validates that the project creation task ran successfully.

```
function [status, log] = callback_PostCreateProject(infoStruct)
% Reference design callback run at the end of the task Create Project
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
```

```
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ToolProjectFolder: the path to synthesis tool project folder
% infoStruct.ToolProjectName: the synthesis tool project name
%
% status: process run status
%     status == true means process run successfully
%     status == false means process run failed
% log:     output log string

status = false;
log = sprintf('Run post create project callback\n%s', evalc('infoStruct'));

% Exporting the InfoStruct Contents
% ...
% ...

end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Create Project** task, it executes the callback function at the end of the task.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the reference design project creation.

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

## Introduced in R2016b

# PostSWInterfaceFcn

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for custom callback function that gets executed after Generate Software Interface Model task in the HDL Workflow Advisor

## Syntax

PostSWInterfaceFcn

## Description

PostSWInterfaceFcn registers a function handle for the callback function that gets executed at the end of the **Generate Software Interface Model** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, use this syntax to register the function handle.

```
hRD.PostSWInterfaceFcn = @my_reference_design.callback_PostSWInterface;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostSWInterface`, and is located in the reference design package folder `+my_reference_design`.

With this callback function, you can change the generated software interface model for the custom reference design.

This example code shows how to create the callback function. The function adds a DocBlock in the software interface model.

```
function [status, log] = callback_PostSWInterface(infoStruct)
% Reference design callback run at the end of the task
% Generate Software Interface Model
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
```

```
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.SWModelDutPath: the block path to the SW interface subsystem
%
% feature controlled by IPCoreSoftwareInterfaceLibrary
% infoStruct.SWLibBlockPath: the block path to the SW interface library block
% infoStruct.SWLibFolderPath: the folder path to the SW interface library
%
% status: process run status
%     status == true means process run successfully
%     status == false means process run failed
% log:     output log string

status = true;
log = '';
swDutPath = infoStruct.SWModelDutPath;
add_block(['simulink/Model-Wide', char(10), 'Utilities/DocBlock'], sprintf('%s/DocBlock

end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Generate Software Interface Model** task, it executes the callback function at the end of the task.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for software interface model generation.

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2016b**

## PostTargetInterfaceFcn

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for callback function that gets executed after Set Target Interface task in the HDL Workflow Advisor

## Syntax

PostTargetInterfaceFcn

## Description

PostTargetInterfaceFcn registers a function handle for the callback function that gets called at the end of the **Set Target Interface** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, then use this syntax to register the function handle.

```
hRD.PostTargetInterfaceFcn = @my_reference_design.callback_PostTargetInterface;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is callback\_PostTargetInterface, and is located in the reference design package folder +my\_reference\_design.

With this callback function, you can enable custom validations. This example code shows how to create the callback function. If the custom parameter DUTPath is set to Rx, the function validates that the reference design does not support the LEDs General Purpose [0:7] interface.

```
function callback_PostTargetInterface(infoStruct)
% Reference design callback run at the end of the task Set Target Interface
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
```



```

% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ProcessorFPGASynchronization: Processor/FPGA synchronization mode
% infoStruct.InterfaceStructCell: target interface table information
%
%                               a cell array of structure, for example:
%                               infoStruct.InterfaceStructCell{1}.PortName
%                               infoStruct.InterfaceStructCell{1}.PortType
%                               infoStruct.InterfaceStructCell{1}.DataType
%                               infoStruct.InterfaceStructCell{1}.IOInterface
%                               infoStruct.InterfaceStructCell{1}.IOInterfaceMapping

hRD = infoStruct.ReferenceDesignObject;
refDesignName = hRD.ReferenceDesignName;

% validate that when specific parameter is set to specific value, reference
% design does not support specific interface
paramStruct = infoStruct.ParameterStruct;
interfaceStructCell = infoStruct.InterfaceStructCell;
for ii = 1:length(interfaceStructCell)
    interfaceStruct = interfaceStructCell{ii};
    if strcmp(paramStruct.DutPath, 'Rx') && ...
        strcmp(interfaceStruct.IOInterface, 'LEDs General Purpose [0:7]')
        error('LEDs General Purpose [0:7] must not be used when the DUT path is Rx');
    end
end
end

```

In the HDL Workflow Advisor, when HDL Coder runs the **Set Target Interface** task, it executes the callback function at the end of the task. If you specify Rx as the **DUT Path** and use the LEDs General Purpose [0:7] interface for your DUT port, the coder generates an error.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to enable custom validations on the DUT in your Simulink model.

## See Also

`hdlcoder.ReferenceDesign`

## **Topics**

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2016b**

# PostTargetReferenceDesignFcn

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor

## Syntax

PostTargetReferenceDesignFcn

## Description

PostTargetReferenceDesignFcn registers a function handle for the callback function that gets called at the end of the **Set Target Reference Design** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, use this syntax to register the function handle:

```
hRD.PostTargetReferenceDesignFcn = @my_reference_design.callback_PostTargetReferenceDesign
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is callback\_PostTargetReferenceDesign, and is located in the reference design package folder +my\_reference\_design.

With the callback function, you can enable custom validations for your design. This example code shows how to create the callback function and validate that the reset type is synchronous.

```
function callback_PostTargetReferenceDesign(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in st
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
```

```
mdlName = bdroot(infoStruct.HDLModelDutPath);
hRD = infoStruct.ReferenceDesignObject;
refDesignName = hRD.ReferenceDesignName;

isResetSync = strcmpi(hdlget_param(mdlName, 'ResetType'), 'Synchronous');

% Reset must be synchronous
if ~isResetSync
    error('Invalid Reset type. Reset type must be synchronous');
end
end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Set Target Reference Design** task, it executes the callback function. If the reset type is not synchronous, the coder generates an error.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to enable custom validations on the DUT in your Simulink model.

## See Also

`hdlcoder.ReferenceDesign`

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Define Custom Parameters and Callback Functions for Custom Reference Design”

“Board and Reference Design Registration System”

## Introduced in R2016b

# validateReferenceDesign

**Class:** hdlcoder.ReferenceDesign

**Package:** hdlcoder

Check property values in reference design object

## Syntax

```
validateReferenceDesign
```

## Description

validateReferenceDesign checks that the hdlcoder.ReferenceDesign object has nondefault values for all required properties, and that property values have valid data types. This method does not check the correctness of property values for the target board. If validation fails, the software displays an error message.

## See Also

hdlcoder.ReferenceDesign

## Topics

Define and Register Custom Board and Reference Design for SoC Workflow

“Register a Custom Board”

“Register a Custom Reference Design”

“Board and Reference Design Registration System”

**Introduced in R2015a**

