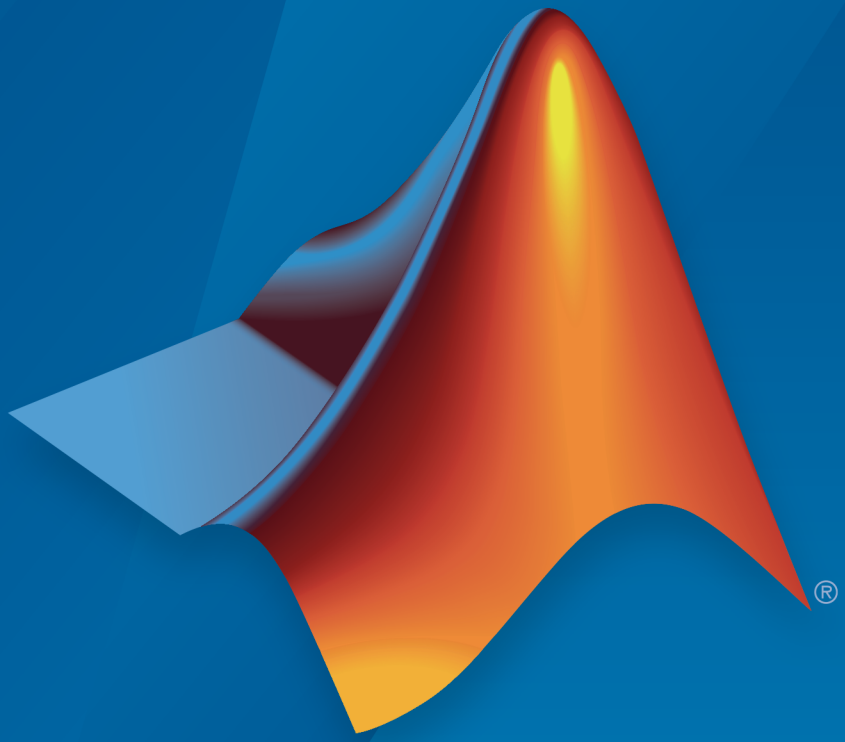
HDL Coder™

Reference



MATLAB® & SIMULINK®

R2015b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

HDL Coder™ Reference

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)

Contents

7

8 System object Reference

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.

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`

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

"What Is the HDL Workflow Advisor?" | "Using the HDL Workflow Advisor Window"

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

string

Model name, specified as a string.

Example: `'sfir_fixed'`

`optimizationCfg` — Optimization configuration

`hdlcoder.OptimizationConfig`

Optimization configuration, specified as an `hdlcoder.OptimizationConfig` object.

`cpGuidanceFile` — File containing saved optimization data

`''` (default) | string

File that contains saved data from the final optimization iteration, including relative path, specified as a string. 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'`

More About

- "Automatic Iterative Optimization"

See Also

Classes

hdlcoder.OptimizationConfig

Functions

`hdlcoder.supportedDevices`

Properties

`SynthesisTool` | `SynthesisToolChipFamily` | `SynthesisToolDeviceName` | `SynthesisToolPackageName` | `SynthesisToolSpeedValue`

hdlcoder.supportedDevices

Show supported target hardware and device details

Syntax

```
hdlcoder.supportedDevices
```

Description

`hdlcoder.supportedDevices` shows a link to a report that contains 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
```

More About

- "Synthesis Tool Path Setup"

See Also

`SynthesisToolChipFamily` | `SynthesisToolDeviceName` | `SynthesisToolPackageName` | `SynthesisToolSpeedValue`

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 pass in the string `'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(gcb,'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"

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 the string `'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.

```
hdldispmdlparams(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.

```
 hdldispmdlparams(bdroot,'all')

%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL CodeGen Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%

AddPipelineRegisters                  : 'off'
Backannotation                        : 'on'
BlockGenerateLabel                    : '_gen'
CheckHDL                              : 'off'
ClockEnableInputPort                  : 'clk_enable'
.
.
.
VerilogFileExtension                  : '.v'
```

See Also

"View HDL Model Parameters"

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 string designating 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]
```

More About

Tips

- Use hdlget_param only to obtain the value of HDL block parameters (see "HDL Block Properties" 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

hdlset_param | hdlsaveparams | hdlrestoreparams

hdllib

Display blocks that are compatible with HDL code generation

Syntax

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

Description

`hdllib` displays in the Library Browser the blocks that are supported for HDL code generation and for which you have a license. 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, enter `hdllib('off')`.

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

`hdllib('html')` creates a library of blocks that are compatible with HDL code generation, and 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 `hdlsupported`. 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 you install a new release.

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

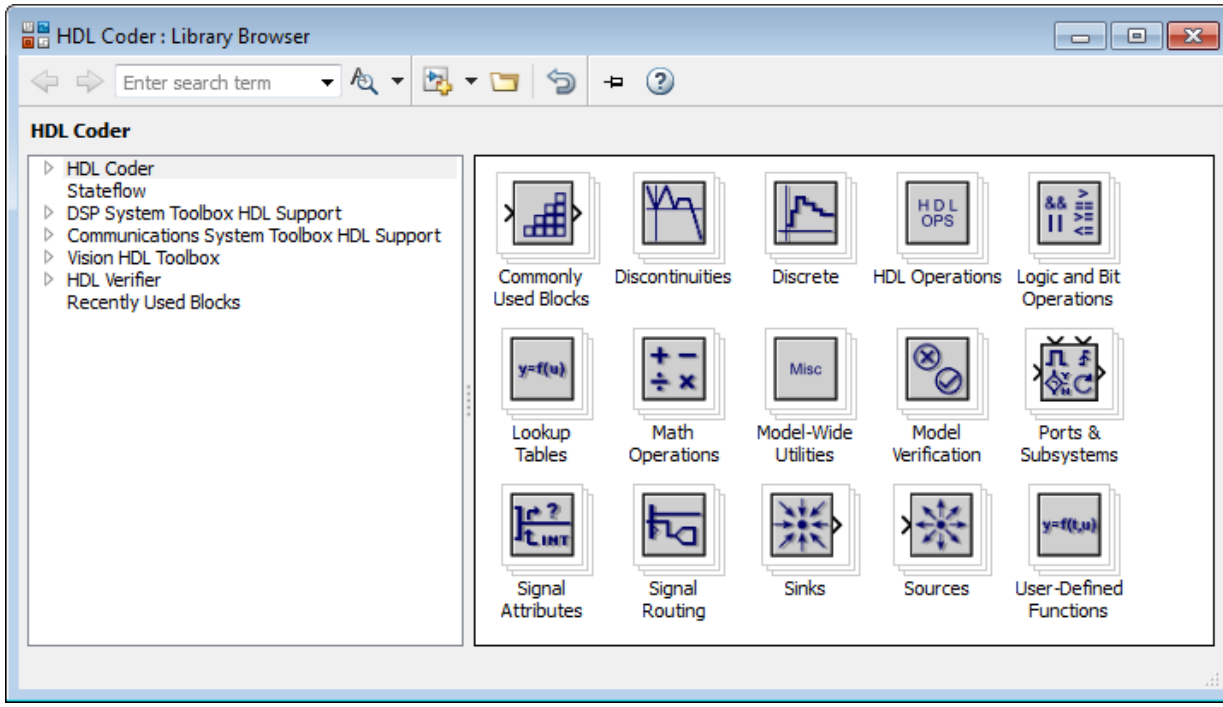
Examples

Display Supported Blocks in the Library Browser

To display HDL code generation compatible blocks 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".
```

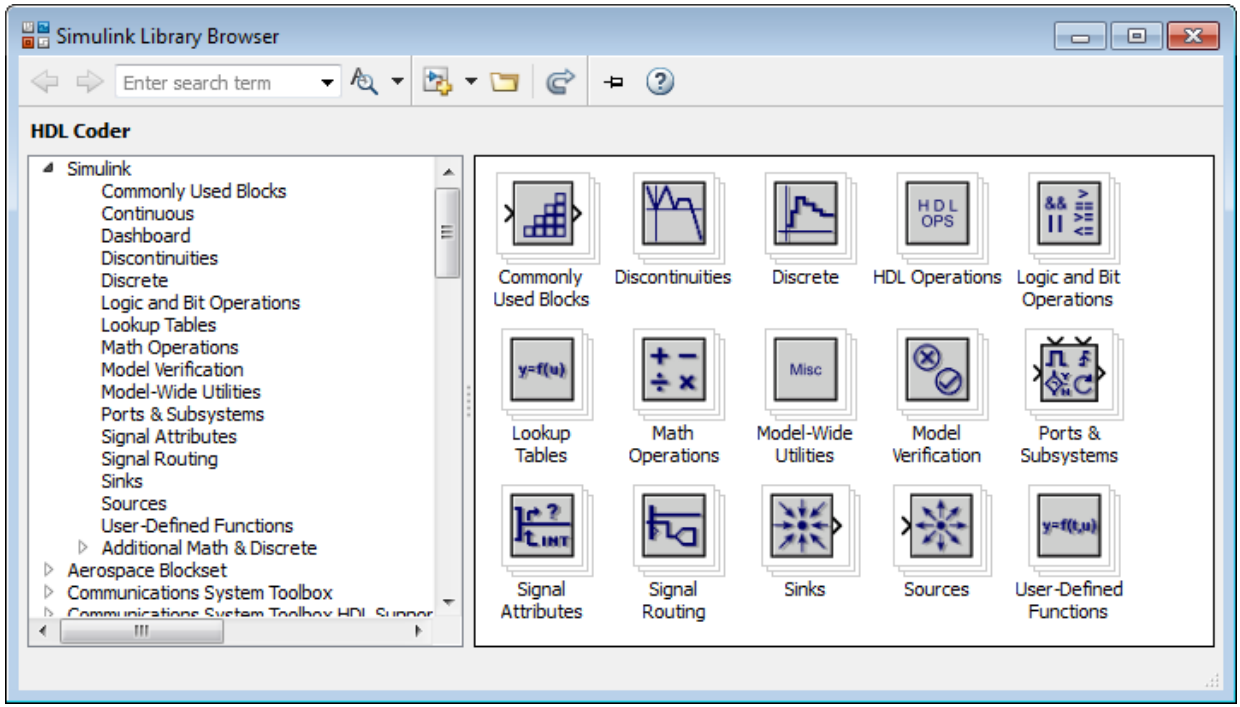


Display All Blocks in the Library Browser

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 fi
```



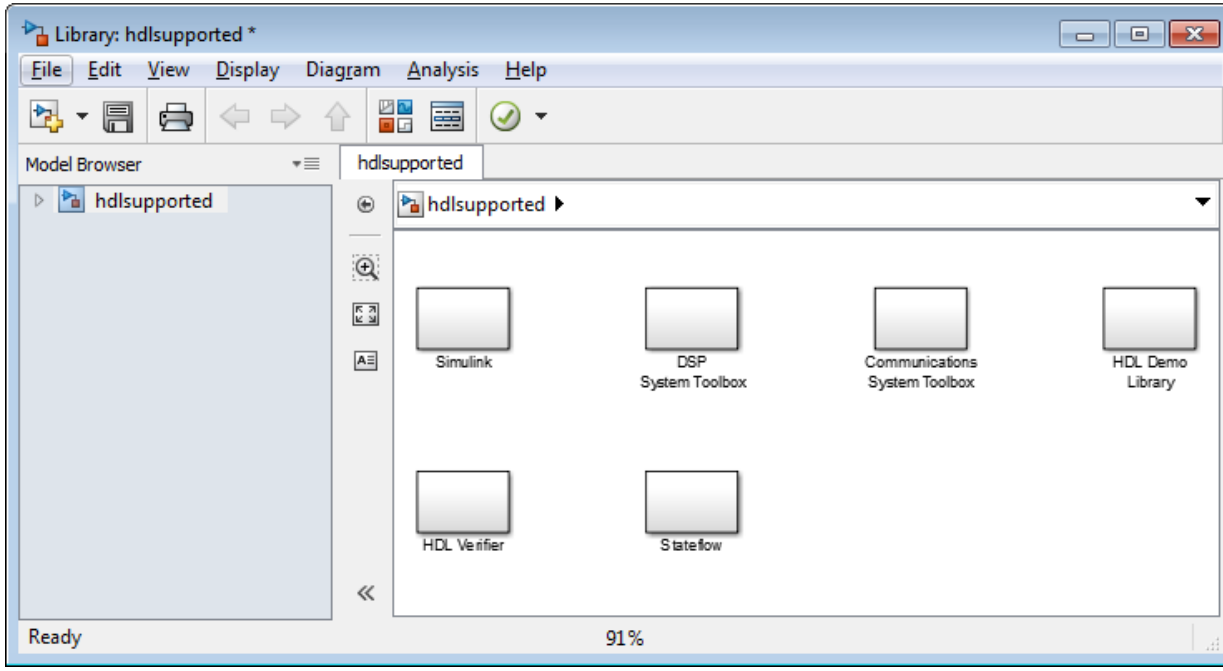
Create a Supported Blocks Library and HTML Reports

To create a library and HTML reports showing blocks supported for 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.

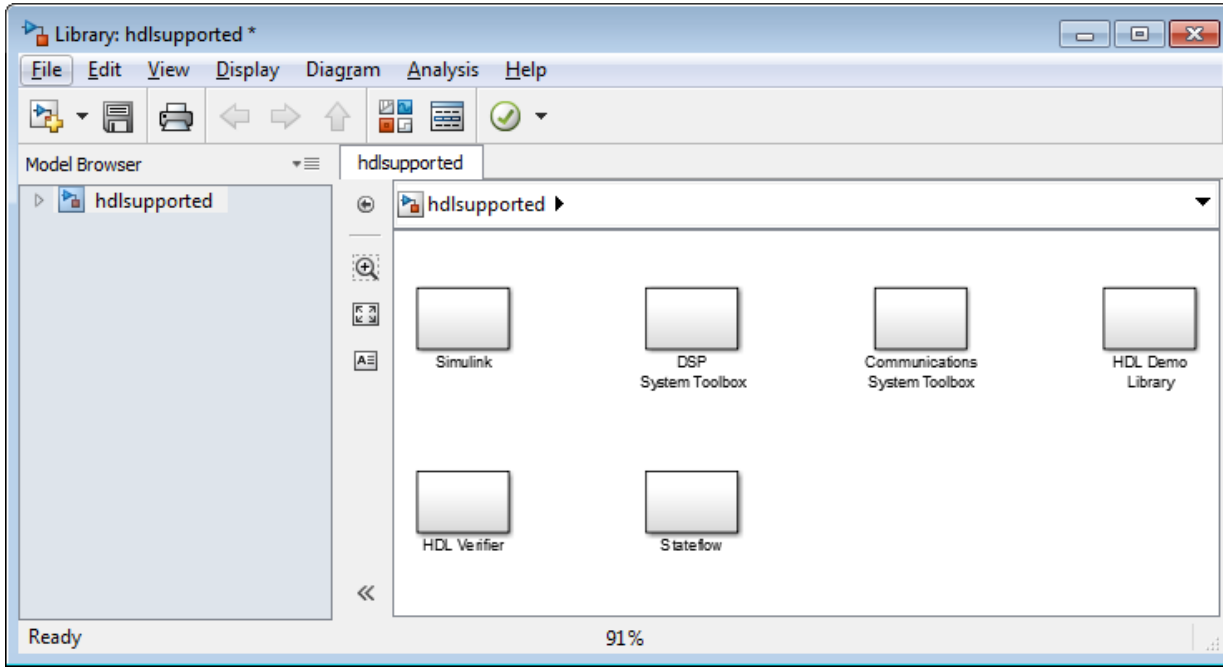


Create a Supported Blocks Library

To create a library that contains blocks supported for HDL code generation:

```
hdllib('librarymodel')
```

The `hdlsupported` block library opens.



- “Show Blocks Supported for HDL Code Generation”
- “View HDL-Specific Block Documentation”
- “Prepare Simulink Model For HDL Code Generation”

See Also

“Supported Blocks”

Introduced in R2006b

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

Reset and Restore HDL-Related Model Parameters

Open the model.

```
sfir_fixed
```

Verify that model parameters have default values.

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

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

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

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

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

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')

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')

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

Input Arguments

`dut` — DUT subsystem name

string

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file

string

Name of file containing previously saved HDL model parameters.

Example: `'mymodel_saved_params.m'`

See Also

`hdlsaveparams`

hdlsaveparams

Save nondefault block- and model-level HDL parameters

Syntax

```
hdlsaveparams(dut)
hdlsaveparams(dut,filename)
```

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.

Examples

Display HDL-Related Nondefault Model Parameters

Open the model.

```
sfir_fixed
```

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

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

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

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

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

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

Save and Restore HDL-Related Model Parameters

Open the model.

```
sfir_fixed
```

Verify that model parameters have default values.

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

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed');
```

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

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

Verify that model parameters are set.

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

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

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

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

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')
```

```
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')
```

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

Input Arguments

`dut` — DUT subsystem name

string

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file

string

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

Example: `'mymodel_saved_params.m'`

See Also

`hdlrestoreparams`

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 string specifying the name of 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” 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;
```

More About

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.

- “Set and View HDL Block Parameters”
- “Set HDL Block Parameters for Multiple Blocks”

See Also

`hdlget_param` | `hdlsaveparams` | `hdlrestoreparams`

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 **Solver** options used by HDL Coder. These options are:

- **Type**: `Fixed-step`. This is the recommended solver type for most HDL applications.

  HDL Coder also supports variable-step solvers under the following conditions:

  - The device under test (DUT) is single-rate.
  - The sample times of all signals driving the DUT are greater than 0.
- **Solver**: `Discrete (no continuous states)`. You can use other fixed-step solvers, but this option is usually best for simulating discrete systems.
- **Tasking mode**: `SingleTasking`. HDL Coder does not support multitasking mode.

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

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 Altera Quartus II

The following command sets the synthesis tool path to point to an installed Altera Quartus II 14.0 executable file. You must have already installed Altera Quartus II.

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath',...
 'C:\altera\14.0\quartus\bin\quartus.exe');
```

Note: In this example, the path to the Quartus II executable file is C:\altera \14.0\quartus\bin\quartus.exe. If the path to your executable file is different, use your path.

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');
```

Note: In this example, the path to the ISE executable file is C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\ise.exe. If the path to your executable file is different, use your path.

Set Up Xilinx Vivado

The following command sets the synthesis tool path to point to an installed Vivado® Design Suite 2014.2 batch file. You must have already installed Xilinx Vivado.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
 'C:\Xilinx\Vivado\2014.2\bin\vivado.bat');
```

Note: In this example, the path to the Vivado batch file is C:\Xilinx\Vivado\2014.2\bin\vivado.bat. If the path to your batch file is different, use your path.

Input Arguments

`TOOLNAME` — Synthesis tool name

string

Synthesis tool name, specified as a string.

Example: `'Xilinx Vivado'`

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

string

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

Example: `'C:\Xilinx\Vivado\2014.2\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.

More About

- "Supported Third-Party Tools and Hardware"
- "Tool Setup"
- "Add Synthesis Tool for Current HDL Workflow Advisor Session"

See Also

`setenv | startup`

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.

`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

Generate VHDL code for the current model.

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

```
makehdl(bdroot)
```

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;
```

The model opens in a new Simulink window.

Generate Verilog for the `symmetric_fir` subsystem.

```
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.
```

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;
```

The model opens in a new Simulink window.

Check the `symmetric_fir` subsystem for compatibility with HDL code generation. Generate code with code generation options set to default values.

```
makehdl('sfir_fixed/symmetric_fir','CheckHDL','on')
```

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
string

DUT model or subsystem name, 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 single 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'`

For more information, see `TargetLanguage`.

`'TargetDirectory'` — Output directory
`'hdlsrc'` (default) | string

For more information, see `TargetDirectory`.

`'CheckHDL'` — Check HDL code generation compatibility
`'off'` (default) | `'on'`

For more information, see `CheckHDL`.

`'GenerateHDLCode'` — Generate HDL code
`'on'` (default) | `'off'`

For more information, see `GenerateHDLCode`.

`'SplitEntityArch'` — Split VHDL entity and architecture into separate files
`'off'` (default) | `'on'`

For more information, see `SplitEntityArch`.

`'UseSingleLibrary'` — Generate VHDL code for model references into a single library
`'off'` (default) | `'on'`

For more information, see `UseSingleLibrary`.

`'Verbosity'` — Level of message detail
`1` (default) | `0`

For more information, see `Verbosity`.

Report Generation

`'HDLCodingStandard'` — Specify HDL coding standard
string

For more information, see `HDLCodingStandard`.

`'HDLCodingStandardCustomizations'` — Specify HDL coding standard customization object
`hdlcoder.CodingStandard` object

For more information, see `HDLCodingStandardCustomizations`.

`'Traceability'` — Generate report with mapping links between HDL and model
`'off'` (default) | `'on'`

For more information, see `Traceability`.

`'ResourceReport'` — Resource utilization report generation
`'off'` (default) | `'on'`

For more information, see `ResourceReport`.

`'OptimizationReport'` — Optimization report generation
`'off'` (default) | `'on'`

For more information, see `OptimizationReport`.

`'GenerateWebview'` — Include model Web view
`'on'` (default) | `'off'`

For more information, see `GenerateWebview`.

Speed and Area Optimization

`'BalanceDelays'` — Delay balancing
`'on'` (default) | `'off'`

For more information, see `BalanceDelays`.

`'DistributedPipeliningPriority'` — Specify priority for distributed pipelining algorithm
`'NumericalIntegrity'` (default) | `'Performance'`

For more information, see `DistributedPipeliningPriority`.

`'HierarchicalDistPipelining'` — Hierarchical distributed pipelining
`'off'` (default) | `'on'`

For more information, see `HierarchicalDistPipelining`.

`'PreserveDesignDelays'` — Prevent distributed pipelining from moving design delays
`'off'` (default) | `'on'`

For more information, see `PreserveDesignDelays`.

`'ClockRatePipelining'` — Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths
`'on'` (default) | `'off'`

For more information, see `ClockRatePipelining`.

`'MaxOversampling'` — Limit the maximum sample rate
0 (default) | N, where N is an integer greater than 1

For more information, see `MaxOversampling`.

`'MaxComputationLatency'` — Specify the maximum number of time steps for which your DUT inputs are guaranteed to be stable
1 (default) | N, where N is an integer greater than 1

For more information, see `MaxComputationLatency`.

`'MinimizeClockEnables'` — Omit clock enable logic for single-rate designs
`'off'` (default) | `'on'`

For more information, see `MinimizeClockEnables`.

`'RAMMappingThreshold'` — Minimum RAM size for mapping to RAMs instead of registers
256 (default) | positive integer

The minimum RAM size required for mapping to RAMs instead of registers, specified in bits.

For more information, see `RAMMappingThreshold`.

`'HighlightFeedbackLoops'` — Highlight feedback loops inhibiting delay balancing and optimizations
`'off'` (default) | `'on'`

For more information, see `HighlightFeedbackLoops`.

`'HighlightFeedbackLoopsFile'` — Feedback loop highlighting script file name
`'highlightFeedbackLoop'` (default) | string

For more information, see `HighlightFeedbackLoopsFile`.

Coding Style

`'UserComment'` — HDL file header comment
string

For more information, see `UserComment`.

`'UseAggregatesForConst'` — Represent constant values with aggregates
`'off'` (default) | `'on'`

For more information, see `UseAggregatesForConst`.

`'UseRisingEdge'` — Use VHDL `rising_edge` or `falling_edge` function to detect clock transitions
`'off'` (default) | `'on'`

For more information, see `UseRisingEdge`.

`'LoopUnrolling'` — Unroll VHDL `FOR` and `GENERATE` loops
`'off'` (default) | `'on'`

For more information, see `LoopUnrolling`.

`'UseVerilogTimescale'` — Generate `'timescale` compiler directives
`'on'` (default) | `'off'`

For more information, see `UseVerilogTimescale`.

`'InlineConfigurations'` — Include VHDL configurations
`'on'` (default) | `'off'`

For more information, see `InlineConfigurations`.

`'SafeZeroConcat'` — Type-safe syntax for concatenated zeros
`'on'` (default) | `'off'`

For more information, see `SafeZeroConcat`.

`'DateComment'` — Include time stamp in header
`'on'` (default) | `'off'`

For more information, see `DateComment`.

`'ScalarizePorts'` — Flatten vector ports into scalar ports
`'off'` (default) | `'on'`

For more information, see `ScalarizePorts`.

`'MinimizeIntermediateSignals'` — Minimize intermediate signals
`'off'` (default) | `'on'`

For more information, see `MinimizeIntermediateSignals`.

`'RequirementComments'` — Link from code generation reports to requirement documents
`'on'` (default) | `'off'`

For more information, see `RequirementComments`.

`'InlineMATLABBlockCode'` — Inline HDL code for MATLAB Function blocks
`'off'` (default) | `'on'`

For more information, see `InlineMATLABBlockCode`.

`'MaskParameterAsGeneric'` — Reusable code generation for subsystems with identical mask parameters
`'off'` (default) | `'on'`

For more information, see `MaskParameterAsGeneric`.

`'InitializeBlockRAM'` — Initial signal value generation for RAM blocks
`'on'` (default) | `'off'`

For more information, see `InitializeBlockRAM`.

`'RAMArchitecture'` — RAM architecture
`'WithClockEnable'` (default) | `'WithoutClockEnable'`

For more information, see `RAMArchitecture`.

`'HandleAtomicSubsystem'` — Reusable code generation for identical atomic subsystems
`'on'` (default) | `'off'`

For more information, see `HandleAtomicSubsystem`.

Clocks and Reset

`'ClockEdge'` — Active clock edge
`'Rising'` (default) | `'Falling'`

For more information, see `ClockEdge`.

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

Single or multiple clock inputs, specified as a string.

For more information, see `ClockInputs`.

`'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`.

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

For more information, see `ResetAssertedLevel`.

`'ResetType'` — Reset type
`'async'` (default) | `'sync'`

For more information, see `ResetType`.

`'TriggerAsClock'` — Use trigger signal as clock in triggered subsystems
`'off'` (default) | `'on'`

For more information, see `TriggerAsClock`.

`'TimingControllerArch'` — Generate reset for timing controller
`'default'` (default) | `'resettable'`

For more information, see `TimingControllerArch`.

Test Bench

`'Verbosity'` — Level of message detail
0 (default) | `n`

For more information, see `Verbosity`.

`'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`.

`'GenerateValidationModel'` — Generate validation model
`'off'` (default) | `'on'`

For more information, see `GenerateValidationModel`.

`'SimulatorFlags'` — Options for generated compilation scripts
string

For more information, see `SimulatorFlags`.

`'TestBenchReferencePostFix'` — Suffix for test bench reference signals
`'_ref'` (default) | string

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 string
`'vlib work\n'` (default) | string

For more information, see `HDLCompileInit`.

`'HDLCompileTerm'` — Compilation script termination string
`''` (default) | string

For more information, see `HDLCompileTerm`.

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

For more information, see `HDLCompileFilePostfix`.

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

Verilog compilation command, specified as a string. 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) | string

VHDL compilation command, specified as a string. 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'`

HDL lint tool, specified as a string.

For more information, see `HDLLintTool`.

`'HDLLintInit'` — HDL lint initialization string
string

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

For more information, see `HDLLintInit`.

`'HDLLintCmd'` — HDL lint command
string

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

For more information, see `HDLLintCmd`.

'HDLLintTerm' — HDL lint termination string
string

HDL lint termination, specified as a string. 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'

HDL synthesis tool, specified as a string.

For more information, see HDLSynthTool.

'HDLSynthCmd' — HDL synthesis command
string

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

For more information, see HDLSynthCmd.

'HDLSynthFilePostfix' — Postfix for synthesis script file name
string

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

For more information, see HDLSynthFilePostfix.

'HDLSynthInit' — Synthesis script initialization string
string

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

For more information, see HDLSynthInit.

'HDLSynthTerm' — Synthesis script termination string
string

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

For more information, see `HDLSynthTerm`.

Generated Model

`'CodeGenerationOutput'` — Display and generation of generated model

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

For more information, see `CodeGenerationOutput`.

`'GeneratedModelName'` — Generated model name

same as original model name (default) | string

For more information, see `GeneratedModelName`.

`'GeneratedModelNamePrefix'` — Prefix for generated model name

`'gm_'` (default) | string

For more information, see `GeneratedModelNamePrefix`.

`'HighlightAncestors'` — Highlight parent blocks of generated model blocks differing from original model

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

For more information, see `HighlightAncestors`.

`'HighlightColor'` — Color of highlighted blocks in generated model

`'cyan'` (default) | `'yellow'` | `'magenta'` | `'red'` | `'green'` | `'blue'` | `'white'` | `'magenta'` | `'black'`

For more information, see `HighlightColor`.

Synthesis

`'SynthesisTool'` — Synthesis tool

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

For more information, see `SynthesisTool`.

`'MulticyclePathInfo'` — Multicycle path constraint file generation
`'off'` (default) | `'on'`

For more information, see `MulticyclePathInfo`.

Port Names and Types

`'ClockEnableInputPort'` — Clock enable input port name
`'clk_enable'` (default) | string

Clock enable input port name, specified as a string.

For more information, see `ClockEnableInputPort`.

`'ClockEnableOutputPort'` — Clock enable output port name
`'ce_out'` (default) | string

Clock enable output port name, specified as a string.

For more information, see `ClockEnableOutputPort`.

`'ClockInputPort'` — Clock input port name
`'clk'` (default) | string

Clock input port name, specified as a string.

For more information, see `ClockInputPort`.

`'InputType'` — HDL data type for input ports
`'wire'` or `'std_logic_vector'` (default) | `'signed/unsigned'`

HDL data type for input ports, specified as a string.

VHDL inputs can have `'std_logic_vector'` or `'signed/unsigned'` data type. Verilog inputs must be `'wire'`.

For more information, see `InputType`.

`'OutputType'` — HDL data type for output ports
`'Same as input data type'` (default) | `'std_logic_vector'` | `'signed/unsigned'` | `'wire'`

HDL data type for output ports, specified as a string.

VHDL output can be `'Same as input data type'`, `'std_logic_vector'` or `'signed/unsigned'`. Verilog output must be `'wire'`.

For more information, see `OutputType`.

`'ResetInputPort'` — Reset input port name
`'reset'` (default) | string

Reset input port name, specified as a string.

For more information, see `ResetInputPort`.

File and Variable Names

`'VerilogFileExtension'` — Verilog file extension
`'.v'` (default) | string

For more information, see `VerilogFileExtension`.

`'VHDLFileExtension'` — VHDL file extension
`'.vhd'` (default) | string

For more information, see `VHDLFileExtension`.

`'VHDLArchitectureName'` — VHDL architecture name
`'rtl'` (default) | string

For more information, see `VHDLArchitectureName`.

`'VHDLLibraryName'` — VHDL library name
`'work'` (default) | string

For more information, see `VHDLLibraryName`.

`'SplitEntityFilePostfix'` — Postfix for VHDL entity file names
`'_entity'` (default) | string

For more information, see `SplitEntityFilePostfix`.

`'SplitArchFilePostfix'` — Postfix for VHDL architecture file names
`'_arch'` (default) | string

For more information, see `SplitArchFilePostfix`.

'PackagePostfix' — Postfix for package file name
'_pkg' (default) | string

For more information, see PackagePostfix.

'HDLMapFilePostfix' — Postfix for mapping file
'_map.txt' (default) | string

For more information, see HDLMapFilePostfix.

'BlockGenerateLabel' — Block label postfix for VHDL GENERATE statements
'_gen' (default) | string

For more information, see BlockGenerateLabel.

'ClockProcessPostfix' — Postfix for clock process names
'_process' (default) | string

For more information, see ClockProcessPostfix.

'ComplexImagPostfix' — Postfix for imaginary part of complex signal
'_im' (default) | string

For more information, see ComplexImagPostfix.

'ComplexRealPostfix' — Postfix for imaginary part of complex signal names
'_re' (default) | string

For more information, see ComplexRealPostfix.

'EntityConflictPostfix' — Postfix for duplicate VHDL entity or Verilog module names
'_block' (default) | string

For more information, see EntityConflictPostfix.

'InstanceGenerateLabel' — Instance section label postfix for VHDL GENERATE statements
'_gen' (default) | string

For more information, see InstanceGenerateLabel.

'InstancePostfix' — Postfix for generated component instance names
'' (default) | string

For more information, see `InstancePostfix`.

`'InstancePrefix'` — Prefix for generated component instance names
`'u_'` (default) | string

For more information, see `InstancePrefix`.

`'OutputGenerateLabel'` — Output assignment label postfix for VHDL `GENERATE` statements
`'outputgen'` (default) | string

For more information, see `OutputGenerateLabel`.

`'PipelinePostfix'` — Postfix for input and output pipeline register names
`'_pipe'` (default) | string

For more information, see `PipelinePostfix`.

`'ReservedWordPostfix'` — Postfix for names conflicting with VHDL or Verilog reserved words
`'_rsvd'` (default) | string

For more information, see `ReservedWordPostfix`.

`'TimingControllerPostfix'` — Postfix for timing controller name
`'_tc'` (default) | string

For more information, see `TimingControllerPostfix`.

`'VectorPrefix'` — Prefix for vector names
`'vector_of_'` (default) | string

For more information, see `VectorPrefix`.

`'EnablePrefix'` — Prefix for internal enable signals
`'enb'` (default) | string

Prefix for internal clock enable and control flow enable signals, specified as a string.

For more information, see `EnablePrefix`.

`'ModulePrefix'` — Prefix for modules or entity names
`''` (default) | string

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`.

See Also

`checkhdl` | `makehdltb`

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.

`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.

Input Arguments

`dut` — DUT subsystem or model reference name

string

DUT subsystem or model reference name, specified as a string, 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 single 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'`

For more information, see `TargetLanguage`.

`'TargetDirectory'` — Output directory

`'hdlsrc'` (default) | string

For more information, see `TargetDirectory`.

`'SplitEntityArch'` — Split VHDL entity and architecture into separate files

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

For more information, see `SplitEntityArch`.

Test Bench

`'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'`

Single or multiple clock inputs, specified as a string.

For more information, see `ClockInputs`.

`'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 `ResetLength`.

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

For more information, see `ResetAssertedLevel`.

`'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`.

`'UseFileIOInTestBench'` — Use file I/O to read/write test bench data
`'on'` (default) | `'off'`

For more information, see `UseFileIOInTestBench`.

`'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) | string

For more information, see `TestBenchDataPostFix`.

`'TestBenchPostFix'` — Suffix for test bench name
`'_tb'` (default) | string

For more information, see `TestBenchPostFix`.

`'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`.

Coding Style

`'UseVerilogTimescale'` — Generate `'timescale` compiler directives
`'on'` (default) | `'off'`

For more information, see `UseVerilogTimescale`.

`'DateComment'` — Include time stamp in header
`'on'` (default) | `'off'`

For more information, see `DateComment`.

`'InlineConfigurations'` — Include VHDL configurations
`'on'` (default) | `'off'`

For more information, see `InlineConfigurations`.

`'ScalarizePorts'` — Flatten vector ports into scalar ports
`'off'` (default) | `'on'`

For more information, see `ScalarizePorts`.

Script Generation

`'HDLCompileInit'` — Compilation script initialization string
`'vlib work\n'` (default) | string

For more information, see `HDLCompileInit`.

`'HDLCompileTerm'` — Compilation script termination string
`''` (default) | string

For more information, see `HDLCompileTerm`.

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

For more information, see `HDLCompileFilePostfix`.

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

Verilog compilation command, specified as a string. 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) | string

VHDL compilation command, specified as a string. 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) | string

The HDL simulation command, specified as a string.

For more information, see `HDLSimCmd`.

`'HDLSimInit'` — HDL simulation script initialization string
`['onbreak resume\n', 'onerror resume\n']` (default) | string

Initialization for the HDL simulation script, specified as a string.

For more information, see `HDLSimInit`.

`'HDLSimTerm'` — HDL simulation script termination string
`'run -all'` (default) | string

The termination string for the HDL simulation command.

For more information, see `HDLSimTerm`.

`'HDLSimFilePostfix'` — Postscript for HDL simulation script
`'_sim.do'` (default) | string

For more information, see `HDLSimFilePostfix`.

`'HDLSimViewWaveCmd'` — HDL simulation waveform viewing command
`'add wave sim:%s\n'` (default) | string

Waveform viewing command, specified as a string. 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) | string

Clock enable input port name, specified as a string.

For more information, see `ClockEnableInputPort`.

`'ClockEnableOutputPort'` — Clock enable output port name
`'ce_out'` (default) | string

Clock enable output port name, specified as a string.

For more information, see `ClockEnableOutputPort`.

`'ClockInputPort'` — Clock input port name
`'clk'` (default) | string

Clock input port name, specified as a string.

For more information, see `ClockInputPort`.

`'ResetInputPort'` — Reset input port name
`'reset'` (default) | string

Reset input port name, specified as a string.

For more information, see `ResetInputPort`.

File and Variable Names

`'VerilogFileExtension'` — Verilog file extension
`'.v'` (default) | string

For more information, see `VerilogFileExtension`.

`'VHDLFileExtension'` — VHDL file extension
`'.vhd'` (default) | string

For more information, see `VHDLFileExtension`.

`'VHDLArchitectureName'` — VHDL architecture name
`'rtl'` (default) | string

For more information, see `VHDLArchitectureName`.

'VHDLLibraryName' — VHDL library name
'work' (default) | string

For more information, see VHDLLibraryName.

'SplitEntityFilePostfix' — Postfix for VHDL entity file names
'_entity' (default) | string

For more information, see SplitEntityFilePostfix.

'SplitArchFilePostfix' — Postfix for VHDL architecture file names
'_arch' (default) | string

For more information, see SplitArchFilePostfix.

'PackagePostfix' — Postfix for package file name
'_pkg' (default) | string

For more information, see PackagePostfix.

'ComplexImagPostfix' — Postfix for imaginary part of complex signal
'_im' (default) | string

For more information, see ComplexImagPostfix.

'ComplexRealPostfix' — Postfix for imaginary part of complex signal names
'_re' (default) | string

For more information, see ComplexRealPostfix.

'EnablePrefix' — Prefix for internal enable signals
'enb' (default) | string

Prefix for internal clock enable and control flow enable signals, specified as a string.

For more information, see EnablePrefix.

See Also

makehdl

2

Supported Blocks

1-D Lookup Table

1-D Lookup Table implementations, properties, and restrictions for HDL code generation

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`.

2-D Lookup Table

2-D Lookup Table implementations, properties, and restrictions for HDL code generation

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`.

Abs

Abs implementations, properties, and restrictions for HDL code generation

Description

The Abs block is available with Simulink.

For information on the simulation behavior and block parameters, see `Abs`.

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 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.

Add

Add implementations, properties, and restrictions for HDL code generation

Description

The Add block is available with Simulink.

For information on 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

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

The default `Linear` implementation supports complex data.

Assertion

Assertion implementations, properties, and restrictions for HDL code generation

Description

The Assertion block is available with Simulink.

For information on 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.

Assignment

Assignment implementations, properties, and restrictions for HDL code generation

Description

The Assignment block is available with Simulink.

For information on 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.

Atomic Subsystem

Atomic Subsystem implementations, properties, and restrictions for HDL code generation

Description

The Atomic Subsystem block is available with Simulink.

For information on 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`	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. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of 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

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. 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 values are loaded in the corresponding fields.

ProcessorFPGASynchronization

Processor / FPGA synchronization mode, specified as a string.

In the HDL Workflow Advisor, you can set this property in the **Processor/FPGA Synchronization** field.

Values: `Free running` (default) | `Coprocessing - blocking`

Example: `'Free running'`

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 (;).

In the HDL Workflow Advisor, you can set this property in the **Additional source files** field.

Values: `''` (default) | string

Example: `'C:\myprojfiles\led_blinking_file1.vhd;C:\myprojfiles\led_blinking_file2.vhd;'`

IPCoreName

IP core name, specified as a string.

In the HDL Workflow Advisor, you can set this property using 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) | string

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a string.

In the HDL Workflow Advisor, you can set this property using 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) | string

Example: `'1.3'`

More About

- "External Component Interfaces"
- "Generate Black Box Interface for Subsystem"

Backlash

Backlash implementations, properties, and restrictions for HDL code generation

Description

The Backlash block is available with Simulink.

For information on 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.

Bias

Bias implementations, properties, and restrictions for HDL code generation

Description

The Bias block is available with Simulink.

For information on 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”.

Bit Clear

Bit Clear implementations, properties, and restrictions for HDL code generation

Description

The Bit Clear block is available with Simulink.

For information on 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”.

Bit Concat

Bit Concat implementations, properties, and restrictions for HDL code generation

Description

The Bit Concat block is available with Simulink.

For information on the simulation behavior and block parameters, see `Bit Concat`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Bit Reduce

Bit Reduce implementations, properties, and restrictions for HDL code generation

Description

The Bit Reduce block is available with Simulink.

For information on the simulation behavior and block parameters, see `Bit Reduce`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Bit Rotate

Bit Rotate implementations, properties, and restrictions for HDL code generation

Description

The Bit Rotate block is available with Simulink.

For information on the simulation behavior and block parameters, see `Bit Rotate`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Bit Set

Bit Set implementations, properties, and restrictions for HDL code generation

Description

The Bit Set block is available with Simulink.

For information on 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”.

Bit Shift

Bit Shift implementations, properties, and restrictions for HDL code generation

Description

The Bit Shift block is available with Simulink.

For information on the simulation behavior and block parameters, see `Bit Shift`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Bit Slice

Bit Slice implementations, properties, and restrictions for HDL code generation

Description

The Bit Slice block is available with Simulink.

For information on the simulation behavior and block parameters, see `Bit Slice`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Bitwise Operator

Bitwise Operator implementations, properties, and restrictions for HDL code generation

Description

The Bitwise Operator block is available with Simulink.

For information on 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”.

Biquad Filter

Biquad Filter implementations, properties, and restrictions for HDL code generation

Description

The Biquad Filter block is available with DSP System Toolbox™.

For information on the simulation behavior and block parameters, see `Biquad Filter`.

HDL Architecture

Programmable Filter Support

HDL Coder supports programmable filters for Biquad Filters. A fully parallel architecture is supported.

1. Select **Input port(s)** as coefficient source on the filter block mask.
2. Connect the coefficient port with a vector signal.
3. Specify the implementation architecture and parameters from the HDL Coder property interface.

   The following configurations are not supported for programmable filters:

   - Fully serial and partly serial architectures
   - `CoeffMultipliers` as `csd` or `factored-csd`
4. Generate HDL code.

Serial Architecture Support

The Biquad Filter block supports fully parallel, fully serial, and partly serial architectures for `Direct form I` and `Direct form II` filter structures. Serial architecture is not supported for `Direct form I transposed` and `Direct form II transposed` filter structures.

AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on 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.	Where `NS` is number of sections: `NS-1`

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 this parameter to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. For more information, see 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

- Data vector and frame inputs are not supported for HDL code generation.
- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- You must select **Optimize unity scale values**.

BPSK Demodulator Baseband

BPSK Demodulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The BPSK Demodulator Baseband block is available with Communications System Toolbox™.

For information on 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”.

BPSK Modulator Baseband

BPSK Modulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The BPSK Modulator Baseband block is available with Communications System Toolbox.

For information on 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”.

Bus Assignment

Bus Assignment implementations, properties, and restrictions for HDL code generation

Description

The Bus Assignment block is available with Simulink.

For information on 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”.

Restrictions

For HDL code generation:

- Nonvirtual inputs are not supported.

- A bus signal cannot be connected to an Assignment port.

More About

- "Buses"

Bus Creator

Bus Creator implementations, properties, and restrictions for HDL code generation

Description

The Bus Creator block is available with Simulink.

For information on 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”.

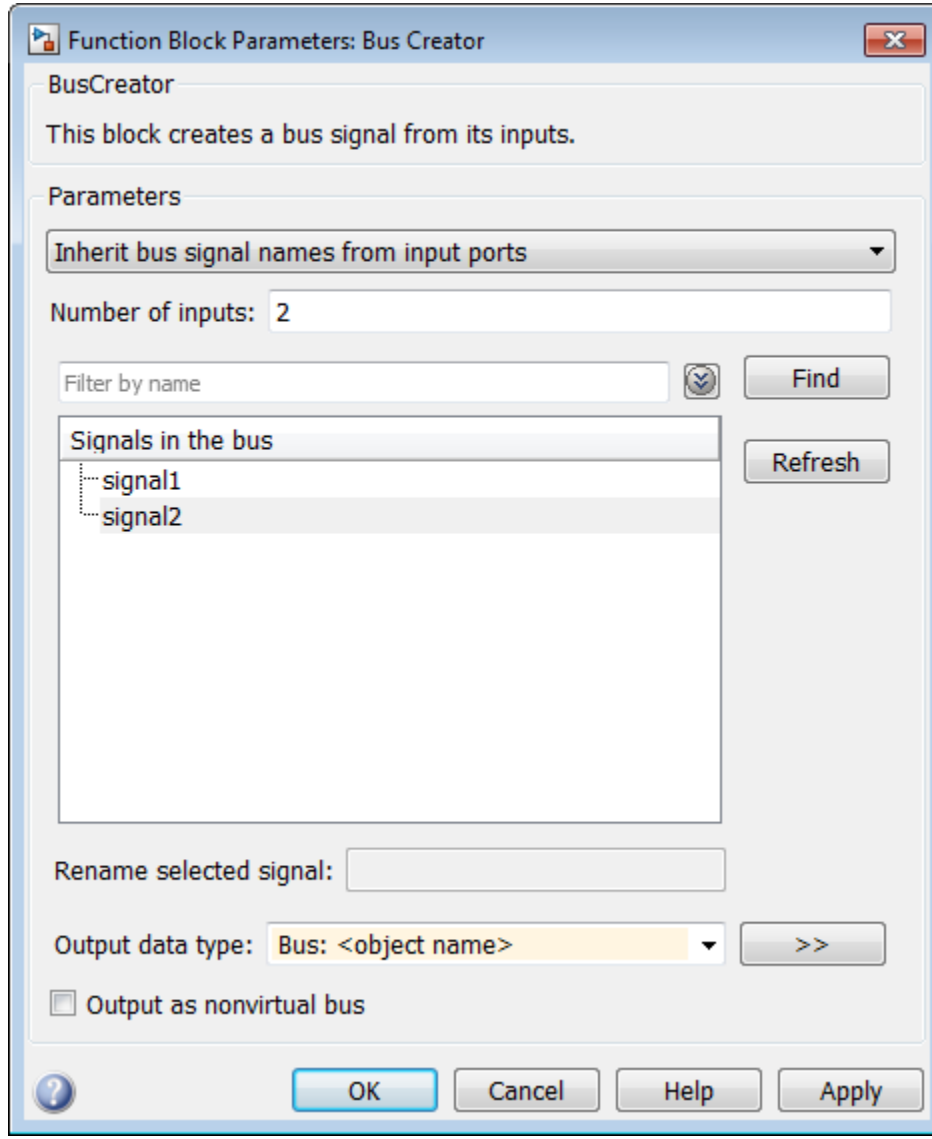
OutputPipeline

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

- Set the **Simulation > Configuration Parameters > Diagnostics > Connectivity+Mux blocks used to create bus signals** parameter to `error`. For details, see "Prevent Bus and Mux Mixtures".
- For **Output data type**, specify a bus object.


Function Block Parameters: Bus Creator
BusCreator
This block creates a bus signal from its inputs.
Parameters
Inherit bus signal names from input ports
Number of inputs: 2
Filter by name
Find
Signals in the bus
signal1
signal2
Refresh
Rename selected signal:
Output data type: Bus: <object name>
>>
Output as nonvirtual bus
OK
Cancel
Help
Apply

More About

- “Buses”

Bus Selector

Bus Selector implementations, properties, and restrictions for HDL code generation

Description

The Bus Selector block is available with Simulink.

For information on 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

- You must set the **Simulation > Configuration Parameters > Diagnostics > Connectivity+Mux blocks used to create bus signals** parameter to `error`. For details, see “Prevent Bus and Mux Mixtures”.

- Inputs must be bus signals. Non-bus inputs are not supported for code generation.

More About

- "Buses"

Chart

Chart implementations, properties, and restrictions for HDL code generation

Description

The Chart block is available with Stateflow®.

For information on the simulation behavior and block parameters, see `Chart`.

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 "Use Active State Output Data".

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 string, 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" 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.

Enable the **Initialize Outputs Every Time Chart Wakes Up** chart property to generate HDL code that is more readable and has better synthesis results. 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 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.

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 Using Input Events” and “Activate a Simulink Block Using Output Events” in the Stateflow documentation.

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".
- 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.

See Also

`Message Viewer` | `State Transition Table` | `Truth Table`

Related Examples

- "Generate HDL for Mealy and Moore Finite State Machines"

- “Design Patterns Using Advanced Chart Features”

More About

- “Hardware Realization of Stateflow Semantics”

Check Dynamic Gap

Check Dynamic Gap implementations, properties, and restrictions for HDL code generation

Description

The Check Dynamic Gap block is available with Simulink.

For information on 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.

Check Dynamic Range

Check Dynamic Range implementations, properties, and restrictions for HDL code generation

Description

The Check Dynamic Range block is available with Simulink.

For information on 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.

Check Static Gap

Check Static Gap implementations, properties, and restrictions for HDL code generation

Description

The Check Static Gap block is available with Simulink.

For information on 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.

Check Static Range

Check Static Range implementations, properties, and restrictions for HDL code generation

Description

The Check Static Range block is available with Simulink.

For information on 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.

Check Discrete Gradient

Check Discrete Gradient implementations, properties, and restrictions for HDL code generation

Description

The Check Discrete Gradient block is available with Simulink.

For information on 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.

Check Dynamic Lower Bound

Check Dynamic Lower Bound implementations, properties, and restrictions for HDL code generation

Description

The Check Dynamic Lower Bound block is available with Simulink.

For information on 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.

Check Dynamic Upper Bound

Check Dynamic Upper Bound implementations, properties, and restrictions for HDL code generation

Description

The Check Dynamic Upper Bound block is available with Simulink.

For information on 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.

Check Input Resolution

Check Input Resolution implementations, properties, and restrictions for HDL code generation

Description

The Check Input Resolution block is available with Simulink.

For information on 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.

Check Static Lower Bound

Check Static Lower Bound implementations, properties, and restrictions for HDL code generation

Description

The Check Static Lower Bound block is available with Simulink.

For information on 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.

Check Static Upper Bound

Check Static Upper Bound implementations, properties, and restrictions for HDL code generation

Description

The Check Static Upper Bound block is available with Simulink.

For information on 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.

Chroma Resampler

Chroma Resampler implementations, properties, and restrictions for HDL code generation

Description

The Chroma Resampler block is available with Vision HDL Toolbox™.

For information on 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”.

CIC Decimation

CIC Decimation implementations, properties, and restrictions for HDL code generation

Description

The CIC Decimation block is available with DSP System Toolbox.

For information on 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 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 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 decimator` is not supported for HDL code generation. From the **Filter Structure** drop-down list, select `Decimator`.

CIC Interpolation

CIC Interpolation implementations, properties, and restrictions for HDL code generation

Description

The CIC Interpolation block is available with DSP System Toolbox.

For information on 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 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`.

Closing

Closing implementations, properties, and restrictions for HDL code generation

Description

The Closing block is available with Vision HDL Toolbox.

For information on 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”.

Color Space Converter

Color Space Converter implementations, properties, and restrictions for HDL code generation

Description

The Color Space Converter block is available with Vision HDL Toolbox.

For information on 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”.

Coulomb and Viscous Friction

Coulomb and Viscous Friction implementations, properties, and restrictions for HDL code generation

Description

The Coulomb and Viscous Friction block is available with Simulink.

For information on 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.

Compare To Constant

Compare To Constant implementations, properties, and restrictions for HDL code generation

Description

The Compare To Constant block is available with Simulink.

For information on 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.

Compare To Zero

Compare To Zero implementations, properties, and restrictions for HDL code generation

Description

The Compare To Zero block is available with Simulink.

For information on 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.

Complex to Magnitude-Angle HDL Optimized

Complex to Magnitude-Angle HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The Complex to Magnitude-Angle HDL Optimized block is available with DSP System Toolbox.

For information on 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.

Complex to Real-Imag

Complex to Real-Imag implementations, properties, and restrictions for HDL code generation

Description

The Complex to Real-Imag block is available with Simulink.

For information on 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.

Constant

Constant implementations, properties, and restrictions for HDL code generation

Description

The Constant block is available with Simulink.

For information on 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'`.
	`{'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.
- When **Output data type** is a bus object, **Constant value** cannot be 0.

Constellation Diagram

Constellation Diagram implementations, properties, and restrictions for HDL code generation

Description

The Constellation Diagram block is available with Communications System Toolbox.

For information on 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.

Convert 1-D to 2-D

Convert 1-D to 2-D implementations, properties, and restrictions for HDL code generation

Description

The Convert 1-D to 2-D block is available with DSP System Toolbox.

For information on 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.

Convolutional Deinterleaver

Convolutional Deinterleaver implementations, properties, and restrictions for HDL code generation

Description

The Convolutional Deinterleaver block is available with Communications System Toolbox.

For information on the simulation behavior and block parameters, see `Convolutional Deinterleaver`.

HDL Architecture

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.

Convolutional Encoder

Convolutional Encoder implementations, properties, and restrictions for HDL code generation

Description

The Convolutional Encoder block is available with Communications System Toolbox.

For information on 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:

- ½ to 1/7
- 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.

Convolutional Interleaver

Convolutional Interleaver implementations, properties, and restrictions for HDL code generation

Description

The Convolutional Interleaver block is available with Communications System Toolbox.

For information on the simulation behavior and block parameters, see `Convolutional Interleaver`.

HDL Architecture

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.

Cosine

Cosine implementations, properties, and restrictions for HDL code generation

Description

The Cosine block is available with Simulink.

For information on 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”.

Restrictions

This block does not have restrictions for HDL code generation.

Counter Free-Running

Counter Free-Running implementations, properties, and restrictions for HDL code generation

Description

The Counter Free-Running block is available with Simulink.

For information on 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”.

Counter Limited

Counter Limited implementations, properties, and restrictions for HDL code generation

Description

The Counter Limited block is available with Simulink.

For information on 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”.

Data Type Conversion

Data Type Conversion implementations, properties, and restrictions for HDL code generation

Description

The Data Type Conversion block is available with Simulink.

For information on the simulation behavior and block parameters, see `Data Type 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.

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.

Data Type Duplicate

Data Type Duplicate implementations, properties, and restrictions for HDL code generation

Description

The Data Type Duplicate block is available with Simulink.

For information on 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.

Data Type Propagation

Data Type Propagation implementations, properties, and restrictions for HDL code generation

Description

The Data Type Propagation block is available with Simulink.

For information on 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.

Dead Zone

Dead Zone implementations, properties, and restrictions for HDL code generation

Description

The Dead Zone block is available with Simulink.

For information on 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”.

DC Blocker

DC Blocker implementations, properties, and restrictions for HDL code generation

Description

The DC Blocker block is available with DSP System Toolbox.

For information on 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.

Dead Zone Dynamic

Dead Zone Dynamic implementations, properties, and restrictions for HDL code generation

Description

The Dead Zone Dynamic block is available with Simulink.

For information on 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”.

Decrement Real World

Decrement Real World implementations, properties, and restrictions for HDL code generation

Description

The Decrement Real World block is available with Simulink.

For information on 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”.

Decrement Stored Integer

Decrement Stored Integer implementations, properties, and restrictions for HDL code generation

Description

The Decrement Stored Integer block is available with Simulink.

For information on 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”.

Delay

Delay implementations, properties, and restrictions for HDL code generation

Description

The Delay block is available with Simulink. For information on simulation behavior and block parameters, see `Delay`.

To generate a reset port in the HDL code, set **External reset** to `Level`.

To generate an enable port in the HDL code, select **Show enable port**.

To specify an initial condition, in the Block Parameters dialog box, for **Initial condition**, set **Source** to `Dialog` and enter the 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”.

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

- **Initial condition** with **Source** set to `Input port` is not supported.

Delay (Obsolete)

Delay implementations, properties, and restrictions for HDL code generation

Description

The Delay block is available with DSP System Toolbox.

For information on the simulation behavior and block parameters, see `Delay`.

Note: The Delay block from the `dspsigops` library has been replaced by the `Delay` block from the Discrete library in Simulink. Existing instances of the `dspsigops` Delay block will be replaced with Simulink Delay block when there is an exact match in functionality between the two blocks. For new models, use the Delay block from the Discrete library in Simulink.

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”.

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.

Demosaic Interpolator

Demosaic Interpolator implementations, properties, and restrictions for HDL code generation

Description

The Demosaic Interpolator block is available with Vision HDL Toolbox.

For information on 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”.

Demux

Demux implementations, properties, and restrictions for HDL code generation

Description

The Demux block is available with Simulink.

For information on 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.

Deserializer1D

Deserializer1D implementations, properties, and restrictions for HDL code generation

Description

The Deserializer1D block is available with Simulink.

For information on the simulation behavior and block parameters, see `Deserializer1D`.

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”.

Digital Filter (Obsolete)

Digital Filter implementations, properties, and restrictions for HDL code generation

Description

The Digital Filter block is available with DSP System Toolbox.

For information on 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". 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`.
- If you specify **Discrete-time filter object** as the `Coefficient source`, the filter object must be one of the following:
  - `dfilt.dffir`
  - `dfilt.dfsymfir`
  - `dfilt.dfasymfir`

Distributed Arithmetic Support

Distributed Arithmetic properties **DALUTPartition** and **DARadix** are supported for the following filter structures.

Architecture	Supported FIR Structures
default	• dfilt.dffir • dfilt.dfsymfir • dfilt.dfasymfir

AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on 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.	`ceil(log2(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 this parameter to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. For more information, see 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 this parameter 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 this parameter 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. Often, you can use complex data and complex coefficients in combination. The following table shows the filter structures that support complex data or coefficients, and the permitted combinations.

Filter Structure	Complex Data	Complex Coefficients	Complex Data and Coefficients
`dfilt.dffir`	Y	Y	Y
`dfilt.dfsymfir`	Y	Y	Y
`dfilt.dfasymfir`	Y	Y	Y
`dfilt.dffirt`	Y	Y	Y
`dfilt.scalar`	Y	Y	Y
`dfilt.delay`	Y	N/A	N/A
`mfilt.cicdecim`	Y	N/A	N/A
`mfilt.cicinterp`	Y	N/A	N/A
`mfilt.firdecim`	Y	Y	N
`mfilt.firinterp`	Y	Y	N
`dfilt.df1sos`	Y	Y	Y
`dfilt.df1tsos`	Y	Y	Y
`dfilt.df2sos`	Y	Y	Y
`dfilt.df2tsos`	Y	Y	Y

Restrictions

- If you select the Digital Filter block **Discrete-time filter object** option, you must have the DSP System Toolbox software to generate code for the block.
- 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.

Direct Lookup Table (n-D)

Direct Lookup Table (n-D) implementations, properties, and restrictions for HDL code generation

Description

The Direct Lookup Table (n-D) block is available with Simulink.

For information on 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

- “Required Block Settings” on page 2-104

- "Table Data Typing and Sizing" on page 2-104

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.

Discrete FIR Filter

Discrete FIR Filter implementations, properties, and restrictions for HDL code generation

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 on the simulation behavior and block parameters, see `Discrete FIR Filter`.

Multichannel Filter Support

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks.

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 **StreamingFactor** of the subsystem to the number of channels. See the Streaming section, under "Subsystem Optimizations" on page 2-106.

Programmable Filter Support

HDL Coder supports programmable filters for Discrete FIR Filter blocks.

1. On the filter block mask, select **Input port(s)** as the coefficient source.
2. Connect the coefficient port with a vector signal.

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, select a serial **Architecture**. See "Configuring 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. See "HDL Filter Properties" on page 2-115.

Distributed Arithmetic

When you 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. See "HDL Filter Properties" on page 2-115.

Multichannel Area Reduction

To share logic between channels, you can use the subsystem-level **StreamingFactor** or the block-level **ChannelSharing** option. **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, whereas **ChannelSharing** excludes the filter from other optimizations.

Subsystem Optimizations

For the block to participate in subsystem-level optimizations, set the **Architecture** to `Fully parallel`.

Sharing

The filter block supports sharing resources within the filter and across multiple blocks in the subsystem. When you specify a **SharingFactor**, the optimization tools generate a filter implementation in HDL that shares resources using time-multiplexing. The sharing algorithm shares multipliers that have the same input and output data types. To

generate an HDL implementation that uses the minimum number of multipliers, set the **SharingFactor** to a number greater than or equal to the total number of multipliers. Alternatively, calculate a **SharingFactor** to target a particular system clock rate.

Resource sharing applies to multipliers by default. To share adders, select the check box under **Resource sharing** on the **Configuration Parameters > HDL Code Generation > Global Settings > Optimizations** dialog box.

For more information, see "Resource Sharing", and the "" on page 2-111 example.

Streaming

Streaming refers to sharing an atomic part of the design across multiple channels. To generate a streaming HDL implementation of a multichannel subsystem, set **StreamingFactor** to the number of channels in your design.

If the subsystem contains a single filter block, the block-level **ChannelSharing** option and the subsystem-level **StreamingFactor** option result in similar HDL implementations. Use **StreamingFactor** when your subsystem contains more than one filter block, or additional logic that can participate in the optimization. Set block-level **ChannelSharing** to `off` to use **StreamingFactor** at the subsystem level.

See "Area Reduction of Multichannel Filter Subsystem" on page 2-107.

Pipelining

You can enable **DistributedPipelining** at the subsystem level to allow the filter to participate in pipeline optimizations. The optimization tools operate on pipeline stages specified in **InputPipeline** and **OutputPipeline**, but they do not move design delays within the filter architecture.

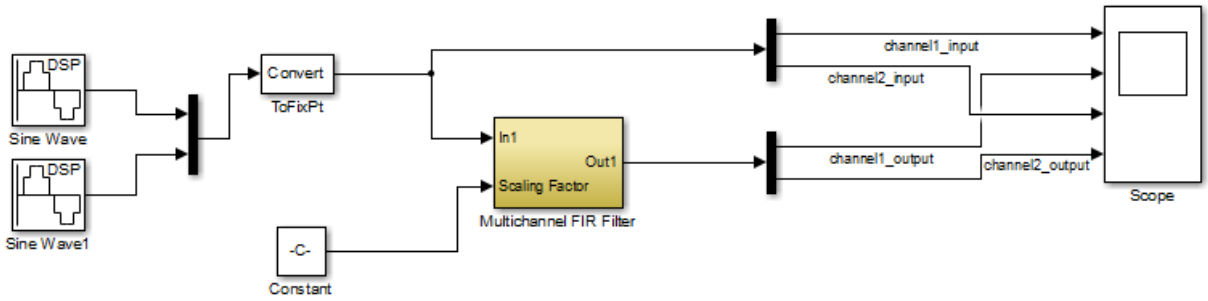
The filter block also participates in clock-rate pipelining, if enabled in **Configuration Parameters**. This feature is enabled by default.

Examples

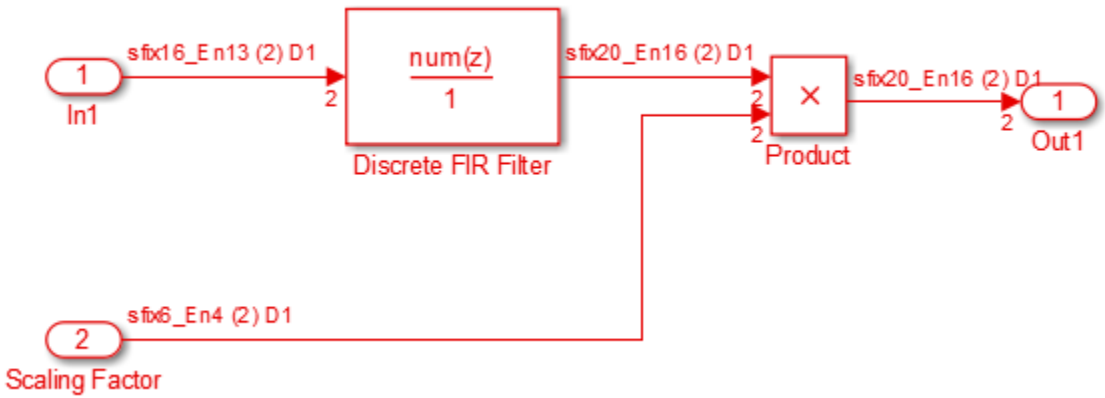
Area Reduction of Multichannel Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multichannel filter and surrounding logic, use the **StreamingFactor** HDL Coder™ optimization.

```
open_system('DiscreteFIRStreaming')
```

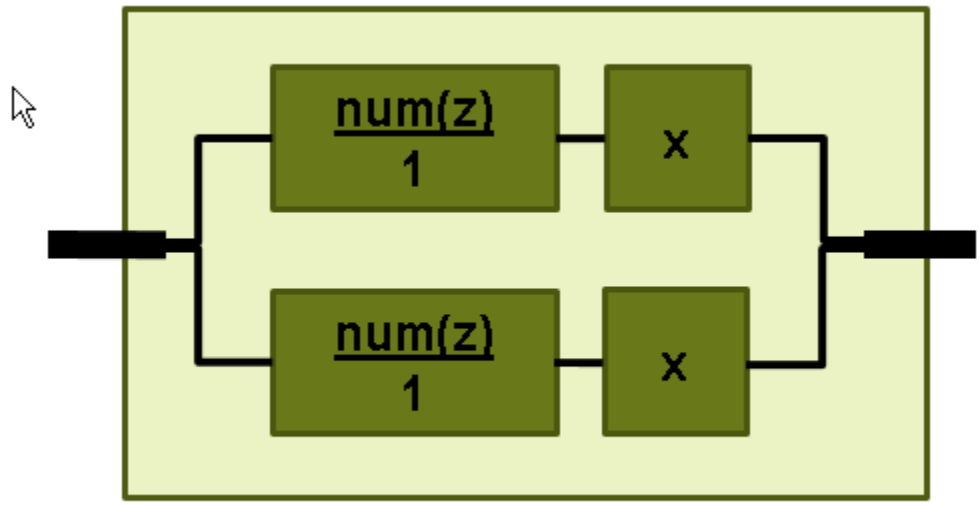


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

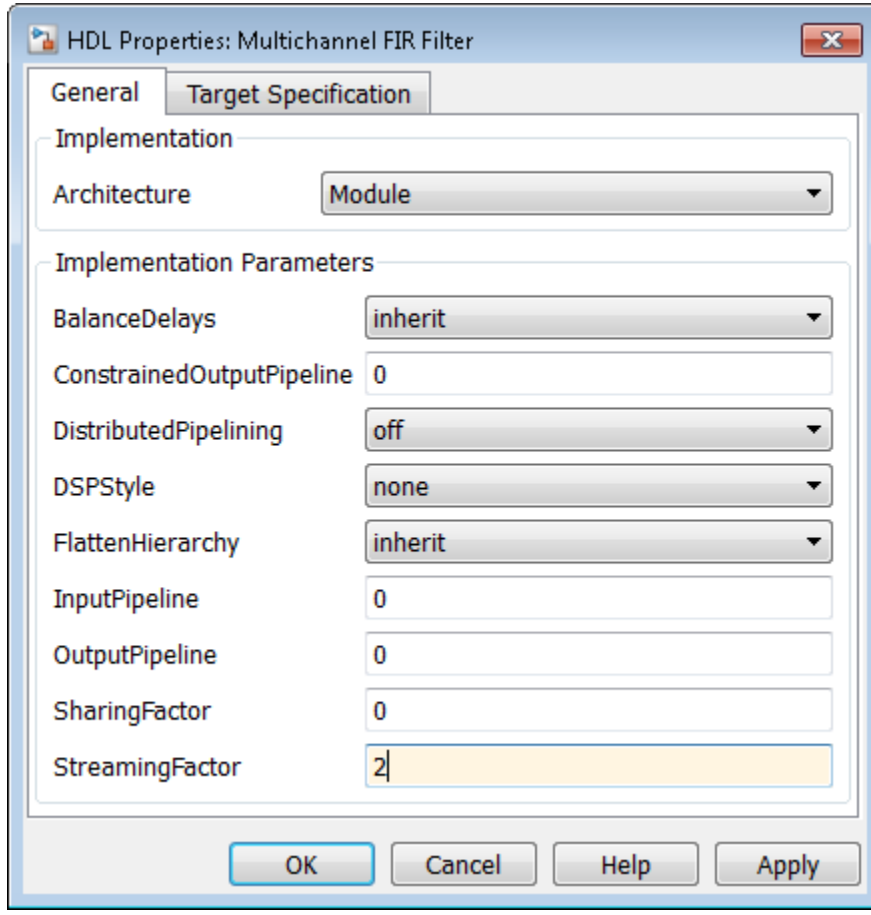


The subsystem contains a **Discrete FIR Filter** and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

The filter has 44 symmetric coefficients. With no optimizations enabled, the generated HDL code takes advantage of symmetry. The non-optimized HDL implementation uses 46 multipliers: 22 for each channel of the filter and 1 for each channel of the Product block.



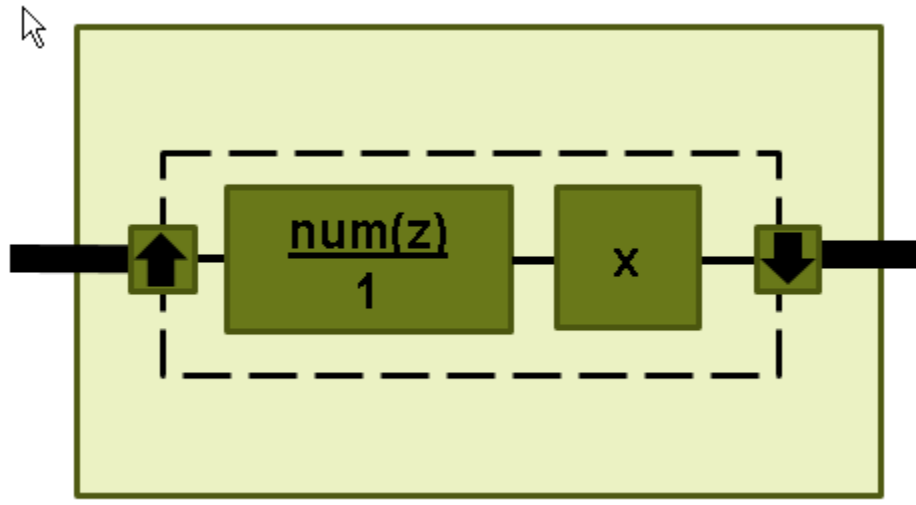
To enable streaming optimization for the **Multichannel FIR Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **StreamingFactor** to 2, because this design is a two-channel system.

To observe the effect of the optimization, on **Configuration Parameters > HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the **Multichannel FIR Filter Subsystem** and select **HDL Code > Generate HDL for Subsystem**.

With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



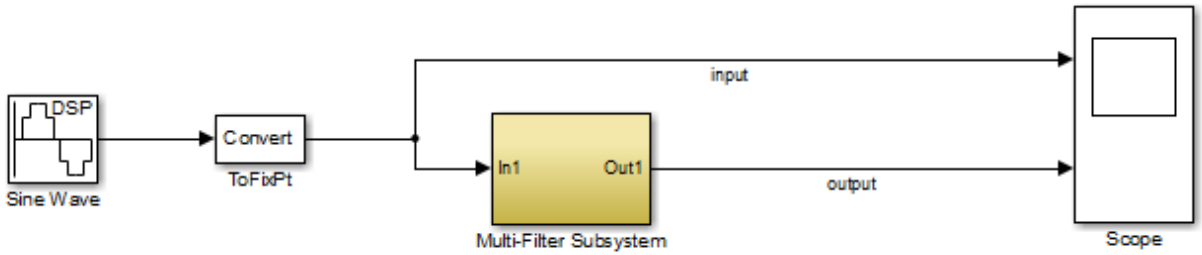
In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses 23 multipliers, compared to 46 in the non-optimized code. The multipliers in the filter kernel and subsequent scaling are shared between the channels.

Multipliers	23
Adders/Subtractors	44
Registers	92
RAMs	0
Multiplexers	28

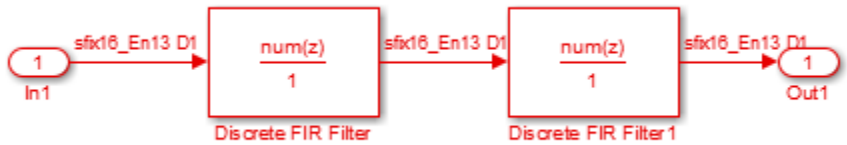
Area Reduction of Filter Subsystem

To reduce the number of multipliers in the HDL implementation of a multifilter design, use the **SharingFactor** HDL Coder™ optimization.

```
open_system('DiscreteFIRSharing')
```



The model includes a sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

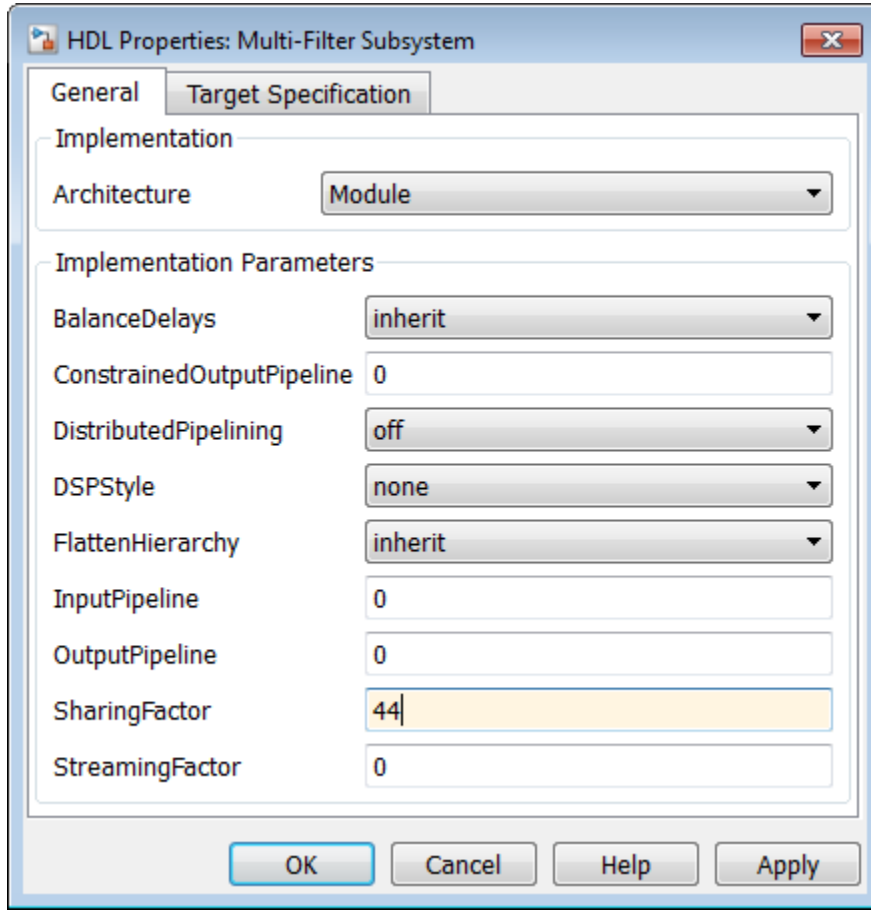


The subsystem contains a lowpass and a highpass filter, each implemented with a **Discrete FIR Filter** block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

Each filter has 43 symmetric coefficients. With no optimizations enabled, the generated HDL code takes advantage of symmetry. The non-optimized HDL implementation of the subsystem uses 44 multipliers.

Multipliers	44
Adders/Subtractors	84
Registers	84
RAMs	0
Multiplexers	0

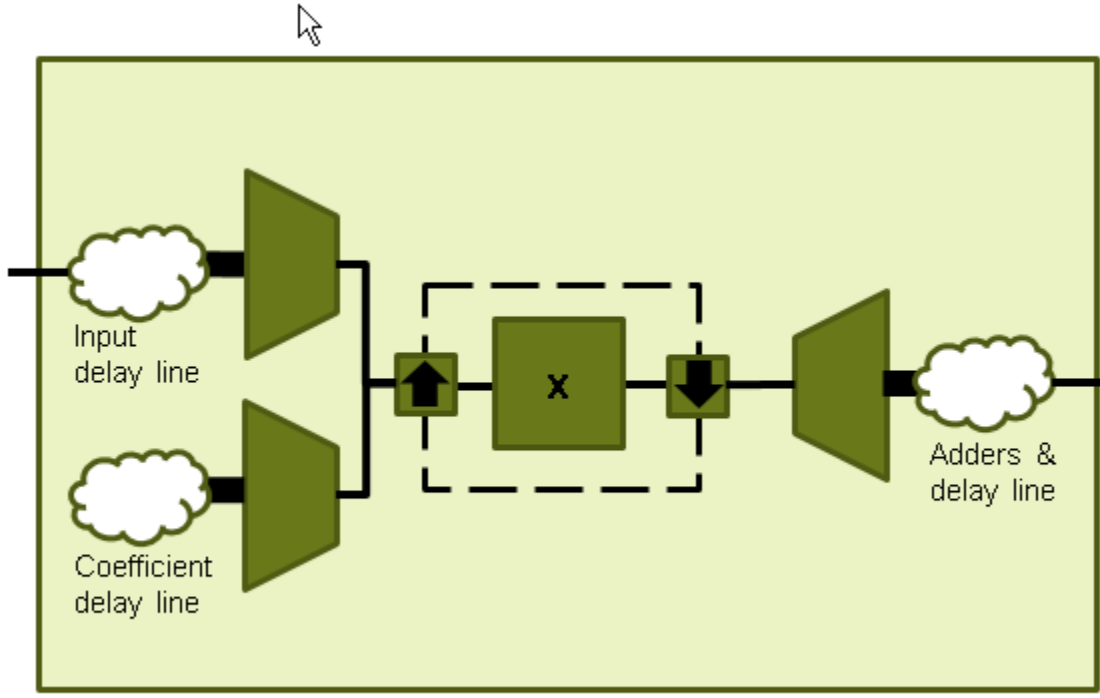
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **SharingFactor** to 44 to reduce the design to a single multiplier.

To observe the effect of the optimization, on **Configuration Parameters > HDL Code Generation**, select **Generate resource utilization report** and **Generate optimization report**. Then, to generate HDL code, right-click the **Multi-Filter Subsystem** and select **HDL Code > Generate HDL for Subsystem**.

With the **SharingFactor** applied, the subsystem upsamples the rate by 44 to share a single multiplier for all the coefficients.



In the **Code Generation Report** window, click **High-level Resource Report**. The generated HDL code now uses one multiplier.

Multipliers	1
Adders/Subtractors	86
Registers	198
RAMs	0
Multiplexers	7

- Generate HDL Code for FIR Programmable Filter

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 this parameter to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. For more information, see 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 this parameter 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 this parameter 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 Coder does not support unsigned inputs for the Discrete FIR Filter block.
- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- The coder does not support the following options of the Discrete FIR Filter block:
  - **Filter Structure**: `Lattice MA`
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, the **CoeffMultipliers** property 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 as `csd` or `factored-csd`

Discrete PID Controller

Discrete PID Controller implementations, properties, and restrictions for HDL code generation

Description

The Discrete PID Controller block is available with Simulink.

For information on 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**.

Discrete Transfer Fcn

Discrete Transfer Fcn implementations, properties, and restrictions for HDL code generation

Description

The Discrete Transfer Fcn block is available with Simulink.

For information on the simulation behavior and block parameters, see `Discrete Transfer Fcn`.

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 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

Eye Diagram

Eye Diagram implementations, properties, and restrictions for HDL code generation

Description

The Eye Diagram block is available with Communications System Toolbox.

For information on 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.

Discrete-Time Integrator

Discrete-Time Integrator implementations, properties, and restrictions for HDL code generation

Description

The Discrete-Time Integrator block is available with Simulink.

For information on the simulation behavior and block parameters, see `Discrete-Time Integrator`.

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

- 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`.
- Width of input and output signals must not exceed 32 bits.

Dilation

Dilation implementations, properties, and restrictions for HDL code generation

Description

The Dilation block is available with Vision HDL Toolbox.

For information on 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”.

Display

Display implementations, properties, and restrictions for HDL code generation

Description

The Display block is available with Simulink.

For information on 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.

Divide

Divide implementations, properties, and restrictions for HDL code generation

Description

The Divide block is available with Simulink.

For information on the simulation behavior and block parameters, see `Divide`.

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 2-127.

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. View the generated model or validation model to see the added latency. 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.
`ReciprocalRsqrtBasedNewton`	`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.
`ReciprocalRsqrtBasedNewtonSingleRate`	`Iterations`	Signed input: (`Iteration` * 4) + 8 Unsigned input: (`Iteration` * 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:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i(1.5 - 0.5ax_i^2)$$

`ReciprocalRsqrtBasedNewton` and `ReciprocalRsqrtBasedNewtonSingleRate` implement the Newton-Raphson method with:

$$f(x) = \frac{1}{x^2} - 1$$

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".

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

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.

DocBlock

DocBlock implementations, properties, and restrictions for HDL code generation

Description

The DocBlock block is available with Simulink.

For information on 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.

Related Examples

- "Generate Code with Annotations or Comments"
- "Integrate Custom HDL Code Using DocBlock"

Dot Product

Dot Product implementations, properties, and restrictions for HDL code generation

Description

The Dot Product block is available with Simulink.

For information on 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. Both input signals must be either row vectors or column vectors. A mix of row and column vector inputs is not supported.

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”.

Downsample

Downsample implementations, properties, and restrictions for HDL code generation

Description

The Downsample block is available with DSP System Toolbox.

For information on 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

- For **Frame based mode**, select `Maintain input frame size`.
- For **Sample based mode**, select `Allow multirate`.

With these block settings, if **Sample offset** is set to 0, **Initial conditions** has no effect on generated code.

DSP Constant (Obsolete)

DSP Constant (Obsolete) implementations, properties, and restrictions for HDL code generation

Description

HDL support for the DSP Constant (Obsolete) block will be removed in a future release. Use the `Constant` block instead.

Dual Port RAM

Dual Port RAM implementations, properties, and restrictions for HDL code generation

Description

The Dual Port RAM block is available with Simulink.

For information on the simulation behavior and block parameters, see `Dual Port RAM`.

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 2-138.

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.

Dual Rate Dual Port RAM

Dual Rate Dual Port RAM implementations, properties, and restrictions for HDL code generation

Description

The Dual Rate Dual Port RAM block is available with Simulink.

For information on the simulation behavior and block parameters, see `Dual Rate Dual Port RAM`.

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`.

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 2-141.

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.

Edge Detector

Edge Detector implementations, properties, and restrictions for HDL code generation

Description

The Edge Detector block is available with Vision HDL Toolbox.

For information on 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”.

Enable

Enable implementations, properties, and restrictions for HDL code generation

Description

The Enable block is available with Simulink.

For information on 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”.

See Also

`Enabled Subsystem`

Enabled Subsystem

Enabled Subsystem implementations, properties, and restrictions for HDL code generation

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 "Create an Enabled Subsystem" in the Simulink documentation.

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`	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. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of 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.

HDL Block Properties

General

BalanceDelays

Delay balancing. 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".

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. 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
```

See Also

`Enable` | `Subsystem`

Enumerated Constant

Enumerated Constant implementations, properties, and restrictions for HDL code generation

Description

The Enumerated Constant block is available with Simulink.

For information on 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”.

Error Rate Calculation

Error Rate Calculation implementations, properties, and restrictions for HDL code generation

Description

The Error Rate Calculation block is available with Communications System Toolbox.

For information on 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.

Erosion

Erosion implementations, properties, and restrictions for HDL code generation

Description

The Erosion block is available with Vision HDL Toolbox.

For information on 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”.

Extract Bits

Extract Bits implementations, properties, and restrictions for HDL code generation

Description

The Extract Bits block is available with Simulink.

For information on 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”.

FFT HDL Optimized

FFT HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The FFT HDL Optimized block is available with DSP System Toolbox.

For information on 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”.

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”.

FIR Decimation

FIR Decimation implementations, properties, and restrictions for HDL code generation

Description

The FIR Decimation block is available with DSP System Toolbox.

For information on the simulation behavior and block parameters, see `FIR Decimation`.

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**.

HDL Architecture

Observe the following limitations for FIR Decimation filters:

- HDL Coder supports `SerialPartition` only for the FIR Direct Form structure.
- Accumulator reuse is not supported.

Distributed Arithmetic Support

Distributed Arithmetic properties **DALUTPartition** and **DARadix** are supported for the following filter structures.

Architecture	Supported FIR Structures
default, Distributed Arithmetic (DA)	Direct form

AddPipelineRegisters Support

When you use **AddPipelineRegisters**, registers are placed based on filter structure. The pipeline register placement determines the latency.

Filter Structure	Pipeline Register Placement	Latency (clock cycles)
Direct form	One pipeline register is added between levels of a tree-based adder, and one is added after the products.	`ceil(log2(NZ))`. `NZ` is the number of non-zero coefficients.
Direct form transposed	No pipelining is added.	0

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 this parameter to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. For more information, see 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 this parameter 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:
  - `Slope and Bias scaling`
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, the **CoeffMultipliers** property is hidden from the HDL Block Properties dialog box.

FIR Interpolation

FIR Interpolation implementations, properties, and restrictions for HDL code generation

Description

The FIR Interpolation block is available with DSP System Toolbox.

For information on 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 filter structure. The pipeline register placement determines the latency.

Pipeline Register Placement	Latency (clock cycles)
A pipeline register is added between levels of a tree-based adder.	`ceil(log2(PL))-1`.

Pipeline Register Placement	Latency (clock cycles)
	`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 this parameter to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. For more information, see 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 this parameter 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, the **CoeffMultipliers** property is hidden from the HDL Block Properties dialog box.

FIR Rate Conversion HDL Optimized

FIR Rate Conversion HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The FIR Rate Conversion HDL Optimized block is available with DSP System Toolbox.

For information on 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”.

Floating Scope

Floating Scope implementations, properties, and restrictions for HDL code generation

Description

The Floating Scope block is available with Simulink.

For information on 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.

Frame Conversion

Frame Conversion implementations, properties, and restrictions for HDL code generation

Description

The Frame Conversion block is available with DSP System Toolbox.

For information on 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.

From

From implementations, properties, and restrictions for HDL code generation

Description

The From block is available with Simulink.

For information on 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”.

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”.

Gain

Gain implementations, properties, and restrictions for HDL code generation

Description

The Gain block is available with Simulink.

For information on 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

`ConstMultiplierOptimizatio`	Description
`none`(*Default*)	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.
`fcsd`	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.

ConstMultiplierOptimizatio	Description
auto	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 auto, 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

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”.

Complex Data Support

This block supports code generation for complex signals.

Gamma Corrector

Gamma Corrector implementations, properties, and restrictions for HDL code generation

Description

The Gamma Corrector block is available with Vision HDL Toolbox.

For information on 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”.

General CRC Generator HDL Optimized

General CRC Generator HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The General CRC Generator HDL Optimized block is available with Communications System Toolbox.

For information on 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”.

General CRC Syndrome Detector HDL Optimized

General CRC Syndrome Detector HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The General CRC Syndrome Detector HDL Optimized block is available with Communications System Toolbox.

For information on 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”.

General Multiplexed Deinterleaver

General Multiplexed Deinterleaver implementations, properties, and restrictions for HDL code generation

Description

The General Multiplexed Deinterleaver block is available with Communications System Toolbox.

For information on 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”.

General Multiplexed Interleaver

General Multiplexed Interleaver implementations, properties, and restrictions for HDL code generation

Description

The General Multiplexed Interleaver block is available with Communications System Toolbox.

For information on 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”.

Goto

Goto implementations, properties, and restrictions for HDL code generation

Description

The Goto block is available with Simulink.

For information on the simulation behavior and block parameters, see `Goto`.

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”.

Ground

Ground implementations, properties, and restrictions for HDL code generation

Description

The Ground block is available with Simulink.

For information on 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.

HDL Cosimulation

HDL Cosimulation implementations, properties, and restrictions for HDL code generation

Description

The HDL Cosimulation block is available with HDL Verifier.

For information on 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 on timing, latency, data typing, frame-based processing, and other issues when setting up an HDL cosimulation, see the “Define HDL Cosimulation Block Interface ” section of the HDL Verifier documentation.

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".

More About

- "Generate a Cosimulation Model"

HDL Counter

HDL Counter implementations, properties, and restrictions for HDL code generation

Description

The HDL Counter block is available with Simulink.

For information on the simulation behavior and block parameters, see `HDL Counter`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

HDL FIFO

HDL FIFO implementations, properties, and restrictions for HDL code generation

Description

The HDL FIFO block is available with Simulink.

For information on the simulation behavior and block parameters, see `HDL FIFO`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

HDL Minimum Resource FFT

HDL Minimum Resource FFT implementations, properties, and restrictions for HDL code generation

Description

The HDL Minimum Resource FFT block is available with DSP System Toolbox.

For information on 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”.

HDL Reciprocal

HDL Reciprocal implementations, properties, and restrictions for HDL code generation

Description

The HDL Reciprocal block is available with Simulink.

For information on the simulation behavior and block parameters, see `HDL Reciprocal`.

The Newton-Raphson iterative method:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + (x_i - a{x_i}^2)$$

HDL Reciprocal implements the Newton-Raphson method with:

$$f(x) = \frac{1}{x} - a$$

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. See “Generated Model and Validation Model”.

Architecture	Additional cycles of latency	Description
`ReciprocalNewton` (default)	`Iterations` + 1	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

Architecture	Additional cycles of latency	Description
		10. If `Iterations` is outside the recommended range, HDL Coder displays a message.
`ReciprocalNewtonSingleRate`	(`Iterations` * 2) + 1	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.

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”.

HDL Streaming FFT

HDL Streaming FFT implementations, properties, and restrictions for HDL code generation

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”.

Histogram

Histogram implementations, properties, and restrictions for HDL code generation

Description

The Histogram block is available with Vision HDL Toolbox.

For information on 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”.

Hit Crossing

Hit Crossing implementations, properties, and restrictions for HDL code generation

Description

The Hit Crossing block is available with Simulink.

For information on 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`.

IFFT HDL Optimized

IFFT HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The IFFT HDL Optimized block is available with DSP System Toolbox.

For information on 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”.

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”.

Image Filter

Image Filter implementations, properties, and restrictions for HDL code generation

Description

The Image Filter block is available with Vision HDL Toolbox.

For information on 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”.

Image Statistics

Image Statistics implementations, properties, and restrictions for HDL code generation

Description

The Image Statistics block is available with Vision HDL Toolbox.

For information on 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”.

Increment Real World

Increment Real World implementations, properties, and restrictions for HDL code generation

Description

The Increment Real World block is available with Simulink.

For information on 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”.

Increment Stored Integer

Increment Stored Integer implementations, properties, and restrictions for HDL code generation

Description

The Increment Stored Integer block is available with Simulink.

For information on 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”.

Index Vector

Index Vector implementations, properties, and restrictions for HDL code generation

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`.

Inport

Inport implementations, properties, and restrictions for HDL code generation

Description

The Inport block is available with Simulink.

For information on the simulation behavior and block parameters, see `Inport`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

BidirectionalPort

BidirectionalPort Setting	Description
`on`	Specify the port as bidirectional. The following requirements apply: • The port must be in a Subsystem block with black box implementation. • There must also be no logic between the bidirectional port and the corresponding top-level DUT subsystem port. For more information, see “Specify Bidirectional Ports”.
`off` (default)	Do not specify the port as bidirectional.

Target Specification

IOInterface

Target platform interface type for DUT ports, specified as a string. 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 string. 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')
```

Related Examples

- “Save Target Hardware Settings in Model”

Integer-Input RS Encoder HDL Optimized

Integer-Input RS Encoder HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The Integer-Input RS Encoder HDL Optimized block is available with Communications System Toolbox.

For information on 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”.

Integer-Output RS Decoder HDL Optimized

Integer-Output RS Decoder HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The Integer-Output RS Decoder HDL Optimized block is available with Communications System Toolbox.

For information on 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”.

LMS Filter

LMS Filter implementations, properties, and restrictions for HDL code generation

Description

The LMS Filter block is available with DSP System Toolbox.

For information on 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 `ceil(log2(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.

Logical Operator

Logical Operator implementations, properties, and restrictions for HDL code generation

Description

The Logical Operator block is available with Simulink.

For information on 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”.

Lookup Table

Lookup Table implementations, properties, and restrictions for HDL code generation

Description

The Lookup Table block is available with Vision HDL Toolbox.

For information on 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”.

M-PSK Demodulator Baseband

M-PSK Demodulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The M-PSK Demodulator Baseband block is available with Communications System Toolbox.

For information on 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”.

M-PSK Modulator Baseband

M-PSK Modulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The M-PSK Modulator Baseband block is available with Communications System Toolbox.

For information on 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”.

Magnitude-Angle to Complex

Magnitude-Angle to Complex implementations, properties, and restrictions for HDL code generation

Description

The Magnitude-Angle to Complex block is available with Simulink.

For information on 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. View the generated model or validation model to see the added latency. 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`.

Math Function

Math Function implementations, properties, and restrictions for HDL code generation

Description

The Math Function block is available with Simulink.

For information on 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. View the generated model or validation model to see the added latency. See “Generated Model and Validation Model”.

Architecture	Parameters	Additional cycles of latency	Description
`Math` (default) `Reciprocal`	None	0	Compute reciprocal as `1/N`, using

Architecture	Parameters	Additional cycles of latency	Description
			the HDL divide (/) operator to implement the division.
`ReciprocalRsqrtBasedNewton`	`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.
`ReciprocalRsqrtBasedNewtonSingleRate`	`Iterations`		

Architecture	Parameters	Additional cycles of latency	Description
		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 from 2 through 10. If `Iterations` 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

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

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.

MATLAB Function

MATLAB Function implementations, properties, and restrictions for HDL code generation

Description

The MATLAB Function block is available with Simulink.

For information on the simulation behavior and block parameters, see `MATLAB Function`.

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 string, with spaces separating the variables.

Complex Data Support

This block supports code generation for complex signals.

See also “Complex Data Type Support”.

Restrictions

- If the block contains a System object™, block inputs cannot have non-discrete (constant or `Inf`) sample time.

For the MATLAB language subset supported for HDL code generation from a MATLAB Function block, see:

- “Data Types and Scope”
- “Operators”
- “Control Flow Statements”
- “Persistent Variables”
- “Persistent Array Variables”
- “HDL Code Generation for System Objects”
- “Complex Data Type Support”
- “Fixed-Point Bitwise Functions”
- “Fixed-Point Run-Time Library Functions”

Related Examples

- “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”

More About

- “HDL Applications for the MATLAB Function Block”

MATLAB System

MATLAB System implementations, properties, and restrictions for HDL code generation

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 on the Simulink behavior and block parameters, see `MATLAB System`.

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 string, 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”.

Related Examples

- “Generate Code for User-Defined System Objects”

More About

- “HDL Code Generation for System Objects”

Matrix Concatenate

Matrix Concatenate implementations, properties, and restrictions for HDL code generation

Description

The Matrix Concatenate block is the Vector Concatenate block with **Mode** set to `Multidimensional array`. For HDL code generation information, see `Vector Concatenate`.

Matrix Viewer

Matrix Viewer implementations, properties, and restrictions for HDL code generation

Description

The Matrix Viewer block is available with DSP System Toolbox.

For information on 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.

Maximum

Maximum implementations, properties, and restrictions for HDL code generation

Description

The Maximum block is available with DSP System Toolbox.

For information on the simulation behavior and block parameters, see `Maximum`.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. 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”.

Median Filter

Median Filter implementations, properties, and restrictions for HDL code generation

Description

The Median Filter block is available with Vision HDL Toolbox.

For information on 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”.

Memory

Memory implementations, properties, and restrictions for HDL code generation

Description

The Memory block is available with Simulink.

For information on 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.

Message Viewer

Message Viewer implementations, properties, and restrictions for HDL code generation

Description

The Message Viewer block is available with Stateflow.

For information on the simulation behavior and block parameters, see `Message 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`

Minimum

Minimum implementations, properties, and restrictions for HDL code generation

Description

The Minimum block is available with DSP System Toolbox.

For information on the simulation behavior and block parameters, see `Minimum`.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. 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”.

MinMax

MinMax implementations, properties, and restrictions for HDL code generation

Description

The MinMax block is available with Simulink.

For information on the simulation behavior and block parameters, see `MinMax`.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. 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”.

Model

Model implementations, properties, and restrictions for HDL code generation

Description

The Model block is available with Simulink. For information on 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".
`BlackBox`	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. For more information, see "Generate Black Box Interface for Referenced Model".

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

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Restrictions

When you generate HDL code for referenced models, the following limitations apply:

- You must set the block parameters for the Model block to their default values.
- If multiple model references refer to the same model, their HDL block properties must be the same.
- Referenced models cannot be protected models.
- Hierarchical distributed pipelining must be disabled.

HDL Coder cannot move registers across a model reference. Therefore, referenced models may inhibit the following optimizations:

- Distributed pipelining
- Constrained output pipelining

The coder cannot apply the streaming optimization to a model reference.

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.

More About

- "Model Referencing for HDL Code Generation"
- "Generate Black Box Interface for Referenced Model"
- "Generate Parameterized Code For Referenced Models"

Model Info

Model Info implementations, properties, and restrictions for HDL code generation

Description

The Model Info block is available with Simulink.

For information on the simulation behavior and block parameters, see `Model Info`.

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”.

Model Variants

Model Variants implementations, properties, and restrictions for HDL code generation

Description

The Model Variants block is a version of the Model block. For HDL code generation information, see `Model`.

Multiply-Add

Multiply-Add implementations, properties, and restrictions for HDL code generation

Description

The Multiply-Add block is available with Simulink.

For information on the simulation behavior and block parameters, see `Multiply-Add`.

Hardware Mapping

When you want to map a combined multiply-add operation to a DSP unit in your target hardware, use the Multiply-Add block.

Different synthesis tools map operations differently to hardware. To map to a DSP unit, specify the `SynthesisTool` property for your model.

When you generate HDL code for your model, HDL Coder chooses an implementation for the multiply-add operation that your synthesis tool can map to a DSP unit.

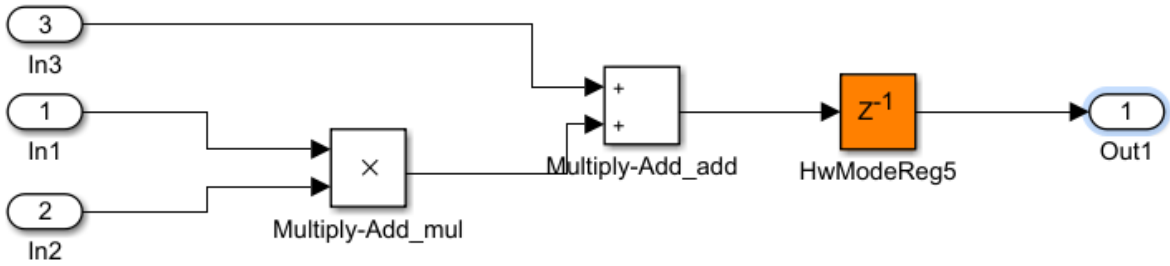
Note: Some DSP units do not have the multiply-add capability. Check the documentation for your hardware to see if it has the multiply-add capability.

Pipeline Depth

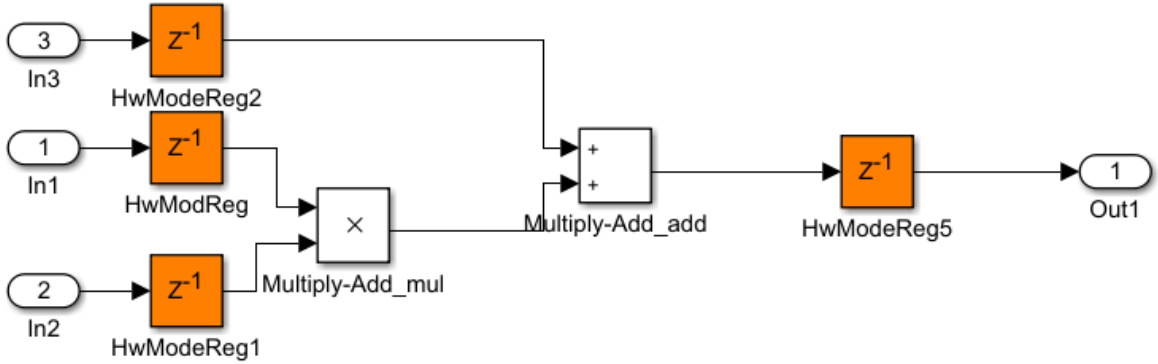
You can specify pipeline depth for a Multiply-Add block with fixed-point inputs. For floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert the pipeline registers.

The following diagrams show the hardware implementation with different synthesis tools and **PipelineDepth** settings.

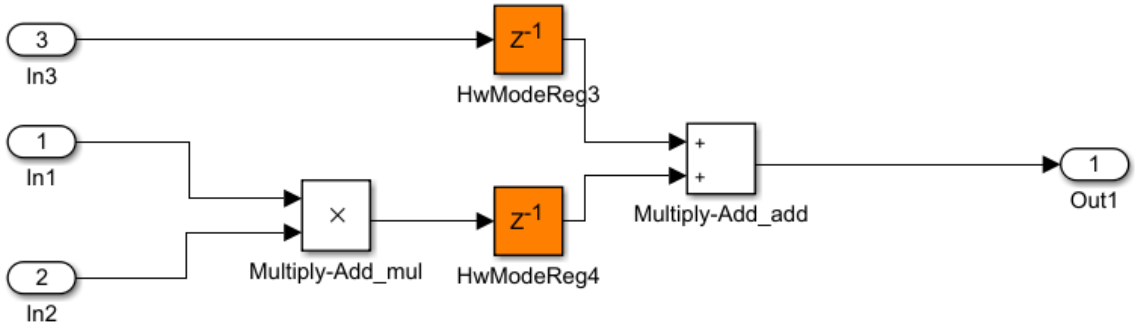
Altera Hardware with PipelineDepth = 1



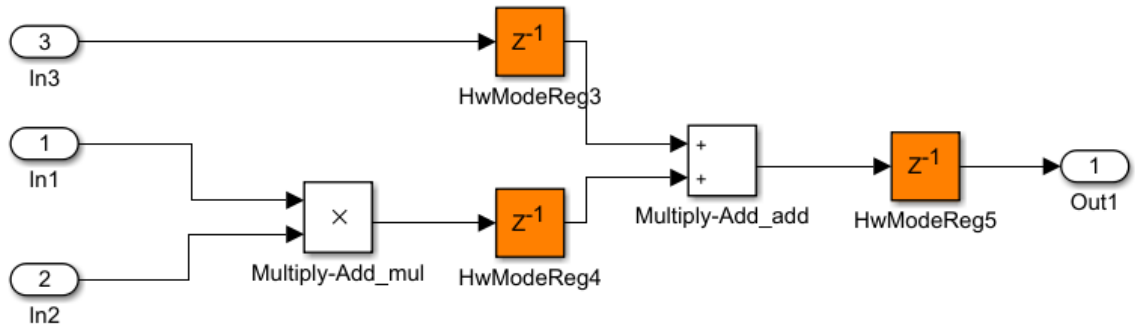
Altera Hardware with PipelineDepth = 2



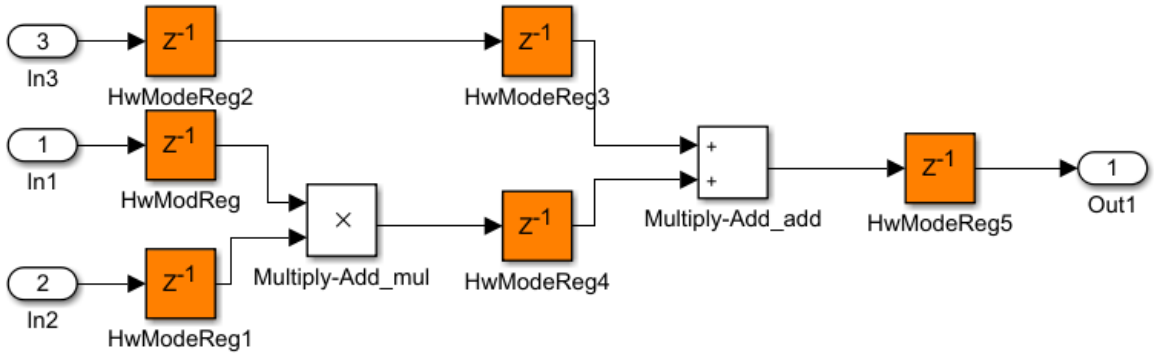
Xilinx Hardware with PipelineDepth = 1



Xilinx Hardware with PipelineDepth = 2



Xilinx Hardware with PipelineDepth = 3



HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

PipelineDepth

Number of pipeline stages, specified as `auto` or as an integer between 0 and 3. The default is `auto`.

If you specify `auto`, the coder determines the number of pipeline stages based on your synthesis tool.

For Altera hardware targets, the maximum pipeline depth is 2.

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 `MaxOversampling` = 1 and `MaxComputationLatency` > 1, you cannot share Multiply-Add blocks with nonzero **PipelineDepth** with the resource sharing optimization.
- 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.

See Also

`Multiply-Add` | `SynthesisTool`

Multiport Selector

Multiport Selector implementations, properties, and restrictions for HDL code generation

Description

The Multiport Selector block is available with DSP System Toolbox.

For information on 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.

Multiport Switch

Multiport Switch implementations, properties, and restrictions for HDL code generation

Description

The Multiport Switch block is available with Simulink.

For information on the simulation behavior and block parameters, see `Multiport 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

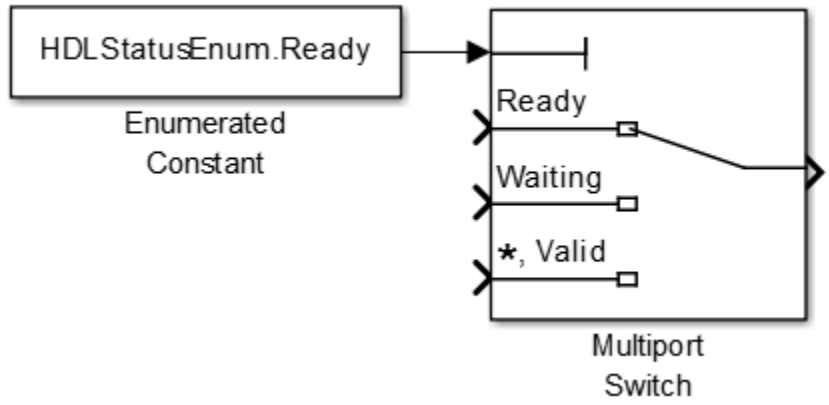
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Complex Data Support

This block supports code generation for complex signals.

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.



Mux

Mux implementations, properties, and restrictions for HDL code generation

Description

The Mux block is available with Simulink.

For information on 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.

n-D Lookup Table

n-D Lookup Table implementations, properties, and restrictions for HDL code generation

Description

The n-D Lookup Table block is available with Simulink.

For information on 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

Required Block Settings

- **Number of table dimensions**: HDL Coder supports a maximum dimension of 2.
- **Index search method**: Select `Evenly spaced points`.
- **Extrapolation method**: The coder supports only `Clip`. The coder does not support extrapolation beyond the table bounds.
- **Interpolation method**: The coder supports only `Flat` or `Linear`.
- **Diagnostic for out-of-range input**: Select `Error`. If you select other options, the coder displays a warning.
- **Use last table value for inputs at or above last breakpoint**: Select this check box.
- **Require all inputs to have the same data type**: Select this check box.
- **Fraction**: Select `Inherit: Inherit via internal rule`.
- **Integer rounding mode**: 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.

NCO

NCO implementations, properties, and restrictions for HDL code generation

Description

HDL support for the NCO block will be removed in a future release. Use the `NCO HDL Optimized` block instead.

NCO HDL Optimized

NCO HDL Optimized implementations, properties, and restrictions for HDL code generation

Description

The NCO HDL Optimized block is available with DSP System Toolbox.

For information on 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.

Opening

Opening implementations, properties, and restrictions for HDL code generation

Description

The Opening block is available with Vision HDL Toolbox.

For information on 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”.

Outport

Outport implementations, properties, and restrictions for HDL code generation

Description

The Outport block is available with Simulink.

For information on the simulation behavior and block parameters, see `Outport`.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

BidirectionalPort

BidirectionalPort Setting	Description
`on`	Specify the port as bidirectional. The following requirements apply: • The port must be in a Subsystem block with black box implementation. • There must also be no logic between the bidirectional port and the corresponding top-level DUT subsystem port. For more information, see “Specify Bidirectional Ports”.
`off` (default)	Do not specify the port as bidirectional.

Target Specification

IOInterface

Target platform interface type for DUT ports, specified as a string. 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 string. 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')
```

Related Examples

- “Save Target Hardware Settings in Model”

PN Sequence Generator

PN Sequence Generator implementations, properties, and restrictions for HDL code generation

Description

The PN Sequence Generator block is available with Communications System Toolbox.

For information on 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

Inputs:

- 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:

- Outputs of type `double` are not supported for HDL code generation. All other output types (including bit packed outputs) are supported.

Prelookup

Prelookup implementations, properties, and restrictions for HDL code generation

Description

The Prelookup block is available with Simulink.

For information on 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 2-258
- “Table Data Typing and Sizing” on page 2-258

Required Block Settings

- **Breakpoint data**: For **Source**, select `Dialog`.
- **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.

Product

Product implementations, properties, and restrictions for HDL code generation

Description

The Product block is available with Simulink.

For information on 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

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”.

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`.

Product of Elements

Product of Elements implementations, properties, and restrictions for HDL code generation

Description

The Product of Elements block is available with Simulink.

For information on 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. View the generated model or validation model to see the added latency. 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”.

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”.

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 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`.

QPSK Demodulator Baseband

QPSK Demodulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The QPSK Demodulator Baseband block is available with Communications System Toolbox.

For information on 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”.

QPSK Modulator Baseband

QPSK Modulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The QPSK Modulator Baseband block is available with Communications System Toolbox.

For information on 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”.

Raised Cosine Receive Filter

Raised Cosine Receive Filter implementations, properties, and restrictions for HDL code generation

Description

The Raised Cosine Receive Filter is available with Communications System Toolbox.

For information on the simulation behavior and block parameters, see `Raised Cosine Receive Filter`.

This block is a subsystem that contains a FIR Interpolation block. You can look under the mask and set **HDL Properties** on the filter block. See `FIR Interpolation`.

Raised Cosine Transmit Filter

Raised Cosine Transmit Filter implementations, properties, and restrictions for HDL code generation

Description

The Raised Cosine Transmit Filter is available with Communications System Toolbox.

For information on the simulation behavior and block parameters, see `Raised Cosine Transmit Filter`.

This block is a subsystem that contains a FIR Decimation block. You can look under the mask and set **HDL Properties** on the filter block. See `FIR Decimation`.

Rate Transition

Rate Transition implementations, properties, and restrictions for HDL code generation

Description

The Rate Transition block is available with Simulink.

For information on the simulation behavior and block parameters, see `Rate Transition`.

Best Practices

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.

Real-Imag to Complex

Real-Imag to Complex implementations, properties, and restrictions for HDL code generation

Description

The Real-Imag to Complex block is available with Simulink.

For information on 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.

Reciprocal Sqrt

Reciprocal Sqrt implementations, properties, and restrictions for HDL code generation

Description

The Reciprocal Sqrt block is available with Simulink.

For information on the simulation behavior and block parameters, see `Reciprocal Sqrt`.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. See "Generated Model and Validation Model".

Architecture	Additional cycles of latency	Description
`SqrtFunction` (default) `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

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”.

Restrictions

- In the Block Parameters dialog box, in the **Algorithm** tab, for **Method**, you must select `Newton-Raphson`.
- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

Rectangular QAM Demodulator Baseband

Rectangular QAM Demodulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The Rectangular QAM Demodulator Baseband block is available with Communications System Toolbox.

For information on 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.

Rectangular QAM Modulator Baseband

Rectangular QAM Modulator Baseband implementations, properties, and restrictions for HDL code generation

Description

The Rectangular QAM Modulator Baseband block is available with Communications System Toolbox.

For information on 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.

Relational Operator

Relational Operator implementations, properties, and restrictions for HDL code generation

Description

The Relational Operator block is available with Simulink.

For information on the simulation behavior and block parameters, see `Relational 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”.

Complex Data Support

The ~= and == operators are supported for code generation.

Relay

Relay implementations, properties, and restrictions for HDL code generation

Description

The Relay block is available with Simulink.

For information on 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”.

Repeat

Repeat implementations, properties, and restrictions for HDL code generation

Description

The Repeat block is available with DSP System Toolbox.

For information on 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

For **Frame based mode**, select `Maintain input frame size`. When the Upsample block is in this mode, **Initial conditions** has no effect on generated code.

Reshape

Reshape implementations, properties, and restrictions for HDL code generation

Description

The Reshape block is available with Simulink.

For information on 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.

Sample and Hold

Sample and Hold implementations, properties, and restrictions for HDL code generation

Description

The Sample and Hold block is available with DSP System Toolbox.

For information on 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”.

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 2-328 for how to match rates.

Saturation

Saturation implementations, properties, and restrictions for HDL code generation

Description

The Saturation block is available with Simulink.

For information on 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”.

Saturation Dynamic

Saturation Dynamic implementations, properties, and restrictions for HDL code generation

Description

The Saturation Dynamic block is available with Simulink.

For information on 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”.

Scope

Scope implementations, properties, and restrictions for HDL code generation

Description

The Scope block is available with Simulink.

For information on 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.

Selector

Selector implementations, properties, and restrictions for HDL code generation

Description

The Selector block is available with Simulink.

For information on the simulation behavior and block parameters, see `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.

Serializer1D

Serializer1D implementations, properties, and restrictions for HDL code generation

Description

The Serializer1D block is available with Simulink.

For information on the simulation behavior and block parameters, see `Serializer1D`.

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”.

Shift Arithmetic

Shift Arithmetic implementations, properties, and restrictions for HDL code generation

Description

The Shift Arithmetic block is available with Simulink.

For information on the simulation behavior and block parameters, see `Shift Arithmetic`.

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

In the Function Block Parameters dialog box, for **Bits to shift**, you must set **Source** to **Dialog**. The **Input port** option is not supported for HDL code generation.

Sign

Sign implementations, properties, and restrictions for HDL code generation

Description

The Sign block is available with Simulink.

For information on 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”.

Signal Conversion

Signal Conversion implementations, properties, and restrictions for HDL code generation

Description

The Signal Conversion block is available with Simulink.

For information on 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.

Signal Specification

Signal Specification implementations, properties, and restrictions for HDL code generation

Description

The Signal Specification block is available with Simulink.

For information on 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.

Simple Dual Port RAM

Simple Dual Port RAM implementations, properties, and restrictions for HDL code generation

Description

The Simple Dual Port RAM block is available with Simulink.

For information on the simulation behavior and block parameters, see `Simple Dual Port RAM`.

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 2-293.

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.

Sine

Sine implementations, properties, and restrictions for HDL code generation

Description

The Sine block is available with Simulink.

For information on 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”.

Restrictions

This block does not have restrictions for HDL code generation.

Sine Wave

Sine Wave implementations, properties, and restrictions for HDL code generation

Description

The Sine Wave block is available with DSP System Toolbox.

For information on 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.

Single Port RAM

Single Port RAM implementations, properties, and restrictions for HDL code generation

Description

The Single Port RAM block is available with Simulink.

For information on the simulation behavior and block parameters, see `Single Port RAM`.

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 2-299.

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.

Spectrum Analyzer

Spectrum Analyzer implementations, properties, and restrictions for HDL code generation

Description

The Spectrum Analyzer block is available with DSP System Toolbox.

For information on 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.

Sqrt

Sqrt implementations, properties, and restrictions for HDL code generation

Description

The Sqrt block is available with Simulink.

For information on the simulation behavior and block parameters, see `Sqrt`.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. View the generated model or validation model to see the added latency. 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: • `off` (default): Use a bitset shift/addition algorithm. • `on`: Use a multiply/add algorithm.
`SqrtNewton`	`Iterations`	`Iterations` + 3	Use the iterative Newton method. Select this option to optimize area.

Architecture	Parameter	Additional cycles of latency	Description
			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

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`.

Restrictions

- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

State Control

State Control implementations, properties, and restrictions for HDL code generation

Description

PLACEHOLDER.

The State Control block is available with Simulink.

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”.

State Transition Table

State Transition Table implementations, properties, and restrictions for HDL code generation

Description

The State Transition Table block is available with Stateflow.

For information on the simulation behavior and block parameters, see `State Transition Table`.

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 “Use Active State Output Data”.

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 string, with spaces separating the variables.

See Also

Chart | Message Viewer | Truth Table

Stop Simulation

Stop Simulation implementations, properties, and restrictions for HDL code generation

Description

The Stop Simulation block is available with Simulink.

For information on 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.

Subsystem

Subsystem implementations, properties, and restrictions for HDL code generation

Description

The Subsystem block is available with Simulink.

For information on 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`	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. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of 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

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. 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 values are loaded in the corresponding fields.

ProcessorFPGASynchronization

Processor / FPGA synchronization mode, specified as a string.

In the HDL Workflow Advisor, you can set this property in the **Processor/FPGA Synchronization** field.

Values: `Free running` (default) | `Coprocessing - blocking`

Example: `'Free running'`

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 (;).

In the HDL Workflow Advisor, you can set this property in the **Additional source files** field.

Values: `''` (default) | string

Example: `'C:\myprojfiles\led_blinking_file1.vhd;C:\myprojfiles\led_blinking_file2.vhd;'`

IPCoreName

IP core name, specified as a string.

In the HDL Workflow Advisor, you can set this property using 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) | string

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a string.

In the HDL Workflow Advisor, you can set this property using 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) | string

Example: `'1.3'`

More About

- "External Component Interfaces"
- "Generate Black Box Interface for Subsystem"

Subtract

Subtract implementations, properties, and restrictions for HDL code generation

Description

The Subtract block is available with Simulink.

For information on 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

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

The default `Linear` implementation supports complex data.

Sum

Sum implementations, properties, and restrictions for HDL code generation

Description

The Sum block is available with Simulink.

For information on 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

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

The default `Linear` implementation supports complex data.

Sum of Elements

Sum of Elements implementations, properties, and restrictions for HDL code generation

Description

The Sum of Elements block is available with Simulink.

For information on 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. View the generated model or validation model to see the added latency. See "Generated Model and Validation Model".

Architecture	Additional cycles of latency	Description
`default` `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".

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

The default `Linear` implementation supports complex data.

Switch

Switch implementations, properties, and restrictions for HDL code generation

Description

The Switch block is available with Simulink.

For information on 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.

Tapped Delay

Tapped Delay implementations, properties, and restrictions for HDL code generation

Description

The Tapped Delay block is available with Simulink.

For information on 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.

Terminator

Terminator implementations, properties, and restrictions for HDL code generation

Description

The Terminator block is available with Simulink.

For information on 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.

Time Scope

Time Scope implementations, properties, and restrictions for HDL code generation

Description

The Time Scope block is available with DSP System Toolbox.

For information on 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.

To File

To File implementations, properties, and restrictions for HDL code generation

Description

The To File block is available with Simulink.

For information on 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.

To VCD File

To VCD File implementations, properties, and restrictions for HDL code generation

Description

The To VCD File block is available with HDL Verifier.

For information on 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.

To Workspace

To Workspace implementations, properties, and restrictions for HDL code generation

Description

The To Workspace block is available with Simulink.

For information on 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.

Trigger

Trigger implementations, properties, and restrictions for HDL code generation

Description

The Trigger block is available with Simulink.

For information on the simulation behavior and block parameters, see `Trigger`.

HDL Architecture

This block has a single, default HDL architecture.

See Also

`Triggered Subsystem`

Triggered Subsystem

Triggered Subsystem implementations, properties, and restrictions for HDL code generation

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 “Create a Triggered Subsystem” in the Simulink documentation.

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`	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. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of 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

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. 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 2-328 for information on 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

See Also

`Subsystem` | `Trigger`

Trigonometric Function

Trigonometric Function implementations, properties, and restrictions for HDL code generation

Description

The Trigonometric Function block is available with Simulink.

For information on 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. View the generated model or validation model to see the added latency. See “Generated Model and Validation Model”.

The Trigonometric Function block supports HDL code generation for the following functions.

Architecture	Function	Approximation Method	UsePipelinedKernel 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
	`sincos`	`CORDIC`	`On`	**Number of iterations** + 1
			`Off`	0

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”.

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. Use this implementation if the block is in a feedback loop.

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 an unsupported function on the Trigonometric Function block.
- You select an **Approximation method** other than `CORDIC`.
- You select the **SinCosCordic** architecture, **UsePipelinedKernel** is `On`, and the block is in a feedback loop.

See Also

`cordiccos` | `cordicsin` | `cordicsincos`

Truth Table

Truth Table implementations, properties, and restrictions for HDL code generation

Description

The Truth Table block is available with Stateflow.

For information on the simulation behavior and block parameters, see `Truth Table`.

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 string, with spaces separating the variables.

See Also

`Chart` | `Message Viewer` | `State Transition Table`

Unary Minus

Unary Minus implementations, properties, and restrictions for HDL code generation

Description

The Unary Minus block is available with Simulink.

For information on 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.

Unit Delay

Unit Delay implementations, properties, and restrictions for HDL code generation

Description

The Unit Delay block is available with Simulink.

For information on 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.

Unit Delay Enabled

Unit Delay Enabled implementations, properties, and restrictions for HDL code generation

Description

The Unit Delay Enabled block is available with Simulink.

For information on the simulation behavior and block parameters, see `Unit Delay Enabled`.

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.

Unit Delay Enabled Resettable

Unit Delay Enabled Resettable implementations, properties, and restrictions for HDL code generation

Description

The Unit Delay Enabled Resettable block is available with Simulink.

For information on the simulation behavior and block parameters, see `Unit Delay Enabled Resettable`.

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”.

Unit Delay Resettable

Unit Delay Resettable implementations, properties, and restrictions for HDL code generation

Description

The Unit Delay Resettable block is available with Simulink.

For information on the simulation behavior and block parameters, see `Unit Delay Resettable`.

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”.

Upsample

Upsample implementations, properties, and restrictions for HDL code generation

Description

The Upsample block is available with DSP System Toolbox.

For information on 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”.

Complex Data Support

This block supports code generation for complex signals.

Variable Selector

Variable Selector implementations, properties, and restrictions for HDL code generation

Description

The Variable Selector block is available with DSP System Toolbox.

For information on 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.

Variant Subsystem

Variant Subsystem implementations, properties, and restrictions for HDL code generation

Description

The Variant Subsystem block is available with Simulink. For information on 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`	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. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals.
`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

BalanceDelays

Delay balancing. 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”.

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, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. 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.

Vector Concatenate

Vector Concatenate implementations, properties, and restrictions for HDL code generation

Description

The Vector Concatenate block is available with Simulink.

For information on 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.

Vector Scope

Vector Scope implementations, properties, and restrictions for HDL code generation

Description

The Vector Scope block is available with DSP System Toolbox.

For information on the simulation behavior and block parameters, see `Vector Scope`.

HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

Viterbi Decoder

Viterbi Decoder implementations, properties, and restrictions for HDL code generation

Description

The Viterbi Decoder block is available with Communications System Toolbox.

For information on 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
- 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.

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

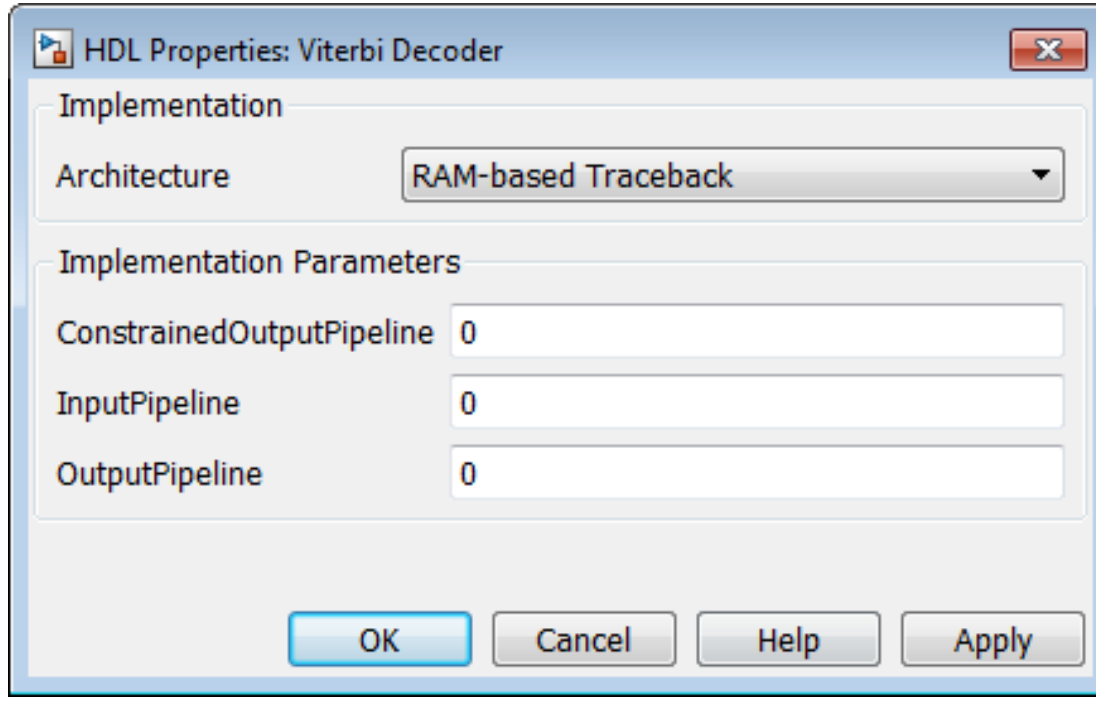
indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the parameter value results in fewer registers along with a decrease in the circuit speed.

For an example using `TracebackStagesPerPipeline`, see the "HDL Code Generation for Viterbi Decoder" example model.

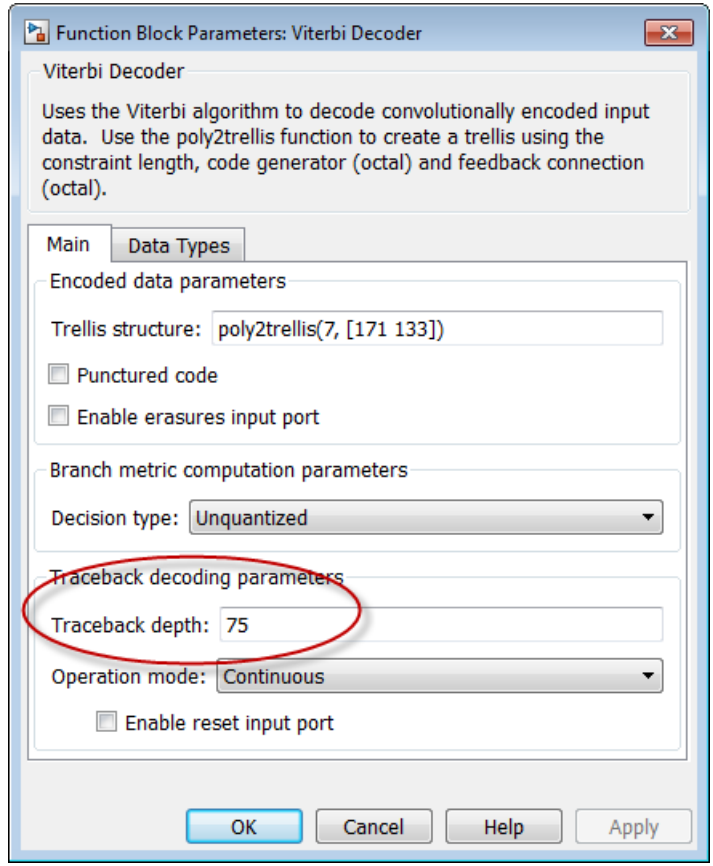
RAM-Based Traceback

Instead of using registers, you can choose to use RAMs to save the survivor branch information.

1. Set the HDL 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.

See HDL Code Generation for Viterbi Decoder.

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 2-350.

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.

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.

Example

The “HDL Code Generation for Viterbi Decoder” example shows HDL code generation for a fixed-point Viterbi Decoder block, with pipelined traceback decoding. To open the example, type the following command:

```
showdemo commviterbihdl_m
```

Waterfall

Waterfall implementations, properties, and restrictions for HDL code generation

Description

The Waterfall block is available with DSP System Toolbox.

For information on 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.

Wrap To Zero

Wrap To Zero implementations, properties, and restrictions for HDL code generation

Description

The Wrap To Zero block is available with Simulink.

For information on 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.

Zero-Order Hold

Zero-Order Hold implementations, properties, and restrictions for HDL code generation

Description

The Zero-Order Hold block is available with Simulink.

For information on 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.

XY Graph

XY Graph implementations, properties, and restrictions for HDL code generation

Description

The XY Graph block is available with Simulink.

For information on 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.

Properties — Alphabetical List

AdderSharingMinimumBitwidth

Minimum bitwidth of shared adders for resource sharing optimization

Settings

N

Default: 0

Minimum bit width of a shared adder when using the resource sharing optimization, specified as an integer greater than or equal to 0.

To use this parameter, you must enable `ShareAdders`. You must also enable resource sharing for the parent subsystem.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`MultiplierSharingMinimumBitwidth` | `ShareAdders` | `ShareAtomicSubsystems` | `ShareMATLABBlocks` | `ShareMultipliers`

More About

- "Resource Sharing"

ClockRatePipelineOutputPorts

Enable clock-rate pipelining for DUT ports

Settings

`'on'`

Enable clock-rate pipelining for DUT ports.

`'off'` (default)

Disable clock-rate pipelining for DUT ports.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockRatePipelining`

More About

- "Clock-Rate Pipelining"

BalanceDelays

Set delay balancing for the model

Settings

`'on'` (default)

Enable delay balancing for the model.

`'off'`

Disable delay balancing for the model.

Usage Notes

You can further control delay balancing within the model by disabling or enabling delay balancing for subsystems within the 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

- “Delay Balancing ”
- “BalanceDelays”

BlockGenerateLabel

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

Settings

`'string'`

Default: `'_gen'`

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`InstanceGenerateLabel`, `OutputGenerateLabel`

BlocksWithNoCharacterizationFile

Highlighting script for blocks without timing information in estimated critical path

Settings

'string'

Default: `'highlightCriticalPathEstimationOffendingBlocks'`

Name of MATLAB script that contains commands to highlight blocks on the estimated critical path without timing information. The script highlights blocks in the generated model. HDL Coder saves the script when you generate code with `CriticalPathEstimation` set to `'on'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`CriticalPathEstimation` | `CriticalPathEstimationFile`

Related Examples

- "Find Estimated Critical Paths Without Synthesis Tools"

CheckHDL

Check model or subsystem for HDL code generation compatibility

Settings

`'on'`

Selected

Check the model or subsystem for HDL compatibility before generating code, and report problems encountered. This is equivalent to executing the `checkhdl` function before calling `makehdl`.

`'off'` (default)

Cleared (default)

Do not check the model or subsystem for HDL compatibility before generating code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`checkhdl`, `makehdl`

ClockEdge

Specify active clock edge

Settings

`'Rising'` (default)

The rising clock edge triggers Verilog `always` or VHDL `process` blocks in the generated code.

`'Falling'`

The falling clock edge triggers Verilog `always` or VHDL `process` blocks in the generated code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ResetAssertedLevel`, `ClockInputPort`, `InputType`, `OutputType`, `ResetInputPort`

ClockEnableInputPort

Name HDL port for model's clock enable input signals

Settings

'string'

Default: `'clk_enable'`

The string specifies the name for the model's clock enable input port.

If you override the default with (for example) the string `'filter_clock_enable'` for the generating subsystem `filter_subsys`, the generated entity declaration might look as follows:

```
ENTITY filter_subsys IS
   PORT( clk                :  IN  std_logic;
         filter_clock_enable :  IN  std_logic;
         reset              :  IN  std_logic;
         filter_subsys_in   :  IN  std_logic_vector (15 DOWNTO 0);
         filter_subsys_out  :  OUT std_logic_vector (15 DOWNTO 0);
         );
END filter_subsys;
```

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

Usage Notes

The clock enable signal is asserted active high (1). Thus, the input value must be high for the generated entity's registers to be updated.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockInputPort`, `InputType`, `OutputType`, `ResetInputPort`

ClockEnableOutputPort

Specify name of clock enable output port

Settings

'string'

Default: `'ce_out'`

The string specifies the name for the generated clock enable output port.

A clock enable output is generated when the design requires one.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

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`

ClockInputs

Specify generation of single or multiple clock inputs

Settings

`'Single'` (Default)

Generates a single clock input for the DUT. If the DUT is multirate, the input clock is the master clock rate, and a timing controller is synthesized to generate additional clocks as required.

`'Multiple'`

Generates a unique clock for each Simulink rate in the DUT. The number of timing controllers generated depends on the contents of the DUT.

Usage Notes

The oversample factor must be 1 (default) to specify multiple clocks.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Example

The following example specifies the generation of multiple clocks.

```
makehdl(gcb, 'ClockInputs','Multiple');
```

ClockInputPort

Name HDL port for model's clock input signals

Settings

'string'

Default: `'clk'`.

The string specifies the clock input port name.

If you override the default with (for example) the string `'filter_clock'` for the generated entity `my_filter`, the generated entity declaration might look as follows:

```
ENTITY my_filter IS
   PORT( filter_clock   :  IN  std_logic;
         clk_enable     :  IN  std_logic;
         reset          :  IN  std_logic;
         my_filter_in   :  IN  std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         my_filter_out  :  OUT std_logic_vector (15 DOWNTO 0); -- sfix16_En15
         );
END my_filter;
```

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockEnableInputPort`, `InputType`, `OutputType`

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`

ClockProcessPostfix

Specify string to append to HDL clock process names

Settings

'string'

Default: '_process'.

HDL Coder uses process blocks for register operations. The label for each of these blocks is derived from a register name and the postfix _process. For example, the coder derives the label delay_pipeline_process in the following block declaration from the register name delay_pipeline and the default postfix string _process:

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

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

See Also

PackagePostfix, ReservedWordPostfix

ClockRatePipelining

Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths

Settings

`'on'` (default)

Insert pipeline registers at clock rate for multi-cycle paths.

`'off'`

Insert pipeline registers at data rate for multi-cycle paths.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockRatePipelineOutputPorts`

More About

- "Clock-Rate Pipelining"

CodeGenerationOutput

Control production of generated code and display of generated model

Settings

`'string'`

Default: `'GenerateHDLCode'`

Generate code but do not display the generated model.

`'GenerateHDLCodeAndDisplayGeneratedModel'`

Generate both code and model, and display model when completed.

`'DisplayGeneratedModelOnly'`

Create and display generated model, but do not proceed to code generation.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`GeneratedModelName` | `GeneratedModelNamePrefix`

More About

- "Generated Model and Validation Model"

ComplexImagPostfix

Specify string to append to imaginary part of complex signal names

Settings

`'string'`

Default: `'_im'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ComplexRealPostfix`

ComplexRealPostfix

Specify string to append to real part of complex signal names

Settings

`'string'`

Default: `'_re'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ComplexImagPostfix`

CriticalPathEstimation

Estimate critical path without running synthesis

Settings

`'on'`

Estimate the critical path without running synthesis. Generate a script that highlights the estimated critical path in the generated model.

`'off'` (default)

Do not estimate the critical path.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`BlocksWithNoCharacterizationFile` | `CriticalPathEstimationFile`

Related Examples

- "Find Estimated Critical Paths Without Synthesis Tools"

CriticalPathEstimationFile

Critical path estimation highlighting script name

Settings

`'string'`

Default: `'criticalPathEstimated'`

Name of MATLAB script that contains commands to highlight the estimated critical path in the generated model. HDL Coder saves the script when you generate code with `CriticalPathEstimation` set to `'on'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`BlocksWithNoCharacterizationFile` | `CriticalPathEstimation`

Related Examples

- "Find Estimated Critical Paths Without Synthesis Tools"

DateComment

Specify whether to include time/date information in the generated HDL file header

Settings

`'on'` (default)

Include time/date information in the generated HDL file header.

```
 -- -------------------------------------------------
 --
 -- File Name: hdlsrc\symmetric_fir.vhd
 -- Created: 2011-02-14 07:21:36
 --
```

`'off'`

Omit time/date information in the generated HDL file header.

```
 -- -------------------------------------------------
 --
 -- File Name: hdlsrc\symmetric_fir.vhd
 --
```

By omitting the time/date information in the file header, you can more easily determine if two HDL files contain identical code. You can also avoid extraneous revisions of the same file when checking in HDL files to a source code management (SCM) system.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

DetectBlackBoxNameCollision

Check for duplicate module or entity names in generated code and black box interface code

Settings

`'None'`

Do not check for black box subsystems that have the same HDL module name as a generated HDL module name.

`'Warning'` (default)

Check for black box subsystems that have the same HDL module name as a generated HDL module name. Display a warning if matching names are found.

`'Error'`

Check for black box subsystems that have the same HDL module name as a generated HDL module name. Display an error if matching names are found.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Related Examples

- "Check for name conflicts in black box interfaces"

DistributedPipeliningBarriers

Highlight blocks that are inhibiting distributed pipelining

Settings

`'on'` (default)

Generate a MATLAB script that highlights blocks that are inhibiting distributed pipelining in the original model and generated model.

`'off'`

Do not generate a script to highlight blocks that are inhibiting distributed pipelining.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`DistributedPipeliningBarriersFile`

Related Examples

- "Find Feedback Loops"

DistributedPipeliningBarriersFile

Distributed pipelining barriers highlighting script name

Settings

'string'

Default: `'highlightDistributedPipeliningBarriers'`

Name of MATLAB script that contains commands to highlight blocks that are inhibiting distributed pipelining in the original model and generated model. HDL Coder saves the script when you generate code with `DistributedPipeliningBarriers` set to `'on'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`DistributedPipeliningBarriers`

Related Examples

- "Find Feedback Loops"

DistributedPipeliningPriority

Specify priority for distributed pipelining algorithm

Settings

`'NumericalIntegrity'` (default)

Prioritize numerical integrity when distributing pipeline registers.

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.

`'Performance'`

Prioritize performance over numerical integrity.

Use this option if your design requires a higher clock frequency and the Simulink behavior does not need to strictly match the generated code behavior.

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.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

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”

EnablePrefix

Specify base name string for internal clock enables in generated code

Settings

'string'

Default: `'enb'`

Specify the string used as the base name for internal clock enables and other flow control signals in generated code.

Usage Notes

Where only a single clock enable is generated, `EnablePrefix` specifies the signal name for the internal clock enable signal.

In some cases multiple clock enables are generated (for example, when a cascade block implementation for certain blocks is specified). In such cases, `EnablePrefix` specifies a base signal name for the first clock enable that is generated. For other clock enable signals, numeric tags are appended to `EnablePrefix` to form unique signal names. For example, the following code fragment illustrates two clock enables that were generated when `EnablePrefix` was set to `'test_clk_enable'` :

```
COMPONENT mysys_tc
    PORT( clk                  :   IN    std_logic;
          reset                :   IN    std_logic;
          clk_enable           :   IN    std_logic;
          test_clk_enable      :   OUT   std_logic;
          test_clk_enable_5_1_0 :   OUT   std_logic
          );
  END COMPONENT;
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

EntityConflictPostfix

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

Settings

`'string'`

Default: `'_block'`

The specified postfix resolves duplicate VHDL entity or Verilog module names.

For example, if HDL Coder detects two entities with the name `MyFilter`, the coder names the first entity `MyFilter` and the second entity `MyFilter_block`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`PackagePostfix`, `ReservedWordPostfix`

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`

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 string specified by the `CosimLibPostfix` property 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 for use with Mentor Graphics ModelSim, the HDL Coder software generates and opens a Simulink model that contains an HDL Cosimulation block for Mentor Graphics ModelSim.

`'Incisive'`

If your installation includes HDL Verifier for use with Cadence Incisive, the HDL Coder software 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”

GeneratedModelName

Specify name of generated model

Settings

`'string'`

By default, the name of a generated model is the same as that of the original model. Assign a string value to `GeneratedModelName` to override 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

`CodeGenerationOutput` | `GeneratedModelNamePrefix`

More About

- "Generated Model and Validation Model"

GeneratedModelNamePrefix

Specify prefix to name of generated model

Settings

`'string'`

Default: `'gm_'`

The specified string is prepended to the name of the generated 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

`CodeGenerationOutput` | `GeneratedModelName`

More About

- "Generated Model and Validation Model"

GenerateHDLCode

Generate HDL code

Settings

`'on'` (default)

Generate HDL code.

`'off'`

Do not generate HDL code.

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 HDL Code Using the Configuration Parameters Dialog Box”

GenerateValidationModel

Generate validation model with HDL code

Settings

`'on'`

Generate a validation model that highlights generated delays and other differences between your original model and the generated model. With a validation model, you can observe the effects of streaming, resource sharing, and delay balancing.

`'off'` (default)

Do not generate a validation model.

Usage Notes

If you enable generation of a validation model, also enable delay balancing to keep the generated DUT model synchronized with the original DUT model. Mismatches between delays in the original DUT model and delays in the generated DUT model cause validation to fail.

You can set this property using `hdlset_param` or `makehdl`.

You can also generate a validation model by selecting one of the following check boxes:

- **Generate validation model** in the **HDL Code Generation** pane of the Configuration Parameters dialog box
- **Generate validation model** in the **Generate RTL Code and Testbench** task of the HDL Workflow Advisor

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

- "Delay Balancing"
- `BalanceDelays`

GenerateWebview

Include model Web view in the code generation report

Settings

`'on'`

Include model Web view in the code generation report.

`'off'` (default)

Omit model Web view in the code generation report.

Usage Notes

With a model Web view, you can click a link in the generated code to highlight the corresponding block in the 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

“Web View of Model in Code Generation Report”

HandleAtomicSubsystem

Enable reusable code generation for identical atomic subsystems

Settings

`'on'` (default)

Generate reusable code for identical atomic subsystems.

`'off'`

Do not generate reusable code for identical atomic subsystems.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`MaskParameterAsGeneric`

Related Examples

- "Generate Reusable Code for Atomic Subsystems"

HDLCodingStandard

Generate HDL code that follows the specified coding standard

Settings

`'None'` (default)

Generate generic synthesizable HDL code.

`'Industry'`

Generate HDL code that follows the industry standard rules supported by the HDL Coder software. When this option is enabled, the coder generates a standard compliance report.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

HDLCodingStandardCustomizations

Specify HDL coding standard customization object

Settings

Specify an HDL coding standard customization object.

Usage Notes

When you create the HDL coding standard customization object, you must specify the same standard as you specify for `HDLCodingStandard`. For example, if you set `HDLCodingStandard` to `'Industry'`, create the coding standard customization object using `hdl.CodingStandard('Industry')`.

To learn how to specify an HDL coding standard customization object, see HDL Coding Standard Customization Properties.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

HDLCompileInit

Specify string written to initialization section of compilation script

Settings

`'string'`

Default: `'vlib %s\n'`.

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

`VHDLLibraryName`

Related Examples

- "Generate Scripts for Compilation, Simulation, and Synthesis"

HDLCompileTerm

Specify string written to termination section of compilation script

Settings

`'string'`

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

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 string appended to file name for generated Mentor Graphics ModelSim compilation scripts

Settings

`'string'`

Default:`'_compile.do'`.

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 string written to compilation script for Verilog files

Settings

'*string*'

Default: `'vlog %s %s\n'`.

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 string written to compilation script for VHDL files

Settings

`'string'`

Default: `'vcom %s %s\n'`.

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”

HDLMapFilePostfix

Specify postfix string appended to file name for generated mapping file

Settings

'*string*'

Default: '_map.txt'.

For example, if the name of the device under test is my_design, HDL Coder adds the postfix _map.txt to form the name my_design_map.txt.

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

HDLSimCmd

Specify simulation command written to simulation script

Settings

`'string'`

Default: `'vsim -novopt %s.%s\n'`.

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 string written to initialization section of simulation script

Settings

`'string'`

The default string 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 string appended to file name for generated Mentor Graphics ModelSim simulation scripts

Settings

'string'

Default: `_sim.do`.

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 string written to termination section of simulation script

Settings

`'string'`

Default: `'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

`'string'`

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

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

`'string'`

Default: `''`

Specify the HDL lint tool command in the Tcl script. The command string 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 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 string

Settings

`'string'`

Default: `''`

Specify the HDL lint script initialization string.

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 string

Settings

`'string'`

Default: `''`

Specify the HDL lint script termination string.

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

'string'

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 strings:

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`

Related Examples

- "Generate an HDL Lint Tool Script"

HDLSynthCmd

Specify command written to synthesis script

Settings

`'string'`

Default: none.

Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis command string. The default string is a format 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 string appended to file name for generated synthesis scripts

Settings

`'string'`

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 the string `'_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 string written to initialization section of synthesis script

Settings

`'string'`

Default: none

Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis initialization string. The default string is a format string passed to `fprintf` to write the initialization section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name. 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 string written to termination section of synthesis script

Settings

`'string'`

Default: none

Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis termination string. The default string is a format string 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

'string'

Default: 'None'.

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 strings:

Tip The value of HDLSynthTool also sets the postfix string (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"

HierarchicalDistPipelining

Specify whether to apply retiming across a subsystem hierarchy

Settings

`'on'`

Enable retiming across a subsystem hierarchy. HDL Coder applies retiming hierarchically down, until it reaches a subsystem where **DistributedPipelining** is `off`.

`'off'` (default)

Distribute pipelining only within a subsystem.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

"DistributedPipelining"

HighlightAncestors

Highlight ancestors of blocks in generated model that differ from original model

Settings

`'on'` (default)

Highlight blocks in a generated model that differ from the original model, and their ancestor (parent) blocks in the model hierarchy. The `HighlightColor` property specifies the highlight color.

`'off'`

Highlight only the blocks in a generated model that differ from the original model without highlighting their ancestor (parent) blocks in the model hierarchy.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HighlightColor`

HighlightColor

Specify color for highlighted blocks in generated model

Settings

'string'

Default: 'cyan'.

Specify the color as one of the following color string values:

- 'cyan'
- 'yellow'
- 'magenta'
- 'red'
- 'green'
- 'blue'
- 'white'
- 'black'

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

See Also

HighlightAncestors

HighlightClockRatePipeliningDiagnostic

Highlight blocks that are inhibiting clock-rate pipelining

Settings

`'on'` (default)

Generate a MATLAB script that highlights blocks that are inhibiting clock-rate pipelining in the original model and generated model.

`'off'`

Do not generate a script to highlight blocks that are inhibiting clock-rate pipelining.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HighlightFeedbackLoopsFile`

Related Examples

- "Find Feedback Loops"

HighlightClockRatePipeliningFile

Clock-rate pipelining highlighting script name

Settings

`'string'`

Default: `'highlightClockRatePipelining'`

Name of MATLAB script that contains commands to highlight blocks that are inhibiting clock-rate pipelining in the original model and generated model. HDL Coder saves the script when you generate code with `HighlightClockRatePipeliningDiagnostic` set to `'on'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HighlightClockRatePipeliningDiagnostic`

Related Examples

- "Find Feedback Loops"

HighlightFeedbackLoops

Highlight feedback loops that can inhibit delay balancing and optimizations

Settings

`'on'`

Generate a MATLAB script that highlights feedback loops in the original model and generated model.

`'off'` (default)

Do not generate a script to highlight feedback loops.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HighlightFeedbackLoopsFile`

Related Examples

- "Find Feedback Loops"

HighlightFeedbackLoopsFile

Feedback loop highlighting script file name

Settings

`'string'`

Default: `'highlightFeedbackLoop'`

Name of MATLAB script that contains commands to highlight feedback loops in the original model and generated model. HDL Coder saves the script when you generate code with `HighlightFeedbackLoops` set to `'on'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HighlightFeedbackLoops`

Related Examples

- "Find Feedback Loops"

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 >= 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

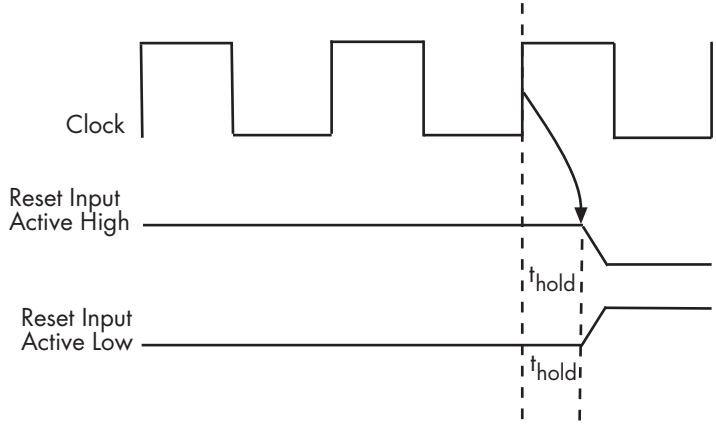
Specify the number of nanoseconds during which the model's data input signals and forced reset input signals are held past the clock rising edge.

The hold time is expressed as a positive integer.

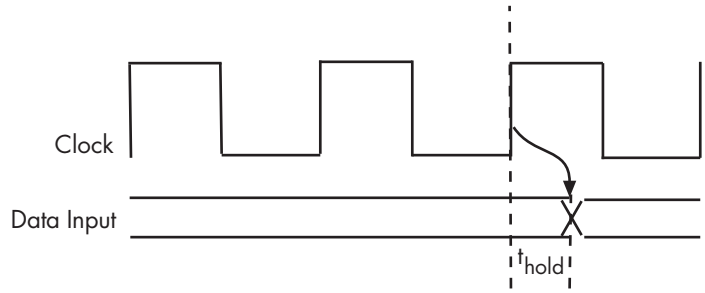
This option applies to reset input signals only if forced resets are enabled.

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_{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`.

InitializeBlockRAM

Enable or suppress generation of initial signal value for RAM blocks

Settings

`'on'` (default)

For RAM blocks, generate initial values of `'0'` for both the RAM signal and the output temporary signal.

`'off'`

For RAM blocks, do not generate initial values for either the RAM signal or the output temporary signal.

Usage Notes

This property applies to RAM blocks in the HDL Operations block library:

- Dual Port RAM
- Simple Dual Port RAM
- Single Port RAM
- Dual Rate Dual Port RAM

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`IgnoreDataChecking`

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`.

InlineConfigurations

Specify whether generated VHDL code includes inline configurations

Settings

`'on'` (default)

Selected (default)

Include VHDL configurations in files that instantiate a component.

`'off'`

Cleared

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

Usage Notes

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`

InlineMATLABBlockCode

Inline HDL code for MATLAB Function blocks

Settings

`'on'`

Inline HDL code for `MATLAB Function` blocks to avoid instantiation of code for custom blocks.

`'off'` (default)

Instantiate HDL code for `MATLAB Function` blocks and do not inline.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Examples

Enable inlining of HDL code:

```
mdl = 'my_custom_block_model';
hdlset_param(mdl,'InlineMATLABBlockCode','on');
```

Enable instantiation of HDL code:

```
mdl = 'my_custom_block_model';
hdlset_param(mdl,'InlineMATLABBlockCode','off');
```

InputType

Specify HDL data type for model input ports

Settings

Default (for VHDL):`'std_logic_vector'`

Default (for VHDL): **std_logic_vector**

Specifies VHDL type `STD_LOGIC_VECTOR` for the model's input ports.

`'signed/unsigned'`

signed/unsigned

Specifies VHDL type `SIGNED` or `UNSIGNED` for the model's input ports.

`'wire'` (Verilog)

wire (Verilog)

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockEnableInputPort`, `OutputType`

InstanceGenerateLabel

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

Settings

`'string'`

Default: `'_gen'`

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`BlockGenerateLabel`, `OutputGenerateLabel`

InstancePostfix

Specify string appended to generated component instance names

Settings

`'string'`

Default: `''` (no postfix appended)

Specify a string to be appended to component instance names in generated code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

InstancePrefix

Specify string prefixed to generated component instance names

Settings

`'string'`

Default: `'u_'`

Specify a string to be prefixed to component instance names in generated code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

LoopUnrolling

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

Settings

`'on'`

Selected

Unroll and omit `FOR` and `GENERATE` loops from the generated VHDL code.

In Verilog code, loops are always unrolled.

If you are using an electronic design automation (EDA) tool that does not support `GENERATE` loops, you can enable this option to omit loops from your generated VHDL code.

`'off'` (default)

Cleared (default)

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

Usage Notes

The setting of this option does not affect results obtained from simulation or synthesis of generated VHDL code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`InlineConfigurations`, `SafeZeroConcat`, `UseAggregatesForConst`, `UseRisingEdge`

MaskParameterAsGeneric

Generate reusable HDL code for subsystems with identical mask parameters that differ only in value

Settings

`'on'`

Generate one HDL file for multiple masked subsystems with different values for tunable mask parameters. HDL Coder automatically detects atomic subsystems with tunable mask parameters that are sharable.

Inside the subsystem, you can use the mask parameter only in the following blocks and parameters:

Block	Parameter	Limitation
Constant	**Constant value** on the Main tab of the dialog box	None
Gain	**Gain** on the Main tab of the dialog box	**Parameter data type** should be the same for all Gain blocks.

`'off'` (default)

Generate a separate HDL file for each masked subsystem.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HandleAtomicSubsystem`

Related Examples

- "Generate Reusable Code for Atomic Subsystems"

More About

- “Generate parameterized HDL code from masked subsystem”

MaxComputationLatency

Specify the maximum number of time steps for which your DUT inputs are guaranteed to be stable

Settings

1 (default)

DUT input data can change every cycle.

N, where N is an integer greater than 1

DUT input data can change every N cycles.

Usage Notes

Use with `MaxOversampling` to prevent or reduce overclocking by constraining resource sharing and streaming optimizations.

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
`MaxOversampling`

More About

- "Maximum Computation Latency"
- "Maximum Oversampling Ratio"
- "Optimization with Constrained Overclocking"

MaxOversampling

Limit the maximum sample rate

Settings

0 (default)

Do not set a limit on the maximum sample rate.

1

Do not allow oversampling.

N, where N is an integer greater than 1

Allow oversampling up to N times the original model sample rate.

Usage Notes

Use with `MaxComputationLatency` to prevent or reduce overclocking by constraining resource sharing and streaming optimizations.

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
`MaxComputationLatency`

More About

- "Maximum Oversampling Ratio"

- “Maximum Computation Latency”
- “Optimization with Constrained Overclocking”

MinimizeClockEnables

Omit generation of clock enable logic for single-rate designs

Settings

`'on'`

Omit generation of clock enable logic for single-rate designs, wherever possible (see "Usage Notes" on page 3-95). The following VHDL code example does not define or examine a clock enable signal. When the clock signal (`clk`) goes high, the current signal value is output.

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      Unit_Delay_out1 <= In1_signed;
    END IF;
  END PROCESS Unit_Delay_process;
```

`'off'` (default)

Generate clock enable logic. The following VHDL code extract represents a register with a clock enable (`enb`)

```
Unit_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay_out1 <= to_signed(0, 32);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Unit_Delay_out1 <= In1_signed;
      END IF;
    END IF;
  END PROCESS Unit_Delay_process;
```

Usage Notes

In some cases, HDL Coder emits clock enables even when `MinimizeClockEnables` is `'on'`. These cases are:

- Registers inside Enabled, State-Enabled, and Triggered subsystems.
- Multirate models.
- The coder emits clock enables for the following blocks:
  - commseqgen2/PN Sequence Generator
  - dspsigops/NCO

    Note: HDL support for the NCO block will be removed in a future release. Use the NCO HDL Optimized block instead.
  - dspsrcs4/Sine Wave
  - hdldemolib/HDL FFT
  - built-in/DiscreteFir
  - dspmlti4/CIC Decimation
  - dspmlti4/CIC Interpolation
  - dspmlti4/FIR Decimation
  - dspmlti4/FIR Interpolation
  - dspadpt3/LMS Filter
  - dsparch4/Biquad Filter

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

MinimizeIntermediateSignals

Specify whether to optimize HDL code for debuggability or code coverage

Settings

`'on'`

Optimize for code coverage by minimizing intermediate signals. For example, suppose that the generated code with this setting *off* is:

```
const3 <= to_signed(24, 7);
subtractor_sub_cast <= resize(const3, 8);
subtractor_sub_cast_1 <= resize(delayout, 8);
subtractor_sub_temp <= subtractor_sub_cast - subtractor_sub_cast_1;
```

With this setting *on*, the output code is optimized to:

```
subtractor_sub_temp <= 24 - (resize(delayout, 8));
```

The intermediate signals `const3`, `subtractor_sub_cast`, and `subtractor_sub_cast_1` are removed.

`'off'` (default)

Optimize for debuggability by preserving intermediate signals.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

ModulePrefix

Specify prefix string for DUT module or entity name

Settings

`'string'`

Default: `''`

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.

Usage Notes

You can specify the module name prefix to avoid name collisions if you plan to instantiate the generated HDL code multiple times in a larger system.

For example, suppose you have a DUT, `myDut`, containing an internal module, `myUnit`. You can prefix the modules within your design with the string, `unit1_`, by entering the following command:

```
hdlset_param ('path/to/myDut', 'ModulePrefix','unit1_')
```

In the generated code, your HDL module names are `unit1_myDut` and `unit1_myUnit`, with corresponding HDL file names. Generated script file names also have the `unit1_` prefix.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

MulticyclePathInfo

Generate text file that reports multicycle path constraint information for use with synthesis tools

Settings

`'on'`

Selected

Generate a multicycle path information file.

`'off'` (default)

Do not generate a multicycle path information file.

Usage Notes

The file name for the multicycle path information file derives from the name of the DUT and the postfix string `'_constraints'`, as follows:

`DUTname``_constraints.txt`

For example, if the DUT name is `symmetric_fir`, the name of the multicycle path information file is `symmetric_fir_constraints.txt`.

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 Multicycle Path Information Files ”

MultifileTestBench

Divide generated test bench into helper functions, data, and HDL test bench code files

Settings

`'on'`

Write separate files for test bench code, helper functions, and test bench data. The file names are derived from the name of the DUT, the `TestBenchPostfix` property, and the `TestBenchDataPostfix` property as follows:

`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

`'off'` (default)

Write a single test bench file containing the HDL test bench code and helper functions 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`

MultiplierPartitioningThreshold

Multiplier partitioning input bit width threshold

Settings

N

Default: `Inf`

N must be an integer greater than or equal to 2.

The maximum input bit width for a multiplier. If a multiplier has an input bit width greater than `MultiplierPartitioningThreshold`, HDL Coder splits the multiplier into smaller multipliers.

To improve your hardware mapping results, set `MultiplierPartitioningThreshold` to the input bit width of the DSP or multiplier hardware on your target device.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

MultiplierSharingMinimumBitwidth

Minimum bit width of shared multipliers for resource sharing optimization

Settings

N

Default: 0

Minimum bit width of a shared multiplier when using the resource sharing optimization, specified as an integer greater than or equal to 0.

To use this parameter, you must enable `ShareMultipliers`. You must also enable resource sharing for the parent subsystem.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`AdderSharingMinimumBitwidth` | `ShareAdders` | `ShareAtomicSubsystems` | `ShareMATLABBlocks` | `ShareMultipliers` | `ShareMultipliers`

More About

- "Resource Sharing"

OptimizationReport

Display HTML optimization report

Settings

`'on'`

Create and display an HTML optimization report.

`'off'` (default)

Do not create an HTML optimization report.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“ Create and Use Code Generation Reports”

OptimizeTimingController

Optimize timing controller entity by implementing separate counters per rate

Settings

`'on'` (default)

A timing controller code file is generated if required by the design, for example:

- When code is generated for a multirate model.
- When a cascade block implementation for certain blocks is specified.

This file contains a module defining timing signals (clock, reset, external clock enable inputs and clock enable output) in a separate entity or module. In a multirate model, the timing controller entity generates the required rates from a single master clock using one or more counters and multiple clock enables.

When `OptimizeTimingController` is set `'on'` (the default), HDL Coder generates multiple counters (one counter for each rate in the model). The benefit of this optimization is that it generates faster logic, and the size of the generated code is usually much smaller.

`'off'`

When `OptimizeTimingController` is set `'off'`, the timing controller uses one counter to generate the rates in the 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

"Code Generation from Multirate Models", `EnablePrefix`, `TimingControllerPostfix`

OutputGenerateLabel

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

Settings

`'string'`

Default: `'outputgen'`

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`BlockGenerateLabel`, `OutputGenerateLabel`

OutputType

Specify HDL data type for model output ports

Settings

`'Same as input data type'` (VHDL default)

Same as input data type (VHDL default)

Output ports have the same type as the specified input port type.

`'std_logic_vector'`

std_logic_vector

Output ports have VHDL type `STD_LOGIC_VECTOR`.

`'signed/unsigned'`

signed/unsigned

Output ports have type `SIGNED` or `UNSIGNED`.

`'wire'` (Verilog)

wire (Verilog)

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockEnableInputPort`, `InputType`

Oversampling

Specify frequency of global oversampling clock as a multiple of model base rate

Settings

N

Default: 1.

N must be an integer greater than or equal to `0`.

`Oversampling` specifies *N*, the *oversampling factor* of a global oversampling clock. The oversampling factor expresses the global oversampling clock rate as a multiple of your model's base rate.

When you specify an oversampling factor greater than 1, HDL Coder generates the global oversampling clock and derives the required timing signals from the clock signal. By default, the coder does not generate a global oversampling clock.

Generation of the global oversampling clock affects only generated HDL code. The clock does not affect the simulation behavior of your model.

If you want to generate a global oversampling clock:

- The oversampling factor must be an integer greater than or equal to 1.
- In a multirate DUT, the other rates in the DUT must divide evenly into the global oversampling rate.

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 Global Oversampling Clock ”

PackagePostfix

Specify string to append to specified model or subsystem name to form name of package file

Settings

'string'

Default: '_pkg'

HDL Coder applies this option only if a package file is required for the design.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockProcessPostfix`, `EntityConflictPostfix`, `ReservedWordPostfix`

PipelinePostfix

Specify string to append to names of input or output pipeline registers generated for pipelined block implementations

Settings

'*string*'

Default: '_pipe'

When you specify a generation of input and/or output pipeline registers for selected blocks, HDL Coder appends the string specified by the `PipelinePostfix` property when generating code for such pipeline registers.

For example, suppose you specify a pipelined output implementation for a Product block in a model, as in the following code:

```
hdlset_param('sfir_fixed/symmetric_fir/Product','OutputPipeline', 2)
```

The following `makehdl` command specifies that the coder appends `'testpipe'` to generated pipeline register names.

```
makehdl(gcs,'PipelinePostfix','testpipe');
```

The following excerpt from generated VHDL code shows process the `PROCESS` code, with postfixed identifiers, that implements two pipeline stages:

```
Product_outtestpipe_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Product_outtestpipe_reg <= (OTHERS => to_signed(0, 33));
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' THEN
        Product_outtestpipe_reg(0) <= Product_out1;
        Product_outtestpipe_reg(1) <= Product_outtestpipe_reg(0);
      END IF;
    END IF;
  END PROCESS Product_outtestpipe_process;
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“HDL Block Properties”, “InputPipeline”, “OutputPipeline”

PreserveDesignDelays

Enable to prevent distributed pipelining from moving design delays

Settings

`'on'`

Prevent distributed pipelining from moving design delays, such as:

- Persistent variable in a MATLAB Function block or Stateflow Chart
- Unit Delay block
- Integer Delay block
- Memory block
- Delay block from DSP System Toolbox
- `dsp.Delay` System object from DSP System Toolbox

`'off'` (default)

Allow distributed pipelining to move design delays.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

More About

- "Distributed Pipelining and Hierarchical Distributed Pipelining"

RAMArchitecture

Select RAM architecture with or without clock enable for all RAMs in DUT subsystem

Settings

`'WithClockEnable'` (default)

Generate RAMs with clock enable.

`'WithoutClockEnable'`

Generate RAMs without clock enable.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

RAMMappingThreshold

Specify the minimum RAM size required for mapping to RAMs instead of registers

Settings

N

Default: 256.

N must be an integer greater than or equal to 0.

`RAMMappingThreshold` defines the minimum RAM size required for mapping to RAM instead of registers. This threshold applies to:

- `Delay` blocks
- Persistent variables in `MATLAB Function` blocks

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Example

To change the RAM mapping threshold for a model, use the `hdlset_param` function. For example:

```
hdlset_param('sfir_fixed', 'RAMMappingThreshold', 1024);
```

That command sets the threshold for the `sfir_fixed` model to 1024 bits.

See Also

- “UseRAM” in the HDL Coder documentation

- “MapPersistentVarsToRAM” in the HDL Coder documentation

RequirementComments

Enable or disable generation of hyperlinked requirements comments in HTML code generation reports

Settings

`'on'` (default)

If the model includes requirements comments, generate hyperlinked requirements comments within the HTML code generation report. The comments link to the corresponding requirements documents.

`'off'`

When generating an HTML code generation report, render requirements as comments within the generated code

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“ Create and Use Code Generation Reports”, “Generate Code with Annotations or Comments”, `Traceability`

ReservedWordPostfix

Specify string appended to identifiers for entities, signals, constants, or other model elements that conflict with VHDL or Verilog reserved words

Settings

`'string'`

Default: `'_rsvd'`.

The reserved word postfix is applied identifiers (for entities, signals, constants, or other model elements) that conflict with VHDL or Verilog reserved words. For example, if your generating model contains a signal named `mod`, HDL Coder adds the postfix `_rsvd` to form the name `mod_rsvd`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockProcessPostfix`, `EntityConflictPostfix`, `ReservedWordPostfix`

ResetAssertedLevel

Specify asserted (active) level of reset input signal

Settings

`'active-high'` (default)

Active-high (default)

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

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

`'active-low'`

Active-low

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

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ResetType`, `ClockInputPort`, `ClockEdge`

ResetInputPort

Name HDL port for model's reset input

Settings

'string'

Default: `'reset'`.

The string specifies the name for the model's reset input port. If you override the default with (for example) the string `'chip_reset'` for the generating system `myfilter`, the generated entity declaration might look as follows:

```
ENTITY myfilter IS
   PORT( clk             :  IN  std_logic;
         clk_enable      :  IN  std_logic;
         chip_reset      :  IN  std_logic;
         myfilter_in     :  IN  std_logic_vector (15 DOWNTO 0);
         myfilter_out    :  OUT std_logic_vector (15 DOWNTO 0);
         );
END myfilter;
```

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

Usage Notes

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockEnableInputPort`, `InputType`, `OutputType`

ResetLength

Define length of time (in clock cycles) during which reset is asserted

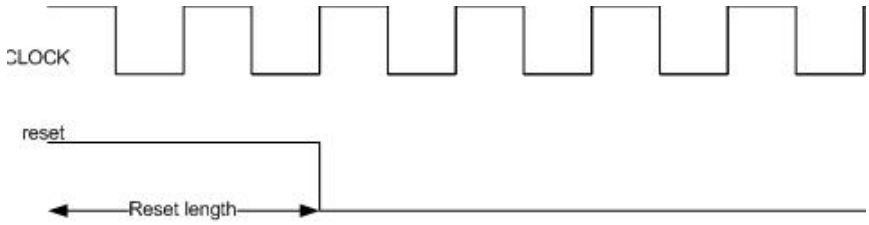
Settings

N

Default: 2.

N must be an integer greater than or equal to 0.

`Resetlength` defines *N*, the number of clock cycles during which reset is asserted. The following figure illustrates the default case, in which the reset signal (active-high) is asserted for 2 clock cycles.



Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

ResetType

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

Settings

`'async'` (default)

Asynchronous (default)

Use asynchronous reset logic. The following process block, generated by a Unit Delay block, illustrates the use of asynchronous resets. When the reset signal is asserted, the process block performs a reset, without checking for a clock event.

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
  END PROCESS Unit_Delay1_process;
```

`'sync'`

Synchronous

Use synchronous reset logic. Code for a synchronous reset follows. The following process block, generated by a Unit Delay block, checks for a clock event, the rising edge, before performing a reset:

```
Unit_Delay1_process : PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
      IF reset = '1' THEN
        Unit_Delay1_out1 <= (OTHERS => '0');
      ELSIF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
```

```
  END IF;
END PROCESS Unit_Delay1_process;
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ResetAssertedLevel`

ResourceReport

Display HTML resource utilization report

Settings

`'on'`

Create and display an HTML resource utilization report (bill of materials).

`'off'` (default)

Do not create an HTML resource utilization report.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“ Create and Use Code Generation Reports”

SafeZeroConcat

Specify syntax for concatenated zeros in generated VHDL code

Settings

`'on'` (default)

Selected (default)

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

`'off'`

Cleared

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

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`LoopUnrolling`, `UseAggregatesForConst`, `UseRisingEdge`

ScalarizePorts

Flatten vector ports into structure of scalar ports in VHDL code

Settings

`'on'`

When generating code for a vector port, generate a structure of scalar ports

`'off'` (default)

Do not generate a structure of scalar ports for a vector port.

Usage Notes

The `ScalarizePorts` property lets you control how HDL Coder generates VHDL code for vector ports.

For example, consider the subsystem `vsum` in the following figure.

By default, `ScalarizePorts` is `'off'`. The coder generates a type definition and port declaration for the vector port `In1` like the following:

```
PACKAGE simplevectorsum_pkg IS
  TYPE vector_of_std_logic_vector16 IS ARRAY (NATURAL RANGE <>)
     OF std_logic_vector(15 DOWNTO 0);
  TYPE vector_of_signed16 IS ARRAY (NATURAL RANGE <>) OF signed(15 DOWNTO 0);
END simplevectorsum_pkg;
```

```
.
.
.
ENTITY vsum IS
  PORT( In1     :  IN    vector_of_std_logic_vector16(0 TO 9);  -- int16 [10]
        Out1    :  OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END vsum;
```

Under VHDL typing rules two types declared in this manner are not compatible across design units. This may cause problems if you need to interface two or more generated VHDL code modules.

You can flatten such a vector port into a structure of scalar ports by enabling `ScalarizePorts` in your `makehdl` command, as in the following example.

```
makehdl(gcs,'ScalarizePorts','on')
```

The listing below shows the generated ports.

```
ENTITY vsum IS
  PORT( In1_0               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_1               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_2               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_3               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_4               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_5               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_6               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_7               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_8               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        In1_9               :   IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1                :   OUT   std_logic_vector(19 DOWNTO 0)  -- sfix20
        );
END vsum;
```

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 Black Box Interface for Referenced Model”

ShareAdders

Share adders with resource sharing optimization

Settings

`'on'`

When resource sharing is enabled, share adders with a bit width greater than or equal to `AdderSharingMinimumBitwidth`.

`'off'` (default)

Do not share adders.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`AdderSharingMinimumBitwidth` | `ShareAtomicSubsystems` | `ShareMATLABBlocks` | `ShareMultipliers`

More About

- "Resource Sharing"

ShareAtomicSubsystems

Share atomic subsystems with resource sharing optimization

Settings

`'on'` (default)

When resource sharing is enabled, share atomic subsystems.

`'off'`

Do not share atomic subsystems.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ShareAdders` | `ShareMATLABBlocks` | `ShareMultipliers`

More About

- "Resource Sharing"

ShareMATLABBlocks

Share MATLAB Function blocks with resource sharing optimization

Settings

`'on'` (default)

When resource sharing is enabled, share MATLAB Function blocks.

`'off'`

Do not share MATLAB Function blocks.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ShareAdders` | `ShareAtomicSubsystems` | `ShareMultipliers`

More About

- "Resource Sharing"

ShareMultipliers

Share multipliers with resource sharing optimization

Settings

`'on'` (default)

When resource sharing is enabled, share multipliers with a bit width greater than or equal to `MultiplierSharingMinimumBitwidth`.

`'off'`

Do not share multipliers.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`MultiplierSharingMinimumBitwidth` | `ShareAdders` | `ShareAtomicSubsystems` | `ShareMATLABBlocks`

More About

- "Resource Sharing"

SimulatorFlags

Specify simulator flags to apply to generated compilation scripts

Settings

`'string'`

Default: `''`

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

Usage Notes

The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command string is specified by the `HDLCompileVHDLCmd` or `HDLCompileVerilogCmd` properties.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

SplitArchFilePostfix

Specify string to append to specified name to form name of file containing model VHDL architecture

Settings

`'string'`

Default: `'_arch'`.

This option applies only if you direct HDL Coder to place the generated VHDL entity and architecture code in separate files.

Usage Notes

The option applies only if you direct HDL Coder to place the filter's entity and architecture in separate 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

`SplitEntityArch`, `SplitEntityFilePostfix`

SplitEntityArch

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

`'on'`

Selected

Write the generated VHDL code to a single file.

`'off'`(default)

Cleared (default)

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

The names of the entity and architecture files derive from the base file name (as specified by the generating model or subsystem name). By default, postfix strings identifying the file as an entity (`_entity`) or architecture (`_arch `) are appended to the base file name. You can override the default and specify your own postfix string.

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

Note: This property is specific to VHDL code generation. It does not apply to Verilog code generation and should not be enabled when generating Verilog code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`SplitArchFilePostfix`, `SplitEntityFilePostfix`

SplitEntityFilePostfix

Specify string to append to specified model name to form name of generated VHDL entity file

Settings

`'string'`

Default: `'_entity'`

This option applies only if you direct HDL Coder to place the generated VHDL entity and architecture code in separate 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

`SplitArchFilePostfix`, `SplitEntityArch`

SynthesisTool

Specify synthesis tool

Settings

`''` (default)

If you do not specify a synthesis tool, the default is `''`.

`'Altera Quartus II'`

Specify Altera Quartus II as your synthesis tool.

`'Xilinx ISE'`

Specify Xilinx ISE as your synthesis tool.

`'Xilinx Vivado'`

Specify Xilinx Vivado as your synthesis tool.

Usage

To specify Altera Quartus II as the synthesis tool for a DUT subsystem, `myDUT`:

```
hdlset_param (myDUT, 'SynthesisTool', 'Altera Quartus II')
```

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

`SynthesisToolDeviceName` | `SynthesisToolPackageName` | `SynthesisToolSpeedValue`

SynthesisToolChipFamily

Specify target device chip family name

Settings

`'string'`

Default: `''`

Specify the target device chip family name for your model.

To find the chip family name for your target device:

1. At the MATLAB command line, enter:

   ```
   hdlcoder.supportedDevices
   ```

2. Open the linked report and find your target device details.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Functions

`hdlcoder.supportedDevices`

Properties

`SynthesisToolDeviceName` | `SynthesisToolPackageName` | `SynthesisToolSpeedValue`

SynthesisToolDeviceName

Specify target device name

Settings

`'string'`

Default: `''`

Specify the target device name for your model.

To find the name for your target device:

1. At the MATLAB command line, enter:

   ```
   hdlcoder.supportedDevices
   ```

2. Open the linked report and find your target device details.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Functions

`hdlcoder.supportedDevices`

Properties

`SynthesisToolChipFamily` | `SynthesisToolPackageName` | `SynthesisToolSpeedValue`

SynthesisToolPackageName

Specify target device package name

Settings

`'string'`

Default: `''`

Specify the target device package name for your model.

To find the package name for your target device:

1. At the MATLAB command line, enter:

   ```
   hdlcoder.supportedDevices
   ```

2. Open the linked report and find your target device details.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Functions
`hdlcoder.supportedDevices`

Properties
`SynthesisToolChipFamily` | `SynthesisToolDeviceName` | `SynthesisToolSpeedValue`

SynthesisToolSpeedValue

Specify target device speed value

Settings

`'string'`

Default: `''`

Specify the target device speed value for your model.

To find the speed value for your target device:

1. At the MATLAB command line, enter:

   ```
   hdlcoder.supportedDevices
   ```

2. Open the linked report and find your target device details.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Functions
`hdlcoder.supportedDevices`

Properties
`SynthesisToolChipFamily` | `SynthesisToolDeviceName` | `SynthesisToolPackageName`

TargetDirectory

Identify folder into which HDL Coder writes generated output files

Settings

'string'

Default: `'hdlsrc'`

Specify a subfolder under the current working folder into which HDL Coder writes generated files. The string can specify a complete path name.

If the target folder does not exist, the coder creates it.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`VerilogFileExtension`, `VHDLFileExtension`

TargetLanguage

Specify HDL language to use for generated code

Settings

`'VHDL'` (default)

VHDL (default)

Generate VHDL code.

`'Verilog'`

Verilog

Generate Verilog code.

The generated HDL code complies with the following standards:

- VHDL-1993 (IEEE® 1076-1993) or later
- Verilog-2001 (IEEE 1364-2001) or later

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

TestBenchClockEnableDelay

Define elapsed time in clock cycles between deassertion of reset and assertion of clock enable

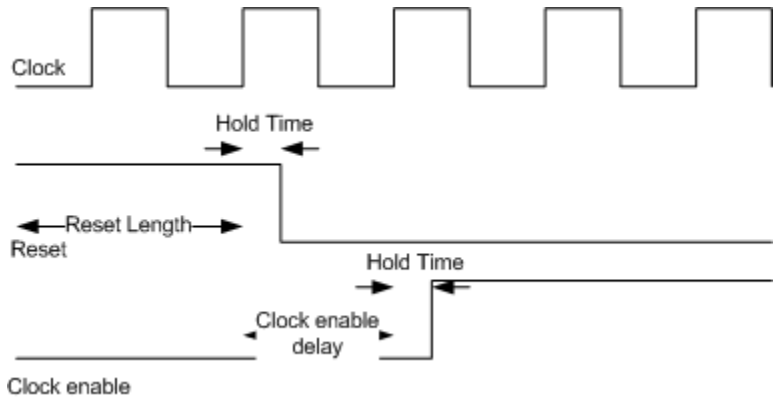
Settings

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`, `ResetLength`

TestBenchDataPostFix

Specify suffix added to test bench data file name when generating multifile test bench

Settings

`'string'`

Default: `'_data'`.

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

`'string'`

Default: `'_tb'`.

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`

TimingControllerArch

Generate reset for timing controller

Settings

`'resettable'`

Generate a reset for the timing controller. If you select this option, the `ClockInputs` property value must be `'Single'`.

`'default'` (default)

Do not generate a reset for the timing controller.

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
`ClockInputs`

Related Examples

- "Generate Reset for Timing Controller"

TimingControllerPostfix

Specify suffix appended to DUT name to form timing controller name

Settings

`'string'`

Default: `'_tc'`.

A timing controller code file is generated if required by the design, for example:

- When code is generated for a multirate model.
- When an area or speed optimization, or block architecture, introduces local multirate.

The timing controller name is based on the name of the DUT. For example, if the name of your DUT is `my_test`, by default, HDL Coder adds the postfix `_tc` to form the timing controller name, `my_test_tc`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`OptimizeTimingController`, “Code Generation from Multirate Models”

TestBenchReferencePostFix

Specify string appended to names of reference signals generated in test bench code

Settings

`'string'`

Default: `'_ref'`.

Reference signal data is represented as arrays in the generated test bench code. The string specified by `TestBenchReferencePostFix` is appended 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`.

Traceability

Enable or disable creation of HTML code generation report with code-to-model and model-to-code hyperlinks

Settings

`'on'`

Create and display an HTML code generation report.

`'off'` (default)

Do not create an HTML code generation report.

Usage Notes

You can use the `RequirementComments` property to generate hyperlinked requirements comments within the HTML code generation report. The requirements comments link to the corresponding requirements documents for your 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

“ Create and Use Code Generation Reports”, “Generate Code with Annotations or Comments”, `RequirementComments`

TriggerAsClock

Use trigger signal in triggered subsystem as a clock

Settings

`'on'`

For triggered subsystems, use the trigger input signal as a clock in the generated HDL code.

`'off'` (default)

For triggered subsystems, do not use the trigger input signal as a clock in the generated HDL code.

Usage Example

Use `hdlset_param` or `makehdl` to set this property.

For example, to generate HDL code that uses the trigger signal as clock for triggered subsystems within the `sfir_fixed/symmetric_fir` DUT subsystem, enter:

```
makehdl ('sfir_fixed/symmetric_sfir','TriggerAsClock','on')
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

More About

- "Use Trigger As Clock in Triggered Subsystems"

UseAggregatesForConst

Specify whether constants are represented by aggregates, including constants that are less than 32 bits

Settings

`'on'`

Selected

Specify that constants, including constants that are less than 32 bits, be represented by aggregates. The following VHDL code show a scalar less than 32 bits represented as an aggregate:

```
GainFactor_gainparam <= (14 => '1',  OTHERS => '0');
```

`'off'` (default)

Cleared(default)

Specify that HDL Coder represent constants less than 32 bits as scalars and constants greater than or equal to 32 bits as aggregates. The following VHDL code was generated by default for a value less than 32 bits:

```
GainFactor_gainparam <= to_signed(16384, 16);
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseRisingEdge`

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`.

UserComment

Specify comment line in header of generated HDL and test bench files

Settings

'string'

The comment is generated in each of the generated code and test bench files. The code generator adds leading comment characters for the target language. When newlines or line feeds are included in the string, the code generator emits single-line comments for each newline.

For example, the following `makehdl` command adds two comment lines to the header in a generated VHDL file.

```
makehdl(gcb,'UserComment','This is a comment line.\nThis is a second line.')
```

The resulting header comment block for subsystem `symmetric_fir` would appear as follows:

```
-- -------------------------------------------------------------
--
-- Module: symmetric_fir
-- Simulink Path: sfir_fixed/symmetric_fir
-- Created: 2006-11-20 15:55:25
-- Hierarchy Level: 0
--
-- This is a comment line.
-- This is a second line.
--
-- Simulink model description for sfir_fixed:
  -- This model shows how to use HDL Coder to check, generate,
  -- and verify HDL for a fixed-point symmetric FIR filter.
--
-- -------------------------------------------------------------
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

UseRisingEdge

Specify VHDL coding style used to detect clock transitions

Settings

`'on'`

Selected

Generated code uses the VHDL `rising_edge` or `falling_edge` function to detect clock transitions.

For example, the following code, generated from a Unit Delay block, uses `rising_edge` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
      END IF;
    END IF;
  END PROCESS Unit_Delay1_process;
```

`'off'` (default)

Cleared (default)

Generated code uses the `'event` syntax.

For example, the following code, generated from a Unit Delay block, uses `clk'event AND clk = '1'` to detect positive clock transitions:

```
Unit_Delay1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Unit_Delay1_out1 <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF clk_enable = '1' THEN
        Unit_Delay1_out1 <= signed(x_in);
```

```
    END IF;
  END IF;
END PROCESS Unit_Delay1_process;
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`LoopUnrolling`, `SafeZeroConcat`, `UseAggregatesForConst`

UseSingleLibrary

Specify whether VHDL code generated for model references is in a single library, or in separate libraries

Settings

`'on'`

Selected

Generate VHDL code for model references into a single library.

`'off'` (default)

Cleared (default)

For each model reference, generate a separate VHDL library.

Note: This property is specific to VHDL code generation. It does not apply to Verilog code generation and should not be enabled when generating Verilog code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`VHDLLibraryName`

UseVerilogTimescale

Use compiler `timescale directives in generated Verilog code

Settings

'on' (default)

Selected (default)

Use compiler `timescale directives in generated Verilog code.

'off'

Cleared

Suppress the use of compiler `timescale directives in generated Verilog code.

Usage Notes

The `timescale directive provides a way of specifying different delay values for multiple modules in a Verilog file. This setting does not affect the generated test bench.

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

See Also

LoopUnrolling, SafeZeroConcat, UseAggregatesForConst, UseRisingEdge

VectorPrefix

Specify string prefixed to vector names in generated code

Settings

'string'

Default: `'vector_of_'`

Specify a string to be prefixed to vector names in generated code.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Verbosity

Specify level of detail for messages displayed during code generation

Settings

Default: 1

0

When `Verbosity` is set to 0, code generation progress messages are not displayed as code generation proceeds. When `Verbosity` is set to 1, more detailed progress messages are displayed.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

VerilogFileExtension

Specify file type extension for generated Verilog files

Settings

'string'

The default file type extension for generated Verilog files is `.v`.

See Also

`TargetLanguage`

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

VHDLArchitectureName

Specify architecture name for generated HDL code

Settings

'string'

The default architecture name is `'rtl'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

VHDLFileExtension

Specify file type extension for generated VHDL files

Settings

'string'

The default file type extension for generated VHDL files is `.vhd`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`TargetLanguage`

VHDLLibraryName

Specify name of target library for generated HDL code

Settings

'string'

The default target library name 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

`HDLCompileInit` | `UseSingleLibrary`

4

Class reference for HDL code generation from Simulink

hdlcoder.WorkflowConfig class

Package: hdlcoder

Configure HDL code generation and deployment workflow

Description

Use the `hdlcoder.WorkflowConfig` object to set HDL workflow options for the `hdlcoder.runWorkflow` function. You can customize the `hdlcoder.WorkflowConfig` object for the following worfklows:

- Generic ASIC/FPGA
- FPGA Turnkey
- IP core generation

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 the `hdlcoder.WorkflowConfig` object. See "Run HDL Workflow Using a Script".

Construction

`hdlcoder.WorkflowConfig('SynthesisTool',tool,'TargetWorkflow', workflow)` creates a workflow configuration object that you use to specify your HDL code generation and deployment workflow.

Input Arguments

`tool` — Synthesis tool

`Xilinx Vivado | Altera QUARTUS II | Xilinx ISE`

Synthesis tool name, specified as a string.

`workflow` — Target workflow

`Generic ASIC/FPGA | FPGA Turnkey | IP Core Generation`

Target workflow, specified as a string.

Properties

Generic ASIC/FPGA Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | string

Path to the folder where your generated project files are saved, specified as a string.

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”.

`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 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` and `GenerateTopLevelWrapper` is `false`.

In the HDL Workflow Advisor, this 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 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 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 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 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 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 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` or `GenerateTopLevelWrapper` is `false`.

In the HDL Workflow Advisor, this 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.

`GenerateRTLTestbench` — 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.

`GenerateCosimulationModel` — Generate cosimulation model

`false` (default) | `true`

Option to generate a cosimulation model, specified as a `logical`. This option requires an HDL Verifier license.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **HDL Code Generation** > **Generate RTL Code and Testbench** task.

`CosimulationModelForUseWith` — Cosimulation tool

`Mentor Graphics ModelSim` (default) | `Cadence Incisive`

Cosimulation tool, specified as a string. This option is available when `GenerateCosimulationModel` is `true`.

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.

`GenerateTopLevelWrapper` — Generate HDL code wrapper and constraint file

`true` | `false`

Generate an HDL code wrapper and a constraint file that contains pin map information and clock constraints, 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) | string

Additional project creation Tcl files you want to include in your synthesis project, specified as a string.

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`. Set to `true` if your tool does not support early timing estimation.

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.

`CriticalPathSource` — Critical path source

`'pre-route'` (default) | `'post-route'`

Critical path source, specified as a string.

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, but omit 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 Turnkey Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | string

Path to the folder where your generated project files are saved, specified as a string.

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”.

`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 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` and `GenerateTopLevelWrapper` is `false`.

In the HDL Workflow Advisor, this 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 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 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 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 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 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 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` or `GenerateTopLevelWrapper` is `false`.

In the HDL Workflow Advisor, this is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** 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`. This task is available only when the target workflow is `FPGA Turnkey` and `GenerateTopLevelWrapper` is `true`.

In the HDL Workflow Advisor, this 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` and `GenerateTopLevelWrapper` is `true`.

In the HDL Workflow Advisor, this is the **HDL Workflow Advisor** > **Download to Target** > **Program Target Device** 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.

`GenerateRTLTestbench` — 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.

`GenerateCosimulationModel` — Generate cosimulation model

`false` (default) | `true`

Option to generate a cosimulation model, specified as a `logical`. This option requires an HDL Verifier license.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **HDL Code Generation** > **Generate RTL Code and Testbench** task.

`CosimulationModelForUseWith` — Cosimulation tool

`Mentor Graphics ModelSim` (default) | `Cadence Incisive`

Cosimulation tool, specified as a string. This option is available when `GenerateCosimulationModel` is `true`.

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.

`GenerateTopLevelWrapper` — Generate HDL code wrapper and constraint file
`true` | `false`

Generate an HDL code wrapper and a constraint file that contains pin map information and clock constraints, 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) | string

Additional project creation Tcl files you want to include in your synthesis project, specified as a string.

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`. Set to `true` if your tool does not support early timing estimation.

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.

`CriticalPathSource` — Critical path source

`'pre-route'` (default) | `'post-route'`

Critical path source, specified as a string.

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, but omit 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.

IP Core Generation Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | string

Path to the folder where your generated project files are saved, specified as a string.

Example: `'project_file_folder'`

`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 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 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 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 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 is the **HDL Workflow Advisor > Embedded System Integration > Program Target Device** task.

`IPCoreRepository` — IP core repository folder path

`''` (default) | string

Full path to an IP core repository folder, specified as a string. 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 about the documentation, 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:

`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".

`OperatingSystem` — Operating system

`''` (default) | string

Operating system for embedded processor, specified as a string. 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 reboots 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.

`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) | string

Full path to a custom synthesis tool build Tcl script file, specified as a string. 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, the following 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: `''L:\sandbox\work\build.tcl'`

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 Using 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.

The script in this example 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 8.6 (R2015b) at 11:33:25 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\generic_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir1'

%% Load the Model
load_system('sfir_fixed');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir1');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg1761');
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', ...
      'Generic ASIC/FPGA');

% 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 Generate RTL Code And Testbench Task
hWC.GenerateRTLCode = true;
hWC.GenerateRTLTestbench = false;
hWC.GenerateCosimulationModel = false;
hWC.CosimulationModelForUseWith = 'Mentor Graphics ModelSim';
hWC.GenerateValidationModel = false;
hWC.GenerateTopLevelWrapper = false;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Run Synthesis Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set Properties related to Run Implementation Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set Properties related to Annotate Model With Synthesis Result 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_fir1', 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 Turnkey Workflow Using 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.

The script in this example 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 Outport 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 Outport 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 Outport 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 Outport 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 Outport 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 Generate RTL Code And Testbench Task
hWC.GenerateTopLevelWrapper = true;

% 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 Using 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.

The script in this example 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
```

- "Run HDL Workflow Using a Script"

See Also

Functions

`hdlcoder.runWorkflow`

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 single 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) | string

Full path to the exported MATLAB script file, specified as a string. If the string 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, specified as a `logical`.

`'Comments'` — Include comments

`true` (default) | `false`

Specify whether to include comments in the command list or script, specified as a `logical`.

`'Headers'` — Include headers

`true` (default) | `false`

Specify whether to include a header in the command list or script, specified as a `logical`.

`'DUT'` — Full path to DUT

`''` (default) | string

Full path to the DUT, specified as a string.

Example: `'hdlcoder_led_blinking/led_counter'`

See Also

Classes

hdlcoder.WorkflowConfig

Related Examples

- "Run HDL Workflow Using 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.

Use this method if you do not want to enable each task individually.
For example, if you want to run a all tasks but one, you can run `hdlcoder.WorkflowConfig.setAllTasks`, then disable the task you want to skip.

See Also

Functions

`hdlcoder.WorkflowConfig.clearAllTasks`

Classes

hdlcoder.WorkflowConfig

Related Examples

- "Run HDL Workflow Using 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.

Use this method if you do not want to disable each task individually. For example, if you want to run a single task, you can run `hdlcoder.WorkflowConfig.clearAllTasks`, then enable the task you want to run.

See Also

Functions

`hdlcoder.WorkflowConfig.setAllTasks`

Classes

hdlcoder.WorkflowConfig

Related Examples

- "Run HDL Workflow Using a Script"

Introduced in R2015b

validate

Class: hdlcoder.WorkflowConfig
Package: hdlcoder

Check property values in HDL Workflow CLI configuration object

Syntax

```
validate
```

Description

`validate` checks that the `hdlcoder.WorkflowConfig` object has acceptable values for all required properties, and that property values have valid data types. If validation fails, the software displays an error message.

See Also

hdlcoder.WorkflowConfig

Related Examples

- "Run HDL Workflow Using 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 Using a Script".

Examples

Run Default Workflow

To run the HDL workflow with default settings for a DUT subsystem, `modelname/DUT`, at the command line, enter:

```
hdlcoder.runWorkflow('modelname/DUT');
```

Run Workflow With Configuration Object

To run the HDL workflow with a workflow configuration object, `hWC`, for a DUT subsystem, `modelname/DUT`, at the command line, enter:

```
hdlcoder.runWorkflow('modelname/DUT', hWC);
```

- "Run HDL Workflow Using a Script"

Input Arguments

`DUT` — Full path to DUT

`''` (default) | string

Full path to the DUT, specified as a string.

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

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:

`hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`	Optimize the design frequency to achieve the maximum clock frequency `hdlcoder.OptimizationConfig.ExplorationMode` is the default.
`hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`	Optimize the design frequency 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) | string

Name of folder that contains previously-generated optimization iteration data, specified as a string. The folder is a subfolder of `hdlexpl`, and the folder name begins with the string, `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`

5

Function Reference for HDL Code Generation from MATLAB

codegen

Generate HDL code from MATLAB code

Syntax

```
codegen -config hdlcfg matlab_design_name
codegen -config hdlcfg -float2fixed fixptcfg matlab_design_name
```

Description

`codegen -config hdlcfg matlab_design_name` generates HDL code from MATLAB code.

`codegen -config hdlcfg -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
```

- "Generate HDL Code from MATLAB Code Using the Command Line Interface"

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

string

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.config` | coder.FixptConfig | coder.HdlConfig

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]);
```

- "Replace the exp Function with a Lookup Table"
- "Replace a Custom Function with a Lookup Table"

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 single 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`

More About

- "Replacing Functions Using Lookup Table Approximations"

See Also

Classes

coder.FixptConfig

Functions

`codegen`

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
```

- "Generate HDL Code from MATLAB Code Using the Command Line Interface"

See Also

`codegen` | coder.FixptConfig | coder.HdlConfig

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

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, `fxpCfg`, with a test bench, `myTestbenchName`.

```
fxpCfg = coder.config('fixpt');
fxpCfg.TestBenchName = 'myTestbenchName';
fxpCfg.addFunctionReplacement('min', 'fi_min');
codegen -float2fixed fxpCfg designName
```

Specify that the floating-point function, `min`, should be replaced with the fixed-point function, `fi_min`.

```
fxpCfg.addFunctionReplacement('min', 'fi_min');
```

When you generate code, the code generation software 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.config` | coder.FixptConfig

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')
```

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
```

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
```

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')
```

6

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`

`LogIOForComparisonPlotting`

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

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
```

- "Generate HDL Code from MATLAB Code Using the Command Line Interface"

Alternatives

You can also generate HDL code from MATLAB code using the HDL Workflow Advisor. For more information, see "HDL Code Generation from a MATLAB Algorithm".

See Also

`codegen` | `coder.config` | coder.HdlConfig

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, specified as a string.

DistributedPipeliningPriority Value	Description
`Numerical Integrity` (default)	Prioritize numerical integrity when distributing pipeline registers. 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.
`Performance`	Prioritize performance over numerical integrity. 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. 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.

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, specified as a string. 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 Properties. 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, specified as a string.

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

HDLLintInit

HDL lint script initialization string.

Value: string

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` using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

Value: string

HDLLintTerm

HDL lint script termination string.

Value: string

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, specified as a string. See “Optimize MATLAB Loops”.

LoopOptimization Value	Description
`LoopNone` (default)	Do not optimize loops in generated code.
`StreamLoops`	Stream loops.
`UnrollLoops`	Unroll Loops.

Values: `'LoopNone'` (default) | `'StreamLoops'` | `'UnrollLoops'`

`MinimizeClockEnables`

Specify whether to omit generation of clock enable logic.

When `true`, omit generation of clock enable logic wherever possible.

When `false`, generate clock enable logic.

Values: `false` (default) | `true`

`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

`PartitionFunctions`

Generate instantiable HDL code modules from functions.

Note: If you enable `PartitionFunctions`, `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, specified as a string.

Values: `'ModelSim'` (default) | `'ISIM'`

`SynthesisTool`

Synthesis tool name, specified as a string.

Values: `'Xilinx ISE'` (default) | `'Altera Quartus II'` | `'Xilinx Vivado'`

`SynthesisToolChipFamily`

Synthesis target chip family name, specified as a string.

Values: `'Virtex4'` (default) | string

`SynthesisToolDeviceName`

Synthesis target device name, specified as a string.

Values: `'xc4vsx35'` (default) | string

SynthesisToolPackageName

Synthesis target package name, specified as a string.

Values: `'ff668'` (default) | string

SynthesisToolSpeedValue

Synthesis target speed, specified as a string.

Values: `'-10'` (default) | string

SynthesizeGeneratedCode

Synthesize generated code or not, specified as a `logical`.

Values: `false` (default) | `true`

TargetLanguage

Target language, specified as a string.

Values: `'VHDL'` (default) | `'Verilog'`

TestBenchName

Test bench function name, specified as a string. You must specify a test bench.

Values: `''` (default) | string

TimingControllerArch

Timing controller architecture, specified as a string.

TimingControllerArch Value	Description
`default` (default)	Do not generate a reset for the timing controller.
`resettable`	Generate a reset for the timing controller.

Values: `'default'` (default) | `'resettable'`

TimingControllerPostfix

Postfix to append to design name to form name of timing controller, specified as a string.

Values: `'_tc'` (default) | string

`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`.

UseMatrixTypesInHDL Value	Description
`false` (default)	Generate HDL vectors with index computation logic for MATLAB matrices. This option can use more area in the synthesized hardware.
`true`	Generate HDL matrices for MATLAB matrices. This option can save area in the synthesized hardware. The following requirements apply: • Matrix elements cannot be complex or `struct` data types. • You cannot use linear indexing to specify matrix elements. For example, if you have a 3x3 matrix, `A`, you cannot use `A(4)`. Instead, use `A(2,1)`. You can also use a colon operator in either the row or column subscript, but not both. For example, you can use `A(3,1:3)` and `A(2:3,1)`, but not `A(2:3, 1:3)`. • If you enable `PartitionFunctions`, `UseMatrixTypesInHDL` has no effect.

Values: `false` (default) | `true`

`VHDLLibraryName`

Target library name for generated VHDL code, specified as a string.

Values: `'work'` (default) | string

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, specified as a string. 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, specified as a string.

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 string. You must override the default value and specify a valid board name.

Values: `'Choose a board'` (default) | string

`FILBoardIPAddress`

IP address of the FPGA board, specified as a string. You must enter a valid IP address.

Values: `192.168.0.2` (default) | string

`FILBoardMACAddress`

MAC address of the FPGA board, specified as a string. You must enter a valid MAC address.

Values: `00-0A-35-02-21-8A` (default) | string

`FILAdditionalFiles`

List of additional source files to include, specified as a string. Separate file names with a semi-colon (";").

Values: `''` (default) | string

`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
```

- "Generate HDL Code from MATLAB Code Using the Command Line Interface"

Alternatives

You can also generate HDL code from MATLAB code using the HDL Workflow Advisor. For more information, see "HDL Code Generation from a MATLAB Algorithm".

See Also

Functions

`codegen` | `coder.config` | `hdlcoder.CodingStandard`

Classes

coder.FixptConfig

Properties
HDL Coding Standard Customization Properties

7

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')
```

```
cso =

                  ShowPassingRules.enable: true
                      HDLKeywords.enable: true    [CGSL-1.A.A.3]
          DetectDuplicateNamesCheck.enable: true    [CGSL-1.A.A.5]
          SignalPortParamNameLength.enable: true    [CGSL-1.A.B.1]
          SignalPortParamNameLength.length: [2 40]  [CGSL-1.A.B.1]
     ModuleInstanceEntityNameLength.enable: true    [CGSL-1.A.C.3]
     ModuleInstanceEntityNameLength.length: [2 32]  [CGSL-1.A.C.3]
                 InitialStatements.enable: true    [CGSL-2.C.D.1]
                       IfElseChain.enable: true    [CGSL-2.G.C.1c]
                       IfElseChain.length: 7       [CGSL-2.G.C.1c]
                    IfElseNesting.enable: true    [CGSL-2.G.C.1a]
                     IfElseNesting.depth: 3       [CGSL-2.G.C.1a]
```

```
MinimizeVariableUsage.enable: true    [CGSL-2.G]
   MultiplierBitWidth.enable: true    [CGSL-2.J.F.5]
    MultiplierBitWidth.width: 16      [CGSL-2.J.F.5]
           LineLength.enable: true    [CGSL-3.A.D.5]
           LineLength.length: 110     [CGSL-3.A.D.5]
     NonIntegerTypes.enable: true    [CGSL-3.B.D.1]
```

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, *`cso`*.

```
cso = hdlcoder.CodingStandard('Industry')
```

```
cso =

                  ShowPassingRules.enable: true
```

```
                    HDLKeywords.enable: true    [CGSL-1.A.A.3]
         DetectDuplicateNamesCheck.enable: true    [CGSL-1.A.A.5]
         SignalPortParamNameLength.enable: true    [CGSL-1.A.B.1]
         SignalPortParamNameLength.length: [2 40]  [CGSL-1.A.B.1]
    ModuleInstanceEntityNameLength.enable: true    [CGSL-1.A.C.3]
    ModuleInstanceEntityNameLength.length: [2 32]  [CGSL-1.A.C.3]
                InitialStatements.enable: true    [CGSL-2.C.D.1]
                      IfElseChain.enable: true    [CGSL-2.G.C.1c]
                      IfElseChain.length: 7       [CGSL-2.G.C.1c]
                   IfElseNesting.enable: true    [CGSL-2.G.C.1a]
                    IfElseNesting.depth: 3       [CGSL-2.G.C.1a]
           MinimizeVariableUsage.enable: true    [CGSL-2.G]
              MultiplierBitWidth.enable: true    [CGSL-2.J.F.5]
               MultiplierBitWidth.width: 16      [CGSL-2.J.F.5]
                     LineLength.enable: true    [CGSL-3.A.D.5]
                     LineLength.length: 110     [CGSL-3.A.D.5]
                NonIntegerTypes.enable: true    [CGSL-3.B.D.1]
```

Customize the coding standard options as follows:

- Do not show passing rules in the coding standard report.
- Set the 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 and check it according to the customized HDL coding standard rules. In this example, the model is *`sfir_fixed`*, with a DUT subsystem, *`symmetric_fir`*.

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry',...
        'HDLCodingStandardCustomizations',cso)
```

- “Generate an HDL Coding Standard Report from Simulink”
- “Generate an HDL Coding Standard Report from MATLAB”

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.

More About

- "HDL Coding Standard Rules"
- "HDL Coding Standard Report"

See Also

Properties

HDL Coding Standard Customization Properties | `HDLCodingStandardCustomizations`

HDL Coding Standard Customization Properties

Customize HDL coding standard

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 uncompilable or unsynthesizable, the coder does not follow the rule.

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 `true` to show passing rules in coding standard report. Set to `false` to show only rules with errors or warnings. The default is `true`.

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 `true` to check for HDL keywords in design names. Set to `false` if you do not want to check for HDL keywords in design names. The default is `true`.

`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 `true` to check for duplicate names in the design. Set to `false` if you do not want to check for duplicate names in the design. The default is `true`.

`SignalPortParamNameLength` — Check signal, port, and parameter name length

structure

Check for signal, port, and parameter name lengths (rule CGSL-1.A.B.1), specified as a structure with the following fields.

Field	Description
enable	Set to `true` to check the length of signal, port, and parameter names. Set to `false` if you do not want to check the length of signal, port, and parameter names. The default is `true`.

Field	Description
length	Minimum and maximum length of signal, port, and parameter names, specified as a 2-element array of positive integers. The first element is the minimum length, and the second element is the maximum length. The default is [2 40].

`ModuleInstanceEntityNameLength` — Check module, instance, and entity name length

structure

Check for module, instance, and entity name lengths (rule CGSL-1.A.C.3), specified as a structure with the following fields.

Field	Description
enable	Set to `true` to check the length of module, instance, and entity names. Set to `false` if you do not want to check the length of module, instance, and entity names. The default is `true`.
length	Minimum and maximum length of module, instance, and entity name names, specified as a 2-element array of positive integers. The first element is the minimum length, and the second element is the maximum length. The default is [2 32].

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	Set to `true` to minimize clock enables in the generated code and check for clock enable signals after code generation. Set to `false` if you do not want to check for clock enable signals in the generated code. The default is `false`.

`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	Set to `true` to minimize reset signals in the generated code and check for reset signals after code generation. Set to `false` if you do not want to check for reset signals in the design. The default is `false`.

`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	Set to `true` to minimize reset signals in the generated code and check for reset signals after code generation. Set to `false` if you do not want to check for reset signals in the design. The default is `false`.

`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	Set to `true` to check for asynchronous reset signals in the generated code. Set to `false` if you do not want to check for asynchronous reset signals in the generated code. The default is `true`.

`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	Set to `true` to minimize use of variables. Set to `false` if you do not want to minimize use of variables. The default is `true`.

`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	Set to `true` to check if-else statement nesting depth. Set to `false` if you do not want to check if-else statement nesting depth.

Field	Description
	The default is `true`.
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 `true` to check if-else statement chain length. Set to `false` if you do not want to check if-else statement chain length. The default is `true`.
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 `true` to check multiplier bit width. Set to `false` if you do not want to check multiplier bit width. The default is `true`.

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 `true` to check line lengths in generated code. Set to `false` if you do not want tocheck line lengths in generated code. The default is `true`.
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.

Field	Description
enable	Set to `true` to check for non-integer constants. Set to `false` if you do not want to check for non-integer constants.

Field	Description
	The default is `true`.

See Also

`hdlcoder.CodingStandard`

Related Examples

- "Generate an HDL Coding Standard Report from MATLAB"
- "Generate an HDL Coding Standard Report from Simulink"

More About

- "HDL Coding Standard Rules"
- "HDL Coding Standard Report"

hdl.BlackBox System object

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.

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) | string

HDL name for clock enable input port, specified as a string.

`ClockInputPort` — Clock input port name

`'clk'` (default) | string

HDL name for clock input port, specified as a string.

`EntityName` — Module or entity name

System object instance name (default) | string

VHDL entity or Verilog module name generated for the black box System object, specified as a string.

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) | string

HDL name for reset input port, specified as a string.

`VHDLArchitectureName` — VHDL architecture name

`'rtl'` (default) | string

VHDL architecture name, specified as a string. The coder generates the architecture name only if `InlineConfigurations` is `'on'`.

`VHDLComponentLibrary` — VHDL component library name

`'work'` (default) | string

Library from which to load the VHDL component, specified as a string.

`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

Related Examples

- “Integrate Custom HDL Code Into MATLAB Design”
- “Generate a Board-Independent IP Core from MATLAB”

More About

- “Generate Black Box Interface for Subsystem”

hdl.RAM System object

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 data is scalar, HDL Coder infers a single RAM block. If your data is a vector, HDL Coder infers an array of parallel RAM banks.

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 as shown in the following table.

Data type of address variable	RAM address size (bits)
`single` or `double`	16
`uint`*N*	*N*
`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 as shown in the following table.

Data type of address variable	RAM address size (bits)
`single` or `double`	16
`uint`*N*	*N*
`embedded.fi`	`WordLength`

The maximum RAM bank address size is 32 bits.

Construction

`H = hdl.RAM` creates a single port RAM System object. This object allows you to read from or write to a memory location. The output data port corresponds to the read/write address passed in with the step method.

`H = hdl.RAM(Name,Value)` creates a single, simple dual, or dual port `hdl.RAM` System object, `H`, with each specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. See "Properties" on page 7-19 for the list of available property names.

Properties

RAMType

Type of RAM to be created

Default: `Single port`

Specify the type of RAM to be created. Values of this property are:

`Single Port`	Create a single port RAM, with 3 inputs and 1 output. Inputs: • Write Data • Address • Write enable Output: Read data
`Simple dual port`	Create a simple dual-port RAM, with 4 inputs and 1 output. Inputs:

	• Write Data • Write address • Write enable • Read address Output: Output data from read address
`Dual port`	Create a dual-port RAM, with 4 inputs and 2 outputs. Inputs: • Write Data • Write address • Write enable • Read address Outputs: • Output data from write address • Output data from read address

`WriteOutputValue`

Behavior for Write output

Default: `New data`

Specify the behavior for Write output for single-port and dual-port RAMs. Values of this property are:

`New data`	Send out new data at the address to the output.
`Old data`	Send out old data at the address to the output.

Methods

step	Read or write input value to memory location

Examples

Create Single-Port RAM System Object

Construct System object to read from or write to a memory location in RAM.

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.

```
H = hdl.RAM('RAMType','Single port','WriteOutputValue','Old data')
```

```
H =

  System: hdl.RAM

  Properties:
             RAMType: 'Single port'
    WriteOutputValue: 'Old data'
```

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.

```
H = hdl.RAM('RAMType','Simple dual port')
```

```
H =

  System: hdl.RAM

  Properties:
    RAMType: 'Simple dual port'
```

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.

```
H = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data')

H =

  System: hdl.RAM

  Properties:
             RAMType: 'Dual port'
    WriteOutputValue: 'New data'
```

Read/Write Single-Port RAM

Create System object that can write to a single port RAM, and read the newly written value out.

Construct single-port RAM System object.

```
hRAM = hdl.RAM('RAMType','Single port','WriteOutputValue','New data');
```

Preallocate memory.

```
dataLength    = 100;
[dataIn dataOut] = deal(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   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = step(hRAM,dataIn(ii),addressIn,writeEnable);
end ;
```

See Also

Blocks

`Dual Port RAM` | `Dual Rate Dual Port RAM` | `Simple Dual Port RAM` | `Single Port RAM`

More About

- “Implement RAM Using MATLAB Code”
- “HDL Code Generation for System Objects”
- “System Objects”

step

System object: hdl.RAM

Read or write input value to memory location

Syntax

```
DATAOUT = step(H,WRITEDATA,READWRITEADDRESS,WRITEENABLE)
READDATAOUT = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)
[WRITEDATAOUT,READDATAOUT] =
step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)
```

Description

`DATAOUT = step(H,WRITEDATA,READWRITEADDRESS,WRITEENABLE)` allows you to read the value in memory location READWRITEADDRESS when WRITEENABLE is `false`. It also allows you to write the value WRITEDATA into the memory location READWRITEADDRESS when WRITEENABLE is `true`. DATAOUT is the new or old data at READWRITEADDRESS when WRITEENABLE is `true`, or the data at READWRITEADDRESS when WRITEENABLE is `false`. This step syntax is appropriate for a single-port RAM System object.

`READDATAOUT = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)` allows you to write the value WRITEDATA into memory location WRITEADDRESS when WRITEENABLE is `true`. READDATAOUT is the old data at the address location READADDRESS. This step syntax is appropriate for a simple dual-port RAM System object.

`[WRITEDATAOUT,READDATAOUT] = step(H,WRITEDATA,WRITEADDRESS,WRITEENABLE,READADDRESS)` allows you to write the value WRITEDATA into the memory location WRITEADDRESS when WRITEENABLE is `true`. WRITEDATAOUT is the new or old data at memory location WRITEADDRESS. READDATAOUT is the old data at the address location READADDRESS. This step syntax is appropriate for a dual-port RAM System object.

hdl.RAM Input Requirements

Inputs must either be all scalar, or all vectors of the same size.

Input	Data Type	Requirement
WRITEDATA	This value can be double, single, integer, or a fixed-point (fi) object, and can be real or complex.	Can be scalar or vector.
WRITEENABL	This value must be logical.	Can be scalar or vector. If WRITEDATA is a vector, this value must be a vector of the same size.
WRITEADDRE and READADDRES	This value can be either fixed-point (fi) objects or integers, and must be real and unsigned.	Can be scalar or vector. If WRITEDATA is a vector, this value must be a vector of the same size.

Examples

Read/Write Single-Port RAM with Scalar Data

Create System object that can write to a single-port RAM and read the newly written value out.

Construct single-port RAM System object.

```
hRAM = hdl.RAM('RAMType','Single port','WriteOutputValue','New data');
```

Preallocate memory.

```
dataLength    = 100;
[dataIn dataOut] = deal(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   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = step(hRAM,dataIn(ii),addressIn,writeEnable);
```

```
end ;
```

Read/Write Dual-Port RAM with Vector Data

Create System object that can write vector data to a dual-port RAM and read vector data out.

Construct dual-port RAM System object.

```
persistent ramObj;
hRAM = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
```

Create vector write data, addresses, and enables, and allocate memory for the read data.

```
% RAM inputs
ramDataIn = fi(randi(1,4),0,16,0);
ramWriteEn = [true, false, false, true];
% address range [a,b]
a = 0;
b = 7;
ramReadAddr = fi(round((b-a).*rand(1,4) + a), 0, 3, 0);
ramWriteAddr = fi(round((b-a).*rand(1,4) + a), 0, 3, 0);
```

Write vector data to the System object, and read vector data out.

```
% RAM access
[wrout,rdout] = step(ramObj, ramDataIn,ramWriteAddr,ramWriteEn,ramReadAddr);
```

- “Implement RAM Using MATLAB Code”

See Also

hdl.RAM

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.

This pragma does not affect MATLAB simulation behavior.

Note: If you specify the `coder.unroll` pragma, it takes precedence over `coder.hdl.loopspec`, and `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, it takes precedence over `coder.hdl.loopspec`, and `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, it takes precedence over `coder.hdl.loopspec`, and `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, it takes precedence over `coder.hdl.loopspec`, and `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 2 iterations and contains 5 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 5 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

More About

- “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-hand side.
- On a sub-expression.
- 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, an expression containing the `t` or `pvar` variables cannot be pipelined:

```
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 "Register Inputs and Outputs".

`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-hand side.
- On a sub-expression.
- 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, an expression containing the `t` or `pvar` variables cannot be pipelined:

```
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 "Register Inputs and Outputs".

Examples

Insert one pipeline register at output of MATLAB expression

Insert a single pipeline register at the output of a MATLAB expression, `a + b * c`.

```
y = coder.hdl.pipeline(a + b * c);
```

Insert multiple pipeline registers at output of MATLAB expression

Insert three pipeline registers at the output of a MATLAB expression, `a + b * c`.

```
y = coder.hdl.pipeline(a + b * c, 3);
```

Insert pipeline registers at intermediate stage of MATLAB expression

For a MATLAB expression, `a + b * c`, insert five pipeline registers after the computation of `b * c`.

```
y = a + coder.hdl.pipeline(b * c, 5);
```

Insert pipeline registers at intermediate stage and at output of MATLAB expression

Use nested `coder.hdl.pipeline` pragmas to insert pipeline registers at an intermediate stage and at the output of a MATLAB expression.

For a MATLAB expression, `a + b * c`, insert five pipeline registers after the computation of `b * c`, and two pipeline registers at the output of the whole expression.

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
```

- "Pipeline MATLAB Expressions"

Input Arguments

`expr` — MATLAB expression to pipeline

MATLAB expression

MATLAB expression to pipeline. Insert pipeline registers at the output of this expression in the generated HDL code.

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

More About

- "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) | string

Board name, specified as a string. 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 string.

Example: `'Xilinx'`

`FPGAFamily` — FPGA family name

`''` (default) | string

FPGA family name, specified as a string.

Example: `'Zynq'`

`FPGADevice` — FPGA device identifier

`''` (default) | string

FPGA device identifier, specified as a string.

Example: `'xc7z020'`

`FPGAPackage` — FPGA package identifier for Xilinx devices

`''` (default) | string

FPGA package identifier for Xilinx devices, specified as a string.

For Altera devices, this property is ignored.

Example: `'clg484'`

`FPGASpeed` — FPGA speed for Xilinx devices

`''` (default) | string

FPGA speed for Xilinx devices, specified as a string.

For Altera devices, this property is ignored.

Example: `'-1'`

`SupportedTool` — Supported synthesis tool

`''` (default) | cell array of strings

Synthesis tool or tools that support this board, specified as a cell array of strings. 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

addExternalIOInterface	Define external IO interface for board object
addExternalPortInterface	Define external port interface for board object
validateBoard	Check property values in board object

See Also

hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- “Register a Custom Board”
- “Register a Custom Reference Design”

More About

- “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.

Tips

- For details about the external IO interface ports, pins, and constraints for your board, view the board documentation.

Input Arguments

`interfacename` — Interface name

string

Interface name, specified as a string. 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 string. 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 `Outport` DUT ports.

Example: `'OUT'`

`portname` — Port name
string

Board top-level port name, specified as a string.

Example: `'GPLEDs'`

`portwidth` — Port bit width
positive integer

Port bit width, specified as a positive integer.

Example: 4

`pins` — Pin names
cell array of strings

FPGA pin names, specified as a cell array of strings.

Example: `{'H18','AA14','AA13','AB15'}`

`constraints` — IO pad constraints
`{}` (default) | cell array of strings

IO pad constraints, specified as a cell array of strings.

Example: `{'IOSTANDARD = LVCMOS25'}`

Example: `{'IOSTANDARD = LVCMOS25','SLEW = SLOW'}`

See Also

hdlcoder.Board | hdlcoder.Board.addExternalPortInterface

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- “Register a Custom Board”

More About

- “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.

Tips

- To get IO constraint names for your board, view the board documentation.

Input Arguments

`constraints` — IO pad constraints

`{}` (default) | cell array of strings

IO pad constraints, specified as a cell array of strings.

Example: `{'IOSTANDARD = LVCMOS25'}`

Example: `{'IOSTANDARD = LVCMOS25','SLEW = SLOW'}`

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.addExternalIOInterface | hdlcoder.Board

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"

More About

- "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

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"

More About

- "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.

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 string.

Example: `'Altera Quartus II'`

Properties

`ReferenceDesignName` — Reference design name

`''` (default) | string

Reference design name, specified as a string. In the HDL Workflow Advisor, this name appears in the **Reference design** dropdown list.

Example: `'Default system (Vivado 2014.2)'`

`BoardName` — Board name

`''` (default) | string

Board associated with this reference design, specified as a string.

Example: `'Enclustra Mars ZX3 with PM3 base board'`

`SupportedToolVersion` — Supported tool version

`{}` (default) | cell array of strings

One or more tool versions that work with this reference design, specified as a cell array of strings.

Example: `{'2014.2'}`

Example: `{'13.7','14.0'}`

`CustomConstraints` — Design constraint file (optional)

`{}` (default) | cell array of strings

One or more design constraint files, specified as a cell array of strings. This property is optional.

Example: `{'MarsZX3_PM3.xdc'}`

Example: `{'MyDesign.qsf'}`

`CustomFiles` — Relative path to required file or folder (optional)

`{}` (default) | cell array of strings

One or more relative paths to files or folders that the reference design requires, specified as a cell array of strings. 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'}`.

Example: `{'my_ip_core'}`

Example: `{fullfile('data', 'ps7_system_prj.xml')}`

Example: `{'ip'}`

Example: `{'ipcore'}`

Example: `{'pcores'}`

Methods

addAXI4SlaveInterface	Add and define AXI4 slave interface
addInternalIOInterface	Add and define internal IO interface between generated IP core and existing IP cores
addClockInterface	Add clock and reset interface
addCustomEDKDesign	Specify Xilinx EDK MHS project file
addCustomQsysDesign	Specify Altera Qsys project file
addCustomVivadoDesign	Specify Xilinx Vivado exported block design Tcl file
validateReferenceDesign	Check property values in reference design object

See Also

hdlcoder.Board

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"
- "Register a Custom Reference Design"

More About

- "Board and Reference Design Registration System"

Introduced in R2015a

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)
```

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.

Tips

- Before running this method, you must run the hdlcoder.ReferenceDesign.addClockInterface method.

Input Arguments

`ref_design_port` — Reference design port name

`''` (default) | string

Reference design port that is connected to the AXI4 or AXI4-Lite interface, specified as a string.

Example: `'axi_interconnect_0/M00_AXI'`

`base_addr` — Base address

`''` (default) | string

Base address for AXI4 or AXI4-Lite slave interface, specified as a string.

Example: `'0x40010000'`

`master_addr_space` — Master interface address space (Vivado only)

`''` (default) | string

Address space of the master interface connected to this slave interface, specified as a string. For Vivado reference designs only.

Example: `'processing_system7_0/Data'`

See Also

hdlcoder.ReferenceDesign.addClockInterface | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"
- "Register a Custom Reference Design"

More About

- "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) | string

Custom internal IO interface name, specified as a string. 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 string.

For example, if the interface is an input to the generated IP core, set `interface_type` to `'IN'`.

`port_name` — Port name

`''` (default) | string

Name of generated IP core port in the HDL code, specified as a string.

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) | string

Internal IO interface port to connect with generated IP core port, specified as a string. 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

Related Examples

- “Define and Register Custom Board and Reference Design for SoC Workflow”
- “Register a Custom Board”
- “Register a Custom Reference Design”

More About

- “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)
```

Description

`addClockInterface('ClockConnection',clock_port,'ResetConnection', reset_port)` adds a clock and reset interface to an `hdlcoder.ReferenceDesign` object.

Tips

- You must run this method before running the hdlcoder.ReferenceDesign.addClockInterface method.

Input Arguments

`clock_port` — Clock port name

`''` (default) | string

Reference design port that is connected to the IP core clock port, specified as a string.

Example: `'processing_system7_1/FCLK_CLK0'`

`reset_port` — Reset port name

`''` (default) | string

Reference design port that is connected to the IP core reset port, specified as a string.

Example: `'proc_sys_reset/peripheral_aresetn'`

See Also

hdlcoder.ReferenceDesign.addAXI4SlaveInterface | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"
- "Register a Custom Reference Design"

More About

- "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.

Tips

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Altera Quartus II, use the `addCustomQsysDesign` method.

Input Arguments

`mhs_project_file` — MHS project file

string

MHS project file for Xilinx EDK embedded system design, specified as a string.

Example: `'system.mhs'`

See Also

hdlcoder.ReferenceDesign.addCustomQsysDesign | hdlcoder.ReferenceDesign.addCustomVivadoDesign | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- “Register a Custom Board”
- “Register a Custom Reference Design”

More About

- “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.

Tips

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Xilinx ISE, use the `addCustomEDKDesign` method.

Input Arguments

`qsys_project_file` — Qsys project file

string

Qsys project file for Altera Qsys embedded system design, specified as a string.

Example: `'system_soc.qsys'`

See Also

hdlcoder.ReferenceDesign.addCustomEDKDesign | hdlcoder.ReferenceDesign.addCustomVivadoDesign | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- “Register a Custom Board”
- “Register a Custom Reference Design”

More About

- “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.

Tips

- 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.

Input Arguments

`bd_tcl_file` — Block design Tcl file

string

Block design Tcl file that you exported from your Xilinx Vivado embedded system design project, specified as a string. The Tcl file name must be the same as the Vivado block diagram name.

Example: `'system_top.tcl'`

See Also

hdlcoder.ReferenceDesign.addCustomQsysDesign | hdlcoder.ReferenceDesign.addCustomEDKDesign | hdlcoder.ReferenceDesign

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- "Register a Custom Board"
- "Register a Custom Reference Design"

More About

- "Board and Reference Design Registration System"

Introduced in R2015a

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

Related Examples

- Define and Register Custom Board and Reference Design for SoC Workflow
- “Register a Custom Board”
- “Register a Custom Reference Design”

More About

- “Board and Reference Design Registration System”

Introduced in R2015a

8

System object Reference

matlab.System class

Package: matlab

Base class for System objects

Description

`matlab.System` is the base class for System objects. In your class definition file, you must subclass your object from this base class (or from another class that derives from this base class). Subclassing allows you to use the implementation and service methods provided by this base class to build your object. Type this syntax as the first line of your class definition file to directly inherit from the `matlab.System` base class, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System
```

Note: You must set `Access = protected` for each `matlab.System` method you use in your code.

Methods

resetImpl	Reset System object states
setupImpl	Initialize System object
stepImpl	System output and state update equations

Attributes

In addition to the attributes available for MATLAB objects, you can apply the following attributes to any property of a custom System object.

`Nontunable`	After an object is locked (after `step` or `setup` has been called), use `Nontunable` to prevent a user from changing that property value. By default, all properties are tunable.

	The `Nontunable` attribute is useful to lock a property that has side effects when changed. This attribute is also useful for locking a property value assumed to be constant during processing. You should always specify properties that affect the number of input or output ports as `Nontunable`.
`Logical`	Use `Logical` to limit the property value to a logical, scalar value. Any scalar value that can be converted to a logical is also valid, such as 0 or 1.
`PositiveInteger`	Use `PositiveInteger` to limit the property value to a positive integer value.
`DiscreteState`	Use `DiscreteState` to mark a property so it will display its state value when you use the `getDiscreteState` method.

To learn more about attributes, see "Property Attributes" in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Basic System Object

Create a simple System object, `AddOne`, which subclasses from `matlab.System`. You place this code into a MATLAB file, `AddOne.m`.

```
classdef AddOne < matlab.System
% ADDONE Compute an output value that increments the input by one

    methods (Access = protected)
       % stepImpl method is called by the step method.
       function y = stepImpl(~,x)
          y = x + 1;
       end
    end
end
```

Use this object by creating an instance of `AddOne`, providing an input, and using the `step` method.

```
hAdder = AddOne;
x = 1;
y = step(hAdder,x)
```

Assign the `Nontunable` attribute to the `InitialValue` property, which you define in your class definition file.

```
properties (Nontunable)
   InitialValue
end
```

See Also

matlab.system.StringSet

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Method Attributes"
- "Define Basic System Objects"
- "Define Property Attributes"

resetImpl

Class: matlab.System
Package: matlab

Reset System object states

Syntax

```
resetImpl(obj)
```

Description

`resetImpl(obj)` defines the state reset equations for a System object. Typically you reset the states to a set of initial values, which is useful for initialization at the start of simulation.

`resetImpl` is called by the `reset` method only if the object is locked. The object remains locked after it is reset. `resetImpl` is also called by the `setup` method, after the `setupImpl` method.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object

Examples

Reset Property Value

Use the `reset` method to reset the state of the counter stored in the `pCount` property to zero.

```
methods (Access = protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

setupImpl

Class: matlab.System
Package: matlab

Initialize System object

Syntax

```
setupImpl(obj)
setupImpl(obj,input1,input2,...)
```

Description

`setupImpl(obj)` sets up a System object and implements one-time tasks that do not depend on any inputs to its `stepImpl` method. You typically use `setupImpl` to set private properties so they do not need to be calculated each time `stepImpl` method is called. To acquire resources for a System object, you must use `setupImpl` instead of a constructor.

`setupImpl` executes the first time the `step` method is called on an object after that object has been created. It also executes the next time `step` is called after an object has been released.

`setupImpl(obj,input1,input2,...)` sets up a System object using one or more of the `stepImpl` input specifications. The number and order of inputs must match the number and order of inputs defined in the `stepImpl` method. You pass the inputs into `setupImpl` to use the specifications, such as size and data types in the one-time calculations.

`setupImpl` is called by the `setup` method, which is done automatically as the first subtask of the `step` method on an unlocked System object.

Note: You can omit this method from your class definition file if your System object does not require any setup tasks.

You must set `Access = protected` for this method.

Do not use `setupImpl` to initialize or reset states. For states, use the `resetImpl` method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Tips

To validate properties or inputs use the `validatePropertiesImpl`, `validateInputsImpl`, or `setProperties` methods. Do not include validation in `setupImpl`.

Do not use the `setupImpl` method to set up input values.

Input Arguments

`obj`

System object handle

`input1,input2,...`

Inputs to the `stepImpl` method

Examples

Setup a File for Writing

This example shows how to open a file for writing using the `setupImpl` method in your class definition file.

```
methods (Access = protected)
   function setupImpl(obj)
      obj.pFileID = fopen(obj.Filename,'wb');
      if obj.pFileID < 0
         error('Opening the file failed');
       end
    end
```

```
end
```

Initialize Properties Based on Step Inputs

This example shows how to use `setupImpl` to specify that `step` initialize the properties of an input. In this case, calls to the object's `step` method, which include input `u`, initialize the object states in a matrix of size u.

```
methods (Access = protected)
   function setupImpl(obj, u)
    obj.State = zeros(size(u),'like', u);
  end
end
```

stepImpl

Class: matlab.System
Package: matlab

System output and state update equations

Syntax

```
[output1,output2,...] = stepImpl(obj,input1,input2,...)
```

Description

`[output1,output2,...] = stepImpl(obj,input1,input2,...)` defines the algorithm to execute when you call the `step` method on the specified object `obj`. The `step` method calculates the outputs and updates the object's state values using the inputs, properties, and state update equations.

`stepImpl` is called by the `step` method.

Note: You must set `Access = protected` for this method.

Tips

The number of input arguments and output arguments must match the values returned by the `getNumInputsImpl` and `getNumOutputsImpl` methods, respectively

Input Arguments

`obj`

System object handle

`input1,input2,...`

Inputs to the `step` method

Output Arguments

`output`

Output returned from the `step` method.

Examples

Specify System Object Algorithm

Use the `stepImpl` method to increment two numbers.

```
methods (Access = protected)
   function [y1,y2] = stepImpl(obj,x1,x2)
      y1 = x1 + 1;
      y2 = x2 + 1;
    end
end
```

How To

- "Define Basic System Objects"

matlab.system.mixin.Nondirect class

Package: matlab.system.mixin

Nondirect feedthrough mixin class

Description

`matlab.system.mixin.Nondirect` is a class that uses the `output` and `update` methods to process nondirect feedthrough data through a System object.

For System objects that use direct feedthrough, the object's input is needed to generate the output at that time. For these direct feedthrough objects, the `step` method calculates the output and updates the state values. For nondirect feedthrough, however, the object's output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with `outputImpl` is separated from updating the state values with `updateImpl`. If you use the `matlab.system.mixin.Nondirect` mixin and include the `stepImpl` method in your class definition file, an error occurs. In this case, you must include the `updateImpl` and `outputImpl` methods instead.

The following cases describe when System objects in Simulink use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the `Propagates` mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the `Propagates` mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method.
- System object does not support code generation — Default `isInputDirectFeedthrough` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file.

Use the `Nondirect` mixin to allow a System object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.

To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.system &...
    matlab.system.mixin.Nondirect
```

Methods

outputImpl	Output calculation from input or internal state of System object
updateImpl	Update object states based on inputs

See Also

matlab.system

Tutorials

- “Use Update and Output for Nondirect Feedthrough”

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes

outputImpl

Class: matlab.system.mixin.Nondirect
Package: matlab.system.mixin

Output calculation from input or internal state of System object

Syntax

```
[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)
```

Description

`[y1,y2,...,yN] = outputImpl(obj,u1,u2,...,uN)` implements the output equations for the System object. The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

`outputImpl` is called by the `output` method. It is also called before the `updateImpl` method in the `step` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object handle

`u1,u2,...uN`

Inputs from the algorithm or `step` method. The number of inputs must match the number of inputs returned by the `getNumInputs` method. Nondirect feedthrough inputs

are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

Output Arguments

`y1,y2,...yN`

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the `getNumOutputs` method.

Examples

Set Up Output that Does Not Depend on Input

Specify in your class definition file that the output does not directly depend on the current input with the `outputImpl` method. `PreviousInput` is a property of the `obj`.

```
methods (Access = protected)
   function [y] = outputImpl(obj, ~)
      y = obj.PreviousInput(end);
   end
end
```

See Also

matlab.system.mixin.Nondirect

How To

- "Use Update and Output for Nondirect Feedthrough"

updateImpl

Class: matlab.system.mixin.Nondirect
Package: matlab.system.mixin

Update object states based on inputs

Syntax

```
updateImpl(obj,u1,u2,...,uN)
```

Description

`updateImpl(obj,u1,u2,...,uN)` implements the state update equations for the system. You use this method when your algorithm outputs depend only on the object's internal state and internal properties. Do not use this method to update the outputs from the inputs.

`updateImpl` is called by the `update` method and after the `outputImpl` method in the `step` method. For sink objects, calling `updateImpl` before `outputImpl` locks the object. For all other types of objects, calling `updateImpl` before `outputImpl` causes an error.

Note: You must set `Access = protected` for this method.

You cannot modify any tunable properties in this method if its System object will be used in the Simulink `MATLAB System` block.

Input Arguments

`obj`

System object handle

`u1,u2,...uN`

Inputs to the algorithm or `step` method. The number of inputs must match the number of inputs returned by the `getNumInputs` method.

Examples

Set Up Output that Does Not Depend on Current Input

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```
methods (Access = protected)
   function updateImpl(obj,u)
      obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
   end
end
```

See Also

matlab.system.mixin.Nondirect

How To

- "Use Update and Output for Nondirect Feedthrough"

matlab.system.StringSet class

Package: matlab.system

Set of valid string values

Description

`matlab.system.StringSet` defines a list of valid string values for a property. This class validates the string in the property and enables tab completion for the property value. A *StringSet* allows only predefined or customized strings as values for the property.

A `StringSet` uses two linked properties, which you must define in the same class. One is a public property that contains the current string value. This public property is displayed to the user. The other property is a hidden property that contains the list of all possible string values. This hidden property should also have the transient attribute so its value is not saved to disk when you save the System object.

The following considerations apply when using StringSets:

- The string property that holds the current string can have any name.
- The property that holds the `StringSet` must use the same name as the string property with the suffix "Set" appended to it. The string set property is an instance of the `matlab.system.StringSet` class.
- Valid strings, defined in the `StringSet`, must be declared using a cell array. The cell array cannot be empty nor can it have any empty strings. Valid strings must be unique and are case-sensitive.
- The string property must be set to a valid `StringSet` value.

Examples

Set String Property Values

Set the string property, `Flavor`, and the `StringSet` property, `FlavorSet` in your class definition file.

```
properties
  Flavor = 'Chocolate';
end

properties (Hidden,Transient)
  FlavorSet = ...
    matlab.system.StringSet({'Vanilla','Chocolate'});
end
```

See Also

matlab.System

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes