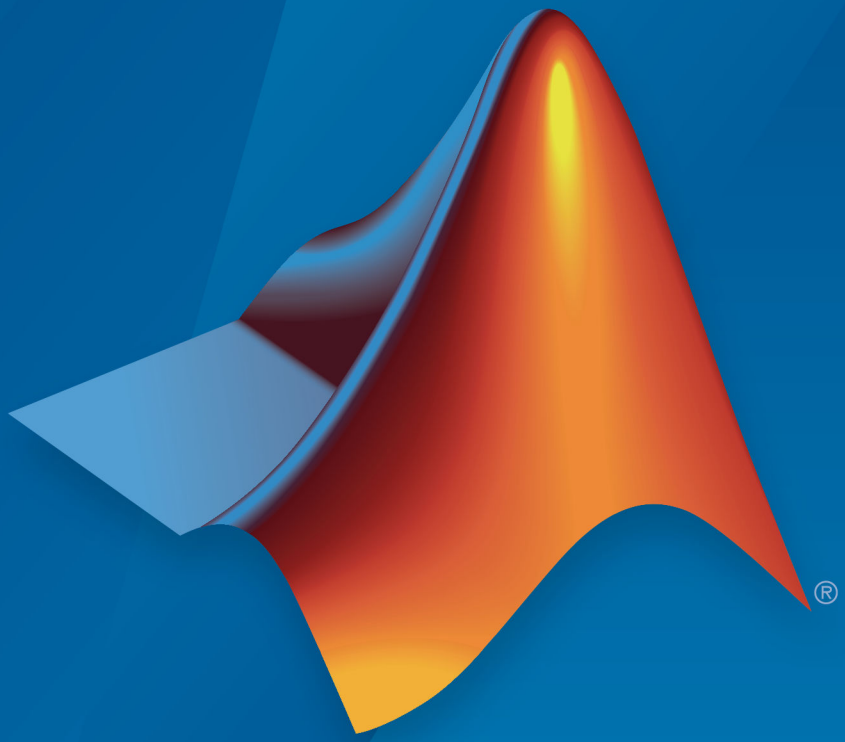
HDL Coder™

Reference



MATLAB® & SIMULINK®

R2017b

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)
October 2015	Online only	Rereleased for Version 3.6.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.8 (Release 2016a)
September 2016	Online only	Revised for Version 3.9 (Release 2016b)
March 2017	Online only	Revised for Version 3.10 (Release 2017a)
September 2017	Online only	Revised for Version 3.11 (Release 2017b)

Contents

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

8

Apps — Alphabetical List

HDL Coder

Generate HDL code from MATLAB code

Description

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

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

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

Open the HDL Coder App

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

- MATLAB command prompt: Enter `hdlcoder`.

Examples

- "HDL Code Generation from a MATLAB Algorithm"

Programmatic Use

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

See Also

Apps
Fixed-Point Converter

Functions
`codegen`

Topics
"HDL Code Generation from a MATLAB Algorithm"
"Guidelines for Efficient HDL Code"
"Create and Set Up Your Project"

Introduced in R2012a

Functions — Alphabetical List

checkhdl

Check subsystem or model for HDL code generation compatibility

Syntax

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

Description

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

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

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

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

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

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

`checkhdl(gcb)` examines the currently selected subsystem.

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

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

`checkhdl` reports three levels of compatibility problems:

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

Examples

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

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

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

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

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

```
output =

1x4 struct array with fields:
    path
    type
    message
    level


output(1)
```

```
ans =

       path: 'sfir_fixed_err/symmetric_fir_err/Product'
       type: 'block'
    message: 'Unhandled mixed double and non-double datatypes at ports of block'
      level: 'Error'
```

See Also

`makehdl`

Topics

"Selecting and Checking a Subsystem for HDL Compatibility"

Introduced in R2006b

hdladvisor

Display HDL Workflow Advisor

Syntax

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

Description

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

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

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

Examples

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

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

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

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

Alternatives

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

See Also

"HDL Workflow Advisor Tasks" | "Getting Started with the HDL Workflow Advisor"

Introduced in R2010a

hdlcoder.optimizeDesign

Automatic iterative HDL design optimization

Syntax

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

Description

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

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

Examples

Maximize clock frequency

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

Open the model and specify the DUT subsystem.

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

Set your synthesis tool and target device options.

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

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

Enable HDL test bench generation.

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

Save your model.

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

Create an optimization configuration object, `oc`.

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

Set the iteration limit to 10.

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

Optimize the model.

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

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

Iteration 0
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 143.66 Iteration 1
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 278.72 Iteration 2
Generate and synthesize HDL code ...
(CP ns) 10.25    (Constraint ns) 12.73    (Elapsed s) 427.22 Iteration 3
Generate and synthesize HDL code ...
(CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37 Iteration 4
Generate and synthesize HDL code ...
(CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04 Iteration 5
```

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

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

Optimize for specific clock frequency

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

Open the model and specify the DUT subsystem.

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

Set your synthesis tool and target device options.

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

Disable HDL test bench generation.

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

Save your model.

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

Create an optimization configuration object, `oc`.

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

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

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

Optimize the model.

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

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

Iteration 0
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 20.00    (Elapsed s) 134.02 Iteration 1
Generate and synthesize HDL code ...
Exiting because constraint (20.00 ns) has been met (16.26 ns).
Summary report: summary.html
Achieved Critical Path (CP) Latency : 16.26 ns        Elapsed : 134.02 s
Iteration 0: (CP ns) 16.26    (Constraint ns) 20.00    (Elapsed s) 134.02
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-14
Validation model: gm_sfir_fixed_vnl
```

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

Resume clock frequency optimization using saved data

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

Open the model and specify the DUT subsystem.

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

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

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

Enable HDL test bench generation.

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

Create an optimization configuration object, `oc`.

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

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

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

Optimize the model.

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

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

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

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

Regenerate code using original design and saved optimization data

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

Open the model and specify the DUT subsystem.

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

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

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

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

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

```
Final results are saved in
  /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-16-52
Validation model: gm_sfir_fixed_vnl
```

Input Arguments

`model` — Model name

character vector

Model name, specified as a character vector.

Example: `'sfir_fixed'`

`optimizationCfg` — Optimization configuration

`hdlcoder.OptimizationConfig`

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

`cpGuidanceFile` — File containing saved optimization data

`''` (default) | character vector

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

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

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

See Also

Classes

`hdlcoder.OptimizationConfig`

Functions

`hdlcoder.supportedDevices`

Properties

SynthesisTool | SynthesisToolChipFamily | SynthesisToolDeviceName | SynthesisToolPackageName | SynthesisToolSpeedValue

Topics

"Automatic Iterative Optimization"

Introduced in R2014a

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
```

See Also

SynthesisToolChipFamily | SynthesisToolDeviceName | SynthesisToolPackageName | SynthesisToolSpeedValue

Topics

“Synthesis Tool Path Setup”

Introduced in R2014a

hdldispblkparams

Display HDL block parameters with nondefault values

Syntax

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

Description

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

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

Input Arguments

`path`

Path to a block or subsystem in the current model.

Default: None

`'all'`

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

Examples

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

```
hdldispblkparams('simplevectorsum/vsum/Sum of Elements')

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

Implementation

    Architecture : Linear

Implementation Parameters

    InputPipeline : 1
```

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

```
hdldispblkparams(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”

Introduced in R2010b

hdldispmdlparams

Display HDL model parameters with nondefault values

Syntax

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

Description

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

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

Input Arguments

`model`

Name of an open model.

Default: None

`'all'`

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

Examples

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

```
 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”

Introduced in R2010b

hdlget_param

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

Syntax

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

Description

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

Input Arguments

`block_path`

Path to a block or subsystem in the current model.

Default: None

`prop`

A character vector that designates one of the following:

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

Default: None

Output Arguments

p

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

Examples

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

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

p =

     3
```

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

```
p = hdlget_param(gcb,'all')

p =

    'Architecture'    'Linear'    'InputPipeline'    [0]    'OutputPipeline'    [0]
```

Tips

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

See Also

hdlrestoreparams | hdlsaveparams | hdlset_param

Introduced in R2010b

hdllib

Display blocks that are compatible with HDL code generation

Syntax

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

Description

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

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

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

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

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

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

The default library name is `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 that you install a new software release.

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

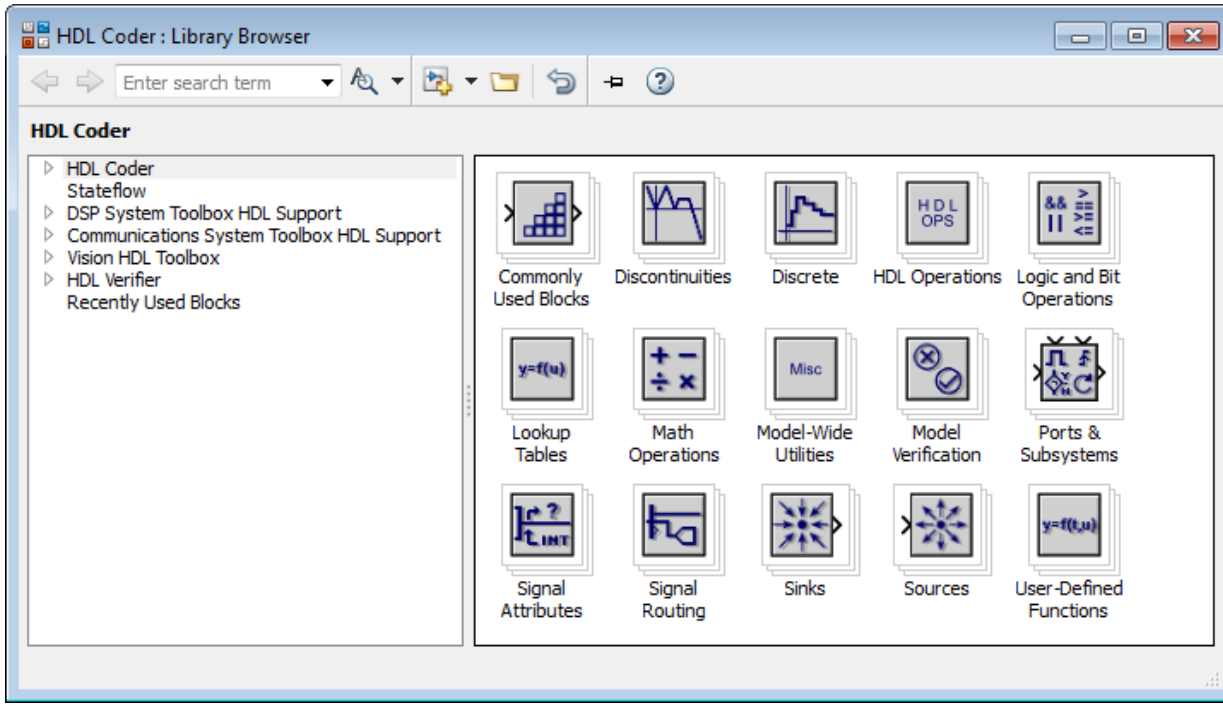
Examples

Display Supported Blocks in the Library Browser

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

```
hdllib
```

```
### Generating view of HDL Coder compatible blocks in Library Browser.
### To restore the Library Browser to the default Simulink view, enter "hdllib off".
```

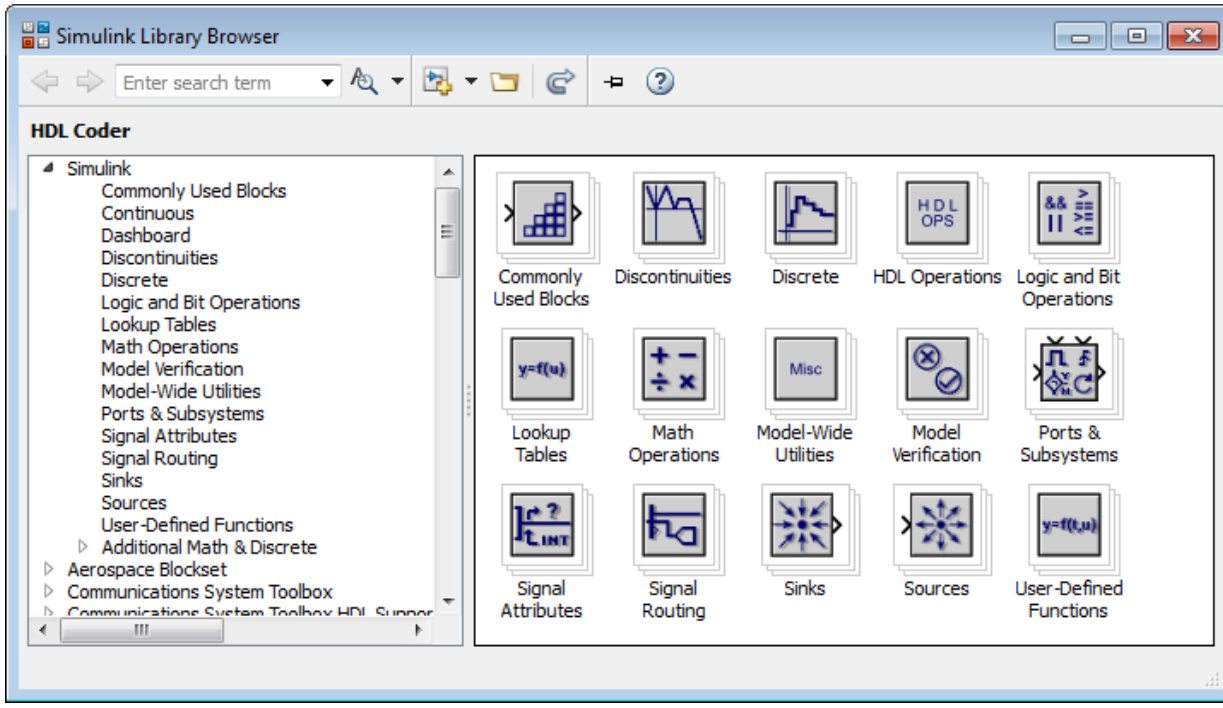


Display All Blocks in the Library Browser

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

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

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

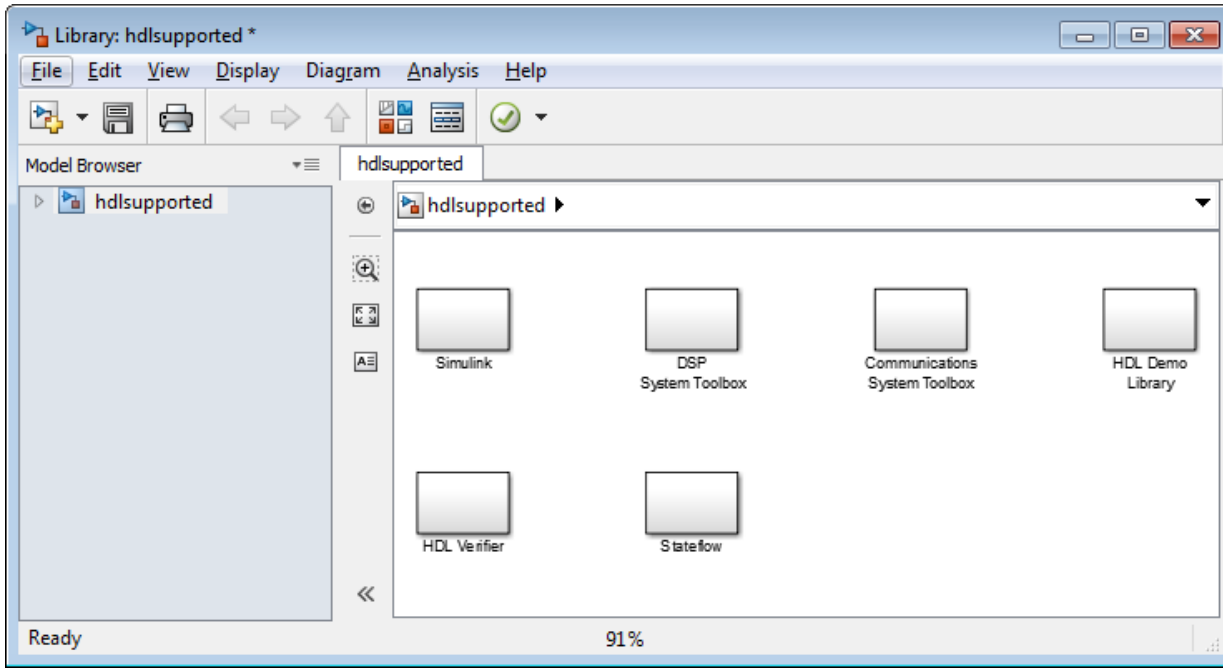


Create a Supported Blocks Library and HTML Reports

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

```
hdllib('html')

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

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

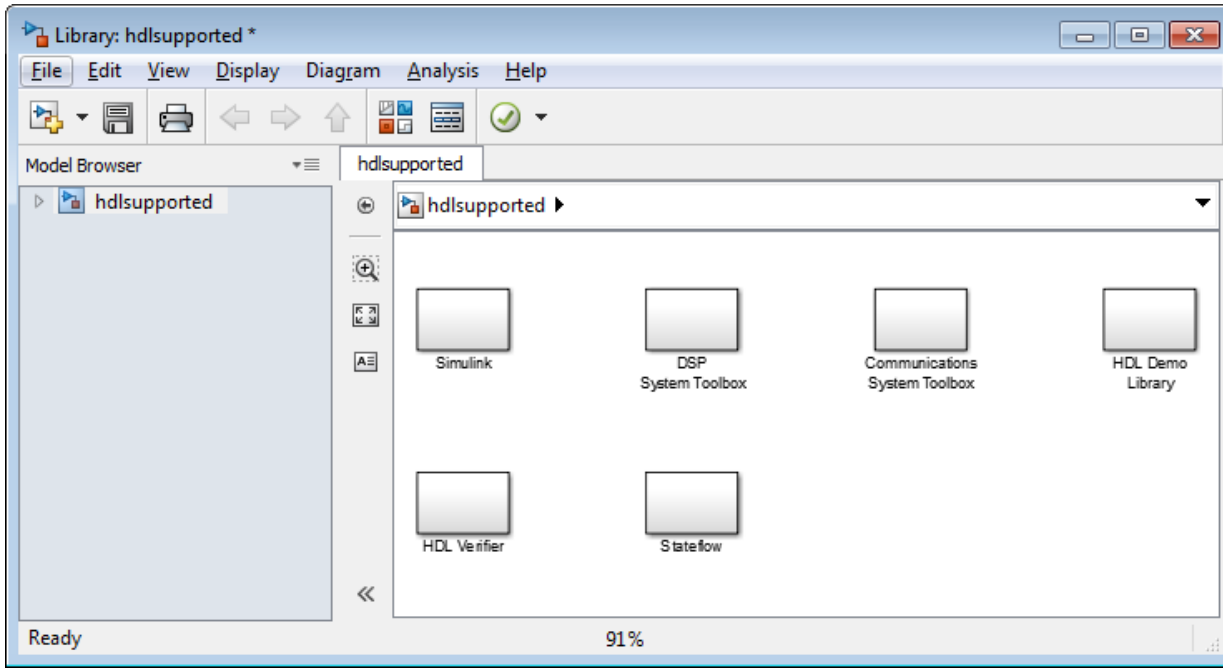


Create a Supported Blocks Library

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

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

The `hdlsupported` block library opens.



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

See Also

“Supported Blocks”

Topics

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

Introduced in R2006b

hdlmodelchecker

Open HDL Model Checker

Syntax

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

Description

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

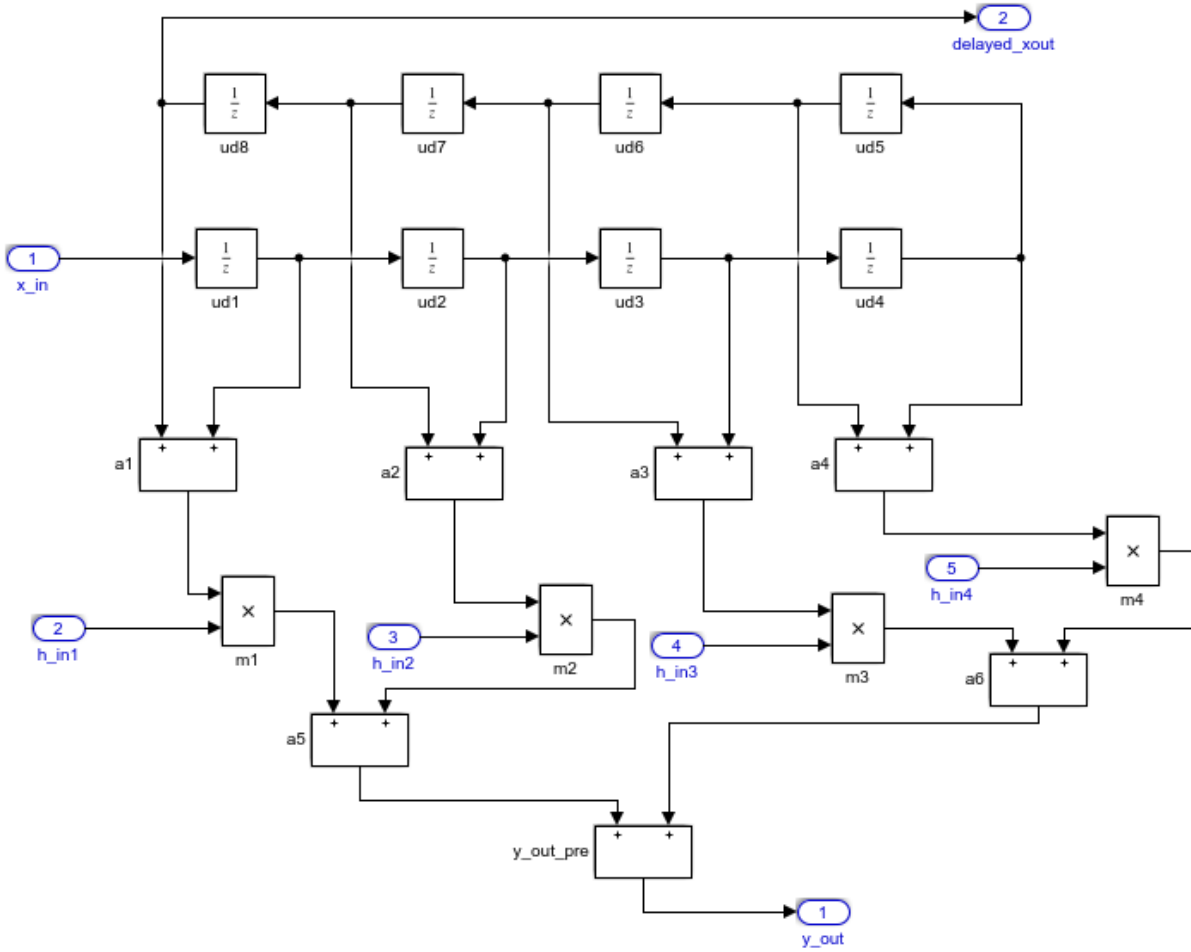
`hdlmodelchecker(model)` opens the Model Checker for the model.

Examples

Open the HDL Model Checker For a Model

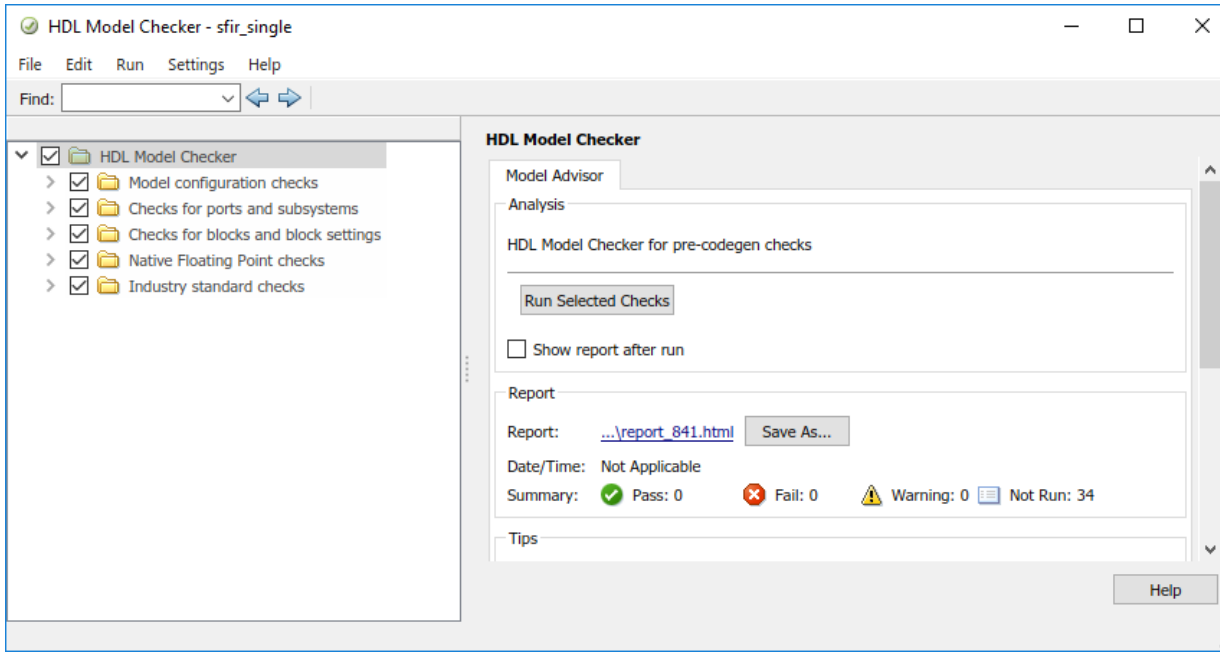
This example shows how to open the HDL Model Checker for the `sfir_single` model.

```
sfir_single
```



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

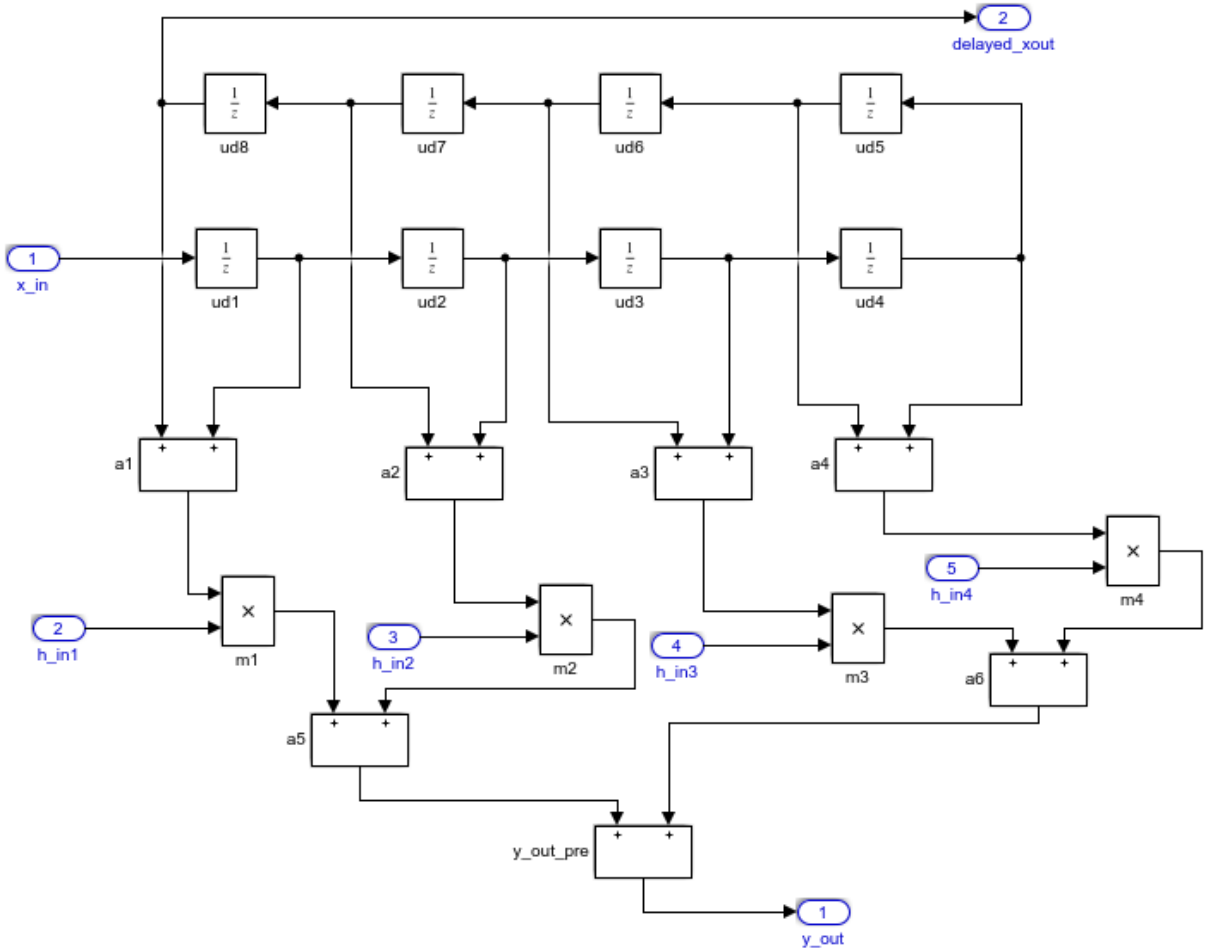
```
hdlmodelchecker('sfir_single')
```



Open the HDL Model Checker For a Subsystem

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

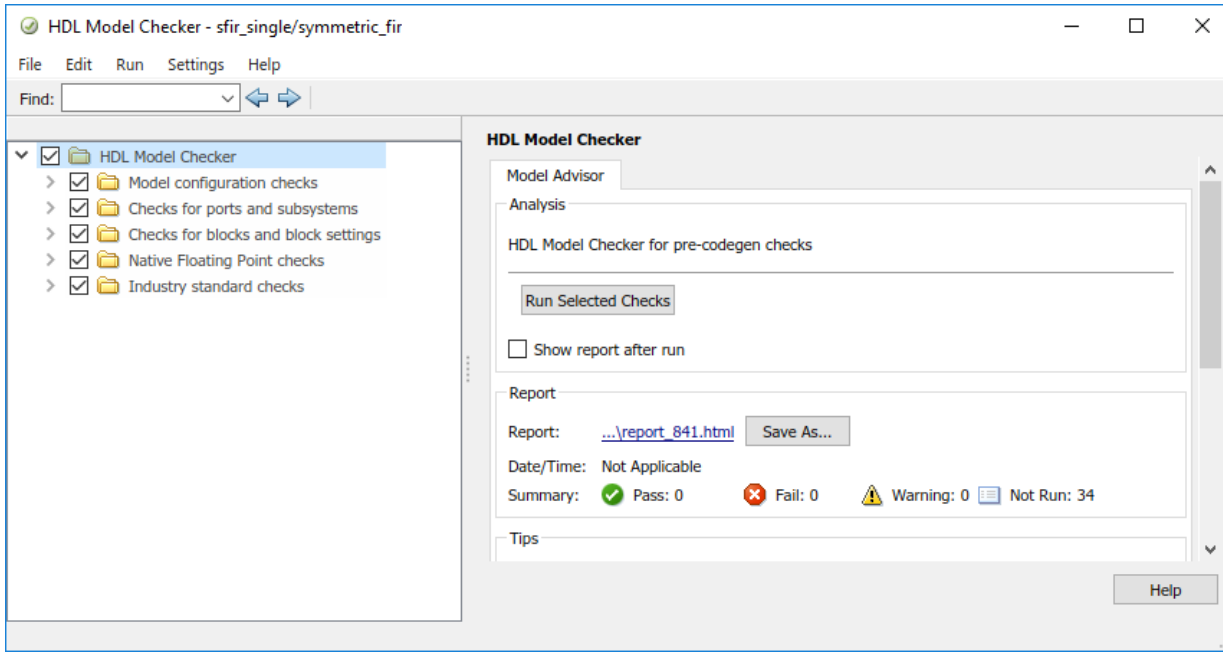
```
sfir_single
```



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

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

```
Updating Model Advisor cache…
Model Advisor cache updated. For new customizations, to update the cache, use the Advis
```



Input Arguments

`subsystem` — Subsystem name

character vector

Subsystem name or handle, specified as a character vector.

Data Types: `char`

`model` — Model name

character vector

Model name or handle, specified as a character vector.

Data Types: `char`

See Also

Topics

"Getting Started with the HDL Model Checker"
"Checks In the HDL Model Checker"

Introduced in R2017b

hdlrestoreparams

Restore block- and model-level HDL parameters to model

Syntax

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

Description

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

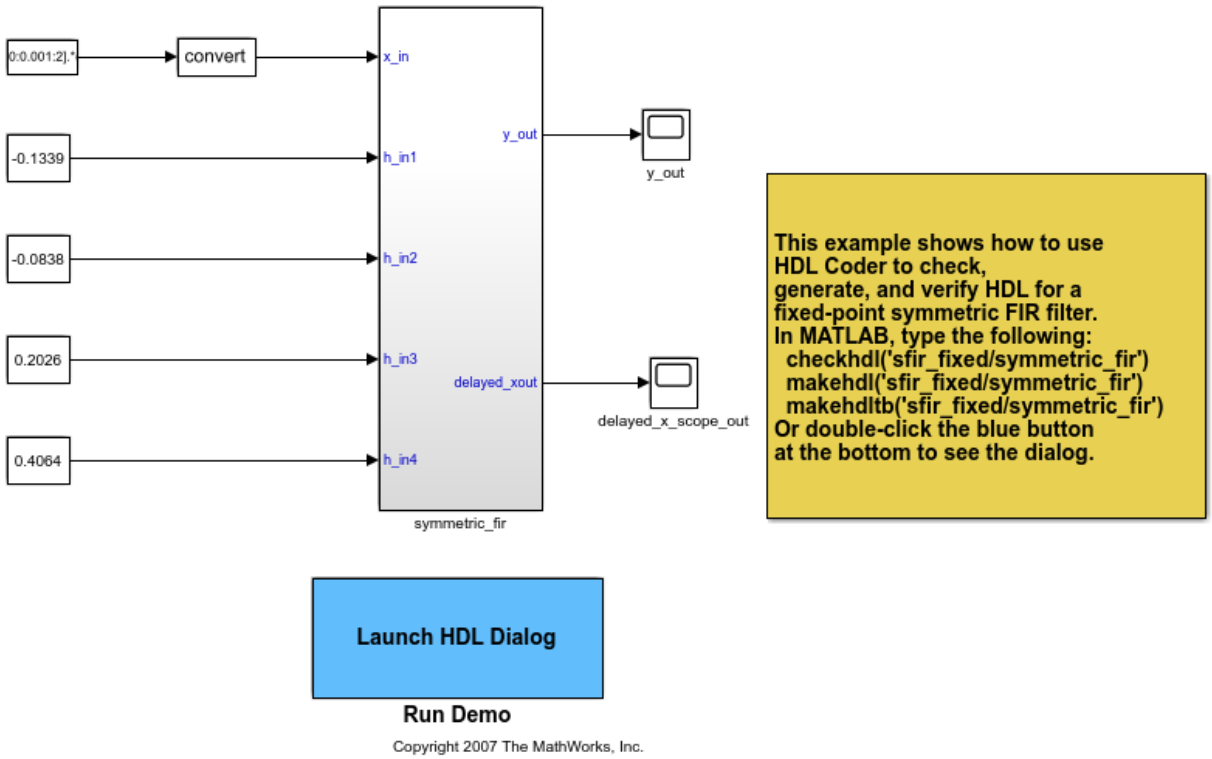
`hdlrestoreparams(dut,filename)` restores to the specified model the block- and model-level HDL settings from a previously saved file.

Examples

Save and Restore HDL-Related Model Parameters

Open the model.

```
sfir_fixed
```



Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Verify that model parameters are set.

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

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

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

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

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

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

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

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

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

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

Verify that the saved model parameters are restored.

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

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

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

Input Arguments

`dut` — DUT subsystem name

character vector

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file

character vector

Name of file containing previously saved HDL model parameters.

Example: `'mymodel_saved_params.m'`

See Also

`hdlsaveparams`

Introduced in R2012b

hdlsaveparams

Save nondefault block- and model-level HDL parameters

Syntax

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

Description

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

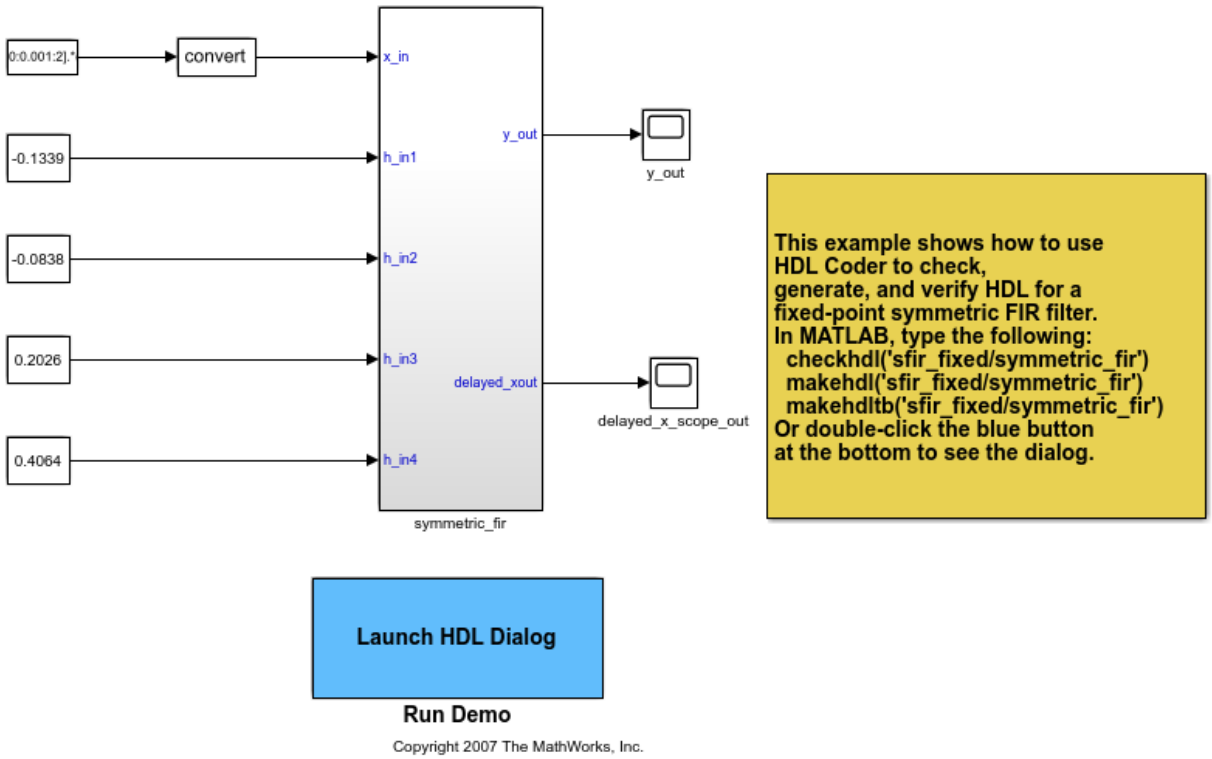
`hdlsaveparams(dut,filename)` saves nondefault block- and model-level HDL parameters to a MATLAB script.

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', 'InputPipeline', 5)
```

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

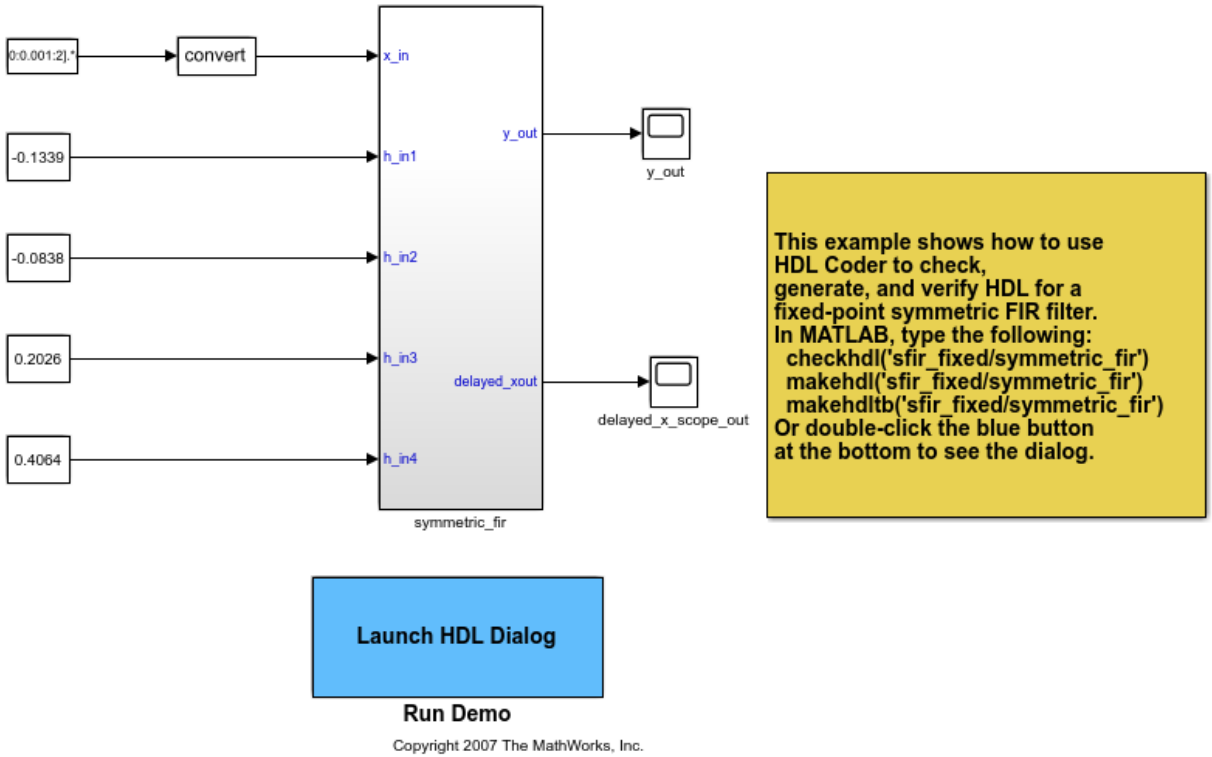
```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Save and Restore HDL-Related Model Parameters

Open the model.

```
sfir_fixed
```



Verify that model parameters have default values.

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

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

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

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

Verify that model parameters are set.

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

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

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

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

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

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

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

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

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

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

Verify that the saved model parameters are restored.

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

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

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

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

Input Arguments

`dut` — DUT subsystem name

character vector

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file

character vector

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

Example: `'mymodel_saved_params.m'`

See Also

`hdlrestoreparams`

Introduced in R2012b

hdlset_param

Set HDL-related parameters at model or block level

Syntax

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

Description

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

Input Arguments

`path`

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

Default: None

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Name`

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

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

Default: None

`Value`

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

Default: Default value is dependent on the property.

Examples

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

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

Tips

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

See Also

`hdlget_param` | `hdlrestoreparams` | `hdlsaveparams`

Topics

"Set and View HDL Block Parameters"
"Set HDL Block Parameters for Multiple Blocks"

Introduced in R2010b

hdlsetup

Set up model parameters for HDL code generation

Syntax

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

Description

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

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

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

How hdlsetup Configures Solver Options

`hdlsetup` configures **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`.

Introduced in R2006b

hdlsetuptoolpath

Set up system environment to access FPGA synthesis software

Syntax

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

Description

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

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

Examples

Set Up 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
character vector

Synthesis tool name, specified as a character vector.

Example: `'Xilinx Vivado'`

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

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

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

See Also

`setenv` | `startup`

Topics

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

Introduced in R2011a

makehdl

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

Syntax

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

Description

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

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

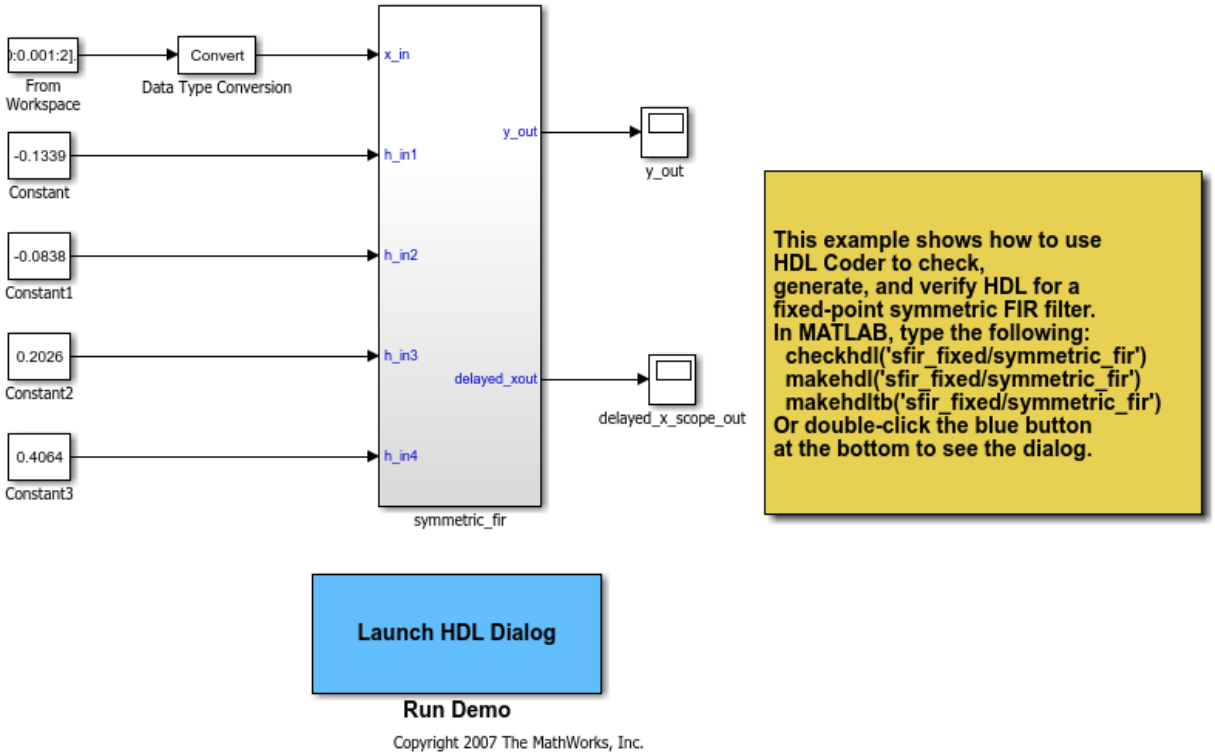
Examples

Generate VHDL for the Current Model

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

Open the `sfir_fixed` model.

```
sfir_fixed
```



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

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

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

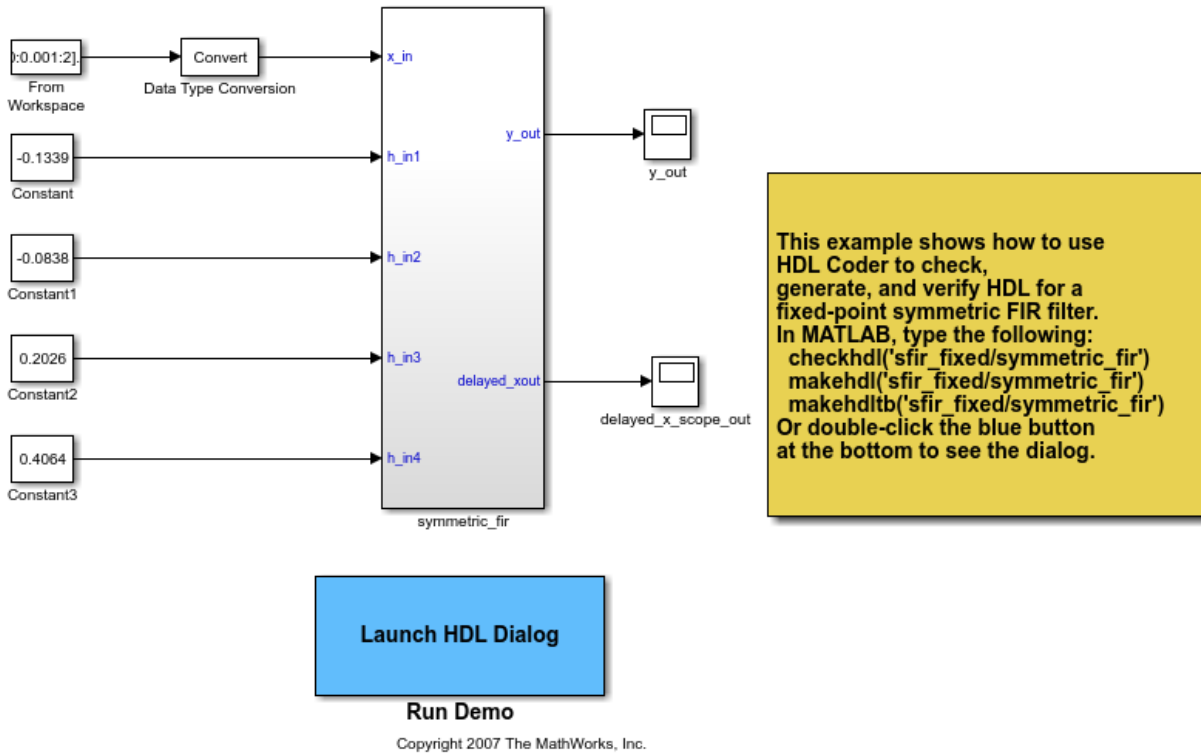
The generated VHDL code is saved in the `hdlsrc` folder.

Generate Verilog for a Subsystem Within a Model

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

Open the `sfir_fixed` model.

```
sfir_fixed;
```



The model opens in a new Simulink® window.

Generate Verilog for the `symmetric_fir` subsystem.

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

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

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

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

Close the model.

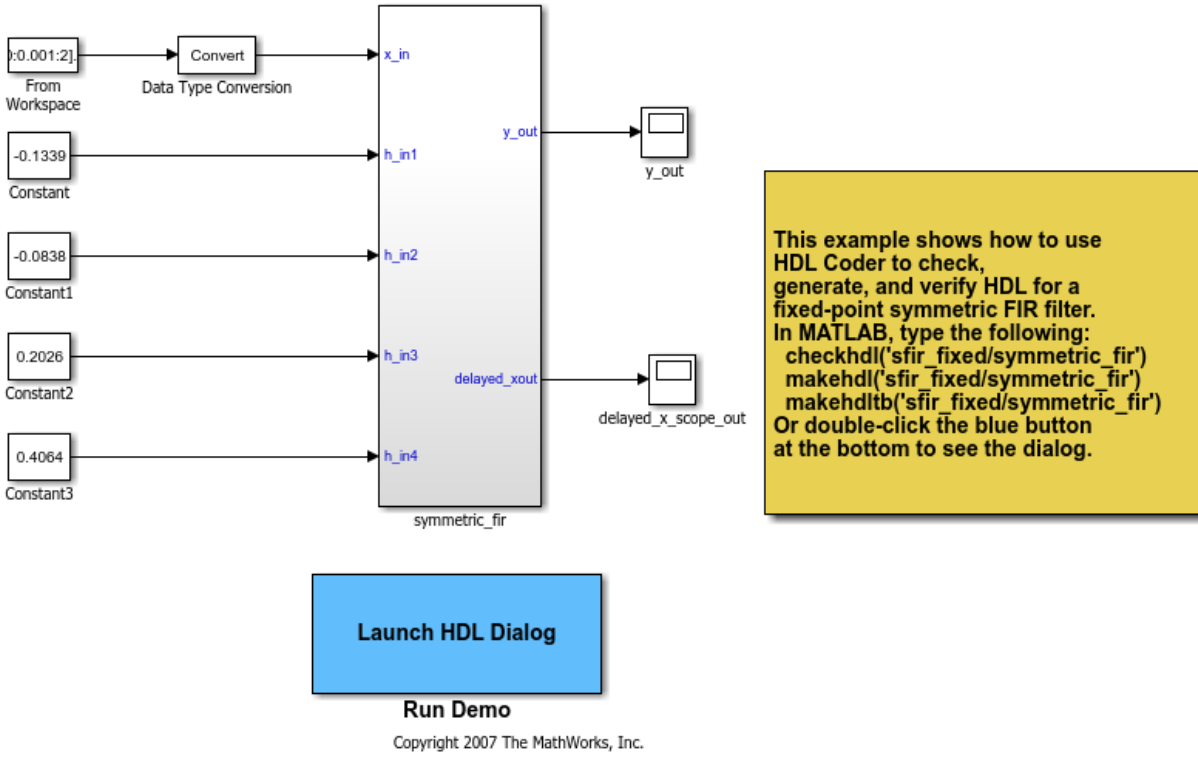
```
bdclose('sfir_fixed');
```

Check Subsystem for Compatibility with HDL Code Generation

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

Open the `sfir_fixed` model.

```
sfir_fixed
```



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', ...
            'TargetDirectory','C:/HDL_Checks/hdlsrc')

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

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

Close the model.

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

Input Arguments

`dut` — DUT model or subsystem name

character vector

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

Example: `'top_level_name'`

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

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside 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) | character vector

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`.

Report Generation

`HDLCodingStandard` — Specify HDL coding standard
character vector

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`.

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

For more information, see `HDLGenerateWebview`.

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

Note `MaxOversampling` is not recommended. Use clock-rate pipelining with Oversampling instead.

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

Note `MaxComputationLatency` is not recommended. Use clock-rate pipelining with Oversampling instead.

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`.

`MapPipelineDelaysToRAM` — Map pipeline registers in the generated HDL code to RAM
`'off'` (default) | `'on'`

For more information, see MapPipelineDelaysToRAM.

`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) | character vector

For more information, see HighlightFeedbackLoopsFile.

Coding Style

`UserComment` — HDL file header comment
character vector

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`.

Clocks and Reset

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

For more information, see `ClockEdge`.

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

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

`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
character vector

For more information, see `SimulatorFlags`.

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

For more information, see `TestBenchReferencePostFix`.

Script Generation

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

For more information, see `EDAScriptGeneration`.

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

For more information, see `HDLCompileInit`.

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

For more information, see `HDLCompileTerm`.

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

For more information, see `HDLCompileFilePostfix`.

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

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

For more information, see `HDLCompileVerilogCmd`.

`HDLCompileVHDLCmd` — VHDL compilation command
`'vcom %s %s\n'` (default) | character vector

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

For more information, see `HDLCompileVHDLCmd`.

`HDLLintTool` — HDL lint tool
`'None'` (default) | `'AscentLint'` | `'Leda'` | `'SpyGlass'` | `'Custom'`

For more information, see `HDLLintTool`.

`HDLLintInit` — HDL lint initialization name
character vector

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

For more information, see `HDLLintInit`.

`HDLLintCmd` — HDL lint command
character vector

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

For more information, see `HDLLintCmd`.

`HDLLintTerm` — HDL lint termination name
character vector

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

For more information, see `HDLLintTerm`.

`HDLSynthTool` — Synthesis tool
`'None'` (default) | `'ISE'` | `'Libero'` | `'Precision'` | `'Quartus'` | `'Synplify'` | `'Vivado'` | `'Custom'`

For more information, see `HDLSynthTool`.

`HDLSynthCmd` — HDL synthesis command
character vector

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

For more information, see `HDLSynthCmd`.

`HDLSynthFilePostfix` — Postfix for synthesis script file name
character vector

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

For more information, see `HDLSynthFilePostfix`.

`HDLSynthInit` — Synthesis script initialization name
character vector

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

For more information, see `HDLSynthInit`.

`HDLSynthTerm` — Synthesis script termination name
character vector

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

For more information, see `HDLSynthTerm`.

Generated Model

`GenerateModel` — Control generation of generated model
`'on'` (default) | `'off'`

For more information, see GenerateModel.

`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) | character vector

For more information, see `GeneratedModelName`.

`GeneratedModelNamePrefix` — Prefix for generated model name
`'gm_'` (default) | character vector

For more information, see `GeneratedModelNamePrefix`.

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) | character vector

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

For more information, see `ClockEnableInputPort`.

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

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

For more information, see `ClockEnableOutputPort`.

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

Clock input port name, specified as a character vector.

For more information, see `ClockInputPort`.

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

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

For more information, see `InputType`.

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

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

For more information, see `OutputType`.

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

Reset input port name, specified as a character vector.

For more information, see `ResetInputPort`.

File and Variable Names

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

For more information, see `VerilogFileExtension`.

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

For more information, see `VHDLFileExtension`.

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

For more information, see `VHDLArchitectureName`.

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

For more information, see `VHDLLibraryName`.

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

For more information, see `SplitEntityFilePostfix`.

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

For more information, see `SplitArchFilePostfix`.

`PackagePostfix` — Postfix for package file name
`'_pkg'` (default) | character vector

For more information, see `PackagePostfix`.

`HDLMapFilePostfix` — Postfix for mapping file
`'_map.txt'` (default) | character vector

For more information, see `HDLMapFilePostfix`.

`BlockGenerateLabel` — Block label postfix for VHDL `GENERATE` statements
`'_gen'` (default) | character vector

For more information, see `BlockGenerateLabel`.

`ClockProcessPostfix` — Postfix for clock process names
`'_process'` (default) | character vector

For more information, see `ClockProcessPostfix`.

`ComplexImagPostfix` — Postfix for imaginary part of complex signal
`'_im'` (default) | character vector

For more information, see `ComplexImagPostfix`.

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

For more information, see `ComplexRealPostfix`.

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

For more information, see `EntityConflictPostfix`.

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

For more information, see `InstanceGenerateLabel`.

`InstancePostfix` — Postfix for generated component instance names
`''` (default) | character vector

For more information, see `InstancePostfix`.

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

For more information, see `InstancePrefix`.

`OutputGenerateLabel` — Output assignment label postfix for VHDL `GENERATE` statements

`'outputgen'` (default) | character vector

For more information, see `OutputGenerateLabel`.

`PipelinePostfix` — Postfix for input and output pipeline register names

`'_pipe'` (default) | character vector

For more information, see `PipelinePostfix`.

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

`'_rsvd'` (default) | character vector

For more information, see `ReservedWordPostfix`.

`TimingControllerPostfix` — Postfix for timing controller name

`'_tc'` (default) | character vector

For more information, see `TimingControllerPostfix`.

`VectorPrefix` — Prefix for vector names

`'vector_of_'` (default) | character vector

For more information, see `VectorPrefix`.

`EnablePrefix` — Prefix for internal enable signals

`'enb'` (default) | character vector

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

For more information, see `EnablePrefix`.

`ModulePrefix` — Prefix for modules or entity names

`''` (default) | character vector

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

For more information, see `ModulePrefix`.

See Also

`checkhdl` | `makehdltb`

Introduced in R2006b

makehdltb

Generate HDL test bench from model or subsystem

Syntax

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

Description

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

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

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

Examples

Generate VHDL Test Bench

Generate VHDL DUT and test bench for a subsystem.

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

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

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

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

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

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

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

Generate Verilog Test Bench

Generate Verilog DUT and test bench for a subsystem.

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

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

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

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

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

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

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

Generate a SystemVerilog DPI Test Bench

Generate SystemVerilog DPI test bench for a subsystem.

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

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

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

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

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

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

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

### HDL TestBench generation complete.
```

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

Input Arguments

`dut` — DUT subsystem or model reference name

character vector

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

Example: `'modelname/subsysTarget'`

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

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside 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) | character vector

For more information, see `TargetDirectory`.

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

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

For more information, see `SplitEntityArch`.

Test Bench Generation

`GenerateHDLTestBench` — Generate HDL test bench

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

The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT. You can disable this property when you use an alternate test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

`GenerateSVDPITestBench` — Generate SystemVerilog DPI test bench

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

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

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

because it does not store the input or expected data in a separate file. For an example, see “Generate a SystemVerilog DPI Test Bench” on page 2-75.

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

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

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

`GenerateCoSimBlock` — Generate HDL Cosimulation block
`'off'` (default) | `'on'`

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

For more information, see `GenerateCoSimBlock`.

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

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

For more information, see `GenerateCoSimModel`.

`HDLCodeCoverage` — Enable code coverage on the generated test bench
`'off'` (default) | `'on'`

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

`SimulationTool` — HDL simulator where you will run the generated test bench
`'ModelSim'` (default) | `'Incisive'` | `'VCS'` | `'Vivado'` | `'Custom'`

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

Test Bench Configuration

`ForceClock` — Force clock input
`'on'` (default) | `'off'`

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

For more information, see `ForceClock`.

`ClockHighTime` — Clock high time
5 (default) | positive integer

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

For more information, see `ClockHighTime`.

`ClockLowTime` — Clock low time
5 (default) | positive integer

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

For more information, see `ClockLowTime`.

`ForceClockEnable` — Force clock enable input
`'on'` (default) | `'off'`

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

For more information, see `ForceClockEnable`.

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

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) | character vector

For more information, see `TestBenchDataPostFix`.

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

For more information, see `TestBenchPostFix`.

Coding Style

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

For more information, see `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 text
`'vlib %s\n'` (default) | character vector

For more information, see `HDLCompileInit`.

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

For more information, see `HDLCompileTerm`.

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

For more information, see `HDLCompileFilePostfix`.

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

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

For more information, see `HDLCompileVerilogCmd`.

`HDLCompileVHDLCmd` — VHDL compilation command
`'vcom %s %s\n'` (default) | character vector

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

For more information, see `HDLCompileVHDLCmd`.

`HDLSimCmd` — HDL simulation command
`'vsim -novopt %s.%s\n'` (default) | character vector

The HDL simulation command, specified as a character vector.

For more information, see `HDLSimCmd`.

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

Initialization for the HDL simulation script, specified as a character vector.

For more information, see `HDLSimInit`.

`HDLSimTerm` — HDL simulation script termination name
`'run -all'` (default) | character vector

The termination name for the HDL simulation command, specified as a character vector.

For more information, see `HDLSimTerm`.

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

For more information, see `HDLSimFilePostfix`.

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

Waveform viewing command, specified as a character vector. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.

For more information, see `HDLSimViewWaveCmd`.

Port Names and Types

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

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

For more information, see `ClockEnableInputPort`.

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

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

For more information, see `ClockEnableOutputPort`.

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

Clock input port name, specified as a character vector.

For more information, see `ClockInputPort`.

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

Reset input port name, specified as a character vector.

For more information, see `ResetInputPort`.

File and Variable Names

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

For more information, see `VerilogFileExtension`.

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

For more information, see `VHDLFileExtension`.

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

For more information, see `VHDLArchitectureName`.

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

For more information, see `VHDLLibraryName`.

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

For more information, see `SplitEntityFilePostfix`.

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

For more information, see `SplitArchFilePostfix`.

`PackagePostfix` — Postfix for package file name
`'_pkg'` (default) | character vector

For more information, see `PackagePostfix`.

`ComplexImagPostfix` — Postfix for imaginary part of complex signal
`'_im'` (default) | character vector

For more information, see `ComplexImagPostfix`.

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

For more information, see `ComplexRealPostfix`.

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

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

For more information, see `EnablePrefix`.

See Also

`makehdl`

Introduced in R2006b

Supported Blocks

1-D Lookup Table

Approximate one-dimensional function (HDL Coder)

Description

The 1-D Lookup Table block is a one-dimensional version of the n-D Lookup Table block. For HDL code generation information, see n-D Lookup Table.

Introduced in R2014a

2-D Lookup Table

Approximate two-dimensional function (HDL Coder)

Description

The 2-D Lookup Table block is a two-dimensional version of the n-D Lookup Table block. For HDL code generation information, see n-D Lookup Table.

Introduced in R2014a

Abs

Output absolute value of input (HDL Coder)

Description

The Abs block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Add

Add inputs (HDL Coder)

Description

The Add block is available with Simulink.

For information about the simulation behavior and block parameters, see Add.

HDL Architecture

The default `Linear` architecture generates a chain of N operations (adders) for N inputs.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Complex Data Support

The default `Linear` implementation supports complex data.

Introduced in R2014a

Assertion

Check whether signal is zero (HDL Coder)

Description

The Assertion block is available with Simulink.

For information about the simulation behavior and block parameters, see Assertion.

HDL Architecture

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

Introduced in R2014a

Assignment

Assign values to specified elements of signal (HDL Coder)

Description

The Assignment block is available with Simulink.

For information about the simulation behavior and block parameters, see Assignment.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Atomic Subsystem

Represent system within another system (HDL Coder)

Description

The Atomic Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see Atomic Subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

You can set this property In the HDL Workflow Advisor, 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 (;).

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

Values: `''` (default) | character vector

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

IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: `''` (default) | character vector

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: `''` (default) | character vector

Example: `'1.3'`

Restrictions

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

See Also

Topics

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

Introduced in R2014a

Backlash

Model behavior of system with play (HDL Coder)

Description

The Backlash block is available with Simulink.

For information about the simulation behavior and block parameters, see Backlash.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

The **Deadband width** and **Initial output** parameters support only scalar values.

Introduced in R2014b

Bias

Add bias to input (HDL Coder)

Description

The Bias block is available with Simulink.

For information about the simulation behavior and block parameters, see Bias.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bilateral Filter

2-D bilateral filtering (HDL Coder)

Description

The Bilateral Filter block is available with Vision HDL Toolbox™.

For information about the simulation behavior and block parameters, see Bilateral Filter.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

Biquad Filter

Model biquadratic IIR (SOS) filters (HDL Coder)

Description

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

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

Programmable Filter Support

HDL Coder supports programmable filters for Biquad Filter blocks.

1. On the filter block mask, set **Coefficient source** to **Input port(s)**.
2. Connect vector signals to the `Num` and `Den` coefficient ports.

The following limitations apply to the HDL optimizations for a programmable Biquad Filter block:

- Fully serial and partly serial architectures are not supported. **Architecture** must be set to `Fully parallel`.
- Canonical signed digit (CSD) multiplier optimization is not supported. `CoeffMultipliers` must be set to `multiplier`.

Multichannel Filter Support

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

1. Connect a vector signal to the Biquad Filter block input port.
2. Specify **Input processing** as `Elements as channels (sample based)`.
3. To reduce area by sharing the filter kernel between channels, set the **StreamingFactor** parameter of the subsystem to the number of channels. See the Streaming section of “Subsystem Optimizations for Filters”.

HDL Architecture

Block Optimizations

To use block-level optimizations to reduce hardware resources, select a serial **Architecture**. Then set either `NumMultipliers` or `Folding Factor`. See “HDL Filter Properties” on page 3-20.

When you select a serial architecture, set **Filter structure** to `Direct form I` or `Direct form II`. The direct form transposed structures are not supported with serial architectures.

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

Filter Structure	Pipeline Register Placement	Latency (Clock Cycles)
Any	Pipeline registers are added between the filter sections.	`NS-1`, where `NS` is number of sections.

Subsystem Optimizations

This block can participate in subsystem-level optimizations such as sharing, streaming, and pipelining. For the block to participate in subsystem-level optimizations, set **Architecture** to `Fully parallel`. See “Subsystem Optimizations for Filters”.

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See alsoCoeffMultipliers.

FoldingFactor

Specify a serial implementation of an IIR SOS filter by the number of cycles it takes to generate the result. See also FoldingFactor.

NumMultipliers

Specify a serial implementation of an IIR SOS filter by the number of hardware multipliers that are generated. See also NumMultipliers.

For HDL filter property descriptions, see "HDL Filter Block Properties".

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

InputPipeline

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

OutputPipeline

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

Restrictions

- Frame input is not supported for HDL code generation.
- You must set **Initial conditions** to `0`. HDL code generation is not supported for nonzero initial states.
- You must select **Optimize unity scale values**.
- You cannot use the Biquad Filter block inside a Resettable Synchronous Subsystem.

Introduced in R2014a

Birds-Eye View

Transform front-facing camera image into top-down view (HDL Coder)

Description

The Birds-Eye View block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Birds-Eye View.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

Bit Clear

Set specified bit of stored integer to zero (HDL Coder)

Description

The Bit Clear block is available with Simulink.

For information about the simulation behavior and block parameters, see Bit Clear.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Concat

Concatenates up to 128 input words into single output (HDL Coder)

Description

The Bit Concat block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Reduce

AND, OR, or XOR bit reduction of all input signal bits to single bit (HDL Coder)

Description

The Bit Reduce block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Rotate

Rotate input signal by bit positions (HDL Coder)

Description

The Bit Rotate block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Set

Set specified bit of stored integer to one (HDL Coder)

Description

The Bit Set block is available with Simulink.

For information about the simulation behavior and block parameters, see Bit Set.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Shift

Logical or arithmetic shift of input signal (HDL Coder)

Description

The Bit Shift block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bit Slice

Return field of consecutive bits from input signal (HDL Coder)

Description

The Bit Slice block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bitwise Operator

Specified bitwise operation on inputs (HDL Coder)

Description

The Bitwise Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Bitwise Operator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

BPSK Demodulator Baseband

Demodulate BPSK-modulated data (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see BPSK Demodulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

BPSK Modulator Baseband

Modulate using binary phase shift keying method (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see BPSK Modulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Bus Assignment

Replace specified bus elements (HDL Coder)

Description

The Bus Assignment block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus Assignment.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

See Also

Topics

“Buses”

Introduced in R2014b

Bus Creator

Create signal bus (HDL Coder)

Description

The Bus Creator block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus Creator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

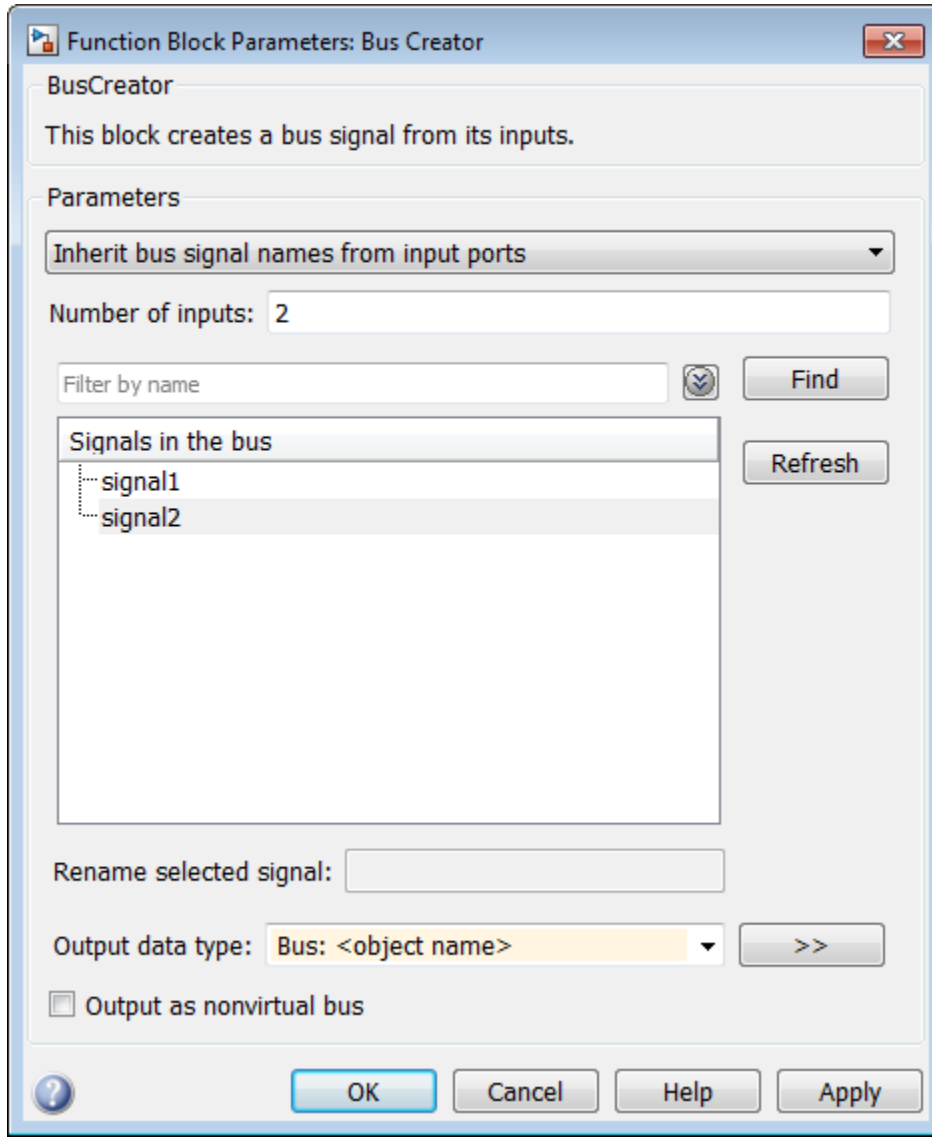
OutputPipeline

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

Restrictions

Setup

For **Output data type**, specify a bus object.


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

See Also

Topics

“Buses”

Introduced in R2014a

Bus Selector

Select signals from incoming bus (HDL Coder)

Description

The Bus Selector block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus Selector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

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

See Also

Topics

"Buses"

Introduced in R2014a

Bus to Vector

Convert virtual bus to vector (HDL Coder)

Description

The Bus to Vector block is available with Simulink.

For information about the simulation behavior and block parameters, see Bus to Vector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

See Also

Topics

“Buses”

Introduced in R2016a

Channelizer HDL Optimized

Polyphase filter bank and fast Fourier transform—optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Channelizer HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017a

Chart

Implement control logic with finite state machine (HDL Coder)

Description

The Chart block is available with Stateflow®.

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

Tunable Parameters

You can use a tunable parameter in a Stateflow Chart intended for HDL code generation.

For more information, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

This block has a single, default HDL architecture.

Active State Output

To generate an output port in the HDL code that shows the active state, select **Create output port for monitoring** in the Properties window of the chart. The output is an enumerated data type. See “Use Active State Output Data” (Stateflow).

Registered Output

If you want to insert an output register that delays the chart output by a simulation cycle, use the OutputPipeline block property.

HDL Block Properties

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

InputPipeline

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

InstantiateFunctions

Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions”.

LoopOptimization

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

MapPersistentVarsToRAM

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

Complex Data Support

This block supports code generation for complex signals.

Restrictions

Location of Charts in the Model

A chart intended for HDL code generation must be part of a Simulink subsystem. If the chart for which you want to generate code is at the root level of your model, embed the chart in a subsystem. Connect the relevant signals to the subsystem inputs and outputs.

Data Types

The current release supports a subset of MATLAB data types in charts intended for use in HDL code generation. Supported data types are

- Signed and unsigned integer
- Double and single

  Note Some results obtained from HDL code generated for models using double or single data types are not bit-true to results from simulation of the original model.
- Fixed point
- Boolean
- Enumeration

Note Except for data types assigned to ports, multidimensional arrays of these types are supported. Port data types must be either scalar or vector.

Chart Initialization

You must enable the chart property **Execute (enter) Chart at Initialization**. This option executes the update chart function immediately following chart initialization. The option is required for HDL because outputs must be available at time 0 (hardware reset). “Execution of a Chart at Initialization” (Stateflow) describes existing restrictions under this property.

The reset action must not entail the delay of combinatorial logic. Therefore, do not perform arithmetic in initialization actions.

To generate HDL code that is more readable and has better synthesis results, enable the **Initialize Outputs Every Time Chart Wakes Up** chart property. If you use a `Moore` state machine, HDL Coder generates an error if you disable the chart property.

If you disable **Initialize Outputs Every Time Chart Wakes Up**, the chart output is persistent, so the generated HDL code must internally register the output values.

Imported Code

A chart intended for HDL code generation must be entirely self-contained. The following restrictions apply:

- Do not call MATLAB functions other than `min` or `max`.
- Do not use MATLAB 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.

Messages

Stateflow messages are not supported for HDL code generation.

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” (Stateflow) and “Activate a Simulink Block Using Output Events” (Stateflow).

Loops

Other than `for` loops, do not explicitly use loops in a chart intended for HDL code generation. Observe the following restrictions on `for` loops:

- The data type of the loop counter variable must be `int32`.
- HDL Coder supports only constant-bounded loops.

The `for` loop example, `sf_for`, shows a design pattern for a `for` loop using a graphical function.

Other Restrictions

HDL Coder imposes additional restrictions on the use of classic chart features. These limitations exist because HDL does not support some features of general-purpose sequential programming languages.

- Do not define local events in a chart from which HDL code is generated.

  Do not use the following implicit events:

  - `enter`
  - `exit`
  - `change`

  You can use the following implicit events:

  - `wakeup`
  - `tick`

  You can use temporal logic if the base events are limited to these types of implicit events.

  Note Absolute-time temporal logic is not supported for HDL code generation.

- Do not use recursion through graphical functions. HDL Coder does not currently support recursion.
- Avoid unstructured code. Although charts allow unstructured code (through transition flow diagrams and graphical functions), this usage results in `goto`

statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements. Therefore, do not use unstructured flow diagrams.

- If you have not selected the **Initialize Outputs Every Time Chart Wakes Up** chart option, do not read from output ports.
- Do not use Data Store Memory objects.
- Do not use pointer (`&`) or indirection (`*`) operators. See "Pointer and Address Operations" (Stateflow).
- If a chart gets a run-time overflow error during simulation, it is possible to disable data range error checking and generate HDL code for the chart. However, in such cases, some results obtained from the generated HDL code might not be bit-true to results from the simulation. The recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

See Also

Message Viewer | State Transition Table | Truth Table

Topics

"Generate HDL for Mealy and Moore Finite State Machines"
"Design Patterns Using Advanced Chart Features"
"Hardware Realization of Stateflow Semantics"

Introduced in R2014a

Check Discrete Gradient

Check that absolute value of difference between successive samples of discrete signal is less than upper bound (HDL Coder)

Description

The Check Discrete Gradient block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Discrete Gradient.

HDL Architecture

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

Introduced in R2014a

Check Dynamic Gap

Check that gap of possibly varying width occurs in range of signal's amplitudes (HDL Coder)

Description

The Check Dynamic Gap block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Gap.

HDL Architecture

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

Introduced in R2014a

Check Dynamic Lower Bound

Check that one signal is always less than another signal (HDL Coder)

Description

The Check Dynamic Lower Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Lower Bound.

HDL Architecture

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

Introduced in R2014a

Check Dynamic Range

Check that signal falls inside range of amplitudes that varies from time step to time step (HDL Coder)

Description

The Check Dynamic Range block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Range.

HDL Architecture

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

Introduced in R2014a

Check Dynamic Upper Bound

Check that one signal is always greater than another signal (HDL Coder)

Description

The Check Dynamic Upper Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Dynamic Upper Bound.

HDL Architecture

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

Introduced in R2014a

Check Input Resolution

Check that input signal has specified resolution (HDL Coder)

Description

The Check Input Resolution block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Input Resolution.

HDL Architecture

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

Introduced in R2014a

Check Static Gap

Check that gap exists in signal's range of amplitudes (HDL Coder)

Description

The Check Static Gap block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Gap.

HDL Architecture

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

Introduced in R2014a

Check Static Lower Bound

Check that signal is greater than (or optionally equal to) static lower bound (HDL Coder)

Description

The Check Static Lower Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Lower Bound.

HDL Architecture

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

Introduced in R2014a

Check Static Range

Check that signal falls inside fixed range of amplitudes (HDL Coder)

Description

The Check Static Range block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Range.

HDL Architecture

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

Introduced in R2014a

Check Static Upper Bound

Check that signal is less than (or optionally equal to) static upper bound (HDL Coder)

Description

The Check Static Upper Bound block is available with Simulink.

For information about the simulation behavior and block parameters, see Check Static Upper Bound.

HDL Architecture

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

Introduced in R2014a

Chroma Resampler

Downsample or upsample chrominance component (HDL Coder)

Description

The Chroma Resampler block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Chroma Resampler.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015a

CIC Decimation

Decimate signal using Cascaded Integrator-Comb filter (HDL Coder)

Description

The CIC Decimation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see CIC Decimation.

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

HDL Architecture

AddPipelineRegisters Support

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

Pipeline Register Placement	Latency (clock cycles)
A pipeline register is added between the comb stages of the differentiators.	`NS-1`, where `NS` is number of sections (at the output side).

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option `Zero-latency decimator` is not supported for HDL code generation. From the **Filter Structure** drop-down list, select `Decimator`.

Introduced in R2014a

CIC Interpolation

Interpolate signal using Cascaded Integrator-Comb filter (HDL Coder)

Description

The CIC Interpolation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see CIC Interpolation.

HDL Coder supports **Coefficient source** options **Dialog parameters** and **Filter object**.

HDL Architecture

AddPipelineRegisters Support

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

Pipeline Register Placement	Latency (clock cycles)
A pipeline register is added between the comb stages of the differentiators.	`NS`, the number of sections (at the input side).

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the **Filter Structure** option `Zero-latency interpolator` is not supported for HDL code generation. From the **Filter Structure** drop-down list, select `Interpolator`.
- When you use **AddPipelineRegisters**, delays in parallel paths are not automatically balanced. Manually add delays where required by your design.

Introduced in R2014a

Closing

Morphological close of binary pixel data (HDL Coder)

Description

The Closing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Closing.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Closing block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Color Space Converter

Convert color information between color spaces (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Color Space Converter.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015a

Compare To Constant

Determine how signal compares to specified constant (HDL Coder)

Description

The Compare To Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Compare To Constant.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Compare To Zero

Determine how signal compares to zero (HDL Coder)

Description

The Compare To Zero block is available with Simulink.

For information about the simulation behavior and block parameters, see Compare To Zero.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Complex to Magnitude-Angle HDL Optimized

Compute magnitude and/or phase angle of complex signal—optimized for HDL code generation using the CORDIC algorithm (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Complex to Magnitude-Angle HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014b

Complex to Real-Imag

Output real and imaginary parts of complex input signal (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Complex to Real-Imag.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Constant

Generate constant value (HDL Coder)

Description

The Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Constant.

Tunable Parameters

You can use a tunable parameter in a Constant block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

Architecture	Parameters	Description
`default Constant`	None	This implementation emits the value of the Constant block.
`Logic Value`	None	By default, this implementation emits the character `'Z'` for each bit in the signal. For example, for a 4-bit signal, the implementation would emit `'ZZZZ'`.
	`{'Value', 'Z'}`	If the signal is in a high-impedance state, use this parameter value. This implementation emits the character `'Z'` for each bit in the signal. For example, for a 4-bit signal, the implementation would emit `'ZZZZ'`.
	`{'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.

Introduced in R2014a

Constellation Diagram

Display constellation diagram for input signals (HDL Coder)

Description

The Constellation Diagram block is available with Communications System Toolbox.

For information about the simulation behavior and block parameters, see Constellation Diagram.

HDL Architecture

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

Introduced in R2014a

Convert 1-D to 2-D

Reshape 1-D or 2-D input to 2-D matrix with specified dimensions (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Convert 1-D to 2-D.

HDL Architecture

This block has a pass-through implementation.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Convolutional Deinterleaver

Restore ordering of symbols that were permuted using shift registers (HDL Coder)

Description

The Convolutional Deinterleaver block is available with Communications System Toolbox.

For information about 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.

Introduced in R2014a

Convolutional Encoder

Create convolutional code from binary data (HDL Coder)

Description

The Convolutional Encoder block is available with Communications System Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Encoder.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- Input data requirements:
  - Must be sample-based,
  - Must have a `boolean` or `ufix1` data type.
- HDL Coder supports only the following coding rates:
  - 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.
- You cannot use the Convolutional Encoder block inside a Resettable Synchronous Subsystem.

Introduced in R2014a

Convolutional Interleaver

Permute input symbols using set of shift registers (HDL Coder)

Description

The Convolutional Interleaver block is available with Communications System Toolbox.

For information about 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.

Introduced in R2014a

Cosine

Implement fixed-point cosine wave using lookup table approach that exploits quarter wave symmetry (HDL Coder)

Description

The Cosine block is available with Simulink.

For information about the simulation behavior and block parameters, see Sine, Cosine.

HDL Architecture

The HDL code implements Cosine using the quarter-wave lookup table that you specify in the Simulink block parameters.

To avoid generating a division operator (/) in the HDL code, for **Number of data points for lookup table**, enter `(2^n)+1`. *n* is an integer.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

See Also

Cosine HDL Optimized | Sine | Sine HDL Optimized

Introduced in R2014a

Cosine HDL Optimized

Implement fixed-point cosine wave by using lookup table approach that exploits quarter wave symmetry optimized for HDL code generation

Description

The Cosine HDL Optimized block is available in the Lookup Tables library in HDL Coder. For information about the simulation behavior and block parameters, see Cosine HDL Optimized.

For the most efficient HDL implementation, configure the block with an exact power of two as the number of elements. In the Block Parameters dialog box, for **Number of data points**, specify an integer that is an exact power of two. By default, the **Number of data points** is 64.

When you specify a power of two for the **Number of data points**, the lookup tables precede a register without reset after HDL code generation. The combination of the lookup table block and register without reset map efficiently to RAM on the target device.

HDL Architecture

The HDL code implements the Cosine HDL Optimized block by using the quarter-wave lookup table that you specify in the Simulink block parameters.

To generate code that is optimized for area and speed, for **Number of data points**, enter `(2^n)`. *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”.

See Also

Cosine | Sine | Sine HDL Optimized

Introduced in R2016b

Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere (HDL Coder)

Description

The Coulomb and Viscous Friction block is available with Simulink.

For information about the simulation behavior and block parameters, see Coulomb and Viscous Friction.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

HDL code generation does not support complex input.

Introduced in R2014b

Counter Free-Running

Count up and overflow back to zero after reaching maximum value for specified number of bits (HDL Coder)

Description

The Counter Free-Running block is available with Simulink.

For information about the simulation behavior and block parameters, see Counter Free-Running.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Counter Limited

Count up and wrap back to zero after outputting specified upper limit (HDL Coder)

Description

The Counter Limited block is available with Simulink.

For information about the simulation behavior and block parameters, see Counter Limited.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Data Type Conversion

Convert input signal to specified data type (HDL Coder)

Description

The Data Type Conversion block is available with Simulink.

For information about the simulation behavior and block parameters, see Data Type Conversion.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

With the HDL Model Checker, you can replace Data Type Conversion blocks that use the `Stored Integer (SI)` mode and convert between floating-point and fixed-point data types with Float Typecast blocks.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

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.

Introduced in R2014a

Data Type Duplicate

Force all inputs to same data type (HDL Coder)

Description

The Data Type Duplicate block is available with Simulink.

For information about the simulation behavior and block parameters, see Data Type Duplicate.

HDL Architecture

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

Introduced in R2014a

Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals (HDL Coder)

Description

The Data Type Propagation block is available with Simulink.

For information about the simulation behavior and block parameters, see Data Type Propagation.

HDL Architecture

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

Introduced in R2014a

DC Blocker

Block DC component (HDL Coder)

Description

The DC Blocker block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see DC Blocker.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014b

Dead Zone

Provide region of zero output (HDL Coder)

Description

The Dead Zone block is available with Simulink.

For information about the simulation behavior and block parameters, see Dead Zone.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014b

Dead Zone Dynamic

Set inputs within bounds to zero (HDL Coder)

Description

The Dead Zone Dynamic block is available with Simulink.

For information about the simulation behavior and block parameters, see Dead Zone Dynamic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014b

Decrement Real World

Decrease real world value of signal by one (HDL Coder)

Description

The Decrement Real World block is available with Simulink.

For information about the simulation behavior and block parameters, see Decrement Real World.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Decrement Stored Integer

Decrease stored integer value of signal by one (HDL Coder)

Description

The Decrement Stored Integer block is available with Simulink.

For information about the simulation behavior and block parameters, see Decrement Stored Integer.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Delay

Delay input signal by fixed or variable sample periods (HDL Coder)

Description

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

Block Parameter Setting	Description
Set **External reset** to `Level` .	Generates a reset port in the HDL code.
Select **Show enable port**.	Generates an enable port in the HDL code.
For **Initial condition**, set **Source** to `Dialog` and enter the value.	Specifies an initial condition for the block.
Set **Input processing** to `Columns as channels (frame based)`.	Expects vector input data, where each element of the vector represents a sample in time.

Additional Settings When Using State Control Block

If you use a State Control block with the Delay block inside a subsystem in your Simulink model, use these additional settings.

Block Parameter Setting	Description
Set **External reset** to `Level hold` for `Synchronous` mode and `Level` for `Classic` mode of the State Control block.	Generates a reset port in the HDL code.
Set **Delay length** to zero for a Delay block with an external enable port.	Treated as a wire in only `Synchronous` mode of the State Control block.
Set **Delay length** to zero for a Delay block with an external reset port.	Treated as a wire in `Synchronous` and `Classic` modes of the State Control block.

For more information about the State Control block, see State Control.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

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

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

UseRAM

Map delays to RAM instead of registers. The default is `off`. See also “UseRAM”.

Complex Data Support

This block supports code generation for complex signals.

Restrictions

For **Initial condition** and **Delay length**, **Source** set to `Input port` is not supported for HDL code generation.

Introduced in R2014a

Demosaic Interpolator

Construct RGB pixel data from Bayer pattern pixels (HDL Coder)

Description

The Demosaic Interpolator block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Demosaic Interpolator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Demosaic Interpolator block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Demux

Extract and output elements of vector signal (HDL Coder)

Description

The Demux block is available with Simulink.

For information about the simulation behavior and block parameters, see Demux.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Deserializer1D

Convert scalar stream or smaller vectors to vector signal (HDL Coder)

Description

The Deserializer1D block is available with Simulink.

For information about 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”.

Introduced in R2014b

Digital Filter (Obsolete)

Filter each channel of input over time using static or time-varying digital filter implementations (HDL Coder)

Description

The Digital Filter block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Digital Filter.

Note Use of Digital Filter block in future releases is not recommended. Existing instances will continue to operate, but certain functionality will be disabled. See "Functionality being removed or replaced for blocks and System objects" (DSP System Toolbox). We strongly recommend using Discrete FIR Filter or Biquad Filter in new designs.

HDL Architecture

When you specify `SerialPartition` and `ReuseAccum` for a Digital Filter block, observe the following constraints.

- If you specify **Dialog parameters** as the `Coefficient source`:
  - Set **Transfer function type** to `FIR (all zeros)`.
  - Select **Filter structure** as one of: `Direct form`, `Direct form symmetric`, or `Direct form asymmetric`.

Distributed Arithmetic Support

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

Architecture	Supported FIR Structures
default	FIR, Asymmetric FIR, and Symmetric FIR

AddPipelineRegisters Support

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

Architecture	Pipeline Register Placement	Latency (clock cycles)
FIR, Asymmetric FIR, and Symmetric FIR filters	A pipeline register is added between levels of a tree-based adder.	`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 **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See alsoCoeffMultipliers.

DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition.

MultiplierInputPipeline

Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline.

MultiplierOutputPipeline

Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline.

ReuseAccum

Enable or disable accumulator reuse in a serial filter implementation. Set **ReuseAccum** to `on` to use a cascade-serial implementation. See also ReuseAccum.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Coefficients and Data Support

Except for decimator and interpolator filter structures, HDL Coder supports use of complex coefficients and complex input signals for all filter structures of the Digital Filter block.

Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- HDL Coder does not support the Digital Filter block **Input port(s)** option for HDL code generation.

Introduced in R2015a

Dilation

Morphological dilate of binary pixel data (HDL Coder)

Description

The Dilation block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Dilation.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Dilation block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Direct Lookup Table (n-D)

Index into N-dimensional table to retrieve element, column, or 2-D matrix (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Direct Lookup Table (n-D).

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

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.

Introduced in R2014a

Discrete FIR Filter

Model finite impulse response filter (HDL Coder)

Description

The Discrete FIR Filter block is available with Simulink, but a DSP System Toolbox license is required to use a filter structure other than direct form.

For information about the simulation behavior and block parameters, see Discrete FIR Filter.

For high throughput sample-based FIR filter implementation with cycle-accurate modeling, use Discrete FIR Filter HDL Optimized.

Multichannel Filter Support

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents an independent channel.

1. Connect a vector signal to the Discrete FIR Filter block input port.
2. Specify **Input processing** as `Elements as channels (sample based)`.
3. To reduce area by sharing the filter kernel between channels, set the **StreamingFactor** of the subsystem to the number of channels. See the Streaming section of “Subsystem Optimizations for Filters”.

Programmable Filter Support

HDL Coder supports programmable filters for Discrete FIR Filter blocks.

1. On the filter block mask, set **Coefficient source** to **Input port**.
2. Connect a vector signal to the `Num` coefficient port.

Frame-Based Input Support

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents a sample in time. You can use an input vector of up to

512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types.

1. Connect a vector signal to the Discrete FIR Filter block input port.
2. Specify **Input processing** as `Columns as channels (frame based)`.
3. Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to `Frame Based`. The block implements a parallel HDL architecture. See "Frame-Based Architecture".

Control Ports

You can generate HDL code for filters with or without the optional enable port, and with or without the optional reset port.

HDL Architecture

To reduce area or increase speed, the Discrete FIR Filter block supports either block-level optimizations or subsystem-level optimizations. When you enable block optimizations, the block cannot participate in subsystem optimizations. Use block optimizations when your design is a single one-channel filter. Use subsystem optimizations to share resources across multiple channels or multiple filters.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

Block Optimizations

To use block-level optimizations to reduce hardware resources, set **Architecture** to one of the serial options. See "HDL Filter Architectures".

When you specify **SerialPartition** and **ReuseAccum** for a Discrete FIR Filter block, set **Filter structure** to `Direct form`, `Direct form symmetric`, or `Direct form asymmetric`. The `Direct form transposed` structure is not supported with serial architectures.

To minimize multipliers by replacing them with LUTs and shift registers, use a distributed arithmetic (DA) filter implementation. See “Distributed Arithmetic for HDL Filters”.

When you select the `Distributed Arithmetic (DA)` architecture and use the **DALUTPartition** and **DARadix** distributed arithmetic properties, set **Filter structure** to `Direct form`, `Direct form symmetric`, or `Direct form asymmetric`. The `Direct form transposed` structure is not supported with distributed arithmetic.

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.

To improve clock speed, use **AddPipelineRegisters** to use a pipelined adder tree rather than the default linear adder. You can also specify the number of pipeline stages before and after the multipliers. See “HDL Filter Architectures”.

Subsystem Optimizations

This block can participate in subsystem-level optimizations such as sharing, streaming, and pipelining. For the block to participate in subsystem-level optimizations, set the **Architecture** to `Fully parallel`. See “Subsystem Optimizations for Filters”.

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

ChannelSharing

For a multichannel filter, generate a single filter implementation to be shared between channels. See also ChannelSharing.

CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully

parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See alsoCoeffMultipliers.

DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition.

DARadix

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 (`2^3`) generates a DA implementation that computes three sums at a time. The default value is `2^1`, which generates a fully serial DA implementation. See also DARadix.

MultiplierInputPipeline

Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline.

MultiplierOutputPipeline

Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline.

ReuseAccum

Enable or disable accumulator reuse in a serial filter implementation. Set **ReuseAccum** to `on` to use a cascade-serial implementation. See also ReuseAccum.

SerialPartition

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- HDL code generation is not supported for:
  - Unsigned input data.
  - Nonzero initial states. You must set **Initial states** to `0`.
  - **Filter Structure**: `Lattice MA`.
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers**is hidden from the HDL Block Properties dialog box.

Programmable filters are not supported for:

- Architectures for which you specify the coefficients by dialog box parameters (for example, complex input and coefficients with serial architecture)
- distributed arithmetic (DA)
- **CoeffMultipliers** set to `csd` or `factored-csd`
- Frame-based input

Frame-based input filters are not supported for:

- Optional block-level reset and enable control signals
- Resettable and enabled subsystems
- Complex input signals or coefficients
- Programmable coefficients
- Multichannel input

- Sharing and streaming optimizations

See Also

Topics

Generate HDL Code for FIR Programmable Filter

Introduced in R2014a

Discrete FIR Filter HDL Optimized

Model finite impulse response filter — HDL optimized (HDL Coder)

Description

The Discrete FIR Filter HDL Optimized block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Discrete FIR Filter HDL Optimized.

HDL Architecture

The block implements a direct-form systolic FIR filter architecture based on multiply-accumulate operations with pipeline registers. This architecture makes the block well-suited for high throughput HDL code generation targeted for FPGAs with dedicated DSP blocks. You can make tradeoffs between throughput and resource utilization of the generated code by using block parameters.

- For highest throughput, clear **DSP resource sharing**. The generated code filters new input data on every cycle.
- For reduced area, select **DSP resource sharing**. Then specify a **Sharing factor** of $N \geq 2$. In this case, the generated code can filter only input samples that are at least N cycles apart.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- The Discrete FIR Filter HDL Optimized block does not support:
  - HDL code generation for floating-point input data types.
  - Complex coefficients.
  - Vector inputs. The block is sample based, accepting one scalar at a time.
  - Programmable filters. Specify filter coefficients by using the **Coefficients** block parameter instead.
  - Resource sharing optimization through HDL Coder. Use the **DSP resource sharing** and **Sharing factor** block parameters instead.

  For FIR filters with programmable or complex coefficients, or with multichannel or frame-based inputs, consider using the Discrete FIR Filter block instead.

Introduced in R2017a

Discrete PID Controller

Simulate discrete-time PID controllers (HDL Coder)

Description

The Discrete PID Controller block is available with Simulink.

For information about the simulation behavior and block parameters, see Discrete PID Controller.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

HDL code generation does not support the following settings:

- **Continuous-time**.
- **Filter method** > **Backward Euler** or **Trapezoidal**.
- **Source** > **external**.
- **External reset** > **rising**, **falling**, **either**, or **level**.
- If inputs are of type Double, **Anti-windup method** > **clamping**.

Introduced in R2014a

Discrete Transfer Fcn

Implement discrete transfer function (HDL Coder)

Description

The Discrete Transfer Fcn block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to minimum or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “Mantissa Multiplier Strategy”.

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

Introduced in R2014a

Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal (HDL Coder)

Description

The Discrete-Time Integrator block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to minimum or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “Mantissa Multiplier Strategy”.

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.

Introduced in R2014a

Display

Show value of input (HDL Coder)

Description

The Display block is available with Simulink.

For information about the simulation behavior and block parameters, see Display.

HDL Architecture

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

Introduced in R2014a

Divide

Divide one input by another (HDL Coder)

Description

The Divide block is available with Simulink.

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

HDL Architecture

To perform an HDL-optimized divide operation, connect a Product block to a Divide block in reciprocal mode. For information about the Divide block in reciprocal mode, see “Reciprocal Mode” on page 3-136.

Default Mode

In default mode, the Divide block supports only integer data types for HDL code generation.

Architecture	Parameters	Description
`default` `Linear`	None	Generate a divide (/) operator in the HDL code.

Reciprocal Mode

When **Number of Inputs** is set to `/`, the Divide block is in reciprocal mode.

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

In reciprocal mode, the Divide block has the HDL block implementations described in the following table.

Architectures	Parameters	Additional cycles of latency	Description
`default` `Linear`	None	0	When you compute a reciprocal, use the HDL divide (/) operator to implement the division.
`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.

Architectures	Parameters	Additional cycles of latency	Description
`ReciprocalRsqrtBasedNewtonSingleRate`	`Iterations`	Signed input: (`Iterations` * 4) + 8 Unsigned input: (`Iterations` * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The default value for `Iterations` is 3. The recommended value for `Iterations` is between 2 and 10. If `Iterations` is outside the recommended range, the coder displays a message.

The Newton-Raphson iterative method:

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

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

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

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “Mantissa Multiplier Strategy”.

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.

Introduced in R2014a

DocBlock

Create text that documents model and save text with model (HDL Coder)

Description

The DocBlock block is available with Simulink.

For information about the simulation behavior and block parameters, see DocBlock.

HDL Architecture

Architecture	Description
`Annotation` (default)	Insert text as comment in the generated code.
`HDLText`	Integrate text as custom HDL code.
`No HDL`	Do not generate HDL code for this block.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

TargetLanguage

Language of the text, either Verilog or VHDL. The default is VHDL.

When **Architecture** is `HDLText`, this property is available. To learn more, see "Integrate Custom HDL Code Using DocBlock".

Restrictions

- **Document type** must be `Text`.

  HDL Coder does not support the `HTML` or `RTF` options.
- You can have a maximum of two DocBlock blocks with **Architecture** set to `HDLText` in the same subsystem.

  If you have two DocBlock blocks, one must have **TargetLanguage** set to `VHDL`, and the other must have **TargetLanguage** set to `Verilog`. When generating code, HDL Coder only integrates the custom code from the DocBlock that matches the target language for code generation.

See Also

Topics

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

Introduced in R2014a

Dot Product

Generate dot product of two vectors (HDL Coder)

Description

The Dot Product block is available with Simulink.

For information about the simulation behavior and block parameters, see Dot Product.

HDL Architecture

Architecture	Description
`Linear` (default)	Generates a linear chain of adders to compute the sum of products.
`Tree`	Generates a tree structure of adders to compute the sum of products.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Downsample

Resample input at lower rate by deleting samples (HDL Coder)

Description

The Downsample block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Downsample.

Best Practices

It is good practice to follow the Downsample block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation”.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Restrictions

- **Input processing** set to `Columns as channels (frame based)` is not supported.
- For **Input processing** set to `Elements as channels (sample based)`, select `Allow multirate processing`. With this setting, if **Sample offset** is set to 0, **Initial conditions** has no effect on generated code.

Introduced in R2014a

Dual Port RAM

Dual port RAM with two output ports (HDL Coder)

Description

The Dual Port RAM block is available with Simulink.

For information about 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 3-147.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Dual Port RAM System

Dual port RAM with two output ports and ability to specify initial value (HDL Coder)

Description

The Dual Port RAM System block is available in the HDL RAMs sublibrary in the HDL Coder block library. For information about the simulation behavior and block parameters, see Dual Port RAM System, Simple Dual Port RAM System, Single Port RAM System.

The Dual Port RAM System block is a MATLAB System block that uses the `hdl.RAM` System object™. The simulation and HDL code generation behavior of the block is similar to the Dual Port RAM block. In addition, you can:

- Specify an initial value for the RAM. Double-click the block to open the Block Parameters dialog box, and then enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.
- Obtain higher performance and support for large data memories.

Note When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents.To reset the RAM, make sure that you implement the reset logic.

HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM` System object.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

See Also

System Objects

`hdl.RAM`

Blocks

Simple Dual Port RAM System | Single Port RAM System

Topics

“HDL Code Generation from hdl.RAM System Object”
“Getting Started with RAM and ROM in Simulink®”
“Implement RAM Using MATLAB Code”
“HDL Code Generation for System Objects”

Introduced in R2017b

Dual Rate Dual Port RAM

Dual Port RAM that supports two rates (HDL Coder)

Description

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

For information about 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 3-152.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Edge Detector

Find edges of objects in image (HDL Coder)

Description

The Edge Detector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Edge Detector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Edge Detector block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Enable

Add enabling port to system (HDL Coder)

Description

The Enable block is available with Simulink.

For information about the simulation behavior and block parameters, see Enable.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

See Also

Enabled Subsystem

Introduced in R2014a

Enabled Subsystem

Represent subsystem whose execution is enabled by external input (HDL Coder)

Description

An enabled subsystem is a subsystem that receives a control signal via an Enable block. The enabled subsystem executes at each simulation step where the control signal has a positive value.

For detailed information on how to construct and configure enabled subsystems, see "Enabled Subsystems" (Simulink).

Best Practices

When using enabled subsystems in models targeted for HDL code generation, it is good practice to consider the following:

- For synthesis results to match Simulink results, the Enable port must be driven by registered logic (with a synchronous clock) on the FPGA.
- Put unit delays on Enabled Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- Enabled subsystems can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of enabled subsystem instances and the number of output ports per subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.

Architecture	Description
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

Restrictions

HDL Coder supports HDL code generation for enabled subsystems that meet the following conditions:

- The enabled subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The enable signal is a scalar.
- The data type of the enable signal is either `boolean` or `ufix1`.
- Outputs of the enabled subsystem have an initial value of 0.

- All inputs and outputs of the enabled subsystem (including the enable signal) run at the same rate.
- The **Show output port** parameter of the Enable block is set to `Off`.
- The **States when enabling** parameter of the Enable block is set to `held` (i.e., the Enable block does not reset states when enabled).
- The **Output when disabled** parameter for the enabled subsystem output ports is set to `held` (i.e., the enabled subsystem does not reset output values when disabled).
- If the DUT contains the following blocks, `RAMArchitecture` is set to `WithClockEnable`:
  - Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- The enabled subsystem does not contain the following blocks:
  - CIC Decimation
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation
  - Downsample
  - Upsample
  - HDL Cosimulation blocks for HDL Verifier
  - Rate Transition

Example

The Automatic Gain Controller example shows how you can use enabled subsystems in HDL code generation. To open the example, enter:

```
hdlcoder_agc
```

See Also

Enable | Subsystem

Introduced in R2014a

Enabled Synchronous Subsystem

Represent enabled subsystem that has synchronous reset and enable behavior (HDL Coder)

Description

The Enabled Synchronous Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see Enabled Synchronous Subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also "BalanceDelays".

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also "DistributedPipelining".

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also "FlattenHierarchy".

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

You can set this property In the HDL Workflow Advisor, 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 (;).

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

Values: `''` (default) | character vector

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

IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: `''` (default) | character vector

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: `''` (default) | character vector

Example: `'1.3'`

Restrictions

- Your DUT cannot be an Enabled Synchronous Subsystem.
- You cannot have a Delay block with an external reset port inside the subsystem.

See Also

Enable | Resettable Synchronous Subsystem | State Control | Synchronous Subsystem

Topics

"Resettable Subsystem Support in HDL Coder™"
"Using the State Control block to generate more efficient code with HDL Coder™"
"Synchronous Subsystem Behavior with the State Control Block"

Introduced in R2016a

Enumerated Constant

Generate enumerated constant value (HDL Coder)

Description

The Enumerated Constant block is available with Simulink.

For information about the simulation behavior and block parameters, see Enumerated Constant.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

OutputPipeline

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

Introduced in R2014a

Erosion

Morphological erode of binary pixel data (HDL Coder)

Description

The Erosion block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Erosion.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Erosion block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Error Rate Calculation

Compute bit error rate or symbol error rate of input data (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Error Rate Calculation.

HDL Architecture

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

Introduced in R2014a

Extract Bits

Output selection of contiguous bits from input signal (HDL Coder)

Description

The Extract Bits block is available with Simulink.

For information about the simulation behavior and block parameters, see Extract Bits.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Eye Diagram

Display multiple traces of modulated signal (HDL Coder)

Description

The Eye Diagram block is available with Communications System Toolbox.

For information about the simulation behavior and block parameters, see Eye Diagram.

HDL Architecture

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

Introduced in R2014b

FFT HDL Optimized

Fast Fourier transform—optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see FFT HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- If you use the FFT HDL Optimized block with the State Control block inside an Enabled Subsystem, the optional reset port is not supported. If you enable the reset port on the FFT HDL Optimized block in such a subsystem, the model will error on Update Diagram.

Introduced in R2014a

FIR Decimation

Filter and downsample input signals (HDL Coder)

Description

The FIR Decimation block is available with DSP System Toolbox.

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

HDL Coder supports **Coefficient source** options **Dialog parameters**, **Filter object**, or **Auto**. Programmable coefficients are not supported.

Frame-Based Input Support

HDL Coder supports the use of vector inputs to FIR Decimation blocks, where each element of the vector represents a sample in time. You can use an input vector of up to 512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types. The output is a column vector of reduced size, corresponding to your decimation factor.

1. Connect a column vector signal to the FIR Decimation block input port.
2. Specify **Input processing** as `Columns as channels (frame based)`.
3. Set **Rate options** to `Enforce single-rate processing`.
4. Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to `Frame Based`. The block implements a parallel HDL architecture. See “Frame-Based Architecture”.

HDL Architecture

To reduce area or increase speed, the FIR Decimator block supports block-level optimizations.

Right-click on the block or the subsystem to open the corresponding **HDL Properties** dialog box and set optimization properties.

Block Optimizations

To use block-level optimizations to reduce hardware resources, set **Architecture** to `Fully Serial` or `Partly Serial`. See "HDL Filter Architectures".

When you specify **SerialPartition** for a FIR Decimator block, set **Filter structure** to `Direct form`. The `Direct form transposed` structure is not supported with serial architectures. Accumulator reuse is not supported for FIR Decimation filters.

To minimize multipliers by replacing them with LUTs and shift registers, use a distributed arithmetic (DA) filter implementation. See "Distributed Arithmetic for HDL Filters".

When you select the `Distributed Arithmetic (DA)` architecture and use the **DALUTPartition** and **DARadix** distributed arithmetic properties, set **Filter structure** to `Direct form`. The `Direct form transposed` structure is not supported with distributed arithmetic.

To improve clock speed, use **AddPipelineRegisters** to use a pipelined adder tree rather than the default linear adder. This option is supported for `Direct form` architecture. You can also specify the number of pipeline stages before and after the multipliers. See "HDL Filter Architectures".

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See alsoCoeffMultipliers.

DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition.

DARadix

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 (`2^3`) generates a DA implementation that computes three sums at a time. The default value is `2^1`, which generates a fully serial DA implementation. See also DARadix.

MultiplierInputPipeline

Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline.

MultiplierOutputPipeline

Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline.

SerialPartition

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - `Slope and Bias scaling`
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers**is hidden from the HDL Block Properties dialog box.
- Frame-based input filters are not supported for:
  - Resettable and enabled subsystems
  - Complex input signals or coefficients
  - Sharing and streaming optimizations

Introduced in R2014a

FIR Interpolation

Upsample and filter input signals (HDL Coder)

Description

The FIR Interpolation block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see FIR Interpolation.

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

HDL Architecture

When you select `Fully Serial` architecture, the `SerialPartition` property is set on the FIR Interpolation Block.

Distributed Arithmetic Support

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

Architecture	Supported FIR Structures
Distributed Arithmetic (DA)	default

AddPipelineRegisters Support

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

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

HDL Filter Properties

AddPipelineRegisters

Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters.

CoeffMultipliers

Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set **CoeffMultipliers** to `csd` or `factored-csd`. The default is `multipliers`, which retains multipliers in the HDL. See alsoCoeffMultipliers.

DALUTPartition

Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set **DALUTPartition** to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition.

DARadix

Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 (`2^3`) generates a DA implementation that computes three sums at a time. The default value is `2^1`, which generates a fully serial DA implementation. See also DARadix.

MultiplierInputPipeline

Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline.

MultiplierOutputPipeline

Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline.

SerialPartition

Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- You must set **Initial conditions** to zero. HDL code generation is not supported for nonzero initial states.
- Vector and frame inputs are not supported for HDL code generation.
- When you select **Dialog parameters**, the following fixed-point options are not supported for HDL code generation:
  - **Coefficients**: `Slope and Bias scaling`
- **CoeffMultipliers** options are supported only when using a fully parallel architecture. When you select a serial architecture, **CoeffMultipliers**is hidden from the HDL Block Properties dialog box.

Introduced in R2014a

FIR Rate Conversion HDL Optimized

Upsample, filter, and downsample input signals—optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see FIR Rate Conversion HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015b

Floating Scope

Display signals generated during simulation (HDL Coder)

Description

The Floating Scope block is available with Simulink.

For information about the simulation behavior and block parameters, see Floating Scope.

HDL Architecture

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

Introduced in R2014a

Float Typecast

Typecast a floating-point type to an unsigned integer or vice versa (HDL Coder)

Description

The Float Typecast block is available in the HDL Floating Point Operations library in the HDL Coder block library. For information about the simulation behavior and ports, see Float Typecast.

The block casts the underlying bits of the input to the corresponding fixed-point or floating point representation. The input and output of the block contain the same number of bits. The block mask, behavior, and output data type changes dynamically depending on the input data type that you specify.

Input Data Type	Output Data Type
`single`	`uint32`
`double`	`ufix64`
`uint32`	`single`
`ufix64`	`double`

The block supports HDL code generation in the `Native Floating Point` mode. To use this mode, specify `single` or `uint32` data types as input to the block. With the HDL Model Checker, you can replace Data Type Conversion blocks that use the `Stored Integer (SI)` mode and convert between floating-point and fixed-point data types.

Complex Data Support

This block supports code generation for complex signals.

See Also

- `typecast`

- “Getting Started with HDL Coder Native Floating-Point Support”

Introduced in R2017b

For Each Subsystem

Repeatedly perform algorithm on each element or subarray of input signal and concatenate results (HDL Coder)

Description

To repeat the same algorithm for each element or subarray of the input signals, use the For Each Subsystem block. The block reduces simulation time because it processes individual elements or subarrays of the input signals simultaneously. For information about the simulation behavior and block parameters, see For Each Subsystem.

By using the For Each block inside the For Each Subsystem, you can specify how to partition elements of the input signals. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which to slice the input signal and the width of each slice respectively. To partition a row vector, specify the **Partition Dimension** as `2`. To partition a column vector, specify the **Partition Dimension** as `1`. To learn more about the block parameters, see For Each.

When you generate HDL code for the For Each Subsystem, the code generator uses a `for-generate` loop that iterates through elements of the input and output signals. The for-generate loop improves readability and reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.

Limitations

- You cannot use the For Each Subsystem block as the DUT.
- You cannot partition mask parameters of the For Each Subsystem for HDL code generation.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.

Architecture	Description
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also "DistributedPipelining".

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also "FlattenHierarchy".

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also "Resource Sharing".

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also "Streaming".

Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

See Also

Topics

"Generate HDL Code for Blocks Inside For Each Subsystem"

Introduced in R2017a

Frame Conversion

Specify sampling mode of output signal (HDL Coder)

Description

The Frame Conversion block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Frame Conversion.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

From

Accept input from Goto block (HDL Coder)

Description

The From block is available with Simulink.

For information about the simulation behavior and block parameters, see From.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

OutputPipeline

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

Introduced in R2014a

Gain

Multiply input by constant (HDL Coder)

Description

The Gain block is available with Simulink.

For information about the simulation behavior and block parameters, see Gain.

Tunable Parameters

You can use a tunable parameter in a Gain block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

`ConstMultiplierOptimization`	Description
`none`(*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.

ConstMultiplierOptimization	Description
`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.
`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

General

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

Note For certain values of the **Gain** parameter, native floating point implements the algorithm differently instead of using multipliers. For example, if you set the **Gain** parameter to `1`, the generated model uses a wire to pass the input to the output. If you set the **Gain** parameter to `-1`, the generated model shows a Unary Minus block that inverts the polarity of the input signal. This implementation reduces the latency and resource usage on the target platform.

You can specify these settings in the **Native Floating Point** tab for the Gain block.

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “Mantissa Multiplier Strategy”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Gamma Corrector

Apply or remove gamma correction (HDL Coder)

Description

The Gamma Corrector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Gamma Corrector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015a

General CRC Generator HDL Optimized

Generate CRC code bits and append to input data, optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see General CRC Generator HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

General CRC Syndrome Detector HDL Optimized

Detect errors in input data using CRC (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see General CRC Syndrome Detector HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

General Multiplexed Deinterleaver

Restore ordering of symbols using specified-delay shift registers (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see General Multiplexed Deinterleaver.

HDL Architecture

The implementation for the General Multiplexed Deinterleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to`none`.

When you set `ResetType` to `none`, reset is not applied to the shift registers. When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Introduced in R2014a

General Multiplexed Interleaver

Permute input symbols using set of shift registers with specified delays (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see General Multiplexed Interleaver.

HDL Architecture

The implementation for the General Multiplexed Interleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to `'none'`.

When you set `ResetType` to `'none'`, reset is not applied to the shift registers. Mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface.)

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Introduced in R2014a

Goto

Pass block input to From blocks (HDL Coder)

Description

The Goto block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

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

Introduced in R2014a

Grayscale Closing

Morphological close of grayscale pixel data (HDL Coder)

Description

The Grayscale Closing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Closing.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Grayscale Closing block inside a Resettable Synchronous Subsystem.

Introduced in R2016a

Grayscale Dilation

Morphological dilate of grayscale pixel data (HDL Coder)

Description

The Grayscale Dilation block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Dilation.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Grayscale Dilation block inside a Resettable Synchronous Subsystem.

Introduced in R2016a

Grayscale Erosion

Morphological erode of grayscale pixel data (HDL Coder)

Description

The Grayscale Erosion block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Erosion.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Grayscale Erosion block inside a Resettable Synchronous Subsystem.

Introduced in R2016a

Grayscale Opening

Morphological open of grayscale pixel data (HDL Coder)

Description

The Grayscale Opening block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Grayscale Opening.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Grayscale Opening block inside a Resettable Synchronous Subsystem.

Introduced in R2016a

Ground

Ground unconnected input port (HDL Coder)

Description

The Ground block is available with Simulink.

For information about the simulation behavior and block parameters, see Ground.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

HDL Cosimulation

Cosimulate hardware component by communicating with HDL module instance executing in HDL simulator (HDL Coder)

Description

The HDL Cosimulation block is available with HDL Verifier.

For information about the simulation behavior and block parameters, see HDL Cosimulation.

HDL Coder supports HDL code generation for the following HDL Cosimulation blocks:

- HDL Verifier for use with Mentor Graphics® ModelSim®
- HDL Verifier for use with Cadence Incisive®

Each of the HDL Cosimulation blocks cosimulates a hardware component by applying input signals to, and reading output signals from, an HDL model that executes under an HDL simulator.

For information about timing, latency, data typing, frame-based processing, and other issues when setting up an HDL cosimulation, see “Define HDL Cosimulation Block Interface” (HDL Verifier).

You can use an HDL Cosimulation block with HDL Coder to generate an interface to your manually written or legacy HDL code. When an HDL Cosimulation block is included in a model, the coder generates a VHDL or Verilog interface, depending on the selected target language.

When the target language is VHDL, the generated interface includes:

- An entity definition. The entity defines ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. Clock enable and reset ports are also declared.
- An RTL architecture including a component declaration, a component configuration declaring signals corresponding to signals connected to the HDL Cosimulation ports, and a component instantiation.

- Port assignment statements as required by the model.

When the target language is Verilog, the generated interface includes:

- A module defining ports (input, output, and clock) corresponding in name and data type to the ports configured on the HDL Cosimulation block. The module also defines clock enable and reset ports, and `wire` declarations corresponding to signals connected to the HDL Cosimulation ports.
- A module instance.
- Port assignment statements as required by the model.

Before initiating code generation, to check the requirements for using the HDL Cosimulation block for code generation, select **Simulation** > **Update Diagram**.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

For implementation parameter descriptions, see “Customize Black Box or HDL Cosimulation Interface”.

See Also

Topics

“Generate a Cosimulation Model”

Introduced in R2014a

HDL Counter

Free-running or count-limited hardware counter (HDL Coder)

Description

The HDL Counter block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

If the bitwidth of the input signal to a HDL Counter exceeds the data type limit, the generated HDL code can produce incorrect simulation results. To accommodate the larger bit width, use a larger data type.

Introduced in R2014a

HDL FIFO

Stores sequence of input samples in first in, first out (FIFO) register (HDL Coder)

Description

The HDL FIFO block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

HDL Minimum Resource FFT

FFT— optimized for HDL code generation using minimum hardware resources (HDL Coder)

Description

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

For information about the DSP System Toolbox simulation behavior and block parameters, see HDL Minimum Resource FFT.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014b

HDL Reciprocal

Calculate reciprocal with Newton-Raphson approximation method (HDL Coder)

Description

The HDL Reciprocal block is available with Simulink.

For information about the simulation behavior and block parameters, see HDL Reciprocal block in the Simulink documentation.

The HDL Reciprocal block uses the Newton-Raphson iterative method to compute the reciprocal of the block input. The Newton-Raphson method uses linear approximation to successively find better approximations to the roots of a real-valued function.

The reciprocal of a real number a is defined as a zero of the function:

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

HDL Coder chooses an initial estimate in the range $0 < x_0 < \frac{2}{a}$ as this is the domain of convergence for the function.

To successively compute the roots of the function, specify the **Number of iterations** parameter in the Block Parameters dialog box. The process is repeated as:

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

$f'(x)$ is the derivative of the function $f(x)$.

Comparison of simulation behavior of HDL Reciprocal with Math Reciprocal block

Math Reciprocal	HDL Reciprocal
Computes the reciprocal as 1/N by using the HDL divide operator (/) to implement the division.	Uses the Newton-Rapshon iterative method to compute an approximate value of reciprocal of the block input. This approximation can yield different simulation results compared to the Math Reciprocal block. To match the simulation results with the Math Reciprocal block, increase the number of iterations for the HDL Reciprocal block. However, increasing the number of iterations increases the number of hardware resources that your design uses.

HDL Architecture

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

Architecture	Additional cycles of latency	Description
`ReciprocalNewton` (default)	`Iterations` + 1	Use the multirate implementation of 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 displays a message.

Architecture	Additional cycles of latency	Description
`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”.

Introduced in R2014b

HDL Streaming FFT

Radix-2 FFT with decimation-in-frequency (DIF) — optimized for HDL code generation (HDL Coder)

Description

The HDL Streaming FFT block will be removed in a future release. Use the FFT HDL Optimized block instead.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014b

Histogram

Frequency distribution (HDL Coder)

Description

The Histogram block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Histogram.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Histogram block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Hit Crossing

Detect crossing point (HDL Coder)

Description

The Hit Crossing block is available with Simulink.

For information about the simulation behavior and block parameters, see Hit Crossing.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restriction

The Hit crossing direction can only be `rising` or `falling`.

Introduced in R2014b

IFFT HDL Optimized

Inverse fast Fourier transform—optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see IFFT HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- If you use the IFFT HDL Optimized block with the State Control block inside an Enabled Subsystem, the optional reset port is not supported. If you enable the reset port on the IFFT HDL Optimized block in such a subsystem, the model will error on Update Diagram.

Introduced in R2014a

Image Filter

2-D FIR filtering (HDL Coder)

Description

The Image Filter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Image Filter.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Image Filter block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Image Statistics

Mean, variance, and standard deviation (HDL Coder)

Description

The Image Statistics block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Image Statistics.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015a

Increment Real World

Increase real world value of signal by one (HDL Coder)

Description

The Increment Real World block is available with Simulink.

For information about the simulation behavior and block parameters, see Increment Real World.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Increment Stored Integer

Increase stored integer value of signal by one (HDL Coder)

Description

The Increment Stored Integer block is available with Simulink.

For information about the simulation behavior and block parameters, see Increment Stored Integer.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Index Vector

Switch output between different inputs based on value of first input (HDL Coder)

Description

The Index Vector block is a Multiport Switch block with **Number of data ports** set to 1. For HDL code generation information, see Multiport Switch.

Introduced in R2014a

Inport

Create input port for subsystem or external input (HDL Coder)

Description

The Inport block is available with Simulink.

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

BidirectionalPort

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 character vector. The `IOInterface` block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid `IOInterface` settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target** > **Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
2. Save the model.

   The `IOInterface` value is saved as an HDL block property of the port.

   For example, to view the `IOInterface` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

   ```
   hdlget_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
   ```

IOInterfaceMapping

Target platform interface port mapping for DUT ports, specified as a character vector. The `IOInterfaceMapping` block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid `IOInterfaceMapping` settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target** > **Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
2. In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.
3. Save the model.

   The `IOInterfaceMapping` value is saved as an HDL block property of the port.

   For example, to view the `IOInterfaceMapping` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED',...
             'IOInterfaceMapping')
```

See Also

Topics

"Save Target Hardware Settings in Model"

Introduced in R2014a

Integer-Input RS Encoder HDL Optimized

Encode data using a Reed-Solomon encoder (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Integer-Input RS Encoder HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Integer-Output RS Decoder HDL Optimized

Decode data using a Reed-Solomon decoder (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Integer-Output RS Decoder HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- You cannot use the Integer-Output RS Decoder HDL Optimized block inside a Resettable Synchronous Subsystem.

Introduced in R2014a

Line Buffer

Store video lines and return neighborhood pixels (HDL Coder)

Description

The Line Buffer block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Line Buffer.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LMS Filter

Compute output, error, and weights using LMS adaptive algorithm (HDL Coder)

Description

The LMS Filter block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see LMS Filter.

HDL Architecture

By default, the LMS Filter implementation uses a linear sum for the FIR section of the filter.

The LMS Filter implements a tree summation (which has a shorter critical path) under the following conditions:

- The LMS Filter is used with real data.
- The word length of the Accumulator **W'u** data type is at least `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.

Introduced in R2014a

Logical Operator

Perform specified logical operation on input (HDL Coder)

Description

The Logical Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Logical Operator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Lookup Table

Map input pixel to output pixel using custom rule (HDL Coder)

Description

The Lookup Table block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Lookup Table.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2015a

LTE Convolutional Decoder

Decode convolutional-encoded samples using Viterbi algorithm (HDL Coder)

Description

The LTE Convolutional Decoder block is available with LTE HDL Toolbox™.

For information about the simulation behavior and block parameters, see Convolutional Decoder .

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LTE Convolutional Encoder

Encode binary samples using tailbiting convolutional algorithm (HDL Coder)

Description

The LTE Convolutional Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Convolutional Encoder.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LTE CRC Decoder

Detect errors in input samples using checksum (HDL Coder)

Description

The LTE CRC Decoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see CRC Decoder.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LTE CRC Encoder

Generate checksum and append to input sample stream (HDL Coder)

Description

The LTE CRC Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see CRC Encoder.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LTE Turbo Decoder

Decode turbo-encoded samples (HDL Coder)

Description

The LTE Turbo Decoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Turbo Decoder .

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

LTE Turbo Encoder

Encode binary samples using turbo algorithm (HDL Coder)

Description

The LTE Turbo Encoder block is available with LTE HDL Toolbox.

For information about the simulation behavior and block parameters, see Turbo Encoder.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2017b

M-PSK Demodulator Baseband

Demodulate PSK-modulated data (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see M-PSK Demodulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

M-PSK Modulator Baseband

Modulate using M-ary phase shift keying method (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see M-PSK Modulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Magnitude-Angle to Complex.

HDL Architecture

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

Block configuration with additional latency	Number of additional cycles
Approximation method is `CORDIC`	**Number of iterations** + 1

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

The Magnitude-Angle to Complex block supports HDL code generation when you set **Approximation method** to `CORDIC`.

Introduced in R2014a

Math Function

Perform mathematical function (HDL Coder)

Description

The Math Function block is available with Simulink.

For information about the simulation behavior and block parameters, see Math Function.

HDL Architecture

conj

Architecture	Description
`ComplexConjugate`	Compute complex conjugate. See Math Function in the Simulink documentation.

hermitian

Architecture	Description
`Hermitian`	Compute hermitian. See Math Function in the Simulink documentation.

reciprocal

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

Architecture	Parameters	Additional cycles of latency	Description
`Math` (default) `Reciprocal`	None	0	Compute reciprocal as `1/N`, using the HDL divide (/) operator to implement the division.
`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`	Signed input: (`Iterations` * 4) + 8 Unsigned input: (`Iterations` * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The default value for `Iterations` is 3.

Architecture	Parameters	Additional cycles of latency	Description
			The recommended value for `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

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

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.

Introduced in R2014a

MATLAB Function

Include MATLAB code in models that generate embeddable C code (HDL Coder)

Description

The MATLAB Function block is available with Simulink.

For information about the simulation behavior and block parameters, see MATLAB Function in Simulink documentation.

Best Practices

- “Design Guidelines for the MATLAB Function Block”
- “Generate Instantiable Code for Functions”
- “Optimize MATLAB Loops”
- “Pipeline MATLAB Expressions”

HDL Block Properties

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

InputPipeline

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

InstantiateFunctions

Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions”.

LoopOptimization

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

MapPersistentVarsToRAM

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

Complex Data Support

This block supports code generation for complex signals.

See also “Complex Data Type Support”.

Tunable Parameter Support

HDL Coder supports both tunable and non-tunable parameters with the following data types:

- Scalar
- Vector
- Complex
- Structure
- Enumeration

When using tunable parameters with the MATLAB Function block:

- The tunable parameter should be a Simulink.Parameter object with the `StorageClass` set to `ExportedGlobal`.

  ```
  x = Simulink.Parameter
  x.Value = 1
  x.CoderInfo.StorageClass = 'ExportedGlobal'
  ```

- In the Ports and Data Manager dialog box, select the **tunable** check box.

For details, see “Generate DUT Ports for Tunable Parameters”.

Restrictions

- If the block contains a System object, block inputs cannot have non-discrete (constant or `Inf`) sample time.
- HDL Coder does not support a MATLAB Function that contains the same variable as the input and output of the function. For example, this MATLAB code is not supported.

```
function y = myFun(y)
%#codegen

y = 3 * y;
```

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

- “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”

See Also

Topics

“Code Generation from a MATLAB Function Block”
“MATLAB Function Block Design Patterns for HDL”
“Distributed Pipeline Insertion for MATLAB Function Blocks”
“Generate DUT Ports for Tunable Parameters”
“HDL Applications for the MATLAB Function Block”

Introduced in R2014a

MATLAB System

Include System object in model (HDL Coder)

Description

You can define a System object and use it in a MATLAB System block for HDL code generation.

The MATLAB System block is available with Simulink.

For information about the Simulink behavior and block parameters, see MATLAB System.

Tunable Parameter Support

HDL Coder supports tunable parameters with the following data types:

- Numeric
- Fixed point
- Character
- Logical

When using tunable parameters with the MATLAB System block, the tunable parameter should be a Simulink.Parameter object with the `StorageClass` set to `ExportedGlobal`.

```
x = Simulink.Parameter
x.Value = 1
x.CoderInfo.StorageClass = 'ExportedGlobal'
```

For details, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

If you use a predefined System object, the HDL block properties available are the same as the properties available for the corresponding block.

By default, the following HDL block properties are available.

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

InputPipeline

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

LoopOptimization

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

MapPersistentVarsToRAM

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

Restrictions

- The DUT subsystem must be single-rate.
- Inputs cannot have non-discrete (constant or `Inf`) sample time.
- The following predefined System objects are supported for code generation when you use them in the MATLAB System block:
  - `hdl.RAM`
  - `comm.HDLCRCDetector`
  - `comm.HDLCRCGenerator`
  - `comm.HDLRSDecoder`
  - `comm.HDLRSEncoder`
  - `dsp.DCBlocker`
  - `dsp.HDLComplexToMagnitudeAngle`
  - `dsp.HDLFFT`
  - `dsp.HDLIFFT`
  - `dsp.HDLNCO`
- If you use a user-defined System object, it must support HDL code generation. For information about user-defined System objects and requirements for HDL code generation, see “HDL Code Generation for System Objects”.

See Also

Topics

“Generate Code for User-Defined System Objects”
“HDL Code Generation for System Objects”

Introduced in R2014a

Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal (HDL Coder)

Description

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

Introduced in R2014a

Matrix Viewer

Display matrices as color images (HDL Coder)

Description

The Matrix Viewer block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Matrix Viewer.

HDL Architecture

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

Introduced in R2014a

Maximum

Find maximum values in input or sequence of inputs (HDL Coder)

Description

The Maximum block is available with DSP System Toolbox.

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

HDL Architecture

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

Architecture	Additional cycles of latency	Description
`default` `Tree`	0	Generates a tree structure of comparators.
`Cascade`	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

InstantiateStages

Generate a VHDL `entity` or Verilog `module` for each cascade stage. The default is `off`. See also “InstantiateStages”.

SerialPartition

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

Introduced in R2014a

Measure Timing

Measure timing of pixel control bus input (HDL Coder)

Description

The Measure Timing block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Measure Timing.

HDL Architecture

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

Introduced in R2016b

Median Filter

2-D median filtering (HDL Coder)

Description

The Median Filter block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Median Filter.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Median Filter block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Memory

Output input from previous time step (HDL Coder)

Description

The Memory block is available with Simulink.

For information about the simulation behavior and block parameters, see Memory.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Message Viewer

Display message or events between blocks during simulation (HDL Coder)

Description

The Message Viewer block is available with Stateflow.

For information about 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

Introduced in R2015b

Minimum

Find minimum values in input or sequence of inputs (HDL Coder)

Description

The Minimum block is available with DSP System Toolbox.

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

HDL Architecture

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

Architecture	Additional cycles of latency	Description
`default` `Tree`	0	Generates a tree structure of comparators.
`Cascade`	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

InstantiateStages

Generate a VHDL `entity` or Verilog `module` for each cascade stage. The default is `off`. See also “InstantiateStages”.

SerialPartition

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

Introduced in R2014a

MinMax

Output minimum or maximum input value (HDL Coder)

Description

The MinMax block is available with Simulink.

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

HDL Architecture

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

Architecture	Additional cycles of latency	Description
`default` `Tree`	0	Generates a tree structure of comparators.
`Cascade`	1, when block has a single vector input port.	This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices”.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

InstantiateStages

Generate a VHDL `entity` or Verilog `module` for each cascade stage. The default is `off`. See also “InstantiateStages”.

SerialPartition

Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition”.

Native Floating Point

LatencyStrategy

To enable this setting, use `Tree` as the **HDL Architecture**. With this setting, you can specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Introduced in R2014a

Model

Include model as block in another model (HDL Coder)

Description

The Model block is available with Simulink. For information about the simulation behavior and block parameters, see Model.

Generate Comments

If you enter text in the Model Block Properties dialog box **Description** field, HDL Coder generates a comment in the HDL code.

Generate Code For Model Arguments

To generate a single Verilog `module` or VHDL `entity` for instances of a referenced model with different model argument values, see “Generate Parameterized Code for Referenced Models”.

HDL Architecture

Architecture	Description
`ModelReference` (default)	When you want to generate code from a referenced model and any nested models, use the `ModelReference` implementation. For more information, see “How To Generate Code for a Referenced Model”.

Architecture	Description
`BlackBox`	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

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also "BalanceDelays".

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also "DistributedPipelining".

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

InputPipeline

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

OutputPipeline

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

ReferenceModelPrefix

Prefix of the referenced model to insert in the generated code. The code generator applies this prefix to submodel file names and HDL identifiers. The default prefix is `modelname_` where `modelname` is the name of the referenced model.

Note

- If you specify an empty prefix, the code generator does not add a prefix to submodel file names. This can cause HDL compilation errors due to naming collisions between the models.
- If you use the referenced model as the DUT, the code generator ignores the prefix that you specify.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Restrictions

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.

When you have model references and generate HDL code, the generated model, validation model, and cosimulation model can fail to compile or simulate. To fix comilation or simulation errors, make sure that the referenced models are loaded or are on the search path.

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

- Distributed pipelining
- Constrained output pipelining
- Streaming

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.

See Also

Topics

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

Introduced in R2014a

Model Info

Display model properties and text in model (HDL Coder)

Description

The Model Info block is available with Simulink.

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

Best Practices

When using Model Info blocks in models targeted for HDL code generation, consider using only ASCII characters in the text that you enter to display on the Model Info block. If you have non-ASCII characters in the generated HDL code, RTL simulation and synthesis tools can fail to compile the code.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Model Variants

Include model as block in another model (HDL Coder)

Description

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

Introduced in R2014a

Multiply-Accumulate

Perform a multiply-accumulate operation on the inputs (HDL Coder)

Description

The Multiply-Accumulate block is available in the HDL Operations sublibrary in the HDL Coder block library. To compute the result `y`, the block adds the dot product of the input vectors `u1` and `u2` to the bias `k`.

```
y = sum(u1.* u2) + k
```

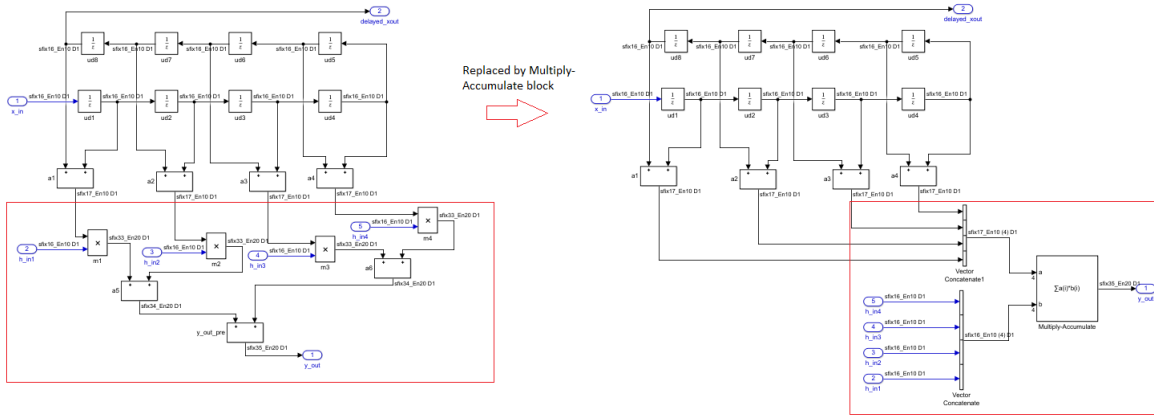
By default, the bias `k` is equal to zero, and the block computes the dot product of the inputs `u1` and `u2`. To learn about the simulation behavior and block parameters, see Multiply-Accumulate.

When you synthesize your design, the generated HDL code for the block maps efficiently to DSP slices on the FPGA. For DSP mapping, before you generate code, set the **Reset type** model parameter to `Synchronous` when you use a Xilinx device and `Asynchronous` when you use an Intel® device.

Benefits

With the Multiply-Accumulate block, you can:

- Perform matrix multiplication operations. For example, if you have two matrix inputs with dimensions `N-by-M` and `M-by-P`, you can compute the result by using `N-by-P` multiply-accumulate operations in parallel.
- Replace a sequence of multiplication and addition operations, such as in filter blocks, and improve the performance on hardware by mapping to DSP slices on the FPGA. This figure shows how you can use the Multiply-Accumulate block with the `sfir_fixed` model.



HDL Architecture

Default: `Auto`

`Auto`

This mode selects the `Serial` architecture by default. When the block is inside a feedback loop, the code generator cannot use the `Serial` architecture if the block is not part of a clock-rate pipelining region and does not have a Delay at the block output. This error occurs because the `Serial` architecture introduces additional latency which cannot be delay balanced inside the feedback loop. When you use the `Auto` mode, the code generator switches to the `Parallel` architecture automatically.

`Parallel`

For input vectors of size `N`, this mode uses `N` Multiply-Add blocks in series to compute the result. This mode uses a combinatorial implementation and does not introduce any latency. If you specify the **Synthesis tool** and **Target frequency**, since the adaptive pipelining optimization is enabled, the code generator inserts pipeline registers for the Multiply-Add blocks. When you synthesize your design, depending on the input bit widths, this architecture maps up to `N` DSP slices on the FPGA.

`Serial`

For input vectors of size `N`, this mode uses a streaming algorithm to implement the multiply-accumulate operation. This architecture has two implementation modes:

- The default mode uses a local multirate implementation. This implementation overclocks the shared resources by `N` and multiplexes the input vectors with a Multiply-Add block. This implementation introduces an additional latency of one at the data rate.
- If you have clock-rate pipelining enabled on the model or subysystem that contains the Multiply-Accumulate block, this architecture uses a single-rate implementation. This implementation runs the shared resources at the clock-rate and multiplexes the input vectors with a Multiply-Add block. This implementation introduces an additional latency of `N` at the clock rate.

When you synthesize your design, depending on the input bit widths, this architecture maps to one DSP slice on the FPGA.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

When you use complex signals, this block can generate HDL code, but does not map to DSP slices.

This block supports code generation for complex signals.

Restrictions

- Scalar inputs are not supported for HDL code generation. To generate code for the block, use vector inputs. With scalar inputs, you can use the Multiply-Add block.
- Matrix data types are not supported at the Simulink block interfaces.

See Also

Dot Product | Multiply-Add

Topics

"Adaptive Pipelining"
"Clock-Rate Pipelining"

Introduced in R2017b

Multiply-Add

Multiply-add combined operation for HDL Coder

Description

The Multiply-Add block is available in the HDL Operations sublibrary in the HDL Coder block library. For information about the simulation behavior and block parameters, see Multiply-Add

Hardware Mapping

To map a combined multiply and add or a multiply and subtract operation to a DSP unit in your target hardware, select the **Function** setting in the Block Parameters dialog box for the Multiply-Add block.

To map to a DSP unit, specify the SynthesisTool property for your model.

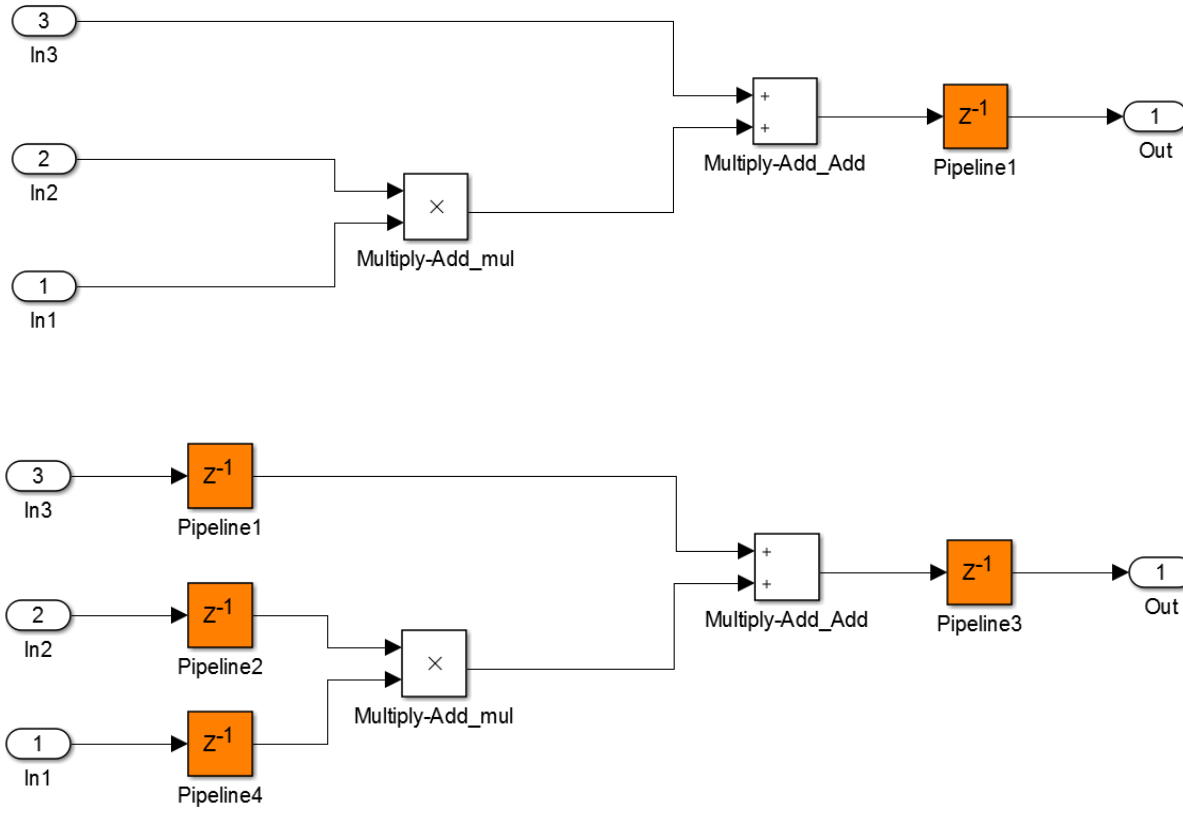
When you generate HDL code for your model, HDL Coder configures the multiply-add operation so that your synthesis tool can map to a DSP unit.

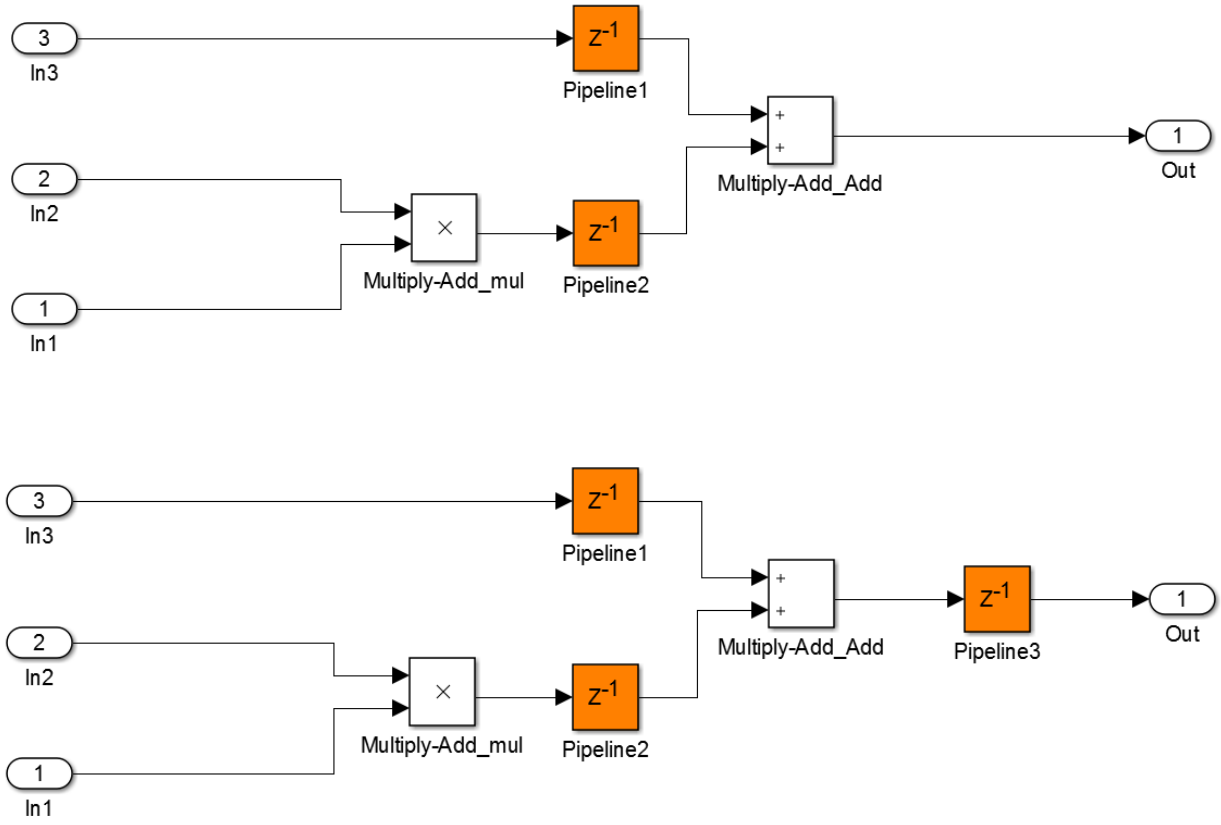
Note Some DSP units do not have the multiply-add capability. To see if your hardware has the multiply-add capability, refer to the documentation for the hardware.

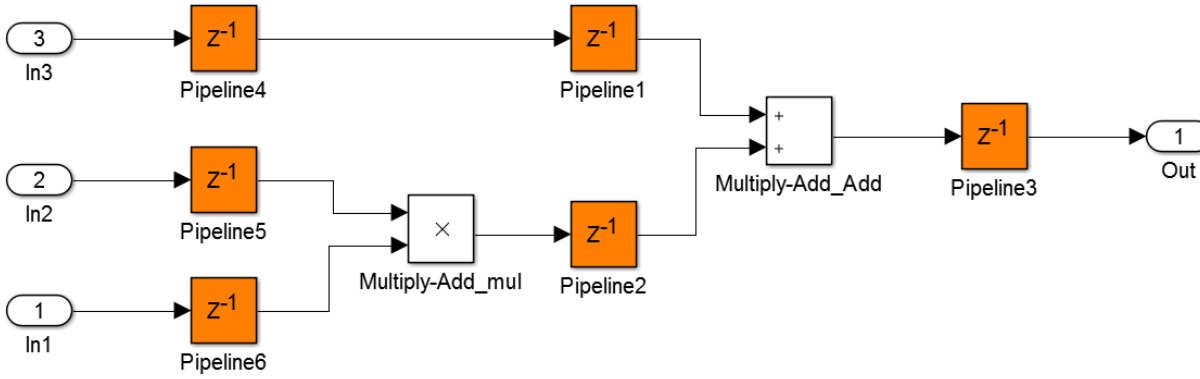
Pipeline Depth

If you have fixed-point inputs to a Multiply-Add block, you can set the **PipelineDepth** for the block. For floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert the pipeline registers.

The following diagrams show different configurations of pipeline registers for different synthesis tools and **PipelineDepth** settings. When you specify the **PipelineDepth** setting, HDL Coder inserts pipeline registers so that the configuration maps efficiently to DSP units.


3
In3
2
In2
1
In1
Multiply-Add_mul
Multiply-Add_Add
Pipeline1
1
Out
3
In3
Pipeline1
2
In2
Pipeline2
1
In1
Pipeline4
Multiply-Add_mul
Multiply-Add_Add
Pipeline3
1
Out


3
In3
Pipeline1
2
In2
Multiply-Add_mul
Pipeline2
Multiply-Add_Add
1
Out
1
In1
3
In3
Pipeline1
2
In2
Multiply-Add_mul
Pipeline2
Multiply-Add_Add
Pipeline3
1
Out
1
In1



HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

PipelineDepth

Number of pipeline stages. The default is `auto` which means that the coder determines the number of pipeline stages based on your synthesis tool.

You can enter an integer between 0 and 3. For Altera hardware targets, the maximum pipeline depth is 2.

Complex Data Support

This block supports code generation for complex signals.

Restrictions

- When the block has floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert pipeline registers.
- If the block is in a feedback loop and you do not have sufficient delays at the block output, the coder reduces the **PipelineDepth** to prevent delay balancing failure. For sufficient delays, add Delay blocks at the output of the Multiply-Add block.
- To map the combined multiply-add operation to a DSP unit, the width of the third input *c* has to be less than 64 bits for Altera and 48 bits for Xilinx respectively.
- The subtraction operation in the **Function** setting `(a.*b)-c` does not map to a DSP unit in Altera FPGA libraries.

See Also

Multiply-Add | SynthesisTool

Introduced in R2015b

Multiport Selector

Distribute arbitrary subsets of input rows or columns to multiple output ports (HDL Coder)

Description

The Multiport Selector block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Multiport Selector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Multiport Switch

Choose between multiple block inputs (HDL Coder)

Description

The Multiport Switch block is available with Simulink.

For information about the simulation behavior and block parameters, see Multiport Switch.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

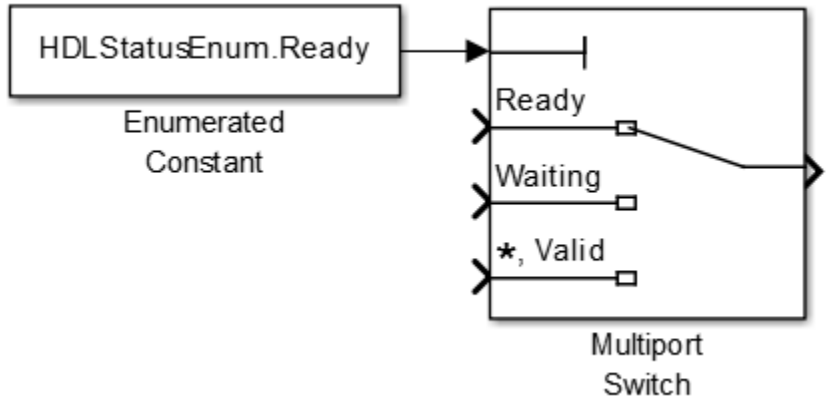
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

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.



Introduced in R2014a

Mux

Combine several input signals into vector (HDL Coder)

Description

The Mux block is available with Simulink.

For information about the simulation behavior and block parameters, see Mux.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Restrictions

Buses are not supported for HDL code generation.

Introduced in R2014a

n-D Lookup Table

Approximate N-dimensional function (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see n-D Lookup Table.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Restrictions

Required Block Settings

- **Number of table dimensions**: HDL Coder supports a maximum dimension of 2.
- **Breakpoints specification**: You can select either `Explicit values` or `Even spacing`.
- **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.

Introduced in R2014a

NCO

Generate real or complex sinusoidal signals (HDL Coder)

Description

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

Introduced in R2014a

NCO HDL Optimized

Generate real or complex sinusoidal signals—optimized for HDL code generation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see NCO HDL Optimized.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

LUTRegisterResetType

The reset type of the lookup table output register. Select `none` to synthesize the lookup table to a ROM when your target is an FPGA. See also “LUTRegisterResetType”.

OutputPipeline

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

Restrictions

- When you set **Dither source** to `Property`, the block adds random dither every cycle. If you generate a validation model with these settings, a warning is displayed. Random generation of the internal dither can cause mismatches between the models. You can increase the error margin for the validation comparison to account for the difference. You can also disable dither or set **Dither source** to `Input port` to avoid this issue.
- You cannot use the NCO HDL Optimized block inside a Resettable Synchronous Subsystem.

Introduced in R2014a

Opening

Morphological open of binary pixel data (HDL Coder)

Description

The Opening block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Opening.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Opening block inside a Resettable Synchronous Subsystem.

Introduced in R2015a

Outport

Create output port for subsystem or external output (HDL Coder)

Description

The Outport block is available with Simulink.

For information about 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 character vector. The `IOInterface` block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid `IOInterface` settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
2. Save the model.

   The `IOInterface` value is saved as an HDL block property of the port.

   For example, to view the `IOInterface` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

   ```
   hdlget_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
   ```

IOInterfaceMapping

Target platform interface port mapping for DUT ports, specified as a character vector. The `IOInterfaceMapping` block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid `IOInterfaceMapping` settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.
2. In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.
3. Save the model.

   The `IOInterfaceMapping` value is saved as an HDL block property of the port.

   For example, to view the `IOInterfaceMapping` value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED',...
             'IOInterfaceMapping')
```

See Also

Topics

“Save Target Hardware Settings in Model”

Introduced in R2014a

Pixel Stream Aligner

Align two streams of pixel data (HDL Coder)

Description

The Pixel Stream Aligner block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see Pixel Stream Aligner.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

You cannot use the Pixel Stream Aligner block inside a Resettable Synchronous Subsystem.

Introduced in R2017a

PN Sequence Generator

Generate pseudonoise sequence (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see PN Sequence Generator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- You can select `Input port` as the **Output mask source** on the block. However, in this case, the `Mask` input signal must be a vector of data type `ufix1`.
- If you select **Reset on nonzero input**, the input to the `Rst` port must have data type `Boolean`.
- Outputs of type `double` are not supported for HDL code generation. All other output types (including bit packed outputs) are supported.
- You cannot use the PN Sequence Generator block inside a Resettable Synchronous Subsystem.

Introduced in R2014a

Prelookup

Compute index and fraction for Interpolation Using Prelookup block (HDL Coder)

Description

The Prelookup block is available with Simulink.

For information about the simulation behavior and block parameters, see Prelookup.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- “Required Block Settings” on page 3-324

- “Table Data Typing and Sizing” on page 3-324

Required Block Settings

- **Breakpoint data**: For **Source**, select `Dialog`.
- **Specification**: You can select either `Explicit values` or `Even spacing`.
- **Index search method**: Select `Evenly spaced points`.
- **Extrapolation method**: Select `Clip`.
- **Diagnostic for out-of-range input**: Select `Error`.
- **Use last breakpoint for input at or above upper limit**: Select this check box.
- **Breakpoint**: For **Data Type**, select `Inherit: Same as input`.
- **Integer rounding mode**: Select `Zero`, `Floor`, or `Simplest`.

Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. If the breakpoint spacing does not meet this condition, HDL Coder issues a warning. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- All ports on the block require scalar values.
- The coder permits floating-point data for breakpoints.

Introduced in R2014a

Product

Multiply and divide scalars and nonscalars or multiply and invert matrices (HDL Coder)

Description

The Product block is available with Simulink.

For information about the simulation behavior and block parameters, see Product.

Divide or Reciprocal

For block implementations of the Product block in divide mode or reciprocal mode, see Divide.

Note In divide mode, **Number of Inputs** is set to `*/`.

In reciprocal mode, **Number of Inputs** is set to `/`.

HDL Architecture

The default `Linear` implementation generates a chain of N operations (multipliers) for N inputs.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also "Denormal Numbers".

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also "Latency Considerations with Native Floating Point".

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also "Mantissa Multiplier Strategy".

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.

Introduced in R2014a

Product of Elements

Copy or invert one scalar input, or collapse one nonscalar input (HDL Coder)

Description

The Product of Elements block is available with Simulink.

For information about the simulation behavior and block parameters, see Product of Elements.

HDL Architecture

HDL Coder supports `Tree` and `Cascade` architectures for Product or Product of Elements blocks that have a single vector input with multiple elements.

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

Architecture	Additional cycles of latency	Description
`Linear` (default)	0	Generates a linear chain of adders to compute the sum of products.
`Tree`	0	Generates a tree structure of adders to compute the sum of products.
`Cascade`	1, when block has a single vector input port.	This implementation optimizes latency * area and is faster than the `Tree` implementation. It computes partial products and cascades multipliers. See “Cascade Architecture Best Practices”.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

MantissaMultiplyStrategy

Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also “Mantissa Multiplier Strategy”.

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.

Introduced in R2014a

QPSK Demodulator Baseband

Demodulate QPSK-modulated data (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see QPSK Demodulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

QPSK Modulator Baseband

Modulate using quaternary phase shift keying method (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see QPSK Modulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Raised Cosine Receive Filter

Apply pulse shaping by downsampling signal using raised cosine FIR filter (HDL Coder)

Description

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

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

This block is a subsystem that contains a FIR Decimation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See Subsystem and FIR Decimation for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Receive Filter block and execute this command.

```
set_param(gcb,'LinkStatus','inactive')
```

Introduced in R2015a

Raised Cosine Transmit Filter

Apply pulse shaping by upsampling signal using raised cosine FIR filter (HDL Coder)

Description

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

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

This block is a subsystem that contains a FIR Interpolation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See Subsystem and FIR Interpolation for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Transmit Filter block and execute this command.

```
set_param(gcb,'LinkStatus','inactive')
```

Introduced in R2015a

Rate Transition

Handle transfer of data between blocks operating at different rates (HDL Coder)

Description

The Rate Transition block is available with Simulink.

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

Best Practices

When the Rate Transition block is operating at a faster input rate and slower output rate, it is good practice to follow the Rate Transition block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation”.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- Sample rate cannot be 0 or `Inf` for block input or output ports.
- **Ensure data integrity during data transfer** must be enabled.
- **Ensure deterministic data transfer (maximum delay)** must be enabled.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Real-Imag to Complex.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Reciprocal Sqrt

Calculate square root, signed square root, or reciprocal of square root (HDL Coder)

Description

The Reciprocal Sqrt block is available with Simulink.

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

HDL Code Generation Support

For the Sqrt block with **Function** set to `rSqrt`, the code generator supports various architectures and data types. The `sqrtfunction` architecture supports code generation in native floating-point mode. For this architecture, you can specify the **HandleDenormals** and **LatencyStrategy** settings from the **Native Floating Point** tab in the HDL Block Properties dialog box.

Architecture	Fixed-Point	Native Floating-Point	HandleDenormals	LatencyStrategy
`sqrtfunction`	—	✓	✓	✓
`recipsqrtnewton`	✓	—	—	—
`recipsqrtnewtonsinglerate`	✓	—	—	—

HDL Architecture

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

Architecture	Additional cycles of latency	Description
`SqrtFunction` (default)	0	Use a bitset shift/addition algorithm. The `SqrtFunction` architecture is equivalent to the `SqrtBitset` architecture with `UseMultiplier` set to `off`.
`RecipSqrtNewton`	`Iterations` + 2	Use the iterative Newton method. Select this option to optimize area.
`RecipSqrtNewtonSingleRate`	(`Iterations` * 4) + 5	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

Iterations

Number of iterations for Newton method. The default is 3.

OutputPipeline

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

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Restrictions

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

Introduced in R2014a

Rectangular QAM Demodulator Baseband

Demodulate rectangular-QAM-modulated data (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Rectangular QAM Demodulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- The block does not support single or double data types for HDL code generation.
- HDL Coder supports the following **Output type** options:
  - `Integer`
  - `Bit` is supported only if the **Decision Type** that you select is `Hard decision`.
- The coder requires that you set **Normalization Method** to `Minimum Distance Between Symbols`, with a **Minimum distance** of 2.
- The coder requires that you set **Phase offset (rad)** to a value that is a multiple of `pi`/4.

Introduced in R2014a

Rectangular QAM Modulator Baseband

Modulate using rectangular quadrature amplitude modulation (HDL Coder)

Description

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

For information about the simulation behavior and block parameters, see Rectangular QAM Modulator Baseband.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

- The block does not support single or double data types for HDL code generation.
- When **Input Type** is set to `Bit`, the block does not support HDL code generation for input types other than `boolean` or `ufix1`.

When the input type is set to `Bit`, but the block input is actually multibit (`uint16`, for example), the Rectangular QAM Modulator Baseband block does not support HDL code generation.

Introduced in R2014a

Relational Operator

Perform specified relational operation on inputs (HDL Coder)

Description

The Relational Operator block is available with Simulink.

For information about the simulation behavior and block parameters, see Relational Operator.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Complex Data Support

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

Introduced in R2014a

Relay

Switch output between two constants (HDL Coder)

Description

The Relay block is available with Simulink.

For information about the simulation behavior and block parameters, see Relay.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Repeat

Resample input at higher rate by repeating values (HDL Coder)

Description

The Repeat block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Repeat.

Best Practices

The Repeat block uses fewer hardware resources than the Upsample block. If your algorithm does not require zero-padding upsampling, use the Repeat block.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Restrictions

Input processing set to `Columns as channels (frame based)` is not supported.

Introduced in R2014a

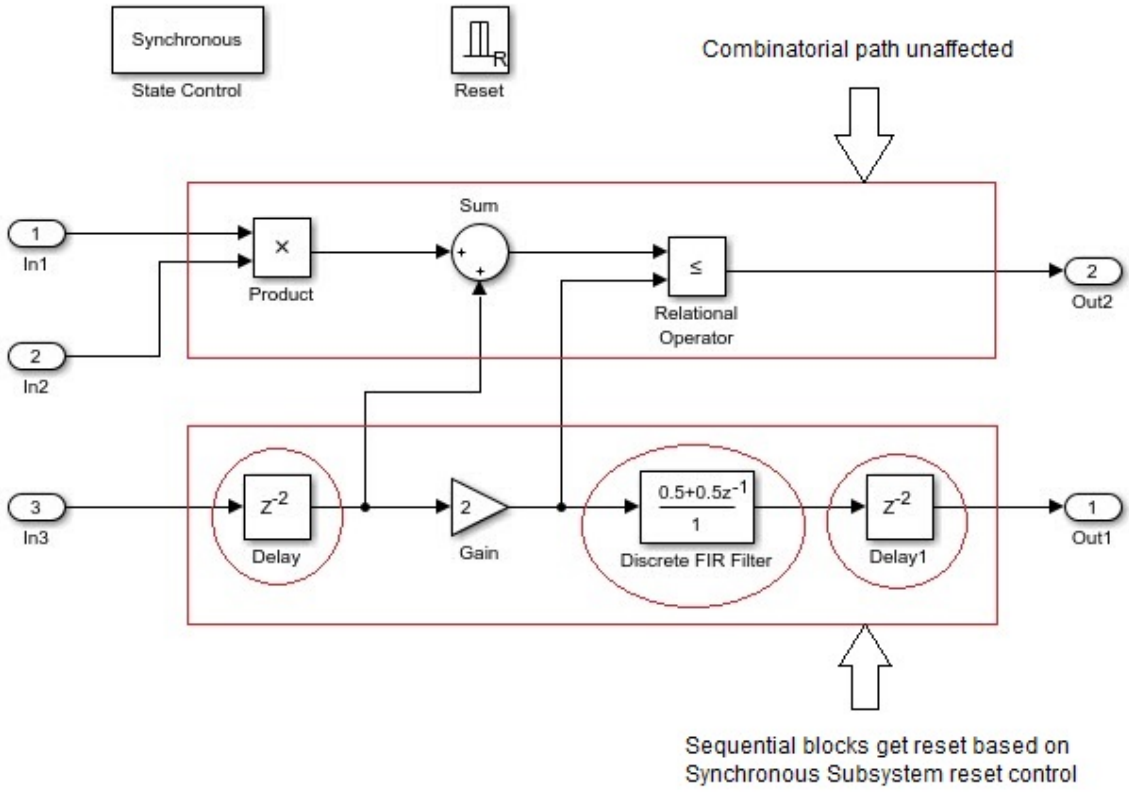
Resettable Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior (HDL Coder)

Description

The Resettable Synchronous Subsystem block is available in the HDL Subsystems block library in HDL Coder. For information about the simulation behavior and block parameters, see Resettable Synchronous Subsystem.

The Resettable Synchronous Subsystem uses the State Control block in **Synchronous** mode with the Resettable Subsystem block. For subsystem blocks with state, the State Control block in **Synchronous** mode provides efficient reset and enable simulation behavior on hardware.



The reset port in the Resettable Synchronous Subsystem block adds reset capability to blocks inside the subsystem that have state. This includes blocks that need not have an external reset port capability, such as filters, Stateflow Chart, and MATLAB Function blocks. For HDL code generation, the **Reset trigger type** of the Reset port is set to `level hold` by default.

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. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a "no-op" in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See "Customize Black Box or HDL Cosimulation Interface".

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also "BalanceDelays".

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also "DistributedPipelining".

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also "FlattenHierarchy".

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also "Resource Sharing".

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also "Streaming".

Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

You can set this property In the HDL Workflow Advisor, 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 (;).

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

Values: `''` (default) | character vector

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

IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: `''` (default) | character vector

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: `''` (default) | character vector

Example: `'1.3'`

Restrictions

- You cannot use the State Control block in **Classic** mode or remove the State Control block from the Resettable Synchronous Subsystem block.
- The **Reset trigger type** of the Reset port inside the subsystem must be set to `level hold`.
- A Delay block with nonvirtual bus input signals inside a Resettable Synchronous Subsystem is not supported if you enable optimizations on the subsystem.
- HDL Coder does not support these blocks inside a Resettable Synchronous Subsystem:
  - All RAM blocks or blocks that infer a RAM in the generated HDL code. The RAM blocks include:
    - Single Port RAM
    - Simple Dual Port RAM
    - Dual Port RAM
    - Dual Rate Dual Port RAM
    - HDL FIFO
    - `hdl.RAM` system object

DSP System Toolbox

- Biquad Filter
- NCO HDL Optimized

Communications System Toolbox

- Convolutional Encoder
- Viterbi Decoder
- PN Sequence Generator
- Integer-Output RS Decoder HDL Optimized

Vision HDL Toolbox

- Demosaic Interpolator
- Edge Detector

- Histogram
- Image Filter
- Median Filter
- Binary and Grayscale Morphology blocks

See Also

Enable | Enabled Synchronous Subsystem | State Control | Synchronous Subsystem

Topics

"Resettable Subsystem Support in HDL Coder™"
"Using the State Control block to generate more efficient code with HDL Coder™"
"Synchronous Subsystem Behavior with the State Control Block"

Introduced in R2016b

Reshape

Change dimensionality of signal (HDL Coder)

Description

The Reshape block is available with Simulink.

For information about the simulation behavior and block parameters, see Reshape.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

ROI Selector

Select a region of interest (ROI) from a pixel stream (HDL Coder)

Description

The ROI Selector block is available with Vision HDL Toolbox.

For information about the simulation behavior and block parameters, see ROI Selector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2016a

Rounding Function

Apply rounding function to signal (HDL Coder)

Description

The Rounding Function block is available with Simulink. For information about the simulation behavior and block parameters, see Rounding Function.

To generate HDL code, use single data types as inputs to the block, and specify the native floating point mode. In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Floating Point Target** tab, for **Library**, select `Native Floating Point`.

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

Topics

“Getting Started with HDL Coder Native Floating-Point Support”
“Generate Target-Independent HDL Code with Native Floating-Point”

Introduced in R2017a

Sample and Hold

Sample and hold input signal (HDL Coder)

Description

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

For information about the DSP System Toolbox simulation behavior and block parameters, see Sample and Hold.

HDL code for the Sample and Hold block is generated as a Triggered Subsystem. Similar restrictions apply to both blocks.

HDL Block Properties

For HDL block property descriptions, see “HDL Block Properties”.

Best Practices

When using the Sample and Hold block in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put a unit delay on the output signal. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems, such as the Sample and Hold block, can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances.

Restrictions

The Sample and Hold block must meet the following conditions:

- The DUT (i.e., the top-level subsystem for which code is generated) must not be the Sample and Hold block.
- The trigger signal must be a scalar.
- The data type of the trigger signal must be either `boolean` or `ufix1`.
- The output of the Sample and Hold block must have an initial value of 0.
- The input, output, and trigger signal of the Sample and Hold block must run at the same rate. If one of the input or the trigger signals is an output of a Signal Builder block, see “Using the Signal Builder Block” on page 3-429 for how to match rates.

Introduced in R2014b

Saturation

Limit range of signal (HDL Coder)

Description

The Saturation block is available with Simulink.

For information about the simulation behavior and block parameters, see Saturation.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Saturation Dynamic

Bound range of input (HDL Coder)

Description

The Saturation Dynamic block is available with Simulink.

For information about the simulation behavior and block parameters, see Saturation Dynamic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Scope

Display signals generated during simulation (HDL Coder)

Description

The Scope block is available with Simulink.

For information about the simulation behavior and block parameters, see Scope.

HDL Architecture

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

Introduced in R2014a

Selector

Select input elements from vector, matrix, or multidimensional signal (HDL Coder)

Description

The Selector block is available with Simulink.

For information about the simulation behavior and block parameters, see Selector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Serializer1D

Convert vector signal to scalar or smaller vectors (HDL Coder)

Description

The Serializer1D block is available with Simulink.

For information about 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”.

Introduced in R2014b

Shift Arithmetic

Shift bits or binary point of signal (HDL Coder)

Description

The Shift Arithmetic block is available with Simulink.

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

You can generate HDL code when **Bits to shift: Source** is **Dialog** or **Input port**.

HDL Architecture

The generated VHDL code uses the `shift_right` function and `sll` operator.

The generated Verilog code uses the >>> and <<< shift operators.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Restrictions

When **Bits to shift: Source** is **Input port**, binary point shifting is not supported.

Introduced in R2014a

Sign

Indicate sign of input (HDL Coder)

Description

The Sign block is available with Simulink.

For information about the simulation behavior and block parameters, see Sign.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Introduced in R2014a

Signal Conversion

Convert signal to new type without altering signal values (HDL Coder)

Description

The Signal Conversion block is available with Simulink.

For information about the simulation behavior and block parameters, see Signal Conversion.

HDL Architecture

This block has a pass-through implementation.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal (HDL Coder)

Description

The Signal Specification block is available with Simulink.

For information about the simulation behavior and block parameters, see Signal Specification.

HDL Architecture

This block has a pass-through implementation.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Simple Dual Port RAM

Dual port RAM with single output port (HDL Coder)

Description

The Simple Dual Port RAM block is available with Simulink.

For information about 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 3-378.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Simple Dual Port RAM System

Dual port RAM with single output port and ability to specify initial value (HDL Coder)

Description

The Simple Dual Port RAM System block is available in the HDL RAMs sublibrary in the HDL Coder block library. For information about the simulation behavior and block parameters, see Dual Port RAM System, Simple Dual Port RAM System, Single Port RAM System.

The Simple Dual Port RAM System block is a MATLAB System block that uses the `hdl.RAM` System object. The simulation and HDL code generation behavior of the block is similar to the Simple Dual Port RAM block. In addition, you can:

- Specify an initial value for the RAM. Double-click the block to open the Block Parameters dialog box, and then enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.
- Obtain higher performance and support for large data memories.

Note When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents.To reset the RAM, make sure that you implement the reset logic.

HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM` System object.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

See Also

System Objects

`hdl.RAM`

Blocks

Dual Port RAM System | Single Port RAM System

Topics

“HDL Code Generation from hdl.RAM System Object”
“Getting Started with RAM and ROM in Simulink®”
“Implement RAM Using MATLAB Code”
“HDL Code Generation for System Objects”

Introduced in R2014a

Sine

Implement fixed-point sine wave using lookup table approach that exploits quarter wave symmetry (HDL Coder)

Description

The Sine block is available with Simulink.

For information about the simulation behavior and block parameters, see Sine, Cosine.

HDL Architecture

The HDL code implements Sine using the quarter-wave lookup table you specify in the Simulink block parameters.

To avoid generating a division operator (/) in the HDL code, for **Number of data points for lookup table**, enter `(2^n)+1`. *n* is an integer.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

See Also

Cosine | Cosine HDL Optimized | Sine HDL Optimized

Introduced in R2014a

Sine HDL Optimized

Implement fixed-point sine wave optimized for HDL code generation

Description

The Sine HDL Optimized block is available in the Lookup Tables library in HDL Coder. For information about the simulation behavior and block parameters, see Sine HDL Optimized.

For the most efficient HDL implementation, configure the block with an exact power of two as the number of elements. In the Block Parameters dialog box, for **Number of data points**, specify an integer that is an exact power of two. By default, the **Number of data points** is 64.

When you specify a power of two for the **Number of data points**, the lookup tables precede a register without reset after HDL code generation. The combination of the lookup table block and register without reset map efficiently to RAM on the target device.

HDL Architecture

The HDL code implements the Sine HDL Optimized block by using the quarter-wave lookup table that you specify in the Simulink block parameters.

To generate code that is optimized for area and speed, for **Number of data points**, enter `(2^n)`. *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”.

See Also

Cosine | Cosine HDL Optimized | Sine

Introduced in R2016b

Sine Wave

Generate continuous or discrete sine wave (HDL Coder)

Description

The Sine Wave block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Sine Wave.

HDL Architecture

This block has a single, default HDL architecture.

Restrictions

For HDL code generation, you must select the following Sine Wave block settings:

- **Computation method**: `Table lookup`
- **Sample mode**: `Discrete`

Output:

- The output port cannot have data types `single` or `double`.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Single Port RAM

Single port RAM (HDL Coder)

Description

The Single Port RAM block is available with Simulink.

For information about 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 3-388.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

Number of output pipeline 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.

Introduced in R2014a

Single Port RAM System

Single port RAM that can read from and write to memory locations with ability to specify initial value (HDL Coder)

Description

The Single Port RAM System block is available in the HDL RAMs sublibrary in the HDL Coder block library. For information about the simulation behavior and block parameters, see Dual Port RAM System, Simple Dual Port RAM System, Single Port RAM System.

The Single Port RAM System block is a MATLAB System block that uses the `hdl.RAM` System object. The simulation and HDL code generation behavior of the block is similar to the Single Port RAM. In addition, you can:

- Specify an initial value for the RAM. Double-click the block to open the Block Parameters dialog box, and then enter a value for **Specify the RAM initial value**.
- Obtain faster simulation results when you use these blocks in your Simulink model.
- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.
- Obtain higher performance and support for large data memories.

Note When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents.To reset the RAM, make sure that you implement the reset logic.

HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM` System object.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

See Also

System Objects

`hdl.RAM`

Blocks

Dual Port RAM System | Simple Dual Port RAM

Topics

“HDL Code Generation from hdl.RAM System Object”
“Getting Started with RAM and ROM in Simulink®”
“Implement RAM Using MATLAB Code”
“HDL Code Generation for System Objects”

Introduced in R2017b

Spectrum Analyzer

Display frequency spectrum of time-domain signals (HDL Coder)

Description

The Spectrum Analyzer block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Spectrum Analyzer.

HDL Architecture

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

Introduced in R2014a

Sqrt

Calculate square root, signed square root, or reciprocal of square root (HDL Coder)

Description

The Sqrt block is available with Simulink.

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

HDL Code Generation Support

For the Sqrt block with **Function** set to `sqrt`, the code generator supports various architectures and data types. The `sqrtfunction` architecture supports code generation in native floating-point mode. For this architecture, you can specify the **HandleDenormals** and **LatencyStrategy** settings from the **Native Floating Point** tab in the HDL Block Properties dialog box.

Architecture	Fixed-Point	Native Floating-Point	HandleDenormals	LatencyStrategy
`sqrtfunction`	✓	✓	✓	✓
`sqrtnewton`	✓	—	—	—
`sqrtnewtonsinglerate`	✓	—	—	—
`sqrtbitset`	✓	—	—	—

HDL Architecture

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

Architecture	Parameter	Additional cycles of latency	Description
`SqrtFunction` (default)	None	0	Use a bitset shift/addition algorithm. The `SqrtFunction` architecture is equivalent to the `SqrtBitset` architecture with `UseMultiplier` set to `off`.
`SqrtBitset`	`UseMultiplier`	0	Algorithm depends on the `UseMultiplier` setting: • `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. The default value for `Iterations` is 3. The recommended value for `Iterations` is from 2 through 10. If `Iterations` is outside the recommended range, HDL Coder generates a message.
`SqrtNewtonSingleRate`	`Iterations`	(`Iterations` * 4) + 6	Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The default value for `Iterations` is 3. The recommended value for `Iterations` is from 2 through 10. If `Iterations` is outside the recommended range, the coder generates a message.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

Iterations

Number of iterations for `SqrtNewton` or `SqrtNewtonSingleRate` implementation.

InputPipeline

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

OutputPipeline

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

UseMultiplier

Select algorithm for `SqrtBitset` implementation. The default is `off`.

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Restrictions

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

Introduced in R2014a

State Control

Properties and restrictions for HDL code generation

Description

The State Control block is available in the HDL Subsystems Block Library. For information about block parameters and simulation behavior, see State Control in the Simulink documentation.

Use the State Control block to toggle subsystem behavior between the default Simulink simulation behavior and the synchronous hardware simulation behavior.

- For default Simulink simulation behavior, set **State control** to `Classic`.
- For synchronous hardware simulation behavior, set **State control** to `Synchronous`.

The `Synchronous` mode of the State Control block:

- Provides efficient reset and enable simulation behavior on hardware.
- Generates cleaner HDL code and uses fewer hardware resources.

See “Synchronous Subsystem Behavior with the State Control Block”.

HDL Architecture

This block has a single, default HDL architecture. HDL Coder does not generate HDL code specific to the State Control block. How you set the State Control block affects other blocks inside the subsystem that have state.

Restrictions

For the State Control block in `Synchronous` mode-

Supported block modes:

- Delay block: When you have an external reset port, set the **External reset** to `Level hold`.
- Enabled Subsystem: You cannot use the optional reset port for blocks inside the subsystem.
- Stateflow Chart: Set the **State Machine Type** to `Moore`.
- MATLAB Function block:
  - You cannot have System Objects inside the MATLAB Function block.
  - If you use nondirect feedthrough in a MATLAB Function block, do not program the outputs to rely on inputs or updated persistent variables. The MATLAB Function block must drive the outputs from persistent variables.

    To use nondirect feedthrough, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box. See "Use Nondirect Feedthrough in a MATLAB Function Block" (Simulink).

Unsupported blocks:

- Triggered Subsystem
- LMS Filter
- HDL Minimum Resource FFT
- DC Blocker
- PN Sequence Generator
- Convolutional Interleaver and Convolutional Deinterleaver
- General Multiplexed Interleaver and General Multiplexed Deinterleaver
- Convolutional Encoder and Viterbi Decoder
- Sample and Hold

See Also

Enable | Enabled Synchronous Subsystem | Resettable Synchronous Subsystem

Topics

"Resettable Subsystem Support in HDL Coder™"
"Using the State Control block to generate more efficient code with HDL Coder™"
"Synchronous Subsystem Behavior with the State Control Block"

Introduced in R2015a

State Transition Table

Represent modal logic in tabular format (HDL Coder)

Description

The State Transition Table block is available with Stateflow.

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

Tunable Parameters

You can use a tunable parameter in a State Transition Table intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

This block has a single, default HDL architecture.

Active State Output

To generate an output port in the HDL code that shows the active state, select **Create output port for monitoring** in the Properties window of the chart. The output is an enumerated data type. See “Use Active State Output Data” (Stateflow).

HDL Block Properties

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

InputPipeline

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

InstantiateFunctions

Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions”.

LoopOptimization

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

MapPersistentVarsToRAM

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

See Also

Chart | Message Viewer | Truth Table

Introduced in R2014a

Stop Simulation

Stop simulation when input is nonzero (HDL Coder)

Description

The Stop Simulation block is available with Simulink.

For information about the simulation behavior and block parameters, see Stop Simulation.

HDL Architecture

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

Introduced in R2014a

Subsystem

Represent system within another system (HDL Coder)

Description

The Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see Subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

You can set this property In the HDL Workflow Advisor, 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 (;).

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

Values: `''` (default) | character vector

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

IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: `''` (default) | character vector

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: `''` (default) | character vector

Example: `'1.3'`

Restrictions

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

See Also

Topics

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

Introduced in R2014a

Subtract

Add or subtract inputs (HDL Coder)

Description

The Subtract block is available with Simulink.

For information about the simulation behavior and block parameters, see Subtract.

HDL Architecture

The default `Linear` implementation generates a chain of N operations (adders) for N inputs.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Complex Data Support

The default `Linear` implementation supports complex data.

Introduced in R2014a

Sum

Add or subtract inputs (HDL Coder)

Description

The Sum block is available with Simulink.

For information about the simulation behavior and block parameters, see Sum.

HDL Architecture

The default `Linear` implementation generates a chain of N operations (adders) for N inputs.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Complex Data Support

The default `Linear` implementation supports complex data.

Introduced in R2014a

Sum of Elements

Add or subtract inputs (HDL Coder)

Description

The Sum of Elements block is available with Simulink.

For information about the simulation behavior and block parameters, see Sum of Elements.

HDL Architecture

HDL Coder supports `Tree` and `Cascade` architectures for Sum of Elements blocks that have a single vector input with multiple elements.

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

Architecture	Additional cycles of latency	Description
`Linear`	0	Generates a linear chain of adders to compute the sum of products.
`Tree`	0	Generates a tree structure of adders to compute the sum of products.
`Cascade`	1, when block has a single vector input port.	This implementation optimizes latency * area and is faster than the `Tree` implementation. It computes partial sums and cascades adders. See “Cascade Architecture Best Practices”.

Note To use the **LatencyStrategy** setting in the **Native Floating Point** tab of the HDL Block Properties dialog box, specify `Linear` or `Tree` as the HDL Architecture.

HDL Block Properties

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Native Floating Point

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Complex Data Support

The `Linear` implementation supports complex data.

The `Tree` implementation supports complex data with + for the **List of signs** block parameter. With native floating point support, the `Tree` implementation supports complex data with both + and – for **List of signs**.

Introduced in R2014a

Switch

Switch output between first input and third input based on value of second input (HDL Coder)

Description

The Switch block is available with Simulink.

For information about the simulation behavior and block parameters, see Switch.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior (HDL Coder)

Description

The Synchronous Subsystem block is available with Simulink.

For information about the simulation behavior and block parameters, see Synchronous Subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

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

OutputPipeline

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

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

If this block is not the DUT, the block property settings in the **Target Specification** tab are ignored.

In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using `hdlset_param`, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

ProcessorFPGASynchronization

Processor/FPGA synchronization mode, specified as a character vector.

You can set this property In the HDL Workflow Advisor, 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 (;).

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

Values: `''` (default) | character vector

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

IPCoreName

IP core name, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.

Values: `''` (default) | character vector

Example: `'my_model_name'`

IPCoreVersion

IP core version number, specified as a character vector.

You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.

Values: `''` (default) | character vector

Example: `'1.3'`

Restrictions

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

See Also

Enable | Enabled Synchronous Subsystem | Resettable Synchronous Subsystem | State Control

Topics

“Resettable Subsystem Support in HDL Coder™”
“Using the State Control block to generate more efficient code with HDL Coder™”
“Synchronous Subsystem Behavior with the State Control Block”

Introduced in R2016a

Tapped Delay

Delay scalar signal multiple sample periods and output the delayed versions (HDL Coder)

Description

The Tapped Delay block is available with Simulink.

For information about the simulation behavior and block parameters, see Tapped Delay.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

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

OutputPipeline

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

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Terminator

Terminate unconnected output port (HDL Coder)

Description

The Terminator block is available with Simulink.

For information about the simulation behavior and block parameters, see Terminator.

HDL Architecture

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

Introduced in R2014a

Time Scope

Display time-domain signals (HDL Coder)

Description

The Time Scope block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Time Scope.

HDL Architecture

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

Introduced in R2014a

To File

Write data to file (HDL Coder)

Description

The To File block is available with Simulink.

For information about the simulation behavior and block parameters, see To File.

HDL Architecture

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

Introduced in R2014a

To VCD File

Generate value change dump (VCD) file (HDL Coder)

Description

The To VCD File block is available with HDL Verifier.

For information about the simulation behavior and block parameters, see To VCD File.

HDL Architecture

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

Introduced in R2014a

To Workspace

Write data to MATLAB workspace (HDL Coder)

Description

The To Workspace block is available with Simulink.

For information about the simulation behavior and block parameters, see To Workspace.

HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

Introduced in R2014a

Trigger

Add trigger port to model or subsystem (HDL Coder)

Description

The Trigger block is available with Simulink.

For information about the simulation behavior and block parameters, see Trigger.

HDL Architecture

This block has a single, default HDL architecture.

See Also

Triggered Subsystem

Introduced in R2014a

Triggered Subsystem

Represent subsystem whose execution is triggered by external input (HDL Coder)

Description

A triggered subsystem is a subsystem that receives a control signal via a Trigger block. The triggered subsystem executes for one cycle each time a trigger event occurs. For detailed information on how to define trigger events and configure triggered subsystems, see “Triggered Subsystems” (Simulink).

Best Practices

When using triggered subsystems in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put unit delays on Triggered Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems can affect synthesis results in the following ways:
  - In some cases, the system clock speed can drop by a small percentage.
  - Generated code uses more resources, scaling with the number of triggered subsystem instances and the number of output ports per subsystem.

Using the Signal Builder Block

When you connect outputs from a Signal Builder block to a triggered subsystem, you might need to use a Rate Transition block. To run all triggered subsystem ports at the same rate:

- If the trigger source is a Signal Builder block, but the other triggered subsystem inputs come from other sources, insert a Rate Transition block into the signal path before the trigger input.

- If all inputs (including the trigger) come from a Signal Builder block, they have the same rate, so special action is not required.

Using the Trigger as Clock

You can generate code that uses the trigger signal as a clock with the `TriggerAsClock` property. See “Use Trigger As Clock in Triggered Subsystems”.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

Restrictions

HDL Coder supports HDL code generation for triggered subsystems that meet the following conditions:

- The triggered subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The trigger signal is a scalar.
- The data type of the trigger signal is either `boolean` or `ufix1`.
- Outputs of the triggered subsystem have an initial value of 0.
- All inputs and outputs of the triggered subsystem (including the trigger signal) run at the same rate. (See “Using the Signal Builder Block” on page 3-429 For information about a special case.)
- The **Show output port** parameter of the Trigger block is set to `Off`.
- If the DUT contains the following blocks, `RAMArchitecture` is set to `WithClockEnable`:
  - Dual Port RAM
  - Simple Dual Port RAM
  - Single Port RAM
- The triggered subsystem does not contain the following blocks:
  - Discrete-Time Integrator
  - CIC Decimation
  - CIC Interpolation
  - FIR Decimation
  - FIR Interpolation
  - Downsample

- Upsample
- HDL Cosimulation blocks for HDL Verifier
- Rate Transition

See Also

Subsystem | Trigger

Introduced in R2014a

Triggered To Workspace

Write input sample to MATLAB workspace when triggered (HDL Coder)

Description

The Triggered To Workspace block is avaiable with DSP System Toolbox. For information about the simulation behavior and block parameters, see Triggered To Workspace.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem.
`BlackBox`	Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.
`No HDL`	Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

Black Box Interface Customization

For the `BlackBox` architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface”.

HDL Block Properties

General

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also "BalanceDelays".

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also "ConstrainedOutputPipeline".

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also "DistributedPipelining".

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also "DSPStyle".

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also "FlattenHierarchy".

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also "InputPipeline".

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also "OutputPipeline".

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

See Also

Subsystem | Trigger

Introduced in R2014a

Trigonometric Function

Specified trigonometric function on input (HDL Coder)

Description

The Trigonometric Function block is available with Simulink.

For information about the simulation behavior and block parameters, see Trigonometric Function.

HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model”.

The Trigonometric Function block supports HDL code generation for the functions in this table.

Architecture	Function	Approximation Method	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

General

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

UsePipelinedKernel

Whether to use a pipelined implementation of the CORDIC algorithm in the generated code. The default is `On`.

Setting	Description
`On` (default)	Use a pipelined implementation of the CORDIC algorithm. The pipelined implementation adds latency.
`Off`	Use a combinatorial implementation of the CORDIC algorithm. The combinatorial implementation does not add latency. If the block is in a feedback loop, use this implementation.

Native Floating Point

HandleDenormals

Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also “Denormal Numbers”.

LatencyStrategy

Specify whether to map the blocks in your design to zero, minimum, or maximum latency for the floating-point operator. The default is `inherit`. See also “Latency Considerations with Native Floating Point”.

Restrictions

- For the `sin` and `cos` functions, only signed fixed-point data types are supported for CORDIC approximations.
- HDL Coder displays an error when you select the **SinCosCordic** architecture, **UsePipelinedKernel** is `On`, and the block is in a feedback loop.

See Also

`cordiccos` | `cordicsin` | `cordicsincos`

Introduced in R2014a

Truth Table

Represent logical decision-making behavior with conditions, decisions, and actions (HDL Coder)

Description

The Truth Table block is available with Stateflow.

For information about the simulation behavior and block parameters, see Truth Table.

Tunable Parameters

You can use a tunable parameter in a Truth Table intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters”.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstMultiplierOptimization

Canonical signed digit (CSD) or factored CSD optimization. The default is `none`. See also “ConstMultiplierOptimization”.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

InstantiateFunctions

Generate a VHDL `entity` or Verilog `module` for each function. The default is `off`. See also “InstantiateFunctions”.

LoopOptimization

Unroll, stream, or do not optimize loops. The default is `none`. See also “LoopOptimization”.

MapPersistentVarsToRAM

Map persistent arrays to RAM. The default is `off`. See also “MapPersistentVarsToRAM”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is `off`. See also “UseMatrixTypesInHDL”.

VariablesToPipeline

Warning `VariablesToPipeline` is not recommended. Use `coder.hdl.pipeline` instead.

Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.

See Also

Chart | Message Viewer | State Transition Table

Introduced in R2014a

Unary Minus

Negate input (HDL Coder)

Description

The Unary Minus block is available with Simulink.

For information about the simulation behavior and block parameters, see Unary Minus.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Unit Delay

Delay signal one sample period (HDL Coder)

Description

The Unit Delay block is available with Simulink.

For information about the simulation behavior and block parameters, see Unit Delay.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Unit Delay Enabled (Obsolete)

Delay signal one sample period, if external enable signal is on (HDL Coder)

Note The Unit Delay Enabled block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Enabled Delay block instead. Existing models that contain the Unit Delay Enabled block continue to work for backward compatibility.

Description

The Unit Delay Enabled block delays a signal by one sample period when the external enable signal is on. While the enable is off, the block is disabled. It holds the current state at the same value and outputs that value.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2007b

Unit Delay Enabled Resettable (Obsolete)

Delay signal one sample period, if external enable signal is on, with external Boolean reset (HDL Coder)

Note The Unit Delay Enabled Resettable block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Enabled Resettable Delay block instead. Existing models that contain the Unit Delay Enabled Resettable block continue to work for backward compatibility.

Description

The Unit Delay Enabled Resettable block can delay the signal one sample period, if external enable signal is on, with external reset as off. If the enable signal is off, the block is disabled.

When the enable and reset signals are on, the block output resets the current state.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

SoftReset

Specify `on` to generate reset logic for the block that is more efficient for synthesis, but does not match the Simulink behavior. The default is `off`. See “SoftReset”.

Introduced in R2010a

Unit Delay Resettable (Obsolete)

Delay signal one sample period, with external Boolean reset (HDL Coder)

Note The Unit Delay Resettable block is not recommended. This block was removed from the Discrete library in R2016b. In new models, use the Resettable Delay block instead. Existing models that contain the Unit Delay Enabled Resettable block continue to work for backward compatibility.

Description

The Unit Delay Resettable block delays the signal one sample period, with external reset. The block can reset both its state and output based on an external reset signal. The block has two input ports. One input port is for the input signal and the other input port is for the external reset signal.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

SoftReset

Specify `on` to generate reset logic for the block that is more efficient for synthesis, but does not match the Simulink behavior. The default is `off`. See “SoftReset”.

Introduced in R2010a

Unit Delay Enabled Synchronous

Delay input signal by one sample period when external Enable signal is true (HDL Coder)

Description

The Unit Delay Enabled Synchronous block is available in the Discrete section of the HDL Coder block library. The block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in **Synchronous mode**. The synchronous behavior of the State Control block generates cleaner HDL code and uses fewer hardware resources. For more information, see State Control.

For information about the simulation behavior and block parameters, see Unit Delay Enabled Synchronous.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2017b

Unit Delay Resettable Synchronous

Delay input signal by one sample period when external Reset signal is false (HDL Coder)

Description

The Unit Delay Resettable Synchronous block is available in the Discrete section of the HDL Coder block library. The block implementation consists of a Synchronous Subsystem that contains a Resettable Delay block with a **Delay length** of one and a State Control block in **Synchronous mode**. The synchronous behavior of the State Control block generates cleaner HDL code and uses fewer hardware resources. For more information, see State Control.

For information about the simulation behavior and block parameters, see Unit Delay Resettable Synchronous.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2017b

Unit Delay Enabled Resettable Synchronous

Delay input signal by one sample period when external Enable signal is true and external Reset signal is false (HDL Coder)

Description

The Unit Delay Enabled Resettable Synchronous block is available in the Discrete section of the HDL Coder block library. The block implementation consists of a Synchronous Subsystem that contains an Enabled Resettable Delay block with a **Delay length** of one and a State Control block in **Synchronous mode**. The synchronous behavior of the State Control block generates cleaner HDL code and uses fewer hardware resources. For more information, see State Control.

For information about the simulation behavior and block parameters, see Unit Delay Enabled Resettable Synchronous.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

ResetType

Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2017b

Upsample

Resample input at higher rate by inserting zeros (HDL Coder)

Description

The Upsample block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Upsample.

Best Practices

Consider whether your model can use the Repeat block instead of the Upsample block. The Repeat block uses fewer hardware resources, so it is a best practice to use Upsample only when your algorithm requires zero-padding upsampling.

See also “Multirate Model Requirements for HDL Code Generation”.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Restrictions

Input processing set to `Columns as channels (frame based)` is not supported.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Variable Selector

Select subset of rows or columns from input (HDL Coder)

Description

The Variable Selector block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Variable Selector.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Variant Subsystem

Represent a subsystem with multiple subsystems (HDL Coder)

Description

The Variant Subsystem block is available with Simulink. For information about the simulation behavior and block parameters, see Variant Subsystem.

HDL Architecture

Architecture	Description
`Module` (default)	Generate code for the subsystem and the blocks within the subsystem. HDL Coder generates code for only the active variant.
`BlackBox`	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

AdaptivePipelining

Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is `inherit`. See also AdaptivePipelining.

BalanceDelays

Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is `inherit`. See also “BalanceDelays”.

ClockRatePipelining

Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is `inherit`. See also ClockRatePipelining.

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

DistributedPipelining

Pipeline register distribution, or register retiming. The default is `off`. See also “DistributedPipelining”.

DSPStyle

Synthesis attributes for multiplier mapping. The default is `none`. See also “DSPStyle”.

FlattenHierarchy

Remove subsystem hierarchy from generated HDL code. The default is `inherit`. See also “FlattenHierarchy”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing”.

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming”.

Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

Restrictions

- The DUT cannot be a Variant Subsystem.

Introduced in R2014a

Vector Concatenate

Concatenate input signals of same data type to create contiguous output signal (HDL Coder)

Description

The Vector Concatenate block is available with Simulink.

For information about the simulation behavior and block parameters, see Vector Concatenate.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Complex Data Support

This block supports code generation for complex signals.

Restrictions

HDL code generation does not support matrices at the input or output ports of the block .

Introduced in R2014a

Vector Scope

Display vector or matrix of time-domain, frequency-domain, or user-defined data (HDL Coder)

Description

The Vector Scope block is available with DSP System Toolbox.

For information about 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.

Introduced in R2014a

Viterbi Decoder

Decode convolutionally encoded data using Viterbi algorithm (HDL Coder)

Description

The Viterbi Decoder block is available with Communications System Toolbox.

For information about the simulation behavior and block parameters, see Viterbi Decoder.

HDL Coder supports the following features of the Viterbi Decoder block:

- Non-recursive encoder/decoder with feed-forward trellis and simple shift register generation configuration
- Sample-based input
- Decoder rates from 1/2 to 1/7
- Constraint length from 3 to 9

HDL Architecture

The Viterbi Decoder block decodes every bit by tracing back through a traceback depth that you define for the block. The block implements a complete traceback for each decision bit, using registers to store the minimum state index and branch decision in the traceback decoding unit. There are two methods to optimize the traceback logic: a pipelined register-based implementation or a RAM-based architecture. See the “HDL Code Generation for Viterbi Decoder” (Communications System Toolbox) example.

Register-Based Traceback

You can specify that the traceback decoding unit be pipelined to improve the speed of the generated circuit. You can add pipeline registers to the traceback unit by specifying the number of traceback stages per pipeline register.

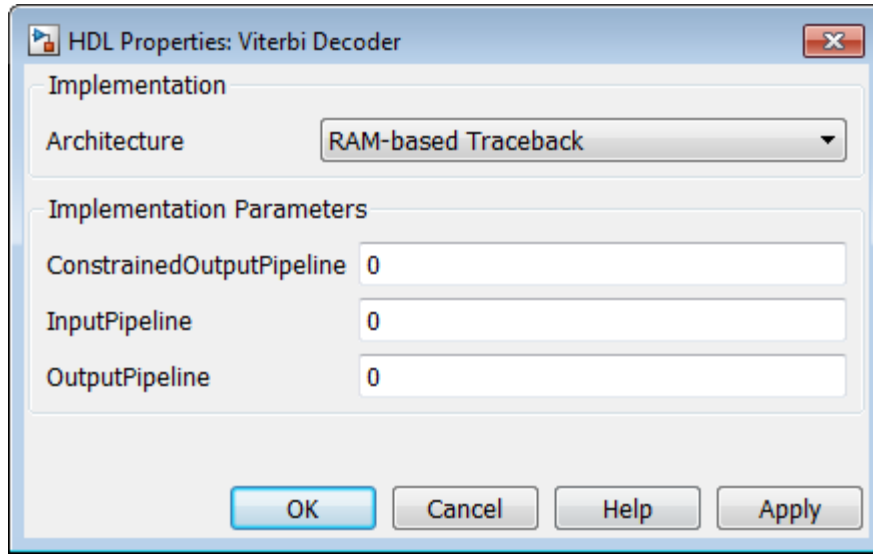
Using the `TracebackStagesPerPipeline` implementation parameter, you can balance the circuit performance based on system requirements. A smaller parameter value

indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the parameter value results in fewer registers along with a decrease in the circuit speed.

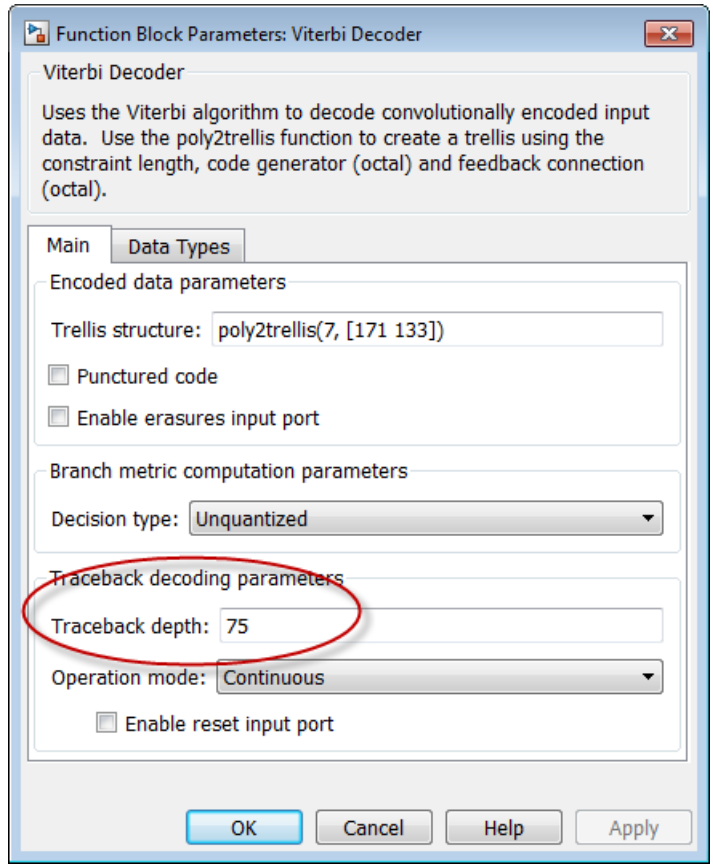
RAM-Based Traceback

Instead of using registers, you can choose to use RAMs to save the survivor branch information. The coder does not support **Enable reset input port** when using RAM-based traceback.

1 Set the **Architecture** property of the Viterbi Decoder block to `RAM-based Traceback`.



2 Set the traceback depth on the Viterbi Decoder block mask.



RAM-based traceback and register-based traceback differ in the following ways:

- The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data. The register-based implementation combines the traceback and decode operations into one step. It uses the best state found from the minimum operation as the decoding initial state.
- RAM-based implementation traces back through M samples, decodes the previous M bits in reverse order, and releases one bit in order at each clock cycle. The register-based implementation decodes one bit after a complete traceback.

Because of the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback.

A longer traceback depth, for example, 10 times the constraint length, is recommended in the RAM-based traceback. This depth achieves a similar bit error rate (BER) as the register-based implementation. The size of RAM required for the implementation depends on the trellis and the traceback depth.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

TracebackStagesPerPipeline

See “Register-Based Traceback” on page 3-469.

Restrictions

- **Punctured code**: Do not select this option. Punctured code requires frame-based input, which HDL Coder does not support.
- **Decision type**: The coder does not support the `Unquantized` decision type.
- **Error if quantized input values are out of range**: The coder does not support this option.
- **Operation mode**: The coder supports only the `Continuous` mode.
- **Enable reset input port**: When you enable both **Enable reset input port** and **Delay reset action to next time step**, HDL support is provided. You must select `Continuous` operation mode, and use register-based traceback.

- You cannot use the Viterbi Decoder block inside a Resettable Synchronous Subsystem.

Input and Output Data Types

- When **Decision type** is set to `Soft decision`, the HDL implementation of the Viterbi Decoder block supports fixed-point inputs and output. For input, the fixed-point data type must be `ufixN`. `N` is the number of soft-decision bits. Signed built-in data types (`int8`, `int16`, `int32`) are not supported. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- When **Decision type** is set to `Hard decision`, the block supports input with data types `ufix1` and `Boolean`. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.
- The HDL implementation of the Viterbi Decoder block does not support double and single input data types. The block does not support floating point output for fixed-point inputs.

See Also

Topics

"HDL Code Generation for Viterbi Decoder" (Communications System Toolbox)

Introduced in R2014a

Waterfall

View vectors of data over time (HDL Coder)

Description

The Waterfall block is available with DSP System Toolbox.

For information about the simulation behavior and block parameters, see Waterfall.

HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

Introduced in R2014a

Wrap To Zero

Set output to zero if input is above threshold (HDL Coder)

Description

The Wrap To Zero block is available with Simulink.

For information about the simulation behavior and block parameters, see Wrap To Zero.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Restrictions

The input signal and **Threshold** parameter must have equal size. For example, if the input is a two-dimensional vector, **Threshold** must also be a two-dimensional vector.

Introduced in R2014b

XY Graph

Display X-Y plot of signals using MATLAB figure window (HDL Coder)

Description

The XY Graph block is available with Simulink.

For information about the simulation behavior and block parameters, see XY Graph.

HDL Architecture

When you use this block in your model, HDL Coder does not generate HDL code for it.

Introduced in R2014a

Zero-Order Hold

Implement zero-order hold of one sample period (HDL Coder)

Description

The Zero-Order Hold block is available with Simulink.

For information about the simulation behavior and block parameters, see Zero-Order Hold.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline

Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. See also “ConstrainedOutputPipeline”.

InputPipeline

Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “InputPipeline”.

OutputPipeline

Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. See also “OutputPipeline”.

Complex Data Support

This block supports code generation for complex signals.

Introduced in R2014a

Properties — Alphabetical List

AdaptivePipelining

Insert adaptive pipeline registers in your design

Settings

`'on'` (default)

Insert adaptive pipeline registers in your design. For HDL Coder to insert adaptive pipelines, you must specify the synthesis tool. If your design has multipliers, specify the synthesis tool and the target frequency for adaptive pipeline insertion.

`'off'`

Do not insert adaptive pipelines in your design.

Usage Notes

You can further control adaptive pipelining 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

Topics

“Adaptive pipelining”
“AdaptivePipelining”

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

Topics

"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

Topics

“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 postfix to block labels used for HDL `GENERATE` statements

Settings

'*postfix*'

Default: `'_gen'`

Specify the postfix as a character vector. HDL Coder appends the postfix 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

'*Highlighting script*'

Default: 'highlightCriticalPathEstimationOffendingBlocks'

Name of MATLAB script that contains commands to highlight blocks on the estimated critical path without timing information. To override the default, specify the highlighting script as a character vector. 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

Topics

"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

'*Enable input port*'

Default: 'clk_enable'

Specify the name for the model's clock enable input port as a character vector.

If you override the default with (for example) '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 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

`'Enable output port'`

Default: `'ce_out'`

Specify the name for the generated clock enable output port as a character vector.

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

'*Clock Input*'

Default: 'clk'.

Specify the clock input port name as a character vector.

For example, if you override the default with '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 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 postfix to append to HDL clock process names

Settings

`'postfix'`

Default: `'_process'`.

Specify the postfix as a character vector. HDL Coder appends the postfix to HDL clock process names. 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 `_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.

Usage Notes

You can further control clock-rate pipelining 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

ClockRatePipelineOutputPorts

Topics

"Clock-Rate Pipelining"
"ClockRatePipelining"

CodeGenerationOutput

Control production of generated code and display of generated model

Settings

`'Generated code output'`

To control the generated code output and display of generated model, specify one of the following:

Default: `'GenerateHDLCode'` generates code but does not display the generated model.

`'GenerateHDLCodeAndDisplayGeneratedModel'` generates HDL code and displays the generated model after code generation.

`'DisplayGeneratedModelOnly'` creates and displays the generated model, but does 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

Topics

"Generated Model and Validation Model"

ComplexImagPostfix

Specify text to append to imaginary part of complex signal names

Settings

'*Signal name postfix*'

Specify text to append to imaginary part of complex signal names as a character vector.

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 text to append to real part of complex signal names

Settings

'*Signal name postfix*'

Specify text to append to real part of complex signal names as a character vector.

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

Topics

"Find Estimated Critical Paths Without Synthesis Tools"

CriticalPathEstimationFile

Critical path estimation highlighting script name

Settings

`'critical path script name'`

Default: `'criticalPathEstimated'`

To override the default, specify the critical path estimation highlighting script name as a character vector. The MATLAB script 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

Topics

“Find Estimated Critical Paths Without Synthesis Tools”

CustomFileFooterComment

Specify a custom file footer comment in the generated HDL code

Settings

Default: `''`

Enter custom comments to appear as footer in the generated HDL file for your design. For example, you can specify arguments such as revision, generated log file, revision number, and so on.

```
//============================================================
//  xxxxxx
//============================================================
//  $Log$
//  Revision 1.2  2009/12/14 04:38:51  sxxxxxx
//  Initial revision
//
//============================================================
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

CustomFileHeaderComment

Specify a custom file header comment in the generated HDL code

Settings

Default: `''`

Enter custom comments to appear as footer in the generated HDL file for your design. For example, you can specify arguments such as title, author, modified date, and so on.

```
// ========================================================
// Title           : <%Title%>
// Project         : <%Project%>
// Author          : <%Author%>
//
// Revision           : $Revision$
// Date Modified      : $Date$
// ========================================================
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

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`.

See Also

Topics

“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

Topics

"Find Feedback Loops"

DistributedPipeliningBarriersFile

Distributed pipelining barriers highlighting script name

Settings

`'Pipelining barriers file'`

Default: `'highlightDistributedPipeliningBarriers'`

Name of MATLAB script that contains commands to highlight blocks that are inhibiting distributed pipelining in the original model and generated model. To override the default, specify the highlighting script name as a character vector. 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

Topics

“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 for internal clock enables in generated code

Settings

`'base name'`

Default: `'enb'`

Specify the base name as a character vector. HDL Coder uses 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`.

EnableTestpoints

Enable creation of DUT output ports in generated HDL code for test point signals

Settings

`'on'`

When you enable this setting, the code generator creates DUT output ports for the test point signals in the generated HDL code. To observe and debug the test point signals, connect a block from the **Sinks** block library, such as a Scope block, to the output ports corresponding to these signals.

`'off'` (default)

When you disable this setting, the code generator preserves the test point signals and does not create DUT output ports in the generated HDL code.

Note The code generator ignores the `EnableTestpoints` setting when you designate test points for states inside a Stateflow Chart.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

"Model and Debug Test Point Signals with HDL Coder™"

EntityConflictPostfix

Specify postfix to duplicate VHDL entity or Verilog module names

Settings

`'postfix'`

Default: `'_block'`

Specify the postfix as a character vector. The 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`

EnumEncodingScheme

Encoding scheme to represent enumeration types in the generated HDL code

Settings

`'default'` (default)

The code generator uses decimal encoding in Verilog and VHDL-native enumerated types in VHDL. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 2'd0,
is_Chart_IN_s_rx = 2'd1,
is_Chart_IN_s_wait_0 = 2'd2,
is_Chart_IN_s_wait_tb = 2'd3;
```

`'onehot'`

The code generator uses a one-hot encoding scheme where a single bit is high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0001,
is_Chart_IN_s_rx = 4'b0010,
is_Chart_IN_s_wait_0 = 4'b0100,
is_Chart_IN_s_wait_tb = 4'b1000;
```

This encoding scheme does not support more than `64` enumeration values or number of states.

`'twohot'`

The code generator uses a two-hot encoding scheme where two bits are high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0011,
```

```
is_Chart_IN_s_rx = 4'b0101,
is_Chart_IN_s_wait_0 = 4'b0110,
is_Chart_IN_s_wait_tb = 4'b1001;
```

`'binary'`

The code generator uses a binary encoding scheme to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 2'b00,
is_Chart_IN_s_rx = 2'b01,
is_Chart_IN_s_wait_0 = 2'b10,
is_Chart_IN_s_wait_tb = 2'b11;
```

In VHDL, the generated code uses `CONSTANT` types to encode nondefault enumeration values in the generated code. For example, this code snippet shows the generated VHDL code when you use the two-hot state encoding for a Stateflow Chart that has four states.

```
 PACKAGE s_pkg IS
  -- Constants
  -- Two-hot encoded enumeration values for type state_type_is_Chart
  CONSTANT IN_s_idle         : std_logic_vector(3 DOWNTO 0) :=
    "0011";
  CONSTANT IN_s_rx           : std_logic_vector(3 DOWNTO 0) :=
    "0101";
  CONSTANT IN_s_wait_0       : std_logic_vector(3 DOWNTO 0) :=
    "0110";
  CONSTANT IN_s_wait_tb      : std_logic_vector(3 DOWNTO 0) :=
    "1001";

END s_pkg;
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

ForceClock

Specify whether test bench forces clock input signals

Settings

`'on'` (default)

Selected (default)

Specify that the test bench forces the clock input signals. When this option is set, the clock high and low time settings control the clock waveform.

`'off'`

Cleared

Specify that a user-defined external source forces the clock input signals.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockLowTime`, `ClockHighTime`, `ForceClockEnable`, `ForceReset`, `HoldTime`

ForceClockEnable

Specify whether test bench forces clock enable input signals

Settings

`'on'` (default)

Selected (default)

Specify that the test bench forces the clock enable input signals to active high (1) or active low (0), depending on the setting of the clock enable input value.

`'off'`

Cleared

Specify that a user-defined external source forces the clock enable input signals.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `HoldTime`

ForceReset

Specify whether test bench forces reset input signals

Settings

`'on'` (default)

Selected (default)

Specify that the test bench forces the reset input signals. If you enable this option, you can also specify a hold time to control the timing of a reset.

`'off'`

Cleared

Specify that a user-defined external source forces the reset input signals.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`ClockHighTime`, `ClockLowTime`, `ForceClock`, `HoldTime`

FPToleranceStrategy

Specify whether to check for floating-point tolerance based on relative error or ULP error

Settings

Use this setting to specify the tolerance strategy for checking the numerical accuracy in the generated test bench. Based on the tolerance strategy that you specify, you can enter a custom tolerance value.

`'relative'` (default)

When you verify the generated code, HDL Coder checks for the floating-point tolerance based on the relative error.

`'ulp'`

When you verify the generated code, HDL Coder checks for the floating-point tolerance based on the ULP error.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Example

To specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');

% When using ULP error, optionally enter tolerance value greater than or equal to 0
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

See Also

- FPToleranceValue
- “Floating point tolerance check based on”
- “Tolerance Value”
- “Getting Started with HDL Coder Native Floating-Point Support”

FPToleranceValue

Enter the tolerance value based on floating-point tolerance check setting

Settings

N

Default: 1e-07

The value of *N* depends on the floating-point tolerance check setting that you specify. Use this setting to specify a custom tolerance value for checking the numerical accuracy in the generated test bench. When you set the **Floating point tolerance check based on** to:

- `relative error`, the default is a tolerance value of `1e-07`. When you use this floating-point tolerance check setting, specify the tolerance value as a double data type.
- `ulp error`, the default is a **Tolerance Value** of `0`. When you use this floating-point tolerance check setting, specify the tolerance value as an integer. You can specify a **Tolerance Value**, N, that is greater than or equal to `0`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Example

To specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% check for floating-point tolerance based on the ULP error
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');
```

```
% When using ULP error, optionally enter tolerance value greater than or equal to 0
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);
```

See Also

- “Tolerance Value”
- FPToleranceStrategy
- “Floating point tolerance check based on”

GenerateCoSimBlock

Generate HDL Cosimulation blocks for use in testing DUT

Settings

`'on'`

If your installation includes one or more of the following HDL simulation features, HDL Coder generates an HDL Cosimulation block for each:

- HDL Verifier for use with Mentor Graphics ModelSim
- HDL Verifier for use with Cadence Incisive

The coder configures the generated HDL Cosimulation blocks to conform to the port and data type interface of the DUT selected for code generation. By connecting an HDL Cosimulation block to your model in place of the DUT, you can cosimulate your design with the desired simulator.

The coder appends the character vector that the `CosimLibPostfix` property specifies to the names of the generated HDL Cosimulation blocks.

`'off'` (default)

Do not generate HDL Cosimulation blocks.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

GenerateCoSimModel

Generate model containing HDL Cosimulation block for use in testing DUT

Settings

`'ModelSim'` (default)

If your installation includes HDL Verifier, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for Mentor Graphics ModelSim.

`'Incisive'`

If your installation includes HDL Verifier, the coder generates and opens a Simulink model that contains an HDL Cosimulation block for Cadence Incisive.

`'None'`

Do not create a cosimulation model.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate a Cosimulation Model”

GeneratedModelName

Specify name of generated model

Settings

`'Generated model name'`

By default, the name of a generated model is the same as that of the original model. Specify the `GeneratedModelName` as a character vector 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

Topics

"Generated Model and Validation Model"

GeneratedModelNamePrefix

Specify prefix to name of generated model

Settings

`'Generated model prefix'`

Default: `'gm_'`

Specify the prefix as a character vector. HDL Coder appends the prefix to name of 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

Topics

“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”

GenerateModel

Control generation of the generated model with HDL code

Settings

`'on'` (default)

Create the generated model.

The generated model is a behavioral model of the HDL code. Using the generated model, you can see the effects of HDL block settings and optimizations such as resource sharing and streaming that you specify in your Simulink model.

`'off'`

Do not create the generated model.

To verify your design by generating a test bench, validation model, or a cosimulation model, set `GenerateModel` 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

GenerateValidationModel

Topics

“Generated Model and Validation Model”

GenerateHDLTestbench

Generate HDL test bench

Settings

`'on'` (default)

Generate HDL test bench.

`'off'`

Do not generate HDL test bench.

The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT. You can disable this property when you use an alternate test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`SimulationTool`

GenerateSVDPITestbench

Generate SystemVerilog DPI test bench

Settings

`'ModelSim'` (default)

Generate System Verilog DPI test bench, and build-and-run scripts, for the Mentor Graphics ModelSim simulator.

`'Incisive'`

Generate System Verilog DPI test bench, and build-and-run scripts, for the Cadence Incisive simulator.

`'VCS'` (default)

Generate System Verilog DPI test bench, and build-and-run scripts, for the Synopsys® VCS® simulator.

`'Vivado'`

Generate System Verilog DPI test bench, and build-and-run scripts, for the Xilinx Vivado simulator.

When you set this property, the coder generates a direct programming interface (DPI) component for your entire Simulink model, including your DUT and data sources. Your entire model must support C code generation with Simulink Coder. The coder generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

Consider using this option if the default HDL test bench takes a long time to generate or simulate. Generation of a DPI test bench is sometimes faster than the default version because it does not run a full Simulink simulation to create the test bench data. Simulation of a DPI test bench with a large data set is faster than the default version because it does not store the input or expected data in a separate file.

To use this feature, you must have HDL Verifier and Simulink Coder licenses. To run the SystemVerilog testbench with generated VHDL code, you must have a mixed-language simulation license for your HDL simulator.

Limitations This test bench is not supported when you generate HDL code for the top-level Simulink model. Your DUT subsystem must meet the following conditions:

- Input and output data types of the DUT cannot be larger than 64 bits.
- Input and output ports of the DUT cannot use enumerated data types.
- Input and output ports cannot be single-precision or double-precision data types.
- The DUT cannot have multiple clocks. You must set the **Clock inputs** code generation option to `Single`.
- **Use trigger signal as clock** must not be selected.
- If the DUT uses vector ports, you must use **Scalarize vector ports** to flatten the interface.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

"Generate a SystemVerilog DPI Test Bench" on page 2-75.

"Verify HDL Design With Large Data Set Using SystemVerilog DPI Test Bench"

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”
- GenerateModel
- “Generated Model and Validation Model”

HDLCodeCoverage

Include HDL code coverage switches in generated test bench scripts

Settings

`'on'`

Generated script includes code coverage switches. When you run the HDL simulation, code coverage is collected for your generated test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

`'off'` (default)

Generated script does not include code coverage switches, and does not collect code coverage.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`SimulationTool`

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.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

HDLCompileInit

Specify text written to initialization section of compilation script

Settings

`'Initialization text'`

Default: `'vlib %s\n'`.

Specify text written to initialization section of compilation script as a character vector. If your `TargetLanguage` is `VHDL`, the implicit argument, `%s`, is the contents of the `VHDLLibraryName` property. If your `TargetLanguage` is `Verilog`, the implicit argument is `work`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

VHDLLibraryName

Topics

"Generate Scripts for Compilation, Simulation, and Synthesis"

HDLCompileTerm

Specify text written to termination section of compilation script

Settings

`'Termination text'`

Specify text written to termination section of compilation script as a character vector. The default is `''`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLCompileFilePostfix

Specify postfix appended to file name for generated Mentor Graphics ModelSim compilation scripts

Settings

`'Compilation file postfix'`

Default:`'_compile.do'`.

Specify the postfix as a character vector. HDL Coder appends the postfix to the file name for generated Mentor Graphics ModelSim compilation scripts.

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

HDLCompileVerilogCmd

Specify command written to compilation script for Verilog files

Settings

'*Compilation command*'

Default: `'vlog %s %s\n'`.

Specify command written to compilation script for Verilog files as a character vector. The two arguments are the contents of the `SimulatorFlags` property and the file name of the current module. To omit the flags, set `SimulatorFlags` to `''` (the default).

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLCompileVHDLCmd

Specify command written to compilation script for VHDL files

Settings

'*Compilation command*'

Default: `'vcom %s %s\n'`.

Specify command written to compilation script for VHDL files as a character vector. The two arguments are the contents of the `SimulatorFlags` property and the file name of the current entity. To omit the flags, set `SimulatorFlags` to `''` (the default).

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLGenerateWebview

Include model Web view in the HDL Code Generation report

Settings

`'on'`

Include model Web view in the code generation report to navigate between the code and model within the same window. With a model Web view, you can click a link in the generated code to highlight the corresponding block in the model.

`'off'` (default)

Omit model Web view in the code generation report.

Dependencies

To include a Web view (Simulink Report Generator) of the model in the Code Generation report, you must have Simulink Report Generator™ installed.

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”

HDLMapFilePostfix

Specify postfix appended to file name for generated mapping file

Settings

'*Map file postfix*'

Default: '_map.txt'.

Specify the postfix as a character vector. HDL Coder appends the postfix to file name for generated mapping file.

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 command written to simulation script

Settings

'*Simulation command*'

Default: `'vsim -novopt %s.%s\n'`.

Specify the command written to simulation script as a character vector. If your `TargetLanguage` is `'VHDL'`, the first implicit argument is the value of VHDLLibraryName. If your `TargetLanguage` is `'Verilog'`, the first implicit argument is `'work'`.

The second implicit argument is the top-level module or entity name.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLSimInit

Specify text written to initialization section of simulation script

Settings

'*Simulation initialization*'

Specify text written to initialization section of simulation script as a character vector. The default is

```
['onbreak resume\n',...
'onerror resume\n']
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLSimFilePostfix

Specify postfix appended to file name for generated Mentor Graphics ModelSim simulation scripts

Settings

'*Simulation file postfix*'

Default: `_sim.do`.

Specify the postfix as a character vector. HDL Coder appends the postfix to the file name for generated Mentor Graphics ModelSim simulation scripts.

For example, if the name of your test bench file is `my_design`, HDL Coder adds the postfix `_sim.do` to form the name `my_design_tb_sim.do`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

HDLSimTerm

Specify text written to termination section of simulation script

Settings

`'Termination text'`

Specify text written to termination section of simulation script as a character vector. Default is `'run -all\n'`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLSimViewWaveCmd

Specify waveform viewing command written to simulation script

Settings

`'Waveform view command'`

Default: `'add wave sim:%s\n'`

Specify waveform viewing command as a character vector. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Generate Scripts for Compilation, Simulation, and Synthesis”

HDLLintCmd

Specify command written to HDL lint script

Settings

'*Script command*'

Default: ''

Specify the command written to the HDL lint Tcl script as a character vector. The command must contain `%s`, which is a placeholder for the HDL file name.

Dependencies

If `HDLLintCmd` is set to the default value, `''`, and you set `HDLLintCmd` to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default command string in the Tcl script.

Usage

If you set `HDLLintTool` to `Custom`, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script. Specify `HDLLintCmd` using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLLintTool`, `HDLLintInit`, `HDLLintTerm`, “Generate an HDL Lint Tool Script”

HDLLintInit

Specify HDL lint script initialization name

Settings

`'Initialization name'`

Default: `''`

Specify the HDL lint script initialization name as a character vector.

Dependencies

If `HDLLintInit` is set to the default value, `''`, and you set `HDLLintCmd` to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default initialization string in the Tcl script.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLLintTool`, `HDLLintCmd`, `HDLLintTerm`, "Generate an HDL Lint Tool Script"

HDLLintTerm

Specify HDL lint script termination name

Settings

`'Script termination name'`

Default: `''`

Specify the HDL lint script termination name as a character vector.

Dependencies

If `HDLLintTerm` is set to the default value, `''`, and you set `HDLLintCmd` to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default termination string in the Tcl script.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLLintTool`, `HDLLintCmd`, `HDLLintInit`, “Generate an HDL Lint Tool Script”

HDLLintTool

Select HDL lint tool for which HDL Coder generates scripts

Settings

`'Lint Tool'`

Default: `'None'`.

`HDLLintTool` enables or disables generation of scripts for third-party HDL lint tools. By default, HDL Coder does not generate a lint script.

To generate a script for one of the supported lint tools, set `HDLLintTool` to one of the following:

HDLLintTool Option	Lint Tool
`'None'`	None. Lint script generation is disabled.
`'AscentLint'`	Real Intent Ascent Lint
`'Leda'`	Synopsys Leda
`'SpyGlass'`	Atrenta SpyGlass
`'Custom'`	A custom lint tool.

Dependencies

If you set `HDLLintTool` to one of the supported third-party tools, you can generate a Tcl script without setting `HDLLintInit`, `HDLLintCmd`, and `HDLLintTerm` to nondefault values. If the `HDLLintInit`, `HDLLintCmd`, and `HDLLintTerm` have default values, HDL Coder automatically writes tool-specific default initialization, command, and termination strings to the Tcl script.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Properties

HDLLintCmd | HDLLintInit | HDLLintTerm

Topics

“Generate an HDL Lint Tool Script”

HDLSynthCmd

Specify command written to synthesis script

Settings

'*Synthesis command*'

Default: none.

Specify command written to synthesis script as a character vector. Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis command string. The default is a formatted text string passed to `fprintf` to write the command section of the synthesis script. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLSynthTool`, `HDLSynthInit`, `HDLSynthTerm`, `HDLSynthFilePostfix`, "Generate Scripts for Compilation, Simulation, and Synthesis"

HDLSynthFilePostfix

Specify postfix appended to file name for generated synthesis scripts

Settings

`'file name postfix'`

Specify `HDLSynthTool` as a character vector.

Default: The value of `HDLSynthFilePostfix` normally defaults to a string that corresponds to the synthesis tool that `HDLSynthTool` specifies.

For example, if the value of `HDLSynthTool` is `'Synplify'`, `HDLSynthFilePostfix` defaults to `'_synplify.tcl'`. Then, if the name of the device under test is `my_design`, HDL Coder adds the postfix `_synplify.tcl` to form the synthesis script file name `my_design_synplify.tcl`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLSynthTool`, `HDLSynthCmd`, `HDLSynthInit`, `HDLSynthTerm`, "Generate Scripts for Compilation, Simulation, and Synthesis"

HDLSynthInit

Specify text written to initialization section of synthesis script

Settings

`'Initialization text'`

Default: none

Specify the text written to the synthesis script initialization as a character vector. Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis script initialization string. The default is a formatted text passed to `fprintf` to write the initialization section of the synthesis script. The default is a synthesis project creation command. The implicit argument is the top-level module or entity name. The content of the string is specific to the selected synthesis tool.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLSynthTool`, `HDLSynthCmd`, `HDLSynthTerm`, `HDLSynthFilePostfix`, “Generate Scripts for Compilation, Simulation, and Synthesis”

HDLSynthTerm

Specify text written to termination section of synthesis script

Settings

`'Termination text'`

Default: none

Specify the synthesis script termination text as a character vector. Your choice of synthesis tool (see `HDLSynthTool`) sets the synthesis termination string. The default is a formatted text passed to `fprintf` to write the termination and clean up section of the synthesis script. This section does not take arguments. The content of the string is specific to the selected synthesis tool.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLSynthTool`, `HDLSynthCmd`, `HDLSynthInit`, `HDLSynthFilePostfix`, "Generate Scripts for Compilation, Simulation, and Synthesis"

HDLSynthTool

Select synthesis tool for which HDL Coder generates scripts

Settings

'*Synthesis tool*'

Default: 'None'.

Specify the synthesis tool as a character vector. HDLSynthTool enables or disables generation of scripts for third-party synthesis tools. By default, HDL Coder does not generate a synthesis script. To generate a script for one of the supported synthesis tools, set HDLSynthTool to one of the following:

Tip The value of HDLSynthTool also sets the postfix (HDLSynthFilePostfix) that the coder appends to generated synthesis script file names.

Choice of HDLSynthTool Value...	Generates Script For...	Sets HDLSynthFilePostfix To...
'None'	N/A; script generation disabled	N/A
'ISE'	Xilinx ISE	'_ise.tcl'
'Libero'	Microsemi Libero	'_libero.tcl'
'Precision'	Mentor Graphics Precision	'_precision.tcl'
'Quartus'	Altera Quartus II	'_quartus.tcl'
'Synplify'	Synopsys Synplify Pro®	'_synplify.tcl'
'Vivado'	Xilinx Vivado	'_vivado.tcl'
'Custom'	A custom synthesis tool	'_custom.tcl'

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`HDLSynthCmd`, `HDLSynthInit`, `HDLSynthTerm`, `HDLSynthFilePostfix`, "Generate Scripts for Compilation, Simulation, and Synthesis"

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”

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

Topics

“Find Feedback Loops”

HighlightClockRatePipeliningFile

Clock-rate pipelining highlighting script name

Settings

`'Pipelining highlighting script'`

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. To override the default, specify the clock-rate pipelining highlighting script name as a character vector. 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

Topics

"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

Topics

“Find Feedback Loops”

HighlightFeedbackLoopsFile

Feedback loop highlighting script file name

Settings

`'Loop file name'`

Default: `'highlightFeedbackLoop'`

Specify the feedback loop highlighting script file name as a character vector. The 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

Topics

"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

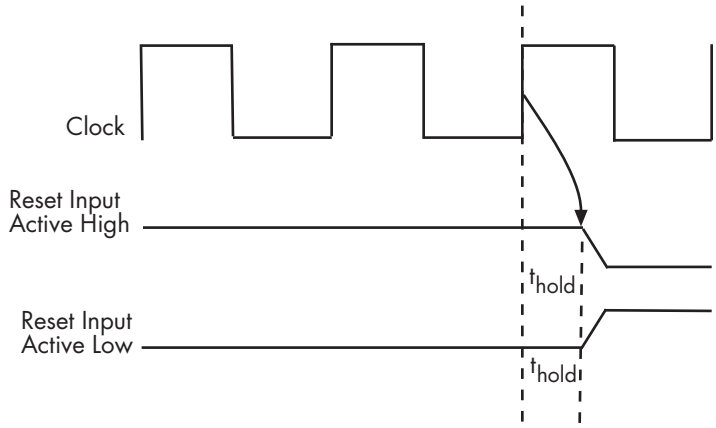
Specify the number of nanoseconds during which the model's data input signals and forced reset input signals are held past the clock rising edge.

The hold time is expressed as a positive integer.

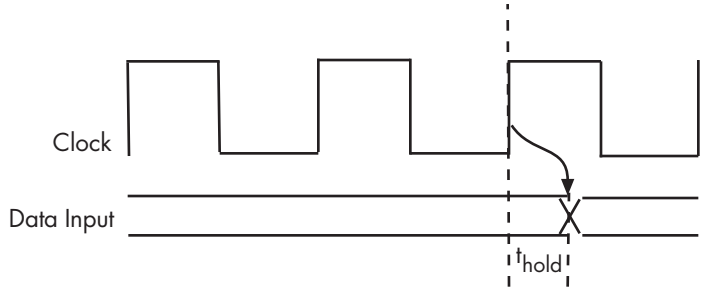
This option applies to reset input signals only if forced resets are enabled.

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 text to append to instance section labels in VHDL `GENERATE` statements

Settings

'*Instance name*'

Default: `'_gen'`

Specify the postfix as a character vector. HDL Coder appends the postfix 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 postfix to generated component instance names

Settings

'*postfix*'

Default: '' (no postfix appended)

Specify the postfix as a character vector. HDL Coder appends the postfix 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 prefix to generated component instance names

Settings

`'prefix'`

Default: `'u_'`

Specify the prefix as a character vector. HDL Coder appends the prefix 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`

MapPipelineDelaysToRAM

Optimize for area by mapping pipeline registers in generated HDL code to RAM

Settings

`'on'`

HDL Coder optimizes for area by mapping pipeline registers in the generated HDL code to RAM.

`'off'` (default)

HDL Coder does not map pipeline registers in the generated HDL code to RAM.

Usage Example

Use `hdlset_param` or `makehdl` to set this property.

For example, to map the pipeline registers in the generated HDL code to RAM for the `sfir_fixed/symmetric_fir` DUT subsystem, enter:

```
makehdl ('sfir_fixed/symmetric_sfir','MapPipelineDelaysToRAM','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

`RAMMappingThreshold`

Topics

"UseRAM"

“RAM Mapping”

MaskParameterAsGeneric

Generate reusable HDL code for subsystems with identical mask parameters that differ only in value

Settings

`'on'`

Generate one reusable HDL file for multiple masked subsystems with different values for the mask parameters. HDL Coder automatically detects 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

Topics

“Generate Reusable Code for Atomic Subsystems”
“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

Note `MaxComputationLatency` is not recommended. Use clock-rate pipelining with Oversampling instead.

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`.

MaxOversampling

Limit the maximum sample rate

Note `MaxOversampling` is not recommended. Use clock-rate pipelining with Oversampling instead.

Settings

inf (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`.

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 4-116). 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

Note If your design uses a RAM block such as a Dual Rate Dual Port RAM with the RAMArchitecture set to `WithoutClockEnable`, the code generator ignores the `MinimizeClockEnables` setting.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

MinimizeGlobalResets

Omit generation of reset logic in the HDL code

Settings

`'on'`

When you enable this setting, the code generator tries to minimize or remove the global reset logic from the HDL code. This code snippet shows the Verilog code generated for a Delay block in the Simulink model. The code snippet shows that HDL Coder removed the reset logic.

```
always @(posedge clk)
   begin : Delay_Synchronous_process
     if (enb) begin
       Delay_Synchronous_out1 <= DataIn;
     end
   end
```

`'off'` (default)

When you disable this parameter, HDL Coder generates the global reset logic in the HDL code. This Verilog code snippet shows the reset logic generated for the Delay block.

```
 always @(posedge clk or posedge reset)
   begin : Delay_Synchronous_process
     if (reset == 1'b1) begin
       Delay_Synchronous_out1 <= 1'b0;
     end
     else begin
       if (enb) begin
         Delay_Synchronous_out1 <= DataIn;
       end
     end
   end
```

Dependencies

If you enable `MinimizeGlobalResets`, the generated HDL code contains registers that do not have a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results. To avoid simulation mismatches, you can initialize the registers by using the NoResetInitializationMode property.

By default, `NoResetInitializationMode` generates an external script to initialize the registers. To initialize registers with the script, use a zero initial value for the blocks in your Simulink model. If these blocks have a non-zero initial value, to initialize the registers, set `NoResetInitializationMode` to `InsideModule`,

Exceptions

Sometimes, when you set `MinimizeGlobalResets` to `'on'`, HDL Coder generates the reset logic, if you have:

- Blocks with state that have a non-zero initial value, such as a Delay block with non-zero **Initial Condition**.
- Enumerated data types for blocks with state.
- Subsystem blocks with `BlackBox` HDL architecture where you request a reset signal.
- Multirate models with TimingControllerArch set to `default`.

  If you set `TimingControllerArch` to `resettable`, HDL Coder generates a reset port for the timing controller. If you set `MinimizeGlobalResets` to `'on'`, the code generator removes this reset port.
- Truth Table
- Chart
- MATLAB Function block

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 for DUT module or entity name

Settings

'*prefix name*'

Default: ''

Specify a prefix for every module or entity name in the generated HDL code as a character vector. 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 `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`.

MulticyclePathConstraints

Generate text file that reports multicycle path constraint information for use with synthesis tools

Settings

`'on'`

In multi-rate designs with a single clock signal, you can use enable-based multicycle path constraints to meet the timing requirements for data paths operating at a rate slower than the base rate. HDL Coder then generates a constraints file with the naming convention `dutname_constraints`. The file contains information about the clock multiples for calculating the setup and hold time information to meet the timing requirements.

`'off'` (default)

Do not generate a multicycle path constraints file.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

“Meet Timing Requirements Using Enable-Based Multicycle Path Constraints”

MulticyclePathInfo

Generate text file that reports multicycle path constraint information for use with synthesis tools

Settings

`'on'`

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 `'_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 at least one of the inputs to the multiplier has a bit width greater than the `MultiplierPartitioningThreshold`, the code generator 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`.

MultiplierPromotionThreshold

Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers

Settings

N

Default: 0

Maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.

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

Topics

“Resource Sharing”
“Multiplier promotion threshold”

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

Topics

"Resource Sharing"

NoResetInitializationMode

Specify initialization mode for registers without a reset port in the HDL code

Settings

`'Script'`

HDL Coder generates a script to initialize registers that do not have a reset port in the generated code.

`'None'`

HDL Coder does not initialize the registers without a reset port.

`'InsideModule'` (default)

HDL Coder initializes the registers that do not have a reset port as part of the HDL code generated for the DUT. In Verilog, an `initial` construct in the corresponding module definition initializes the no-reset registers. In VHDL, the initialization code is part of the signal declaration statements.

Usage Notes

If you have blocks with **ResetType** set to `none` in your Simulink model or specify the adaptive pipelining optimization, the generated HDL code can contain registers without a reset port. If you do not initialize these registers, there can be potential numerical mismatches in the HDL simulation results, because the registers are insensitive to the global reset logic. To avoid simulation mismatches, use the `NoResetInitializationMode` setting to initialize these registers in the generated code. For better simulation results, if you have registers without a reset port at the boundaries of the DUT, set InitializeTestBenchInputs to `on`. Setting this property provides an initial value for the data driven to the DUT, and initializes the registers with these values.

Functionality	Script	None value	InsideModule
Generated HDL code for DUT	The script is generated externally and does not affect the HDL code for the DUT.	HDL Coder does not initialize the registers in the generated code.	The code for initializing the registers is part of the HDL code for the DUT.
HDL simulator support	The syntax of the script is compliant with ModelSim 10.2c or later. Other HDL simulators or older ModelSim versions do not support the syntax of the initialization script. This mode does not support enumeration types, and initializing the registers with non zero values.	There can be numerical mismatches in the HDL simulation results, because this mode does not initialize the registers that do not have a reset port.	All HDL simulators support this initialization mode, and initialize the no-reset registers with appropriate values.
Synthesis tool support	As the script does not affect the HDL code generated for the DUT, all synthesis tools support this initialization mode.	Synthesis tools do not initialize the no-reset registers in this mode.	Later versions of synthesis tools support the initialization constructs in the generated code. However, it is possible that older versions do not synthesize the initialization constructs. To avoid such issues, make sure that synthesis tools can synthesize 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

- “No-reset registers initialization”
- ResetType
- “Adaptive Pipelining”

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 postfix to output assignment block labels for VHDL `GENERATE` statements

Settings

`'postfix'`

Default: `'outputgen'`

Specify the postfix as a character vector. HDL Coder appends this postfix 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 the postfix to append to specified model or subsystem name to form name of package file

Settings

`'postfix'`

Default: `'_pkg'`

Specify the postfix as a character vector. 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 postfix to names of input or output pipeline registers generated for pipelined block implementations

Settings

`'postfix'`

Default: `'_pipe'`

When you specify a generation of input and/or output pipeline registers for selected blocks, HDL Coder appends the postfix that is specified as a character vector 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`.

See Also

Topics

"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”
- “MapPersistentVarsToRAM”

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 postfix appended to identifiers for entities, signals, constants, or other model elements that conflict with VHDL or Verilog reserved words

Settings

`'postfix'`

Default: `'_rsvd'`.

Specify the postfix as a character vector. The reserved word postfix is applied to 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

'*Reset input*'

Default: 'reset'.

Specify the name for the model's reset input port as a character vector.

For example, if you override the default with '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 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

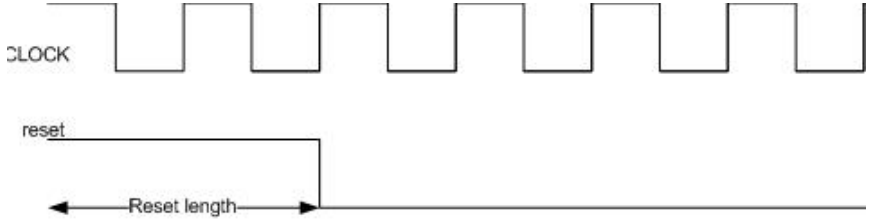
Settings

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

Topics

“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

Topics

"Resource Sharing"

ShareFloatingPointIP

Share floating-point IP blocks with the resource sharing optimization

Settings

`'on'` (default)

When you enable resource sharing, HDL Coder shares floating-point IP blocks. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify for the subsystem.

`'off'`

Do not share floating-point IP 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 | ShareMATLABBlocks | ShareMultipliers

Topics

“Resource Sharing”
“Floating-point IPs”

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

Topics

"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`. For successfully sharing multipliers, the input fixed-point data types must have the same wordlength. The fraction lengths and signs of the fixed-point data types can be different.

`'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

Topics

"Resource Sharing"

SimulationLibPath

Specify the path to the compiled Altera or Xilinx simulation libraries

Settings

'*Simulation library path*'

Default: ''

Specify the path to the compiled Altera or Xilinx simulation libraries. Altera provides the simulation model files in **\quartus\eda\sim_lib** folder.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

Usage Example

If you want to set the path to the compiled Xilinx Simulation library, enter:

```
myDUT = gcb;
libpath = '/apps/Xilinx_ISE/XilinxISE-13.4/Linux/ISE_DS/ISE/vhdl/
   mti_se/6.6a/lin64/xilinxcorelib';
hdlset_param (myDUT, 'SimulationLibPath', libpath);
```

See Also

Topics

"Simulation library path"

SimulationTool

Simulator for which the tool generates build-and-run scripts for the test bench and optional code coverage

Settings

`'Mentor Graphics ModelSim' | 'Cadence Incisive'|'Custom'`

Default: `'Mentor Graphics ModelSim'`

When you select `'Custom'`, the tool uses the custom script properties.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

SimulatorFlags

Specify simulator flags to apply to generated compilation scripts

Settings

`'compilation simulator flags'`

Default: `''`

Specify simulator flags to apply to generated compilation scripts as a character vector. The simulator flags are specific to your application and the simulator you are using. For example, if you must use the 1076–1993 VHDL compiler, specify the flag `-93`.

Usage Notes

The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command is specified by the `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 postfix to form name of file containing model VHDL architecture

Settings

'*postfix*'

Default: '_arch'.

Specify the postfix as a character vector. 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

`SplitEntityArch`, `SplitEntityFilePostfix`

SplitEntityArch

Specify whether generated VHDL entity and architecture code is written to single VHDL file or to separate files

Settings

`'on'`

Write the code for the generated VHDL entity and architecture to separate files.

`'off'`(default)

Write the generated VHDL code to a single file.

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 the postfix as a character vector.

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 postfix to form name of generated VHDL entity file

Settings

'*postfix*'

Default: '_entity'

Specify the postfix as a character vector. 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`

SynthesisProjectAdditionalFiles

Include additional HDL or constraint files in synthesis project

Settings

`''` (default)

Additional project files, such as HDL source files (.v, .vhd) or constraint files (.ucf), that you want to include in your synthesis project, specified as a character vector. Separate file names with a semicolon (;).

You cannot use `SynthesisProjectAdditionalFiles` to include Tcl files. To specify synthesis project Tcl files, use the `AdditionalProjectCreationTclFiles` property of the `hdlcoder.WorkflowConfig` object.

Usage

To include a source file, `src_file.vhd`, and a constraint file, `constraint_file.ucf`, in the synthesis proejct for a DUT subsystem, `myDUT`:

```
hdlset_param (myDUT, 'SynthesisProjectAdditionalFiles', ...
                            'L:\src_file.vhd;L:\constraint_file.ucf;')
```

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`hdlcoder.WorkflowConfig`

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

'*Chip family name*'

Default: ''

Specify the target device chip family name for your model as a character vector.

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

`'Device name'`

Default: `''`

Specify the target device name for your model as a character vector.

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

'*Package name*'

Default: ''

Specify the target device package name for your model as a character vector.

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

'*target speed*'

Default: ''

Specify the target device speed value for your model as a character vector.

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

`'folder'`

Default: `'hdlsrc'`

Specify a subfolder under the current working folder into which HDL Coder writes generated files. The folder name can be a complete path name, specified as a character vector.

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`.

TargetFrequency

Specify the target frequency for multiple features and workflows

Settings

N

Default: 0

This setting is the target frequency in MHz for multiple features and workflows that HDL Coder supports. The supported features are:

- FPGA floating-point target library mapping: Specify the target frequency that you want the IP to achieve when you use `ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)`. If you do not specify the target frequency, HDL Coder sets the target frequency to a default value of `200 MHz`.
- Adaptive pipelining: If your design uses multipliers, specify the synthesis tool and the target frequency. Based on these settings, HDL Coder estimates the number of pipelines that can be inserted to improve area and timing on the target platform. If you do not specify the target frequency, HDL Coder uses a target frequency of `0 MHz` and cannot insert pipelines.

To use this parameter, on the Configuration Parameters dialog box, in the **HDL Code Generation > Target and Optimizations** pane, for **Target Frequency (MHz)**, enter an integer. You can also set the target frequency by using the **Target Frequency (MHz)** setting in the **Set Target Frequency** task in the HDL Workflow Advisor.

Specify the target frequency for these workflows:

- `Generic ASIC/FPGA`: To specify the target frequency that you want your design to achieve. HDL Coder generates a timing constraint file for that clock frequency. It adds the constraint to the FPGA synthesis tool project that you create in the **Create Project** task. If the target frequency is not achievable, the synthesis tool generates an error.
- `IP Core Generation`: To specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that

frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.

- `Simulink Real-Time FPGA I/O`: For Speedgoat boards that are supported with `Xilinx ISE`, specify the target frequency to generate the clock module to produce the clock signal with that frequency.

  The Speedgoat boards that are supported with `Xilinx Vivado` use the `IP Core Generation` workflow infrastructure. Specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. Enter a target frequency value that is within the **Frequency Range (MHz)**. If you do not specify the target frequency, HDL Coder uses the **Default (MHz)** target frequency.
- `FPGA Turnkey`: To generate the clock module to produce the clock signal with that frequency.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

Topics

"Target Frequency"
"Generate HDL Code for FPGA Floating-Point Target Libraries"

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

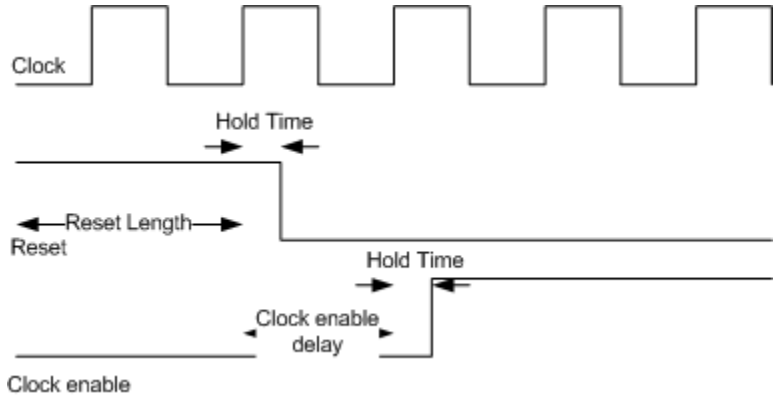
Settings

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

'*Data posfix*'

Default: '_data'.

Specify the postfix as a character vector. HDL Coder applies TestBenchDataPostFix only when generating a multi-file test bench (i.e. when MultifileTestBench is 'on').

For example, if the name of your DUT is my_test, and TestBenchPostFix has the default value _tb, the coder adds the postfix _data to form the test bench data file name my_test_tb_data.

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

See Also

MultifileTestBench, TestBenchPostFix

TestBenchPostFix

Specify suffix to test bench name

Settings

`'testbench suffix'`

Default: `'_tb'`.

Specify the suffix to testbench name as a character vector.

For example, if the name of your DUT is `my_test`, HDL Coder adds the postfix `_tb` to form the name `my_test_tb`.

Set or View This Property

To set this property, use `hdlset_param` or `makehdl`. To view the property value, use `hdlget_param`.

See Also

`MultifileTestBench`, `TestBenchDataPostFix`

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

Topics

“Generate Reset for Timing Controller”

TimingControllerPostfix

Specify suffix appended to DUT name to form timing controller name

Settings

`'DUT postfix'`

Default: `'_tc'`.

Specify the postfix as a character vector. 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 text appended to names of reference signals generated in test bench code

Settings

'*Signal name postfix*'

Default: '_ref'.

Reference signal data is represented as arrays in the generated test bench code. HDL Coder appends the character vector that TestBenchReferencePostFix specifies to the generated signal names.

Set or View This Property

To set this property, use hdlset_param or makehdl. To view the property value, use hdlget_param.

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`

TransformNonZeroInitDelay

Enable this property to optimize Delay blocks with non zero initial condition

Settings

`'on'` (default)

Transform Delay blocks with nonzero **Initial condition** in your Simulink model to Delay blocks with zero **Initial condition** and some additional logic in the generated HDL code.

By using this transformation, HDL Coder can perform optimizations such as sharing, distributed pipelining, and clock-rate pipelining more effectively, and prevent an assertion from being triggered in the validation model.

`'off'`

Do not transform Delay blocks with nonzero **Initial condition** in your Simulink model.

Usage Example

Use `hdlset_param` or `makehdl` to set this property.

For example, if you do not want to transform Delay blocks with nonzero **Initial condition** within the `sfir_fixed/symmetric_fir` subsystem, enter:

```
makehdl ('sfir_fixed/symmetric_sfir','TransformNonZeroInitDelay','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

Topics

“Transform non zero initial value delay”

TreatRealsInGeneratedCodeAs

Check for presence of reals in the generated HDL code

Settings

`'None'`

Do not check for reals in the generated HDL code.

`'Warning'` (default)

Reports a warning if the generated HDL code contains any real data types. Real data types in the generated HDL code are not synthesizable on target FPGA devices.

`'Error'`

Generates an error if there are any real data types in the generated HDL code. If your design contains floating-point data types, set the **Floating Point IP Library** as `Native Floating Point`, and then 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`.

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`.

See Also

Topics

"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

`'comment'`

Specify the comment as a character vector. 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, 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 prefix to vector names in generated code

Settings

`'prefix'`

Default: `'vector_of_'`

Specify the prefix as a character vector. HDL Coder appends this prefix 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`.

VerilogFileExtension

Specify file type extension for generated Verilog files

Settings

`'Verilog extension'`

Specify file type extension for generated Verilog files as a character vector. The default 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

`'Architecture name'`

Specify the architecture name for generated HDL code as a character vector. 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`.

See Also

Topics

"VHDL architecture name"

VHDLFileExtension

Specify file type extension for generated VHDL files

Settings

'*VHDL extension*'

Specify file type extension for generated VHDL files as a character vector. The default 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

`'Target library name'`

Specify name of target library for generated HDL code as a character vector. 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

5

Class reference for HDL code generation from Simulink

hdlcoder.FloatingPointTargetConfig class

Package: hdlcoder

Specify floating-point target configuration for floating-point library

Description

The `hdlcoder.FloatingPointTargetConfig` object sets options for HDL Coder to generate synthesizable floating-point code. To create an `hdlcoder.FloatingPointTargetConfig` object for a floating-point library, use the `hdlcoder.createFloatingPointTargetConfig` function. You can create a floating-point configuration object for these floating-point libraries:

- Native Floating Point
- Altera Megafunctions (ALTERA FP Functions)
- Altera Megafunctions (ALTFP)
- Xilinx LogiCORE®

Construction

`fpconfig = hdlcoder.createFloatingPointConfig(library)` creates an `hdlcoder.FloatingPointTargetConfig` object for a floating-point library.

`fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)` creates an `hdlcoder.FloatingPointTargetConfig` object with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name on page 5-3 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The name-value pair arguments that you can specify depend on the library that you select for creating the floating-point configuration.

Input Arguments

`Library` — Floating point library name

`None` (default) | `NATIVEFLOATINGPOINT` | `ALTERAFPFUNCTIONS` | `ALTFP` | `XILINXLOGICORE`

Floating-point library name, specified as a character vector

Example: `'ALTERAFPFUNCTIONS'`

Properties

Native Floating Point

`HandleDenormals` — Specify whether to handle denormal numbers in your design

`'off'` (default) | `'on'`

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a character vector. Denormal numbers are nonzero numbers that are smaller than the smallest normal number.

`LatencyStrategy` — Specify whether to use maximum or minimum latency for the native floating-point operator

`'MAX'` (default) | `'MIN'` | `'ZERO'`

Specify whether you want HDL Coder to use maximum or minimum latency setting for the floating-operators that your design uses. Specify this property as a character vector.

`MantissaMultiplyStrategy` — Specify how you want HDL Coder to implement the mantissa multiplication operation when your design uses floating-point multipliers

`'FullMultiplier'` (default) | `'PartMultiplierPartAddShift'` | `'NoMultiplierFullAddShift'`

Specify how you want HDL Coder to implement the mantissa multiplication process for floating-point multipliers in your design. With this option, you can control the DSP usage on the target platform for your design. To learn more, see “Mantissa Multiplier Strategy”.

Altera FP Functions

`InitializeIPPipelinesToZero` — Specify whether to initialize pipeline registers in the Altera Megafunction IP to zero
`true` (default) | `false`

Specify whether you want HDL Coder to initialize pipeline registers in the Altera Megafunction IP to zero. Specify this property as a `logical`. To avoid potential numerical mismatches in the HDL simulation, leave `InitializeIPPipelinesToZero` set to `true`.

ALTFP and Xilinx LogiCORE

`LatencyStrategy` — Specify whether to use maximum or minimum latency when mapping your design to FPGA floating-point target libraries
`'MIN'` (default) | `'MAX'`

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

`Objective` — Specify whether to optimize the design for speed or area when mapping your design to FPGA floating-point target libraries
`'SPEED'` (default) | `'AREA'`

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

Methods

createFloatingPointTargetConfig	Create floating-point target configuration for floating-point library that you specify

Examples

Create Floating-Point Configuration with Native Floating Point and Generate Code

This example shows how to create a floating-point target configuration with the native floating-point support in HDL Coder, and then generate code.

Create a Floating-Point Target Configuration

To create a floating-point configuration, use hdlcoder.createFloatingPointTargetConfig.

```
load_system('sfir_single');
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT')
```

```
fpconfig =

  FloatingPointTargetConfig with properties:

           Library: 'NativeFloatingPoint'
    LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
           IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.HandleDenormals = 'off';
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'NoMultiplierFullAddShift';
fpconfig.LibrarySettings
```

```
ans =

  NFPLatencyDrivenMode with properties:

            LatencyStrategy: 'MIN'
            HandleDenormals: 'off'
    MantissaMultiplyStrategy: 'NoMultiplierFullAddShift'
                     Version: '1.0.0'
```

View Latency of Native Floating Point Operators

The IPConfig object displays the maximum amd minimum latency values of the floating-point operators.

```
fpconfig.IPConfig


ans =

       Name              DataType              MaxLatency    MinLatency
    _________    ____________________________    __________    __________

    'ABS'          'SINGLE'                       0             0
    'ADDSUB'       'SINGLE'                      12             7
    'ATAN'         'SINGLE'                      36            36
    'ATAN2'        'SINGLE'                      42            42
    'CONVERT'      'NUMERICTYPE_TO_SINGLE'        6             6
    'CONVERT'      'SINGLE_TO_NUMERICTYPE'        6             6
    'COS'          'SINGLE'                      27            27
    'DIV'          'SINGLE'                      32            32
    'EXP'          'SINGLE'                      23            23
    'FIX'          'SINGLE'                       3             3
    'LOG'          'SINGLE'                      20            20
    'MINMAX'       'SINGLE'                       3             3
    'MOD'          'SINGLE'                       0             0
    'MUL'          'SINGLE'                       8             8
    'POW2'         'SINGLE'                       2             2
    'RECIP'        'SINGLE'                      19            19
    'RELOP'        'SINGLE'                       3             3
    'REM'          'SINGLE'                       0             0
    'ROUNDING'     'SINGLE'                       5             5
    'RSQRT'        'SINGLE'                      17            17
    'SIGNUM'       'SINGLE'                       0             0
    'SIN'          'SINGLE'                      27            27
    'SINCOS'       'SINGLE'                      27            27
    'SQRT'         'SINGLE'                      28            28
    'UMINUS'       'SINGLE'                       0             0
```

Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
                    'TargetDirectory','C:/NativeFloatingPoint/hdlsrc')

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additi
### The delay balancing feature has automatically inserted matching delays for compensa
### The DUT requires an initial pipeline setup latency. Each output port experiences th
```

```
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_comp as C:\NativeFloatingPoint\hdlsrc\
### Working on sfir_single/symmetric_fir/nfp_mul_comp as C:\NativeFloatingPoint\hdlsrc\
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\s
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg
### Creating HDL Code Generation Check Report file://C:\NativeFloatingPoint\hdlsrc\sfir
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated VHDL code is saved in the `hdlsrc` folder.

- "FPGA Floating-Point Library IP Mapping"
- "Single Precision Floating Point Support: Field-Oriented Control Algorithm"

See Also

ShareFloatingPointIP | `hdlcoder.FloatingPointTargetConfig.IPConfig` | `hdlcoder.FloatingPointTargetConfig.IPConfig.customize`

Topics

"FPGA Floating-Point Library IP Mapping"
"Single Precision Floating Point Support: Field-Oriented Control Algorithm"
"Generate HDL Code for FPGA Floating-Point Target Libraries"
"Customize Floating-Point IP Configuration"
"Generate Target-Independent HDL Code with Native Floating-Point"

Introduced in R2016b

createFloatingPointTargetConfig

Class: hdlcoder.FloatingPointTargetConfig
Package: hdlcoder

Create floating-point target configuration for floating-point library that you specify

Syntax

```
fpconfig = hdlcoder.createFloatingPointConfig(library)
fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)
```

Description

To create a floating-point target configuration object for a floating-point library, use the `hdlcoder.createFloatingPointTargetConfig` function. You can create a floating-point configuration object for these libraries:

- Native Floating Point
- Altera Megafunctions (ALTERA FP Functions)
- Altera Megafunctions (ALTFP)
- Xilinx LogiCORE

`fpconfig = hdlcoder.createFloatingPointConfig(library)` creates an `hdlcoder.FloatingPointTargetConfig` object for a given floating-point library.

`fpconfig = hdlcoder.createFloatingPointConfig(library,Name,Value)` creates an `hdlcoder.FloatingPointTargetConfig` object with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`Library` — Floating point library name

`None` (default) | `NATIVEFLOATINGPOINT` | `ALTERAFPFUNCTIONS` | `ALTFP` | `XILINXLOGICORE`

Floating-point library name, specified as a character vector

Example: `'ALTERAFPFUNCTIONS'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The name-value pair arguments that you can specify depend on the library that you select for creating the floating-point configuration.

Native Floating Point

`HandleDenormals` — Specify whether to handle denormal numbers in your design

`'off'` (default) | `'on'`

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a character vector. Denormal numbers are nonzero numbers that are smaller than the smallest normal number. To specify this property, for `Library`, select `NATIVEFLOATINGPOINT`.

`LatencyStrategy` — Specify whether to use maximum or minimum latency for the native floating-point operator

`'MAX'` (default) | `'MIN'` | `'ZERO'`

Specify whether you want HDL Coder to use maximum or minimum latency setting for the floating-operators that your design uses. Specify this property as a character vector. To specify this property, for `Library`, select `NATIVEFLOATINGPOINT`

`MantissaMultiplyStrategy` — Specify how you want HDL Coder to implement the mantissa multiplication operation when your design uses floating-point multipliers

`'FullMultiplier'` (default) | `'PartMultiplierPartAddShift'` | `'NoMultiplierFullAddShift'`

Specify how you want HDL Coder to implement the mantissa multiplication process for floating-point multipliers in your design. With this option, you can control the DSP usage on the target platform for your design. To learn more, see "Mantissa Multiplier Strategy".

Altera FP Functions

`InitializeIPPipelinesToZero` — Specify whether to initialize pipeline registers in the Altera Megafunction IP to zero

`true` (default) | `false`

Specify whether you want HDL Coder to initialize pipeline registers in the Altera Megafunction IP to zero. Specify this property as a `logical`. To avoid potential numerical mismatches in the HDL simulation, leave `InitializeIPPipelinesToZero` set to `true`. To specify this property, for `Library`, select `ALTERAFPFUNCTIONS`.

ALTFP and Xilinx LogiCORE

`LatencyStrategy` — Specify whether to use maximum or minimum latency when mapping your design to FPGA floating-point target libraries

`'MIN'` (default) | `'MAX'`

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

`Objective` — Specify whether to optimize the design for speed or area when mapping your design to FPGA floating-point target libraries

`'SPEED'` (default) | `'AREA'`

Specify whether you want the design to map to minimum or maximum latency with Xilinx LogiCORE or Altera Megafunction IP. Specify this property as a character vector.

Examples

Create Floating-Point Configuration with Native Floating Point and Generate Code

This example shows how to create a floating-point target configuration with the native floating-point support in HDL Coder, and then generate code.

Create a Floating-Point Target Configuration

To create a floating-point configuration, use hdlcoder.createFloatingPointTargetConfig.

```
load_system('sfir_single');
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT')
```

```
fpconfig =

  FloatingPointTargetConfig with properties:

           Library: 'NativeFloatingPoint'
    LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
           IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.HandleDenormals = 'off';
fpconfig.LibrarySettings.LatencyStrategy = 'MIN';
fpconfig.LibrarySettings.MantissaMultiplyStrategy = 'NoMultiplierFullAddShift';
fpconfig.LibrarySettings
```

```
ans =

  NFPLatencyDrivenMode with properties:

            LatencyStrategy: 'MIN'
            HandleDenormals: 'off'
    MantissaMultiplyStrategy: 'NoMultiplierFullAddShift'
                     Version: '1.0.0'
```

View Latency of Native Floating Point Operators

The IPConfig object displays the maximum amd minimum latency values of the floating-point operators.

```
fpconfig.IPConfig


ans =

       Name              DataType              MaxLatency    MinLatency
    _________    ____________________________    __________    __________

    'ABS'          'SINGLE'                       0             0
    'ADDSUB'       'SINGLE'                      12             7
    'ATAN'         'SINGLE'                      36            36
    'ATAN2'        'SINGLE'                      42            42
    'CONVERT'      'NUMERICTYPE_TO_SINGLE'        6             6
    'CONVERT'      'SINGLE_TO_NUMERICTYPE'        6             6
    'COS'          'SINGLE'                      27            27
    'DIV'          'SINGLE'                      32            32
    'EXP'          'SINGLE'                      23            23
    'FIX'          'SINGLE'                       3             3
    'LOG'          'SINGLE'                      20            20
    'MINMAX'       'SINGLE'                       3             3
    'MOD'          'SINGLE'                       0             0
    'MUL'          'SINGLE'                       8             8
    'POW2'         'SINGLE'                       2             2
    'RECIP'        'SINGLE'                      19            19
    'RELOP'        'SINGLE'                       3             3
    'REM'          'SINGLE'                       0             0
    'ROUNDING'     'SINGLE'                       5             5
    'RSQRT'        'SINGLE'                      17            17
    'SIGNUM'       'SINGLE'                       0             0
    'SIN'          'SINGLE'                      27            27
    'SINCOS'       'SINGLE'                      27            27
    'SQRT'         'SINGLE'                      28            28
    'UMINUS'       'SINGLE'                       0             0
```

Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
                    'TargetDirectory','C:/NativeFloatingPoint/hdlsrc')

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### The code generation and optimization options you have chosen have introduced additi
### The delay balancing feature has automatically inserted matching delays for compensa
### The DUT requires an initial pipeline setup latency. Each output port experiences th
```

```
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_comp as C:\NativeFloatingPoint\hdlsrc\
### Working on sfir_single/symmetric_fir/nfp_mul_comp as C:\NativeFloatingPoint\hdlsrc\
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\s
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg
### Creating HDL Code Generation Check Report file://C:\NativeFloatingPoint\hdlsrc\sfir
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated VHDL code is saved in the `hdlsrc` folder.

- "FPGA Floating-Point Library IP Mapping"
- "Single Precision Floating Point Support: Field-Oriented Control Algorithm"

See Also

ShareFloatingPointIP | `hdlcoder.FloatingPointTargetConfig.IPConfig` | `hdlcoder.FloatingPointTargetConfig.IPConfig.customize`

Topics

"FPGA Floating-Point Library IP Mapping"
"Single Precision Floating Point Support: Field-Oriented Control Algorithm"
"Generate HDL Code for FPGA Floating-Point Target Libraries"
"Customize Floating-Point IP Configuration"
"Generate Target-Independent HDL Code with Native Floating-Point"

hdlcoder.FloatingPointTargetConfig.IPConfig class

Package: hdlcoder

Specify IP settings for selected floating-point configuration

Description

Use the `hdlcoder.FloatingPointTargetConfig.IPConfig` object to see the list of supported IP blocks for a floating-point library. The IP configuration depends on the library settings. The library settings are specific to the floating-point library that you choose.

1. Create a floating-point target configuration object for the library.

   ```
   fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
   ```

2. To see the IP settings, use the `IPConfig` object.

   ```
   fpconfig.IPConfig
   ```

Optionally, to customize the IP configuration, use the `customize` method of the floating-point configuration object.

Construction

`fpconfig.IPConfig` shows the IP settings for the `fpconfig` floating-point target configuration that you create for the floating-point library.

Methods

customize	Customize IP configuration for specified floating-point library

Examples

Create and Customize Floating Point Configuration and Generate Code

This example shows how to create a floating-point target configuration with Altera® Megafunctions (ALTFP) in HDL Coder, and then generate code.

Create a Floating-Point Target Configuration

To create a floating-point configuration, use hdlcoder.createFloatingPointTargetConfig. Before creating a configuration, set up the path to your synthesis tool.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', ...
    'ToolPath', 'C:/Altera/16.0/quartus/bin64/quartus.exe');
load_system('sfir_single')
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP')

Prepending following Altera Quartus II path(s) to the system path:
C:\Altera\16.0\quartus\bin64

fpconfig =

  FloatingPointTargetConfig with properties:

           Library: 'ALTFP'
    LibrarySettings: [1×1 fpconfig.LatencyDrivenMode]
          IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
fpconfig.LibrarySettings.Objective = 'AREA';
fpconfig.LibrarySettings


ans =

  LatencyDrivenMode with properties:

    LatencyStrategy: 'MAX'
          Objective: 'AREA'
```

View Latency of Floating-Point IPs

The IPConfig object displays the maximum amd minimum latency values of the floating-point operators.

```
fpconfig.IPConfig
```

```
ans =

      Name              DataType              MinLatency    MaxLatency    Latency    Extr
    _________    ____________________________    __________    __________    _______    ____

    'ABS'        'DOUBLE'                       1             1             -1         ''
    'ABS'        'SINGLE'                       1             1             -1         ''
    'ADDSUB'     'DOUBLE'                       7            14             -1         ''
    'ADDSUB'     'SINGLE'                       7            14             -1         ''
    'CONVERT'    'DOUBLE_TO_NUMERICTYPE'        6             6             -1         ''
    'CONVERT'    'NUMERICTYPE_TO_DOUBLE'        6             6             -1         ''
    'CONVERT'    'NUMERICTYPE_TO_SINGLE'        6             6             -1         ''
    'CONVERT'    'SINGLE_TO_NUMERICTYPE'        6             6             -1         ''
    'COS'        'SINGLE'                      35            35             -1         ''
    'DIV'        'DOUBLE'                      10            61             -1         ''
    'DIV'        'SINGLE'                       6            33             -1         ''
    'EXP'        'DOUBLE'                      25            25             -1         ''
    'EXP'        'SINGLE'                      17            17             -1         ''
    'LOG'        'DOUBLE'                      34            34             -1         ''
    'LOG'        'SINGLE'                      21            21             -1         ''
    'MUL'        'DOUBLE'                      11            11             -1         ''
    'MUL'        'SINGLE'                      11            11             -1         ''
    'RECIP'      'DOUBLE'                      27            27             -1         ''
    'RECIP'      'SINGLE'                      20            20             -1         ''
    'RELOP'      'DOUBLE'                       1             3             -1         ''
    'RELOP'      'SINGLE'                       1             3             -1         ''
    'RSQRT'      'DOUBLE'                      36            36             -1         ''
    'RSQRT'      'SINGLE'                      26            26             -1         ''
    'SIN'        'SINGLE'                      36            36             -1         ''
    'SQRT'       'DOUBLE'                      30            57             -1         ''
    'SQRT'       'SINGLE'                      16            28             -1         ''
```

Customize Latency of ADDSUB IP

Using the customize method of the IPConfig object, you can customize the latency of the floating-point IP and specify any additional arguments.

```
fpconfig.IPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.IPConfig
```

```
ans =
```

Name	DataType	MinLatency	MaxLatency	Latency	Extr
'ABS'	'DOUBLE'	1	1	-1	''
'ABS'	'SINGLE'	1	1	-1	''
'ADDSUB'	'DOUBLE'	7	14	-1	''
'ADDSUB'	'SINGLE'	7	14	6	''
'CONVERT'	'DOUBLE_TO_NUMERICTYPE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_DOUBLE'	6	6	-1	''
'CONVERT'	'NUMERICTYPE_TO_SINGLE'	6	6	-1	''
'CONVERT'	'SINGLE_TO_NUMERICTYPE'	6	6	-1	''
'COS'	'SINGLE'	35	35	-1	''
'DIV'	'DOUBLE'	10	61	-1	''
'DIV'	'SINGLE'	6	33	-1	''
'EXP'	'DOUBLE'	25	25	-1	''
'EXP'	'SINGLE'	17	17	-1	''
'LOG'	'DOUBLE'	34	34	-1	''
'LOG'	'SINGLE'	21	21	-1	''
'MUL'	'DOUBLE'	11	11	-1	''
'MUL'	'SINGLE'	11	11	-1	''
'RECIP'	'DOUBLE'	27	27	-1	''
'RECIP'	'SINGLE'	20	20	-1	''
'RELOP'	'DOUBLE'	1	3	-1	''
'RELOP'	'SINGLE'	1	3	-1	''
'RSQRT'	'DOUBLE'	36	36	-1	''
'RSQRT'	'SINGLE'	26	26	-1	''
'SIN'	'SINGLE'	36	36	-1	''
'SQRT'	'DOUBLE'	30	57	-1	''
'SQRT'	'SINGLE'	16	28	-1	''

Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/FloatingPoint/hdlsrc','SynthesisToolChipFamily','Arria10'

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### Using C:\Altera\16.0\quartus\bin64\qmegawiz for the selected floating point IP libr
```

```
### The code generation and optimization options you have chosen have introduced additi
### The delay balancing feature has automatically inserted matching delays for compensa
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Generating Altera(R) megafunction: altfp_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sf
### Done.
### Generating Altera(R) megafunction: altfp_mul_single for latency of 11.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sf
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\FloatingPoint\hdlsrc\sfir_single\symmetr
### Generating package file C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\FloatingPoint\hdlsrc\sfir_singl
### HDL check for 'sfir_single' complete with 0 errors, 7 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

The generated VHDL code is saved in the `hdlsrc` folder.

- “FPGA Floating-Point Library IP Mapping”
- “Single Precision Floating Point Support: Field-Oriented Control Algorithm”

See Also

ShareFloatingPointIP | `hdlcoder.FloatingPointTargetConfig`

Topics

“FPGA Floating-Point Library IP Mapping”
“Single Precision Floating Point Support: Field-Oriented Control Algorithm”
“Generate HDL Code for FPGA Floating-Point Target Libraries”
“Customize Floating-Point IP Configuration”
“Generate Target-Independent HDL Code with Native Floating-Point”

Introduced in R2016b

customize

Class: hdlcoder.FloatingPointTargetConfig.IPConfig
Package: hdlcoder

Customize IP configuration for specified floating-point library

Syntax

```
fpconfig.IPConfig.customize(Name,DataType,Name,Value)
```

Description

`fpconfig.IPConfig.customize(Name,DataType,Name,Value)` customizes the `fpconfig` floating-point configuration with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Name` — Name of the floating-point IP

`''` (default) | character vector

Name of the floating-point IP to customize, specified as a character vector.

Example: `'ADDSUB'`

`DataType` — Data type of the floating-point IP

`''` (default) | character vector

Data type of the floating-point IP to customize, specified as a character vector.

Example: `'SINGLE'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

`Latency` — Latency of the floating-point IP
-1 (default) | positive integer

Specify a custom latency value for the floating-point IP as an integer.

Example: `fpconfig.IPConfig.customize('ADDSUB','Double','Latency', 6)` specifies a custom latency value of 6 for the ADDSUB IP.

`ExtraArgs` — Specify any additional arguments of the floating-point IP
'' (default) | character vector

Specify any additional arguments of the floating-point IP as a character vector.

Example: `fpconfig.IPConfig.customize('ADDSUB','Double','Latency', 6,'ExtraArgs','CSET c_mult_usage=Full_Usage')` specifies that you want to use DSP blocks on the target device.

Examples

Create and Customize Floating Point Configuration and Generate Code

This example shows how to create a floating-point target configuration with Altera® Megafunctions (ALTFP) in HDL Coder, and then generate code.

Create a Floating-Point Target Configuration

To create a floating-point configuration, use hdlcoder.createFloatingPointTargetConfig. Before creating a configuration, set up the path to your synthesis tool.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II', ...
    'ToolPath', 'C:/Altera/16.0/quartus/bin64/quartus.exe');
load_system('sfir_single')
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP')
```

```
Prepending following Altera Quartus II path(s) to the system path:
C:\Altera\16.0\quartus\bin64

fpconfig =
```

```
  FloatingPointTargetConfig with properties:

           Library: 'ALTFP'
    LibrarySettings: [1×1 fpconfig.LatencyDrivenMode]
           IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom Library Settings

Optionally, to customize the floating-point configuration, specify custom library settings.

```
fpconfig.LibrarySettings.LatencyStrategy = 'MAX';
fpconfig.LibrarySettings.Objective = 'AREA';
fpconfig.LibrarySettings
```

```
ans =

  LatencyDrivenMode with properties:

    LatencyStrategy: 'MAX'
          Objective: 'AREA'
```

View Latency of Floating-Point IPs

The IPConfig object displays the maximum amd minimum latency values of the floating-point operators.

```
fpconfig.IPConfig
```

```
ans =

      Name              DataType              MinLatency    MaxLatency    Latency    Extr
    _________    ____________________________    __________    __________    _______    ____

    'ABS'        'DOUBLE'                         1             1            -1         ''
    'ABS'        'SINGLE'                         1             1            -1         ''
    'ADDSUB'     'DOUBLE'                         7            14            -1         ''
    'ADDSUB'     'SINGLE'                         7            14            -1         ''
    'CONVERT'    'DOUBLE_TO_NUMERICTYPE'          6             6            -1         ''
    'CONVERT'    'NUMERICTYPE_TO_DOUBLE'          6             6            -1         ''
    'CONVERT'    'NUMERICTYPE_TO_SINGLE'          6             6            -1         ''
```

```
    'CONVERT'    'SINGLE_TO_NUMERICTYPE'     6            6           -1          ''
    'COS'        'SINGLE'                   35           35           -1          ''
    'DIV'        'DOUBLE'                   10           61           -1          ''
    'DIV'        'SINGLE'                    6           33           -1          ''
    'EXP'        'DOUBLE'                   25           25           -1          ''
    'EXP'        'SINGLE'                   17           17           -1          ''
    'LOG'        'DOUBLE'                   34           34           -1          ''
    'LOG'        'SINGLE'                   21           21           -1          ''
    'MUL'        'DOUBLE'                   11           11           -1          ''
    'MUL'        'SINGLE'                   11           11           -1          ''
    'RECIP'      'DOUBLE'                   27           27           -1          ''
    'RECIP'      'SINGLE'                   20           20           -1          ''
    'RELOP'      'DOUBLE'                    1            3           -1          ''
    'RELOP'      'SINGLE'                    1            3           -1          ''
    'RSQRT'      'DOUBLE'                   36           36           -1          ''
    'RSQRT'      'SINGLE'                   26           26           -1          ''
    'SIN'        'SINGLE'                   36           36           -1          ''
    'SQRT'       'DOUBLE'                   30           57           -1          ''
    'SQRT'       'SINGLE'                   16           28           -1          ''
```

Customize Latency of ADDSUB IP

Using the customize method of the IPConfig object, you can customize the latency of the floating-point IP and specify any additional arguments.

```
fpconfig.IPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.IPConfig
```

```
ans =

      Name              DataType              MinLatency    MaxLatency    Latency    Extr
    _________    ____________________________    __________    __________    _______    ____

    'ABS'        'DOUBLE'                    1            1           -1          ''
    'ABS'        'SINGLE'                    1            1           -1          ''
    'ADDSUB'     'DOUBLE'                    7           14           -1          ''
    'ADDSUB'     'SINGLE'                    7           14            6          ''
    'CONVERT'    'DOUBLE_TO_NUMERICTYPE'     6            6           -1          ''
    'CONVERT'    'NUMERICTYPE_TO_DOUBLE'     6            6           -1          ''
    'CONVERT'    'NUMERICTYPE_TO_SINGLE'     6            6           -1          ''
    'CONVERT'    'SINGLE_TO_NUMERICTYPE'     6            6           -1          ''
    'COS'        'SINGLE'                   35           35           -1          ''
    'DIV'        'DOUBLE'                   10           61           -1          ''
```

```
    'DIV'       'SINGLE'                6           33           -1        ''
    'EXP'       'DOUBLE'               25           25           -1        ''
    'EXP'       'SINGLE'               17           17           -1        ''
    'LOG'       'DOUBLE'               34           34           -1        ''
    'LOG'       'SINGLE'               21           21           -1        ''
    'MUL'       'DOUBLE'               11           11           -1        ''
    'MUL'       'SINGLE'               11           11           -1        ''
    'RECIP'     'DOUBLE'               27           27           -1        ''
    'RECIP'     'SINGLE'               20           20           -1        ''
    'RELOP'     'DOUBLE'                1            3           -1        ''
    'RELOP'     'SINGLE'                1            3           -1        ''
    'RSQRT'     'DOUBLE'               36           36           -1        ''
    'RSQRT'     'SINGLE'               26           26           -1        ''
    'SIN'       'SINGLE'               36           36           -1        ''
    'SQRT'      'DOUBLE'               30           57           -1        ''
    'SQRT'      'SINGLE'               16           28           -1        ''
```

Generate Code

```
makehdl('sfir_single/symmetric_fir','FloatingPointTargetConfiguration',fpconfig, ...
        'TargetDirectory','C:/FloatingPoint/hdlsrc','SynthesisToolChipFamily','Arria10'

### Generating HDL for 'sfir_single/symmetric_fir'.
### Starting HDL check.
### Using C:\Altera\16.0\quartus\bin64\qmegawiz for the selected floating point IP libr
### The code generation and optimization options you have chosen have introduced additi
### The delay balancing feature has automatically inserted matching delays for compensa
### The DUT requires an initial pipeline setup latency. Each output port experiences th
### Output port 0: 30 cycles.
### Output port 1: 30 cycles.
### Generating Altera(R) megafunction: altfp_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sf
### Done.
### Generating Altera(R) megafunction: altfp_mul_single for latency of 11.
### Found an existing generated file in a previous session: (C:\FloatingPoint\hdlsrc\sf
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\FloatingPoint\hdlsrc\sfir_single\symmetr
### Generating package file C:\FloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\FloatingPoint\hdlsrc\sfir_singl
### HDL check for 'sfir_single' complete with 0 errors, 7 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

The generated VHDL code is saved in the `hdlsrc` folder.

- “FPGA Floating-Point Library IP Mapping”

Tips

Before using this function, create a floating-point target configuration object for the floating-point library that you specify. Select library as `Altera Megafunctions (ALTERA FP FUNCTIONS)`, `Altera Megafunctions (ALTFP)`, or `Xilinx LogiCORE`.

This example creates a floating-point target configuration for the `Altera Megafunctions (ALTFP)` library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTFP');
```

See Also

ShareFloatingPointIP | `hdlcoder.FloatingPointTargetConfig`

Topics

“FPGA Floating-Point Library IP Mapping”
“Generate HDL Code for FPGA Floating-Point Target Libraries”
“Customize Floating-Point IP Configuration”

Introduced in R2016b

hdlcoder.WorkflowConfig class

Package: hdlcoder

Configure HDL code generation and deployment workflows

Description

Use the `hdlcoder.WorkflowConfig` object to set HDL workflow options for the `hdlcoder.runWorkflow` function. You can customize the `hdlcoder.WorkflowConfig` object for these workflows:

- Generic ASIC/FPGA
- FPGA Turnkey
- IP core generation

A best practice is to use the HDL Workflow Advisor to configure the workflow, and then export a workflow script. The commands in the workflow script create and configure the `hdlcoder.WorkflowConfig` object. See "Run HDL Workflow with a Script".

Construction

`hdlcoder.WorkflowConfig(Name,Value)` creates a workflow configuration object for you to specify your HDL code generation and deployment workflows, with additional options specified by one or more `Name,Value` pair arguments.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`SynthesisTool` — Synthesis tool name

`'Xilinx Vivado'` (default) | `'Altera QUARTUS II'` | `'Xilinx ISE'`

Name of the synthesis tool, specified as a character vector.

Example: `'SynthesisTool','Altera QUARTUS II'` creates a workflow configuration object with `'Altera QUARTUS II'` as the synthesis tool and `'Generic ASIC/FPGA'` as the target workflow.

`TargetWorkflow` — Specify the target workflow

`'Generic ASIC/FPGA'` (default) | `'FPGA Turnkey'` | `'IP Core Generation'`

Target workflow for HDL code generation, specified as a `character vector`.

Example: `'TargetWorkflow','IP Core Generation'` creates a workflow configuration object with `'Xilinx Vivado'` as the synthesis tool and `'IP Core Generation'` as the target workflow.

Properties

Generic ASIC/FPGA Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: `'project_file_folder'`

`Objective` — Synthesis tool objective

`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`

High-level synthesis tool objective, specified as one of these values.

`hdlcoder.Objective.None` (default)	Do not generate additional Tcl commands.
`hdlcoder.Objective.SpeedOptimized`	Generate synthesis tool Tcl commands to optimize for speed.
`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 task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`RunTaskVerifyWithHDLCosimulation` — Enable task to verify generated code with HDL cosimulation

`true` (default) | `false`

Enable or disable task to verify the generated code with HDL cosimulation, specified as a `logical`. This option takes effect only when `GenerateCosimulationModel` is `true`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Verify with HDL Cosimulation** task.

`RunTaskCreateProject` — Enable task to create synthesis tool project

`true` (default) | `false`

Enable or disable task to create a synthesis tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

`RunTaskPerformLogicSynthesis` — Enable task to launch synthesis tool and run logic synthesis

`true` (default) | `false`

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

`RunTaskPerformMapping` — Enable task to map synthesized logic to target device
`true` (default) | `false`

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`RunTaskPerformPlaceAndRoute` — Enable task to run place and route process
`true` (default) | `false`

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

`RunTaskRunSynthesis` — Enable task to launch Xilinx Vivado and run synthesis
`true` (default) | `false`

Enable or disable task to launch Xilinx Vivado and run synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Synthesis** task.

`RunTaskRunImplementation` — Enable task to launch Xilinx Vivado and run implementation
`true` (default) | `false`

Enable or disable task to launch Xilinx Vivado and run the implementation step, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Implementation** task.

`RunTaskAnnotateModelWithSynthesisResult` — Enable task to analyze timing information and highlight critical paths
`true` (default) | `false`

Enable or disable task to analyze pre- or post-routing timing information and highlight critical paths in your model, specified as a `logical`. This task is available only when the target workflow is `Generic ASIC/FPGA`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

`GenerateRTLCode` — Generate HDL code
`true` (default) | `false`

Option to generate HDL code in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`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 character vector. When `GenerateCosimulationModel` is `true`, this option is available.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`GenerateValidationModel` — Generate validation model
`false` (default) | `true`

Generate a validation model, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`AdditionalProjectCreationTclFiles` — Additional project creation Tcl files to include in your synthesis project
`''` (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: `'L:\file1.tcl;L:\file2.tcl;'`

`SkipPreRouteTimingAnalysis` — Skip pre-route timing analysis `logical`
`false` (default) | `true`

Skip pre-route timing analysis, specified as a `logical`. If your tool does not support early timing estimation, set to `true`.

When you enable this option, `CriticalPathSource` is set to `'post-route'`

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`IgnorePlaceAndRouteErrors` — Ignore place and route errors
`false` (default) | `true`

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Place and route** task.

`CriticalPathSource` — Critical path source
`'pre-route'` (default) | `'post-route'`

Critical path source, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **FPGA Synthesis and Analysis** > **Perform Synthesis and P/R** > **Perform Mapping** task.

`CriticalPathNumber` — Number of critical paths to annotate
1 (default) | 2 | 3

Number of critical paths to annotate, specified as a positive integer from 1 to 3.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **FPGA Synthesis and Analysis** > **Annotate Model with Synthesis Result** task.

`ShowAllPaths` — Show all critical paths
`false` (default) | `true`

Show all critical paths, including duplicate paths, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **FPGA Synthesis and Analysis** > **Annotate Model with Synthesis Result** task.

`ShowDelayData` — Annotate cumulative timing delay on each critical path
`true` (default) | `false`

Annotate the cumulative timing delay on each critical path, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor** > **FPGA Synthesis and Analysis** > **Annotate Model with Synthesis Result** task.

`ShowUniquePaths` — Show only the first instance of a critical path
`false` (default) | `true`

Show only the first instance of a critical path that is duplicated, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

`ShowEndsOnly` — Show only endpoints of each critical path

`false` (default) | `true`

Show the endpoints of each critical path, omitting connecting signal lines, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

FPGA Turnkey Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: `'project_file_folder'`

`Objective` — Synthesis tool objective

`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`

High-level synthesis tool objective, specified as one of these values.

`hdlcoder.Objective.None` (default)	Do not generate additional Tcl commands.
`hdlcoder.Objective.SpeedOptimized`	Generate synthesis tool Tcl commands to optimize for speed.
`hdlcoder.Objective.AreaOptimized`	Generate synthesis tool Tcl commands to optimize for area.
`hdlcoder.Objective.CompileOptimized`	Generate synthesis tool Tcl commands to optimize for compilation time.

If your synthesis tool is Xilinx ISE and your target workflow is Generic ASIC/FPGA or FPGA Turnkey, set the `Objective` to `hdlcoder.Objective.None`.

For the tool-specific Tcl commands that are added to the synthesis project creation Tcl script, see “Synthesis Objective to Tcl Command Mapping”.

`RunTaskGenerateRTLCode` — Enable task to generate RTL code and HDL top-level wrapper

`true` (default) | `false`

Enable or disable workflow task to generate RTL code and an HDL top-level wrapper, specified as a `logical`. When enabled, this task also generates a constraint file that contains pin mapping information and clock constraints.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code** task.

`RunTaskCreateProject` — Enable task to create synthesis tool project

`true` (default) | `false`

Enable or disable task to create a synthesis tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

`RunTaskPerformLogicSynthesis` — Enable task to launch synthesis tool and run logic synthesis

`true` (default) | `false`

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

`RunTaskPerformMapping` — Enable task to map synthesized logic to target device

`true` (default) | `false`

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`RunTaskPerformPlaceAndRoute` — Enable task to run place and route process
`true` (default) | `false`

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

`RunTaskRunSynthesis` — Enable task to launch Xilinx Vivado and run synthesis
`true` (default) | `false`

Enable or disable task to launch Xilinx Vivado and run synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Synthesis** task.

`RunTaskRunImplementation` — Enable task to launch Xilinx Vivado and run implementation
`true` (default) | `false`

Enable or disable task to launch Xilinx Vivado and run the implementation step, specified as a `logical`. This task is available only when your synthesis tool is Xilinx Vivado.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Implementation** task.

`RunTaskGenerateProgrammingFile` — Enable task to generate FPGA programming file
`true` (default) | `false`

Enable or disable task to generate an FPGA programming file, specified as a `logical`. This task is available only when the target workflow is `FPGA Turnkey`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Programming File** task.

`RunTaskProgramTargetDevice` — Enable task to program target device
`true` (default) | `false`

Enable or disable task to download the FPGA programming file to the target device, specified as a `logical`. This task is available only when the target workflow is `FPGA Turnkey`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Program Target Device** task.

`AdditionalProjectCreationTclFiles` — Additional project creation Tcl files to include in your synthesis project
`''` (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: `'L:\file1.tcl;L:\file2.tcl;'`

`SkipPreRouteTimingAnalysis` — Skip pre-route timing analysis `logical`
`false` (default) | `true`

Skip pre-route timing analysis, specified as a `logical`. If your tool does not support early timing estimation, set to `true`.

When this option is enabled, `CriticalPathSource` is set to `'post-route'`

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`IgnorePlaceAndRouteErrors` — Ignore place and route errors
`false` (default) | `true`

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

IP Core Generation Workflow

`ProjectFolder` — Folder for generated project files

`''` (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: `'project_file_folder'`

`RunTaskGenerateRTLCodeAndIPCore` — Enable task to generate code and IP core

`true` (default) | `false`

Enable or disable workflow task to generate code and IP core for embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor** > **HDL Code Generation** > **Generate RTL Code and IP Core** task.

`RunTaskCreateProject` — Enable task to create embedded system tool project

`true` (default) | `false`

Enable or disable workflow task to create an embedded system tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor** > **Embedded System Integration** > **Create Project** task.

`RunTaskGenerateSoftwareInterfaceModel` — Enable task to generate software interface model

`true` (default) | `false`

Enable or disable workflow task to generate a software interface model with IP core driver blocks for embedded C code generation, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor** > **Embedded System Integration** > **Generate Software Interface Model** task.

`RunTaskBuildFPGABitstream` — Enable task to generate bitstream for embedded system

`true` (default) | `false`

Enable or disable workflow task to generate a bitstream for the embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Build FPGA Bitstream** task.

`RunTaskProgramTargetDevice` — Enable task to program connected target device
`false` (default) | `true`

Enable or disable workflow task to program the connected target device, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Program Target Device** task.

`IPCoreRepository` — IP core repository folder path
`''` (default) | character vector

Full path to an IP core repository folder, specified as a character vector. The coder copies the generated IP core into the IP repository folder.

Example: `'L:\sandbox\work\IPfolder'`

`GenerateIPCoreReport` — Generate HTML documentation for the IP core
`true` (default) | `false`

Option to generate HTML documentation for the IP core, specified as a `logical`. For details, see “Custom IP Core Report”.

`Objective` — Synthesis tool objective
`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`

High-level synthesis tool objective, specified as one of these values.

`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".

`EnableIPCaching` — Create IP cache to reduce reference design synthesis time
`false` (default) | `true`

Enable or disable IP caching, specified as a `logical`. When you enable IP caching, the code generator creates an IP cache. You can reuse this cache in subsequent project runs, which reduces reference design synthesis time.

In the HDL Workflow Advisor, you can specify this setting in the **Create Project** task.

`OperatingSystem` — Operating system
`''` (default) | character vector

Operating system for embedded processor, specified as a character vector. The operating system is board-specific.

`AddLinuxDeviceDriver` — Add IP core device driver
`false` (default) | `true`

Option to insert the IP core node into the operating system device tree on the SD card on your board, specified as a `logical`. This option also restarts the operating system and adds the IP core driver as a loadable kernel module.

To use this option, your board must be connected.

`RunExternalBuild` — Run build process externally
`true` (default) | `false`

Option to run build process in parallel with MATLAB, specified as a `logical`. If this option is disabled, you cannot use MATLAB until the build is finished.

`TclFileForSynthesisBuild` — Use custom or default synthesis tool build script
`hdlcoder.BuildOption.Default` (default) | `hdlcoder.BuildOption.Custom`

Select whether to use a custom or default synthesis tool build script, specified as one of these values:

`hdlcoder.BuildOption.Default` (default)	Use the default build script.
`hdlcoder.BuildOption.Custom`	Use a custom build script instead of the default build script.

`CustomBuildTclFile` — Custom synthesis tool build script file

`''` (default) | character vector

Full path to a custom synthesis tool build Tcl script file, specified as a character vector. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project. If `TclFileForSynthesisBuild` is set to `hdlcoder.BuildOption.Custom`, you must specify a file.

If you want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, this Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

Example: `''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 with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex® 7 device and uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 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;

% 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 with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA Turnkey workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:24:32 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\turnkey_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA'

%% Load the Model
load_system('hdlcoderUARTServoControllerExample');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'HDLSubsystem', 'hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetDirectory', 'hdl_prj\hdlsrc');
```

```
hdlset_param('hdlcoderUARTServoControllerExample', ...
    'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoderUARTServoControllerExample', 'Workflow', 'FPGA Turnkey');

% Set Inport HDL parameters
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterface', 'RS-232 Serial Port Rx');
hdlset_param('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA/uart_rxd', ...
    'IOInterfaceMapping', '[0]');

% Set 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 Create Project Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set Properties related to Perform Mapping Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set Properties related to Perform Place and Route Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `turnkey_workflow_example.m`, at the command line, enter:

```
turnkey_workflow_example.m
```

Configure and Run IP Core Generation Workflow with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an IP core generation workflow script that targets the Altera Cyclone V SoC development kit and uses the Altera Quartus II synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 8.6 (R2015b) at 14:42:16 on 08/07/2015
% Parameter Values:
%     Filename  : 'S:\ip_core_gen_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoder_led_blinking/led_counter'

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters
hdlset_param('hdlcoder_led_blinking', ...
    'HDLSubsystem', 'hdlcoder_led_blinking/led_counter');
hdlset_param('hdlcoder_led_blinking', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_led_blinking', ...
    'ReferenceDesign', 'Default system (Qsys 14.0)');
hdlset_param('hdlcoder_led_blinking', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Altera QUARTUS II');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Cyclone V');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', '5CSXFC6D6F31C6');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_led_blinking', ...
    'TargetPlatform', 'Altera Cyclone V SoC development kit - Rev.D');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter', ...
```

```
    'ProcessorFPGASynchronization', 'Free running');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceMapping', 'x"100"');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction', ...
    'IOInterfaceMapping', 'x"104"');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'External Port');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', 'IOInterface', 'AXI4');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back', ...
    'IOInterfaceMapping', 'x"108"');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Altera QUARTUS II', ...
    'TargetWorkflow','IP Core Generation');

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterfaceModel = false;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskProgramTargetDevice = false;

% Set Properties related to Generate RTL Code And IP Core Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;

% Set Properties related to Create Project Task
hWC.Objective = hdlcoder.Objective.AreaOptimized;
```

```
% Set Properties related to Generate Software Interface Model Task
hWC.OperatingSystem = '';
hWC.AddLinuxDeviceDriver = false;

% Set Properties related to Build FPGA Bitstream Task
hWC.RunExternalBuild = true;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `ip_core_workflow_example.m`, at the command line, enter:

```
ip_core_gen_workflow_example.m
```

- "Run HDL Workflow with a Script"

See Also

Functions

`hdlcoder.runWorkflow`

Topics

"Run HDL Workflow with a Script"

Introduced in R2015b

export

Class: hdlcoder.WorkflowConfig
Package: hdlcoder

Generate MATLAB script that recreates the workflow configuration

Syntax

```
export(Name,Value)
```

Description

`export(Name,Value)` generates MATLAB commands that can recreate the current workflow configuration, with additional options specified by one or more `Name,Value` pair arguments.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside 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) | character vector

Full path to the exported MATLAB script file, specified as a character vector. If the path is empty, the MATLAB commands are displayed in the Command Window, but not saved in a file.

Example: `'L:\sandbox\work\hdlworkflow.m'`

`Overwrite` — Overwrite existing file

`false` (default) | `true`

Specify whether to overwrite the existing file as a `logical`.

`Comments` — Include comments
`true` (default) | `false`

Specify whether to include comments in the command list or script as a `logical`.

`Headers` — Include headers
`true` (default) | `false`

Specify whether to include a header in the command list or script as a `logical`.

`DUT` — Full path to DUT
`''` (default) | character vector

Full path to the DUT, specified as a character vector.

Example: `'hdlcoder_led_blinking/led_counter'`

See Also

Classes
`hdlcoder.WorkflowConfig`

Topics
"Run HDL Workflow with a Script"

Introduced in R2015b

setAllTasks

Class: hdlcoder.WorkflowConfig
Package: hdlcoder

Enable all tasks in workflow

Syntax

```
setAllTasks
```

Description

`setAllTasks` enables all workflow tasks in the `hdlcoder.WorkflowConfig` object.

If you do not want to enable each task individually, use this method. For example, if you want to run all tasks but one, you can run `hdlcoder.WorkflowConfig.setAllTasks`, then disable the task that you want to skip.

See Also

Functions
`hdlcoder.WorkflowConfig.clearAllTasks`

Classes
`hdlcoder.WorkflowConfig`

Topics
“Run HDL Workflow with a Script”

Introduced in R2015b

clearAllTasks

Class: hdlcoder.WorkflowConfig
Package: hdlcoder

Disable all tasks in workflow

Syntax

```
clearAllTasks
```

Description

`clearAllTasks` disables all workflow tasks in the `hdlcoder.WorkflowConfig` object.

If you do not want to disable each task individually, use this method. For example, if you want to run a single task, you can run `hdlcoder.WorkflowConfig.clearAllTasks`, then enable the task that you want to run.

See Also

Functions
`hdlcoder.WorkflowConfig.setAllTasks`

Classes
`hdlcoder.WorkflowConfig`

Topics
"Run HDL Workflow with a Script"

Introduced in R2015b

validate

Class: hdlcoder.WorkflowConfig
Package: hdlcoder

Check property values in HDL Workflow CLI configuration object

Syntax

```
validate
```

Description

`validate` verifies that the `hdlcoder.WorkflowConfig` object has acceptable values for all required properties, and that property values have valid data types. If validation fails, you get an error message.

See Also

`hdlcoder.WorkflowConfig`

Topics

"Run HDL Workflow with a Script"

Introduced in R2015b

hdlcoder.runWorkflow

Run HDL code generation and deployment workflow

Syntax

```
hdlcoder.runWorkflow(DUT)
hdlcoder.runWorkflow(DUT,workflow_config)
```

Description

`hdlcoder.runWorkflow(DUT)` runs the HDL code generation and deployment workflow with default workflow configuration settings.

`hdlcoder.runWorkflow(DUT,workflow_config)` runs the HDL code generation and deployment workflow according to the specified workflow configuration, `workflow_config`.

A best practice is to use the HDL Workflow Advisor to configure the workflow, then export a workflow script. The commands in the workflow script create and configure a workflow configuration object that matches the settings in the HDL Workflow Advisor. The script includes the `hdlcoder.runWorkflow` command. To learn more, see "Run HDL Workflow with a Script".

Examples

Run Workflow with Configuration Object

This example is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex-7 device. It uses the Xilinx Vivado synthesis tool. The example generates HDL code for the `sfir_fixed` model, and performs FPGA synthesis and analysis.

Before running the Workflow

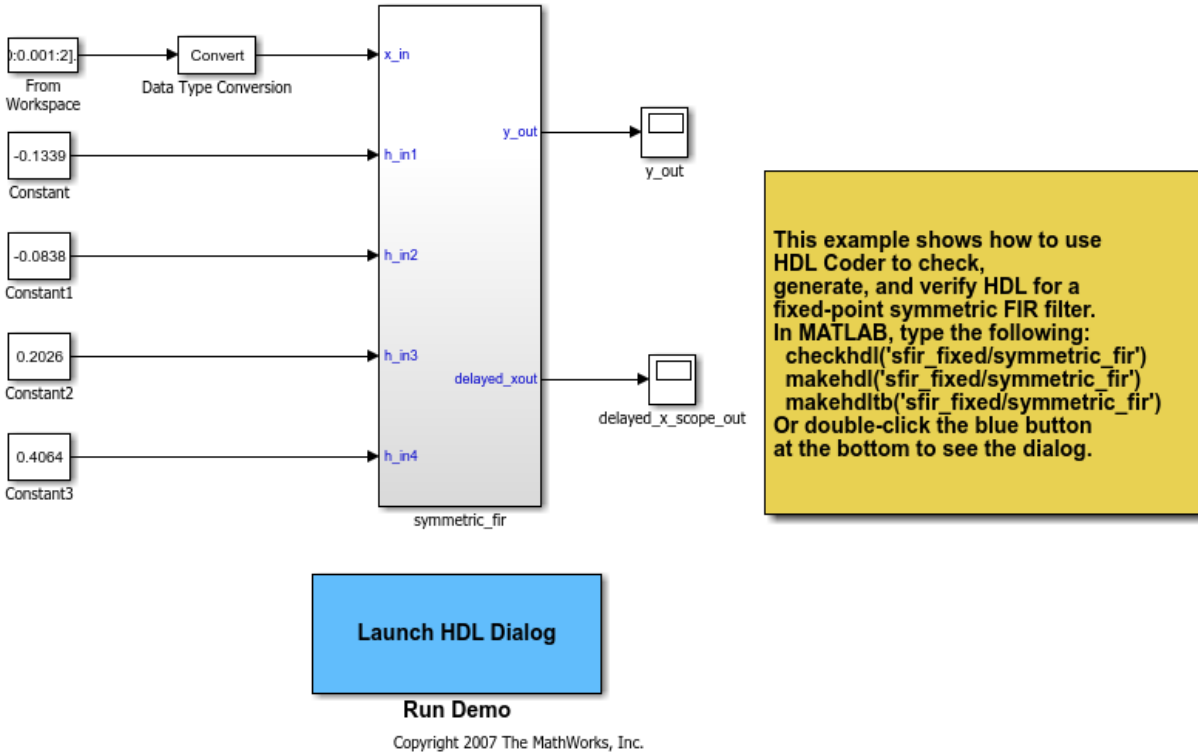
Before running the workflow, you must have the synthesis tool installed. Use hdlsetuptoolpath to specify the path to your synthesis tool.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
'L:\Xilinx\Vivado\2016.2\bin\vivado.bat');

Prepending following Xilinx Vivado path(s) to the system path:
L:\Xilinx\Vivado\2016.2\bin
```

Specify the model for running the workflow

To run the HDL workflow with default settings for a DUT subsystem, `modelname/DUT`, at the command line, enter:

```
open_system('sfir_fixed');
```



Model HDL Parameters

Set Model HDL parameters

```
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');
```

Workflow Configuration Settings

- Construct the Workflow Configuration Object with default settings
- Specify the path to your project folder. This step is optional

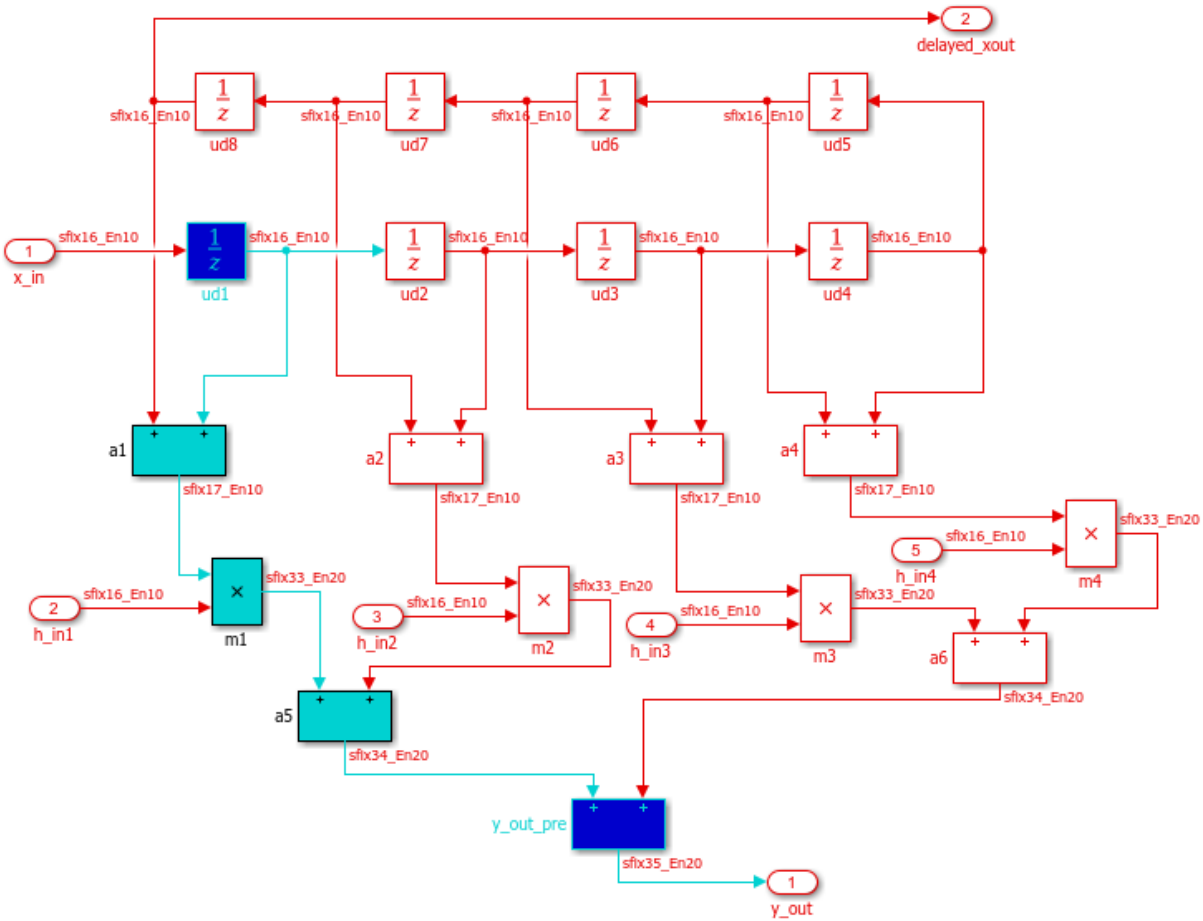
```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado', ...
    'TargetWorkflow','Generic ASIC/FPGA');

hWC.ProjectFolder = 'C:/Temp/hdl_prj';
```

Run the workflow

```
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

```
### Workflow begin.
### Loading settings from model.
### +++++++++++++++ Task Generate RTL Code and Testbench +++++++++++++++
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\Temp\hdl_prj\hdlsrc\sfir_fixed\symmetric_
### Creating HDL Code Generation Check Report file://C:\Temp\hdl_prj\hdlsrc\sfir_fixed\
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
### +++++++++++++++ Task Create Project +++++++++++++++
### Generating Xilinx Vivado 2016.2 project: <a href="matlab:downstream.tool.openTarget
### Generated logfile: <a href="matlab:edit('C:\Temp\hdl_prj\hdlsrc\sfir_fixed\workflow
### Task "Create Project" successful.
### +++++++++++++++ Task Run Synthesis +++++++++++++++
### Generated logfile: <a href="matlab:edit('C:\Temp\hdl_prj\hdlsrc\sfir_fixed\workflow
### Task "Run Synthesis" successful.
### +++++++++++++++ Task Annotate Model with Synthesis Result +++++++++++++++
### Parsing the timing file...
### Matched Source = 'sfir_fixed/symmetric_fir/ud1_out1'
### Matched Destination = 'sfir_fixed/symmetric_fir/y_out'
```

```
### Highlighting CP 1 from 'sfir_fixed/symmetric_fir/ud1_out1' to 'sfir_fixed/symmetric
### Click <a href="matlab:hdlannotatepath('reset')">here</a> to reset highlighting.
### Workflow complete.
```



- “Run HDL Workflow with a Script”

Input Arguments

`DUT` — Full path to DUT

`''` (default) | character vector

Full path to the DUT, specified as a character vector.

Example: `'hdlcoder_led_blinking/led_counter'`

`workflow_config` — Workflow configuration

`hdlcoder.WorkflowConfig`

HDL code generation and deployment workflow configuration, specified as an `hdlcoder.WorkflowConfig` object.

See Also

Functions

`hdlcoder.WorkflowConfig.clearAllTasks` | `hdlcoder.WorkflowConfig.setAllTasks`

Classes

`hdlcoder.WorkflowConfig`

Topics

"Run HDL Workflow with a Script"

Introduced in R2015b

hdlcoder.OptimizationConfig class

Package: hdlcoder

`hdlcoder.optimizeDesign` configuration object

Description

Use the `hdlcoder.OptimizationConfig` object to set options for the `hdlcoder.optimizeDesign` function.

Maximum Clock Frequency Configuration

To configure `hdlcoder.optimizeDesign` to maximize the clock frequency of your design:

- Set `ExplorationMode` to `hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`.
- Set `ResumptionPoint` to the default, `''`.

You can optionally set `IterationLimit` and `TestbenchGeneration` to nondefault values. HDL Coder ignores the `TargetFrequency` setting.

Target Clock Frequency Configuration

To configure `hdlcoder.optimizeDesign` to meet a target clock frequency:

- Set `ExplorationMode` to `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`.
- Set `TargetFrequency` to your target clock frequency.
- Set `ResumptionPoint` to the default, `''`

You can optionally set `IterationLimit` and `TestbenchGeneration` to nondefault values.

Resume From Interruption Configuration

To configure `hdlcoder.optimizeDesign` to resume after an interruption, specify `ResumptionPoint`.

When you set `ResumptionPoint` to a nondefault value, the other properties are ignored.

Construction

optimcfg = `hdlcoder.OptimizationConfig` creates an `hdlcoder.OptimizationConfig` object for automatic iterative HDL design optimization.

Properties

`ExplorationMode` — Optimization target mode

`hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency` (default) | `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`

Optimization target mode, specified as one of these values:

`hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`	Optimizes the design to try to achieve the maximum clock frequency `hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency` is the default.
`hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`	Optimizes the design to try to achieve the specified target clock frequency

`IterationLimit` — Maximum number of iterations

1 (default) | positive integer

Maximum number of optimization iterations before exiting, specified as a positive integer.

If `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.BestFrequency`, HDL Coder runs this number of iterations.

If `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`, HDL Coder runs the number of iterations needed to meet the target frequency. Otherwise, the coder runs the maximum number of iterations.

`ResumptionPoint` — Folder containing optimization data from earlier iteration

`''` (default) | character vector

Name of folder that contains previously-generated optimization iteration data, specified as a character vector. The folder is a subfolder of `hdlexpl`, and the folder name begins with the character vector, `Iter`.

When you set `ResumptionPoint` to a nondefault value, `hdlcoder.optimizeDesign` ignores the other configuration object properties.

Example: `'Iter1-26-Sep-2013-10-19-13'`

`TargetFrequency` — Target clock frequency

`Inf` (default) | double

Target clock frequency, specified as a double in MHz. Specify when `ExplorationMode` is `hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency`.

Examples

Configure hdlcoder.optimizeDesign for maximum clock frequency

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model,'HDLSubsystem',[model,'/',dutSubsys]);
```

Set your synthesis tool and target device options.

```
hdlset_param(model,'SynthesisTool','Xilinx ISE', ...
                   'SynthesisToolChipFamily','Zynq', ...
                   'SynthesisToolDeviceName','xc7z030', ...
                   'SynthesisToolPackageName','fbg484', ...
                   'SynthesisToolSpeedValue','-3')
```

Enable HDL test bench generation.

```
hdlset_param(model,'GenerateHDLTestBench','on');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Set the iteration limit to 10.

```
oc.IterationLimit = 10;
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');

Iteration 0
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 143.66 Iteration 1
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 278.72 Iteration 2
Generate and synthesize HDL code ...
(CP ns) 10.25    (Constraint ns) 12.73    (Elapsed s) 427.22 Iteration 3
Generate and synthesize HDL code ...
(CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37 Iteration 4
Generate and synthesize HDL code ...
(CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04 Iteration 5
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 9.55 ns        Elapsed : 741.04 s
```

```
Iteration 0: (CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 143.66
Iteration 1: (CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 278.72
Iteration 2: (CP ns) 10.25    (Constraint ns) 12.73    (Elapsed s) 427.22
Iteration 3: (CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-04-41
Validation model: gm_sfir_fixed_vnl
```

Then HDL Coder stops after five iterations because the fourth and fifth iterations had the same critical path, which indicates that the coder has found the minimum critical path. The design's maximum clock frequency after optimization is 1 / 9.55 ns, or 104.71 MHz.

Configure hdlcoder.optimizeDesign for target clock frequency

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
open_system(model);
hdlset_param(model,'HDLSubsystem',[model,'/',dutSubsys]);
```

Set your synthesis tool and target device options.

```
hdlset_param(model,'SynthesisTool','Xilinx ISE', ...
                   'SynthesisToolChipFamily','Zynq', ...
                   'SynthesisToolDeviceName','xc7z030', ...
                   'SynthesisToolPackageName','fbg484', ...
                   'SynthesisToolSpeedValue','-3')
```

Disable HDL test bench generation.

```
hdlset_param(model,'GenerateHDLTestBench','off');
```

Save your model.

You must save your model if you want to regenerate code later without rerunning the iterative optimizations, or resume your run if it is interrupted. When you use `hdlcoder.optimizeDesign` to regenerate code or resume an interrupted run, HDL Coder checks the model checksum and generates an error if the model has changed.

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to stop after it reaches a clock frequency of 50MHz, or 10 iterations, whichever comes first.

```
oc.ExplorationMode = ...
    hdlcoder.OptimizationConfig.ExplorationMode.TargetFrequency;
oc.TargetFrequency = 50;
oc.IterationLimit = 10; =
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed','GenerateHDLTestBench','off');
hdlset_param('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed','SynthesisTool','Xilinx ISE');
hdlset_param('sfir_fixed','SynthesisToolChipFamily','Zynq');
hdlset_param('sfir_fixed','SynthesisToolDeviceName','xc7z030');
hdlset_param('sfir_fixed','SynthesisToolPackageName','fbg484');
hdlset_param('sfir_fixed','SynthesisToolSpeedValue','-3');

Iteration 0
Generate and synthesize HDL code ...
(CP ns) 16.26    (Constraint ns) 20.00    (Elapsed s) 134.02 Iteration 1
Generate and synthesize HDL code ...
Exiting because constraint (20.00 ns) has been met (16.26 ns).
Summary report: summary.html
Achieved Critical Path (CP) Latency : 16.26 ns        Elapsed : 134.02 s
Iteration 0: (CP ns) 16.26    (Constraint ns) 20.00    (Elapsed s) 134.02
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-14
Validation model: gm_sfir_fixed_vnl
```

Then HDL Coder stops after one iteration because it has achieved the target clock frequency. The critical path is 16.26 ns, a clock frequency of 61.50 GHz.

Configure hdlcoder.optimizeDesign to resume from interruption

Open the model and specify the DUT subsystem.

```
model = 'sfir_fixed';
dutSubsys = 'symmetric_fir';
```

```
open_system(model);
hdlset_param(model,'HDLSubsystem',[model,'/',dutSubsys]);
```

Set your synthesis tool and target device options to the same values as in the interrupted run.

```
hdlset_param(model,'SynthesisTool','Xilinx ISE', ...
                   'SynthesisToolChipFamily','Zynq', ...
                   'SynthesisToolDeviceName','xc7z030', ...
                   'SynthesisToolPackageName','fbg484', ...
                   'SynthesisToolSpeedValue','-3')
```

Enable HDL test bench generation.

```
hdlset_param(model,'GenerateHDLTestBench','on');
```

Create an optimization configuration object, `oc`.

```
oc = hdlcoder.OptimizationConfig;
```

Configure the automatic iterative optimization to run using data from the first iteration of a previous run.

```
oc.ResumptionPoint = 'Iter5-07-Jan-2014-17-04-29';
```

Optimize the model.

```
hdlcoder.optimizeDesign(model,oc)
```

```
hdlset_param('sfir_fixed','HDLSubsystem','sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed','SynthesisTool','Xilinx ISE');
hdlset_param('sfir_fixed','SynthesisToolChipFamily','Zynq');
hdlset_param('sfir_fixed','SynthesisToolDeviceName','xc7z030');
hdlset_param('sfir_fixed','SynthesisToolPackageName','fbg484');
hdlset_param('sfir_fixed','SynthesisToolSpeedValue','-3');

Try to resume from resumption point: Iter5-07-Jan-2014-17-04-29
Iteration 5
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 9.55 ns        Elapsed : 741.04 s
Iteration 0: (CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 143.66
Iteration 1: (CP ns) 16.26    (Constraint ns) 5.85    (Elapsed s) 278.72
Iteration 2: (CP ns) 10.25    (Constraint ns) 12.73    (Elapsed s) 427.22
```

```
Iteration 3: (CP ns) 9.55    (Constraint ns) 9.73    (Elapsed s) 584.37
Iteration 4: (CP ns) 9.55    (Constraint ns) 9.38    (Elapsed s) 741.04
Final results are saved in
    /tmp/hdlsrc/sfir_fixed/hdlexpl/Final-07-Jan-2014-17-07-30
Validation model: gm_sfir_fixed_vnl
```

Then coder stops after one additional iteration because it has achieved the target clock frequency. The critical path is 9.55 ns, or a clock frequency of 104.71 MHz.

See Also

`hdlcoder.optimizeDesign`

6

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

character vector

Name of top-level MATLAB function for which you want to generate HDL code.

`fixptcfg` — Floating-point to fixed-point conversion configuration

`coder.FixptConfig`

Floating-point to fixed-point conversion configuration options, specified as a `coder.FixptConfig` object.

Use `fixptcfg` when generating HDL code from floating-point MATLAB code. Create a `coder.FixptConfig` object using the HDL `coder.config` function.

See Also

`coder.FixptConfig` | `coder.HdlConfig` | `coder.config`

Topics

"Generate HDL Code from MATLAB Code Using the Command Line Interface"

Introduced in R2013a

coder.approximation

Create function replacement configuration object

Syntax

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

Description

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the`codegen` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

Examples

Replace `log` Function with Default Lookup Table

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```

Replace `log` Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...
'FunctionNamePrefix','log_replace_');
```

Replace `log` Function with Optimized Lookup Table

Create a function replacement configuration object using the `'OptimizeLUTSize'` option to specify to replace the `log` function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
    logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...
'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, `saturateExp`, with an optimized lookup table.

Create a custom function, `saturateExp`.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the `saturateExp` function with an optimized lookup table. Because the `saturateExp` function is not listed as a function for which `coder.approximation` can generate an approximation automatically, you must specify the `CandidateFunction` property.

```
saturateExp = @(x) 1/(1+exp(-x));
custAppx = coder.approximation('Function','saturateExp',...
'CandidateFunction', saturateExp,...
'NumberOfPoints',50,'InputRange',[0,10]);
```

- “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`

See Also

Classes

`coder.FixptConfig`

Functions

`codegen`

Topics

"Replace the exp Function with a Lookup Table"

"Replace a Custom Function with a Lookup Table"
"Replacing Functions Using Lookup Table Approximations"

Introduced in R2014b

coder.config

Create HDL Coder code generation configuration objects

Syntax

```
config_obj = coder.config('hdl')
config_obj = coder.config('fixpt')
```

Description

`config_obj = coder.config('hdl')` creates a `coder.HdlConfig` configuration object for use with the HDL `codegen` function when generating HDL code from MATLAB code.

`config_obj = coder.config('fixpt')` creates a `coder.FixptConfig` configuration object for use with the HDL `codegen` function when generating HDL code from floating-point MATLAB code. The `coder.FixptConfig` object configures the floating-point to fixed-point conversion.

Examples

Generate HDL Code from Floating-Point MATLAB Code

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Create a `coder.HdlConfig` object, `hdlcfg`, with default settings.

```
hdlcfg = coder.config('hdl');
```

Convert your floating-point MATLAB design to fixed-point, and generate HDL code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

- "Generate HDL Code from MATLAB Code Using the Command Line Interface"

See Also

`codegen` | `coder.FixptConfig` | `coder.HdlConfig`

Topics

"Generate HDL Code from MATLAB Code Using the Command Line Interface"

Introduced in R2013a

addDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Add design range specification to parameter

Syntax

```
addDesignRangeSpecification(fcnName,paramName,designMin, designMax)
```

Description

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)` specifies the minimum and maximum values allowed for the parameter, `paramName`, in function, `fcnName`. The fixed-point conversion process uses this design range information to derive ranges for downstream variables in the code.

Input Arguments

`fcnName` — Function name

string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name

string

Parameter name, specified as a string.

Data Types: `char`

`designMin` — Minimum value allowed for this parameter

scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

See Also

addFunctionReplacement

Class: coder.FixptConfig
Package: coder

Replace floating-point function with fixed-point function during fixed-point conversion

Syntax

```
addFunctionReplacement(floatFn,fixedFn)
```

Description

`addFunctionReplacement(floatFn,fixedFn)` specifies a function replacement in a `coder.FixptConfig` object. During floating-point to fixed-point conversion in the HDL code generation workflow, the conversion process replaces the specified floating-point function with the specified fixed-point function. The fixed-point function must be in the same folder as the floating-point function or on the MATLAB path.

Input Arguments

`floatFn` — Name of floating-point function

`''` (default) | string

Name of floating-point function, specified as a string.

`fixedFn` — Name of fixed-point function

`''` (default) | string

Name of fixed-point function, specified as a string.

Examples

Specify Function Replacement in Fixed-Point Conversion Configuration Object

Create a fixed-point code configuration object, `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 generator replaces instances of `min` with `fi_min` during floating-point to fixed-point conversion.

Alternatives

You can specify function replacements in the HDL Workflow Advisor. See “Function Replacements”.

See Also

`codegen` | `coder.FixptConfig` | `coder.config`

clearDesignRangeSpecifications

Class: coder.FixptConfig
Package: coder

Clear all design range specifications

Syntax

```
clearDesignRangeSpecifications()
```

Description

`clearDesignRangeSpecifications()` clears all design range specifications.

Examples

Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

getDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Get design range specifications for parameter

Syntax

```
[designMin, designMax] = getDesignRangeSpecification(fcnName,
paramName)
```

Description

`[designMin, designMax] = getDesignRangeSpecification(fcnName, paramName)` gets the minimum and maximum values specified for the parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Function name

string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name

string

Parameter name, specified as a string.

Data Types: `char`

Output Arguments

`designMin` — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Get Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the  design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')
```

```
designMin =

    -1


designMax =

     1
```

See Also

hasDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Determine whether parameter has design range

Syntax

```
hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)
```

Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)` returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

Input Arguments

`fcnName` — Name of function

string

Function name, specified as a string.

Example: 'dti'

Data Types: `char`

`paramName` — Parameter name

string

Parameter name, specified as a string.

Example: 'dti'

Data Types: `char`

Output Arguments

`hasDesignRange` — Parameter has design range
true | false

Parameter has design range, returned as a boolean.

Data Types: `logical`

Examples

Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

```
hasDesignRanges =

     1
```

See Also

removeDesignRangeSpecification

Class: coder.FixptConfig
Package: coder

Remove design range specification from parameter

Syntax

```
removeDesignRangeSpecification(fcnName,paramName)
```

Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Name of function

string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name

string

Parameter name, specified as a string.

Data Types: `char`

Examples

Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

7

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.HdlConfig` | `coder.config`

Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”

coder.HdlConfig class

Package: coder

HDL `codegen` configuration object

Description

A `coder.HdlConfig` object contains the configuration parameters that the HDL `codegen` function requires to generate HDL code. Use the `-config` option to pass this object to the `codegen` function.

Construction

`hdlcfg` `= coder.config('hdl')` creates a `coder.HdlConfig` object for HDL code generation.

Properties

Basic

`AdderSharingMinimumBitwidth`

Minimum bit width for shared adders, specified as a positive integer.

If `ShareAdders` is `true` and `ResourceSharing` is greater than 1, share adders only if adder bit width is greater than or equal to `AdderSharingMinimumBitwidth`.

Values: integer greater than or equal to 2

`ClockEdge`

Specify active clock edge.

Values: `'Rising'` (default) | `'Falling'`

`DistributedPipeliningPriority`

Priority for distributed pipelining algorithm.

`DistributedPipeliningPriority` Value	Description
`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 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. Generates a compliance report showing errors, warnings, and messages.

Values: `'None'` (default) | `'Industry'`

HDLCodingStandardCustomizations

HDL coding standard rules and report customizations, specified using HDL coding standard customization. If you want to customize the coding standard rules and report, you must set `HDLCodingStandard` to `'Industry'`.

Value: HDL coding standard customization object

HDLLintTool

HDL lint tool script to generate.

Values: `'None'` (default) | `'AscentLint'` | `'Leda'` | `'SpyGlass'` |`'Custom'`

HDLLintInit

HDL lint script initialization name, specified as a character vector.

HDLLintCmd

HDL lint script command.

If you set `HDLLintTool` to `Custom`, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script. Specify `HDLLintCmd` as a character vector using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

HDLLintTerm

HDL lint script termination name, specified as a character vector.

InitializeBlockRAM

Specify whether to initialize all block RAM to `'0'` for simulation.

Values: `true` (default) | `false`

InlineConfigurations

Specify whether to include inline configurations in generated VHDL code.

When `true`, include VHDL configurations in files that instantiate a component.

When `false`, suppress the generation of configurations and require user-supplied external configurations. Set to `false` if you are creating your own VHDL configuration files.

Values: `true` (default) | `false`

`LoopOptimization`

Loop optimization in generated code. See "Optimize MATLAB Loops".

LoopOptimization Value	Description
`LoopNone` (default)	Do not optimize loops in generated code.
`StreamLoops`	Stream loops.
`UnrollLoops`	Unroll Loops.

`MinimizeClockEnables`

Specify whether to omit generation of clock enable logic.

When `true`, omit generation of clock enable logic wherever possible.

When `false` (default), generate clock enable logic.

`MultiplierPartitioningThreshold`

Specify maximum input bit width for hardware multipliers. If a multiplier input bit width is greater than this threshold, HDL Coder splits the multiplier into smaller multipliers.

To improve your hardware mapping results, set this threshold to the input bit width of the DSP or multiplier hardware on your target device.

Values: integer greater than or equal to 2

`MultiplierSharingMinimumBitwidth`

Minimum bit width for shared multipliers, specified as a positive integer.

If `ShareMultipliers` is `true` and `ResourceSharing` is greater than 1, share multipliers only if multiplier bit width is greater than or equal to `MultiplierSharingMinimumBitwidth`.

Values: integer greater than or equal to 2

`InstantiateFunctions`

Generate instantiable HDL code modules from functions.

Note If you enable `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

Values: `false` (default) | `true`

`PreserveDesignDelays`

Prevent distributed pipelining from moving design delays or allow distributed pipelining to move design delays, specified as a `logical`.

Persistent variables and `dsp.Delay` System objects are design delays.

Values: `false` (default) | `true`

`ShareAdders`

Share adders, specified as a `logical`.

If `true`, share adders when `ResourceSharing` is greater than 1 and adder bit width is greater than or equal to `AdderSharingMinimumBitwidth`.

Values: `false` (default) | `true`

`ShareMultipliers`

Share multipliers, specified as a `logical`.

If `true`, share multipliers when `ResourceSharing` is greater than 1, and multiplier bit width is greater than or equal to `MultiplierSharingMinimumBitwidth`.

Values: `true` (default) | `false`

`SimulateGeneratedCode`

Simulate generated code, specified as a `logical`.

Values: `false` (default) | `true`

`SimulationIterationLimit`

Maximum number of simulation iterations during test bench generation, specified as an integer. This property affects only test bench generation, not simulation during fixed-point conversion.

Values: unlimited (default) | positive integer

`SimulationTool`

Simulation tool name.

Values: `'ModelSim'` (default) | `'ISIM'`

`SynthesisTool`

Synthesis tool name.

Values: `'Xilinx ISE'` (default) | `'Altera Quartus II'` | `'Xilinx Vivado'`

`SynthesisToolChipFamily`

Synthesis target chip family name, specified as a character vector.

Values: `'Virtex4'` (default) | `'Family name'`

`SynthesisToolDeviceName`

Synthesis target device name, specified as a character vector.

Values: `'xc4vsx35'` (default) | `'Device name'`

`SynthesisToolPackageName`

Synthesis target package name, specified as a character vector.

Values: `'ff668'` (default) | `'Package name'`

`SynthesisToolSpeedValue`

Synthesis target speed, specified as a character vector.

Values: `'-10'` (default) | `'Speed value'`

`SynthesizeGeneratedCode`

Synthesize generated code or not, specified as a `logical`.

Values: `false` (default) | `true`

`TargetLanguage`

Target language of the generated code.

Values: `'VHDL'` (default) | `'Verilog'`

`TestBenchName`

Test bench function name, specified as a character vector. You must specify a test bench.

Values: `''` (default) | `'Testbench name'`

`TimingControllerArch`

Timing controller architecture.

TimingControllerArch Value	Description
`default` (default)	Do not generate a reset for the timing controller.
`resettable`	Generate a reset for the timing controller.

`TimingControllerPostfix`

Postfix to append to design name to form name of timing controller, specified as a character vector.

Values: `'_tc'` (default) | `'Postfix'`

`UseFileIOInTestBench`

Create and use data files for reading and writing test bench input and output data.

Values: `'on'` (default) | `'off'`

`UseMatrixTypesInHDL`

Generate 2-D matrix types in HDL code for MATLAB matrices, specified as a `logical`.

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 `InstantiateFunctions`, `UseMatrixTypesInHDL` has no effect.

`VHDLLibraryName`

Target library name for generated VHDL code, specified as a character vector.

Values: `'work'` (default) | `'Library name'`

Cosimulation

`GenerateCosimTestBench`

Generate a cosimulation test bench or not, specified as a `logical`.

Values: `false` (default) | `true`

SimulateCosimTestBench

Simulate generated cosimulation test bench, specified as a `logical`. This option is ignored if `GenerateCosimTestBench` is `false`.

Values: `false` (default) | `true`

CosimClockEnableDelay

Time (in clock cycles) between deassertion of reset and assertion of clock enable.

Values: `0` (default)

CosimClockHighTime

The number of nanoseconds the clock is high.

Values: `5` (default)

CosimClockLowTime

The number of nanoseconds the clock is low.

Values: `5` (default)

CosimHoldTime

The hold time for input signals and forced reset signals, specified in nanoseconds.

Values: `2` (default)

CosimLogOutputs

Log and plot outputs of the reference design function and HDL simulator.

Values: `false` (default) | `true`

CosimResetLength

Specify time (in clock cycles) between assertion and deassertion of reset.

Values: `2` (default)

`CosimRunMode`

HDL simulator run mode during simulation. When in Batch mode, you do not see the HDL simulator GUI, and the HDL simulator automatically shuts down after simulation.

Values: `Batch` (default) | `GUI`

`CosimTool`

HDL simulator for the generated cosim test bench.

Values: `ModelSim` (default) | `Incisive`

FPGA-in-the-loop

`GenerateFILTestBench`

Generate a FIL test bench or not, specified as a `logical`.

Values: `false` (default) | `true`

`SimulateFILTestBench`

Simulate generated cosimulation test bench, specified as a `logical`. This option is ignored if `GenerateCosimTestBench` is `false`.

Values: `false` (default) | `true`

`FILBoardName`

FPGA board name, specified as a character vector. You must override the default value and specify a valid board name.

Values: `'Choose a board'` (default) | `'A board name'`

`FILBoardIPAddress`

IP address of the FPGA board, specified as a character vector. You must enter a valid IP address.

Values: `192.168.0.2` (default)

`FILBoardMACAddress`

MAC address of the FPGA board, specified as a character vector. You must enter a valid MAC address.

Values: `00-0A-35-02-21-8A` (default)

`FILAdditionalFiles`

List of additional source files to include, specified as a character vector. Separate file names with a semi-colon (";").

Values: `''` (default) | `'Additional source files'`

`FILLogOutputs`

Log and plot outputs of the reference design function and FPGA.

Values: `false` (default) | `true`

Examples

Generate Verilog Code from MATLAB Code

Create a `coder.HdlConfig` object, `hdlcfg`.

```
hdlcfg = coder.config('hdl'); % Create a default 'hdl' config
```

Set the test bench name. In this example, the test bench function name is `mlhdlc_dti_tb`.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Set the target language to Verilog.

```
hdlcfg.TargetLanguage = 'Verilog';
```

Generate HDL code from your MATLAB design. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -config hdlcfg mlhdlc_dti
```

Generate Cosim and FIL Test Benches

Create a `coder.FixptConfig` object with default settings and provide test bench name.

```
fixptcfg = coder.config('fixpt');
fixptcfg.TestBenchName = 'mlhdlc_sfir_tb';
```

Create a `coder.HdlConfig` object with default settings and set enable rate.

```
hdlcfg = coder.config('hdl'); % Create a default 'hdl' config
hdlcfg.EnableRate = 'DUTBaseRate';
```

Instruct MATLAB to generate a cosim test bench and a FIL test bench. Specify FPGA board name.

```
hdlcfg.GenerateCosimTestBench = true;
hdlcfg.FILBoardName = 'Xilinx Virtex-5 XUPV5-LX110T development board';
hdlcfg.GenerateFILTestBench = true;
```

Perform code generation, Cosim test bench generation, and FIL test bench generation.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_sfir
```

- “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

Topics

“Generate HDL Code from MATLAB Code Using the Command Line Interface”

8

Shared Class and Function Reference for HDL Code Generation from MATLAB and Simulink

hdlcoder.CodingStandard

Create HDL coding standard customization object

Syntax

```
cso = hdlcoder.CodingStandard(standardName)
```

Description

`cso = hdlcoder.CodingStandard(standardName)` creates an HDL coding standard customization object that you can use to customize the rules and the appearance of the coding standard report.

If you do not want to customize the rules or appearance of the coding standard report, you do not need to create an HDL coding standard customization object.

Examples

Customize coding standard rules for MATLAB to HDL workflow

Create an HDL coding standard customization object, *cso*.

```
cso = hdlcoder.CodingStandard('Industry');
```

Customize the coding standard options as follows:

- Do not show passing rules in the coding standard report.
- Set the maximum if-else nesting depth to 2.
- Disable the check for line length.

```
cso.ShowPassingRules.enable = false;
cso.IfElseNesting.depth = 2;
cso.LineLength.enable = false;
```

Create an HDL codegen configuration object.

```
hdlcfg = coder.config('hdl');
```

Specify the coding standard and coding standard customization object.

```
hdlcfg.HDLCodingStandard = 'Industry';
hdlcfg.HDLCodingStandardCustomizations = cso;
```

Specify your test bench function name. In this example, the test bench function is *mlhdlc_dti_tb*.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Generate HDL code for the design and check the code according to the customized HDL coding standard rules. In this example, the design function is *mlhdlc_dti*.

```
codegen -config hdlcfg mlhdlc_dti
```

Customize coding standard rules for Simulink to HDL workflow

Create an HDL coding standard customization object

- Load the sfir_fixed model
- Create a coding standard customization object *cso*

```
load_system('sfir_fixed')
cso = hdlcoder.CodingStandard('Industry');
```

Customize the coding standard options

- Do not show passing rules in the report.
- Set maximum line length to 80 characters.
- Check that module, instance, and entity names are between 5 and 50 characters long.

```
cso.ShowPassingRules.enable = false;
cso.LineLength.length = 80;
cso.ModuleInstanceEntityNameLength.length = [5 50];
```

Generate HDL code for your design

Generate HDL code and check it according to the customized HDL coding standard rules. The DUT subsystem is symmetric_fir.

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry',...
        'HDLCodingStandardCustomizations',cso, 'TargetDirectory', 'C:/coding_standard/h

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\coding_standard\hdlsrc\sfir_fixed\symmetr
### Industry Compliance report with 4 errors, 77 warnings, 6 messages.
### Generating Industry Compliance Report <a href="matlab:web('C:\coding_standard\hdlsr
### Creating HDL Code Generation Check Report file://C:\coding_standard\hdlsrc\sfir_fix
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

- “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'`

at

Output Arguments

`cso` — HDL coding standard customizations

HDL coding standard customization object

HDL coding standard customizations, returned as an HDL coding standard customization object.

See Also

Properties

HDL Coding Standard Customization | HDLCodingStandardCustomizations

Topics

"Generate an HDL Coding Standard Report from Simulink"
"Generate an HDL Coding Standard Report from MATLAB"
"HDL Coding Standard Report"
"Basic Coding Practices"
"RTL Description Techniques"
"RTL Design Methodology Guidelines"

Introduced in R2014b

HDL Coding Standard Customization Properties

Customize HDL coding standard

Description

HDL coding standard customization properties control how HDL Coder generates and checks code according to a specified coding standard. By changing property values, you can customize the rules and the appearance of the coding standard report.

Use dot notation to refer to a particular object and property:

```
cso = hdlcoder.CodingStandard('Industry');
len = cso.SignalPortParamNameLength.length;
cso.ShowPassingRules.enable = false;
```

The generated code follows the customized coding standard rules as much as possible. However, if following a coding standard rule could cause the HDL code to be uncompilable or unsynthesizable, the coder does not follow the rule.

Properties

Coding Standard Report

`ShowPassingRules` — Show passing rules in coding standard report

structure

Show or do not show passing rules in coding standard report, specified as a structure with the following field.

Field	Description
enable	Set to `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`.

`ModuleInstanceEntityNameLength` — Check module, instance, and entity name length
structure

Check for module, instance, and entity name lengths (rule CGSL-1.A.B.1), specified as a structure with the following fields.

Field	Description
enable	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].

`SignalPortParamNameLength` — Check signal, port, and parameter name length
structure

Check for signal, port, and parameter name lengths (rule CGSL-1.A.C.3), specified as a structure with the following fields.

Field	Description
enable	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].

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`.

`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 `false`.

`ConditionalRegionCheck` — Check for length of conditional statements in a process or always block

structure

Check for length of conditional statements (if-else, case, and loops) which are described separately in a process block or an always block (rule CGSL-2.F.B.1), specified as a structure with the following fields.

Field	Description
enable	Set to `true` to check length of conditional statements. Set to `false` if you do not want to check the length of conditional statements. The default is `true`.
length	Number of conditional statements which are described separately within a process block (`VHDL`) or an always block (`Verilog`). The default is 1.

`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. 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`.
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. The default is `true`.

See Also

`hdlcoder.CodingStandard`

Topics

“Generate an HDL Coding Standard Report from MATLAB”
“Generate an HDL Coding Standard Report from Simulink”
“HDL Coding Standard Report”

"Basic Coding Practices"
"RTL Description Techniques"
"RTL Design Methodology Guidelines"

hdl.BlackBox System object

Package: hdl

Black box for including custom HDL code

Description

`hdl.BlackBox` provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`B = hdl.BlackBox` creates a black box System object for HDL code generation.

Properties

`AddClockEnablePort` — Add clock enable port

`'on'` (default) | `'off'`

If `'on'`, add a clock enable input port to the interface generated for the black box System object. The name of the port is specified by `ClockEnableInputPort`.

`AddClockPort` — Add clock port
`'on'` (default) | `'off'`

If `'on'`, add a clock input port to the interface generated for the black box System object. The name of the port is specified by `ClockInputPort`.

`AddResetPort` — Add reset port
`'on'` (default) | `'off'`

If `'on'`, add a reset input port to the interface generated for the black box System object. The name of the port is specified by `ResetInputPort`.

`AllowDistributedPipelining` — Register placement for distributed pipelining
`'off'` (default) | `'on'`

If `'on'`, allow HDL Coder to move registers across the black box System object, from input to output or output to input.

`ClockEnableInputPort` — Clock enable input port name
`'clk_enable'` (default) | character vector

HDL name for clock enable input port, specified as a character vector.

`ClockInputPort` — Clock input port name
`'clk'` (default) | character vector

HDL name for clock input port, specified as a character vector.

`EntityName` — Module or entity name
System object instance name (default) | character vector

VHDL entity or Verilog module name generated for the black box System object, specified as a character vector.

Example: `'myBlackBoxName'`

`ImplementationLatency` — Latency in clock cycles
-1 (default) | integer

Latency of black box System object in clock cycles, specified as an integer.

If 0 or greater, this value is used for delay balancing.

If -1, latency is unknown. This disables delay balancing.

`InlineConfigurations` — Generate VHDL configuration
`InlineConfigurations` global property value (default) | `'on'` | `'off'`

When `'on'`, generate a VHDL configuration.

When `'off'`, do not generate a VHDL configuration and require a user-supplied external configuration. Set to `'off'` if you are creating your own VHDL configuration.

`InputPipeline` — Input pipeline stages
0 (default) | positive integer

Number of input pipeline stages, or pipeline depth, to insert in the generated code.

`OutputPipeline` — Output pipeline stages
0 (default) | positive integer

Number of output pipeline stages, or output pipeline depth, to insert in the generated code.

`ResetInputPort` — Reset port name
`'reset'` (default) | character vector

HDL name for reset input port, specified as a character vector.

`VHDLArchitectureName` — VHDL architecture name
`'rtl'` (default) | character vector

VHDL architecture name, specified as a character vector. The coder generates the architecture name only if `InlineConfigurations` is `'on'`.

`VHDLComponentLibrary` — VHDL component library name
`'work'` (default) | character vector

Library from which to load the VHDL component, specified as a character vector.

`NumInputs` — Number of custom input ports
1 (default) | positive integer

Number of additional input ports in the custom HDL code, specified as a positive integer.

`NumOutputs` — Number of custom output ports

1 (default) | positive integer

Number of additional output ports in the custom HDL code, specified as a positive integer.

See Also

`coder.HdlConfig`

Topics

"Integrate Custom HDL Code Into MATLAB Design"
"Generate a Board-Independent IP Core from MATLAB"
"Generate Black Box Interface for Subsystem"

Introduced in R2015a

hdl.RAM System object

Package: hdl

Single, simple dual, or dual-port RAM for memory read/write access

Description

`hdl.RAM` reads from and writes to memory locations for a single, simple dual, or dual-port RAM. The output data is delayed one step. If your input data is scalar, the address and write enable inputs must be scalar, andHDL Coder infers a single RAM block. If your data is a vector, HDL Coder infers an array of parallel RAM banks. With vector data input, the address and write enable inputs can be both scalars or vectors. When you specify scalar inputs for the write enable and address ports, the system object applies the same operation to each RAM bank.

To read from or write to memory locations in the RAM:

1. Create the hdl.RAM object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
ram = hdl.RAM
ram = hdl.RAM(Name,Value)
```

Description

`ram = hdl.RAM` returns a single port RAM System object that you can write to or read from a memory location.

`ram = hdl.RAM(Name,Value)` returns a single, simple dual, or dual port RAM System object with properties set using one or more name-value pairs. Enclose each property name in single quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`RAMType` — Type of RAM

`'Single port'` (default) | `'Simple dual port'` | `'Dual port'`

Type of RAM, specified as either:

- `'Single port'` — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- `'Simple dual port'` — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- `'Dual port'` — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

`WriteOutputValue` — Behavior for Write output

`'New data'` (default) | `'Old data'`

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `Old data'` — Send out old data at the address to the output.

Dependencies

Specify this property when you set **RamType** to `'Single port'` or `'Dual port'`. This property does not apply for `Simple Dual Port RAM` object.

`RAMInitialValue` — Initial output of RAM

`'0.0'` (default) | `Scalar` | `Vector`

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

Usage

Syntax

```
dataOut = ram(wrData,rwAddress,wrEn)

rdDataOut = ram(wrData,wrAddress,wrEn,rdAddress)

[wrDataOut,rdDataOut] = ram(wrData,wrAddress,wrEn,rdAddress)
```

Description

`dataOut = ram(wrData,rwAddress,wrEn)` reads the value in memory location `rwAddress` when `wrEn` is false. When `wrEn` is true, you write the value `wrData` into the memory location `rwAddress`. `dataOut` is the new or old data at `rwAddress`. Use this syntax when you create a single port RAM System object.

`rdDataOut = ram(wrData,wrAddress,wrEn,rdAddress)` writes the value `wrData` into memory location `wrAddress` when `wrEn` is true. `rdDataOut` is the old data at the address location `rdAddress`. Use this syntax when you create a simple dual port RAM System object.

`[wrDataOut,rdDataOut] = ram(wrData,wrAddress,wrEn,rdAddress)` writes the value `wrData` into the memory location `wrAddress` when `wrEn` is true. `wrDataOut` is

the new or old data at memory location `wrAddress`. `rdDataOut` is the old data at the address location `rdAddress`. Use this syntax when you create a dual port RAM System object.

Input Arguments

`wrData` — Write data

`Scalar` (default) | `Vector`

Data that you write into the RAM memory location when `wrEn` is true. This value can be `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fi`

`rwAddress` — Write or Read address

`Scalar` (default) | `Vector`

Address that you write the `wrData` into when `wrEn` is true. The System object reads the value in memory location `rwAddress` when `wrEn` is false. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned. Specify this address when you create a single port RAM object.

Data Types: `uint8` | `uint16` | `fi`

`wrEn` — Write enable

`Scalar` (default) | `Vector`

When `wrEn` is true, you write the `wrData` into the RAM memory location. If you create a single port RAM, the System object reads the value in the memory location when `wrEn` is false. This value must be logical.

Data Types: `logical`

`rdAddress` — Read address

`Scalar` (default) | `Vector`

Address that you read the data from when you create a simple dual port RAM or dual port RAM System object. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Data Types: `uint8` | `uint16` | `fi`

`wrAddress` — Write address

`Scalar` (default) | `Vector`

Address that you write the data into when you create a simple dual port RAM or dual port RAM System object. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Data Types: `uint8` | `uint16` | `fi`

Output Arguments

`dataOut` — Output data

`Scalar` (default) | `Vector`

Output data that the System object reads from the memory location `rwAddress` a single port RAM object when `wrEn` is false.

`rdDataOut` — Data from Read address

`Scalar` (default) | `Vector`

Old output data that the System object reads from the memory location `rdAddress` of a simple dual port RAM or dual port RAM System object.

`wrDataOut` — Data from Write address

`Scalar` (default) | `Vector`

New or old output data that the System object reads from the memory location `wrAddress` of a simple dual port RAM or dual port RAM System object.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step Run System object algorithm

clone	Create duplicate System object
getNumInputs	Number of inputs required to call the System object
getNumOutputs	Number of outputs from calling the System object
isLocked	Determine if System object is locked
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Observe Previous Data at Write Time

Construct System object to read from or write to a memory location in RAM. Set `WriteOutputValue` to `Old data` to return the previous value stored at the write address.

The output data port corresponds to the read/write address passed in. During a write operation, the old data at the write address is sent out as the output.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_1p = hdl.RAM('RAMType','Single port',...
                           'WriteOutputValue','Old data')

dataLength    = 10;
dataIn = 1:10;
dataOut = zeros(1,dataLength);
```

```
ram_1p =

  hdl.RAM with properties:

            RAMType: 'Single port'
   WriteOutputValue: 'Old data'
     RAMInitialValue: 0
```

Write a count pattern to the memory. Previous values on the first writes are all zero.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
```

```
dataOut =

     0     0     0     0     0     0     0     0     0     0
```

Read the data back.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = false;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
```

```
dataOut =

     0     1     2     3     4     5     6     7     8     9
```

Now, write the count in reverse order. The previous values are the original count.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(dataLength-ii+1),addressIn,writeEnable);
end
dataOut
```

```
dataOut =

    10     1     2     3     4     5     6     7     8     9
```

Read/Write Single-Port RAM

Create System object that writes to a single port RAM and reads the newly written value.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Construct single-port RAM System object. When you write a location, the object returns the new value. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

```
ram_1p = hdl.RAM('RAMType','Single port','WriteOutputValue','New data');
dataLength       = 16;
[dataIn,dataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and then read data back out again.

```
for ii = 1:dataLength
  dataIn(ii)  = randi([0 63],1,1,'uint8');
  addressIn   = fi((ii-1),0,4,0);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
for ii = 1:dataLength
  addressIn   = fi((ii-1),0,4,0);
  writeEnable = false;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
```

```
dataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    0   52   57    8   58   40    6   17   35   61   61   10   62   61   31

  Column 16
```

```
   51

dataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    9   52   57    8   58   40    6   17   35   61   61   10   62   61   31

  Column 16

   51
```

Create Simple Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

The output data port corresponds to the read address. If a read operation is performed at the same address as the write operation, old data at that address is read out as the output. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_2p = hdl.RAM('RAMType','Simple dual port');
dataLength       = 16;
[dataIn,dataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and read the old data from the same address.

```
for ii = 1:dataLength
  dataIn(ii)  = randi([0 63],1,1,'uint8');
  wrAddr  = fi((ii-1),0,4,0);
  writeEnable = true;
  ataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);
end
dataOut
```

```
dataOut =

  1x16 uint8 row vector

   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Write and read from different addresses. The object returns the read result after one cycle delay.

```
for ii = 1:dataLength
  wrAddr   = fi((ii-1),0,4,0);
  rdAddr   = fi(dataLength-ii+1,0,4,0);
  writeEnable = true;
  dataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
dataOut
```

```
dataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    0    9    9   51   31   61   62   10   61   61   35   17    6   40   58

  Column 16

    8
```

Create Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

There are two output ports: a write output data port and a read output data port. The write output data port sends out the new data at the write address. The read output data port sends out the old data at the read address. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
dataLength       = 16;
[dataIn,wrDataOut,rdDataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and read the old data from the same address.

```
for ii = 1:dataLength
  dataIn(ii)  = randi([0 63],1,1,'uint8');
  wrAddr  = fi((ii-1),0,4,0);
  writeEnable = true;
  [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);
end
wrDataOut
rdDataOut
```

```
wrDataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    0   52   57    8   58   40    6   17   35   61   61   10   62   61   31

  Column 16

   51


rdDataOut =

  1x16 uint8 row vector

   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Write and read from different addresses. The object returns the read result after one cycle delay.

```
for ii = 1:dataLength
  wrAddr   = fi((ii-1),0,4,0);
  rdAddr   = fi(dataLength-ii+1,0,4,0);
  writeEnable = true;
  [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
wrDataOut
rdDataOut


wrDataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    9   52   57    8   58   40    6   17   35   61   61   10   62   61   31

  Column 16

   51


rdDataOut =

  1x16 uint8 row vector

  Columns 1 through 15

    0    9    9   51   31   61   62   10   61   61   35   17    6   40   58

  Column 16

    8
```

Create Dual-Port RAM with Multiple Banks

Create a System object that can write vector data to a dual-port RAM and read vector data out. Each element of the vector corresponds to a separate bank of RAM. This example creates 4 16-bit banks. Each bank has eight entries.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

Construct dual-port RAM System object.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
```

Create vector write data and addresses. Use a 3-bit address (for 8 locations), and write 16-bit data. Read and write addresses are independent. Allocate memory for the output data.

```
ramDataIn = fi(randi((2^16)-1,1,4),0,16,0);
ramReadAddr = fi([1,1,1,1],0,3,0);
ramWriteAddr = fi([1,1,1,1],0,3,0);
[wrOut,rdOut] = deal(fi(zeros(1,4),0,16,0));
```

First, write locations in bank 1 and 4, then read all banks. The write data is echoed in the `wrOut` output argument. The object returns read results after one cycle delay.

```
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[true,false,false,true],ramReadAddr);
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr);
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr)
```

```
wrOut =

       53393           0           0       59859

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 0

rdOut =

       53393           0           0       59859

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 0
```

- “HDL Code Generation from hdl.RAM System Object”

- “Getting Started with RAM and ROM in Simulink®”

Algorithms

In your Simulink model, you can use the `hdl.RAM` inside a MATLAB System or a MATLAB Function block. If you log the output of a MATLAB System block, the output data has at least three dimensions because the MATLAB System block has at least two dimensions, and the time data adds a third dimension. For example, if you input scalar data to the block, the logged output data has the dimension `1x1xN`, where `N` is the number of time steps. To obtain an output dimension that is same as the input dimension, add a Reshape block at the output with **Output dimensionality** set to `Derive from reference input port`.

RAM Inference with Scalar Data

If your data is scalar, the RAM size, or number of locations, is inferred from the data type of the address variable.

Data type of address variable	RAM address size (bits)
`single` or `double`	16
`uint`*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.

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.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

See Also

Blocks

Dual Port RAM | Dual Rate Dual Port RAM | Simple Dual Port RAM | Single Port RAM

Topics

“HDL Code Generation from hdl.RAM System Object”
“Getting Started with RAM and ROM in Simulink®”
“Implement RAM Using MATLAB Code”
“HDL Code Generation for System Objects”

Introduced in R2015a

coder.hdl.loopspec

Unroll or stream loops in generated HDL code

Syntax

```
coder.hdl.loopspec('unroll')
coder.hdl.loopspec('unroll',unroll_factor)
coder.hdl.loopspec('stream')
coder.hdl.loopspec('stream',stream_factor)
```

Description

`coder.hdl.loopspec('unroll')` fully unrolls a loop in the generated HDL code. Instead of a loop statement, the generated code contains multiple instances of the loop body, with one loop body instance per loop iteration.

The `coder.hdl.loopspec` pragma does not affect MATLAB simulation behavior.

Note If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

`coder.hdl.loopspec('unroll',unroll_factor)` unrolls a loop by the specified unrolling factor, `unroll_factor`, in the generated HDL code.

The generated HDL code is a loop statement that contains `unroll_factor` instances of the original loop body. The number of loop iterations in the generated code is (*`original_loop_iterations`* / `unroll_factor`). If (*`original_loop_iterations`* / `unroll_factor`) has a remainder, the remaining iterations are fully unrolled as loop body instances outside the loop.

This pragma does not affect MATLAB simulation behavior.

Note If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

`coder.hdl.loopspec('stream')` generates a single instance of the loop body in the HDL code. Instead of using a loop statement, the generated code implements local oversampling and added logic to match the functionality of the original loop.

You can specify this pragma for loops at the top level of your MATLAB design.

This pragma does not affect MATLAB simulation behavior.

Note If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

`coder.hdl.loopspec('stream',stream_factor)` unrolls the loop with `unroll_factor` set to *`original_loop_iterations`* / `stream_factor` rounded down to the nearest integer, and also oversamples the loop. If (*`original_loop_iterations`* / `stream_factor`) has a remainder, the remainder loop body instances outside the loop are not oversampled, and run at the original rate.

You can specify this pragma for loops at the top level of your MATLAB design.

This pragma does not affect MATLAB simulation behavior.

Note If you specify the `coder.unroll` pragma, this pragma takes precedence over `coder.hdl.loopspec`. `coder.hdl.loopspec` has no effect.

Examples

Completely unroll MATLAB loop in generated HDL code

Unroll loop in generated code.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));
```

```
    coder.hdl.loopspec('unroll');
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end
```

Partially unroll MATLAB loop in generated HDL code

Generate a loop statement in the HDL code that has two iterations and contains five instances of the original loop body.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('unroll', 5);
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end
```

Completely stream MATLAB loop in generated HDL code

In the generated code, implement the 10-iteration MATLAB loop as a single instance of the original loop body that is oversampled by a factor of 10.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('stream');
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end
```

Partially stream MATLAB loop in generated HDL code

In the generated code, implement the 10-iteration MATLAB loop as five instances of the original loop body that are oversampled by a factor of 2.

```
function y = hdltest
    pv = uint8(1);
    y = uint8(zeros(1,10));

    coder.hdl.loopspec('stream', 2);
    % Optional comment between pragma and loop statement
    for i = 1:10
        y(i) = pv + i;
    end
end
```

Input Arguments

`stream_factor` — Loop streaming factor

positive integer

Loop streaming factor, specified as a positive integer.

Setting `stream_factor` to the number of original loop iterations is equivalent to fully streaming the loop, or using `coder.hdl.loopspec('stream')`.

Example: 4

`unroll_factor` — Loop unrolling factor

positive integer

Number of loop body instances, specified as a positive integer.

Setting `unroll_factor` to the number of original loop iterations is equivalent to fully unrolling the loop, or using `coder.hdl.loopspec('unroll')`.

Example: 10

See Also

Topics

"Optimize MATLAB Loops"

Introduced in R2015a

coder.hdl.pipeline

Insert pipeline registers at output of MATLAB expression

Syntax

```
out = coder.hdl.pipeline(expr)
out = coder.hdl.pipeline(expr,num)
```

Description

`out = coder.hdl.pipeline(expr)` inserts one pipeline register at the output of `expr` in the generated HDL code. This pragma does not affect MATLAB simulation behavior.

Use this pragma to specify exactly where to insert pipeline registers. For example, in a MATLAB assignment statement, you can specify the coder.hdl.pipeline pragma:

- On the entire right side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see "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 side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see "Register Inputs and Outputs".

Examples

Insert one pipeline register at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert a single pipeline register.

```
y = coder.hdl.pipeline(a + b * c);
```

Insert multiple pipeline registers at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert three pipeline registers.

```
y = coder.hdl.pipeline(a + b * c, 3);
```

Insert pipeline registers at intermediate stage of MATLAB expression

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers.

```
y = a + coder.hdl.pipeline(b * c, 5);
```

Insert pipeline registers at intermediate stage and at output of MATLAB expression

At an intermediate stage and at the output of a MATLAB expression, use nested `coder.hdl.pipeline` pragmas to insert pipeline registers.

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers, and insert two pipeline registers at the output of the whole expression.

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
```

- "Pipeline MATLAB Expressions"

Input Arguments

`expr` — MATLAB expression to pipeline

MATLAB expression

MATLAB expression to pipeline. At the output of this expression in the generated HDL code, insert pipeline registers.

Example: `a + b`

`num` — Number of registers

MATLAB expression

Number of pipeline registers to insert at the output of `expr` in the generated HDL code, specified as a positive integer.

Example: 3

See Also

Topics

"Pipeline MATLAB Expressions"
"Pipelining MATLAB Code"

Introduced in R2015a

hdlcoder.Board class

Package: hdlcoder

Board registration object that describes SoC custom board

Description

`board = hdlcoder.Board` creates a board object that you use to register a custom board for an SoC platform.

To specify the characteristics of your board, set the properties of the board object.

Construction

`board` `= hdlcoder.Board` creates an `hdlcoder.Board` object that you can use to register a custom board for an SoC platform.

Properties

`BoardName` — Board name

`''` (default) | character vector

Board name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Target platform** dropdown list.

Example: `'Enclustra Mars ZX3 with PM3 base board'`

`FPGAVendor` — Vendor name

`''` (default) | `'Altera'` | `'Xilinx'`

FPGA vendor name, specified as a character vector.

Example: `'Xilinx'`

`FPGAFamily` — FPGA family name

`''` (default) | character vector

FPGA family name, specified as a character vector.

Example: `'Zynq'`

`FPGADevice` — FPGA device identifier

`''` (default) | character vector

FPGA device identifier, specified as a character vector.

Example: `'xc7z020'`

`FPGAPackage` — FPGA package identifier for Xilinx devices

`''` (default) | character vector

FPGA package identifier for Xilinx devices, specified as a character vector.

For Altera devices, this property is ignored.

Example: `'clg484'`

`FPGASpeed` — FPGA speed for Xilinx devices

`''` (default) | character vector

FPGA speed for Xilinx devices, specified as a character vector.

For Altera devices, this property is ignored.

Example: `'-1'`

`SupportedTool` — Supported synthesis tool

`''` (default) | cell array of character vectors

Synthesis tool or tools that support this board, specified as a cell array of character vectors. In the HDL Workflow Advisor, the **Synthesis tool** dropdown list shows the values in this cell array.

Example: `{'Altera Quartus II'}`

Example: `{'Xilinx Vivado'}`

Example: `{'Xilinx Vivado','Xilinx ISE'}`

`JTAGChainPosition` — Optional JTAG chain position number

2 (default) | positive integer

JTAG chain position number, specified as a positive integer. The JTAG chain position number is used when programming the FPGA via JTAG.

This property is optional.

Example: 3

Methods

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`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a

addExternalIOInterface

Class: hdlcoder.Board
Package: hdlcoder

Define external IO interface for board object

Syntax

```
addExternalIOInterface('InterfaceID',interfacename,'InterfaceType',
interfacetype,'PortName',portname,'PortWidth',portwidth,'FPGAPin',
pins,'IOPadConstraint',constraints)
```

Description

`addExternalIOInterface('InterfaceID',interfacename,'InterfaceType', interfacetype,'PortName',portname,'PortWidth',portwidth,'FPGAPin', pins,'IOPadConstraint',constraints)` adds an external IO interface to an `hdlcoder.Board` object. You can add multiple external IO interfaces to your board object.

Use this method if your board has more than one external interface, or if you want to be able to predefine FPGA pin names for mapping from the HDL Workflow Advisor.

Input Arguments

`interfacename` — Interface name

character vector

Interface name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Target Platform Interfaces** dropdown list.

Example: `'LEDs General Purpose'`

`interfacetype` — Interface direction

`'IN'` | `'OUT'`

Interface direction, specified as a character vector. In the HDL Workflow Advisor, when you specify a target interface for each of your DUT ports, this external IO interface is available only for ports with a matching direction.

For example, if you set `interfacetype` to `'OUT'`, this external IO interface is available only for `Outport` DUT ports.

Example: `'OUT'`

`portname` — Port name

character vector

Board top-level port name, specified as a character vector.

Example: `'GPLEDs'`

`portwidth` — Port bit width

positive integer

Port bit width, specified as a positive integer.

Example: 4

`pins` — Pin names

cell array of character vectors

FPGA pin names, specified as a cell array of character vectors.

Example: `{'H18','AA14','AA13','AB15'}`

`constraints` — IO pad constraints

`{}` (default) | cell array of character vectors

IO pad constraints, specified as a cell array of character vectors.

Example: `{'IOSTANDARD = LVCMOS25'}`

Example: `{'IOSTANDARD = LVCMOS25','SLEW = SLOW'}`

Tips

- For details about the external IO interface ports, pins, and constraints for your board, view the board documentation.

See Also

`hdlcoder.Board` | `hdlcoder.Board.addExternalPortInterface`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Board and Reference Design Registration System"

Introduced in R2015a

addExternalPortInterface

Class: hdlcoder.Board
Package: hdlcoder

Define external port interface for board object

Syntax

```
addExternalPortInterface('IOPadConstraint',constraints)
```

Description

`addExternalPortInterface('IOPadConstraint',constraints)` adds a generic external port interface to an `hdlcoder.Board` object. You can add at most one external port interface to your board object.

Use this method if you want the `External Port` option to be available in the HDL Workflow Advisor **Target Platform Interface** dropdown list. If you use this method to add an external port, in the HDL Workflow Advisor, you must manually specify pin names in the **Bit Range / Address / FPGA Pin** field.

Input Arguments

`constraints` — IO pad constraints

`{}` (default) | cell array of character vectors

IO pad constraints, specified as a cell array of character vectors.

Example: `{'IOSTANDARD = LVCMOS25'}`

Example: `{'IOSTANDARD = LVCMOS25','SLEW = SLOW'}`

Tips

- To get IO constraint names for your board, view the board documentation.

Alternatives

If you know the details of the external interface, and want to make them available as UI dropdown list options in the HDL Workflow advisor, use the `hdlcoder.Board.addExternalIOInterface` method instead. For example, using `hdlcoder.Board.addExternalIOInterface`, you can predefine characteristics of the interface such as the name, port bit width, signal direction, and valid pin names.

See Also

`hdlcoder.Board` | `hdlcoder.Board.addExternalIOInterface`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Board and Reference Design Registration System"

Introduced in R2015a

validateBoard

Class: hdlcoder.Board
Package: hdlcoder

Check property values in board object

Syntax

```
validateBoard
```

Description

`validateBoard` checks that the `hdlcoder.Board` object has nondefault values for all required properties, and that property values have valid data types. This method does not check the correctness of property values for the target board. If validation fails, the software displays an error message.

See Also

`hdlcoder.Board`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Board and Reference Design Registration System"

Introduced in R2015a

hdlcoder.ReferenceDesign class

Package: hdlcoder

Reference design registration object that describes SoC reference design

Description

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool', toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

To specify the characteristics of your reference design, set the properties of the reference design object.

Use a reference design tool version that is compatible with the supported tool version. If you choose a different tool version, it is possible that HDL Coder is unable to create the reference design project for IP core integration.

Construction

`refdesign = hdlcoder.ReferenceDesign('SynthesisTool',toolname)` creates a reference design object that you use to register a custom reference design for an SoC platform.

Input Arguments

`toolname` — Synthesis tool name

`Xilinx Vivado` (default) | `Altera Quartus II` | `Xilinx ISE` | `Xilinx Vivado`

Synthesis tool name, specified as a character vector.

Example: `'Altera Quartus II'`

Properties

`ReferenceDesignName` — Reference design name

`''` (default) | character vector

Reference design name, specified as a character vector. In the HDL Workflow Advisor, this name appears in the **Reference design** dropdown list.

Example: `'Default system (Vivado 2015.4)'`

`BoardName` — Board name

`''` (default) | character vector

Board associated with this reference design, specified as a character vector.

Example: `'Enclustra Mars ZX3 with PM3 base board'`

`SupportedToolVersion` — Supported tool version

`{}` (default) | cell array of character vectors

One or more tool versions that work with this reference design, specified as a cell array of character vectors.

Example: `{'2015.4'}`

Example: `{'13.7','14.0'}`

`CustomConstraints` — Design constraint file (optional)

`{}` (default) | cell array of character vectors

One or more design constraint files, specified as a cell array of character vectors. This property is optional.

Example: `{'MarsZX3_PM3.xdc'}`

Example: `{'MyDesign.qsf'}`

`CustomFiles` — Relative path to required file or folder (optional)

`{}` (default) | cell array of character vectors

One or more relative paths to files or folders that the reference design requires, specified as a cell array of character vectors. This property is optional.

Examples of required files or folders:

- Existing IP core used in the reference design.

  For example, if the IP core, *my_ip_core*, is in the reference design folder, set `CustomFiles` to `{'my_ip_core']`
- PS7 definition XML file.

  For example, to include a PS7 definition XML file, *ps7_system_prj.xml*, in a folder, *data*, set `CustomFiles` to `{fullfile('data', 'ps7_system_prj.xml')}`
- Folder containing existing IP cores used in the reference design. HDL Coder only supports a specific IP core folder name for each synthesis tool:
  - For Altera Qsys, IP core files must be in a folder named `ip`. Set `CustomFiles` to `{'ip'}`.
  - For Xilinx Vivado, IP core files, or a zip file containing the IP core files, must be in a folder named `ipcore`. Set `CustomFiles` to `{'ipcore'}`.
  - For Xilinx EDK, IP core files must be in a folder named `pcores`. Set `CustomFiles` to `{'pcores'}`.

Note To add IP modules to the reference design, it is recommended to create an IP repository folder that contains these IP modules, and then use the `addIPRepository` on page 8-77 method.

Example: `{'my_ip_core'}`

Example: `{fullfile('data', 'ps7_system_prj.xml')}`

Example: `{'ip'}`

Example: `{'ipcore'}`

Example: `{'pcores'}`

`IPCacheZipFile` — IP cache file to include in the project

`''` (default) | `'ipcache.zip'` | character vector

Specify the IP cache zip file to include in your project. When you run the `IP Core Generation` workflow in the HDL Workflow Advisor, the code generator extracts this file in the **Create Project** task. The **Build FPGA Bitstream** task reuses the IP cache, which accelerates reference design synthesis.

This property is optional.

Example: `'ipcache.zip'`

Methods

addAXI4MasterInterface	Add and define AXI4 Master interface
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
addIPRepository	Include IP modules from your IP repository folder in your custom reference design
addParameter	Add and define custom parameters for your reference design
CallbackCustomProgrammingMethod	Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor
EmbeddedCoderSupportPackage	Specify whether to use an Embedded Coder support package
PostBuildBitstreamFcn	Function handle for callback function that gets executed after Build FPGA Bitstream task in the HDL Workflow Advisor
PostCreateProjectFcn	Function handle for callback function that gets executed after Create Project task in the HDL Workflow Advisor
PostSWInterfaceFcn	Function handle for custom callback function that gets executed after Generate Software Interface Model task in the HDL Workflow Advisor
PostTargetInterfaceFcn	Function handle for callback function that gets executed after Set Target Interface task in the HDL Workflow Advisor
PostTargetReferenceDesignFcn	Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor
validateReferenceDesign	Check property values in reference design object

See Also

`hdlcoder.Board`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Define Custom Parameters and Callback Functions for Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a

addAXI4MasterInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add and define AXI4 Master interface

Syntax

```
addAXI4MasterInterface('InterfaceID',
Interface_ID,'InterfaceConnection',Interface_Connection)
addAXI4MasterInterface('InterfaceID',
Interface_ID,'InterfaceConnection',
Interface_Connection,'TargetAddressSegments',
Target_Address_Segments)
addAXI4MasterInterface('InterfaceID',
Interface_ID,'InterfaceConnection',Interface_Connection, Name,Value)
addAXI4MasterInterface('InterfaceID',
Interface_ID,'InterfaceConnection',
Interface_Connection,'TargetAddressSegments',
Target_Address_Segments, Name,Value)
```

Description

`addAXI4MasterInterface('InterfaceID', Interface_ID,'InterfaceConnection',Interface_Connection)` adds and defines an AXI4 Master interface for an Qsys reference design.

`addAXI4MasterInterface('InterfaceID', Interface_ID,'InterfaceConnection', Interface_Connection,'TargetAddressSegments', Target_Address_Segments)` adds and defines an AXI4 Master interface for a Xilinx Vivado reference design.

```
addAXI4MasterInterface('InterfaceID',
Interface_ID,'InterfaceConnection',Interface_Connection, Name,Value)
```

adds and defines an AXI4 Master interface for an Qsys reference design, with additional options specified by one or more `Name,Value` pair arguments.

`addAXI4MasterInterface('InterfaceID', Interface_ID,'InterfaceConnection', Interface_Connection,'TargetAddressSegments', Target_Address_Segments, Name,Value)` adds and defines an AXI4 Master interface for a Xilinx Vivado reference design, with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Interface_ID` — AXI4 Master interface name

`'AXI4 Master'` (default) | character vector

Name of the AXI4 Master interface that you add to the reference design, specified as a character vector. If you create multiple AXI4 Master interfaces, make sure that you use unique names for each interface.

Example: `'AXI4 Master 1'`

`Interface_Connection` — Reference design port name

`''` (default) | character vector

Name of the reference design port that is connected to the AXI4 Master interface, specified as a character vector.

Example: `'axi_interconnect_1/S01_AXI'`

`Target_Address_Segments` — Reference design address segments

`''` (default) | character vector

Target address segment of the Xilinx Vivado reference design, specified as a character vector.

Example: `'{{'mig_7series_0/memmap/ memaddr',hex2dec('40000000'),hex2dec('40000000')}}'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:

`ReadSupport` — AXI4 Master read interface support
`'true'` (default) | `'false'`

Specify whether you want the AXI4 Master interface to support a read channel as a `Boolean`.

Example: `'ReadSupport','true'` specifies support for an AXI4 Master read interface connection.

`WriteSupport` — AXI4 Master write interface support
`'true'` (default) | `'false'`

Specify whether you want the AXI4 Master interface to support a write channel as a `Boolean`.

Example: `'WriteSupport','true'` specifies support for an AXI4 Master write interface connection.

`MaxDataWidth` — Maximum data width
128 (default) | `Integer`

Maximum width for the `Data` signal that is transferred across the AXI4 Master interface, specified as an integer.

Example: `'MaxDataWidth',32` specifies maximum data width of 32 bits.

`AddrWidth` — Address width
32 (default) | `Integer`

Width of the AXI4 Master interface read and write addresses, specified as an integer.

Example: `'AddrWidth',32` specifies an address size of 32 bits.

`DefaultReadBaseAddr` — Starting read address
0 (default) | `Integer`

Default starting address of the AXI4 Master read interface, specified as an integer.

Example: `'DefaultReadBaseAddr',hex2dec('40000000')` specifies `hex2dec('40000000')` as the starting read address.

`DefaultWriteBaseAddr` — Starting write address
0 (default) | `Integer`

Default starting address of the AXI4 Master write interface, specified as an integer.

Example: `'DefaultReadBaseAddr',hex2dec('41000000')` specifies `hex2dec('41000000')` as the starting write address.

See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addClockInterface`

Topics

"Model Design for AXI4 Master Interface Generation"
Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2017b

addAXI4SlaveInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add and define AXI4 slave interface

Syntax

```
addAXI4SlaveInterface('InterfaceConnection',
ref_design_port,'BaseAddress',base_addr)
addAXI4SlaveInterface('InterfaceConnection',
ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace',
master_addr_space)
addAXI4SlaveInterface('InterfaceConnection',
ref_design_port,'BaseAddress',base_addr,Name,Value)
addAXI4SlaveInterface('InterfaceConnection',
ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace',
master_addr_space,Name,Value)
```

Description

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port,'BaseAddress',base_addr)` adds and defines an AXI4 interface for an Altera reference design, or an AXI4 or AXI4-Lite interface for a Xilinx ISE reference design.

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace', master_addr_space)` adds and defines an AXI4 or AXI4-Lite interface for Xilinx Vivado reference designs.

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port,'BaseAddress',base_addr,Name,Value)` adds and defines an AXI4 interface for an Altera reference design, or an AXI4 or AXI4-Lite interface for a Xilinx ISE reference design, with additional options specified by one or more `Name,Value` pair arguments.

`addAXI4SlaveInterface('InterfaceConnection', ref_design_port,'BaseAddress',base_addr,'MasterAddressSpace', master_addr_space,Name,Value)` adds and defines an AXI4 or AXI4-Lite interface for Xilinx Vivado reference designs, with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`ref_design_port` — Reference design port name

`''` (default) | character vector

Reference design port that is connected to the AXI4 or AXI4-Lite interface, specified as a character vector.

Example: `'axi_interconnect_0/M00_AXI'`

`base_addr` — Base address

`''` (default) | character vector

Base address for AXI4 or AXI4-Lite slave interface, specified as a character vector.

Example: `'0x40010000'`

`master_addr_space` — Master interface address space (Vivado only)

`''` (default) | character vector

Address space of the master interface connected to this slave interface, specified as a character vector. For Vivado reference designs only.

Example: `'processing_system7_0/Data'`

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`.

`InterfaceType` — Interface type

`{'AXI4-Lite','AXI4'}` (default) | `'AXI4'` | `'AXI4-Lite'`

Type of interface connection, specified as a `character vector` or a `cell array` of character vectors.

Example: `'InterfaceType','AXI4-Lite'` specifies an `'AXI4-Lite'` interface type connection.

`InterfaceID` — Interface name

`{'AXI4-Lite','AXI4'}` (default) | character vector

Name of the interface, specified as a character vector. When you provide the `InterfaceID`, `InterfaceType` must be set to either `'AXI4'` or `'AXI4-Lite'`.

Example: `'InterfaceID','MyAXI4','InterfaceType','AXI4'` specifies interface name as `'MyAXI4'` and interface type as `'AXI4'`.

Tips

- Before running this method, you must run the `hdlcoder.ReferenceDesign.addClockInterface` method.
- The `addAXI4SlaveInterface` method is optional. You can define your own custom reference design without the AXI4 slave interface.

See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addClockInterface`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2015a

addInternalIOInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add and define internal IO interface between generated IP core and existing IP cores

Syntax

```
addInternalIOInterface('InterfaceID',interface_name,'InterfaceType',
interface_type,'PortName',port_name,'PortWidth',
port_width,'InterfaceConnection',interface_connection)
```

Description

`addInternalIOInterface('InterfaceID',interface_name,'InterfaceType', interface_type,'PortName',port_name,'PortWidth', port_width,'InterfaceConnection',interface_connection)` adds and defines an internal IO interface between the generated IP core and other IP cores in the reference design.

In the HDL Workflow Advisor, if you target a custom reference design that has an internal IO interface, you must map a DUT port to the internal IO interface. In the Target Platform Interface Table, you cannot leave the internal IO interface unassigned.

Input Arguments

`interface_name` — Custom internal IO interface name

`''` (default) | character vector

Custom internal IO interface name, specified as a character vector. In the HDL Workflow Advisor, when you select the custom reference design, this name appears as an option in the Target Platform Interface Table.

Example: `'MyCustomInternalInterface'`

`interface_type` — Interface direction
`'IN'` (default) | `'OUT'`

Interface direction relative to the generated IP core, specified as a character vector.

For example, if the interface is an input to the generated IP core, set `interface_type` to `'IN'`.

`port_name` — Port name
`''` (default) | character vector

Name of generated IP core port in the HDL code, specified as a character vector.

Example: `'MyIPCoreInternalIOInterfacePort'`

`port_width` — Port bit width
8 (default) | integer

Bit width of generated IP core port, specified as an integer.

`interface_connection` — Internal IO interface connection
`''` (default) | character vector

Internal IO interface port to connect with generated IP core port, specified as a character vector. The internal IO interface port is an existing port in the reference design. Its port bit width must match `port_width`.

Different synthesis tools have different formats for the internal IO interface port.

Synthesis Tool	Format Example
Altera Quartus II	`'internal_ip_0.In0'`
Xilinx Vivado	`'internal_ip_0/In0'`
Xilinx ISE	`'internal_In0'`

Example: `'internal_ip_0.In0'`

Example: `'internal_ip_0/In0'`

Example: `'internal_In0'`

See Also

`hdlcoder.ReferenceDesign`

Topics

“Define and Register Custom Board and Reference Design for SoC Workflow”
“Register a Custom Board”
“Register a Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2015b

addClockInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add clock and reset interface

Syntax

```
addClockInterface('ClockConnection',clock_port,'ResetConnection',
reset_port)
addClockInterface('ClockConnection',clock_port,'ResetConnection',
reset_port,Name,Value)
```

Description

`addClockInterface('ClockConnection',clock_port,'ResetConnection', reset_port)` adds a clock and reset interface to an `hdlcoder.ReferenceDesign` object.

`addClockInterface('ClockConnection',clock_port,'ResetConnection', reset_port,Name,Value)` adds a clock and reset interface to the `hdlcoder.ReferenceDesign` object with additional options specified by one or more `Name,Value` pair arguments. When you specify these arguments, in the HDL Workflow Advisor, HDL Coder adds a **Set Target Frequency** task. To modify the output clock frequency setting in the reference design clock wizard, in this task, specify the **Target Frequency (MHz)**.

Input Arguments

`clock_port` — Clock port name

`''` (default) | character vector

Reference design port that is connected to the IP core clock port, specified as a character vector.

Example: `'processing_system7_1/FCLK_CLK0'`

`reset_port` — Reset port name
`''` (default) | character vector

Reference design port that is connected to the IP core reset port, specified as a character vector.

Example: `'proc_sys_reset/peripheral_aresetn'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`DefaultFrequencyMHz` — The default frequency in MHz
`0` (default) | integer

The default clock frequency in MHz of the clock module IP in the reference design, specified as an integer. When you open the HDL Workflow Advisor, HDL Coder populates this information for **Default (MHz)** in the **Set Target Frequency** task.

Example: `'DefaultFrequencyMHz', 50` specifies the default frequency as 50 MHz.

`MinFrequencyMHz` — The minimum frequency in MHz
`0` (default) | integer

The minimum clock frequency in MHz of the clock module IP in the reference design, specified as an integer.

Example: `'MinFrequencyMHz', 5` specifies the minimum clock frequency as 5 MHz.

`MaxFrequencyMHz` — The maximum frequency in MHz
`0` (default) | integer

The maximum clock frequency in MHz of the clock module IP in the reference design, specified as an integer. In the HDL Workflow Advisor, HDL Coder sets the **Frequency Range (MHz)** in the **Set Target Frequency** task based on the `MinFrequencyMHz` and `MaxFrequencyMHz` values that you specify.

Example: `'MaxFrequencyMHz', 500` specifies the maximum clock frequency as 500 MHz.

`ClockNumber` — Clock output port number
`1` (default) | integer

Port number of the clock output from the clock module IP in the reference design, specified as an integer.

Example: `'ClockNumber', 2` specifies to use the second output port in the clock module IP as the clock port.

`ClockModuleInstance` — Clock module name
`'clk_wiz_0'` (default) | character vector

The name of the clock module IP in the reference design, specified as a character vector.

Example: `'ClockModuleInstance', 'clk_wiz_1'` specifies `clk_wiz_1` as the name of the clock module IP.

See Also

`hdlcoder.ReferenceDesign` |
`hdlcoder.ReferenceDesign.addAXI4SlaveInterface`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2015a

addCustomEDKDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify Xilinx EDK MHS project file

Syntax

```
addCustomEDKDesign('CustomEDKMHS',mhs_project_file)
```

Description

`addCustomEDKDesign('CustomEDKMHS',mhs_project_file)` specifies the MHS project file that contains the Xilinx EDK embedded system design. Use this method if your synthesis tool is Xilinx ISE.

Input Arguments

`mhs_project_file` — MHS project file

character vector

MHS project file for Xilinx EDK embedded system design, specified as a character vector.

Example: `'system.mhs'`

Tips

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Altera Quartus II, use the `addCustomQsysDesign` method.

See Also

`hdlcoder.ReferenceDesign` |
`hdlcoder.ReferenceDesign.addCustomQsysDesign` |
`hdlcoder.ReferenceDesign.addCustomVivadoDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a

addCustomQsysDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify Altera Qsys project file

Syntax

```
addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file)
```

Description

`addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file)` specifies the Qsys project file that contains the Altera Qsys embedded system design. Use this method if your synthesis tool is Altera Quartus II.

Input Arguments

`qsys_project_file` — Qsys project file

character vector

Qsys project file for Altera Qsys embedded system design, specified as a character vector.

Example: `'system_soc.qsys'`

Tips

- If your synthesis tool is Xilinx Vivado, use the `addCustomVivadoDesign` method.
- If your synthesis tool is Xilinx ISE, use the `addCustomEDKDesign` method.

See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addCustomEDKDesign` | `hdlcoder.ReferenceDesign.addCustomVivadoDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a

addCustomVivadoDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify Xilinx Vivado exported block design Tcl file

Syntax

```
addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)
```

Description

`addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)` specifies the exported block design Tcl file that contains the Xilinx Vivado embedded system design. Use this method if your synthesis tool is Xilinx Vivado.

Input Arguments

`bd_tcl_file` — Block design Tcl file

character vector

Block design Tcl file that you exported from your Xilinx Vivado embedded system design project, specified as a character vector. The Tcl file name must be the same as the Vivado block diagram name.

Example: `'system_top.tcl'`

Tips

- If your synthesis tool is Xilinx ISE, use the `hdlcoder.ReferenceDesign.addCustomEDKDesign` method.
- If your synthesis tool is Altera Quartus II, use the `hdlcoder.ReferenceDesign.addCustomQsysDesign` method.

See Also

`hdlcoder.ReferenceDesign` | `hdlcoder.ReferenceDesign.addCustomEDKDesign` | `hdlcoder.ReferenceDesign.addCustomQsysDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a

addIPRepository

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Include IP modules from your IP repository folder in your custom reference design

Syntax

```
addIPRepository('IPListFunction',IP_list_function)
addIPRepository('IPListFunction',IP_list_function,Name,Value)
```

Description

`addIPRepository('IPListFunction',IP_list_function)` adds IP modules that are in the IP repository folder to your reference design project.

`addIPRepository('IPListFunction',IP_list_function,Name,Value)` adds IP modules that are in the IP repository folder to your reference design project with additional options specified by one or more `Name,Value` pair arguments.

Before you use this method, define the IP list function that points to the IP modules in the repository folder. To learn more, see “Define and Add IP Repository to Custom Reference Design”.

Input Arguments

`IP_list_function` — Name and path to the function that points to the IP repository

`''` (default) | character vector

Name and path to the function that points to IP modules in the IP repository folder to add to the reference design project, specified as a character vector.

Example: `'adi.hdmi.vivado.hdlcoder_video_iplist'`

Example: `'mathworks.hdlcoder.vivado.hdlcoder_video_iplist'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`NotExistMessage` — Error message to display if IP function is not found

`''` (default) | character vector

Error message that you create to be displayed if IP list function is not found on the MATLAB path, specified as a character vector.

Example: `'IP repository cannot be found'`

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Define and Add IP Repository to Custom Reference Design"
"Board and Reference Design Registration System"
"Register a Custom Board"
"Register a Custom Reference Design"

Introduced in R2017a

addParameter

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add and define custom parameters for your reference design

Syntax

```
addParameter('ParameterID',parameter_name,'DisplayName',
display_name,'DefaultValue',default_value)
addParameter('ParameterID',parameter_name,'DisplayName',
display_name,'DefaultValue',default_value,Name,Value)
```

Description

`addParameter('ParameterID',parameter_name,'DisplayName', display_name,'DefaultValue',default_value)` adds and defines a custom parameter for your reference design with a text box that displays the default value of the parameter.

`addParameter('ParameterID',parameter_name,'DisplayName', display_name,'DefaultValue',default_value,Name,Value)` adds and defines a custom parameter for your reference design with additional options specified by one or more `Name,Value` pair arguments.

The custom parameters are optional. In the HDL Workflow Advisor **Set Target Reference Design** task, HDL Coder populates the Reference design parameters section with the custom parameters and the options that you specify.

Input Arguments

`parameter_name` — Custom parameter name

`''` (default) | character vector

Custom parameter name, specified as a character vector.

Example: `'DUTPath'`

Example: `'ChannelMapping'`

`display_name` — Custom parameter display name
`''` (default) | character vector

Name that you want to display for the custom parameter in the HDL Workflow Advisor, specified as a character vector. This name appears in the **Reference design parameters** section in the **Set Target Reference Design** task.

Example: `'DUT Path'`

Example: `'Channel Mapping'`

`default_value` — Custom parameter default value
`''` (default) | character vector

Default value to set for the custom parameter, specified as a character vector. In the **Set Target Reference Design** task in the HDL Workflow Advisor, HDL Coder displays the default value of the custom parameter inside a text box.

Example: `'1'`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`ParameterType` — Parameter widget
`hdlcoder.ParameterType.Edit` (default) | `hdlcoder.ParameterType.DropDown` | `hdlcoder.ParameterType.Edit`

Specify the widget type to use for the parameter values. By default, the`ParameterType` is a text box. If you specify the drop-down list for `ParameterType`, use the `Choice` property to list the parameter values as a cell array of character vectors.

Example: `'ParameterType',hdlcoder.ParameterType.Dropdown` specifies a drop-down list with the values that the parameter can take.

`Choice` — Choice of parameter values

`''` (default) | cell array of character vectors

The list of choices that you can specify for the custom parameter, specified as a cell array of character vectors. To specify this list, set `ParameterType` to `hdlcoder.ParameterType.Dropdown`.

Example: `'ParameterType',hdlcoder.ParameterType.Dropdown,'Choice', {'Rx', 'Tx'}` specifies a drop-down list with `Rx` and `Tx` as the drop-down values.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Define Custom Parameters and Callback Functions for Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2016b

CallbackCustomProgrammingMethod

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor

Syntax

```
CallbackCustomProgrammingMethod
```

Description

`CallbackCustomProgrammingMethod` registers a function handle for the callback function that gets executed when running the **Program Target Device** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax to register the function handle:

```
hRD.CallbackCustomProgrammingMethod = @my_reference_design.callback_CustomProgrammingMe
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostBuildBitstream`, located in the reference design package folder, `+my_reference_design`.

With this callback function, you can specify a custom programming method to program the target device. This example code shows how to create the callback function.

```
function [status, log] = callback_CustomProgrammingMethod(infoStruct)
% Reference design callback function for custom programming method
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
```

```
% infoStruct.BitstreamPath: the path to the generated FPGA bitstream file
% infoStruct.ToolProjectFolder: the path to synthesis tool project folder
% infoStruct.ToolProjectName: the synthesis tool project name
% infoStruct.ToolCommandString: the command for running a tcl file
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string
status = true;
log = sprintf('Run custom programming method callback...\n');


% Enter your commands for custom programming here
% ...
% ...

end
```

In the HDL Workflow Advisor, HDL Coder selects the custom programming method to program the target SoC device. If you do not specify the custom programming method, HDL Coder provides `JTAG`, `Ethernet`, and `Download` as the options to program the target device.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016a

EmbeddedCoderSupportPackage

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify whether to use an Embedded Coder support package

Syntax

```
EmbeddedCoderSupportPackage
```

Description

`EmbeddedCoderSupportPackage` specifies if you want to use an Embedded Coder® support package for your reference design. Use this parameter if you are targeting a standalone FPGA board or an SoC device such as the Xilinx Zynq®-7000 platform.

If you are targeting a standalone FPGA board, the reference designs do not require an Embedded Coder support package. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax:

```
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.None;
```

When you are not using the support package, HDL Coder does not have the **Generate Software Interface Model** task in the HDL Workflow Advisor.

If you are targeting SoC devices, use this syntax depending on whether you are using an Altera SoC or a Xilinx Zynq platform.

```
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.Zynq;
hRD.EmbeddedCoderSupportPackage = hdlcoder.EmbeddedCoderSupportPackage.AlteraSoC;
```

See Also

`hdlcoder.ReferenceDesign`

Topics

“IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705”
“IP Core Generation Workflow for Standalone FPGA Devices”
“Register a Custom Board”
“Register a Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

PostBuildBitstreamFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed after Build FPGA Bitstream task in the HDL Workflow Advisor

Syntax

```
PostBuildBitstreamFcn
```

Description

`PostBuildBitstreamFcn` registers a function handle for the callback function that gets called at the end of the **Build FPGA Bitstream** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax to register the function handle:

```
hRD.PostBuildBitstreamFcn = @my_reference_design.callback_PostBuildBitstream;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostBuildBitstream`, located in the reference design package folder `+my_reference_design`.

With this callback function, you can specify custom settings when HDL Coder runs the build process and generates the bitstream. This example code shows how to create the callback function. The function displays the status after running the task, and the board and reference design information.

```
function [status, log] = callback_PostBuildBitstream(infoStruct)
% Reference design callback run at the end of the task Build FPGA Bitstream
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
```

```
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.BitstreamPath: the path to generated FPGA bitstream file
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string


status = false;
log = sprintf('Run post build bitstream callback\n%s\n%s\n', infoStruct.HDLModelDutPath

% Exporting the InfoStruct Contents
% ...
% ...

end
```

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

PostCreateProjectFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed after Create Project task in the HDL Workflow Advisor

Syntax

```
PostCreateProjectFcn
```

Description

`PostCreateProjectFcn` registers a function handle for the callback function that gets called at the end of the **Create Project** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax to register the function handle.

```
hRD.PostCreateProjectFcn = @my_reference_design.callback_PostCreateProject;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostCreateProject`, and is located in the reference design package folder `+my_reference_design`.

With this callback function, you can specify custom settings for reference design project creation. This example code shows how to create the callback function. The function exports the contents of the board and reference design object to a `PostCreateProjectInfo.txt` file, and validates that the project creation task ran successfully.

```
function [status, log] = callback_PostCreateProject(infoStruct)
% Reference design callback run at the end of the task Create Project
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
```

```
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ToolProjectFolder: the path to synthesis tool project folder
% infoStruct.ToolProjectName: the synthesis tool project name
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string

status = false;
log = sprintf('Run post create project callback\n%s', evalc('infoStruct'));

% Exporting the InfoStruct Contents
% ...
% ...


end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Create Project** task, it executes the callback function at the end of the task.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the reference design project creation.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

PostSWInterfaceFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for custom callback function that gets executed after Generate Software Interface Model task in the HDL Workflow Advisor

Syntax

```
PostSWInterfaceFcn
```

Description

`PostSWInterfaceFcn` registers a function handle for the callback function that gets executed at the end of the **Generate Software Interface Model** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, use this syntax to register the function handle.

```
hRD.PostSWInterfaceFcn = @my_reference_design.callback_PostSWInterface;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostSWInterface`, and is located in the reference design package folder `+my_reference_design`.

With this callback function, you can change the generated software interface model for the custom reference design.

This example code shows how to create the callback function. The function adds a DocBlock in the software interface model.

```
function [status, log] = callback_PostSWInterface(infoStruct)
% Reference design callback run at the end of the task
% Generate Software Interface Model
%
% infoStruct: information in structure format
```

```
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.SWModelDutPath: the block path to the SW interface subsystem
%
% feature controlled by IPCoreSoftwareInterfaceLibrary
% infoStruct.SWLibBlockPath: the block path to the SW interface library block
% infoStruct.SWLibFolderPath: the folder path to the SW interface library
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string


status = true;
log = '';
swDutPath = infoStruct.SWModelDutPath;
add_block(['simulink/Model-Wide', char(10), 'Utilities/DocBlock'], sprintf('%s/DocBlock

end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Generate Software Interface Model** task, it executes the callback function at the end of the task.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for software interface model generation.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

PostTargetInterfaceFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed after Set Target Interface task in the HDL Workflow Advisor

Syntax

```
PostTargetInterfaceFcn
```

Description

`PostTargetInterfaceFcn` registers a function handle for the callback function that gets called at the end of the **Set Target Interface** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, then use this syntax to register the function handle.

```
hRD.PostTargetInterfaceFcn = @my_reference_design.callback_PostTargetInterface;
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostTargetInterface`, and is located in the reference design package folder `+my_reference_design`.

With this callback function, you can enable custom validations. This example code shows how to create the callback function. If the custom parameter `DUTPath` is set to `Rx`, the function validates that the reference design does not support the `LEDs General Purpose [0:7]` interface.

```
function callback_PostTargetInterface(infoStruct)
% Reference design callback run at the end of the task Set Target Interface
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
```

```
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ProcessorFPGASynchronization: Processor/FPGA synchronization mode
% infoStruct.InterfaceStructCell: target interface table information
%                                 a cell array of structure, for example:
%                                 infoStruct.InterfaceStructCell{1}.PortName
%                                 infoStruct.InterfaceStructCell{1}.PortType
%                                 infoStruct.InterfaceStructCell{1}.DataType
%                                 infoStruct.InterfaceStructCell{1}.IOInterface
%                                 infoStruct.InterfaceStructCell{1}.IOInterfaceMapping


hRD = infoStruct.ReferenceDesignObject;
refDesignName = hRD.ReferenceDesignName;

% validate that when specific parameter is set to specific value, reference
% design does not support specific interface
paramStruct = infoStruct.ParameterStruct;
interfaceStructCell = infoStruct.InterfaceStructCell;
for ii = 1:length(interfaceStructCell)
    interfaceStruct = interfaceStructCell{ii};
    if strcmp(paramStruct.DutPath, 'Rx') && ...
            strcmp(interfaceStruct.IOInterface, 'LEDs General Purpose [0:7]')
        error('LEDs General Purpose [0:7] must not be used when the DUT path is Rx');
    end
end
end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Set Target Interface** task, it executes the callback function at the end of the task. If you specify `Rx` as the **DUT Path** and use the `LEDs General Purpose [0:7]` interface for your DUT port, the coder generates an error.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to enable custom validations on the DUT in your Simulink model.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

PostTargetReferenceDesignFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor

Syntax

```
PostTargetReferenceDesignFcn
```

Description

`PostTargetReferenceDesignFcn` registers a function handle for the callback function that gets called at the end of the **Set Target Reference Design** task in the HDL Workflow Advisor. If `hRD` is the reference design object that you construct with the `hdlcoder.ReferenceDesign` class, use this syntax to register the function handle:

```
hRD.PostTargetReferenceDesignFcn = @my_reference_design.callback_PostTargetReferenceDes
```

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is `callback_PostTargetReferenceDesign`, and is located in the reference design package folder `+my_reference_design`.

With the callback function, you can enable custom validations for your design. This example code shows how to create the callback function and validate that the reset type is synchronous.

```
function callback_PostTargetReferenceDesign(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in str
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
```

```
mdlName = bdroot(infoStruct.HDLModelDutPath);
hRD = infoStruct.ReferenceDesignObject;
refDesignName = hRD.ReferenceDesignName;

isResetSync = strcmpi(hdlget_param(mdlName, 'ResetType'), 'Synchronous');

% Reset must be synchronous
if ~isResetSync
    error('Invalid Reset type. Reset type must be synchronous');
end
end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Set Target Reference Design** task, it executes the callback function. If the reset type is not synchronous, the coder generates an error.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to enable custom validations on the DUT in your Simulink model.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
“Register a Custom Board”
“Register a Custom Reference Design”
“Define Custom Parameters and Callback Functions for Custom Reference Design”
“Board and Reference Design Registration System”

Introduced in R2016b

validateReferenceDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Check property values in reference design object

Syntax

```
validateReferenceDesign
```

Description

`validateReferenceDesign` checks that the `hdlcoder.ReferenceDesign` object has nondefault values for all required properties, and that property values have valid data types. This method does not check the correctness of property values for the target board. If validation fails, the software displays an error message.

See Also

`hdlcoder.ReferenceDesign`

Topics

Define and Register Custom Board and Reference Design for SoC Workflow
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

Introduced in R2015a