HDL Coder™

Reference

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

HDL Coder™ Reference

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

Contents

1 **Apps**

2 **Functions**

3 **Blocks**

4 **Classes for HDL Code Generation from Simulink**

5 **Functions for HDL Code Generation from MATLAB**

6 **Classes for HDL Code Generation from MATLAB**

7 **Shared Classes and Functions for HDL Code Generation from MATLAB and Simulink**

8 **Block Reference Page Examples**

Apps

HDL Coder

Generate HDL and SystemC code from MATLAB code

Description

The **HDL Coder** app generates synthesizable HDL code and SystemC code from MATLAB® code that is supported for hardware. You can generate VHDL, Verilog HDL, or SystemC 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.
- Select the code generation workflow as **MATLAB to HDL** or **MATLAB to SystemC**.
- Propose input data types or autodefine data types by specifying the MATLAB testbench file.
- Convert floating-point MATLAB code to fixed-point HDL code or SystemC code.
- Specify the target device and synthesis tool to deploy the generated HDL or SystemC 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.
- Verify the numerical behavior of generated SystemC code with HDL test bench.
- 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.
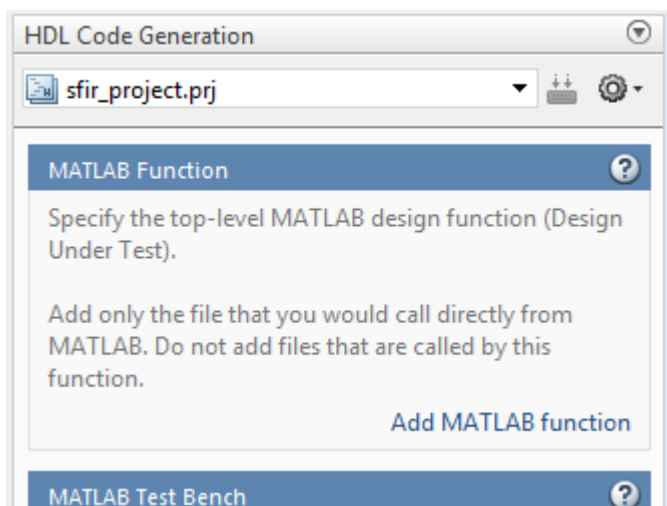
Open the HDL Coder App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the **HDL Coder** app icon.
- MATLAB command prompt: Enter `hdlcoder`.

Examples

- "Basic HDL Code Generation and FPGA Synthesis from MATLAB"
- "Get Started with MATLAB to SystemC Workflow Using HDL Coder App"

Programmatic Use

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

Version History
Introduced in R2012a

See Also

Apps
Fixed-Point Converter

Functions
`codegen`

Topics
"Basic HDL Code Generation and FPGA Synthesis from MATLAB"
"Get Started with MATLAB to SystemC Workflow Using HDL Coder App"
"Guidelines for Writing MATLAB Code to Generate Efficient HDL and SystemC Code"
"Create and Set Up Your Project"

Functions

checkhdl

Check subsystem or model for HDL code generation compatibility

Syntax

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

Description

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

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

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

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

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

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

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

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

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

The name-value pair arguments that you specify with `makehdl` and `makehdltb` can also be specified with `checkhdl`. For a list of name-value pair arguments, see "Name-Value Pair Arguments" on page 2-88.

`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 HDL Code Generation Compatibility

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 and Return Information on HDL Code Generation Compatibility

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'
```

Version History
Introduced in R2006b

See Also
`makehdl`

Topics
"Create HDL-Compatible Simulink Model"
"Check Your Model for HDL Compatibility"

hdl.treeprod

Product of array elements using tree architecture

Syntax

```
B = hdl.treeprod(A)
B = hdl.treeprod(A, 'all')
B = hdl.treeprod(A, dim)
```

Description

`B = hdl.treeprod(A)` returns the product of the elements of A.

- If A is a vector, then `hdl.treeprod(A)` returns the product of the elements.
- If A is a matrix, then `hdl.treeprod(A)` returns a row vector containing the products of each column.

The function `hdl.treeprod` uses a tree architecture to multiply elements. The tree architecture multiplication yields a shorter critical path, which leads to reduced latency when generating HDL code from a MATLAB Function block. When generating HDL code, the function `hdl.treeprod` reduces the amount of matching delays required to multiply elements.

`B = hdl.treeprod(A, 'all')` computes the product of all elements of A.

`B = hdl.treeprod(A, dim)` returns the product along dimension `dim`. For example, if A is a matrix, `hdl.treeprod(A,2)` is a column vector containing the products of each row.

Examples

Product of Vector Elements

Create a vector and compute the product of its elements.

```
A = 1:5;
B = hdl.treeprod(A)

B =
    120
```

Product of Matrix Elements

Create a matrix and compute the product of its elements.

```
A = [1 3 2; 4 2 5; 6 1 4]

A = 3×3

     1     3     2
```

```
        4       2       5
        6       1       4
```

```
B = hdl.treeprod(A,'all')
```

```
B =
     5760
```

Product of Matrix Columns

Create a matrix and compute the product of the elements in each column.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3
```

```
        1       3       2
        4       2       5
        6       1       4
```

```
B = hdl.treeprod(A)
```

```
B = 1×3
```

```
       24       6      40
```

Product of Matrix Rows

Create a matrix and compute the product of the elements in each row.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3
```

```
        1       3       2
        4       2       5
        6       1       4
```

```
B = hdl.treeprod(A,2)
```

```
B = 3×1
```

```
        6
       40
       24
```

Input Arguments

A — Input array
scalar | vector | matrix

Input array, specified as a scalar, vector, or 2-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(S,dim)` is `1`, while the sizes of the other dimensions remain the same.

Consider a two-dimensional input array, `A`:

- `sum(A,1)` operates on successive elements in the columns of `A` and returns a row vector of the sums of each column.

sum(A,1)

- `sum(A,2)` operates on successive elements in the rows of `A` and returns a column vector of the sums of each row.

sum(A,2)

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

B — Product array
scalar | vector | matrix

Product array, returned as a scalar, vector, or matrix.

The data type of B is the same data type as that of input A.

Version History

Introduced in R2022a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`hdl.treesum` | `prod`

hdl.treesum

Sum of array elements using tree architecture

Syntax

```
S = hdl.treesum(A)
S = hdl.treesum(A, 'all')
S = hdl.treesum(A, dim)
```

Description

`S = hdl.treesum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.

- If A is a vector, then `hdl.treesum(A)` returns the sum of the elements.
- If A is a matrix, then `hdl.treesum(A)` returns a row vector containing the sum of each column.

The function `hdl.treesum` uses a tree architecture to sum elements. The tree architecture summation yields a shorter critical path, which leads to reduced latency when generating HDL code from a MATLAB Function block. When generating HDL code, the function `hdl.treesum` reduces the amount of matching delays required to sum elements.

`S = hdl.treesum(A, 'all')` computes the sum of all elements of A.

`S = hdl.treesum(A, dim)` returns the sum along dimension `dim`. For example, if A is a matrix, then `hdl.treesum(A,2)` is a column vector containing the sum of each row.

Examples

Sum of Vector Elements

Create a vector and compute the sum of its elements.

```
A = 1:10;
S = hdl.treesum(A)

S = 55
```

Sum of Matrix Elements

Create a matrix and compute the sum of its elements.

```
A = [1 3 2; 4 2 5; 6 1 4]

A = 3×3

     1     3     2
     4     2     5
```

```
        6        1        4
```

```
S = hdl.treesum(A,'all')
```

```
S =
    28
```

Sum of Matrix Columns

Create a matrix and compute the sum of the elements in each column.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3

     1     3     2
     4     2     5
     6     1     4
```

```
S = hdl.treesum(A)
```

```
S = 1×3

    11     6    11
```

Sum of Matrix Rows

Create a matrix and compute the sum of the elements in each row.

```
A = [1 3 2; 4 2 5; 6 1 4]
```

```
A = 3×3

     1     3     2
     4     2     5
     6     1     4
```

```
S = hdl.treesum(A,2)
```

```
S = 3×1

     6
    11
    11
```

Input Arguments

A — Input array
scalar | vector | matrix

Input array, specified as a scalar, vector, or 2-D matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes
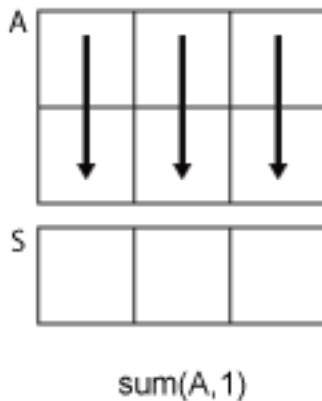
`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.
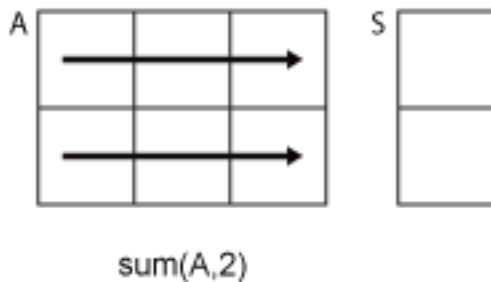
Dimension `dim` indicates the dimension whose length reduces to 1. The `size(S,dim)` is `1`, while the sizes of the other dimensions remain the same.

Consider a two-dimensional input array, `A`:

- `sum(A,1)` operates on successive elements in the columns of `A` and returns a row vector of the sums of each column.

sum(A,1)

- `sum(A,2)` operates on successive elements in the rows of `A` and returns a column vector of the sums of each row.

sum(A,2)

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`S` — Sum array
scalar | vector | matrix

Sum array, returned as a scalar, vector, or matrix.

The data type of S is the same data type as that of input A.

Version History
Introduced in R2022a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`hdl.treeprod` | `sum`

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 device under test (DUT) to be checked. HDL Workflow Advisor is not available in Simulink® Online™.

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

Display HDL Workflow Advisor for Subsystem

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

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

Display HDL Workflow Advisor for Subsystem using System Selector

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 your model window using these steps:

- Open the **HDL Coder** app from the **Apps** tab on the Simulink Toolstrip.
- Click the **Workflow Advisor** button.

Version History
Introduced in R2010a

See Also

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

hdlcoder.optimizeDesign

Automatic iterative HDL design optimization

Syntax

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

Description

`hdlcoder.optimizeDesign(model, optimizationCfg)` optimizes your generated HDL code based on the optimization configuration that 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 Vivado', ...
                   '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 Vivado');
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 Vivado', ...
                   '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 Vivado');
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 Vivado', ...
                   '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 Vivado');
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.

Use Critical Path Estimation for Faster Optimization

Use `CriticalPathEstimation` as the `TimingStrategy` to shorten the run time of iterative optimization for a model, `sfir_fixed`.

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 Vivado', ...
                   'SynthesisToolChipFamily','Zynq', ...
                   'SynthesisToolDeviceName','xc7z030', ...
                   'SynthesisToolPackageName','fbg484', ...
                   'SynthesisToolSpeedValue','-1')
```

Create an optimization configuration object `oc`.

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

Set the iteration limit to 10.

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

Set the timing strategy to `CriticalPathEstimation`.

```
oc.TimingStrategy = 'CriticalPathEstimation';
```

Optimize the model by using critical path estimation as the timing strategy.

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

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-1');

Iteration 0
Generate and synthesize HDL code ...
...
(CP ns) 9.94    (Constraint ns) 5.46    (Elapsed s) 4.80
Iteration 9
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 6.21 ns        Elapsed : 31.33 s
Iteration 0: (CP ns) 9.94    (Constraint ns) 5.46    (Elapsed s) 4.80
```

```
Iteration 1: (CP ns) 8.81    (Constraint ns) 5.46    (Elapsed s) 7.90
Iteration 2: (CP ns) 8.77    (Constraint ns) 7.14    (Elapsed s) 11.14
Iteration 3: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 14.74
Iteration 4: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 17.99
Iteration 5: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 21.25
Iteration 6: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 24.75
Iteration 7: (CP ns) 6.21    (Constraint ns) 7.14    (Elapsed s) 27.95
Iteration 8: (CP ns) 6.21    (Constraint ns) 5.96    (Elapsed s) 31.33
```

For comparison to the previous results, set the timing strategy to `Synthesis` (its default) and optimize the model by using `hdlcoder.optimizeDesign`.

```
oc.TimingStrategy = 'Synthesis';
hdlcoder.optimizeDesign(model,oc)

%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-1');


Iteration 0
Generate and synthesize HDL code ...
(CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 252.66
Iteration 3
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 5.49 ns        Elapsed : 704.51 s
Iteration 0: (CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 252.66
Iteration 1: (CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 479.88
Iteration 2: (CP ns) 5.49    (Constraint ns) 7.95    (Elapsed s) 704.51
```

	Synthesis Timing Strategy	Critical Path Estimation Timing Strategy
Total elapsed time, in seconds	704.51	31.33

When you use `CriticalPathEstimation` as the `TimingStrategy`, the time to run `hdlcoder.optimizeDesign` is shorter.

The critical path calculated by using `CriticalPathEstimation` as the `TimingStrategy` is the estimated critical path delay of the optimized model. The critical path calculated by using `Synthesis` as the `TimingStrategy` is the actual critical path delay of the optimized model.

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 Vivado', ...
                   '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'`

Version History
Introduced in R2014a

See Also

Classes
hdlcoder.OptimizationConfig

Functions
hdlcoder.supportedDevices

Topics
"Automatic Iterative Optimization"
"Tool and Device Parameters"

importhdl

Import Verilog code and generate Simulink model

Syntax

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

Description

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

`importhdl(FileNames)` imports the specified Verilog files and generates the corresponding Simulink model while removing unconnected components that do not directly contribute to the output.

`importhdl(FileNames,Name,Value)` imports the specified Verilog files and generates the corresponding Simulink model while removing unconnected components that do not directly contribute to the output, with options specified by one or more name-value pair arguments.

Examples

Generate Simulink Model From Single Verilog File

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

Specify Input Verilog File

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

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

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

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

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

parameter d = 2'b11;

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

endmodule
```

Import Verilog File

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

```
importhdl('comparator.v')

### Parsing <a href="matlab:edit('comparator.v')">comparator.v</a>.
### Top Module of the source: 'comparator'.
### Identified ClkName::clk.
### Identified RstName::rst.
### Hdl Import parsing done.
### Creating Target model comparator
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'comparator'.
### Laying out components.
### Working on hierarchy at ---> 'comparator/comparator'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting the model parameters.
### Generated model as C:\Temp\examples\examples\hdlcoder-ex77699673\hdlimport\comparator\compara
### HDL Import completed.
```

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

2-23

Examine Generated Simulink™ Model

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

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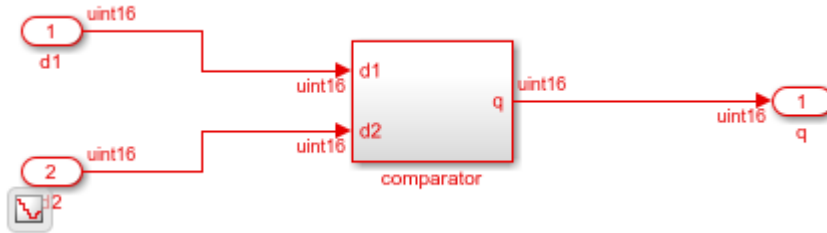

Generate Simulink Model From Multiple Verilog Files

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

Specify input Verilog File

Make sure that the input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three Verilog files that use module instantiation to form a hierarchical design. One module `example1.v` implements a simple sequential circuit based on an if-else condition. The other module `example2.v` implements a simple combinational arithmetic expression.

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

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

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

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

parameter g = 8'd5;

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

endmodule


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

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

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

endmodule
```

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

```
edit('example.v')
```

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


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


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


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


endmodule
```

Import Verilog Files

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

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

```
### Parsing <a href="matlab:edit('example.v')">example.v</a>.
### Parsing <a href="matlab:edit('example1.v')">example1.v</a>.
### Parsing <a href="matlab:edit('example2.v')">example2.v</a>.
### Top Module of the source: 'example'.
### Identified ClkName::clk.
### Hdl Import parsing done.
### Creating Target model example
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'example'.
### Laying out components.
### Working on hierarchy at ---> 'example/example'.
### Laying out components.
### Working on hierarchy at ---> 'example/example/example1'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Working on hierarchy at ---> 'example/example/example2'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting the model parameters.
### Generated model as C:\Temp\examples\examples\hdlcoder-ex56732899\hdlimport\example\example.s
### HDL Import completed.
```

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

Examine Generated Simulink™ Model

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

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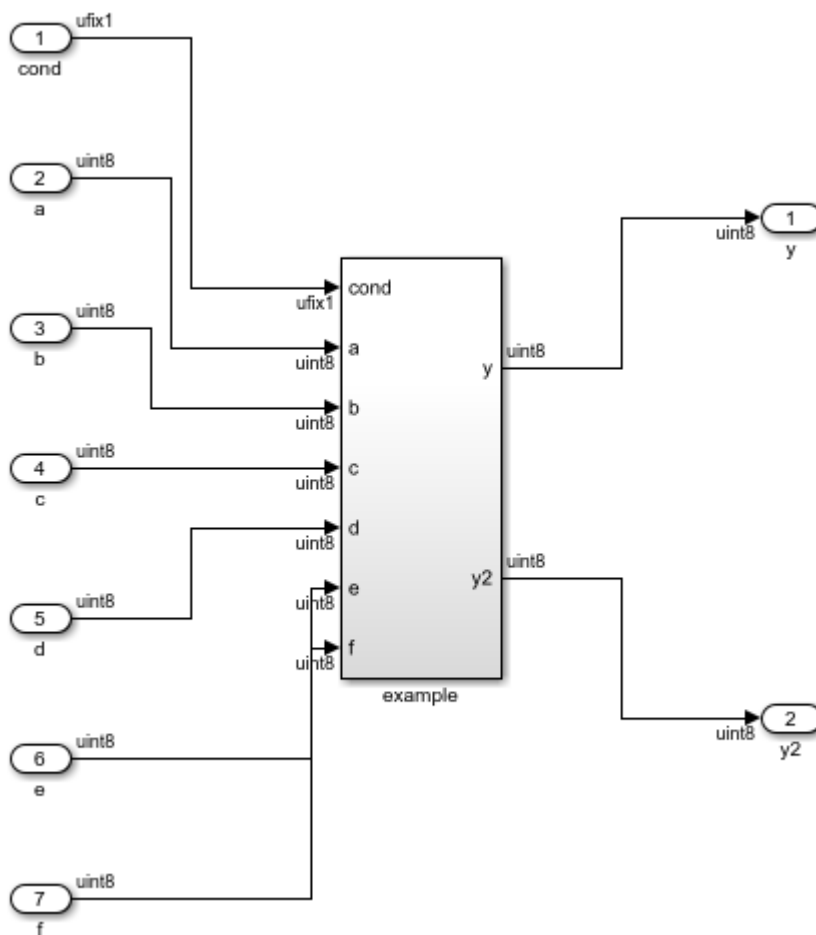

To avoid a division by zero, you can suppress the warning diagnostic before simulation.

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

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

2-27

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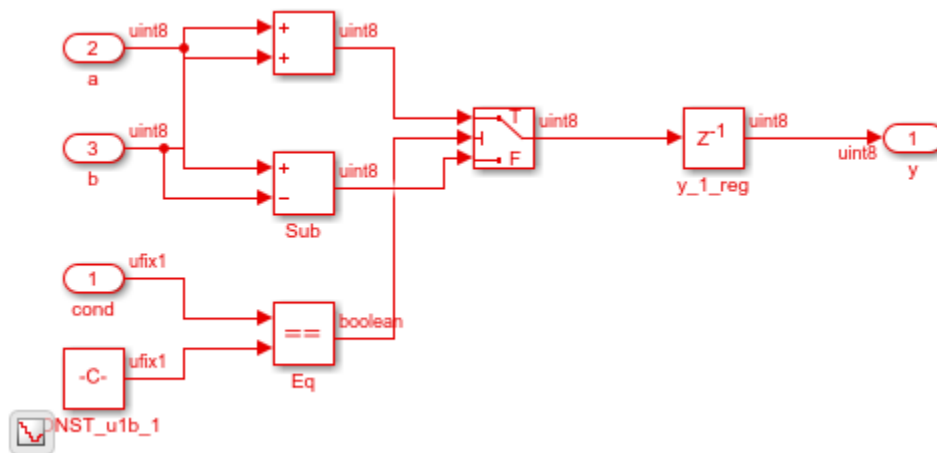


```
open_system('example/example/example2')
```
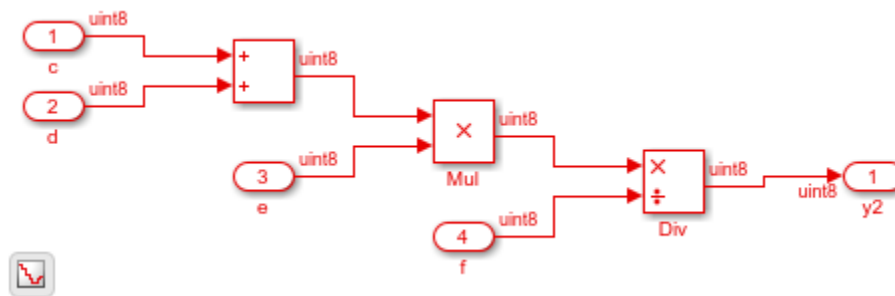

Generate Simulink Model From Verilog Files with BlackBox Modules

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

Specify input Verilog Files

Make sure that the input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three Verilog files that use module instantiation to form a hierarchical design. One module `sequentialexp.v` implements a simple sequential circuit based on an if-else condition. The other module `comditionalcomb.v` implements a simple combinational arithmetic expression.

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

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

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

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

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

endmodule


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

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

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

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

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

endmodule
```

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

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

input dataa, datab;
output datac;

assign datac = dataa + datab;

endmodule
```

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

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

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

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

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

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

endmodule
```

Import Verilog Files

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

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

### Parsing <a href="matlab:edit('blackboxtop.v')">blackboxtop.v</a>.
### Parsing <a href="matlab:edit('conditionalcomb.v')">conditionalcomb.v</a>.
### Parsing <a href="matlab:edit('sequentialexp.v')">sequentialexp.v</a>.
### Parsing <a href="matlab:edit('intelip.v')">intelip.v</a>.
### Top Module of the source: 'top'.
```

```
### Identified ClkName::clk.
### Hdl Import parsing done.
### Creating Target model top
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'top'.
### Laying out components.
### Working on hierarchy at ---> 'top/top'.
### Laying out components.
### Working on hierarchy at ---> 'top/top/u_comb'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Working on hierarchy at ---> 'top/top/u_seq'.
### Laying out components.
### Working on hierarchy at ---> 'top/top/u_seq/u_intelip'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Setting the model parameters.
### Generated model as C:\Temp\examples\examples\hdlcoder-ex63017378\hdlimport\top\top.slx.
### HDL Import completed.
```

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

Examine Generated Simulink™ Model

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

```
addpath('hdlimport/top')
open_system('top.slx')
set_param('top','SimulationCommand','update')
```
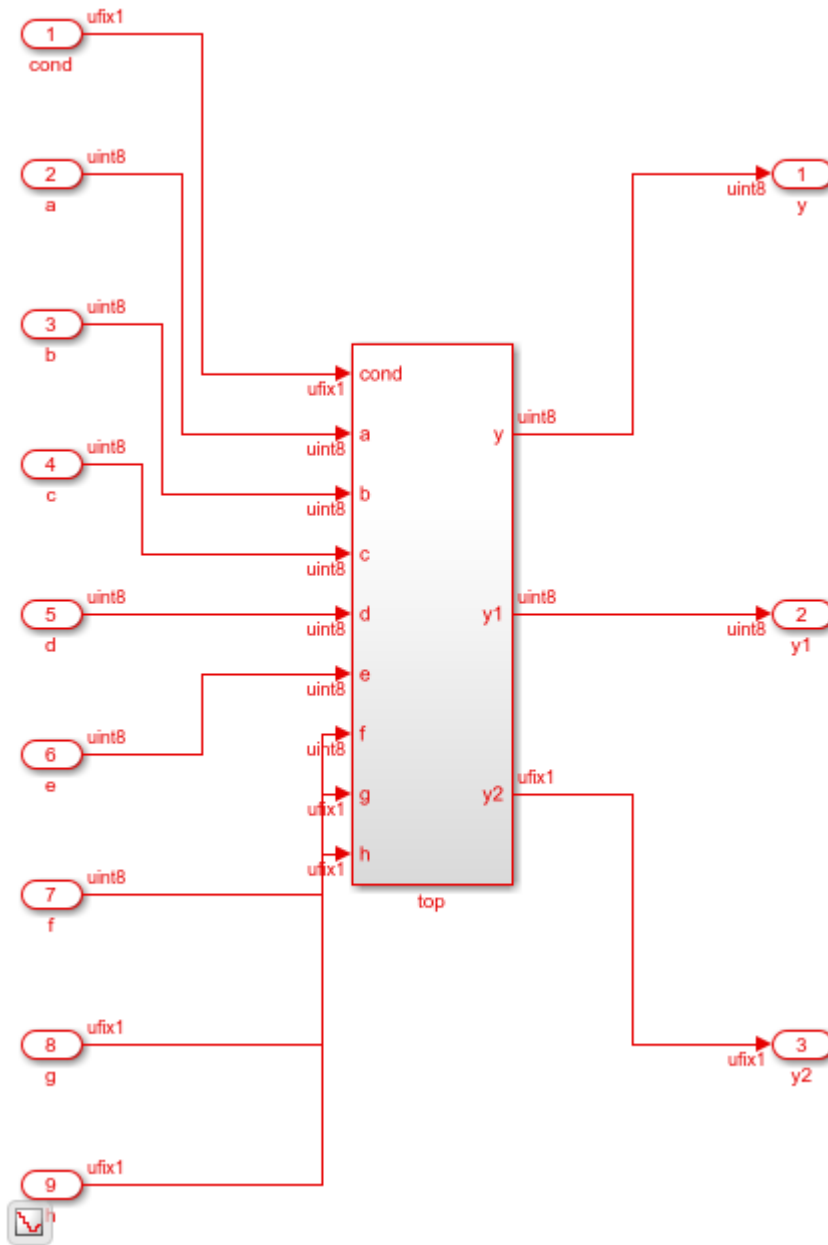
To avoid a division by zero, you can suppress the warning diagnostic before simulation.

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

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

```
open_system('top/top/u_comb')
```
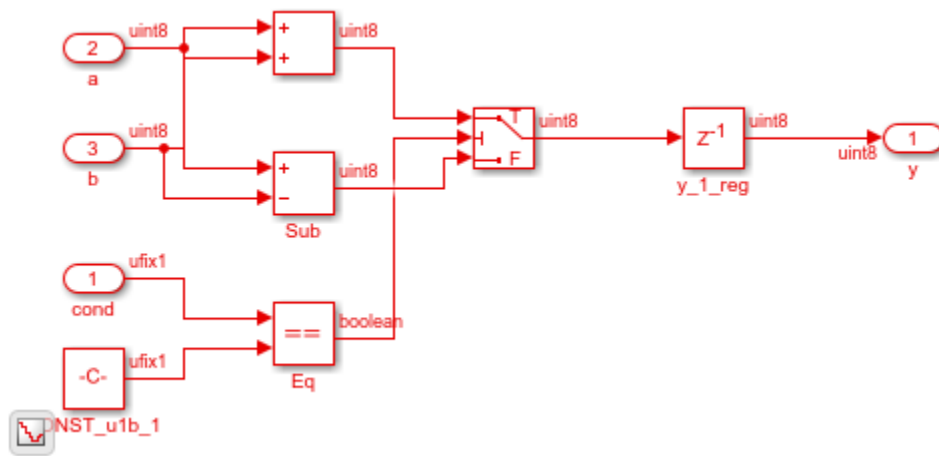
If you open the Subsystem that implements the sequential circuit, you can open the `u_intelip` Subsystem to see the blackbox implementation.

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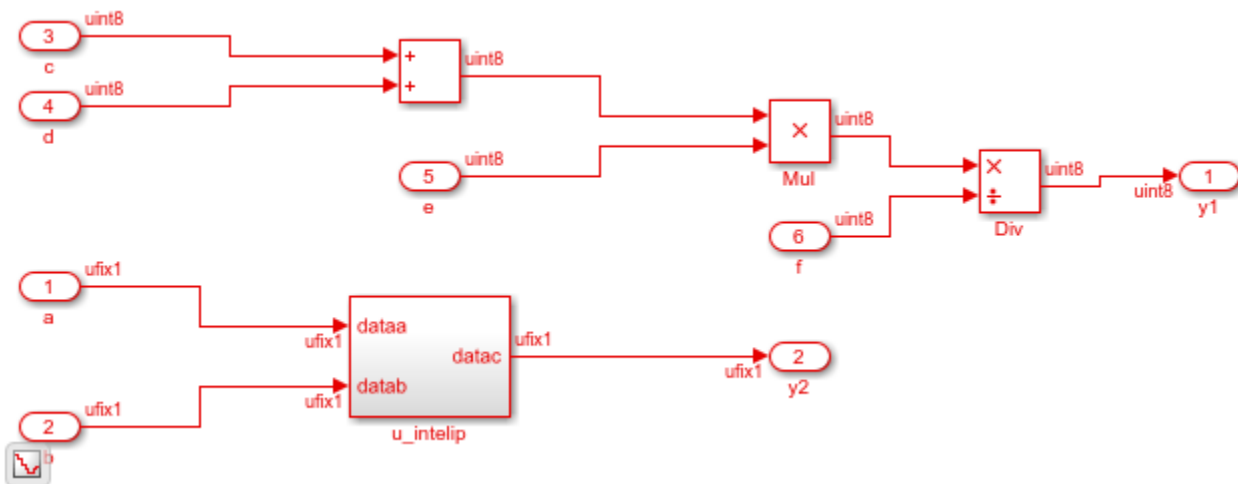


```
open_system('top/top/u_seq/u_intelip')
```
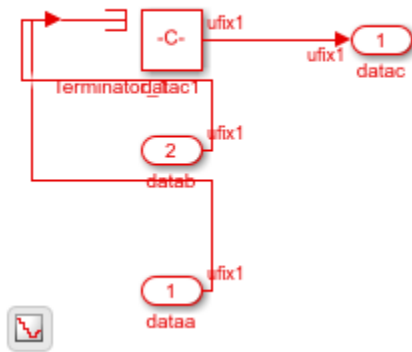
Generate Simulink Model from Verilog Code for Various Operators

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

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

Specify Input Verilog File

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

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

```verilog
`timescale 1 ns / 1 ns

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

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

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

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

   end

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

endmodule
```

Import Verilog File

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

```
importhdl('VerilogOperators.v')

### Parsing <a href="matlab:edit('VerilogOperators.v')">VerilogOperators.v</a>.
### Top Module of the source: 'VerilogOperators'.
### Hdl Import parsing done.
### Creating Target model VerilogOperators
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'VerilogOperators'.
### Laying out components.
### Working on hierarchy at ---> 'VerilogOperators/VerilogOperators'.
### Laying out components.
### Applying Dot Layout...
### Drawing block edges...
### Applying Dot Layout...
### Drawing block edges...
### Generated model as C:\Temp\examples\examples\hdlcoder-ex29847655\hdlimport\VerilogOperators\
### HDL Import completed.
```

HDL import parses the input file and displays messages of the import process in the MATLAB™ Command Window. The import provides a link to the generated Simulink™ model

`VerilogOperators.slx`. The generated model uses the same name as the top module in the input Verilog file.

Examine Generated Simulink™ Model

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

```
addpath('hdlimport/VerilogOperators')
open_system('VerilogOperators.slx')
sim('VerilogOperators.slx')
```
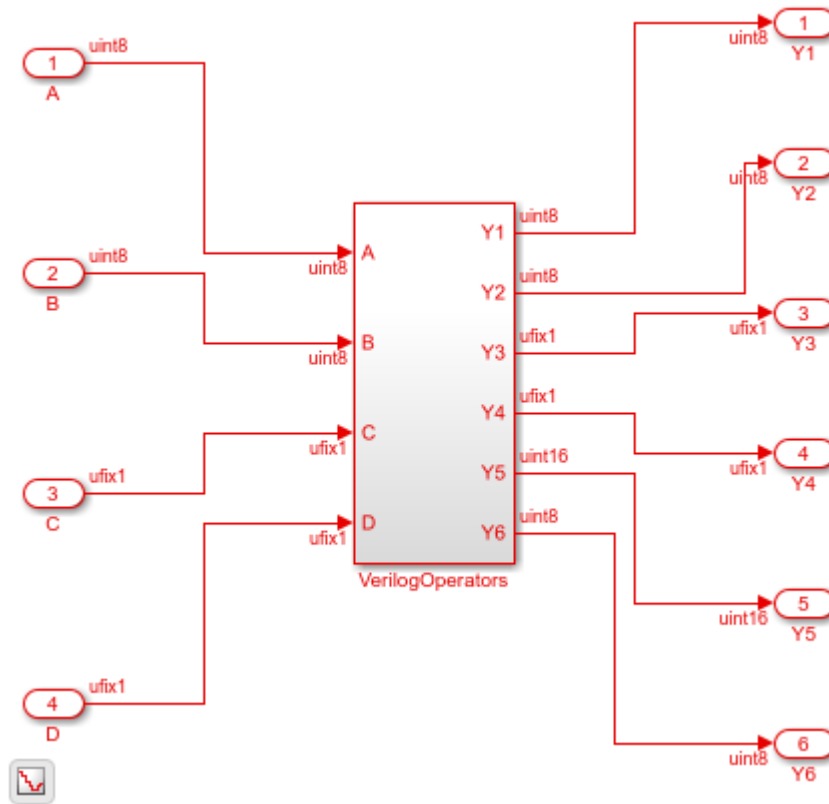
Implicit Data Type Conversion when Importing Verilog Code

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

Specify input Verilog File

Make sure that input HDL files do not contain any syntax errors, are synthesizable, and use constructs that are supported by HDL import. For example, this code shows three Verilog files that use module instantiation to form a hierarchical design. The modules `NG1_implicit.v` and `round_const.v` perform implicit data type conversion.

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

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

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

parameter IN_B_AND = 5'b10011;

assign OUT_A = IN_A == IN_B_AND & IN_B;

endmodule


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

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

always@(i) begin
  rc = 0;
end

endmodule
```

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

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

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

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

NG1 NG1(A, B, Y1);

rconst rconst(C, Y2);

endmodule
```

Import Verilog Files

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

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

```
### Parsing <a href="matlab:edit('implicit_top.v')">implicit_top.v</a>.
### Parsing <a href="matlab:edit('NG1_implicit.v')">NG1_implicit.v</a>.
### Parsing <a href="matlab:edit('round_constant.v')">round_constant.v</a>.
### Top Module name: 'top'.
Warning: Unused input port 'i' in 'rconst' module.
### Hdl Import parsing done.
### Creating Target model top
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'top'.
### Laying out components.
### Working on hierarchy at ---> 'top/top'.
### Laying out components.
### Working on hierarchy at ---> 'top/top/NG1'.
### Laying out components.
Configurable Subsystem block 'simulink/Ports & Subsystems/Configurable Subsystem' must be conver

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

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

Examine Generated Simulink™ Model

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

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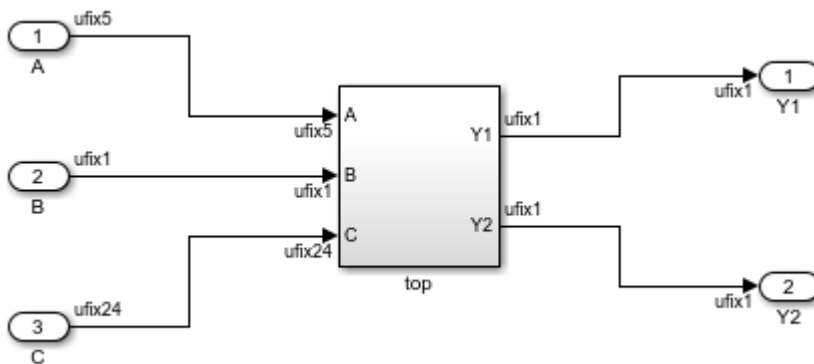

In the `rconst` Subsystem, one of the input ports is unconnected. It is good practice to avoid unterminated outputs by adding a Terminator block.

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

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

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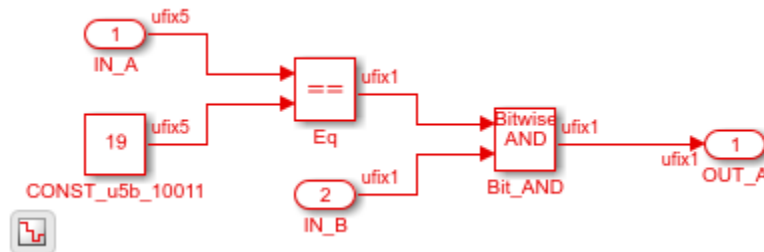


```
open_system('top/top/rconst')
```
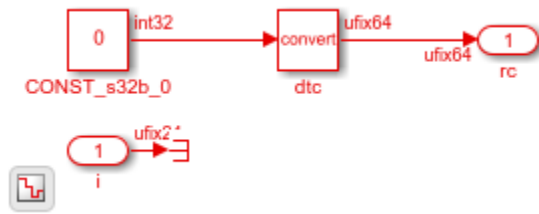
Generate Simulink Model from Verilog Code That Infers RAMs

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

Specify Input Verilog File

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

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

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

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

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

  reg  [7:0] ram1[7:0], ram2[7:0];

  // This is the first RAM block
  always @(posedge clk) begin
      if (enb) begin
        if (wr_en) begin
          ram1[wr_addr] <= wr_din;
        end
        rd_out1 <= ram1[rd_addr];
      end
    end

  // This is the second RAM block
  always @(posedge clk) begin
      if (enb) begin
        if (wr_en) begin
          ram2[wr_addr] <= wr_din;
        end
        rd_out2 <= ram1[rd_addr];
      end
    end

endmodule
```

Import Verilog File

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

```
importhdl('simple_dual_port_ram.v')

### Parsing <a href="matlab:edit('simple_dual_port_ram.v')">simple_dual_port_ram.v</a>.
### Top Module name: 'SimpleDualPortRAM'.
### Identified ClkName::clk.
```

```
### Hdl Import parsing done.
### Creating Target model SimpleDualPortRAM
### Generating Dot Layout...
### Start Layout...
### Working on hierarchy at ---> 'SimpleDualPortRAM'.
### Laying out components.
### Working on hierarchy at ---> 'SimpleDualPortRAM/SimpleDualPortRAM'.
### Laying out components.
Configurable Subsystem block 'simulink/Ports & Subsystems/Configurable Subsystem' must be convert

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

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

Examine Generated Simulink™ Model

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

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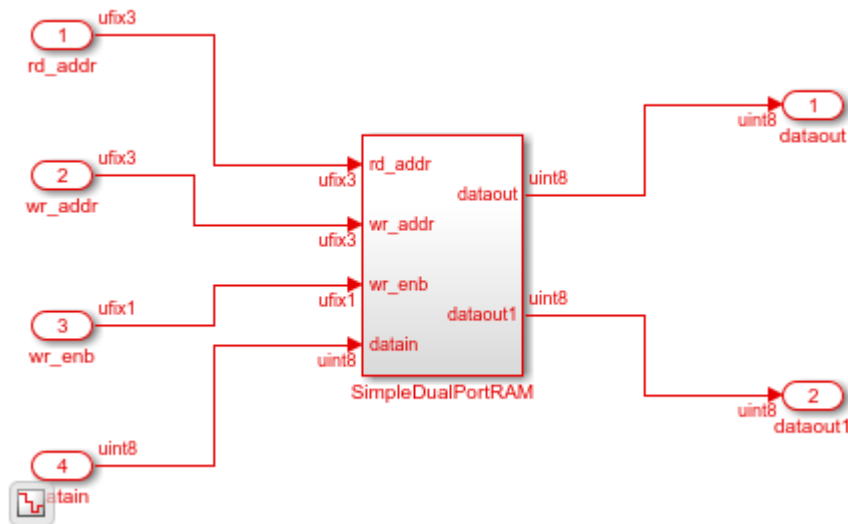

If you navigate the model, you see the Simple Dual Port RAM block.

```
open_system('SimpleDualPortRAM/SimpleDualPortRAM')
```
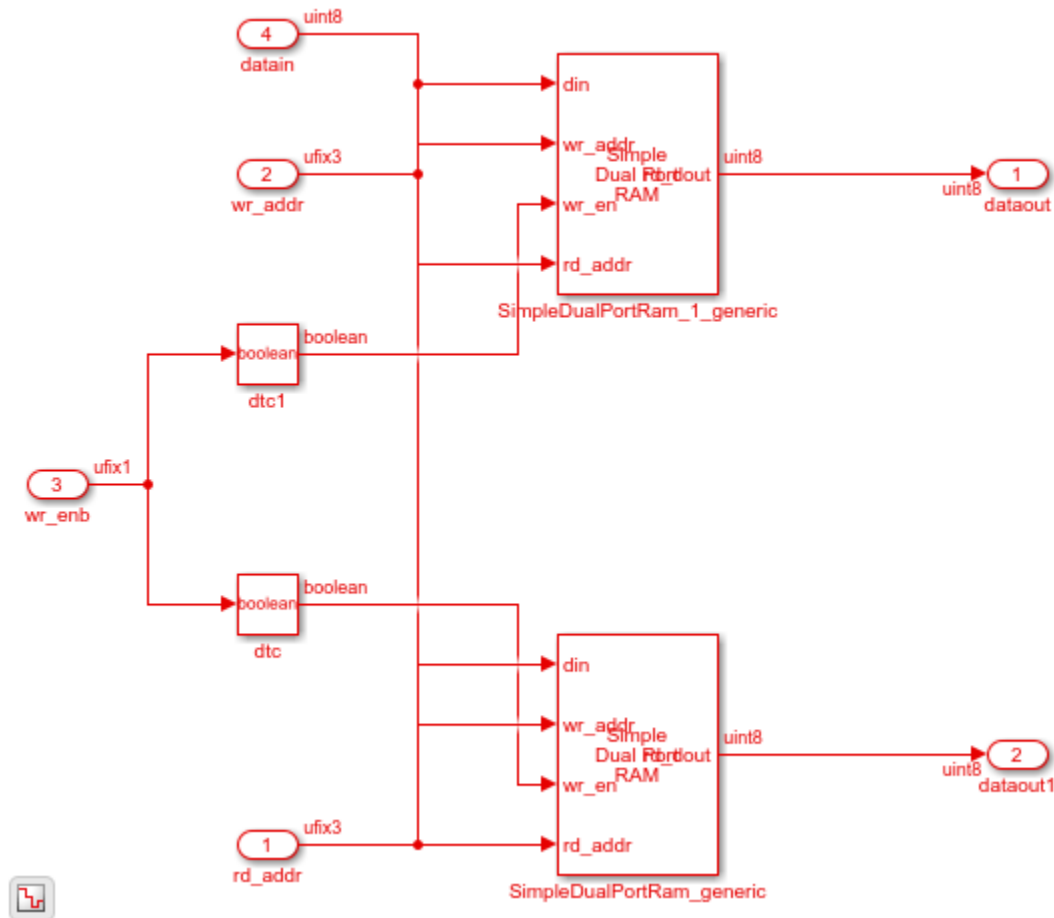
Input Arguments

FileNames — Names of HDL files to import
'Filename' | {'Filename1','Filename2',...,'FilenameN'} | 'Foldername'

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

- One HDL file, specify the file name as a character vector.
- Multiple HDL files, specify the file names as a cell array of character vectors.
- All HDL files in a folder, specify the folder name as a character vector.
- Multiple folders and combinations of files and folders, specified as cell array of character vectors. You can also use subfolders that contain recursive folders.

Example: `importhdl('example')` imports the specified Verilog file. If `example` is a subfolder in the current working folder, HDL import parses the subfolder `example` and module instantiations inside the subfolder, and then generates a Simulink model for all `.v` files. If `importhdl` cannot find a subfolder that has the name `example`, it searches the MATLAB path for HDL files that have the name `example`.

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

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

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

Example: `importhdl('subfolder')` imports all Verilog files under specified subfolder and generates the corresponding Simulink model. By default, `importhdl` parses subfolders that contain recursive folders.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

Language — Language of input HDL file
`'Verilog'` (default)

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

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

topModule — Name of top module or entity
`Identified by parsing input file` (default) | character vector | string scalar

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

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

clockBundle — Clock bundle names
`{'clock', 'reset', 'enable'}` (default) | cell array of character vectors

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

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

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

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

blackBoxModule — BlackBox module names
'' (default) | character vector | cell array of character vectors

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

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

autoPlace — Arrange blocks for improved layout
'on' (default) | 'off'

Automatically arrange blocks in the Simulink model generated by running importhdl. By default, autoPlace is on. importhdl then uses Simulink.BlockDiagram.arrangeSystem to improve the model layout by realigning, resizing, and moving blocks, and straightening signal lines.

Example: importhdl('example.v','autoPlace','on') imports the Verilog file example.v and generates the Simulink model with an enhanced model layout.

Limitations

The importhdl function is not supported on Mac platforms.

Version History
Introduced in R2018b

See Also
makehdl | checkhdl

Topics
"Generate Simulink Model From CORDIC Atan2 Verilog Code"
"Import Verilog Code and Generate Simulink Model"
"Supported Verilog Constructs for HDL Import"
"Verilog Dataflow Modeling with HDL Import"

hdlcoder.supportedDevices

Show supported target hardware and device details

Syntax

```
hdlcoder.supportedDevices
```

Description

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

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

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

Examples

Set the target device for your model

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

Show the supported target device reports.

```
hdlcoder.supportedDevices
```

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

Click the `Xilinx ISE Device List` link to open the supported target device report and view details for your target device.

Open the model, `sfir_fixed`.

```
sfir_fixed
```

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

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

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

```
hdldispmdlparams
```

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

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

Version History
Introduced in R2014a

See Also

Topics
"Synthesis Tool Path Setup"
"Tool and Device Parameters"

hdldispblkparams

Display HDL block parameters with nondefault values

Syntax

```
hdldispblkparams(path)
hdldispblkparams(path,"all")
```

Description

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

hdldispblkparams(path,"all") displays, for the specified block or subsystem, the names and values of all HDL block parameters.

Examples

HDL Block Parameters and Values

This example displays all of the HDL block parameters and values for the subsystem symmetric_fir in the current model, sfir_fixed.

```
open_system("sfir_fixed")
hdldispblkparams("sfir_fixed/symmetric_fir","all")


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
HDL Block Parameters ('sfir_fixed/symmetric_fir')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Implementation

    Architecture  : Module

Implementation Parameters

    AdaptivePipelining : inherit
    AXI4RegisterReadback : off
    AXI4SlaveIDWidth :
    AXI4SlavePortToPipelineRegisterRatio : auto
    BalanceDelays : inherit
    ClockRatePipelining : inherit
    ConstrainedOutputPipeline : 0
    DistributedPipelining : off
    DSPStyle : none
    FlattenHierarchy : inherit
    GenerateDefaultAXI4Slave : on
    InputPipeline : 0
    IPCoreAdditionalFiles :
    IPCoreName :
    IPCoreVersion :
    IPDataCaptureBufferSize : 128
    IPDataCaptureSequenceDepth : 1
    OutputPipeline : 0
    ProcessorFPGASynchronization :
```

```
SharingFactor : 0
StreamingFactor : 0
```

Input Arguments

`path` — Path to block or subsystem in current model
string scalar | character vector

Path to a block or subsystem in the current model.

`"all"` — Display all HDL block properties
`"all"` | `'all'`

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

Version History
Introduced in R2010b

See Also
"Set and View HDL Model and Block Parameters"

hdldispmdlparams

Display HDL model parameters with nondefault values

Syntax

```
hdldispmdlparams(model)
hdldispmdlparams(model,"all")
```

Description

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

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

Examples

Nondefault HDL Properties of the Model

This example displays HDL properties of the current model, `sfir_fixed`, that have nondefault values.

```
open_system("sfir_fixed")
hdldispmdlparams(bdroot)
```

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

HDLSubsystem             : 'sfir_fixed/symmetric_fir'
```

All HDL Properties of the Model

This example displays all of the HDL properties and values of the current model, `sfir_fixed` .

```
open_system("sfir_fixed")
hdldispmdlparams(bdroot,"all")
```

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

AdaptivePipelining                      : 'off'
AdderSharingMinimumBitwidth             : 0
AutoPlace                               : 'on'
AutoRoute                               : 'on'
Backannotation                          : 'off'
.
.
.
VerilogFileExtension                    : '.v'
```

```
Workflow                                    : 'Generic ASIC/FPGA'
sschdlMatrixProductSumCustomLatency         : -1
```

Input Arguments

model — Model name
string scalar | character vector

Name of an open model.

"all" — Display all HDL model properties
"all" | 'all'

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

Version History
Introduced in R2010b

See Also
"View HDL Model Parameters"

hdlget_param

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

Syntax

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

Description

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

Examples

HDL Block Property Value

Open the `sfir_fixed` model. Set the `OutputPipeline` parameter for the DUT subsystem `symmetric_fir` to 3 by using the `hdlset_param` function. Return the value of the `OutputPipeline` parameter to the variable `p` by using the `hdlget_param` function.

```
open sfir_fixed
hdlset_param("sfir_fixed/symmetric_fir","OutputPipeline",3)
p = hdlget_param("sfir_fixed/symmetric_fir","OutputPipeline")

p =

    3
```

All HDL Block Properties

Return HDL block parameters and values for the product block `m1` in the `symmetric_fir` subsystem that is in the `sfir_fixed` model to the cell array `p`.

```
p = hdlget_param("sfir_fixed/symmetric_fir/m1","all")

p =

  1×18 cell array

  Columns 1 through 8

    {'Architecture'}  {'Linear'}  {'ConstrainedOutp…'}  {[0]}  {'DSPStyle'}  {'none'}  {'HandleDenormals'}  {'inherit'}

  Columns 9 through 16

    {'InputPipeline'}  {[0]}  {'LatencyStrategy'}  {'inherit'}  {'MantissaMultipl…'}  {'inherit'}  {'NFPCustomLatency'}  {[0]}

  Columns 17 through 18

    {'OutputPipeline'}  {[0]}
```

Input Arguments

block_path — Block or subsystem path
string scalar | character vector

Path to a block or subsystem in the current model.

Example: "modelname/subsysA/blockName"

prop — HDL block property
string scalar | character vector

A string scalar or character vector that designates one of the following:

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

Example: "OutputPipeline"

Output Arguments

p — Variable containing HDL block property value
none

`p` stores 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.

Tips

- Use `hdlget_param` to obtain the value of HDL block parameters. For a list of block implementation parameters, see "HDL Block Properties: General".
- Use `hdldispmdlparams` to see the values of HDL model parameters. To obtain the value of general model parameters, use the `get_param` function.

Version History
Introduced in R2010b

See Also
hdlset_param | hdlsaveparams | hdlrestoreparams

hdllib

Display blocks that are compatible with HDL code generation

Syntax

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

Description

`hdllib` displays 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".
```
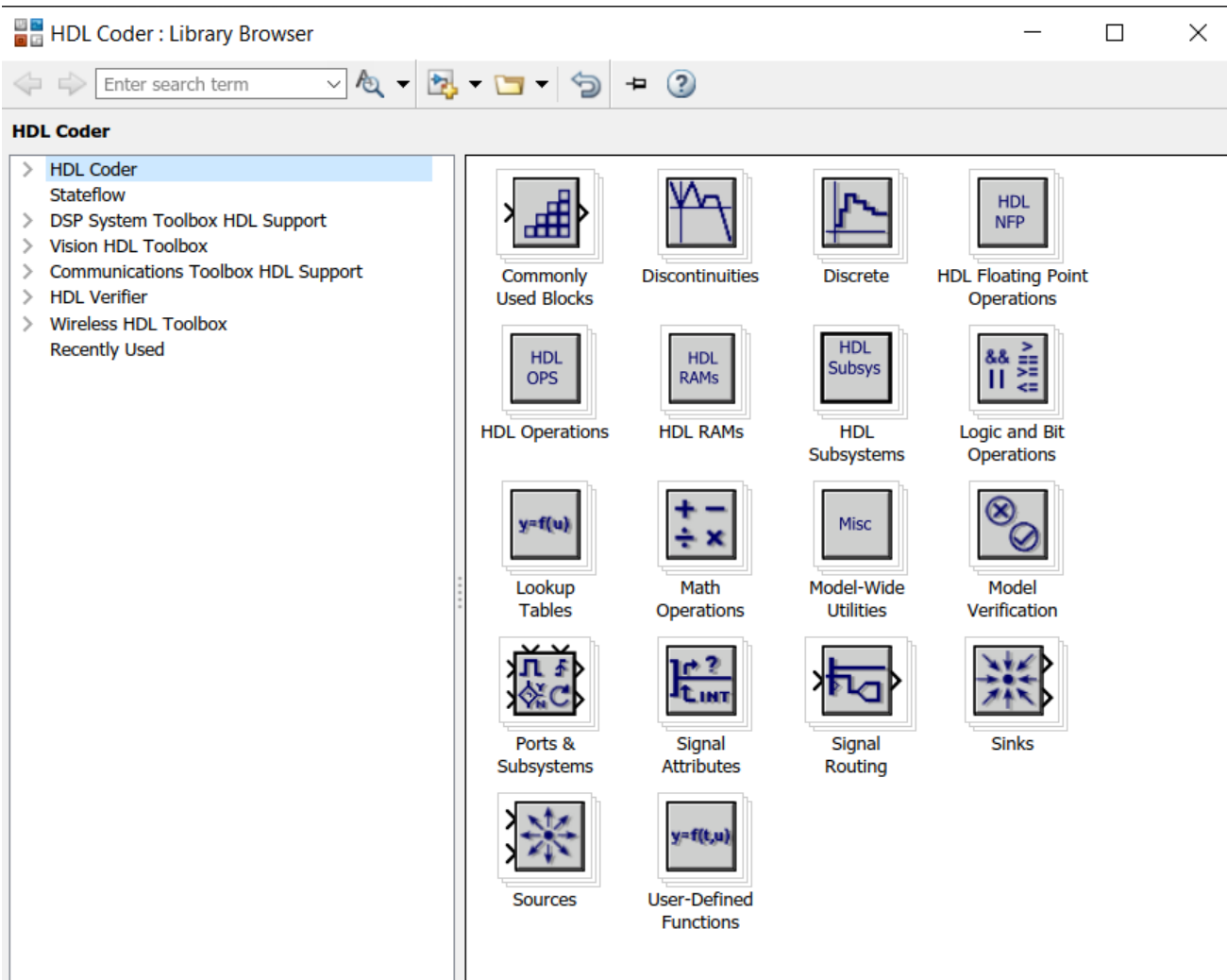
Display All Blocks in the Library Browser

To restore the Library Browser to the default view, in the Library Browser, click the button. Alternatively, at the command line, enter:

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

```
### Restoring Library Browser to default view; removing the HDL Coder compatibility filter.
```
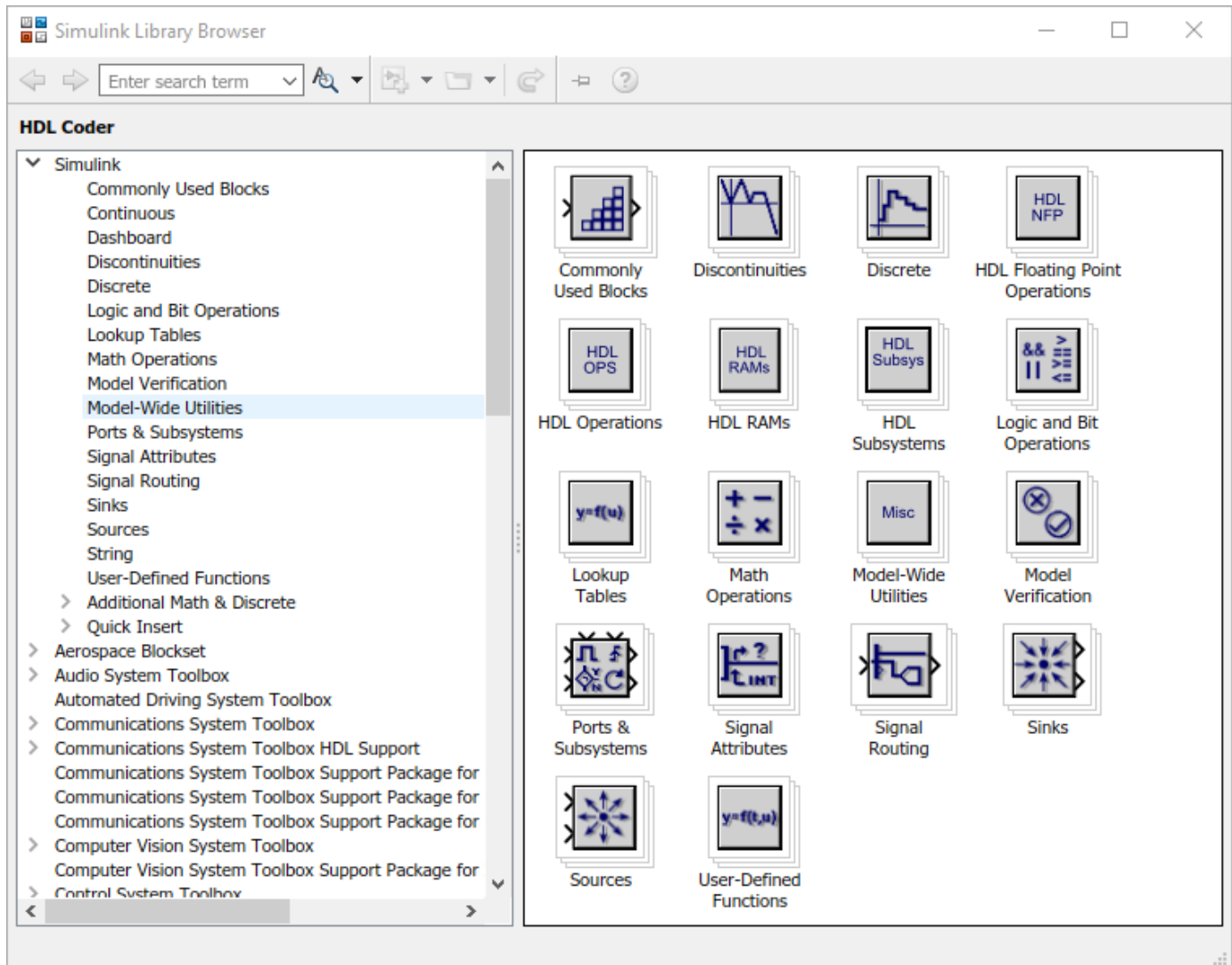
Create a Supported Blocks Library and HTML Reports

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

```
hdllib('html')
```

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

The `hdlsupported` library opens. To view the reports, click the `hdlblklist.html` and `hdlsupported.html` links.
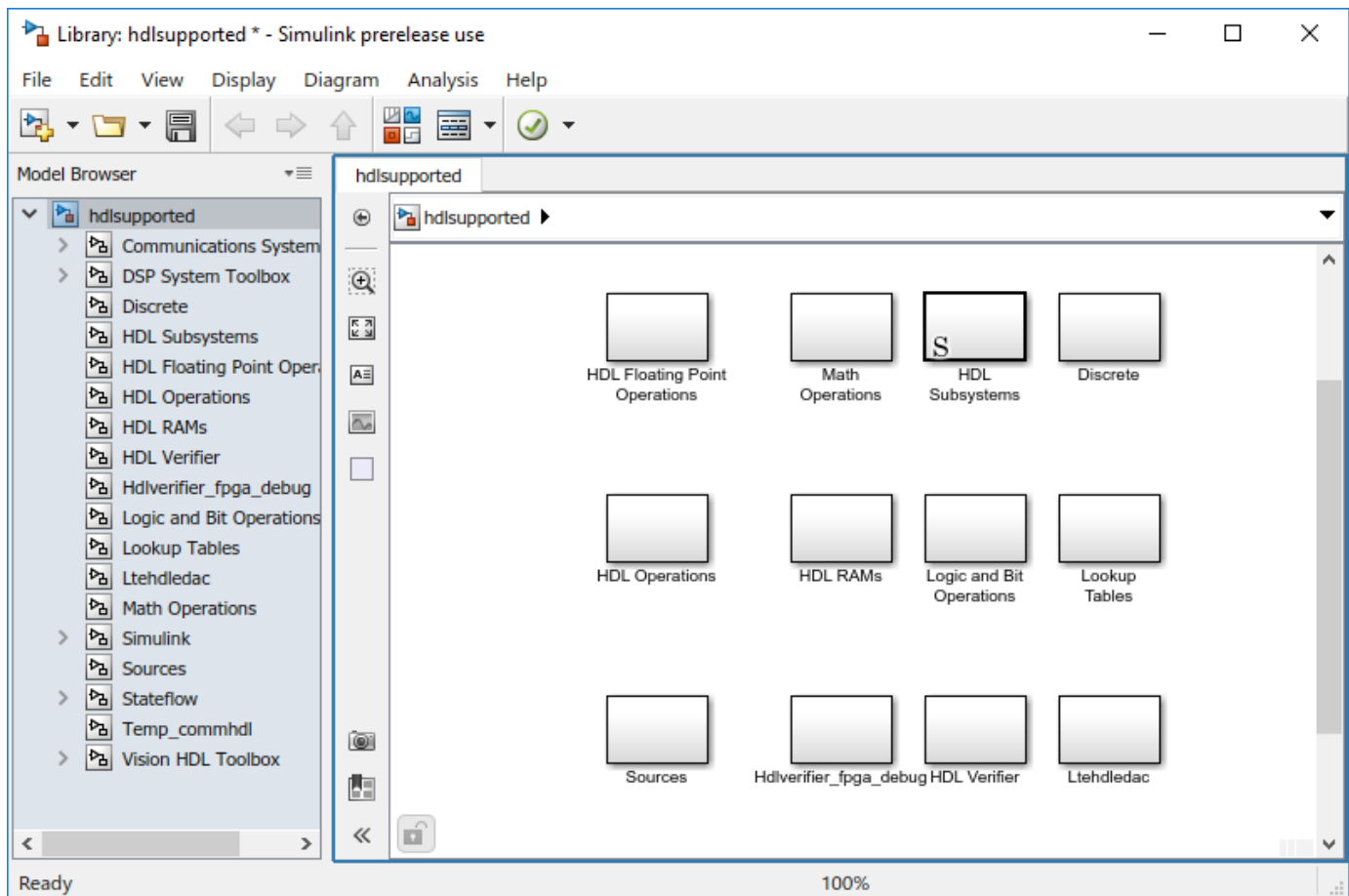
Create a Supported Blocks Library

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

hdllib('librarymodel')

The hdlsupported block library opens.
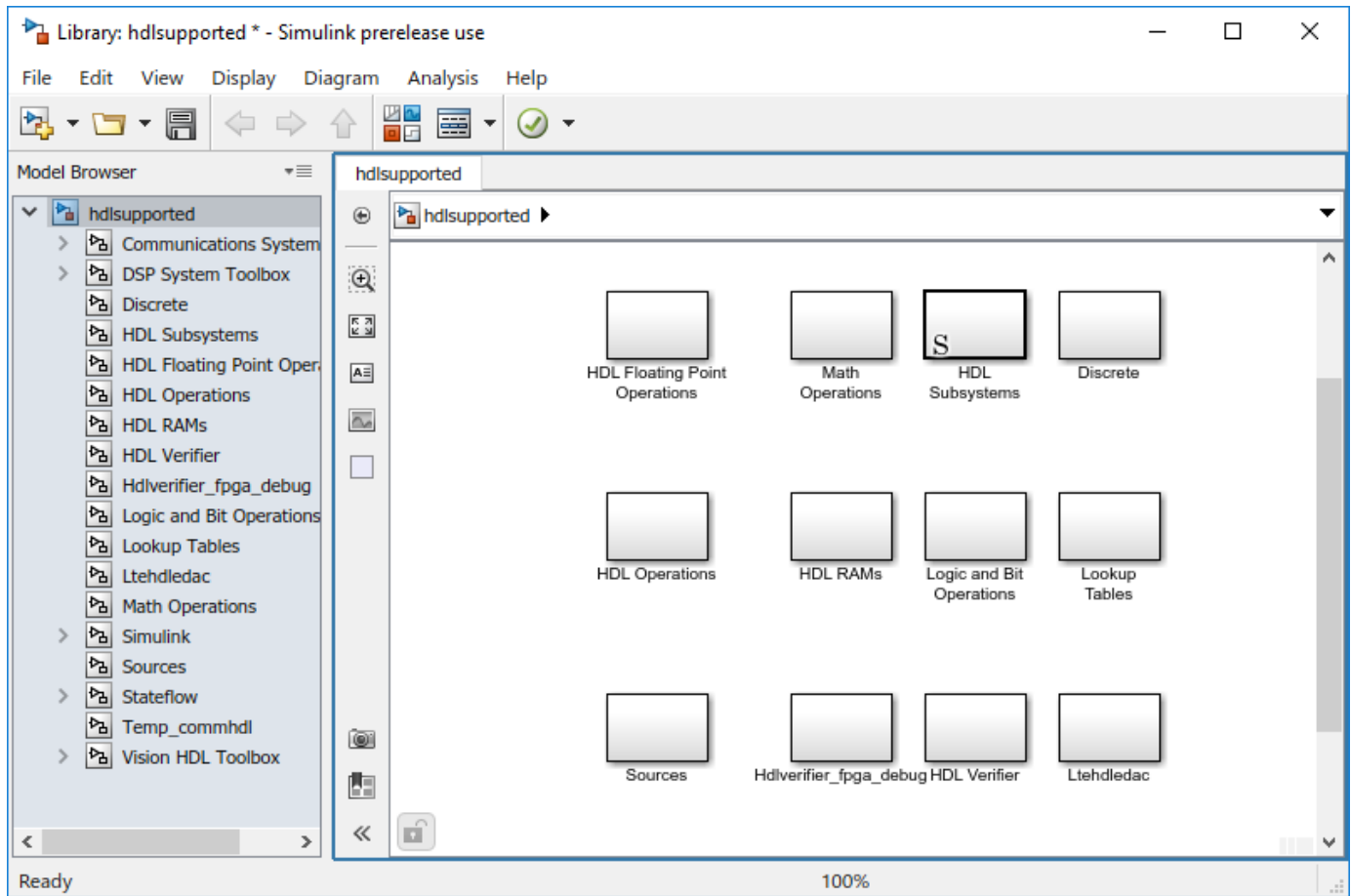
Version History
Introduced in R2006b

See Also

Topics
"Display Blocks for HDL Code Generation in Library Browser"
"View HDL-Supported Blocks and HDL-Specific Block Documentation"
"Create HDL-Compatible Simulink Model"

hdlcodeadvisor

Open HDL Code Advisor

Syntax

```
hdlcodeadvisor(subsystem)
hdlcodeadvisor(model)
```

Description

hdlcodeadvisor(subsystem) opens the HDL Code Advisor for the subsystem within the model.

hdlcodeadvisor(model) opens the HDL Code Advisor for the model.

Examples

Open the HDL Code Advisor For a Model

This example shows how to open the HDL Code Advisor for the `sfir_single` model.
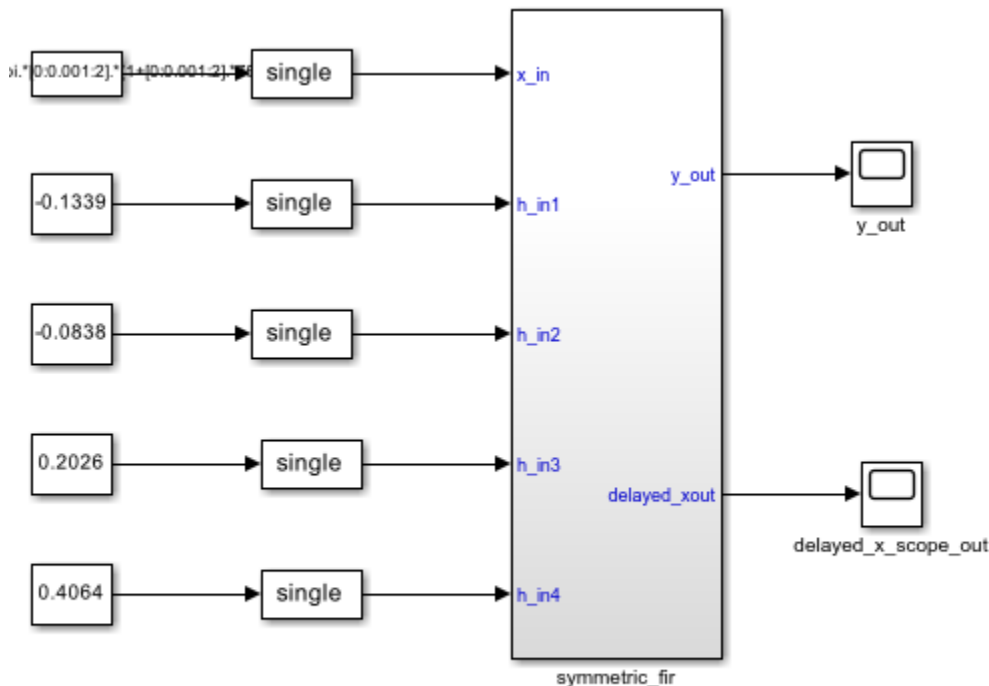
`sfir_single`

Copyright 2016-2017 The MathWorks, Inc.

To open the HDL Code Advisor for the sfir_single model, enter:

```
hdlcodeadvisor('sfir_single')
```
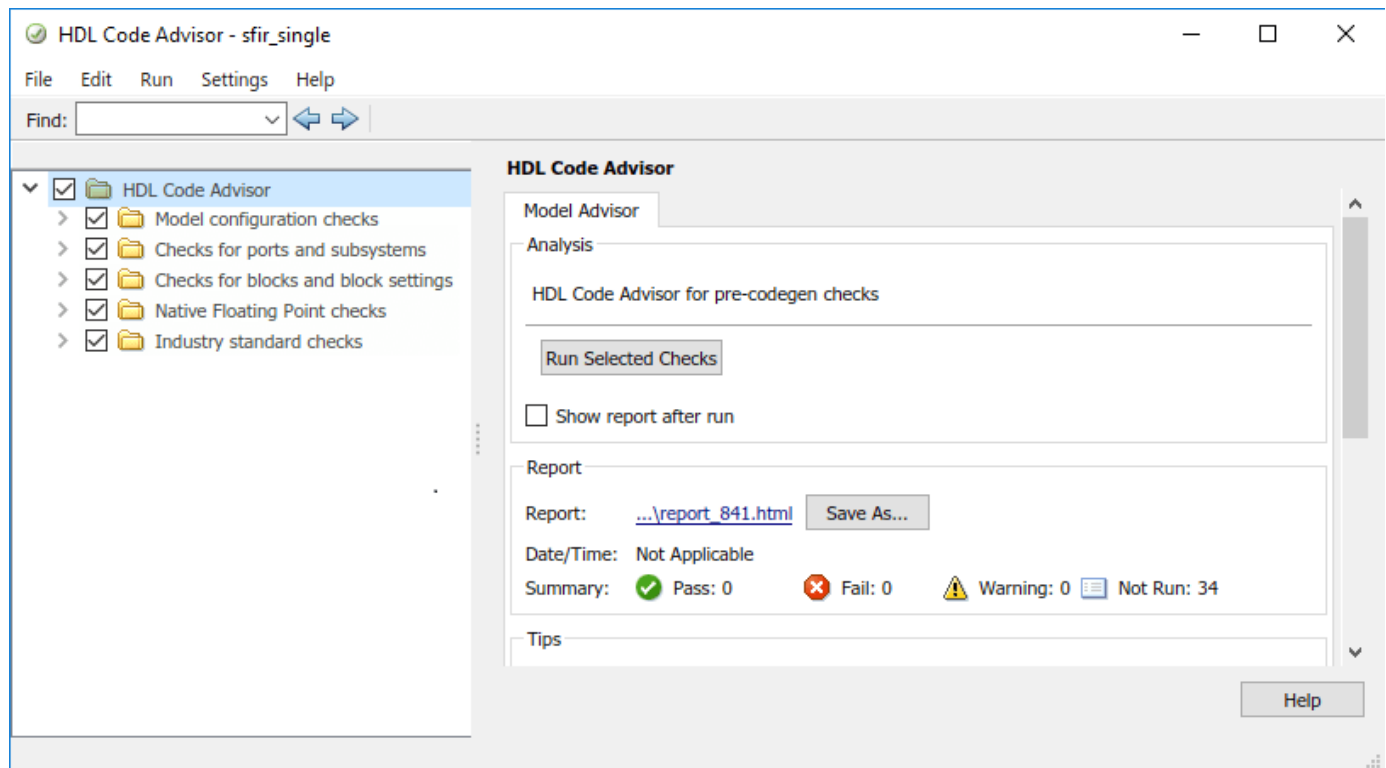

Open the HDL Model Checker For a Subsystem

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

```
sfir_single
```
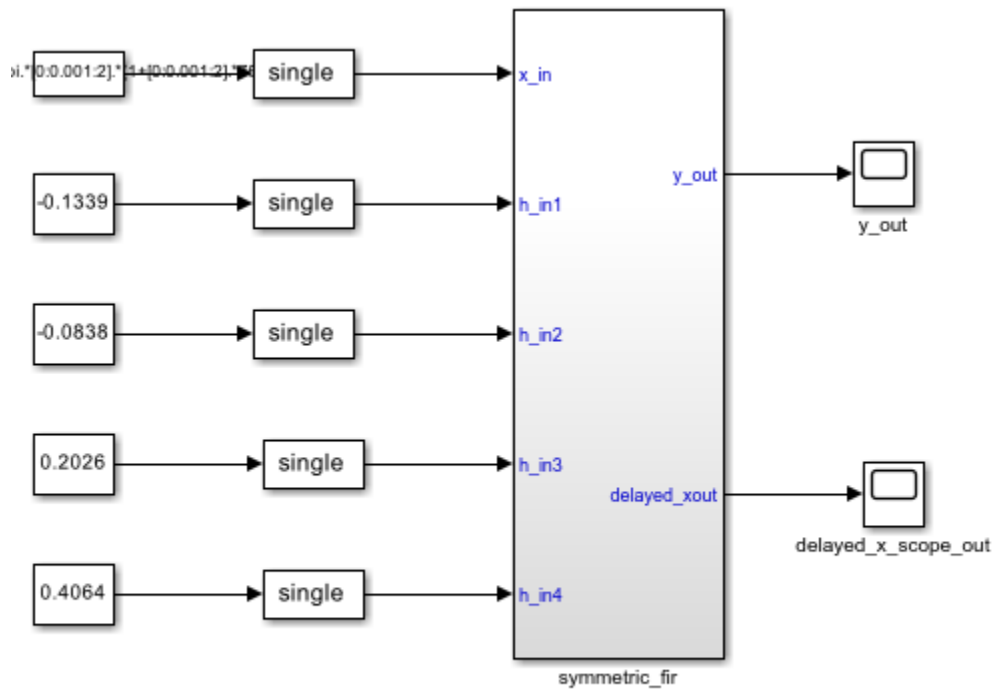
Copyright 2016-2017 The MathWorks, Inc.

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

```
hdlcodeadvisor('sfir_single/symmetric_fir')
```
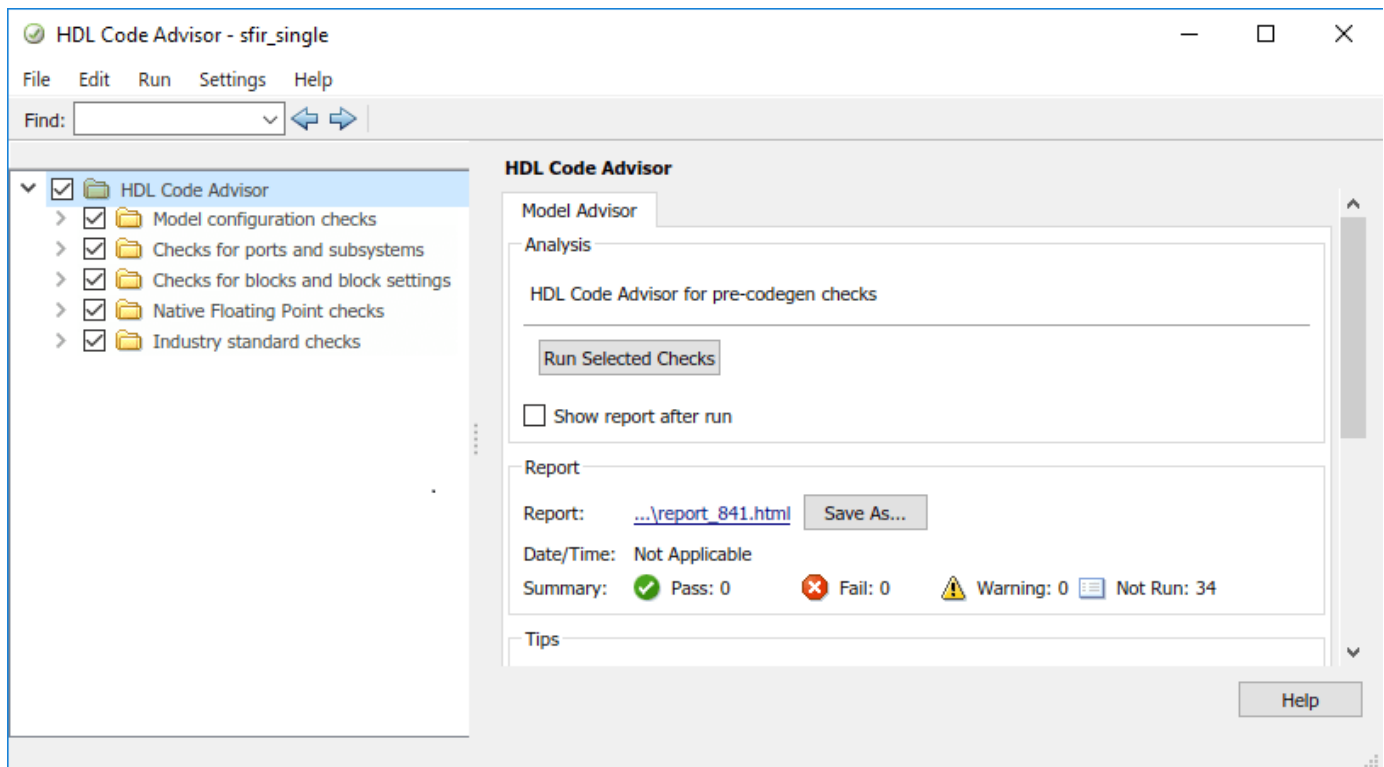
Input Arguments

subsystem — Subsystem name
character vector

Subsystem name or handle, specified as a character vector.

Data Types: char

model — Model name
character vector

Model name or handle, specified as a character vector.

Data Types: char

Version History
Introduced in R2017b

See Also

Topics
"Check HDL Compatibility of Simulink Model Using HDL Code Advisor"
"HDL Code Advisor Checks"

hdlrestoreparams

Restore block- and model-level HDL parameters to model

Syntax

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

Description

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

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

Examples

Save and Restore HDL-Related Model Parameters

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

Set Model HDL Parameters

Open the `sfir_fixed` model.

```
sfir_fixed
```

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

% Set SubSystem HDL parameters
```

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

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

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

Verify Saved Parameters

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

Input Arguments

`dut` — DUT subsystem name
character vector

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file
character vector

Name of file containing previously saved HDL model parameters.

Example: `'mymodel_saved_params.m'`

Version History
Introduced in R2012b

See Also
`hdlsaveparams`

hdlsaveparams

Save nondefault block- and model-level HDL parameters

Syntax

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

Description

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

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

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

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

Examples

Display HDL-Related Nondefault Model Parameters

Open the model.

```
sfir_fixed
```

Copyright 2007 The MathWorks, Inc.

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

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

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

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Save and Restore HDL-Related Model Parameters

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

Set Model HDL Parameters

Open the `sfir_fixed` model.

`sfir_fixed`

symmetric_fir

Launch HDL Coder App

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

`hdlsaveparams('sfir_fixed/symmetric_fir')`

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

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

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

Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

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

Verify Saved Parameters

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

Overwrite Previously Saved HDL Parameters File

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

Set Model HDL Parameters

Open the `sfir_fixed` model.

```
sfir_fixed
```

Copyright 2007 The MathWorks, Inc.

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

Set HDL-related model parameters for the symmetric_fir subsystem.

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

Save Model HDL Parameters

Verify that model parameters are set.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

% Set SubSystem HDL parameters
```

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

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

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

Verify Saved Parameters

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

Modify Saved HDL Parameters

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

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

Overwrite Saved Parameters File

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

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

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

Verify Resaved Parameters

Reset HDL-related model parameters to default values.

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

Verify that model parameters have default values.

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

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

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

Verify that the saved model parameters are restored

```
hdlsaveparams('sfir_fixed/symmetric_fir')

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

% Set SubSystem HDL parameters
hdlset_param('sfir_fixed/symmetric_fir', 'InputPipeline', 5);
hdlset_param('sfir_fixed/symmetric_fir', 'OutputPipeline', 2);
hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 4);
```

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

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

Open the model

```
sfir_single
sim('sfir_single')
```
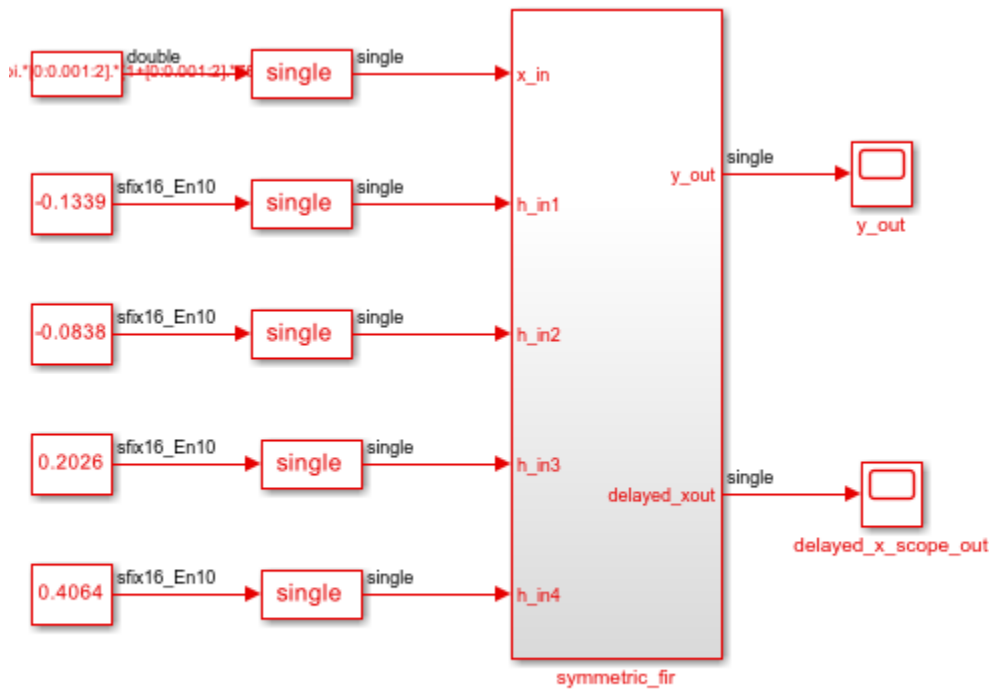
Copyright 2016-2017 The MathWorks, Inc.

Save HDL Model and Block parameters

```
hparams = hdlsaveparams('sfir_single/symmetric_fir');

%% Set Model 'sfir_single' HDL parameters
fpconfig = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT' ...
, 'MantissaMultiplyStrategy', 'FullMultiplier' ...
);
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
hdlset_param('sfir_single', 'HDLSubsystem', 'sfir_single/symmetric_fir');
hdlset_param('sfir_single', 'UseFloatingPoint', 'on');

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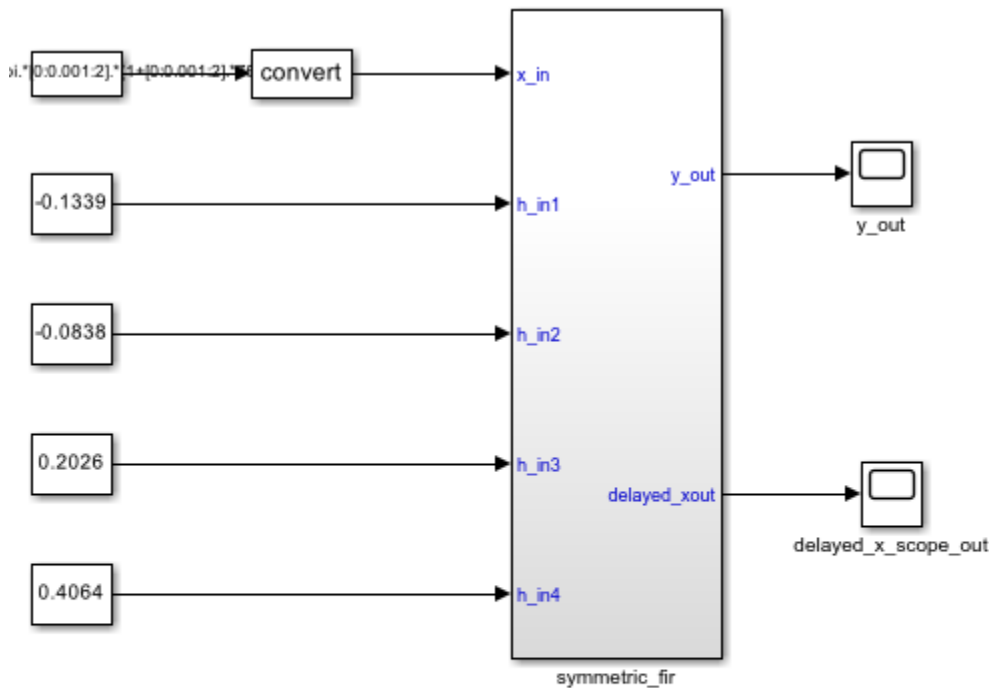```

### View and Access Block Parameters

```
hparams


hparams =

  1x5 struct array with fields:

    object
    parameter
```

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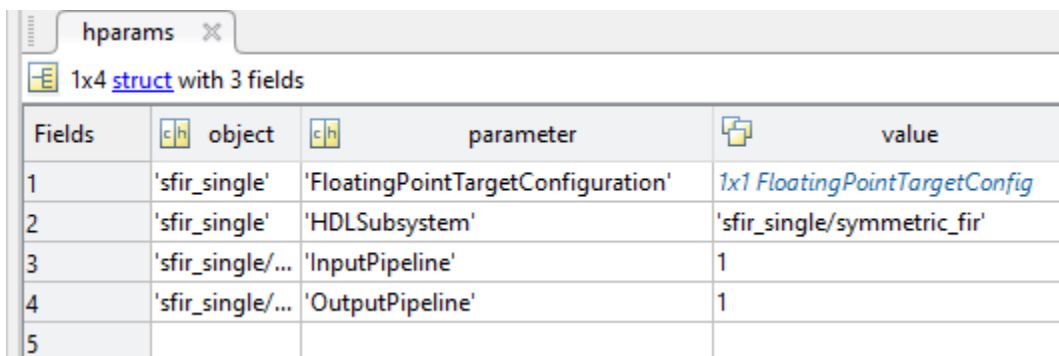
To see specific non-default parameters saved in the structure, you can access individual elements of the structure.

```
hparams(2)
```

```
ans =

  struct with fields:

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

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

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

hparams — 1x4 struct with 3 fields

Input Arguments

`dut` — DUT subsystem name
character vector

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

Example: `'modelname/subsysTarget'`

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

`filename` — Name of file
character vector

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

Example: `'mymodel_saved_params.m'`

`force_overwrite` — Overwrite parameters file
boolean

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

Example: `'true'`

Output Arguments

`varname` — Name of variable containing saved parameters
struct

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

Example: `'hparams'`

Version History
Introduced in R2012b

See Also
`hdlrestoreparams`

hdlset_param

Set HDL-related parameters at model or block level

Syntax

hdlset_param(path,Name,Value)

Description

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

Examples

HDL Block Property for Multiple Blocks

This example uses the sfir_fixed model to show 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;
```

Input Arguments

path — model, block, or subsystem path
string scalar | character vector

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

Example: "modelname/subsysA/blockName"

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: hdlset_param("sfir_fixed", TargetLanguage = "Verilog")

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: hdlset_param("sfir_fixed", "TargetLanguage", "Verilog")

Name — HDL property name
string scalar | character vector

Name is a string scalar or 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. For a list of block implementation parameters, see "HDL Block Properties: General".

Value — HDL property value
string scalar | character vector

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

Tips

- When you set multiple parameters on the same model or block, use a single `hdlset_param` command that has 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 parameters.

Version History
Introduced in R2010b

See Also
`hdlget_param` | `hdlsaveparams` | `hdlrestoreparams`

Topics
"Set and View HDL Model and Block Parameters"
"Set HDL Block Parameters for Multiple Blocks Programmatically"

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.

Open the model before you invoke the hdlsetup command. After using hdlsetup, you can use set_param to modify these default settings. For example:

set_param(gcs, "Solver", "VariableStepDiscrete")

See this table for the solver configuration parameters that hdlsetup configures. These parameters reside in the **Solver** pane of the Configuration Parameters dialog box.

Configuration Parameter Setting	Command-line Parameter Setting	Description
Set **Type** to Fixed-step and **Solver** to Discrete (no continuous states).	Set Solver to FixedStepDiscrete.	This fixed-step solver is best suited for simulating discrete systems. Variable-step solvers are supported under limited conditions.
Set **Fixed-step size (fundamental sample time)** to auto.	Set FixedStep to auto.	If **Fixed step size** is set to auto the step size is chosen automatically, based on the sample times specified in the model.
Disable the **Treat each discrete rate as a separate task** check box.	Set EnableMultiTasking to off.	HDL Coder does not currently support models that execute in multitasking mode..

hdlsetup also configures other parameters that control error severity levels, data logging, and model display options. To view the complete set of model parameters affected by hdlsetup, see "Check for model parameters suited for HDL code generation".

Version History
Introduced in R2006b

hdlcleanup

Clear all HDL code generation states

Syntax

```
hdlcleanup
```

Description

`hdlcleanup` clears all HDL code generation states preserved in the MATLAB session.

Examples

Clear Code Generation States

The following command generates HDL code for the design under test (DUT) and the clears all the code generation states preserved in the MATLAB session.

```
load_system('sfir_fixed');
open_system('sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed','ResourceReport','on');
makehdl('sfir_fixed/symmetric_fir');
hdlcleanup;
```

Version History
Introduced in R2021a

See Also
checkhdl | hdlsetup | makehdl | makehdltb

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. You cannot set up third-party FPGA synthesis tools in Simulink Online.

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". HDL Workflow Advisor is not available in Simulink Online.

Examples

Set Up Intel Quartus Prime Standard

The following command sets the synthesis tool path to point to an installed Intel® Quartus® Prime Standard Edition 20.1.1 executable file. You must have already installed Altera Quartus II.

```
hdlsetuptoolpath('ToolName','Altera Quartus II','ToolPath',...
 'C:\intel\20.1.1\quartus\bin\quartus.exe');
```

Set Up Intel Quartus Pro

The following command sets the synthesis tool path to point to an installed Intel Quartus Pro 21.3 executable file. You must have already installed Intel Quartus Pro.

```
hdlsetuptoolpath('ToolName','Intel Quartus Pro','ToolPath',...
 'C:\intel\21.3_pro\quartus\bin64\qpro.exe');
```

Note An installation of Quartus Pro contains both `quartus.exe` and `qpro.exe` executable files. When both tools are added to the path by using `hdlsetuptoolpath`, HDL Coder checks the tool availability before running the HDL Workflow Advisor.

Set Up Xilinx ISE

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

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

Set Up Xilinx Vivado

The following command sets the synthesis tool path to point to an installed Vivado® Design Suite 2020.2 batch file. You must have already installed Xilinx Vivado.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
 'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Set Up Microchip Libero SoC

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

```
hdlsetuptoolpath('ToolName','Microchip Libero SoC','ToolPath',...
 'C:\Microsemi\Libero_SoC_v12.0\Designer\bin\libero.exe');
```

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\2020.2\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.

Version History
Introduced in R2011a

See Also
`startup` | `setenv`

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

makehdl

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

Syntax

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

Description

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

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

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

Examples

Generate VHDL for the Current Model

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

Open the `sfir_fixed` model.
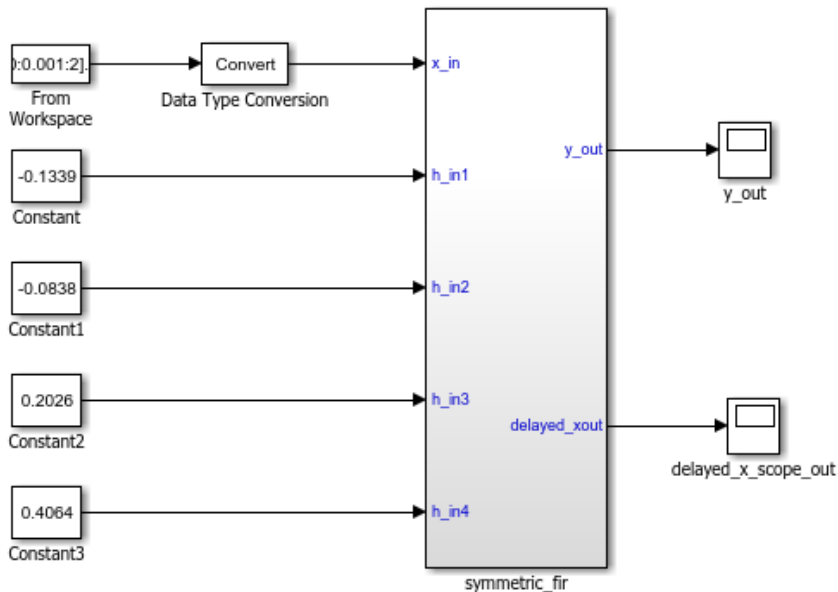
```
sfir_fixed
```

Copyright 2007 The MathWorks, Inc.

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

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

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\GenVHDL\hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Creating HDL Code Generation Check Report file://C:\GenVHDL\hdlsrc\sfir_fixed\symmetric_fir_r
### 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.
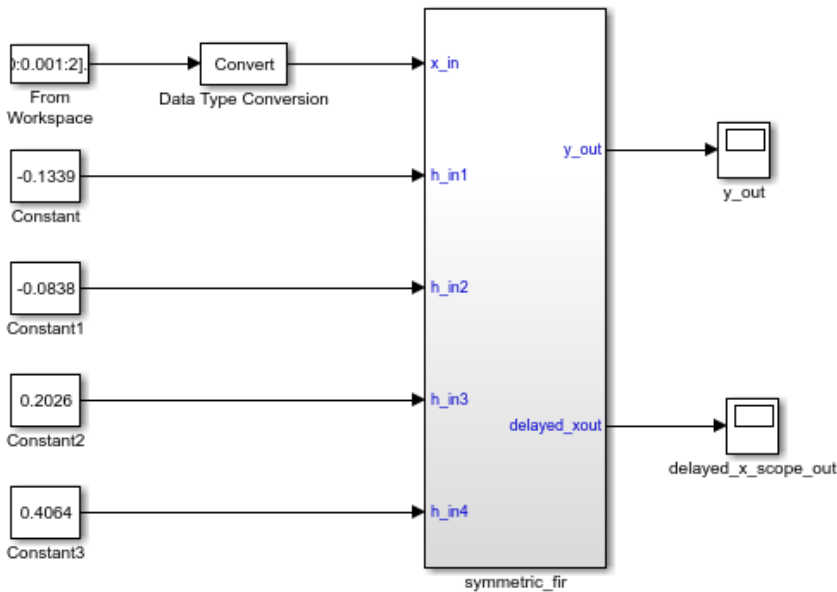
```
sfir_fixed;
```

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\symmetric_fir.v
### Creating HDL Code Generation Check Report file://C:\Generate_Verilog\hdlsrc\sfir_fixed\symme
### 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.
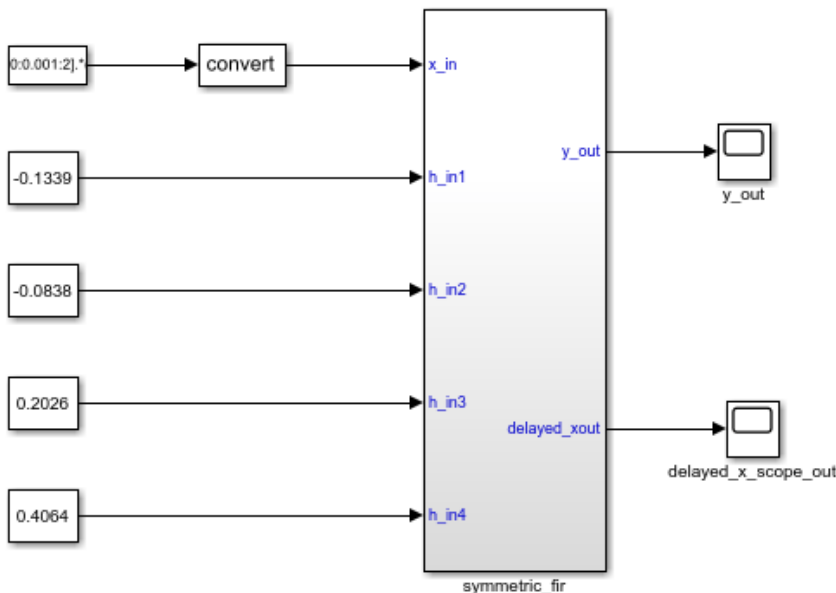
`sfir_fixed`

Copyright 2007 The MathWorks, Inc.

The model opens in a new Simulink® window.

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

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

```
### Starting HDL check.
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\sfir_fixed\symmetric_f:
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
```

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

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

```
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
```

```
### Starting HDL check.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as C:\HDL_Checks\hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\sfir_fixed\symmetric_f
### 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TargetLanguage','Verilog'`

Target Language and Folder Selection Options

HDLSubsystem — DUT Subsystem
character vector

Specify the Subsystem in your model to generate HDL code for. For more information, see "Generate HDL for".

TargetLanguage — Target language
`'VHDL'` (default) | `'Verilog'`

Specify whether to generate VHDL or Verilog code. For more information, see "Language".

TargetDirectory — Output directory
`'hdlsrc'` (default) | character vector

Specify a path to write the generated files and HDL code into. For more information, see "Folder".

Tool and Synthesis Options

`SynthesisTool` — Synthesis tool
`''` (default) | `'Altera Quartus II'` | `'Xilinx ISE'` | `'Xilinx Vivado'` | `'Intel Quartus Pro'` | `'Microchip Libero SoC'`

Specify the synthesis tool for targeting the generated HDL code as a character vector. For more information, see "Synthesis Tool".

`SynthesisToolChipFamily` — Synthesis tool chip family
`''` (default) | character vector

Specify the synthesis tool chip family for the target device as a character vector. For more information, see "Family".

`SynthesisToolDeviceName` — Synthesis tool device name
`''` (default) | character vector

Specify the synthesis tool device name for the target device as a character vector. For more information, see "Device".

`SynthesisToolPackageName` — Synthesis tool package name
`''` (default) | character vector

Specify the synthesis tool package name for the target device as a character vector. For more information, see "Package".

`SynthesisToolSpeedValue` — Synthesis tool speed value
`''` (default) | character vector

Specify the synthesis tool speed value for the target device as a character vector. For more information, see "Speed".

`TargetFrequency` — Target frequency in MHz
`''` (default) | character vector

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

General Optimizations and Multicycle Path Constraints Settings

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

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

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

Specify the minimum RAM size required for mapping to RAMs instead of registers. You can specify either:

- A single integer to define the mapping threshold that maps any delay or persistent array greater than that threshold bit size to RAM.
- A string that define two thresholds, one for delay length or array size and one for word length.

For more information, see "RAM mapping threshold".

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

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

TransformNonZeroInitValDelay — Transform delays with nonzero initial value
'on' (default) | 'off'

Specify whether to transform Delay blocks that have nonzero initial value to Delay blocks that have a zero initial value. For more information, see "Transform non zero initial value delay".

MultiplierPartitioningThreshold — Partition multipliers based on a threshold
'Inf' (default) | positive integer

Partition multipliers in the design based on a threshold value. The threshold must be a positive integer value, N. For more information, see "Multiplier partitioning threshold".

DeleteUnusedPorts — Removes ports in your design that are unused from the generated HDL code
'on' (default) | 'off'

For more information, see "Remove Unused Ports".

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

Generate a multicycle path constraints text file. For more information, see "Register-to-register path info".

MulticyclePathConstraints — Enable-based multicycle path constraint file generation
'off' (default) | 'on'

Generate an enable-based multicycle path constraints file. For more information, see "Enable-based constraints".

Pipelining and Speed Optimization Options

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

Specify whether to prioritize the distributed pipelining optimization for numerical integrity or performance. For more information, see "Distributed pipelining priority".

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

Distribute design delays in your model. For more information, see "Preserve design delays".

UseSynthesisEstimatesForDistributedPipelining — Use synthesis estimates for distributed pipelining
'off' (default) | 'on'

Use synthesis timing estimates for distributed pipelining to more accurately reflect how components function on hardware to better distribute pipelines and increase clock speed for your target device. For more information, see "Use synthesis estimates for distributed pipelining".

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

Insert pipeline registers at the clock rate instead of the data rate for multi-cycle paths. For more information, see "Clock-rate pipelining".

ClockRatePipelineOutputPorts — Clock-rate pipelining for DUT ports
'on' (default) | 'off'

Enable clock-rate pipelining for DUT ports. For more information, see "Allow clock-rate pipelining of DUT output ports".

AdaptivePipelining — Insert adaptive pipelines
'off' (default) | 'on'

Insert adaptive pipeline registers in your design. For more information, see "Adaptive pipelining".

Resource Sharing and Area Optimization Options

ShareAdders — Share adders in the design
'off' (default) | 'on'

Use resource sharing optimization to share adders in your design. For more information, see "Share Adders".

AdderSharingMinimumBitwidth — Minimum bitwidth of shared adder for resource sharing
0 (default) | positive integer

Minimum bitwidth of a shared adder for the resource sharing optimization, specified as a positive integer. For more information, see "Adder sharing minimum bitwidth".

ShareMultipliers — Share multipliers in the design
'on' (default) | 'on'

Use resource sharing optimization to share multipliers in your design. For more information, see "Share Multipliers".

MultiplierSharingMinimumBitwidth — Minimum bitwidth of shared multiplier for resource sharing
0 (default) | positive integer

Minimum bitwidth of a shared multiplier for the resource sharing optimization, specified as a positive integer. For more information, see "Multiplier sharing minimum bitwidth".

MultiplierPromotionThreshold — Minimum promotion wordlength
0 (default) | positive integer

Minimum wordlength by which the code generator promotes a multiplier for sharing with other multipliers. For more information, see "Multiplier promotion threshold".

ShareMultiplyAdds — Share Multiply-Add blocks in the design
'on' (default) | 'on'

Use resource sharing optimization to share Multiply-Add blocks in your design. For more information, see "Share Multipliers".

MultiplyAddSharingMinimumBitwidth — Minimum bitwidth of shared Multiply-Add block for resource sharing
0 (default) | positive integer

Minimum bitwidth of a shared Multiply-Add block for the resource sharing optimization, specified as a positive integer. For more information, see "Multiply-Add block sharing minimum bitwidth".

ShareAtomicSubsystems — Share atomic subsystems in the design
'on' (default) | 'on'

Use resource sharing optimization to share Atomic Subsystem blocks in your design. For more information, see "Share Atomic subsystems".

ShareMATLABBlocks — Share MATLAB Function blocks in the design
'on' (default) | 'on'

Use resource sharing optimization to share MATLAB Function blocks in your design. For more information, see "Share MATLAB Function blocks".

ShareFloatingPointIPs — Share floating-point IPs in the design
'on' (default) | 'on'

Use resource sharing optimization to share floating-point IPs in your design. For more information, see "Share Floating-Point IPs".

Floating Point Target

FloatingPointTargetConfiguration — Floating point target configuration
'' (default) | character vector

For more information, see "Floating Point IP Library Parameters".

Code Generation Report Options

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

Generate a traceability report that has hyperlinks for navigating from code-to-model and from model-to-code. For more information, see "Generate traceability report".

TraceabilityStyle — Line-level or comment-based traceability style
'LineLevel' (default) | 'CommentBased'

Generate a traceability report that has hyperlinks from each line or to a comment indicating block of code for navigating from code-to-model and from model-to-code. For more information, see "Traceability style".

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

Generate a resource utilization report that displays the number of hardware resources that the generated HDL code uses. For more information, see "Generate resource utilization report".

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

Generate an optimization report that displays the effect of optimizations such as streaming, sharing, and distributed pipelining. For more information, see "Generate optimization report".

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

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

Clock Settings

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

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

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

Specify whether to use an active-high or active-low asserted level for the reset input signal. For more information, see "Reset asserted level".

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

Specify the clock input port name as a character vector. For more information, see "Clock input port".

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

Specify the clock enable input port name as a character vector. For more information, see "Clock enable input port".

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

Reset input port name, specified as a character vector.

For more information, see "Reset input port".

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

Specify the active clock edge for the generated HDL code. For more information, see "Clock edge"

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

Specify whether to generate single or multiple clock inputs in the HDL code. For more information, see "Clock inputs".

Oversampling — Oversampling factor for global clock
1 (default) | integer greater than or equal to 0

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

General File and Variable Name Options

UserComment — HDL file header comment
character vector

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

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

Specify the file name extension for generated Verilog files. For more information, see "Verilog file extension".

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

Specify the file name extension for generated VHDL files. For more information, see "VHDL file extension".

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

Specify the postfix as a character vector that resolves duplicate entity or module names. For more information, see "Entity conflict postfix".

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

Specify the postfix for the package file name as a character vector. For more information, see "Package postfix".

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

For more information, see "Reserved word postfix".

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

For more information, see "Split entity and architecture".

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

For more information, see "Split entity file postfix".

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

For more information, see "Split arch file postfix".

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

For more information, see "VHDL architecture name".

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

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

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

For more information, see **Complex imaginary part postfix** in "Complex Signals Postfix Parameters".

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

For more information, see **Complex real part postfix** in "Complex Signals Postfix Parameters".

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

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

ModulePrefix — Prefix for modules or entity names
'' (default) | character vector

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

For more information, see **ModulePrefix** in "Language-Specific Identifiers and Postfix Parameters".

TimingControllerPostfix — Postfix for timing controller name
'_tc' (default) | character vector

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

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

For more information, see "Pipeline postfix".

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

For more information, see "VHDL library name".

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

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

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

For more information, see "Block generate label".

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

For more information, see "Output generate label".

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

For more information, see "Instance generate label".

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

For more information, see "Instance postfix".

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

For more information, see "Instance prefix".

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

For more information, see "Vector prefix".

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

For more information, see "Map file postfix".

Configuration Options for Ports

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

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

For more information, see "Input and Output Port and Clock Enable Output Type Parameters".

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 "Input and Output Port and Clock Enable Output Type Parameters".

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

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

For more information, see "Clock Enable output port".

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

For more information, see "Minimize Clock Enables and Reset Signal Parameters".

MinimizeGlobalResets — Omit global reset logic for single-rate designs
'off' (default) | 'on'

For more information, see "Minimize Clock Enables and Reset Signal Parameters".

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

For more information, see "Use trigger signal as clock".

EnableTestPoints — Enable HDL DUT port generation for test points
'off' (default) | 'on'

For more information, see "Enable HDL DUT port generation for test points".

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

For more information, see "Scalarize ports".

Coding Style

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

For more information, see "Represent constant values by aggregates".

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

For more information, see "Inline MATLAB Function block code".

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

For more information, see "Initialize all RAM blocks".

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

For more information, see "RAM Architecture".

NoResetInitializationMode — Initialize no-reset registers
'InsideModule' (default) | 'None' | 'Script'

For more information, see "No-reset registers initialization".

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

For more information, see "Minimize Intermediate Signals".

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

For more information, see "Unroll For-Generate Loops in VHDL Code".

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

For more information, see "Generate Parameterized HDL Code from Masked Subsystem".

EnumEncodingScheme — Unroll VHDL FOR and GENERATE loops
'default' (default) | 'onehot' | 'twohot' | 'binary'

For more information, see "Enumerated Type Encoding Scheme".

ScalarizedPortIndexing — Starting index for the names of scalarized vector ports
'Zero-based' (default) | 'One-based'

For more information, see "Indexing for Scalarized Port Naming".

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

For more information, see "Use "rising_edge/falling_edge" Style for Registers".

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

For more information, see "Inline VHDL configuration".

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

For more information, see "Concatenate type safe zeros".

ObfuscateGeneratedHDLCode — Obfuscate generated HDL code
'off' (default) | 'on'

Specify whether you want to obfuscate the generated HDL code. For more information, see "Generate obfuscated HDL code".

GenerateRecordType — Generate record types for bus
'off' (default) | 'on'

Specify whether you want to generate code with VHDL construct record types for bus signals at design under test (DUT) interface and different subsystem-level interfaces. For more information, see "Generate Record Types for Bus".

OptimizeTimingController — Optimize timing controller
'on' (default) | 'off'

For more information, see "Optimize timing controller".

TimingControllerArch — Specify the architecture of the generated timing controller
'default' (default) | 'resettable' | 'external'

For more information, see "Timing controller architecture".

CustomFileHeaderComment — Custom file header comment
'' (default) | character vector

For more information, see "Custom File Header Comment".

CustomFileFooterComment — Custom file footer comment
'' (default) | character vector

For more information, see "Custom File Footer Comment".

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

For more information, see **Emit time/date stamp in header** in "RTL Annotation Parameters".

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

For more information, see "Include Requirements in Block Comments".

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

For more information, see "Use Verilog `timescale directives".

Timescale — Use verilog 'timescale specification
'timescale 1ns/1ns' (default) | character vector

For more information, see "Verilog timescale specification".

Coding Standards

HDLCodingStandard — Specify HDL coding standard
character vector

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

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

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

Model Generation Parameters

`GeneratedModel` — Output generated model with HDL code
`'on'` (default) | `'off'`

For more information, see "Generated model".

`GenerateValidationModel` — Output validation model with generated model
`'off'` (default) | `'on'`

For more information, see "Validation model".

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

For more information, see "Prefix for generated model name".

`ValidationModelNameSuffix` — Suffix for generated validation model name
`'_vnl'` (default) | character vector

For more information, see "Suffix for validation model name".

`LayoutStyle` — Select the layout style of the generated HDL model for better layout visualization
`'Default'` (default) | `'None'` | `'AutoArrange'`

For more information, see "Layout Style".

`AutoRoute` — Automatic signal routing in generated model
`'on'` (default) | `'off'`

For more information, see "Auto signal routing".

`InterBlkHorzScale` — Inter-block horizontal scaling
`1.7` (default) | positive integer

For more information, see "Inter-block horizontal scaling".

`InterBlkVertScale` — Inter-block vertical scaling
`1.2` (default) | positive integer

For more information, see "Inter-block vertical scaling".

Diagnostics and Code Generation Output Parameters

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

Specify whether to highlight feedback loops in your design. For more information, see "Highlight feedback loops inhibiting delay balancing and optimizations".

HighlightClockRatePipeliningDiagnostic — Highlight blocks inhibiting clock-rate pipelining
'on' (default) | 'off'

Specify whether to highlight barriers for clock-rate pipeling optimization. For more information, see "Highlight blocks inhibiting clock-rate pipelining".

DistributedPipeliningBarriers — Highlight blocks inhibiting distributed pipelining
'on' (default) | 'off'

For more information, see "Highlight blocks inhibiting distributed pipelining".

DetectBlackBoxNameCollision — Check for name conflicts in black box interfaces
'warning' (default) | 'none' | 'error'

For more information, see "Check for name conflicts in black box interfaces".

TreatRealsInGeneratedCodeAs — Automatic block placement in generated model
'error' (default) | 'warning' | 'none'

For more information, see "Check for presence of reals in generated HDL code".

CodeGenerationOutput — Generation of HDL code and display of generated model
'GenerateHDLCode' (default) | 'GenerateHDLCodeAndDisplayGeneratedModel' | 'DisplayGeneratedModelOnly'

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

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

Generate HDL code for the model. For more information, see the **Generate HDL code** section in "Code Generation Output Parameter".

Script Generation

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

For more information, see "Generate EDA scripts".

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

For more information, see "Compile initialization".

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

For more information, see "Compile termination".

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

For more information, see "Compile file postfix".

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 "Compile command for Verilog".

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 "Compile command for VHDL".

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

For more information, see "Choose HDL lint tool".

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 "Lint initialization".

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 "Lint command".

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 "Lint termination".

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

For more information, see "Choose synthesis tool".

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 "Synthesis command".

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 "Synthesis file postfix".

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 "Synthesis initialization".

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 "Synthesis termination".

Version History
Introduced in R2006b

See Also
`makehdltb` | `checkhdl`

makehdltb

Generate HDL test bench from model or subsystem

Syntax

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

Description

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

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

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 makehdltb calls during the same MATLAB session.

- Test bench code and model code must both be generated in the same target language. If the target language specified for makehdltb differs from the target language specified for the previous makehdl execution, makehdltb regenerates model code in the same language specified for the test bench.

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\toolbox\hdlverifier\dpigenerator\rt
### 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 hdlsrc\sfir_fixed\symmetric_fir_dpi_t

### 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TargetLanguage','Verilog'`

Target Language and Folder Selection Options

`HDLSubsystem` — DUT Subsystem
character vector

Specify the Subsystem in your model to generate the test bench for. For more information, see "Generate HDL for".

TargetLanguage — Target language
'VHDL' (default) | 'Verilog'

Specify whether to generate VHDL or Verilog code. For more information, see "Language".

TargetDirectory — Output directory
'hdlsrc' (default) | character vector

Specify a path to write the generated files and HDL code into. For more information, see "Folder".

Test Bench Generation Output Options

GenerateHDLTestBench — Generate HDL test bench
'on' (default) | 'off'

The coder generates an HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT. For more information, see "HDL test bench".

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 2-105.

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.

See also "SystemVerilog DPI 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 "Cosimulation model".

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

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

For more information, see "Cosimulation model".

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.

When the GenerateCoSimModel argument is set to Xilinx Vivado Simulator, code coverage is not supported, and this input is ignored.

For more information, see "HDL code coverage".

SimulationTool — HDL simulator where you will run the generated test bench
'Mentor Graphics ModelSim' (default) | 'Cadence Incisive' | 'Xilinx Vivado Simulator' | 'Custom'

This property applies to generated test benches. When you select 'Custom', the tool uses the custom script settings. See the "Script Generation" properties.

For more information, see "Simulation tool".

Clock and Reset Input Options

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 "Force clock".

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

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

For more information, see "Clock high time (ns)".

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

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

For more information, see "Clock low time (ns)".

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

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

For more information, see "Force clock enable".

TestBenchClockEnableDelay — Clock cycles between reset and clock enable
1 (default) | positive integer

Number of clock cycles between deassertion of reset and assertion of clock enable, specified as a positive integer.

For more information, see "Clock enable delay (in clock cycles)"

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

Specify that the generated test bench drives the reset input.

For more information, see "Force reset".

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 "Reset length (in clock cycles)".

Testbench Stimulus and Response Parameters

HoldInputDataBetweenSamples — Hold valid data for signals
'on' (default) | 'off'

Hold valid data between samples for signals clocked at slower rate.

For more information, see "Hold input data between samples".

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 "Hold time (ns)".

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 "Ignore output data checking (number of samples)".

InitializeTestBenchInputs — Initialize test bench inputs
'off' (default) | 'on'

Initialize test bench inputs to zero. For more information, see "Initialize test bench inputs".

Testbench Configuration Parameters

TestBenchDataPostFix — Postfix for test bench data file name
'_data' (default) | character vector

Postfix for test bench data file name, specified as a character vector.

For more information, see "Test bench data file name postfix".

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

Postfix for test bench name, specified as a character vector.

For more information, see "Test bench name postfix".

TestBenchReferencePostFix — Postfix for test bench reference signal
'_ref' (default) | character vector

Postfix for test bench reference signal name, specified as a character vector.

For more information, see "Test bench reference postfix".

MultifileTestBench — Generate multiple testbench files
'off' (default) | 'on'

Divide generated test bench into helper functions, data, and HDL test bench files.

For more information, see "Multi-file test bench".

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

For more information, see "Use file I/O to read/write test bench data".

Floating Point Tolerance Options

FPToleranceStrategy — Floating-point tolerance strategy
'relative' (default) | 'ulp'

Floating-point tolerance check based on relative error or ULP. For more information, see "Floating point tolerance check based on".

FPToleranceValue — Floating-point tolerance strategy
'relative' (default) | 'ulp'

Floating-point tolerance value depending on the FPToleranceStrategy specified. For more information, see "Tolerance Value".

Port Names and Types

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

Specify whether to generate single or multiple clock inputs in the HDL code. For more information, see "Clock inputs".

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

Specify whether to use an active-high or active-low asserted level for the reset input signal. For more information, see "Reset asserted level".

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

Specify the clock enable input port name as a character vector. For more information, see "Clock enable input port".

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

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

For more information, see "Clock Enable output port".

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

Specify the clock input port name as a character vector. For more information, see "Clock input port".

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

Reset input port name, specified as a character vector.

For more information, see "Reset input port".

File and Variable Names

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

Specify the file name extension for generated Verilog files. For more information, see "Verilog file extension".

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

Specify the file name extension for generated VHDL files. For more information, see "VHDL file extension".

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

For more information, see "VHDL architecture name".

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

For more information, see "VHDL library name".

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

For more information, see "Split entity file postfix".

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

For more information, see "Split arch file postfix".

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

Specify the postfix for the package file name as a character vector. For more information, see "Package postfix".

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

For more information, see **Complex imaginary part postfix** in "Complex Signals Postfix Parameters".

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

For more information, see **Complex real part postfix** in "Complex Signals Postfix Parameters".

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

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

Coding Style

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

For more information, see "Split entity and architecture".

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

For more information, see "Use Verilog `timescale directives".

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

For more information, see **Emit time/date stamp in header** in "RTL Annotation Parameters".

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

For more information, see "Inline VHDL configuration".

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

For more information, see "Scalarize ports".

Script Generation

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

For more information, see "Compile initialization".

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

For more information, see "Compile termination".

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

For more information, see "Compile file postfix".

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 "Compile command for Verilog".

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 "Compile command for VHDL".

HDLSimCmd — HDL simulation command
'vsim -voptargs=+acc %s.%s\n' (default) | character vector

The HDL simulation command, specified as a character vector.

For more information, see "Simulation command".

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 "Simulation initialization".

2-113

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 "Simulation termination".

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

For more information, see "Simulation file postfix".

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 "Simulation waveform viewing command".

Version History
Introduced in R2006b

See Also
makehdl

genhdltdb

Generate timing databases for specified target device, device speed grade, and synthesis tool

Syntax

```
genhdltdb('Name','Value')
```

Description

`genhdltdb('Name','Value')` generates timing databases for the target device with options specified by using name-value arguments. You can specify the device family name, the device name, the device package, and the device speed grade. This function exports the generated timing database MAT-files to the specified timing database path.

To generate timing databases, the function characterizes basic design components (such as Simulink blocks, block architectures, and subcomponents of those blocks) for the specified target device. HDL Coder analyzes these timing databases to estimate the critical path in your design. For more information, see "Critical Path Estimation Without Running Synthesis".

Note This function generates timing databases for Xilinx devices only, not Intel and Microchip devices.

Examples

Generate Timing Data for Xilinx Artix-7 Target Device

Generate timing databases for the Xilinx Artix®-7 target device. Export generated timing database MAT-files to the `C:\Work\Database` folder. Set the synthesis tool path to `'C:\Xilinx\Vivado\2019.2\bin\vivado.bat'`. To generate timing database for the target device, use either of these commands:

- Specify target device configuration parameters, such as device family, device name, device package, and device speed grade, in a single argument `'SynthesisDeviceConfiguration'`.

```
genhdltdb('SynthesisDeviceConfiguration',{'Artix7','xa7a100t','csg324','-1I'}, ...
          'TimingDatabaseDirectory','C:\Work\Database', ...
          'SynthesisToolName','Xilinx Vivado', ...
          'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
```

  or

- Specify target device configuration parameter in separate arguments.

```
genhdltdb('SynthesisDeviceFamily','Artix7', ...
          'SynthesisDeviceName','xa7a100t', ...
          'SynthesisDevicePackage','csg324', ...
          'SynthesisDeviceSpeedGrade','-1I', ...
          'TimingDatabaseDirectory','C:\Work\Database', ...
```

```
                     'SynthesisToolName','Xilinx Vivado', ...
                     'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
```

> **Note** When you use the `genhdltdb` function to generate a timing database for a specific device, the function can run for a long time until the database is created.

Generate Timing Data for Xilinx Kintex UltraScale+ Target Device

Generate timing databases for the Xilinx Kintex® UltraScale+™ target device. The target device is `xcku11p-CIV-ffva1156-1-e`. Enter the device name without `'CIV'`, as it is not required for setting the part number. Export generated timing database MAT-files to the `C:\Work\Database` folder. Set the synthesis tool path to `'C:\Xilinx\Vivado\2019.2\bin\vivado.bat'`. To generate timing database for the target device, use either of these commands:

- Specify target device configuration parameters, such as device family and device name, in a single argument `'SynthesisDeviceConfiguration'`. In this target device, the device package and speed grade is included in the device name.

  ```
  genhdltdb( 'SynthesisDeviceConfiguration',{'Kintex Ultrascale+','xcku11p-ffva1156-1-e'}, ...
             'TimingDatabaseDirectory','C:\Work\Database', ...
             'SynthesisToolName','Xilinx Vivado', ...
             'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
  ```

  or

- Specify target device configuration parameters in separate arguments.

  ```
  genhdltdb('SynthesisDeviceFamily','Kintex Ultrascale+', ...
            'SynthesisDeviceName','xcku11p-ffva1156-1-e', ...
            'TimingDatabaseDirectory','C:\Work\Database', ...
            'SynthesisToolName','Xilinx Vivado', ...
            'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat');
  ```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat'` sets the synthesis tool path to `C:\Xilinx\Vivado\2019.2\bin\vivado.bat`.

SynthesisDevicePart — Target device part number
character vector | string scalar

Specify target device part number, specified as a character vector or string scalar.

Example: `'SynthesisDevicePart','xczu19eg-ffve1924-1-i'`

Data Types: `char` | `string`

SynthesisDeviceConfiguration — Target device configuration parameters
character vector | string scalar

Specify target device configuration parameters, such as device family name, device name, device package, and device speed grade, in a single argument, specified as a character vector or string scalar.

Example: `'SynthesisDeviceConfiguration',{'KintexU','xc7z100','ffg1156','-1'}`

Data Types: `char` | `string`

SynthesisDeviceFamily — Target device family
character vector | string scalar

Target device family, specified as a character vector or string scalar.

Example: `'SynthesisDeviceFamily', 'Artix7'`

Data Types: `char` | `string`

SynthesisDeviceName — Target device name
character vector | string scalar

Target device name, specified as a character vector or string scalar.

Example: `'SynthesisDeviceName','xa7a100t'`

Data Types: `char` | `string`

SynthesisDevicePackage — Target device package
character vector | string scalar

Target device package, specified as a character vector or string scalar.

Example: `'SynthesisDevicePackage','csg324'`

Data Types: `char` | `string`

SynthesisDeviceSpeedGrade — Target device speed grade
character vector | string scalar

Target device speed grade, specified as a character vector or string scalar.

Example: `'SynthesisDeviceSpeedGrade','-1I'`

Data Types: `char` | `string`

TimingDatabaseDirectory — Export timing database MAT-files to the specified directory path
character vector | string scalar

Export timing database MAT-files to the specified folder path, specified as a character vector or string scalar.

Example: `'TimingDatabaseDirectory','C:\Work\Database'`

Data Types: `char` | `string`

SynthesisToolName — Synthesis tool name
character vector | string scalar

Synthesis tool name, specified as a character vector or string scalar.

Example: `'SynthesisToolName','Xilinx Vivado'`

Data Types: `char` | `string`

SynthesisToolPath — Full path to synthesis tool executable or batch file
character vector | string scalar

Full path to the synthesis tool executable or batch file, specified as a character vector or string scalar. To set the synthesis tool path, use either this argument or the `hdlsetuptoolpath` function. If the path is set by both, the `genhdltdb` function uses the tool path set by this argument.

Example: `'SynthesisToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat'`

Data Types: `char` | `string`

Override — Override generated timing database MAT-files
`'on'` (default) | `'off'`

Override the generated timing database MAT-files in the output path set by the `outpath` argument, specified as one of these values:

- `'on'` — Generate new timing database MAT-files for the blocks for every call to this function.
- `'off'` — Generate timing database MAT-files for only the blocks whose MAT-files are not present in the output path.

Data Types: `char` | `string`

Version History
Introduced in R2021a

See Also
`hdlsetuptoolpath` | **Custom Timing Database Directory**

Topics
"Critical Path Estimation Without Running Synthesis"

addAXI4SlaveInterface

Write data to IP core or read data from IP core using AXI4 or AXI4-Lite interface

Syntax

```
addAXI4SlaveInterface(hFPGA)
addAXI4SlaveInterface(hFPGA, Name,Value)
```

Description

`addAXI4SlaveInterface(hFPGA)` adds an AXI4 slave interface that you can use to control the DUT ports mapped to AXI4 or AXI4-Lite interfaces in the HDL Coder generated IP core from MATLAB.

`addAXI4SlaveInterface(hFPGA, Name,Value)` adds an AXI4 slave interface that you can use to control the DUT ports mapped to AXI4 or AXI4-Lite interfaces in the HDL Coder generated IP core from MATLAB, with one or more properties specified as name-value pair arguments. Enclose each property and value pair in single quotes.

Examples

Add AXI4 Slave Interface for Xilinx Target

Add an AXI4 slave interface for a Xilinx target.

Create a target object, `hFPGA`, for the target device.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the `hFPGA` object by using the `addAXI4SlaveInterface` function.

```
%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mmwr0", ...
```

```
    "ReadDeviceName", "mwipcore0:mmrd0");
```

After you have added the interfaces, use the `mapPort` function to map the ports to that interface, and then read and write data. See "Map DUT Ports in HDL IP Core to AXI4 Slave Interfaces" on page 2-132.

Add AXI4 Slave Interface for Intel Target

Add an AXI4 slave interface for an Intel target.

Create a target object, `hFPGA`, for an Intel target.

```
hFPGA = fpga("Intel")

hFPGA =

  fpga with properties:

       Vendor: "Intel"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the `hFPGA` object by using the `addAXI4SlaveInterface` function.

```
%% AXI4
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mmwr0", ...
    "ReadDeviceName", "mwipcore0:mmrd0");
```

After you have added the interfaces, use the `mapPort` function to map the ports to that interface, and then read and write data. See "Map DUT Ports in HDL IP Core to AXI4 Slave Interfaces" on page 2-132.

Add AXI4 Slave Interface for Standalone FPGA Boards

Add an AXI4 slave interface for a standalone Xilinx target.

Create a target object, `hFPGA`, for the target device.

```
hFPGA = fpga("Xilinx")

hFPGA =
```

```
fpga with properties:

     Vendor: "Xilinx"
 Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

As standalone FPGA boards do not have an embedded ARM® processor, you can use the AXI Manager driver. Use the `aximanager` object to specify the AXI Manager driver and then add this information to the `addAXI4SlaveInterface` function.

```matlab
% Create an "aximanager" object
hAXIMDriver = aximanager("Xilinx");

% Pass it into the addInterface command
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID",  "AXI4-Lite", ...
    "BaseAddress",  0xB0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDriver", hAXIMDriver, ...
    "ReadDriver", hAXIMDriver, ...
    "DriverAddressMode", "Full");
```

After you have added the interfaces, use the `mapPort` function to map the ports to that interface, and then read and write data. See "Map DUT Ports in HDL IP Core to AXI4 Slave Interfaces" on page 2-132.

Input Arguments

hFPGA — Target FPGA object
`fpga object`

`fpga` object for the target vendor, specified as an `fpga` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `addAXI4SlaveInterface(hFPGA, "InterfaceID", "AXI4-Lite")` creates an AXI4 slave interface with `InterfaceID` as `AXI4-Lite`.

InterfaceID — Name of AXI4 slave interface
`"AXI4-Lite"` | `"AXI4"` | `string`

Name of AXI4-Lite or AXI4 interface that you want to map the DUT ports to, specified as a string.

BaseAddress — Base address
`0x0` (default) | numeric value

Base address for AXI4 or AXI4-Lite slave interface, specified as a numeric value.

Example: `0x40010000`

AddressRange — Address range
numeric value

Address range for AXI4 or AXI4-Lite interface, specified as a numeric value.

Example: `0x10000`

WriteDeviceName — IIO device name to write data
`"mwipcore0:mmwr0"` (default) | string array

Name and path of the IIO device that you want to write to. When you generate the IP core by using the `IP Core Generation` workflow, the default name is `mwipcore0:mmwr0`.

Example: `"mwipcore0:mmwr0"`

ReadDeviceName — IIO device name to read data
`"mwipcore0:mmrd0"` (default) | string array

Name and path of the IIO device that you want to read from. When you generate the IP core by using the `IP Core Generation` workflow, the default name is `mwipcore0:mmrd0`.

Example: `"mwipcore0:mmrd0"`

WriteDriver — AXI driver to perform write operation
driver object

Name of the AXI driver that you use to write data into. You can specify this property as the HDL Verifier `aximanager` object for standalone FPGA boards. For SoC platforms, HDL Coder creates the drivers automatically.

Example: `"aximanager('Xilinx')"`

ReadDriver — AXI driver to perform read operation
driver object

Name of the AXI driver that you use to read data from. You can specify this property as the HDL Verifier `aximanager` object for standalone FPGA boards that do not have an embedded ARM processor. For SoC platforms, HDL Coder creates the drivers automatically.

Example: `"aximanager('Xilinx')"`

DriverAddressMode — AXI driver to perform read operation
`"Offset"` (default) | `"Full"`

Specify whether the AXI driver expects a full address that includes the base address and the offset address, or whether it expects only an offset address.

Example: `"Offset"`

Version History
Introduced in R2020b

See Also

Objects
fpga | hdlcoder.DUTPort

Functions
mapPort | writePort | readPort | addAXI4StreamInterface

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

addAXI4StreamInterface

Write data to IP core or read data from IP core using AXI4-Stream interface

Syntax

```
addAXI4StreamInterface(hFPGA)
addAXI4StreamInterface(hFPGA, Name,Value)
```

Description

addAXI4StreamInterface(hFPGA) adds an AXI4-Stream interface that you can use to control the DUT ports mapped to AXI4-Stream interfaces in the HDL Coder generated IP core from MATLAB.

addAXI4StreamInterface(hFPGA, Name,Value) adds an AXI4-Stream interface that you can use to control the DUT ports mapped to AXI4-Stream interfaces in the HDL Coder generated IP core from MATLAB, with one or more properties specified as name-value pair arguments. Enclose each property and value pair in single quotes.

Examples

Add AXI4-Stream Interface to Control HDL IP Core

Add an AXI4-Stream interface to control HDL IP core generated for a Xilinx target.

Create a target object, hFPGA, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the hFPGA object by using the addAXI4StreamInterface function.

```
addAXI4StreamInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
    "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mm2s0", ...
```

```
"ReadDeviceName", "mwipcore0:s2mm0",...
"WriteDataWidth", 32, ...
"ReadDataWidth", 32);
```

After you have added the interfaces, use the `mapPort` function to map the ports to that interface, and then read and write data. See "Map DUT Ports in HDL IP Core to AXI4-Stream Interfaces" on page 2-133.

Add AXI4-Stream Interface with Only Write Channel

Add an AXI4-Stream interface for a Xilinx target with only a write channel.

Create a target object, `hFPGA`, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

      Vendor: "Xilinx"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the `hFPGA` object by using the `addAXI4StreamInterface` function. Set `ReadEnable` to `false`.

```
addAXI4StreamInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Stream", ...
    "ReadEnable", false, ...
    "WriteFrameLength", 1024, ...
    ... % Driver properties
    "WriteDeviceName", "mwipcore0:mm2s0");
```

After you have added the interfaces, use the `mapPort` function to map the ports to that interface, and then read and write data. See "Map DUT Ports in HDL IP Core to AXI4-Stream Interfaces" on page 2-133.

Input Arguments

hFPGA — Target FPGA object
`fpga object`

`fpga` object for the target vendor, specified as an `fpga` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `addAXI4StreamInterface(hFPGA, "InterfaceID", "AXI4-Stream")` creates an AXI4-Stream interface with `InterfaceID` as `AXI4-Stream`.

`InterfaceID` — Name of AXI4-Stream interface
`"AXI4-Stream"` | string

Name of AXI4-Stream interface that you want to map the DUT ports to, specified as a string.

`WriteEnable` — Write channel enable signal
`true` (default) | `false`

Write channel enable signal, specified as either `true` or `false`. By default, `WriteEnable` is `true`, and you can write data into the slave channel of the AXI4-Stream interface on the IP core.

`ReadEnable` — Read channel enable signal
`true` (default) | `false`

Read channel enable signal, specified as either `true` or `false`. By default, `ReadEnable` is `true`, and you can read data from the master channel of the AXI4-Stream interface on the IP core.

`WriteDeviceName` — IIO device to write data
`"mwipcore0:mm2s0"` (default) | string array

Name and path of the IIO core device that you want to write to, specified as a string. When you generate the IP core by using the `IP Core Generation` workflow, the default name is `mwipcore0:mmw2s0`.

`ReadDeviceName` — IIO device to read data
`"mwipcore0:s2mm0"` (default) | string array

Name and path of the IIO device that you want to read from. When you generate the IP core by using the `IP Core Generation` workflow, the default name is `mwipcore0:s2mm0`.

`WriteFrameLength` — Write channel frame length
`1024` (default) | scalar

Size of the data vector to be written to the IIO device, specified as a scalar. Use this parameter to determine the number of samples written for each DMA transfer.

`ReadFrameLength` — Write channel frame length
`1024` (default) | scalar

Size of the data vector that is read from the IIO device, specified as a scalar. Use this parameter to determine the number of samples read for each DMA transfer.

`WriteTimeout` — Timeout for AXI4-Stream write
`0` (default) | scalar

Maximum timeout for the AXI4-Stream write, specified as a scalar.

ReadTimeout — Timeout for AXI4-Stream read
0 (default) | `scalar`

Maximum timeout for the AXI4-Stream read, specified as a scalar.

WriteDataWidth — Write channel data width
32 (default) | 8 | 16 | 32 | 64 | 128

Write channel data width, specified as an integer.

Example: `'WriteDataWidth',32` specifies write channel data width of 32 bits.

ReadDataWidth — Read channel data width
32 (default) | 8 | 16 | 32 | 64 | 128

Read channel data width, specified as an integer.

Example: `'ReadDataWidth',32` specifies read channel data width of 32 bits.

Version History
Introduced in R2020b

See Also

Objects
`fpga` | `hdlcoder.DUTPort`

Functions
`mapPort` | `writePort` | `readPort` | `addAXI4SlaveInterface`

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

addMemoryInterface

Access memory regions on your FPGA or SoC hardware

Syntax

```
addMemoryInterface(hFPGA)
addMemoryInterface(hFPGA,Name=Value)
```

Description

addMemoryInterface(hFPGA) adds a memory interface to the FPGA, hFPGA, that you can use to access regions of memory on your hardware by using MATLAB. Typically, these are regions of memory that are accessible by the AXI4 Master interface on your HDL Coder IP core. Accessing the same regions of memory from MATLAB allows you to exchange data with your IP core.

addMemoryInterface(hFPGA,Name=Value) adds a memory interface with one or more options specified as name-value arguments.

Examples

Add Memory Interface

Create a memory interface to read and write data from accessible memory regions.

Create an fpga object, hFPGA, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the hFPGA object by using the addMemoryInterface function. Save the memory interface to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA)

hInterface =

  Memory with properties:

     InterfaceID: "Memory"
     BaseAddress: "0x0"
    AddressRange: "0x100000000"
     WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
```

```
        ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
        InputPorts: [0×0 string]
       OutputPorts: [0×0 string]
```

Add Memory Interface for a Specified Address Space

Use name-value arguments to create a memory interface at a specified address space.

Create a `fpga` object, `hFPGA`, for a Xilinx target.

```
hFPGA = fpga("Xilinx")


hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the `hFPGA` object by using the `addMemoryInterface` function. Specify the memory interface name, base address and address range. Save the memory interface object to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA, InterfaceID = "myInterface",...
BaseAddress = 0x80000000,AddressRange = 0x20000000)

hInterface =

  Memory with properties:

      InterfaceID: "myInterface"
      BaseAddress: "0x80000000"
     AddressRange: "0x20000000"
      WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
       ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
       InputPorts: [0×0 string]
      OutputPorts: [0×0 string]
```

Input Arguments

hFPGA — Target FPGA object
`fpga object`

Target FPGA object, specified as an `fpga` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `hInterface = addMemoryInterface(hFPGA,InterfaceID = "myInterface")` creates a memory interface object with `InterfaceID` as `myInterface`.

`InterfaceID` — Name of memory interface
`"Memory"` | string

Name of memory interface, specified as a string.

`BaseAddress` — Base address
`0x0` (default) | numeric value

Base address of the memory interface, specified as a numeric value.

Example: `0x40010000`

`AddressRange` — Address range
numeric value

Address range for the memory interface, specified as a numeric value.

Example: `0x10000`

`WriteDeviceName` — Name and path of IIO device
`"mwipcore0:sharedmem0:wr0"` (default) | string array

Name and path of the IIO device to write to, specified as a string array. When you generate an IP core using the **HDL Workflow Advisor**, the default name is `mwipcore0:sharedmem0:wr0`.

Example: `"mwipcore0:sharedmem0:wr0"`

`ReadDeviceName` — Name and path of IIO device
`"mwipcore0:sharedmem0:rd0"` (default) | string array

Name and path of the IIO device to read from, specified as a string array. When you generate an IP core using the **HDL Workflow Advisor**, the default name is `mwipcore0:sharedmem0:rd0`.

Example: `mwipcore0:sharedmem0:rd0`

`WriteDriver` — AXI driver to perform memory write operation
driver object

Name of the AXI driver that you use to write data into. You can specify this property as the HDL Verifier `aximanager` object for standalone FPGA boards. For SoC platforms, HDL Coder creates the drivers automatically.

Example: `"aximanager('Xilinx')"`

`ReadDriver` — AXI driver to perform memory read operation
driver objectobject

Name of the AXI driver that you use to read data from. You can specify this property as the HDL Verifier `aximanager` object for standalone FPGA boards that do not have an embedded ARM processor. For SoC platforms, HDL Coder creates the drivers automatically.

Example: `"aximanager('Xilinx')"`

`AddressWidth` — Address width of memory region accessed by memory interface
32 (default) | 64

Address width of the memory region that is accessed by this memory interface, specified as either 32 bits or 64 bits.

Example: `AddressWidth=32`

Version History
Introduced in R2023a

See Also

Objects
`fpga` | `hdlcoder.DUTPort`

Functions
`mapPort` | `writePort` | `readPort` | `addAXI4StreamInterface` | `readMemory` | `writeMemory`

mapPort

Maps a DUT port to specified AXI4 interface in HDL IP core

Syntax

```
mapPort(hFPGA, hDUTPorts)
```

Description

mapPort(hFPGA, hDUTPorts) maps the DUT port or ports in the generated HDL IP core to the AXI4 interface. You create the DUT port objects as an object array by using the hdlcoder.DUTPort object.

Examples

Map DUT Ports in HDL IP Core to AXI4 Slave Interfaces

This example shows how to map the DUT ports in the HDL IP core to AXI4 slave interfaces.

Create an fpga object with Xilinx as Vendor.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the hFPGA object by using the addAXI4SlaveInterface function.

```
%% AXI4 Slave
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000);
```

Specify the DUT ports in the HDL IP core as an hdlcoder.DUTPort object array and then map the port to the AXI4 slave interface.

```
hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
```

```
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");
```

Map the DUT port objects to the AXI4 slave interface. This information is saved as a property on the hFPGA object.

```
mapPort(hFPGA, hPort_h_in1, hPort_h_in2)
hFPGA.Interfaces
```

```
ans =

  AXI4Slave with properties:

     InterfaceID: "AXI4-Lite"
     BaseAddress: "0xA0000000"
    AddressRange: "0x10000"
     WriteDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIOWrite]
      ReadDriver: [1×1 fpgaio.driver.AXIMemoryMappedIIORead]
      InputPorts: "h_in1"
     OutputPorts: [0×0 string]
```

Map DUT Ports in HDL IP Core to AXI4-Stream Interfaces

This example shows how to map the DUT ports in the generated HDL IP core to AXI4-Stream interfaces.

Create an object for the target device.

```
hFPGA = fpga("Xilinx")
```

```
hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the hFPGA object by using the addAXI4StreamInterface function.

```
%% AXI4-Stream
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
        "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024);
```

Specify the DUT port as an hdlcoder.DUTPort object array and then map the port to the AXI4-Stream interface.

```
hPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

hPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,20), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");
```

Map the DUT port objects to the AXI4-Stream interface. This information is saved as a property on the hFPGA object.

```
%% Map DUT port to AXI4 Slave driver
mapPort(hFPGA, [hPort_x_in_data, hPort_y_out_data]);
hFPGA.Interfaces


ans =

   AXI4Stream with properties:

          InterfaceID: "AXI4-Stream"
          WriteEnable: 1
           ReadEnable: 1
      WriteFrameLength: 1024
       ReadFrameLength: 1024
          WriteDriver: [1×1 fpgaio.driver.AXIStreamIIOWrite]
           ReadDriver: [1×1 fpgaio.driver.AXIStreamIIORead]
           InputPorts: "x_in_data"
          OutputPorts: "y_out_data"
```

Input Arguments

hFPGA — Target FPGA object
fpga object

fpga object for the target vendor, specified as an fpga object.

hDUTPorts — Object array of DUT ports
hdlcoder.DUTPort object

DUT port or ports that you want to map to an IP core interface. Each port is an object array that you create by using the hdlcoder.DUTPort function.

Version History
Introduced in R2020b

See Also

Objects
fpga | hdlcoder.DUTPort

Functions
writePort | readPort | addAXI4SlaveInterface | addAXI4StreamInterface

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

readMemory

Read data from memory regions on FPGA or SoC hardware using MATLAB

Syntax

```
data = readMemory(hFPGA,addr,dataLen)
```

Description

`data = readMemory(hFPGA,addr,dataLen)` reads `dataLen` locations of data from the FPGA, `hFPGA`, starting from the address specified in `addr` and incrementing the address for each word. The address must be in the range of one of the AXI4 Slave or Memory interfaces added to the `hFPGA` object. The read operation connects to the appropriate interface based on the address specified.

Examples

Read Scalar Data from Memory

Read scalar data from FPGA-accessible memory.

Create a `fpga` object, `hFPGA`, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the `hFPGA` object by using the `addMemoryInterface` function. Specify the memory interface name, base address, and address range. Save the memory interface object to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA, InterfaceID = "myInterface",...
BaseAddress = 0x80000000,AddressRange = 0x20000000)

hInterface =

  Memory with properties:

      InterfaceID: "myInterface"
      BaseAddress: "0x80000000"
     AddressRange: "0x20000000"
      WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
       ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
```

```
        InputPorts: [0×0 string]
       OutputPorts: [0×0 string]
```

Write data to a memory location.

```
writeMemory(hFPGA,0x80000000,uint32(5));
```

Read data from a single location.

```
data = readMemory(hFPGA,0x80000000,1)

data =

    uint32

        5
```

Read Vector Data from Memory

Read vector data from FPGA accessible memory.

Create a `fpga` object, `hFPGA`, for a Xilinx target.

```
hFPGA = fpga("Xilinx")


hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the `hFPGA` object by using the `addMemoryInterface` function. Specify the memory interface name, base address, and address range. Save the memory interface object to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA, InterfaceID = "myInterface",...
BaseAddress = 0x80000000,AddressRange = 0x20000000)

hInterface =

  Memory with properties:

      InterfaceID: "myInterface"
      BaseAddress: "0x80000000"
     AddressRange: "0x20000000"
      WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
       ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
       InputPorts: [0×0 string]
      OutputPorts: [0×0 string]
```

Write data to a memory location.

```
writeMemory(hFPGA,0x80000000,uint32(1:5));
```

Read data from a single location.

```
data = readMemory(hFPGA,0x80000000,5)

data =

    uint32

        1 2 3 4 5
```

Input Arguments

hFPGA — Target FPGA object
fpga object

Target FPGA object, specified as an `fpga` object.

addr — Starting address for read operation
nonnegative integer | nonnegative hexadecimal value

Starting address for the read operation, specified as a nonnegative integer that is a multiple of 4 or hexadecimal value that is a multiple of 4. The function casts the address to the `uint32` or `uint64` data type, depending on the memory interface address width. The address must be in the range of one of the AXI4 Slave or Memory interfaces added to the `hFPGA` object.

Example: `0x80000000` specifies a starting address of `0x80000000`.

Data Types: `uint32` | `uint64`

dataLen — Length of data to read
nonnegative integer

Length of data to read, starting at `addr`, specified as a nonnegative integer. By default, the function reads data from a contiguous address block, incrementing the address for each operation.

When you specify a large operation size, such as reading a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size of 256 words.

Example: 5 specifies five contiguous memory locations.

Output Arguments

data — Read data
scalar | vector

Read data, returned as a scalar or vector depending on the value you specify for the `dataLen`.

Version History
Introduced in R2023a

See Also

Objects
fpga | addMemoryInterface

Functions
mapPort | writePort | readPort | addAXI4StreamInterface | writeMemory

readPort

Reads output data and returns it with the port data type and dimension

Syntax

```
data = readPort(hFPGA, portName)
data,valid = readPort(hFPGA, portName)
```

Description

`data = readPort(hFPGA, portName)` reads the output data and returns this value with the port data type and dimension.

`data,valid = readPort(hFPGA, portName)` reads the output data and returns this value with the port data type and dimension and an optional output whether the output data is valid.

Examples

Read Data from DUT Port Mapped to AXI4 Slave Interfaces

This example shows how to read data from the DUT ports that are mapped to AXI4 slave interfaces.

Create an `fpga` object with `Xilinx` as `Vendor`.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the `hFPGA` object by using the `addAXI4SlaveInterface` function.

```
%% AXI4 Slave
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000);
```

Specify the DUT ports in the HDL IP core as an `hdlcoder.DUTPort` object array and then map the port to the AXI4 slave interface.

```
hPort = hdlcoder.DUTPort("h_out1", ...
```

```
    "Direction", "OUT", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x104");
```

Map the DUT port objects to the AXI4 slave interface and then read data by using the `readPort` function.

```
mapPort(hFPGA, hPort);

data = readPort(hFPGA, "h_out1");
```

Read Data from DUT Port Mapped to AXI4-Stream Interfaces

This example shows how to read data from the DUT ports that are mapped to AXI4-Stream interfaces.

Create an object for the target device.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the `hFPGA` object by using the `addAXI4StreamInterface` function.

```
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
     "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024);
```

Specify the DUT port as an `hdlcoder.DUTPort` object array and then map the port to the AXI4-Stream interface.

```
hPort = hdlcoder.DUTPort("y_out", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");
```

Map the DUT port objects to the AXI4-Stream interface and then read data by using the `readPort` function.

```
mapPort(hFPGA, hPort);

data = readPort(hFPGA, "y_out");
```

Input Arguments

hFPGA — Target FPGA object
`fpga object`

`fpga` object for the target vendor, specified as an `fpga` object.

portName — DUT port name
`string`

DUT port name, specified as a string. You create the DUT port as an `hdlcoder.DUTPort` object array. Before you specify the `portName`, you must have mapped the port to an AXI interface by using the `mapPort` function.

Output Arguments

data — Output data
`Scalar` (default) | `Vector`

Output data that is read from the DUT port, `PortName`, returned as a scalar or a vector.

valid — Read data valid true or false result
true | false | boolean

Read data valid, returned as true or false of data type `boolean`. This argument is only available when the `ReadTimeout` value is set to a finite value and the port is mapped to an AXI4-Stream interface.

Version History
Introduced in R2020b

See Also

Objects
`fpga`

Functions
`mapPort` | `writePort`

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

release

Release the hardware resources associated with the fpga object

Syntax

```
release(hFPGA)
```

Description

`release(hFPGA)` releases the hardware resources associated with the `fpga` object, represented by hFPGA.

Examples

Release Connection to a Xilinx Target

Create an hFPGA object for a Xilinx target and then release the hardware resource.

Create an hFPGA object for a Xilinx target and connect to the Xilinx device. To release the hardware resource, use the `release` function.

```
hFPGA = fpga("Xilinx")
release(hFPGA)

hFPGA =

  fpga with properties:

      Vendor: "Xilinx"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Release Connection to an Intel Target

Create an hFPGA object for an Intel target and then release the connection you have set up.

Create an hFPGA for an Intel target and connect to the Intel device. To release the connection, use the `release` function.

```
hFPGA = fpga("Intel")
release(hFPGA)

hFPGA =

  fpga with properties:
```

```
       Vendor: "Intel"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Input Arguments

hFPGA — Target FPGA object
fpga object

fpga object for the target vendor, specified as an fpga object.

Version History
Introduced in R2020b

See Also

Objects
fpga | hdlcoder.DUTPort

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

writeMemory

Write data to memory regions on FPGA or SoC hardware

Syntax

```
writeMemory(hFPGA,addr,data)
```

Description

writeMemory(hFPGA,addr,data) writes all words specified in data to the FPGA, hFPGA, starting from the address specified in addr and incrementing the address for each word. The address must be in the range of one of the AXI4 Slave or Memory interfaces added to the hFPGA object. The write operation writes to the appropriate interface based on the address provided.

Examples

Write Scalar Data to Memory

Write scalar data to FPGA-accessible memory.

Create a fpga object, hFPGA, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

       Vendor: "Xilinx"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the hFPGA object by using the addMemoryInterface function. Specify the memory interface name, base address, and address range. Save the memory interface object to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA, InterfaceID = "myInterface",...
BaseAddress = 0x80000000,AddressRange = 0x20000000)

hInterface =

  Memory with properties:

       InterfaceID: "myInterface"
       BaseAddress: "0x80000000"
      AddressRange: "0x20000000"
       WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
        ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
```

```
        InputPorts: [0×0 string]
       OutputPorts: [0×0 string]
```

Write data to a memory location.

```
writeMemory(hFPGA,0x80000000,uint32(5));
```

Write Vector Data to Memory

Write vector data to FPGA-accessible memory.

Create a `fpga` object, `hFPGA`, for a Xilinx target.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the memory interface to the `hFPGA` object by using the `addMemoryInterface` function. Specify the memory interface name, base address, and address range. Save the memory interface object to *hInterface*.

```
hInterface = addMemoryInterface(hFPGA, InterfaceID = "myInterface",...
BaseAddress = 0x80000000,AddressRange = 0x20000000)

hInterface =

  Memory with properties:

      InterfaceID: "myInterface"
      BaseAddress: "0x80000000"
     AddressRange: "0x20000000"
      WriteDriver: [1×1 matlabshared.libiio.sharedmem.write]
       ReadDriver: [1×1 matlabshared.libiio.sharedmem.read]
       InputPorts: [0×0 string]
      OutputPorts: [0×0 string]
```

Write data to a memory location.

```
writeMemory(hFPGA,0x80000000,uint32(1:5));
```

Input Arguments

hFPGA — Target FPGA object
`fpga object`

Target FPGA object, specified as an `fpga` object.

addr — Starting address for write operation

nonnegative integer | nonnegative hexadecimal value

Starting address for the write operation, specified as a nonnegative integer that is a multiple of 4 or hexadecimal value that is a multiple of 4. The function casts the address to the `uint32` or `uint64` data type, depending on the memory interface address width. The address must be in the range of one of the AXI4 Slave or Memory interfaced added to the `hFPGA` object.

Example: `0x80000000` specifies a starting address of `0x80000000`.

Data Types: `uint32` | `uint64`

data — Data words to write

scalar | vector

Data words to write, specified as a `uint32` scalar or vector. By default, the function writes the data to a contiguous address block and increments the address for each operation.

When you specify a large operation size, such as when you write a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size of 256 words.

Example: `[1:100]` specifies 100 contiguous memory locations.

Data Types: `uint32`

Version History
Introduced in R2023a

See Also

Objects
`fpga` | `addMemoryInterface`

Functions
`mapPort` | `writePort` | `readPort` | `addAXI4StreamInterface` | `readMemory`

writePort

Write data to a DUT port from MATLAB

Syntax

```
writePort(hFPGA, portName, data)
valid=writePort(hFPGA, portName, data)
```

Description

writePort(hFPGA, portName, data) casts the input data, specified by data, to the data type of the port, portName, and dispatches to the interface mapped to that port to write the data. Before you write the data, set up a connection from MATLAB to the target FPGA or SoC device, hFPGA, and then use the mapPort function to map the portName to that interface.

valid=writePort(hFPGA, portName, data) returns a valid signal to indicate that the data write was completed.

Examples

Write Data to DUT Port Mapped to AXI4 Slave Interfaces

This example shows how to write data to the DUT ports that are mapped to AXI4 slave interfaces.

Create an fpga object with Xilinx as Vendor.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

   Top-Level Properties

      Vendor: "Xilinx"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the hFPGA object by using the addAXI4SlaveInterface function.

```
%% AXI4 Slave
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000);
```

Specify the DUT ports in the HDL IP core as an `hdlcoder.DUTPort` object array and then map the port to the AXI4 slave interface.

```
hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");
```

Map the DUT port objects to the AXI4 slave interface and then write data by using the `writePort` function.

```
mapPort(hFPGA, hPort_h_in1);

writePort(hFPGA, "h_in1", 5);
```

Write Data to DUT Port Mapped to AXI4-Stream Interfaces

This example shows how to write data to the DUT ports that are mapped to AXI4-Stream interfaces.

Create an object for the target device.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the `hFPGA` object by using the `addAXI4StreamInterface` function.

```
%% AXI4-Stream
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
     "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024);
```

Specify the DUT port as an `hdlcoder.DUTPort` object array and then map the port to the AXI4-Stream interface.

```
hPort = hdlcoder.DUTPort("x_in", ...
```

```
        "Direction", "IN", ...
        "DataType", numerictype(1,16,10), ...
        "Dimension", [1 1], ...
        "IOInterface", "AXI4-Stream");
```

Map the DUT port objects to the AXI4-Stream interface and then write data by using the `writePort` function.

```
mapPort(hFPGA, hPort);
```

```
writePort(hFPGA, "x_in", sin(linspace(0, 2*pi, 1024)));
```

Input Arguments

hFPGA — Target FPGA object
`fpga object`

`fpga` object for the target vendor, specified as an `fpga` object.

portName — DUT port name
`string`

DUT port name, specified as a string. You create the DUT port as an `hdlcoder.DUTPort` object array. Before you specify the `portName`, you must have mapped the port to an AXI interface by using the `mapPort` function.

data — Input data
`Scalar` (default) | `Vector`

Input data to write to the DUT port, `PortName`, specified as a scalar or a vector.

Output Arguments

valid — Write data valid true or false result
true | false | boolean

Write data valid, returned as true or false of data type `boolean`. This argument is only available when the `WriteTimeout` value is set to a finite value and the port is mapped to an AXI4-Stream interface.

Version History
Introduced in R2020b

See Also

Objects
`fpga` | `hdlcoder.DUTPort`

Functions
`mapPort` | `readPort` | `addAXI4SlaveInterface` | `addAXI4StreamInterface`

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

system

Run command in Linux shell on SoC board

Syntax

```
result = system(p, command)
```

Description

`result = system(p, command)` runs the system command specified in a Linux® shell on a connected SoC board.

Examples

Connect to Xilinx SoC Board from MATLAB

Create a hardware processor object to connect to a Xilinx SoC board. You can use the object to perform basic Linux shell operations on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Run the `uname` command in a Linux shell on the SoC board by using the `xilinxsoc` object function `system`.

```
system(p, 'uname')

ans =

    'Linux'
```

Input Arguments

p — Connection to processor on SoC board
`xilinxsoc` object | `intelsoc` object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a `xilinxsoc` object or `intelsoc` object. This connection allows the function `system` to send system commands from MATLAB to the hardware board.

command — Linux Command
character vector | string scalar

Linux shell command that you want to run on your SoC board. You can run multiple different Linux system commands, such as listing files in a directory, displaying information about the system, or creating a new directory.

Example: `'ls -al'`

Example: `'uname'`

Output Arguments

`result` — Result of system command
character vector

Output of the system command run in the Linux shell on your connected SoC board. If there is no output based on the system command you run, `result` is an empty character vector.

Version History
Introduced in R2022a

See Also

Objects
`xilinxsoc` | `intelsoc`

Functions
`getFile` | `putFile` | `deleteFile` | `dir` | `programFPGA`

getFile

Transfer file from SoC board to host computer

Syntax

```
getFile(p, source)
getFile(p, source, destination)
```

Description

getFile(p, source) transfers the specified file from the board to the current MATLAB folder on the host computer.

getFile(p, source, destination) transfers the specified file from the board to the location specified on the host computer.

Examples

Transfer File from SoC Board to Current Folder

Transfer a file from a specified SoC board to the current MATLAB folder on the host computer.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the getFile function to transfer a file located in the file path filePath specified with the file name exampleFile.txt to the current MATLAB folder.

```
getFile(p, "filePath/exampleFile.txt")
```

The file exampleFile.txt is now transferred to your current folder.

Transfer File from SoC Board to a Specified Folder

Transfer a file from a specified SoC board to the specified folder on the host computer.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the getFile function to transfer a file located in the file path filePath specified with the file name exampleFile.txt to the Documents folder on your host computer located in the file path C:/User/.

```
getFile(p, "filePath/exampleFile.txt", "C:/User/Documents")
```

The file `exampleFile.txt` is now transferred to your `Documents` folder on your host computer.

Input Arguments

p — Connection to processor on SoC board
xilinxsoc object | intelsoc object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a `xilinxsoc` object or `intelsoc` object.

source — Path and name of file on SoC board
character vector | string scalar

Path and name of file on the SoC board that you want to transfer to your host computer. You can use either the absolute path from the root folder, or the relative path from the your home directory.

Example: `"~/exampleFile.txt"`

Example: `"/home/user/exampleFile.txt"`

destination — Path and name of folder or file on host computer
character vector | string scalar

Path of folder or file on the host computer where you want to transfer the file. If `destination` is the path to a folder, the file is copied inside that folder with the original file name. If `destination` is the path to a file, the file is copied with the new file name. This function does not create new folders, so folders in the path must already exist.

If no `destination` is specified, the current MATLAB folder is used.

Example: `"C:/User/Documents"`

Example: `"C:/User/Documents/exampleFileHost.txt"`

Version History
Introduced in R2022a

See Also

Objects
xilinxsoc | intelsoc

Functions
putFile | system | deleteFile | dir | programFPGA

putFile

Transfer file from host computer to SoC board

Syntax

```
putFile(p, source)
putFile(p, source, destination)
```

Description

`putFile(p, source)` transfers the specified file from the host computer to the home directory on the SoC board.

`putFile(p, source, destination)` transfers the specified file from the host computer to the location specified on the SoC board.

Examples

Transfer File from Host Computer to Home Directory on SoC Board

Transfer a file from your host computer to the user home directory on the specified SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the `putFile` function to transfer a file located in the working MATLAB folder specified with the file name `exampleFile.txt` to the user home directory on the SoC board.

```
putFile(p, "exampleFile.txt")
```

The file `exampleFile.txt` is now transferred to your home directory on the connected SoC board.

Transfer File from Host Computer to a Specified Folder on SoC Board

Transfer a file from your host computer to a specified folder on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the `putFile` function located in the working MATLAB folder specified with the file name `exampleFile.txt` to the `Documents` folder located in the file path `/home/user/` on the SoC board.

```
putFile(p, "exampleFile.txt", "/home/user/Documents")
```

The file `exampleFile.txt` is now transferred to your `Documents` folder on the connected SoC board.

Input Arguments

p — Connection to processor on SoC board
`xilinxsoc` object | `intelsoc` object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a `xilinxsoc` object or `intelsoc` object.

source — Path and name of file on host computer
character vector | string scalar

Path and name of file on the host computer that you want to transfer to your SoC board. You can use either the absolute path or the relative path from the your current working MATLAB folder.

Example: `"exampleFile.txt"`

Example: `"C:/User/Documents/MATLAB/exampleFile.txt"`

destination — Path and name of folder or file on SoC board
character vector | string scalar

Path and name of folder or file on the SoC board where you want to transfer a file from the host computer. The `destination` path can either be:

- A relative path from the user home directory.
- An absolute path to a folder.

The `destination` can be either:

- A folder path where the file is stored in that folder with the original name from the host computer file.
- A path to a new file, and the file transferred is stored in the specified path as the new file name.

If no `destination` is specified, the user home directory on the SoC board is used.

Example: `"/home/user/Documents"`

Example: `"Documents"`

Example: `"/home/user/Documents/exampleFileBoard.txt"`

Version History
Introduced in R2022a

See Also

Objects
`xilinxsoc` | `intelsoc`

Functions
getFile | system | deleteFile | dir | programFPGA

deleteFile

Delete file on SoC board

Syntax

```
deleteFile(p, filename)
```

Description

`deleteFile(p, filename)` deletes the file specified by the input `filename` on the SoC board.

Examples

Delete File from SoC Board

Delete a file from a specified SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the `deleteFile` function to delete a file located in your home directory on the SoC board, specified with the file name `exampleFile.txt`.

```
deleteFile(p, "exampleFile.txt")
```

The file `exampleFile.txt` is now deleted from your home directory on the connected SoC board.

Input Arguments

p — Connection to processor on SoC board
xilinxsoc object | intelsoc object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a `xilinxsoc` object or `intelsoc` object.

filename — Path and name of file on SoC board
character vector | string scalar

Path and name of file on the SoC board that you want to delete. You can use either the absolute path or the relative path from the your home directory.

Example: "exampleFile.txt"

Example: "/home/user/exampleFile.txt"

Version History
Introduced in R2022a

2-159

See Also

Objects
xilinxsoc | intelsoc

Functions
putFile | getFile | system | dir | programFPGA

dir

List directory contents on SoC board

Syntax

```
result = dir(p)
result = dir(p,path)
```

Description

`result = dir(p)` lists the directory contents of the home directory of the SoC board connected.

`result = dir(p,path)` lists the directory contents of the specified `path` on the SoC board connected.

Examples

List Directory Contents of SoC Board Folder

List directory contents of a folder on a specified SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Use the `dir` function to list the directory contents of the `Documents` folder located in your home directory.

```
result = dir(p, 'Documents')

result =

  1×3 struct array with fields:

    name
    folder
    isdir
    bytes
```

Input Arguments

p — Connection to processor on SoC board
xilinxsoc object | intelsoc object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a `xilinxsoc` object or `intelsoc` object.

path — Path to a file or folder
character vector | string scalar

Path and name of file on the SoC board that you want the directory content listed for. You can use either the absolute path or the relative path from the home directory.

If no path is specified, your home directory is used.

Example: `"/home/user/Documents"`

Example: `"Documents"`

Output Arguments

`result` — Contents of directory
struct

Returns information on contents of directory specified. If no directory is specified by using the input `path`, the home directory contents are listed.

Version History
Introduced in R2022a

See Also

Objects
xilinxsoc | intelsoc

Functions
putFile | getFile | system | deleteFile | programFPGA

programFPGA

Program FPGA and set corresponding device tree from processor on SoC board

Syntax

```
programFPGA(p,bitFile,dtbFile)
```

Description

programFPGA(p,bitFile,dtbFile) programs the FPGA on your SoC device with the bitstream specified by bitFile and sets the device tree for the processor specified by dtbFile.

Examples

Program FPGA on Xilinx SoC Board

Create a hardware processor object to connect to a Xilinx SoC board. You can use the object to program your FPGA on-board the SoC board at the MATLAB command-line and to create an fpga object that connects to the FPGA through the processor on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Program the FPGA and set the corresponding device tree from processor on the SoC board by using the xilinxsoc object function programFPGA.

```
programFPGA(p, "myBitstream.bit", "myDeviceTree.dtb");
```

For more information on device trees, see "Generate Device Tree for IP Core".

Connect to the FPGA on-board the SoC board by using the fpga function. Use the processor hardware object p as an input to the fpga function.

```
f = fpga(p);
```

Input Arguments

p — Connection to processor on SoC board
xilinxsoc object | intelsoc object

Hardware processor object that represents the connection from MATLAB to the processor on a specific SoC hardware board, specified as a xilinxsoc object or intelsoc object.

bitFile — Bitstream file
character vector | string scalar

Path to the bitstream file used to program the FPGA. This file must be present on the host computer.

Example: `"myBitstream.bit"`

dtbFile — Devicetree blob file
character vector | string scalar

Devicetree blob file used to configure the processor. If the file is present on the host computer at the specified path, it is copied from the host computer to the SoC board. If the file is not present on the host computer, it must be present in the SD card folder `/mnt` on the SoC board.

The device tree must contain nodes corresponding to resources in the bitstream that need to be accessed from the processor. For more information, see "Generate Device Tree for IP Core".

Example: `"myDeviceTree.dtb"`

Version History
Introduced in R2022a

See Also

Objects
`xilinxsoc` | `intelsoc`

Functions
`putFile` | `getFile` | `system` | `deleteFile` | `dir`

Simulink.ModelReference.protect

Obscure referenced model contents to hide intellectual property

Syntax

```
Simulink.ModelReference.protect(model)
Simulink.ModelReference.protect(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.protect(model,'Harness',true)
[~ ,neededVars] = Simulink.ModelReference.protect(model)
```

Description

`Simulink.ModelReference.protect(model)` creates a protected model from the specified `model`. It places the protected model in the current working folder. The protected model has the same name as the source model. It has the extension `.slxp`.

`Simulink.ModelReference.protect(model,Name,Value)` uses additional options specified by one or more name-value pair arguments.

`[harnessHandle] = Simulink.ModelReference.protect(model,'Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.protect(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Protect Referenced Model

Protect a referenced model and place the protected model in the current working folder.

```
openExample('sldemo_mdlref_bus');
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder.

Place Protected Model in Specified Folder

Protect a referenced model and place the protected model in a specified folder.

```
openExample('sldemo_mdlref_bus');
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in `C:\Work`.

Generate Code for Protected Model

Protect a referenced model, generate code for it in normal mode, and obfuscate the code.

```
openExample('sldemo_mdlref_bus');
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Path','C:\Work','Mode','CodeGeneration',...
'ObfuscateCode',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the `C:\Work` folder. The protected model runs as a child of the parent model. The code generated for the protected model is obfuscated by the software.

Generate HDL Code for Protected Model

Protect a referenced model, and generate HDL code for it in normal mode.

```
openExample('hdlcoder/ParentModelWithModelReferenceExample')
parent_model= 'hdlcoder_protected_model_parent_harness';
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','HDLCodeGeneration')
```

A protected model named `hdlcoder_referenced_model_gain.slxp` is created. The protected model file is placed in the same folder as the parent model and the referenced model. The protected model runs as a child of the parent model.

Set the **hdl** option to `true` with **Mode** set to `CodeGeneration` to enable both C code generation and HDL code generation support for a protected model that you create.

```
openExample('hdlcoder/ParentModelWithModelReferenceExample')
parent_model= 'hdlcoder_protected_model_parent_harness';
reference_model_to_protect = 'hdlcoder_referenced_model_gain';

Simulink.ModelReference.protect(reference_model_to_protect, ...
                        'Mode','CodeGeneration','hdl',true)
```

Control Code Visibility for Protected Model

Control code visibility by allowing users to view only binary files and headers in the code generated for a protected model.

```
openExample('sldemo_mdlref_bus');
model= 'sldemo_mdlref_counter_bus'

Simulink.ModelReference.protect(model,'Mode','CodeGeneration','OutputFormat',...
'CompiledBinaries');
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created. The protected model file is placed in the current working folder. Users can view only binary files and headers in the code generated for the protected model.

Create Harness Model for Protected Model

Create a harness model for a protected model and generate an HTML report.

```
openExample('sldemo_mdlref_bus');
modelPath= 'sldemo_mdlref_bus/CounterA'

[harnessHandle] = Simulink.ModelReference.protect(modelPath,'Path','C:\Work',...
'Harness',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter_bus.slxp` is created, along with an untitled harness model. The protected model file is placed in the `C:\Work` folder. The folder also contains an HTML report. The handle of the harness model is returned in `harnessHandle`.

Determine Variables Required by Protected Model

To simulate a model that references a protected model, you might need to define variables in the base workspace or data dictionaries. For example, the `sldemo_mdlref_counter_bus` model needs the variables that specify the buses at the root input and output ports of the model. When you ship a protected model, you must include definitions of the required variables or the model is unusable.

Tip To automatically package required variable definitions with the protected model in a project, set `Project` to `true`.

Generate the protected model and determine the required variables.

```
openExample('sldemo_mdlref_bus');
model= 'sldemo_mdlref_counter_bus'

[~, neededVars] = Simulink.ModelReference.protect(model)
```

The second output, `neededVars`, determines the variables you must send to the recipient. The value of `neededVars` is a cell array that contains the names of the variables required by the protected model. However, the cell array might also contain the names of variables that the model does not need.

Before you share the protected model, edit `neededVars` to delete the names of any variables that the model does not need. Save the required variables in a data dictionary.

Input Arguments

`model` — Model name
character vector | string scalar

Model name, specified as a character vector or string scalar. It contains the name of a model or the path name of a Model block that references the model to be protected.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

File Options

`Project` — Option to collect dependencies in project
`false` (default) | `true`

Option to collect dependencies in project, specified as `true` or `false`.

The protected model, its dependencies, and its harness model are saved in a project archive (`.mlproj`). The project archive provides a way to share a project in a single file. You must open the project archive to create the interactive project.

Note Before sharing the project, check whether the project contains the necessary supporting files. If supporting files are missing, simulating or generating code for the related harness model can help identify them. Add the missing dependencies to the project and update the harness model as needed.

Example: `'Project',true`

Data Types: `logical`

`ProjectName` — Custom project name
character vector | string scalar

Custom project name, specified as a character vector or string scalar.

If you do not specify a custom project name, the default name for the project is the protected model name followed by `_protected`.

Example: `'ProjectName','myname'`

Dependencies

To enable `ProjectName`, set `Project` to `true`.

Data Types: `char` | `string`

`Harness` — Option to create harness model
`false` (default) | `true`

Option to create harness model, specified as a Boolean value.

When you create a harness model for a protected model that relies on base workspace definitions, Simulink creates a MAT-file that contains the base workspace definitions.

The harness model must have access to supporting files, such as a MAT-file with base workspace definitions or a data dictionary.

Example: `'Harness',true`

Dependencies

To set `Harness` to `true`, set `Mode` to a value that supports simulation. For example, set `Mode` to `'Accelerator'`.

Data Types: `logical`

Path — Folder for protected model
current working folder (default) | character vector | string scalar

Folder for protected model, specified as a character vector or string scalar.

Example: `'Path','C:\Work'`

Data Types: `char` | `string`

Functionality Options

Report — Option to generate report
`false` (default) | `true`

Option to generate report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, and interface for the protected model.

Example: `'Report',true`

Data Types: `logical`

Webview — Option to include read-only Web view of protected model
`false` (default) | `true`

Option to include read-only Web view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

- Right-click the protected-model badge icon and select **Show Web view**.
- Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, call:

  `Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');`
- Double-click the `.slxp` protected model file in the Current Folder browser.
- In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

Data Types: `logical`

Mode — Model protection mode
`'Accelerator'` (default) | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode, specified as one of these values:

- `'Accelerator'`: A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.
- `'CodeGeneration'`: A model that references the protected model can run in normal, accelerator, or rapid accelerator mode and can support code generation.
- `'HDLCodeGeneration'`: A model that references the protected model can run in normal, accelerator, or rapid accelerator mode and can support HDL code generation. Requires HDL Coder license.
- `'ViewOnly'`: The protected model supports only read-only view. It does not support simulation or code generation.

For more information about simulation modes in model hierarchies, see "Choose Simulation Modes for Model Hierarchies".

Example: `'Mode','Accelerator'`

CodeInterface — Interface through which generated code is accessed by Model block
`'Model reference'` (default) | `'Top model'`

Interface through which generated code is accessed by a Model block, specified as one of the following values:

- `'Model reference'`: Code access through the model reference code interface, which allows use of the protected model within a model reference hierarchy. Users of the protected model can generate code from a parent model that contains the protected model. In addition, users can run Model block SIL and PIL simulations with the protected model.
- `'Top model'`: Code access through the standalone interface. Users of the protected model can run Model block SIL and PIL simulations with the protected model.

Example: `'CodeInterface','Top model'`

Dependencies

The system target file (`SystemTargetFile`) must be set to an ERT-based system target file, for example, `ert.tlc`). Requires Embedded Coder® license.

hdl — Option to generate HDL code
`false` (default) | `true`

Option to generate HDL code, specified as a Boolean value.

This option requires an HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

Data Types: `logical`

TunableParameters — Tunable parameters for simulation
`'None'` or `{}` (default) | `'All'` | string array | cell array of character vectors

Tunable parameters for simulation, specified as `'None'`, `{}`, `'All'`, a string array, or a cell array of character vectors.

Recipients of the protected model can tune the specified parameters during simulation.

Parameters that are not tunable are not listed as needed variables unless:

- They are used by the model interface.
- They specify part of a data type, such as a bus object, enumerated type, or value type.
- They are used by data store memory.

When you protect a model that references one or more protected models, you must specify the tunable parameters of each referenced protected model as tunable for the model you are protecting.

Example: `'TunableParameters',{'param1','param2'}`

Example: `TunableParameters=["param1","param2"]`

Data Types: `char` | `string` | `cell`

Callbacks — Option to specify callbacks for protected model
cell array

Option to specify callbacks for protected model, specified as a cell array of `Simulink.ProtectedModel.Callback` objects.

Example: `'Callbacks',{pmcallback_sim, pmcallback_cg}`

Data Types: `cell`

Protection Options

ObfuscateCode — Option to obfuscate generated code
`true` (default) | `false`

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation during protection is enabled. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

Data Types: `logical`

OutputFormat — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Protected code visibility, specified as one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. All code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. All code in the build folder is visible. All headers referenced by the code are also visible.

This argument determines what part of the code generated for a protected model is visible to users.

Example: `'OutputFormat','AllReferencedHeaders'`

Dependencies

This argument affects the output only when you specify Mode as `'Accelerator'` or `'CodeGeneration'`. When you specify Mode as `'Normal'`, only a MEX-file is part of the output package.

Encrypt — Option to encrypt protected model
false (default) | true

Option to encrypt protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model: `Simulink.ModelReference.ProtectedModel.setPasswordForView`
- Password for simulation: `Simulink.ModelReference.ProtectedModel.setPasswordForSimulation`
- Password for code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration`
- Password for HDL code generation: `Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration`

Example: `'Encrypt',true`

Data Types: logical

Sign — Option to sign protected model with digital certificate
character vector | string scalar

Option to sign protected model with digital certificate, specified as a character vector or string scalar that specifies the digital certificate. If the certificate file is password-protected, use the `Simulink.ModelReference.ProtectedModel.setPasswordForCertificate` function to provide the password before you use the certificate.

Example: `'Sign','my_certificate.pfx'`

Data Types: char | string

Modifiable — Option to create modifiable protected model
false (default) | true

Option to create modifiable protected model, specified as a Boolean value. To use this option:

- Add a password for modification by using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. If a password has not been added at the time that you create the modifiable protected model, you are prompted to create one.
- Modify the options of your protected model by first providing the modification password using the `Simulink.ModelReference.ProtectedModel.setPasswordForModify` function. Then, use the `Simulink.ModelReference.modifyProtectedModel` function to make your option changes.

Example: `'Modifiable',true`

Data Types: logical

`CustomPostProcessingHook` — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle.

The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. It also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

Output Arguments

`harnessHandle` — Handle of harness model
double

Handle of harness model, returned as a double. When model protection does not create a harness model or packages the harness model in a project archive, the returned value is `0`.

To create a harness model that is open after model protection, set `Harness` to `true` and `Project` to `false` (default).

`neededVars` — Names of base workspace variables
cell array

Names of base workspace variables that the protected model uses, returned as a cell array.

The cell array can also include variables that the protected model does not use.

Alternatives

"Protect Models to Conceal Contents"

Version History
Introduced in R2012b

R2023a: `Mode` will not support normal mode restrictions
Behavior change in future release

In a future release, when you set `Mode` to `'Normal'`, the function will create a protected model that supports simulation when the parent model simulates in normal, accelerator, or rapid accelerator mode. This change will standardize simulation support for new protected models. Protected models created before this change will be unaffected.

Currently, when you set `Mode` to `'Normal'`, the function creates a protected model that supports simulation only when the parent model simulates in normal mode.

R2023a: Default Mode value is 'Accelerator'
Behavior changed in R2023a

Starting in R2023a, the default value of the Mode argument is 'Accelerator'. A model that references the protected model can run in normal, accelerator, or rapid accelerator mode.

With this change, the Create Protected Model dialog box and Simulink.ModelReference.protect function use the same default model protection mode.

Previously, the default value of the Mode argument was 'Normal'. The 'Normal' model protection mode restricts the use of the protected model such that the protected model supports simulation only when the parent model simulates in normal mode.

R2022a: Tunable parameters for simulation are specified by the TunableParameters argument
Behavior changed in R2022a

Starting in R2022a, you must specify the parameters of the protected model that you want to be tunable during simulation by using the TunableParameters name-value argument. By default, no parameters are tunable during simulation.

Previously, a parameter was tunable when its code generation storage class was set to a value other than Auto. By default, the storage class for an individual data element is Auto.

See Also

Simulink.ModelReference.modifyProtectedModel |
Simulink.ModelReference.ProtectedModel.setPasswordForModify |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation |
Simulink.ModelReference.ProtectedModel.setPasswordForView |
Simulink.ModelReference.ProtectedModel.clearPasswords |
Simulink.ModelReference.ProtectedModel.clearPasswordsForModel |
Simulink.ProtectedModel.getTunableParameters

Topics
"Protect Models to Conceal Contents"
"Explore Protected Model Capabilities"
"Test Protected Models"
"Package and Share Protected Models"
"Specify Custom Obfuscators for Protected Models"
"Configure and Run SIL Simulation" (Embedded Coder)
"Code Interfaces for SIL and PIL" (Embedded Coder)
"Define Callbacks for Protected Models"
"Reference Protected Models from Third Parties"

Simulink.ModelReference.modifyProtectedModel

Modify existing protected model

Syntax

```
Simulink.ModelReference.modifyProtectedModel(model)
Simulink.ModelReference.modifyProtectedModel(model,Name,Value)

[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(model,'
Harness',true)
[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(model)
```

Description

`Simulink.ModelReference.modifyProtectedModel(model)` modifies options for an existing protected model created from the specified `model`. If `Name,Value` pair arguments are not specified, the modified protected model is updated with default values and supports only simulation.

`Simulink.ModelReference.modifyProtectedModel(model,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. These options are the same options that are provided by the `Simulink.ModelReference.protect` function. However, these options have additional options to change encryption passwords for read-only view, simulation, and code generation. When you add functionality to the protected model or change encryption passwords, the unprotected model must be available. The software searches for the model on the MATLAB path. If the model is not found, the software reports an error.

`[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(model,'
Harness',true)` creates a harness model for the protected model. It returns the handle of the harnessed model in `harnessHandle`.

`[~ ,neededVars] = Simulink.ModelReference.modifyProtectedModel(model)` returns a cell array that includes the names of base workspace variables used by the protected model.

Examples

Update Protected Model with Default Values

Create a modifiable protected model with support for code generation, then reset it to default values.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

2-175

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Modify the model to use default values.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter');
```

The resulting protected model is updated with default values and supports only simulation.

Remove Functionality from Protected Model

Create a modifiable protected model with support for code generation and Web view, then modify it to remove the Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Webview',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Remove support for Web view from the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter', 'Mode', 'CodeGeneration','Report',true);
```

Change Encryption Password for Code Generation

Change an encryption password for a modifiable protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add the password that the protected model user must provide to generate code.

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','cgpassword');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Encrypt',true,'Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Change the encryption password for simulation.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Encrypt',true,...
'Report',true,'ChangeSimulationPassword',...
{'cgpassword','new_password'});
```

Add Harness Model for Protected Model

Add a harness model for an existing protected model.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with a report and support for code generation with encryption.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration','Modifiable',true,'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Add a harness model for the protected model.

```
[harnessHandle] = Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Report',true,...
'Harness',true);
```

Input Arguments

model — Model name
string or character vector (default)

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Mode','CodeGeneration','OutputFormat','Binaries','ObfuscateCode',true` specifies that obfuscated code be generated for the protected model. It also specifies that only binary files and headers in the generated code be visible to users of the protected model.

General

Path — Folder for protected model
current working folder (default) | string or character vector

Folder for protected model, specified as a string or character vector.

Example: `'Path','C:\Work'`

Report — Option to generate a report
false (default) | true

Option to generate a report, specified as a Boolean value.

To view the report, right-click the protected-model badge icon and select **Display Report**. Or, call the `Simulink.ProtectedModel.open` function with the `report` option.

The report is generated in HTML format. It includes information on the environment, functionality, license requirements, and interface for the protected model.

Example: `'Report',true`

hdl — Option to generate HDL code
false (default) | true

Option to generate HDL code, specified as a Boolean value.

This option requires HDL Coder license. When you enable this option, make sure that you specify the **Mode**. You can set this option to `true` in conjunction with the **Mode** set to `CodeGeneration` to enable both C code and HDL code generation support for the protected model.

If you want to enable only simulation and HDL code generation support, but not C code generation, set **Mode** to `HDLCodeGeneration`. You do not have to set the **hdl** option to `true`.

Example: `'hdl',true`

Harness — Option to create a harness model
false (default) | true

Option to create a harness model, specified as a Boolean value.

Example: `'Harness',true`

CustomPostProcessingHook — Option to add postprocessing function for protected model files
function handle

Option to add a postprocessing function for protected model files, specified as a function handle. The function accepts a `Simulink.ModelReference.ProtectedModel.HookInfo` object as an input variable. This object provides information on the source code files and other files generated during protected model creation. The object also provides information on exported symbols that you must not modify. Prior to packaging the protected model, the postprocessing function is called.

For a protected model with a top model interface, the `Simulink.ModelReference.ProtectedModel.HookInfo` object cannot provide information on exported symbols.

Example: `'CustomPostProcessingHook',@(protectedMdlInf)myHook(protectedMdlInf)`

Functionality

Mode — Model protection mode
`'Normal'` (default) | `'Accelerator'` | `'CodeGeneration'` | `'HDLCodeGeneration'` | `'ViewOnly'`

Model protection mode. Specify one of the following values:

- `'Normal'`: If the top model is running in `'Normal'` mode, the protected model runs as a child of the top model.
- `'Accelerator'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode.
- `'CodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support code generation.
- `'HDLCodeGeneration'`: The top model can run in `'Normal'`, `'Accelerator'`, or `'Rapid Accelerator'` mode and support HDL code generation.
- `'ViewOnly'`: Turns off Simulate and Generate code functionality modes. Turns on the read-only view mode.

Example: `'Mode','Accelerator'`

OutputFormat — Protected code visibility
`'CompiledBinaries'` (default) | `'MinimalCode'` | `'AllReferencedHeaders'`

Note This argument affects the output only when you specify Mode as `'Accelerator'` or `'CodeGeneration`. When you specify Mode as `'Normal'`, only a MEX-file is part of the output package.

Protected code visibility. This argument determines what part of the code generated for a protected model is visible to users. Specify one of the following values:

- `'CompiledBinaries'`: Only binary files and headers are visible.
- `'MinimalCode'`: Includes only the minimal header files required to build the code with the chosen build settings. Code in the build folder is visible. Users can inspect the code in the protected model report and recompile it for their purposes.
- `'AllReferencedHeaders'`: Includes header files found on the include path. Code in the build folder is visible. Header files referenced by the code are also visible.

Example: `'OutputFormat','AllReferencedHeaders'`

ObfuscateCode — Option to obfuscate generated code
true (default) | false

Option to obfuscate generated code, specified as a Boolean value. Applicable only when code generation is enabled for the protected model. Obfuscation is not supported for HDL code generation.

Example: `'ObfuscateCode',true`

Webview — Option to include a Web view
false (default) | true

Option to include a read-only view of protected model, specified as a Boolean value.

To open the Web view of a protected model, use one of the following methods:

* Right-click the protected-model badge icon and select **Show Web view**.
* Use the `Simulink.ProtectedModel.open` function. For example, to display the Web view for protected model `sldemo_mdlref_counter`, you can call:

  `Simulink.ProtectedModel.open('sldemo_mdlref_counter', 'webview');`
* Double-click the `.slxp` protected model file in the Current Folder browser.
* In the Block Parameter dialog box for the protected model, click **Open Model**.

Example: `'Webview',true`

Encryption

ChangeSimulationPassword — Option to change the encryption password for simulation
cell array of two character vectors

Option to change the encryption password for simulation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeSimulationPassword',{'old_password','new_password'}`

ChangeViewPassword — Option to change the encryption password for read-only view
cell array of two character vectors

Option to change the encryption password for read-only view, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeViewPassword',{'old_password','new_password'}`

ChangeCodeGenerationPassword — Option to change the encryption password for code generation
cell array of two character vectors

Option to change the encryption password for code generation, specified as a cell array of two character vectors. The first vector is the old password, the second vector is the new password.

Example: `'ChangeCodeGenerationPassword',{'old_password','new_password'}`

Encrypt — Option to encrypt protected model
false (default) | true

Option to encrypt a protected model, specified as a Boolean value. Applicable when you have specified a password during protection, or by using the following methods:

- Password for read-only view of model:
  Simulink.ModelReference.ProtectedModel.setPasswordForView

- Password for simulation:
  Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

- Password for code generation:
  Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration

- Password for HDL code generation:
  Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration

Example: `'Encrypt',true`

Output Arguments

harnessHandle — Handle of the harness model
double

Handle of the harness model, returned as a double or `0`, depending on the value of `Harness`.

If `Harness` is `true`, the value is the handle of the harness model; otherwise, the value is `0`.

neededVars — Names of base workspace variables
cell array

Names of base workspace variables used by the protected model, returned as a cell array.

The cell array can also include variables that the protected model does not use.

Version History
Introduced in R2014b

See Also
Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForModify

Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration

Add or provide encryption password for HDL code generation from protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(model, password)` adds an encryption password for HDL code generation if you create a protected model. If you use a protected model, the function provides the required password to generate code from the model.

Examples

Create a Protected Model with Encryption for HDL Code Generation

Create a protected model with encryption for HDL code generation.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(...
'hdlcoder_referenced_model_gain','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Mode','HDLCodeGeneration','Encrypt',true,'Report',true);
```

A protected model named `hdlcoder_referenced_model_gain.slxp` is created that requires an encryption password for HDL code generation. To generate HDL code for `sldemo_mdlref_counter`, set floating-point library to `Native Floating-Point` and the Oversampling Factor to 14 in HDL code generation configuration parameters.

Generate HDL Code from an Encrypted Protected Model

Use a protected model with encryption for HDL code generation.

Provide the encryption password required for HDL code generation from the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration(...
'hdlcoder_referenced_model_gain','password');
```

After you have provided the encryption password, you can generate code from the protected model.

Input Arguments

model — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

password — Password for protected model code generation
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for code generation, the password is required.

Version History
Introduced in R2019a

See Also
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForView

Topics
"Create Protected Models to Conceal Contents and Generate HDL Code"

Simulink.ModelReference.ProtectedModel.setPasswordForModify

Add or provide password for modifying protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForModify(model,password)` adds a password for a modifiable protected model. After the password has been created, the function provides the password for modifying the protected model.

Examples

Add Functionality to Protected Model

Create a modifiable protected model with support for code generation, then modify it to add Web view support.

Add the password for when a protected model is modified. If you skip this step, you are prompted to set a password when a modifiable protected model is created.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter','password');
```

Create a modifiable protected model with support for code generation and Web view.

```
Simulink.ModelReference.protect('sldemo_mdlref_counter','Mode',...
'CodeGeneration', 'Modifiable',true, 'Report',true);
```

Provide the password to modify the protected model.

```
Simulink.ModelReference.ProtectedModel.setPasswordForModify(...
'sldemo_mdlref_counter', 'password');
```

Add support for Web view to the protected model that you created.

```
Simulink.ModelReference.modifyProtectedModel(...
'sldemo_mdlref_counter','Mode','CodeGeneration','Webview',true,...
'Report',true);
```

Input Arguments

model — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model to be modified.

password — Password to modify protected model
string or character vector

Password, specified as a string or character vector. The password is required for modification of the protected model.

Version History
Introduced in R2014b

See Also
Simulink.ModelReference.protect | Simulink.ModelReference.modifyProtectedModel

Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Add or provide encryption password for simulation of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,
password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(model,
password)` adds an encryption password for simulation if you create a protected model. If you use a protected model, the function provides the required password to simulate the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for simulation.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for simulation.

Simulate an Encrypted Protected Model

Use a protected model with encryption for simulation.

Provide the encryption password required for simulation of the protected model.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you can simulate the protected model.

Input Arguments

model — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

password — Password for protected model simulation
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for simulation, the password is required.

Version History
Introduced in R2014b

See Also
Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForView

Simulink.ModelReference.ProtectedModel.setPasswordForView

Add or provide encryption password for read-only view of protected model

Syntax

```
Simulink.ModelReference.ProtectedModel.setPasswordForView(model,password)
```

Description

`Simulink.ModelReference.ProtectedModel.setPasswordForView(model,password)` adds an encryption password for read-only view if you create a protected model. If you use a protected model, the function provides the required password for a read-only view of the model.

Examples

Create a Protected Model with Encryption

Create a protected model with encryption for read-only view.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
Simulink.ModelReference.protect('sldemo_mdlref_counter',...
'Webview',true,'Encrypt',true,'Report',true);
```

A protected model named `sldemo_mdlref_counter.slxp` is created that requires an encryption password for read-only view.

View an Encrypted Protected Model

Use a protected model with encryption for read-only view.

Provide the encryption password required for the read-only view of the protected model.

```
openExample('sldemo_mdlref_counter');
Simulink.ModelReference.ProtectedModel.setPasswordForView(...
'sldemo_mdlref_counter','password');
```

After you have provided the encryption password, you have access to the read-only view of the protected model.

Input Arguments

model — Model name
string or character vector

Model name, specified as a string or character vector. It contains the name of a model or the path name of a Model block that references the protected model.

password — Password for read-only view of protected model
string or character vector

Password, specified as a string or character vector. If the protected model is encrypted for read-only view, the password is required.

Version History
Introduced in R2014b

See Also
Simulink.ModelReference.protect |
Simulink.ModelReference.ProtectedModel.setPasswordForCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForHDLCodeGeneration |
Simulink.ModelReference.ProtectedModel.setPasswordForSimulation

Simulink.ModelReference.ProtectedModel.clearPasswords

Clear cached passwords for protected models

Syntax

`Simulink.ModelReference.ProtectedModel.clearPasswords()`

Description

`Simulink.ModelReference.ProtectedModel.clearPasswords()` clears protected model passwords that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for protected models

After using protected models, clear passwords cached for the models during the MATLAB session.

`Simulink.ModelReference.ProtectedModel.clearPasswords()`

Version History
Introduced in R2014b

See Also
`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel`

Topics
"Protect Models to Conceal Contents"

Simulink.ModelReference.ProtectedModel.-clearPasswordsForModel

Clear cached passwords for a protected model

Syntax

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)`

Description

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)` clears protected model passwords for `model` that have been cached during the current MATLAB session. If this function is not called, cached passwords are cleared at the end of a MATLAB session.

Examples

Clear cached passwords for a protected model

After using a protected model, clear passwords cached for the model during the MATLAB session.

`Simulink.ModelReference.ProtectedModel.clearPasswordsForModel(model)`

Input Arguments

model — Protected model name
string or character vector

Model name specified as a string or character vector

Example: 'rtwdemo_counter'

Data Types: char

Version History
Introduced in R2014b

See Also
`Simulink.ModelReference.ProtectedModel.clearPasswords`

Topics
"Protect Models to Conceal Contents"

Simulink.ProtectedModel.open

Open protected model

Syntax

```
Simulink.ProtectedModel.open(model)
Simulink.ProtectedModel.open(model,type)
```

Description

`Simulink.ProtectedModel.open(model)` opens a protected model. If you do not specify how to view the protected model, the software first tries to open the Web view. If the Web view is not enabled for the protected model, the software then tries to open the report. If you did not create a report, the software reports an error.

`Simulink.ProtectedModel.open(model,type)` opens a protected model using the specified viewing method. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report. If the method that you specify is not enabled, the software reports an error. The protected model is not opened.

Examples

Open a Protected Model

Open a protected model without a specified method.

Load the model and save a local copy.

```
openExample('sldemo_mdlref_counter');
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model enabling support for code generation and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Report',true);
```

Open the protected model without specifying how to view it.

```
Simulink.ProtectedModel.open('mdlref_counter')
```

The protected model does not have Web view enabled, so the protected model report is opened.

Open a Protected Model Web View

Open a protected model, specifying the Web view.

Load the model and save a local copy.

```
openExample('sldemo_mdlref_counter');
save_system('sldemo_mdlref_counter','mdlref_counter.slx');
```

Create a protected model with support for code generation, Web view, and reporting.

```
Simulink.ModelReference.protect('mdlref_counter','Mode',...
'CodeGeneration', 'Webview',true,'Report',true);
```

Open the protected model and specify that you want to see the Web view.

```
Simulink.ProtectedModel.open('mdlref_counter','webview')
```

The protected model Web view is opened.

Input Arguments

`model` — Model name
string or character vector

Protected model name, specified as a string or character vector.

`type` — Open method
'webview' | 'report'

Method for viewing the protected model. If you specify 'webview', the software opens the Web view for the protected model. If you specify 'report', the software opens the protected model report.

Version History
Introduced in R2015a

See Also
`Simulink.ModelReference.protect`

sschdladvisor

Open Simscape HDL Workflow Advisor

Syntax

```
sschdladvisor(subsystem)
sschdladvisor(model)
```

Description

sschdladvisor(`subsystem`) opens the Simscape HDL Workflow Advisor for the subsystem within the model.

sschdladvisor(`model`) opens the Simscape HDL Workflow Advisor for the model.

Examples

Open Simscape HDL Workflow Advisor

This example shows how to open advisor for the model and a subsystem inside a model.

Open Simscape HDL Advisor for a Model

For example: To open the advisor for the Boost Converter model, enter:

```
Modelname = 'sschdlexBoostConverterExample';
open_system(Modelname)
sschdladvisor(Modelname)
```

```
### Running Simscape HDL Workflow Advisor for <a href="matlab:(sschdlexBoostConverterExample)">ss
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manager
```
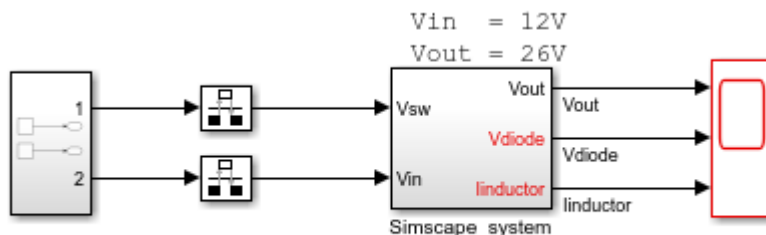

Copyright 2018 The MathWorks, Inc.

Open Simscape HDL Advisor for a Subsystem

For example: To open the advisor for the `Simscape_system` block inside the Buck Converter model, enter:

```
Modelname = 'sschdlexBuckConverterExample';
Subsysname = 'sschdlexBuckConverterExample/Simscape_system';
load_system(Modelname)
open_system(Subsysname)
sschdladvisor(Subsysname)
```

Running Simscape HDL Workflow Advisor for sschdlexBuckConverterExample
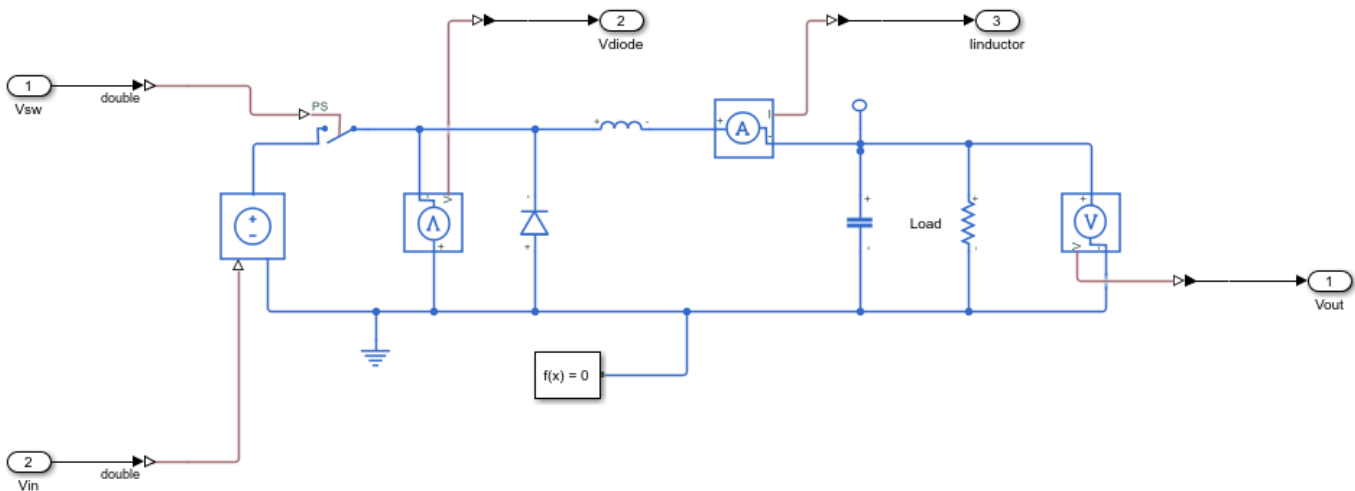
Input Arguments

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

Version History
Introduced in R2018b

See Also
simscape.findNonlinearBlocks

Topics
"Generate HDL Code for Simscape Models"

coder.hdl.arraydistance

Specify minimum or maximum array distance inside pipelined `for`-loop

Syntax

```
coder.hdl.arraydistance(arr_name,'min',arr_distance)
coder.hdl.arraydistance(arr_name,'max',arr_distance)
```

Description

`coder.hdl.arraydistance(arr_name,'min',arr_distance)` enables you to specify the minimum array distance inside a pipelined `for`-loop. The array distance is the number of clock cycles between a memory read and memory write operation in an array. You must insert this pragma at the start of the `for`-loop body.

This pragma does not affect MATLAB simulation behavior.

`coder.hdl.arraydistance(arr_name,'max',arr_distance)` enables you to specify the maximum array distance inside a pipelined `for`-loop. You must insert this pragma at the start of the `for`-loop body.

This pragma does not affect MATLAB simulation behavior.

Examples

Minimum Array Distance in Generated SystemC Code

In the MATLAB function `myMin`, the minimum `arr_distance` inside the pipelined `for`-loop is 1.

```
function out = myMin(in)

    persistent arr1;
    if isempty(arr1)
        arr1 = int8(zeros(1,100));
    end

    coder.hdl.loopspec('pipeline',1);
    for i = 4:100
        coder.hdl.arraydistance('arr1', 'min',1);
        y = arr1(i-3);
        arr1(i) = in;
    end
    out = y;
end
```

Maximum Array Distance in Generated SystemC Code

In the MATLAB function `myMax`, the maximum `arr_distance` inside the pipelined `for`-loop is 2.

```
function out = myMax(in)

    persistent arr1;
    if isempty(arr1)
        arr1 = int8(zeros(1,100));
    end

    coder.hdl.loopspec('pipeline',1);
    for i = 4:100
        coder.hdl.arraydistance('arr1', 'max',2);
        y = arr1(i-3);
        arr1(i) = in;
    end
    out = y;
end
```

Input Arguments

arr_name — Name of the persistent array
character vector

Name of the persistent array mapped to RAM, specified as a character vector.

Example: `'myArr'`

arr_distance — Distance between read and write operations
positive integer

Distance, in terms of the number of clock cycles, between a memory read and a memory write operation in an array, specified as a positive integer.

Example: 7

Version History
Introduced in R2022b

See Also
coder.hdl.loopspec

Topics
"Persistent Variables and Persistent Array Variables"
"Pipelining of for-Loops"

coder.hdl.stable

Define stable inputs in MATLAB design

Syntax

```
coder.hdl.stable(var_name)
```

Description

`coder.hdl.stable(var_name)` enables you to define stable inputs in your MATLAB code and thus facilitate area optimization of the synthesized SystemC™ code. This pragma can be used when an input port holds a stable value during MATLAB simulation. When you define a variable as stable, it is stored in the hardware ports instead of memory (RAM).

You must insert this pragma at the start of the MATLAB design. The specified stable inputs are stored inside the `ml.tcl` file that is read by the Cadence® Stratus importer.

This pragma does not affect MATLAB simulation behavior.

Examples

Define Stable Inputs in MATLAB Code

In the following MATLAB function `myFun`, you declare the input array `in2` stable by using the `coder.hdl.stable` pragma.

```
function out = myFun(in1, in2)
    coder.hdl.stable('in2');
    out = int16(in1);
    for i = 1:100
        out = out + in2;
    end
end
```

Input Arguments

var_name — Input variable name
character vector

Name of the input variable, specified as a character vector.

Example: `'in2'`

Version History
Introduced in R2022b

See Also
coder.hdl.loopspec

coder.hdl.interface

Specify input to be mapped to the line buffer interface in HLS

Syntax

```
coder.hdl.interface(inputVar, 'Line Buffer', imageSize, origin,
'ConstantFill', constantFillValue)
```

Description

`coder.hdl.interface(inputVar, 'Line Buffer', imageSize, origin, 'ConstantFill', constantFillValue)` enables you to map the input variable to the line buffer interface in Cadence Stratus HLS. Various properties of the line buffer interface can be specified as arguments to the pragma.

You must insert this pragma at the start of the MATLAB design. The specified line buffer properties are stored inside the `ml.tcl` file that is read by the Cadence Stratus importer.

This pragma does not affect MATLAB simulation behavior.

Examples

Map Line Buffer Interface in MATLAB Code

In the following MATLAB function `line_buffer_average` takes `in1` as working set and calculates the average of the values in the working set.

```
function out = line_buffer_average(in1)

    coder.hdl.interface(in1, 'Line Buffer', [20, 20], [2, 2], 'ConstantFill', 0);
    sum = 0;
    for i = 1:size(in1,1)
        for j = 1:size(in1,2)
            sum = sum + in1(i,j);
        end
    end
    out = sum / numel(in1);
end
```

The MATLAB test bench demonstrates the usage of the `hdl.WorkingSet` class and `getWorkingSet` method. It generates the working sets for each pixel of the input image.

```
image = rand(20, 20);
ws = hdl.WorkingSet(image, [3 3], [2 2], -1);
for x = 1:20
    for y = 1:20
        workingSet = ws.getWorkingSet(x, y);
        out = line_buffer_average(workingSet);
    end
end
```

> **Note** `hdl.WorkingSet` class and `getWorkingSet` method must be used inside the MATLAB test bench.

Input Arguments

`inputVar` — Working set input variable name
matrix

Working set input variable, specified as a matrix.

`imageSize` — Input image size
2 element row vector

Original image size from which the working sets are to be generated, specified as a 2 element row vector of positive integers.

Example: `[20, 20]`

`origin` — Origin of working set
2 element row vector

Origin of the working set, specified as a 2 element row vector of positive integers.

Example: `[2, 2]`

`constantFillValue` — Constant value to fill outside working set
scalar

Specify the constant value to be filled in the working set for the pixels falling outside the image.

Example: `1`

Version History
Introduced in R2023a

See Also
`hdl.WorkingSet` | `getWorkingSet`

hdl.WorkingSet class

Package: hdl

Working set object that generates working sets form the input image

Description

`ws = hdl.WorkingSet(image, workingSetSize, origin, constantFill)` creates a working set object that you can use to generate working sets from the input image based on the specified properties.

Working set object has the simulation behavior and must be used inside the MATLAB test bench.

Creation

`ws = hdl.WorkingSet(image, workingSetSize, origin, constantFill)` creates an `hdl.WorkingSet` object that you can use to generate the working sets from the input image.

Properties

image — Input image
matrix

Input image, specified as a matrix.

workingSetSize — Size of the working set
2 element row vector

Size of the working set, specified as a 2 element row vector of positive integers.

Example: [3 3]

origin — Origin of working set
2 element row vector

Origin of the working set, specified as a 2 element row vector of positive integers.

Example: [2 2]

constantFill — Constant value to fill outside working set
scalar

The constant value to be filled in the working set for the pixels falling outside the image, specified as a scalar.

Example: 1

Methods

Public Methods
getWorkingSet   Get working set from the input image at a specified position

Version History
Introduced in R2023a

See Also
`coder.hdl.interface`

getWorkingSet

Class: hdl.WorkingSet
Package: hdl

Get working set from the input image at a specified position

Syntax

```
getWorkingSet(x, y)
```

Description

getWorkingSet(x, y) returns a working set corresponding to the pixel at position (x, y) from the input image.

Input Arguments

x — Row subscript
positive integer

Row subscript of the input image, specified as a positive integer.

Example: 7

y — Column subscript
positive integer

Column subscript of the input image, specified as a positive integer.

Example: 9

Examples

Generate Working Set from Input Image

In the following MATLAB code, the line_buffer_average function takes working set as the input and maps it to the line buffer interface in Cadence Stratus HLS using coder.hdl.interface pragma.

```
function out = line_buffer_average(in1)
% in1 is the working set of size 3

    coder.hdl.interface(in1, 'Line Buffer', ...
                [20 20], [2 2], 'ConstantFill', 0);
    sum = 0;
    for i = 1:size(in1,1)
        for j = 1:size(in1,2)
            sum = sum + in1(i,j);
        end
    end
```

```
    out = sum / numel(in1);

end
```

The MATLAB test bench demonstrates the usage of the `hdl.WorkingSet` class and `getWorkingSet` method. It generates the working sets for each pixel of the input image.

```
image = rand(20, 20);
ws = hdl.WorkingSet(image, [3 3], [2 2], -1);
for x = 1:20
    for y = 1:20
        workingSet = ws.getWorkingSet(x, y);
        out = line_buffer_average(workingSet);
    end
end
```

Note `hdl.WorkingSet` class and `getWorkingSet` method must be used inside the MATLAB test bench.

Version History
Introduced in R2023a

See Also
`hdl.WorkingSet` | `coder.hdl.interface`

Blocks

Atan2

Implement control signal based `atan2` function

Description
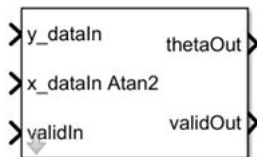
The Atan2 block performs the four-quadrant arctangent of real-valued input signals, `y_dataIn` and `x_dataIn`. The block uses the coordinate rotation digital computer (CORDIC) approximation method to compute the four-quadrant arctangent of the inputs. For more information, see CORDIC approximation method in "Algorithms" on page 3-4. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

`open_system('HDLMathLib')`

Latency Considerations

You can simulate the Atan2 block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on the **Number of iterations**.

The maximum latency L of Atan2 block is given by this equation:

$L = N + 3$

where N is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

- The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

y_dataIn — *y*-coordinate data input signal
scalar | vector

y-coordinate data input signal. The input data must be a real-valued.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

x_dataIn — *x*-coordinate data input signal
scalar | vector

x-coordinate data input signal. The input must be a real-valued.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

thetaOut — Four-quadrant arctangent output signal
scalar | vector

Four quadrant arctangent of *x*- and *y*-coordinate input signals. The output is the angle between a vector from the origin to the point (`x_dataIn`, `y_dataIn`) and the positive *x*-axis.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Number of iterations — Number of iterations for CORDIC algorithm

11 (default) | `Positive integer`

Specify the number of iterations for the CORDIC algorithm.

Programmatic Use
Block Parameter: `iter`
Type: character vector

Values: `Integer values`
Default: `'11'`

Latency strategy — Latency strategy

`Max` (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: `latencyMode`
Type: character vector
Values: `'Max'` | `'Custom'` | `'Zero'`
Default: `'Max'`

Custom latency — Specify the custom latency value

`0` (default)

When you set **Latency strategy** to `Custom`, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range $[0, L]$, where L is the maximum latency value of Atan2 block. For more information, see "CustomLatency".

Dependency

To enable this parameter, set **Latency strategy** to `Custom`.

Programmatic Use
Block Parameter: `customLatencyValue`
Type: Integer
Values: `0 to Max latency`
Default: `0`

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Version History
Introduced in R2020b

See Also
Trigonometric Function | atan2

Bit Concat

Concatenates up to 128 input words into single output
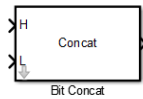
Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bit Concat block concatenates up to 128 input words into a single output. The input port labeled L designates the lowest-order input word. The input port labeled H designates the highest-order input word. For scalar inputs, two or more input signals should be connected to the block. For vector inputs, at least one input should be connected to the block. The block uses the `bitconcat` function such that the right-to-left ordering of words in the output follows the low-to-high ordering of input signals. To learn how the block operates, see "Algorithms" on page 3-7.

Ports

Input

H — Highest-order Input Word
scalar | vector

The highest-order input word for the block to concatenate that becomes the left-most value in the concatenated output signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

L — Lowest-order Input Word
scalar | vector

The lowest-order input word for the block to concatenate that becomes the right-most value in the concatenated output signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

Output — Concatenated Word
scalar | vector

The output is the concatenated word in low-to-high ordering of the input words. The maximum concatenated output word size is 128 bits.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Parameters

Number of Inputs — Number of input signals
2 (default) | positive integer value less than or equal to 128

Enter an integer specifying the number of input signals. The number of block input ports updates when you change **Number of Inputs**.

- Default: 2
- Minimum: 1
- Maximum: 128

Caution Make sure that the **Number of Inputs** is equal to the number of signals you connect to the block. If the block has unconnected inputs, an error occurs during code generation.

Programmatic Use
Block parameter: `numInputs`
Type: string scalar | character vector
Value: A positive integer value less than or equal to 128
Default: `'2'`

Algorithms

The block uses the `bitconcat` function to compute the result. How the block operates depends on the number and dimensions of the inputs, as follows:

- Single input: The input is a scalar or a vector. When the input is a vector, the code generator concatenates the individual vector elements. For example, if the input vector is `[1 2]` that has the data type `ufix4`, the output concatenates the elements 1 and 2 such that 1 forms the MSB (Most Significant Bit). The output is:

  y = dec2bin('00010010') = 18

- Two inputs: Inputs are any combination of scalar and vector.

  - When one input is scalar and the other is a vector, the code generator performs scalar expansion. Each vector element is concatenated with the scalar, and the output has the same dimension as the vector. For example, consider a vector `[1 2]` input to the H port and a scalar value 3 as input to the L port. Both inputs have the data type `ufix4`. The output is a vector that concatenates such that the MSB is a concatenation of elements 1 and 3, and the LSB is a concatenation of elements 2 and 3.

    y = [dec2bin('00010011') dec2bin('00100100')] = [19 35]

  - When both inputs are vectors, they must have the same size. In this case, the last element is the lowest-order word and the first element is the highest order word. For example, consider two input vectors `[1 2]` and `[3 4]` that have the data type `ufix4`. The output is a vector that concatenates such that the MSB is a concatenation of elements 1 and 3, and the LSB is a concatenation of elements 2 and 4.

    y = [dec2bin('00010011') dec2bin('00100100')] = [19 36]

- Three or more inputs (up to a maximum of 128 inputs): Inputs are uniformly scalar or vector. All vector inputs must have the same size. For example, consider three vector inputs `[1 2]`, `[3 4]`,

and [5 6] such that vector [1 2] is input to the H port and [5 6] is input to the L port. In this case, the output is a vector that first concatenates [1 2] and [3 4].

temp = [dec2bin('00010011') dec2bin('00100100')] = [19 36]

The result of this computation is then concatenated with the vector [5 6] to produce the output.

y = [dec2bin('000100110101') dec2bin('001001000110')] = [309 582]

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Blocks
Bit Reduce | Bit Rotate | Bit Shift | Bit Slice

Topics
"Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation"

Bit Reduce

AND, OR, or XOR bit reduction on all input signal bits to single bit
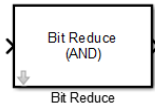
Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bit Reduce block performs a selected bit-reduction operation (AND, OR, or XOR) on all the bits of the input signal, for a single-bit result.

Ports

Input

Port_1 — Input Signal
scalar | vector

Input signal on which the bit-reduction operation is performed.

- Minimum bit width: 2
- Maximum bit width: 128

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

Port_1 — Output Signal
scalar | vector

Output signal that is bit-reduced.

Data Types: `ufix1`

Parameters

Reduction Mode — Reduction operation on input
AND (default) | OR | XOR

Specifies the reduction operation:

- `AND` (default): Perform a bitwise AND reduction of the input signal.
- `OR`: Perform a bitwise OR reduction of the input signal.
- `XOR`: Perform a bitwise XOR reduction of the input signal.

Programmatic Use
Block parameter: mode
Type: string scalar | character vector
Value: "AND" | "OR" | "XOR"
Default: 'AND'

Version History

Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Blocks
Bit Concat | Bit Rotate | Bit Shift | Bit Slice

Bit Rotate

Rotate input signal by bit positions
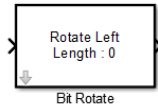
Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bit Rotate block rotates the input signal left or right by the specified number of bit positions.

Ports

Input

Port_1 — Input Signal
scalar | vector

Input signal on which the bit-rotation is performed.

- Minimum bit width: 2
- Maximum bit width: 128

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

Output

Port_1 — Output Signal
scalar | vector

Output signal that is bit-rotated. The output signal has the same data type as the input signal.

Parameters

Rotate Mode — Specifies direction of rotation, left or right
Rotate Left (default) | Rotate Right

Specifies direction of rotation, left or right.

Programmatic Use
Block parameter: mode
Type: string scalar | character vector
Value: "Rotate Left" | "Rotate Right"
Default: 'Rotate Left'

Rotate Length — Specifies the number of bits to rotate
0 (default) | integer greater than or equal to zero

Specifies the number of bits to rotate.

Programmatic Use
Block parameter: N
Type: string scalar | character vector
Value: integer greater than or equal to zero
Default: '0'

Version History

Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Blocks
Bit Concat | Bit Reduce | Bit Shift | Bit Slice

Bit Shift

Logical or arithmetic shift of input signal

Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bit Shift block performs a logical or arithmetic shift on the input signal.

This block is different from the Shift Arithmetic block in terms of simulation and HDL code generation behavior. The Bit Shift block can perform logical shifting of a signed number without having to perform a `reinterpretcast` operation. This block uses a MATLAB Function block based implementation and might be slower in operation.

The Shift Arithmetic block shifts the bits or binary point of the input number. This block has additional block options for HDL code generation in comparison to the Bit Shift block. When you want to perform a variable shift operation, use the Shift Arithmetic block instead of the Bit Shift block. If you have a signed number as input, the block performs a sign extension of the number. The Shift Arithmetic block requires using additional Data Type Conversion blocks that have the `Stored Integer (SI)` option selected.

Ports

Input

Port_1 — Input Signal
scalar | vector

Input signal on which the logical or arithmetic shift is performed.

- Minimum bit width: 2
- Maximum bit width: 128

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

Port_1 — Output Signal
scalar | vector

Output signal that is logically or arithmetically shifted. The output signal has the same data type and bit width as the input signal.

- Minimum bit width: 2

- Maximum bit width: 128

Parameters

Shift Mode — Type and direction of shift
Shift Left Logical (default) | Shift Right Logical | Shift Right Arithmetic

Specifies the type and direction of shift.

Programmatic Use
Block parameter: mode
Type: string scalar | character vector
Value: "Shift Left Logical" | "Shift Right Logical" | "Shift Right Arithmetic"
Default: 'Shift Left Logical'

Shift Length — Number of bits to be shifted
0 (default)

Specifies the number of bits to be shifted. Specify a value greater than or equal to zero.

Programmatic Use
Block parameter: N
Type: string scalar | character vector
Value: A value greater than or equal to zero
Default: '0'

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".

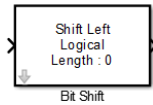| General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Blocks
Bit Concat | Bit Reduce | Bit Rotate | Bit Slice

Bit Slice

Return field of consecutive bits from input signal

Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bit Slice block returns a field of consecutive bits from the input signal. Specify the lower and upper boundaries of the bit field by using zero-based indices in the **LSB Position** and **MSB Position** parameters. For an input word size WS, **LSB Position** and **MSB Position** must satisfy these constraints:

```
WS > MSB Position >= LSB Position >= 0
```

Ports

Input

Port_1 — Input Signal
scalar | vector

Input signal on which the bit-slice is performed. The input signal has a maximum bit width of 128.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Output

Port_1 — Output Signal
scalar | vector

Output signal that is bit-sliced. The word length of the output is computed as (`MSB Position - LSB Position`) `+ 1`, where **MSB Position** and **LSB Position** are block parameters.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Parameters

MSB Position — Most significant bit position
7 (default)

Specifies the bit position (zero-based) of the most significant bit (MSB) of the field to extract.

For an input word size WS,**MSB Position** must satisfy these constraints:

```
WS > MSB Position >= LSB Position >= 0
```

Programmatic Use
Block parameter: `lidk`
Type: string scalar | character vector
Value: positive integer value such that satisfies the constraints mentioned previously
Default: `'7'`

LSB Position — Least significant bit position
0 (default)

Specifies the bit position (zero-based) of the least significant bit (LSB) of the field to extract.

For an input word size WS, **LSB Position** must satisfy these constraints:

```
WS > MSB Position >= LSB Position >= 0
```

Programmatic Use
Block parameter: `ridk`
Type: string scalar | character vector
Value: positive integer value such that satisfies the constraints mentioned previously
Default: `'0'`

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".

General	
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Blocks
Bit Concat | Bit Reduce | Bit Rotate | Bit Shift

Topics
"Create Hardware Design Patterns Using the MATLAB Function Block For HDL Code Generation"

Bits to Word

Convert vector of bits to integer

Libraries:
HDL Coder / Logic and Bit Operations

Description

The Bits to Word block converts a length-*N* input vector of bits to an *N*-bit integer. The output of the block is an unsigned integer that has word length *N*.

The block treats the first element of the input vector as the least significant bit (LSB) of the output and treats subsequent bits as the next significant bits in ascending order. This figure shows the conversion of the bit vector `[1 0 1 1 0]` to the integer value 13.
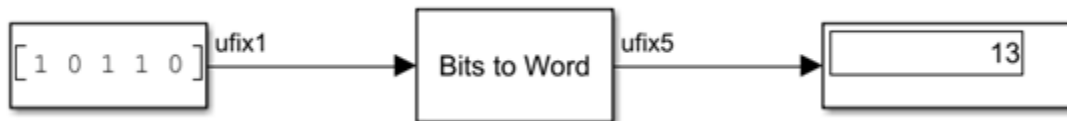
Limitations

The block has these limitations:

- The input signal must have the `ufix1` data type.
- Matrix input is not supported.
- Complex input is not supported.
- The input vector length cannot exceed 128.

Ports

Input

In — Input signal
binary-valued scalar | binary-valued vector

Input signal to the block, specified as binary-valued scalar or vector. You can specify an vector containing up to 128 elements.

Example: `[0 1 1 0 0 1 1 1]`

Data Types: `fixdt(0,1,0)`

Output

Out — Output signal
scalar

N-bit integer, where *N* is the size of the input vector, returned as a nonnegative scalar. For example, when you specify a 1-by-8 input signal, then the output signal is an unsigned integer with the `ufix8` data type.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Version History
Introduced in R2023a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also
Word to Bits

Cos

Implement control signal based cosine function

Description
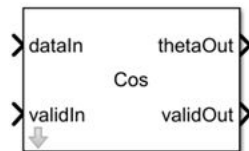
The Cos block computes the cosine of input signal. The block uses the coordinate rotation digital computer (CORDIC) approximation method. For more information, see CORDIC approximation method in "Algorithms" on page 3-23. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the Cos block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on the **Number of iterations**.

The maximum latency L of Cos block is given by this equation:

$L = N + 1$

where N is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

• The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dataIn — Input data signal
scalar | vector

Input data signal to compute cosine function. The input value ranges from -2π to 2*π.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

thetaOut — Output data signal
scalar | vector

Output data signal that is the cosine of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Number of iterations — Number of iterations for CORDIC algorithm

11 (default) | `Positive integer`

Specify the number of iterations for CORDIC algorithm.

Programmatic Use
Block Parameter: `iter`
Type: character vector
Values: `Integer values`
Default: `'11'`

Latency strategy — Latency strategy

Max (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: `latencyMode`
Type: character vector
Values: `'Max'` | `'Custom'` | `'Zero'`
Default: `'Max'`

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to `Custom`, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, *L*], where *L* is the maximum latency value of Cos block. For more information, see "CustomLatency".

Dependency

To enable this parameter, set **Latency strategy** to `Custom`.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: `0 to Max latency`
Default: `0`

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Version History
Introduced in R2020b

See Also
Trigonometric Function | Sine, Cosine

Cos+jSin

Implement control signal based `Cos+jSin` function

Description
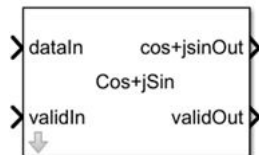
The Cos+jSin block computes the complex exponential of the input signal. The block uses the coordinate rotation digital computer (CORDIC) approximation method. For more information, see CORDIC approximation method in "Algorithms" on page 3-26. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the Cos+jSin block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on the **Number of iterations**.

The maximum latency L of Cos+jSin block is given by this equation:

$L = N + 1$

where N is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇓ icon on the block.

Limitations

- The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dataIn — Input data signal
scalar | vector

Input data signal to compute `Cos+jSin` function. The input value ranges from -2π to 2*π.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

cos+jsinOut — Output data signal
scalar | vector

Output data signal that is the `Cos+jSin` of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Number of iterations — Number of iterations for CORDIC algorithm

11 (default) | `Positive integer`

Specify the number of iterations for CORDIC algorithm.

Programmatic Use
Block Parameter: `iter`
Type: character vector
Values: `Integer values`
Default: `'11'`

Latency strategy — Latency strategy

Max (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: `latencyMode`
Type: character vector
Values: `'Max'` | `'Custom'` | `'Zero'`
Default: `'Max'`

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to `Custom`, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, L], where L is the maximum latency value of `Cos+jSin` block. For more information, see "CustomLatency".

Dependency

To enable this parameter, set **Latency strategy** to `Custom`.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: `0 to Max latency`
Default: `0`

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Version History
Introduced in R2020b

See Also
Trigonometric Function | Sine, Cosine

Sine HDL Optimized and Cosine HDL Optimized

Implement fixed-point sine and cosine wave by using lookup table approach optimized for HDL code generation
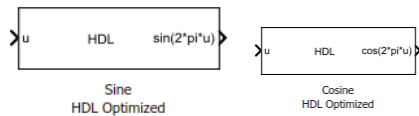
Libraries:
HDL Coder / Lookup Tables

Description

The Sine HDL Optimized block and Cosine HDL Optimized block implements a fixed-point sine or cosine wave by using a lookup table method that exploits quarter-wave symmetry.

For the most efficient HDL implementation, configure the block with an exact power of two as the number of elements. In the Block Parameters dialog box, for **Number of data points**, specify an integer that is an exact power of two. That is, specify the lookup table data points to be (2^n), where *n* is an integer. By default, the **Number of data points** is 64.

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

Depending on your selection of the **Output formula** parameter, the blocks can output these functions of the input signal:

- $\sin(2\pi u)$
- $\cos(2\pi u)$
- $\exp(i2\pi u)$
- $\sin(2\pi u)$ and $\cos(2\pi u)$

Use the **Table data type** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.

Ports

Input

Input — Input signal to implement sine or cosine wave
real-valued signal

Input signal, u, specified as a real-valued scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

sin(2*pi*u) — Fixed-point sine wave
real-valued fixed-point signal

Fixed-point sine wave, implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to `sin(2*pi*u)` or `sin(2*pi*u) and cos(2*pi*u)`.

Data Types: `fixed point`

cos(2*pi*u) — Fixed-point cosine wave
real-valued fixed-point signal

Fixed-point cosine wave, implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to `cos(2*pi*u)` or `sin(2*pi*u) and cos(2*pi*u)`.

Data Types: `fixed point`

exp(j*2*pi*u) — Complex exponential
complex-valued fixed-point signal

Complex exponential, implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to `exp(j*2*pi*u)`.

Data Types: `fixed point`

Parameters

Output formula — Select the signal(s) to output

`cos(2*pi*u)` | `sin(2*pi*u)` | `exp(j*2*pi*u)` | `sin(2*pi*u) and cos(2*pi*u)`

Select the signal(s) to output.

Number of data points — Specify the number of data points to retrieve from the lookup table

`64` (default) | `integer, greater than or equal to 2`

Specify the number of data points to retrieve from the lookup table. The implementation is most efficient when you specify the lookup table data points to be (`2^n`), where n is an integer.

Table data type — Specify the table data type

`fixdt(0,16,15)` (default) | `fixed point expression`

Specify the table data type. You can specify an expression that evaluates to a data type, for example, `fixdt(1,16,0)`.

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the table data type.

Data type assistant — Select the mode of data type specification

Fixed point (default) | Expression

Select the mode of data type specification. If you select Expression, enter an expression that evaluates to a data type, for example, fixdt(1,16,0).

If you select Fixed point, you can use the options in the **Data Type Assistant** to specify the fixed-point data type. In the Fixed point mode, you can choose binary point scaling, and specify the signedness, word length, fraction length, and the data type override setting.

Simulate RAM Delay — Insert a unit delay
off (default) | on

Selecting this check box inserts a unit delay without reset. You can simulate this delay in the Simulink modeling environment. When you generate HDL code, a no-reset register is inserted after the block. The combination of lookup table with no-reset register maps to RAM on the target hardware.

Version History
Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

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

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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
MapToRAM	Map lookup tables (LUTs) to RAM. The default is on. See also "MapToRAM".

General	
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

Limitations and Considerations

When the blocks have Delay block at the output and **Number of data points** is greater than 1024, the blocks does not map to the Block RAM if the target frequency is greater than 400 MHz.

See Also

Blocks
Sine, Cosine | Trigonometric Function

Deserializer1D

Convert scalar stream or smaller vectors to vector signal
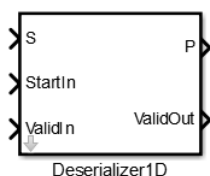
Libraries:
HDL Coder / HDL Operations

Description

The Deserializer1D block buffers a faster, scalar stream or vector signals into a larger, slower vector signal. The faster input signal is converted to a slower signal based on the **Ratio** and **Idle Cycle** values, the conversion changes sample time. Also, the output signal is delayed one slow signal cycle because the serialized data needs to be collected before it can be output as a vector. See the examples below for more details.

You can configure the deserialization to depend on a valid input signal ValidIn and a start signal StartIn. If the **ValidIn** and **StartIn** block parameters are both selected, data collection starts only if both ValidIn and StartIn signals are true. Consider this example:
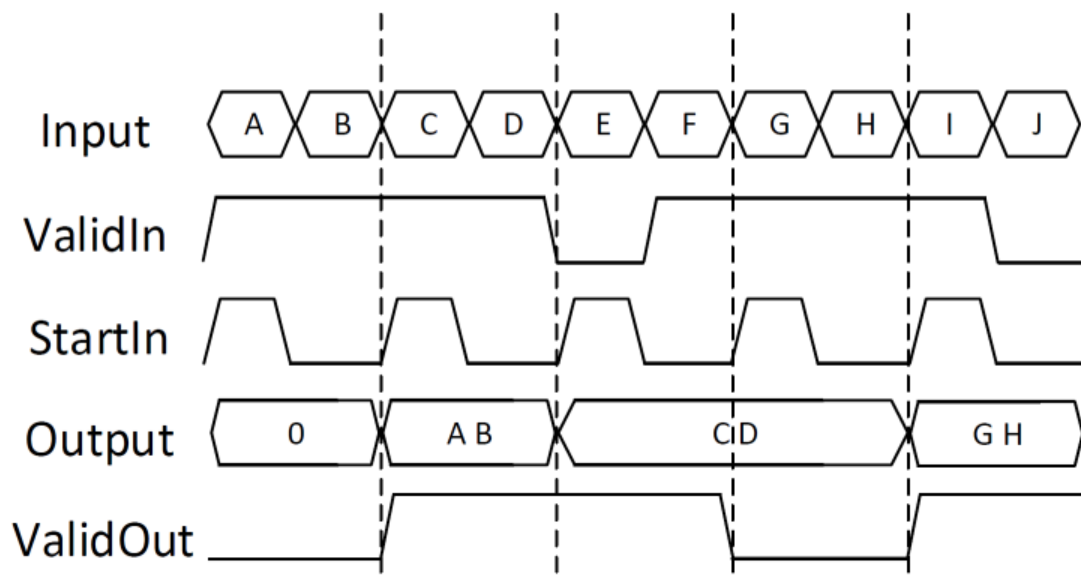
- **Ratio** is 2 and **Idle Cycles** is 0, so each output cycle is two input signals long with all data points considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection begins for A and B. The block outputs the deserialized vector in the next valid cycle, so the AB vector is output in the next cycle. This is also true in the second cycle for C and D.

In the third cycle, starting at E, StartIn is true, but ValidIn is not. E is dropped. At F, ValidIn is true, but StartIn is not, so F is also dropped. Since it cannot collect data for E or F, Deserializer1D outputs the previous cycle vector, CD, but ValidOut changes to false.

Another scenario to consider is when the StartIn signal arrives too early. If the length between two StartIn signals is not long enough to collect a full ratio cycle, the insufficient signal data is dropped. Consider this example:
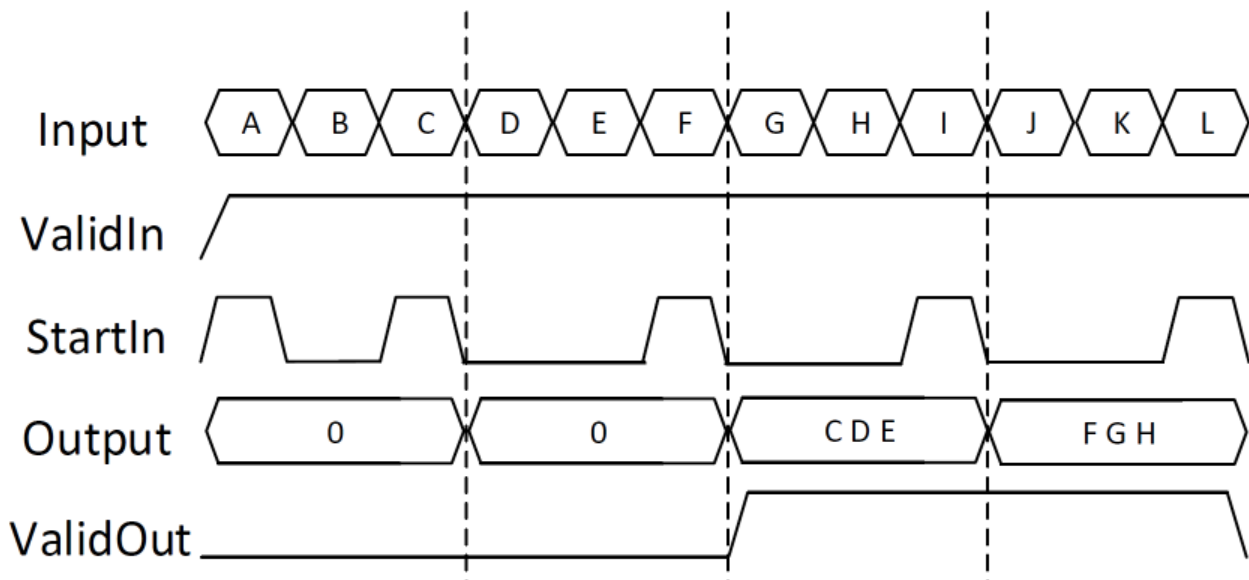
- **Ratio** is 3, so each cycle is two sections long.
- **Idle Cycles** is 0, so all data inputs are considered.
- **ValidIn** and **StartIn** are selected, so data collection can begin only when both StartIn and ValidIn signals are true.
- **ValidOut** is selected.

In the first cycle, ValidIn and StartIn are true, so data collection can begin for A and B. However, at C another StartIn signal arrives before three signals can be collected. Because the StartIn arrived early, A and B are dropped and no valid vector is collected during the first cycle. Therefore, the output of the second cycle is still zero. Deserialization begins at the StartIn at C, for C, D, and E. This vector is output at the next valid cycle, which is cycle 3. Similarly, deserialization starts again at the StartIn at F, and outputs the FGH vector in the fourth cycle.

You specify the block output for the first sampling period with the value of the **Initial condition** parameter.

Ports

Input

S — Serial input data signal
scalar

Input signal to deserialize. Bus data types are not supported.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated`

ValidIn — Valid input control signal
scalar

Indicates valid input signal. Use with the Serializer1D block. This port is available when you select the **ValidIn** check box.

Data Types: `Boolean`

StartIn — Start In control signal
scalar

Input control signal indicates when to start deserialization. Use with the Serializer1D block. Data collection begins at the rising edge of the StartIn signal.

This port is available when you select the **StartIn** check box.

Data Types: `Boolean`

Output

P — Deserialized (Parallel) data output signal
scalar | vector

Deserialized output signal. Bus data types are not supported.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated`

ValidOut — Valid out control signal
scalar

Indicates valid output signal. This port is available when you select the **ValidOut** check box.

Data Types: `Boolean`

Parameters

Ratio — Ratio of output vector size to input vector size
1 (default)

Enter the deserialization ratio, specified as positive scalar

The ratio is the output vector size, divided by the input vector size. The ratio must be divisible by the input vector size.

Idle Cycles — Number of idle cycles
0 (default)

Enter the number of idle cycles added to the end of each serialized input.

The value of **Idle Cycles** affects the deserialized output rate. For example, if **Ratio** is 2 and the input signal is A, B, B, C, D, D, ..., without idle cycles the output would be AB, BC, DD.... However for the same input and ratio with **Idle Cycles** set to 1, the output is AB, CD.... The idle cycles, B and D, are dropped.

The Deserializer1D behavior changes if **Idle Cycles** is not zero, and **ValidIn** or **StartIn** are on. The idle cycles value affects only the output rate, while **ValidIn** and **StartIn** control what input data is deserialized.

Initial condition — Initial output data signal value
0 (default)

Specify the initial output of the simulation.

StartIn — Enable StartIn control port
off (default) | on

Select to activate the StartIn port.

ValidIn — Enable ValidIn control port
off (default) | on

Select to activate the ValidIn port.

ValidOut — Enable ValidOut control port
off (default) | on

Select to activate ValidOut port.

Input data port dimensions (-1 for inherited) — Size of the input data signal
0 (default)

Enter the size of the input data signal. The input size must be divisible by the ratio plus the number of idle cycles. By default, the block inherits size based on context within the model.

Input sample time (-1 for inherited) — Input sample time
-1 (default)

Enter the time interval between sample time hits or specify another appropriate sample time such as continuous. By default, the block inherits its sample time based on context within the model. For more information, see "Sample Time".

Input signal type — Input signal type
auto (default) | real | complex

Specify the input signal type of the block as auto, real, or complex.

Version History
Introduced in R2014b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

Note For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also
Serializer1D

Divide

Implement control signal based division operation

Description

The Divide block performs the division operation on the input data signals. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the Divide block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on input and output data type. The latency of the block is calculated from word-length and fractional-length of fixed-point input and output.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

*   The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dividendDataIn — Dividend input signal
scalar | vector

Dividend input to calculate the division.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

divisorDataIn — Divisor input signal
scalar | vector

Divisor input to calculate the division.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

dataOut — Output data signal
scalar | vector

Output signal that is the division of the dividend and divisor input signals.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Latency strategy — Select the latency strategy

`Max` (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: `latencyMode`
Type: character vector
Values: `'Max'` | `'Custom'` | `'Zero'`
Default: `'Max'`

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to `Custom`, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, L], where L is the maximum latency value of Divide block. For more information, see "CustomLatency".

Dependency

To use this parameter, set **Latency strategy** to `Custom`.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: 0 to Max latency
Default: 0

Output data type — Select the output data type of the block

Inherit: Inherit via internal rule (default) | Inherit: Same as first input |
Inherit: Inherit via back propagation | Inherit: Keep MSB | Inherit: Match
Scaling | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) |
fixdt(1,16,0) | <data type expression>

Specify the output data type. The data type can be inherited or specified directly.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back
propagation' | 'Inherit: Same as first input' | 'Inherit: Keep MSB' | 'Inherit:
Match Scaling' | 'int8' | 'uint8' | int16 | 'uint16' | 'int32' | 'uint32' | 'int64' |
'uint64' | fixdt(1,16,0) | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional
configuration options that affect HDL implementation and synthesized logic. For more information,
see HDL Block Properties.

Version History
Introduced in R2020b

See Also
Math Function | Divide

Dual Port RAM

Dual port RAM with two output ports
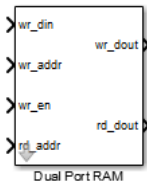
Libraries:
HDL Coder / HDL RAMs

Description

The Dual Port RAM block models a RAM that supports simultaneous read and write operations, and has both a read data output port and write data output port. You can use this block to generate HDL code that maps to RAM in most FPGAs.

If you do not need to use the write output data, `wr_dout`, you can achieve better RAM inference with synthesis tools by using the Simple Dual Port RAM block.

Read-During-Write Behavior

During a write, new data appears at the output of the write port (`wr_dout`) of the Dual Port RAM block. If a read operation occurs simultaneously at the same address as a write operation, old data appears at the read output port (`rd_dout`).

Ports

Input

wr_din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when `wr_en` is true. The data inherits the width and data type from the input signal. `wr_din` can be a `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data type: scalar fixed point, integer, or complex

Data Types: `int8` | `int16` | `int32` | `int64` | `Boolean` | `fixed point`

wr_addr — Write address
Scalar (default) | Vector

Address that you write the data into when `wr_en` is true. This value can be either `fixed-point(fi)` or `integer`, must be unsigned, and have a fraction length of 0.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

wr_en — Write enable
Scalar (default) | Vector

When wr_en is true, the RAM writes the data into the memory location that you specify.

Data Types: Boolean

rd_addr — Read address
Scalar (default) | Vector

Address that you read the data from. This value can be either fixed-point(fi) or integer, must be unsigned, and have a fraction length of 0.

Data Types: uint8 | uint16 | uint32 | uint64 | fixed point

Output

wr_dout — Output data from write address
Scalar (default) | Vector

Output data from write address, wr_addr.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

rd_dout — Output data from read address
Scalar (default) | Vector

Output data from read address, rd_addr.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Parameters

Address port width — Address bit width
8 (default)

Minimum bit width is 2, and maximum bit width is 29.

Programmatic Use
Block parameter: ram_size
Type: string scalar | character vector
Value: A minimum value of 2 and maximum value of 29
Default: '8'

Algorithms

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 3-41.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map the RAM blocks in your design to RAM blocks on the target FPGA. You cannot map Dual Port RAM blocks to UltraRAM. Mapping to UltraRAM requires the block to have a fixed read behavior. For more details, see "RAMDirective".

Complex Data Support

This block supports code generation for complex signals.

See Also

Blocks
Simple Dual Port RAM | Dual Rate Dual Port RAM | Single Port RAM

Dual Port RAM System

Dual Port RAM block based on the hdl.RAM system object with ability to provide initial value

Dual Port RAM System

Libraries:
HDL Coder / HDL RAMs

Description

The blocks are MATLAB System blocks that use the `hdl.RAM` System object™. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Dual Port RAM System block behaves similar to the Dual Port RAM, the Single Port RAM System behaves similar to the Single Port RAM, and so on. With the MATLAB System blocks, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.

- Obtain faster simulation results when you use these blocks in your Simulink model.

- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.

- Obtain higher performance and support for large data memories.

Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.

- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

- The RAM write address can be either `fixed-point (fi)` or `integer`, must be unsigned, and must be between 2 and 31 bits long.

Ports

Input

din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

wr_addr — Write address
Scalar (default) | Vector

RAM address that you write the data into. This value can be either `fixed-point (fi)` or `integer`, must be unsigned, and must be between 2 and 31 bits long.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

wr_en — Write enable
Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location `addr` when `wrEn` is false.

Data Types: `Boolean`

rd_addr — Read address
Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

Output

rd_dout — Read data
Scalar (default) | Vector

Old output data that the RAM reads from the memory location `rd_addr`.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

wr_dout — Write data output
Scalar (default) | Vector

New or old output data that the RAM reads from the memory location `wr_addr`.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Dual port`.

Parameters

Specify the type of RAM — RAM type
Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- Single port — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- Simple dual port — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- Dual port — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

Specify the output data for a write operation — Write output behavior
New data (default) | Old data

Behavior for Write output, specified as either:

- 'New data' — Send out new data at the address to the output.
- Old data' — Send out old data at the address to the output.

Specify the RAM initial value — Initial simulation output of RAM
'0.0' (default) | Scalar | Vector

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Architecture

The block has a MATLABSystem architecture which indicates that the block implementation uses the hdl.RAM System object.

HDL Block Properties

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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to the RAM blocks on the target FPGA. For UltraRAM mapping, **Specify the RAM initial value** must be set to 0. For more details, see "RAMDirective".

Complex Data Support
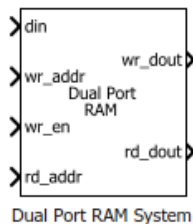
This block supports code generation for complex signals.

See Also

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"

Dual Rate Dual Port RAM

Dual Port RAM that supports two rates
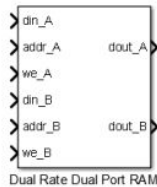
Libraries:
HDL Coder / HDL RAMs

Description

The Dual Rate Dual Port RAM block models a RAM that supports simultaneous read and write operations to different addresses at two clock rates. Port A of the RAM can run at one rate, and port B can run at a different rate.

In high-performance hardware applications, you can use this block to access the RAM twice per clock cycle. If you generate HDL code, this block maps to a dual-clock dual-port RAM in most FPGAs.

Simultaneous Access

You can access different addresses from ports A and B simultaneously. You can also read the same address from ports A and B simultaneously.

However, do not access an address from one RAM port while it is being written from the other RAM port. During simulation, if you access an address from one RAM port at the same time as you write that address from the other RAM port, the software reports an error.

Read-During-Write Behavior

The RAM has write-first behavior. When you write to the RAM, the new write data is immediately available at the output port.

Ports

Input

din_A — Port A write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when we_A is true. The data inherits the width and data type from the input signal. din_A can be a `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data type: scalar fixed point, integer, or complex

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

addr_A — Port A write address
Scalar (default) | Vector

Address that you write the data into when we_A is true. This value can be either `fixed-point(fi)` or `integer`, must be unsigned, and have a fraction length of 0.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

we_A — Port A write enable
Scalar (default) | Vector

Write enable for RAM port A. Set we_A to `true` for a write operation, or `false` for a read operation.

Data Types: `Boolean`

din_B — Port B write data
Scalar (default) | Vector

Data that you write into the RAM memory location when we_B is true. The data inherits the width and data type from the input signal. din_B can be a `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data type: scalar fixed point, integer, or complex

Data Types: `int8` | `int16` | `int32` | `int64` | `fixed point`

addr_B — Port B write address
Scalar (default) | Vector

Address that you write the data into when we_B is true. This value can be either `fixed-point(fi)` or `integer`, must be unsigned, and have a fraction length of 0.

Data type: scalar fixed point, integer, or complex

Data Types: `int8` | `int16` | `int32` | `int64` | `fixed point`

we_B — Port B write enable
Scalar (default) | Vector

Write enable for RAM port B. Set we_B to `true` for a write operation, or `false` for a read operation.

Data Types: `Boolean`

Output

dout_A — Port A output data
Scalar (default) | Vector

Output data from RAM port A address, `addr_A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

dout_B — Port B output data
Scalar (default) | Vector

Output data from RAM port B address, `addr_B`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Address port width — Address bit width
8 (default)

Minimum bit width is 2, and maximum bit width is 28.

Programmatic Use
Block parameter: `ram_size`
Type: string scalar | character vector
Value: A minimum value of 2 and maximum value of 28
Default: `'8'`

Algorithms

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 3-41.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

Note For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

Note The **RAMDirective** property is not available for use with the Dual Rate Dual Port RAM because the block does not have a single clock interface.

Complex Data Support

This block supports code generation for complex signals. You cannot have mixed real-value and complex inputs.

See Also

Blocks
Simple Dual Port RAM | Single Port RAM | Dual Port RAM

Enabled Synchronous Subsystem

Represent enabled subsystem that has synchronous reset and enable behavior

Libraries:
HDL Coder / HDL Subsystems

Description

An Enabled Synchronous Subsystem is an Enabled Subsystem that uses the `Synchronous` mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create an Enabled Synchronous Subsystem block, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to `Synchronous` inside an Enabled subsystem.

For more information, see State Control and "Using Enabled Subsystems".

Ports

Input

Port_1 — Input Signal to subsystem
scalar | vector

See Inport for information on the data types accepted by a subsystem's input ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

Port_1 — Output Signal
scalar | vector

See Outport for information on the data types output by a subsystem's output ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Show port labels — Display options for port labels

`FromPortIcon` (default) | `FromPortBlockName` | `SignalName` | `none`

Select how to display port labels on the Synchronous Subsystem block icon.

> Do not display port labels.

FromPortIcon

> If the corresponding port icon displays a signal name, display the signal name on the Subsystem block. Otherwise, display the port block name or the port number if the block name is a default name.

FromPortBlockName

> Display the name of the corresponding port block on the Subsystem block.

SignalName

> If the signal connected to the port is named, display the name of the signal on the Subsystem block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: ShowPortLabels
Type: character vector
Value: 'FromPortIcon' | 'FromPortBlockName' | 'SignalName' | 'none'
Default: 'FromPortIcon'

Read/Write permissions — Levels of access to contents of subsystem

ReadWrite (default) | ReadOnly | NoReadOrWrite

Control user access to the contents of the subsystem.

Settings

Default: ReadWrite

ReadWrite

> Enables opening and modification of subsystem contents.

ReadOnly

> Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

NoReadOrWrite

> Disables opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

Programmatic Use
Parameter: Permissions
Type: character vector
Value: 'ReadWrite' | 'ReadOnly' | 'NoReadOrWrite'
Default: 'ReadWrite'

Name of error callback function — Name of function to be called if error occurs

' ' (default) | function name

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

Programmatic Use
Parameter: `ErrorFcn`
Type: character vector
Value: `''` | `'<function name>'`
Default: `''`

Permit hierarchical resolution — Resolution for workspace variable names

`All` (default) | `ExplicitOnly` | `None`

Select whether to resolve names of workspace variables referenced by this subsystem.

For more information, see "Symbol Resolution" and "Symbol Resolution Process".

`All`

    Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

`ExplicitOnly`

    Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as "must resolve".

`None`

    Do not resolve workspace variable names.

Programmatic Use
Parameter: `PermitHierarchicalResolution`
Type: character vector
Value: `'All'` | `'ExplicitOnly'` | `'None'`
Default: `'All'`

Function packaging — Code format

`Auto` (default) | `Inline` | `Nonreusable function` | `Reusable function`

Select the code format to be generated for an atomic (nonvirtual) subsystem.

`Auto`

    Simulink Coder chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

`Inline`

    Simulink Coder inlines the subsystem unconditionally.

`Nonreusable function`

    Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

`Reusable function`

> Simulink Coder software generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

> This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

Programmatic Use
Parameter: RTWSystemCode
Type: character vector
Value: `'Auto'` | `'Inline'` | `'Nonreusable function'` | `'Reusable function'`
Default: `'Auto'`

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

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`. For more details, see "ConstrainedOutputPipeline".
DistributedPipelining	Pipeline register distribution, or register retiming. The default is `inherit`. 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`. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "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".

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.

Target Specification	
AdditionalTargetInterfaces	Additional target interfaces, specified as a character vector. To save this block property on the model, in the **Set Target Interface** task of the **IP Core Generation** workflow, corresponding to the DUT ports that you want to add more interfaces, select **Add more...**. You can then add more interfaces in the Add New Target Interfaces dialog box. Specify the type of interface, number of additional interfaces, and a unique name for each additional interface. Values: `''` (default) | cell array of character vectors Example: `'{{'AXI4-Stream','InterfaceID','AXI4-Stream1'}}'`
ProcessorFPGASynchronization	Processor/FPGA synchronization mode, specified as a character vector. To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `Free running` (default) | `Coprocessing - blocking` Example: `'Free running'`
TestPointMapping	To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `''` (default) | cell array of character vectors Example: `'{{'TestPoint','AXI4-Lite','x"108"'}}'`
TunableParameterMapping	To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `''` (default) | cell array of character vectors Example: `'{{'myParam','AXI4-Lite','x"108"'}}'`
AXI4RegisterReadback	To save this block property on the model, specify whether you want to enable readback on AXI4 subordinate write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'`
AXI4SlaveIDWidth	To save this block property on the model, specify the number of AXI manager interfaces that you want to connect the DUT IP core to by using the **AXI4 Slave ID Width** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface". Values: `'off'` (default) | `'on'`

Target Specification	
AXI4SlavePortToPipelineRegisterRatio	To save this block property on the model, specify the number of AXI4 subordinate ports for which you want a pipeline register to be inserted by using the **AXI4 Slave port to pipeline register ratio** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'` `'10'` `'20'` `'35'` `'50'`
GenerateDefaultAXI4 Slave	To save this block property on the model, specify whether you want to disable generation of default AXI4 subordinate interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. Values: `'on'` (default) | `'off'`

IPCoreAdditionalFiles	Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).
	
	 You can set this property in the HDL Workflow Advisor, in the **Additional source files** field.
	
	 Values: `''` (default) | character vector
	
	 Example: `'C:\myprojfiles`
	`\led_blinking_file1.vhd;C:\myprojfiles`
	`\led_blinking_file2.vhd;'`

IPCoreName	IP core name, specified as a character vector. You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT. Values: `''` (default) | character vector Example: `'my_model_name'`
IPCoreVersion	IP core version number, specified as a character vector. You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version. Values: `''` (default) | character vector Example: `'1.3'`

Target Specification	
IPDataCaptureBuffer Size	FPGA Data Capture buffer size, specified as a character vector. Use FPGA Data Capture to observe signals in a design when running on an FPGA. The buffer size uses values that are 128*2^n, where n is an integer. By default, the buffer size is 128 (n=0). The maximum value of n is 13, which means that the maximum value for buffer size is 1048576 (=128*2^13). Values: ' ' (default) | character vector Example: `'1.3'`

Restrictions

- Your DUT cannot be an Enabled Synchronous Subsystem.
- You cannot have a Delay block with an external reset port inside the subsystem.
- You cannot generate HDL for the LTE Turbo Decoder block inside an Enabled Synchronous Subsystem.
- HDL code generation supports only boolean datatype at the Enable port.

See Also
State Control | Enable | Resettable Synchronous Subsystem | 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"

Float Typecast

Typecast a floating-point type to an unsigned integer or vice versa

Float Typecast
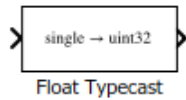
Libraries:
HDL Coder / HDL Floating Point Operations

Description

The block casts the underlying bits of the input to the corresponding fixed-point or floating point representation. The input and output of the block contain the same number of bits.

Input Data Type	Output Data Type
half	uint16
single	uint32
double	uint64
uint32	single
uint64	double
uint16	half

This figure shows how the block mask, behavior, and output data type changes dynamically depending on the input data type that you specify.

single → uint32

Float Typecast

double → ufix64

Float Typecast1

uint32 → single

Float Typecast2

ufix64 → double

Float Typecast3
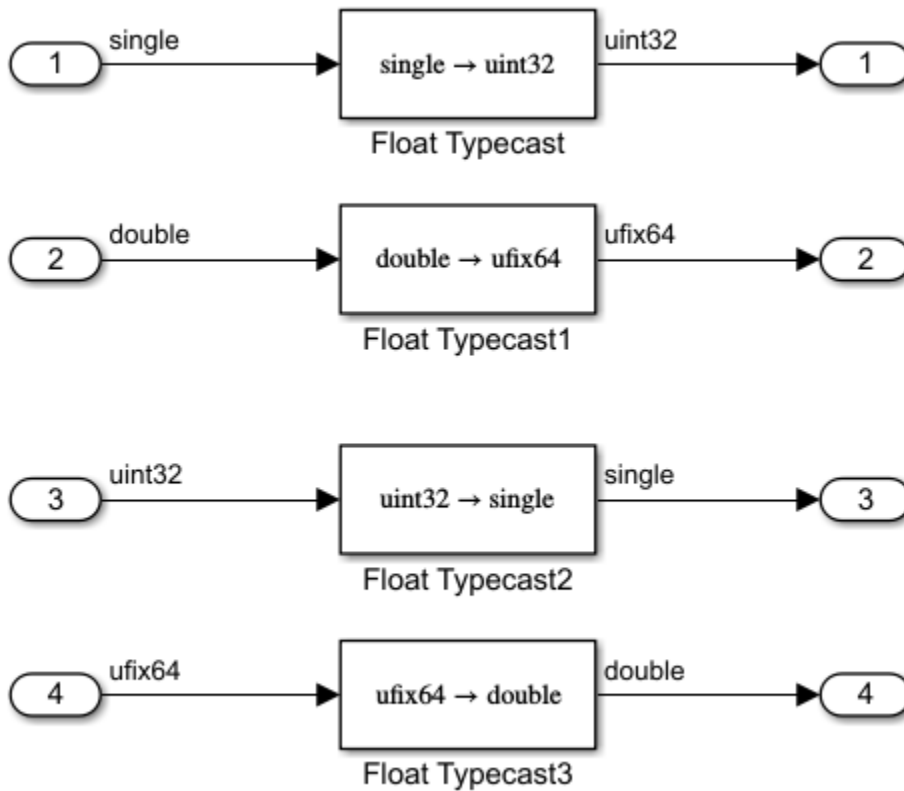
Ports

Input

Port_1(u) — Input signal
scalar | vector

Port to provide input to the block.

Data Types: single | double | uint32 | fixed point

Output

Port_1(y) — Output signal
scalar | vector

Port to obtain calculated output from the block.

Data Types: single | double | uint32 | fixed point

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

The block supports HDL code generation in the `Native Floating Point` mode. To use this mode, specify `single` or `uint32` data types as input to the block. With the HDL Code Advisor, you can replace Data Type Conversion blocks that use the `Stored Integer (SI)` mode and convert between floating-point and fixed-point data types.

The block supports code generation for complex signals.

See Also

Functions
typecast

Topics
"Getting Started with HDL Coder Native Floating-Point Support"

HDL Counter

Free-running or count-limited hardware counter
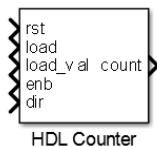
Libraries:
HDL Coder / Sources

Description

The HDL Counter block models a free-running, count-limited, or modulo hardware counter that supports signed and unsigned integer and fixed-point data types. The counter emits its value for the current sample time. During simulation, this block does not report warnings or errors due to wrap on overflow. To report these warnings, see `Simulink.restoreDiagnostic`.

By default, the counter does not have input ports. The counter counts up from an initial value to a threshold value based on the **Counter type**, the **Count to value**, and the **Word length**. The output data type of the counter depends on the **Counter output data**, **Word length**, and **Fraction length**.

Ports

Input

rst — Local reset port
scalar

Local reset port for the counter that when high resets the count value.

Dependencies

To enable this port, set **Local reset port**.

Data Types: `Boolean`

load — Load port
scalar

Load port that when high sets the counter to the load value, `load_val`.

Dependencies

To enable this port, set **Load ports**.

Data Types: `Boolean`

load_val — Load port value
scalar

Data value to load for setting the count value when high input is given to the load port.

3-63

Dependencies

To enable this port, set **Load ports**.

Data Types: `Boolean`

enb — Count enable port
scalar

Enable signal that specifies whether the counter should count from the previous value. When this signal is high, the counter counts continues up or down depending on the direction. When this signal is low, the counter holds the previous value.

Dependencies

To enable this port, set **Count enable port**.

Data Types: `Boolean`

dir — Count direction port
scalar

Count direction that specifies whether to count up or count down. This port interacts with **Step value** to determine count direction.

- `1`: This value is the default that results in an up counter. The **Step value** is added to the current counter value to compute the next value.
- `0`: This value results in a down counter. The **Step value** is subtracted from the current counter value to compute the next value.

Dependencies

To enable this port, set **Count direction port**.

Data Types: `Boolean`

Output

count — Counter value
scalar

This is the counter value. By default, if you do not enable the input ports, the counter counts up to a value that is determined based on the **Counter type**, the **Count to value**, and the **Word length**.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

count_hit — Counter limit indicator
scalar

Counter limit indicator, returned as a Boolean scalar.

- `1`: indicates that the counter reached its limit.
- `0`: indicates that the counter did not reach its limit.

Dependencies

To enable this port, select the **Count hit output port** parameter.

Data Types: `Boolean`

Parameters

Counter type — Counter behavior
`Free running` (default) | `Count limited` | `Modulo`

Counter behavior that determines whether to model a free running, count-limited, or module hardware counter.

- `Free running` (default): The counter continues to increment or decrement by the **Step value** until reset.
- `Count limited`: The counter increments or decrements by the **Step value** until it is exactly equal to the **Count to value**. If the **Step value** value is such that the count value does not exactly equal the **Count to value**, then it can continue counting to a threshold value that is determined by the word length.
- `Modulo`: The counter increments or decrements by the **Step value** until it reaches the **Count to value**. If the **Step value** value is such that the count value does not exactly equal the **Count to value**, then the counter wraps to a value that is determined by the wrapping step value.

Programmatic Use
Block parameter: `CountType`
Type: character vector
Value: `'Free running'` | `'Count limited'` | `'Modulo'`
Default: `'Free running'`

Initial value — Counter value after reset
`0` (default)

The value to which the counter resets. The default value is `0`.

Programmatic Use
Block parameter: `CountInit`
Type: character vector
Value: An integer greater than or equal to zero
Default: `'0'`

Step value — Step value for count
`1` (default)

Value added to counter at each sample time. The default value is `1`.

Programmatic Use
Block parameter: `CountStep`
Type: character vector
Value: An integer greater than or equal to zero
Default: `'1'`

Count to value — Threshold count value
`25` (default)

When you use a `Count limited` counter, if the count is exactly equal to **Count to value**, the count restarts at the **Initial value**. If the count value exceeds the **Count to value**, the counter continues counting to a threshold value that depends on the **Word length**. The default is 25.

When you use a `Modulo` counter, if the count reaches the **Count to value**, the count restarts at a value that is determined by the wrapping step value.

Dependencies

To enable this parameter, set **Counter type** to `Count limited` or `Modulo`.

Programmatic Use
Block parameter: `CountMax`
Type: character vector
Value: An integer greater than or equal to zero
Default: `'25'`

Count from — Criteria for count from value
`Initial value` (default) | `Specify`

Specifies the parameter that sets the start value after rollover when you use a `Count limited` or `Free running` counter. When you use a `Modulo` counter, the counter can rollover to a wrapping step value that is different from the value to count from. When set to `Specify`, the **Count from value** parameter is the start value after rollover. The default is `Initial value`.

Programmatic Use
Block parameter: `CountFromType`
Type: character vector
Value: `'Initial value'` | `'Specify'`
Default: `'Initial value'`

Count from value — Initial value to count from
`0` (default)

Counter value after rollover when **Count from** is set to `Specify`. The default is 0.

Programmatic Use
Block parameter: `CountFrom`
Type: character vector
Value: `'Initial value'` | `'Specify'`
Default: `'Initial value'`

Local reset port — Local port to reset counter
`off` (default) | on

When selected, creates a local reset port, `rst`.

Programmatic Use
Block parameter: `CountResetPort`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Load ports — Port for load value
`off` (default) | on

When selected, creates a load data port, `load_val`, and load trigger port, `load`.

Programmatic Use
Block parameter: `CountLoadPort`

Type: character vector
Value: 'off' | 'on'
Default: 'off'

Count enable port — Port to enable counting
off (default) | on

When selected, creates a count enable port, enb.

Programmatic Use
Block parameter: CountEnbPort
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Count direction port — Port for count direction
off (default) | on

When selected, creates a count direction port, dir.

Enabling this parameter disables the **Count hit output port** parameter.

Programmatic Use
Block parameter: CountDirPort
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Count hit output port — Port for count limit
off (default) | on

Select this parameter to enable the **count_hit** output port.

Enabling this parameter clears the **Count direction port** parameter.

Programmatic Use
Block parameter: CountHitOutputPort
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Counter output data is — Output data type signedness
Unsigned (default) | Signed

Output data type signedness. The default is Unsigned.

Programmatic Use
Block parameter: CountDataType
Type: character vector
Value: 'Unsigned' | 'Signed'
Default: 'off'

Word length — Counter word length
8 (default)

Bit width, including sign bit, for an integer counter; word length for a fixed-point data type counter. The minimum value if **Output data type** is Unsigned is 1, 2 if Signed. The maximum value is 125. The default is 8.

Programmatic Use
Block parameter: CountWordLen
Type: character vector
Value: An integer greater than or equal to one
Default: '8'

Fraction length — Counter fraction length
0 (default)

Fixed-point data type fraction length. The default is 0.

Programmatic Use
Block parameter: CountFracLen
Type: character vector
Value: An integer greater than or equal to zero
Default: '0'

Sample time — Counter sample time
1 (default)

Sample time. The default is 1. This parameter is not available, and the block inherits its sample time from the input ports when any of these parameters is selected:

- **Local reset port**
- **Load ports**
- **Count enable port**
- **Count direction port**

Programmatic Use
Block parameter: CountSampTime
Type: character vector
Value: An integer greater than or equal to one
Default: '1'

Algorithms

Free Running Counter Behavior

By default, when you do not enable the control ports, the counter counts upwards from zero in free-running mode. In this mode, the counter increments in steps of one at each sample time unit until it reaches the threshold value. The count threshold value is calculated as 2^(Word length) - 1. When it reaches the count value, the counter resets to the initial value.

The counter behavior depends on control ports that you specify. This table shows the priority of the control signals and how the counter value is updated in relation to the control signals.

Local reset, rst	Load trigger, load	Count enable, enb	Count direction, dir	Next Counter Value
1	–	–	–	Initial value
0	1	–	–	`load_val` value
0	0	0	–	Current value
0	0	1	1	Current value + step value
0	0	1	0	Current value - step value

The **Step value** parameter and optional count direction port, `dir`, interact to determine the actual count direction.

dir Signal Value	Step Value Sign	Actual Count Direction
1	+ (positive)	Up
1	- (negative)	Down
0	+ (positive)	Down
0	- (negative)	Up

This figure illustrates the free running mode of operation. In this example, the counter has a **Word length** of 4. The **Initial value** is 2, load value is 7, and the **Step value** is 1. When the **Enb** signal is high, the counter increments by steps of one. When **Rst** signal becomes high, the counter resets to the initial value, 2. When the **Dir** port goes low, the counter decrements from 4 to 3 at time step `11s`. The count value is tied to the load value when the **Load** port becomes high.
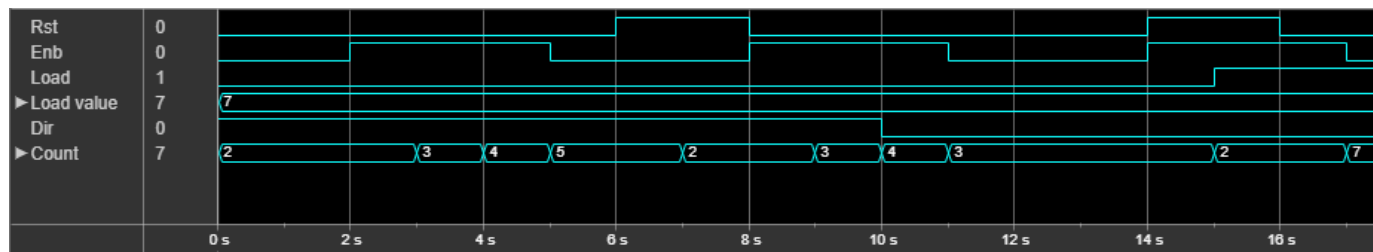
Count Limited and Modulo Operation Modes

You can use the **Counter type** parameter to specify the counter behavior. The `Count limited` mode of the counter wraps the count to the initial value when the counter exactly reaches the **Count to value**. If the counter does not exactly reach the **Count to value**, it can exceed this value. For an up counter, the count value can reach a threshold value that is calculated as $2\text{^}(\text{Word length}) - 1$. For a down counter, the count value can reach the **Initial value**. When it reaches or exactly matches this threshold value, the counter resets to a value that is determined by the wrapping step value.

The `Modulo` mode of the counter wraps the count when it reaches or exactly matches the **Count to value**. Instead of restarting at the initial value, the counter wraps back to a value that is determined by a wrapping step value. For an up counter, the wrapping step value is calculated as *step value - (count to value + 1) + count from value*. For a down counter, the wrapping step value is calculated as *count from value - step value + (count to value + 1)*.

This figure illustrates the `Count limited` and `Modulo` modes of operation. In this example, the counter has a **Word length** of 4, **Initial value** of 2, **Step value** of 3, and **Count to value** of 12.
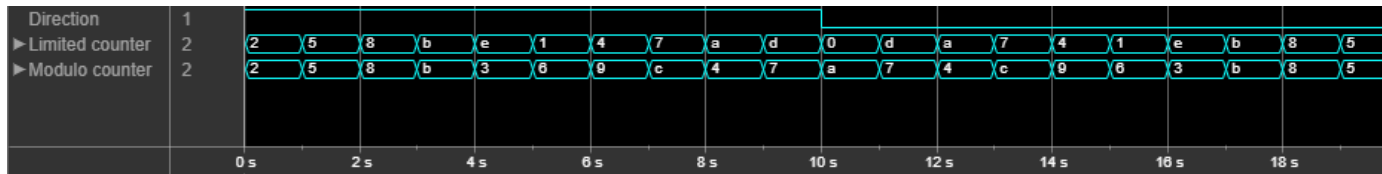
In the `Count limited` mode, as the count value reaches 11, it exceeds the **Count to value** and reaches 14. As the threshold value is 15 (2^4-1), the counter resets to a value that is determined by the wrapping step value. When counting down, the counter exceeds the **Initial value** and can reach zero. It then resets to a value that is determined by the wrapping step value.

In the `Modulo` mode, as the count value reaches 11, the counter resets to a value that is determined by the wrapping step value. The wrapping step value is 3 - (12 + 1) + 2 = -8. Therefore, the counter resets to the value 3 (11 + (-8)). When counting down, as the count value reaches 4, the counter resets to a value that is determined by the wrapping step value. The wrapping step value is 2 - 3 + (12 + 1) = 12. Therefore, the counter resets to the value 12.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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.
- The block does not support vectors. Only scalar types are supported for block inputs and outputs.
- Load value used in `Modulo` counter mode must be within the range of the **Count from** and **Count to value** to avoid simulation mismatches.

See Also

HDL FIFO

Topics
"Generate HDL Code from Simulink Model from Command Line"

HDL FIFO

Stores sequence of input samples in first in, first out (FIFO) register
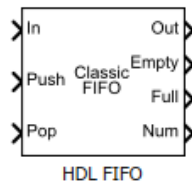
Libraries:
HDL Coder / HDL RAMs

Description

The HDL FIFO block stores a sequence of input samples in a first in, first out (FIFO) register. The data written first into the FIFO register comes out first. The block implementation resembles the FIFO unit in hardware platforms in terms of functionality and behavior.

The HDL FIFO block uses the Simple Dual Port RAM block internally. You can use the HDL FIFO block to generate HDL code that maps to RAM in most FPGAs.

Ports

Input

In — Data input signal
scalar

Data input signal to the FIFO block. When you write data into the FIFO, the newest data is pushed to the end of the FIFO register. The block pushes subsequent data entries below this entry.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point
Complex Number Support: Yes

Push — Write control signal
scalar

When this port receives a value of 1, the block pushes the input at the In port onto the end of the FIFO register.

Data Types: Boolean

Pop — Read control signal
scalar

When this port receives a value of 1, the block pops the first element off the FIFO register and holds the Out port at that value.

Data Types: Boolean

Note If two or more of the control input ports are triggered in the same time step, the pop operation executes first, followed by the push operation.

rst — Reset control signal
scalar

When the reset port receives a value of 1, it resets the Empty, Full, and Num outputs of the HDL FIFO block.

Dependencies

To enable this port, select the block parameter **Local reset port**.

Data Types: Boolean

Output

Out — Data output signal
scalar

Data output signal from the FIFO block. When you read data from the FIFO, the data that you wrote first into the FIFO register comes off the FIFO and is held at the output.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point
Complex Number Support: Yes

Empty — Data output signal
scalar

Control signal output from the FIFO that becomes 1 when the FIFO register is empty and does not contain data entries.

Dependencies

To enable this port, **Show empty register indicator port (Empty)** must be selected.

Data Types: Boolean

Full — Data output signal
scalar

Control signal output from the FIFO that becomes 1 when the FIFO register is full and cannot take more data entries.

Dependencies

To enable this port, **Show full register indicator port (Full)** must be selected.

Data Types: Boolean

Num — Number of entries
scalar

Number of data entries that are currently inside the FIFO register. **Num** increments by 1 for every data that you write into the FIFO. **Num** decrements by 1 for every data that you read from the FIFO.

Dependencies

To enable this port, **Show number of entries register port (Num)** must be selected.

Data Types: `double`

Parameters

Register size — Number of entries
10 (default)

Specify the number of entries that the FIFO register can hold. The default value for **Register size** is 10. The minimum value for **Register size** is 4.

Programmatic Use
Block parameter: `fifo_size`
Type: character vector
Value: An integer greater than or equal to four
Default: `'10'`

Mode — Operation mode
`'Classic'` (default) | `'FWFT'`

Specify the operation mode of the FIFO. By default, the **Mode** is set to `Classic`. You see that the block icon displays `Classic FIFO`. You can change the **Mode** to `FWFT`. When you change the **Mode**, the block icon displays `FWFT FIFO`. By using the `FWFT` mode, you can lookahead and see the first word written to the FIFO register without placing a read request. The `FWFT` mode is especially useful when you apply the back-pressure with AXI4-Stream interfaces.

Programmatic Use
Block parameter: `mode`
Type: character vector
Value: `'Classic'` | `'FWFT'` |
Default: `'Classic'`

The ratio of output sample time to input sample time — Sample rate ratio
1 (default)

Specify the ratio of output sample time to input sample time. The default ratio is 1, which means that the inputs `In` and `Push`, and outputs `Out` and `Pop`, run at the same sample rate. The inputs and outputs can run at different sample times. Use a positive integer or `1/N`, where `N` is a positive integer. For example, if you enter `1/2`, the output sample time is half the input sample time, or the outputs run faster. The `Full`, `Empty`, and `Num` signals run at the faster rate.

Programmatic Use
Block parameter: `ratio`
Type: character vector
Value: An integer greater than or equal to one
Default: `'1'`

Push onto full register — Overflow condition
`'Warning'` (default) | `'Ignore'` | `'Error'`

Specify how you want the block to respond when you write to a FIFO that is full. The default is `Warning`.

Programmatic Use
Block parameter: push_msg
Type: character vector
Value: 'Warning' | 'Ignore' | 'Error'
Default: 'Warning'

Pop onto empty register — Underflow condition
'Warning' (default) | 'Ignore' | 'Error'

Specify how you want the block to respond when you read from a FIFO that is empty. The default is
Warning.

Programmatic Use
Block parameter: pop_msg
Type: character vector
Value: 'Warning' | 'Ignore' | 'Error'
Default: 'Warning'

Show empty register indicator port (Empty) — Optional empty port
on (default) | off

Specify whether to enable the Empty output port. This port outputs a 1 when the FIFO register is
empty and 0 when the FIFO contains one or more data entries.

Programmatic Use
Block parameter: show_empty
Type: character vector
Value: 'on' | 'off'
Default: 'on'

Show full register indicator port (Full) — Optional full port
on (default) | off

Specify whether to enable the Full output port. This port outputs a 1 when the FIFO register is full.

Programmatic Use
Block parameter: show_full
Type: character vector
Value: 'on' | 'off'
Default: 'on'

Show num register indicator port (Num) — Optional num port
on (default) | off

Specify whether to enable the Num output port. This port outputs the number of data entries that are
currently available in the FIFO queue.

Programmatic Use
Block parameter: show_num
Type: character vector
Value: 'on' | 'off'
Default: 'on'

Local reset port — Optional reset port
off (default) | on

Specify whether to insert an additional `rst` input port.

Programmatic Use
Block parameter: `rst_port`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Algorithms

FIFO Write Operation

This figure displays the FIFO write operation. The **Push** input port acts as the enable signal for the write operation. This signal is denoted by the **write_en** signal in the figure.
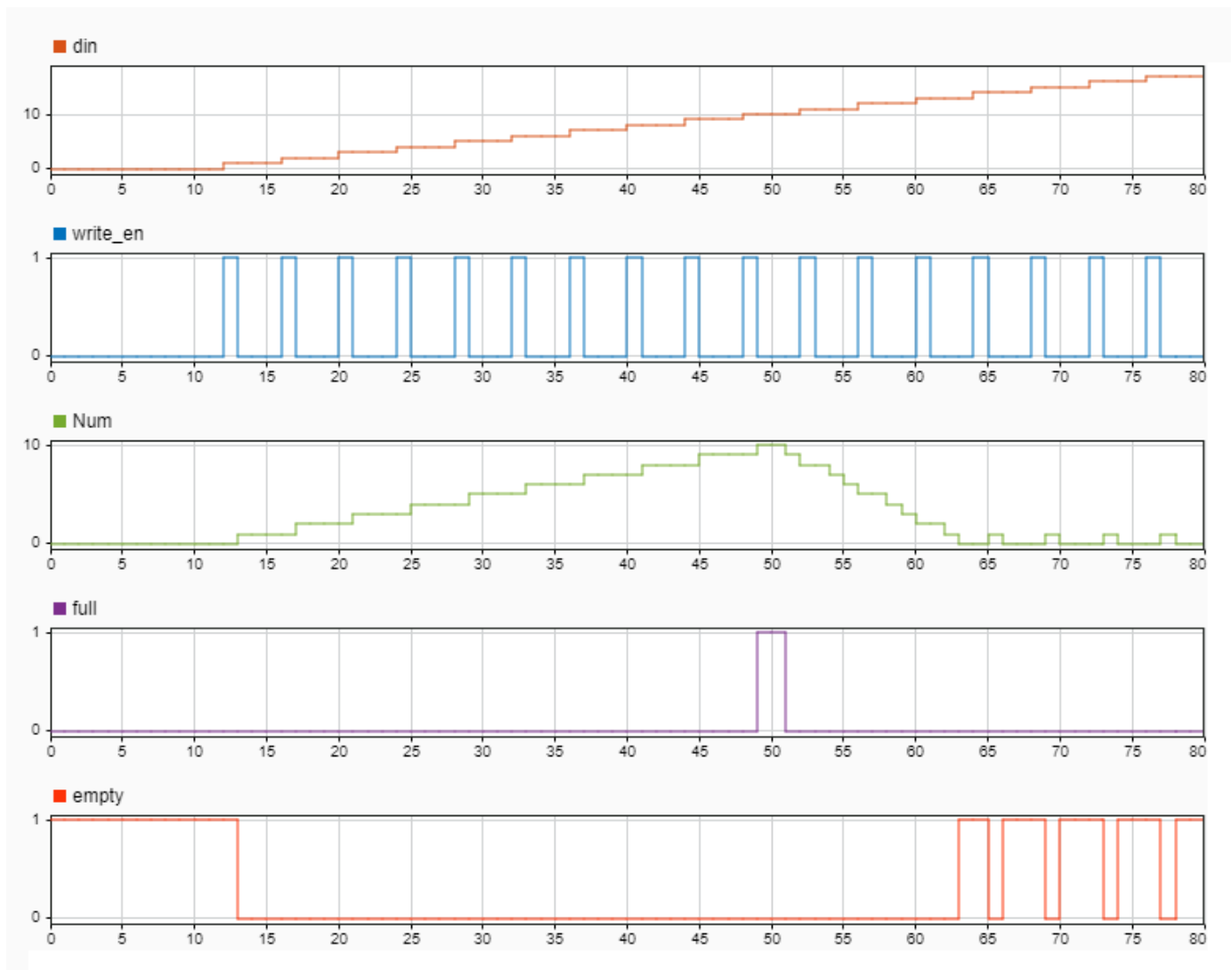
When the **write_en** signal is `0`, the block does not write data to the FIFO and the **empty** flag is asserted.

When the **write_en** becomes 1, the block pushes the **din** signal at input port **In** to the end of the FIFO register in the next time step. The **Num** signal indicates the number of data entries in the FIFO register. Every time you write data into the FIFO, the **Num** signal increments by 1. At time step 12, **write_en** is 1. At the next time step 13, data is written to the FIFO. **Num** signal increments by 1 and the **empty** flag is de-asserted.

This FIFO uses the default register size of 10. When the **Num** signal becomes equal to the **Register size** at time step 49, the **Full** signal is asserted. After the **Full** signal becomes 1, if you try to write more entries into the FIFO, the block generates a warning.

Classic FIFO Read Operation

This figure displays the FIFO read operation. The **Pop** input port acts as the enable signal for the read operation. This signal is denoted by the **read_en** signal in the figure.
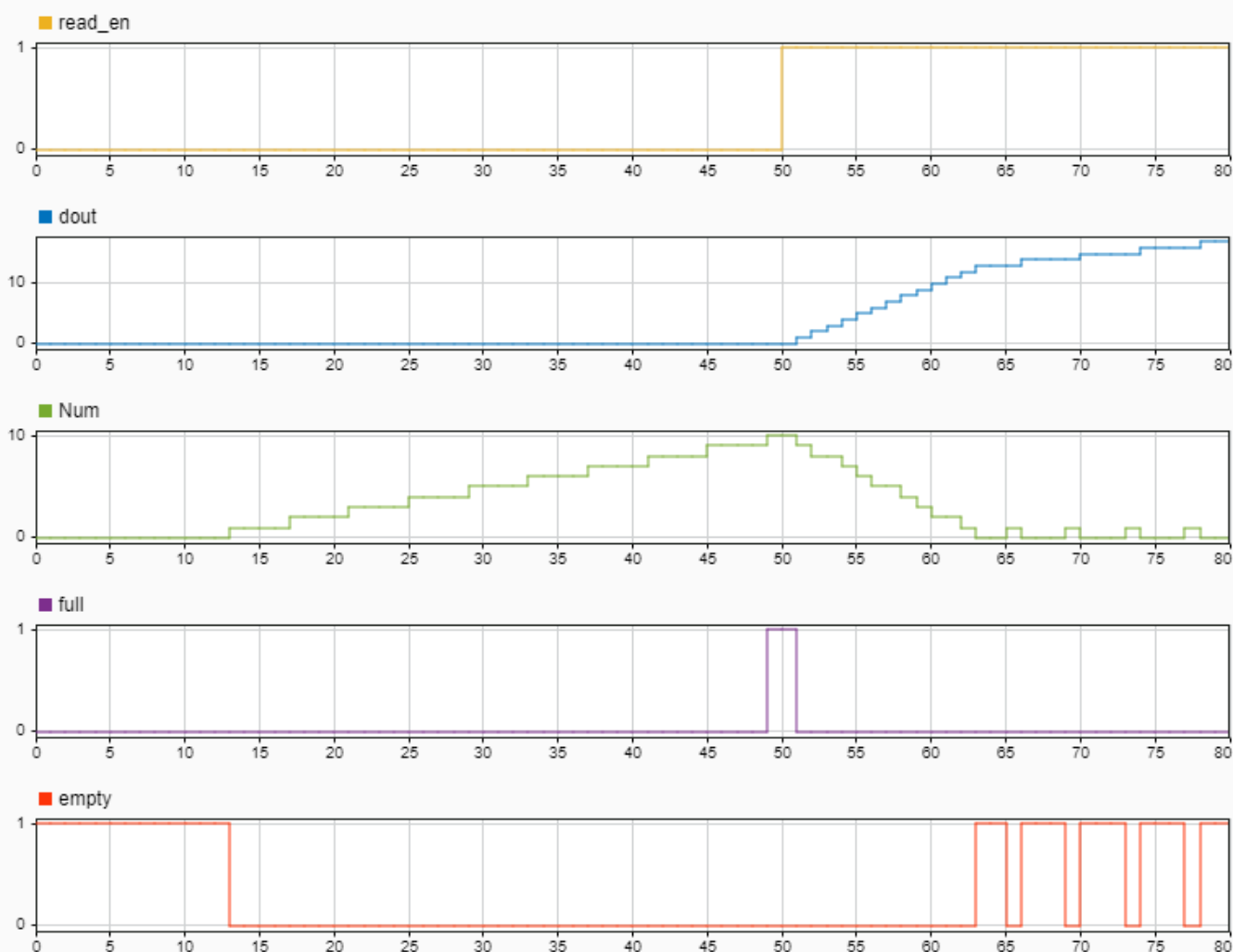
When the **read_en** signal is 0, the block output **dout** is 0. When the **read_en** signal becomes 1 at time step 50, the **dout** signal outputs the oldest entry in the FIFO in the next time step 51. The **Full** flag is de-asserted and the **Num** signal decrements by 1 starting from time step 51 as you read data from the FIFO.

When the **Num** signal becomes equal to `0`, the **Empty** signal is asserted. After the **Empty** signal becomes `0`, if you try to read more entries from the FIFO, the block generates a warning.

FWFT FIFO Read Operation

This figure displays the FIFO read operation. The **Pop** input port acts as the enable signal for the read operation. This signal is denoted by the **read_en** signal in the figure.
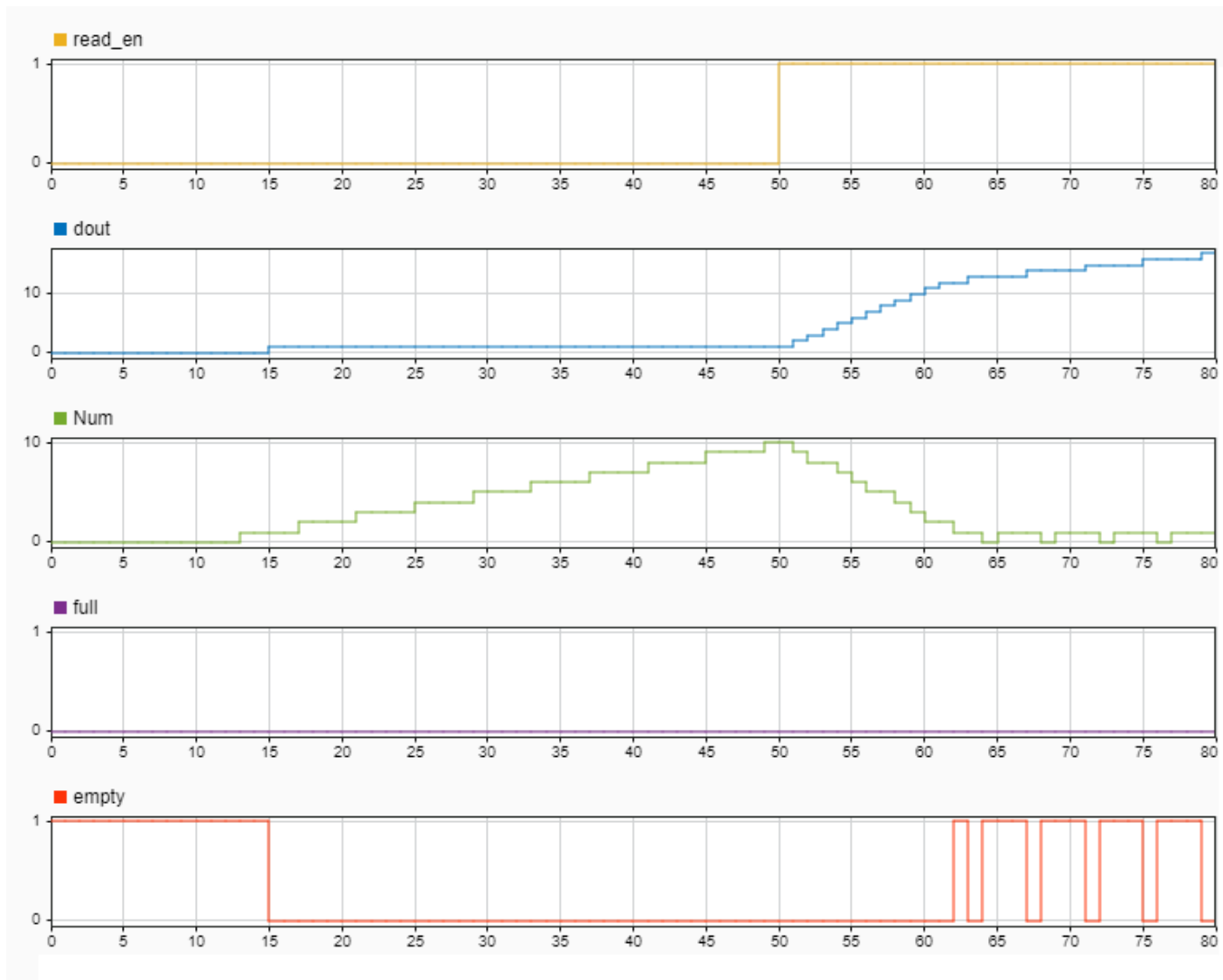
By default, the HDL FIFO works in the `Classic` mode. You can also use a first-word-fall-through (FWFT) mode for the FIFO. In the Block Parameters dialog box, specify the **Mode** as `FWFT`.

In the `FWFT` mode, the write operation works in the same way as the `Classic` mode. The `FWFT` mode differs from the `Classic` mode when you perform a read operation. In the `Classic` mode, after you place a read request or input a `1` to the **Pop** port, the data becomes available at the FIFO output in the next clock cycle. In the `FWFT` mode, the first word you write to the FIFO falls through to the output and is available at the output signal **Out**.

In the figure, though **read-en** becomes 1 at time step 50, the FIFO read the first word **dout** at time step 15. You can use this capability to look ahead and see the first word that has been written to the FIFO.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to the memory blocks, such as distributed RAMs, block RAMs, or UltraRAM, on the target FPGA. The default value is `empty string`. See also "RAMDirective".

See Also
Dual Rate Dual Port RAM | Simple Dual Port RAM

Topics
"Design Considerations for RAM Blocks and Blocks in HDL Operations Library"

Multiply-Accumulate

Perform a multiply-accumulate operation on the inputs
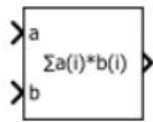
Multiply-Accumulate

Libraries:
HDL Coder / HDL Operations

Description

The Multiply-Accumulate block performs this operation on inputs `a` , `b`, and bias `c` to compute result `dataOut`.

`dataOut = sum(a.* b) + c`

By default, the block operates in the vector mode. The inputs `a` and `b` can be scalars, vectors, or 2-D matrices. By default, the bias value `c` is equal to zero. The block computes the dot product of inputs `a` and `b`. You can specify a nonzero value for `c` by using `Dialog` or `Input port` as the **Source**. The block adds this bias to the dot product of `a` and `b`. The multiplication operation is full precision irrespective of the **Output data type** setting. The **Output data type** and **Integer rounding mode** settings apply to the addition operation.

By using the **Operation Mode** setting, you can specify streaming modes of operation for the Multiply-Accumulate block. For HDL code generation, when you use the streaming operation mode, you must input scalar values to the block. The block has two streaming modes: `Streaming - using Start and End ports` and `Streaming - using Number of Samples`. When you select these streaming modes, you can specify the control signals to use with the mode. The control signals specify when to start and end accumulation and when the output is valid.

Limitations for HDL Code Generation

- Scalar inputs are not supported for HDL code generation. To generate code for the block, use vector inputs. For scalar inputs, use the Multiply-Add block.
- Matrix data types are not supported at the block port interfaces. If you have matrix type signals, use the Product, Matrix Multiply block in matrix multiplication mode.
- Streaming modes of operation for the block are not supported inside a Resettable Subsystem block for HDL code generation.

Ports

Input

a — Input signal
`vector | matrix | array | bus`

Port to provide input to the block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

b — Input signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide input to the block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

c — Bias signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the bias signal to the block. The block adds this bias to the inputs. Make sure that the bias signal data type matches that of the dot product of the inputs.

Dependencies

To enable this port, set **Source** to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

startIn — Start of accumulation control signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to start accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the **dataOut** signal, both **startIn** and **validIn** signals must be high. The **dataOut** signal produces the accumulated result from the next clock cycle.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

validIn — Valid input control signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate that the input signal is valid for accumulation. It is recommended that you use a `boolean` data type signal as input to the port. To start obtaining the accumulated output value from the **dataOut** signal, both **validIn** and **startIn** signals must be high. The **dataOut** signal produces the accumulated result from the next clock cycle. The **validIn** signal has higher priority than **startIn** and **endIn** signals.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` or `Streaming - using Number of Samples`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

endIn — End of accumulation control signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port to provide the control signal to indicate end of accumulation. You can use the **startIn** and **endIn** signals with the **validIn** signal to indicate a frame that contains the accumulated output.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **End input and output ports**.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point`

Output

dataOut — Output signal
`scalar | vector | matrix | array | bus`

Port that generates the output data from the multiply-accumulate operation. By default, the block uses the `Vector` mode of operation and computes the dot product of the input signals, and adds the bias to produce the result. If you specify a streaming mode of operation as **Operation Mode**, the value of the **dataOut** signal depends on the control signals that you provide. The data type of the output signal is same as that of the accumulator.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point`

startOut — Start of accumulation output control signal
`scalar | vector | matrix | array | bus`

Port that generates output control signal to indicate the start of accumulation. When both **validIn** and **startIn** are high, the **startOut** signal becomes high in the next clock cycle. The clock cycle at which **startOut** becomes high indicates the start of a frame and that the **dataOut** signal has started producing valid accumulated output.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **Start output port**.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point`

validOut — Valid output control signal
`scalar | vector | matrix | array | bus`

Port that generates the output control signal to indicate that the **dataOut** signal is valid. When the **validIn** signal becomes high , the **validOut** signal becomes high in the next clock cycle and indicates that the **dataOut** is valid.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **Valid output port**.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point`

endOut — End of accumulation output control signal
`scalar | vector | matrix | array | bus`

Port that generates the output control signal to indicate the end of accumulation. You can use the clock cycles between when the **startOut** signal becomes high and when the **endOut** signal becomes high to indicate a valid frame that contains the accumulated output.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports` and then select **End input and output ports**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

countOut — Count output control signal
`scalar` | `vector` | `matrix` | `array` | `bus`

Port that generates the output control signal to indicate number of samples to accumulate. The value of this signal increases from 1 to the value that you specify for **Number of Samples**. As long as the **validIn** signal is high, the **countOut** increments by 1 every clock cycle.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples` and then select **Count output port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Operation Mode — Mode of accumulation of inputs
`'Vector'` (default) | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`

You can specify the **Operation Mode** as:

- `Vector`: You can use scalars or vectors as inputs. The block performs the dot product of the inputs u1 and u2 and adds bias k to produce the result.
- `Streaming - using Start and End Ports`: Use scalar inputs for HDL code generation. In this mode, you can use the **startIn** and **endIn** control signals to determine when to start and stop accumulation. The output data is valid when **validIn** is high.
- `Streaming - using Number of Samples`: Use scalar inputs for HDL code generation. In this mode, you can specify the **Number of Samples** and use the **countIn** control signal to determine when to start and stop accumulation. The output data is valid when **validIn** is high.

Programmatic Use
Block parameter: opMode
Type: character vector
Value: `'Vector'` | `'Streaming - using Start and End Ports'` | `'Streaming - using Number of Samples'`
Default: `'Vector'`

Bias — Offset to add to the input dot product
`{'0.0'}` (default)

You can specify the bias with:

- **Source** as `Dialog`. Then, specify the **Value**.
- **Source** as `Input port`. This setting creates an external input port c to input the bias signal to the block.

Programmatic Use
Block parameter: initValueSetting
Type: character vector
Value: 'Dialog' | 'Input port'
Default: 'Dialog'

If you set **Source** as Dialog, you can specify the initial value by using the initValue2 setting.
Block parameter: initValue2
Type: character vector
Value: An integer greater than or equal to zero
Default: '0.0'

Number of Samples — Number of samples of valid accumulated output signal
{'2'} (default)

You can specify the **Number of Samples** to specify a frame containing the number of samples of valid accumulated output **dataOut**.

Dependencies

To enable this port, set **Operation Mode** to Streaming - using Number of Samples.

Programmatic Use
Block parameter: num_samples
Type: character vector
Value: An integer greater than or equal to zero
Default: '2'

Output data type — Data type of the block output
Inherit: Inherit via back propagation (default)

Set the output data type to:

- A rule that inherits a data type, such as Inherit: Same as first input.
- A built-in data type, such as single or int16.
- The name of a data type object. for instance, a Simulink.NumericType object.
- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

The streaming modes do not support Inherit: Inherit via internal rule. When you set the **Output data type**, you can use the **Data Type Assistant**. To display the assistant, click the **Show data type assistant** >> .

Programmatic Use
Block parameter: OutDataTypeStr
Type: character vector
Default: {'Inherit: Inherit via internal rule'}

To see possible values that you can specify for this parameter, see "Block-Specific Parameters".

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding action as:

Ceiling

> Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

Convergent

> Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer™ `convergent` function.

Floor

> Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

Nearest

> Rounds the number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round

> Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest

> Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

> Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Programmatic Use
Block parameter: RndMeth
Type: character vector
Default: {'Floor'}

To see possible values that you can specify for this parameter, see "Block-Specific Parameters".

Valid output port — Control generation of validOut output port
off (default) | on

Control generation of the **validOut** output port. This port indicates whether **dataOut** is valid.

☐ off

> Does not display the **validOut** output port.

☑ on

> Display the **validOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples` or `Streaming - using Start and End Ports`.

Programmatic Use
Block parameter: validOut
Type: character vector
Values: 'off' | 'on'

Default: `'off'`

End input and output ports — Control generation of endIn input port and endOut output port
off (default) | on

Control generation of the **endIn** input port and the **endOut** output port. The ports indicate the end of a frame containing valid accumulation output.

☐ off

Does not display the **endIn** input port and the **endOut** output port.

☑ on

Display the **endIn** input port and the **endOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Programmatic Use
Block parameter: `endInandOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Start output port — Control generation of startOut output port
off (default) | on

Control generation of the **startOut** output port. This port generates the **startOut** signal that indicates the start of a frame containing valid accumulated output.

☐ off

Does not display the **startOut** output port.

☑ on

Display the **startOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Start and End Ports`.

Programmatic Use
Block parameter: `startOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Count output port — Control generation of countOut output port
off (default) | on

Control generation of the **countOut** output port. This port generates the counter that indicates a frame containing valid samples.

☐ off

Does not display the **countOut** output port.

☑ on

Display the **countOut** output port.

Dependencies

To enable this port, set **Operation Mode** to `Streaming - using Number of Samples`.

Programmatic Use
Block parameter: `countOut`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Benefits

With the Multiply-Accumulate block, you can:

- Perform matrix multiplication operations. For example, if you have two matrix inputs with dimensions `N-by-M` and `M-by-P`, you can compute the result by using `N-by-P` multiply-accumulate operations in parallel.
- Replace a sequence of multiplication and addition operations, such as in filter blocks, and improve the performance on hardware by mapping to DSP slices on the FPGA. This figure shows how you can use the Multiply-Accumulate block with the `sfir_fixed` model.
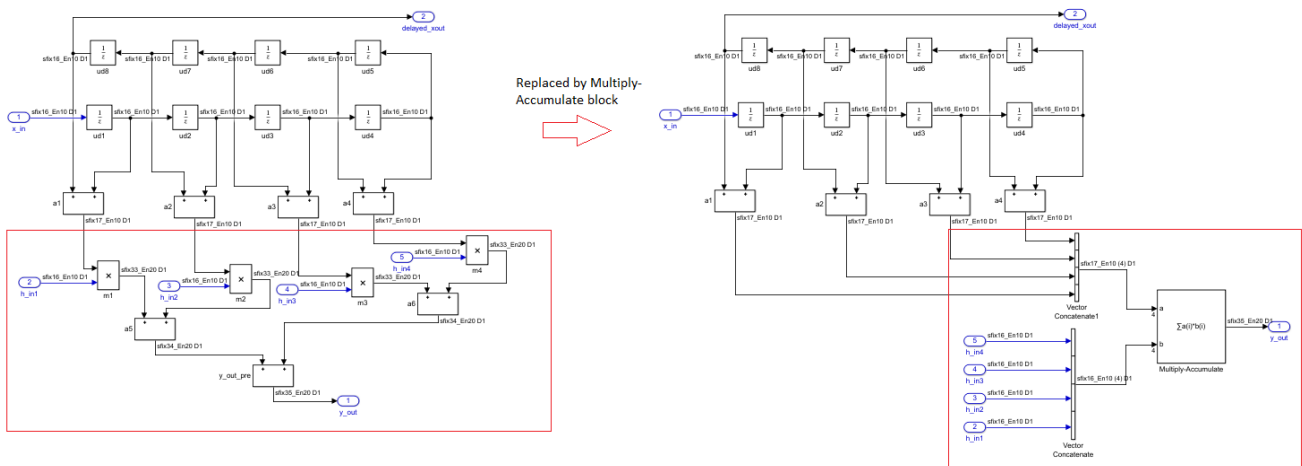
Algorithms

Streaming Mode Using Start and End Ports

You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select `Streaming - using Start and End Ports`, you see three additional settings enabled by default. The settings include:

- **Valid output port**
- **End input and output ports**

- **Start output port**

It is recommended that you leave these settings enabled. When you apply the settings, three additional input ports and three additional output ports appear:

Input Ports	Output Ports
startIn	**startOut**
validIn	**validOut**
endIn	**endOut**

This figure illustrates the streaming mode of operation using the start and end ports. In this example, the bias value is 8.
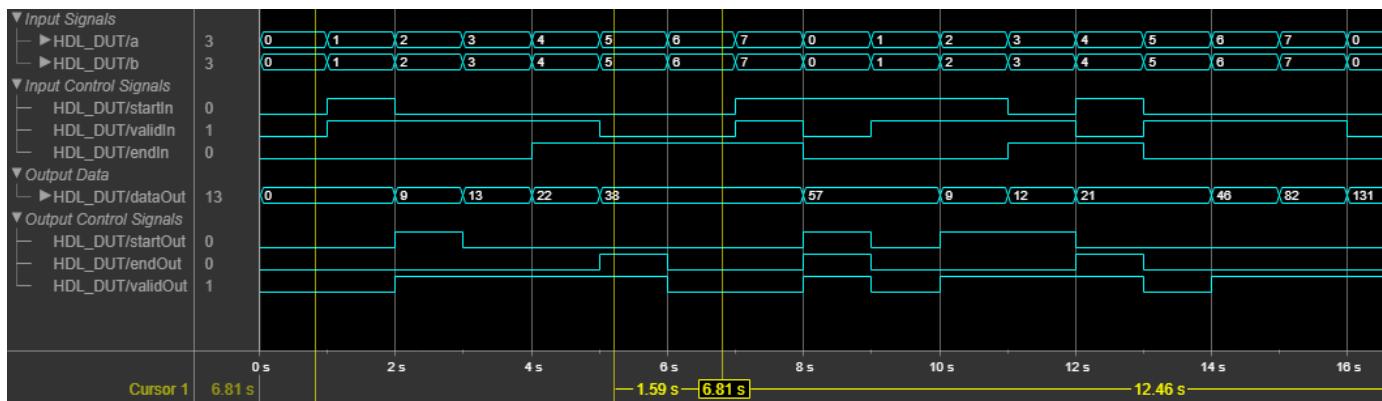
Initially, when **validIn** is low, **dataOut** is zero. At time `1s`, both **startIn** and **validIn** become high. Therefore, **validOut** becomes high in the next clock cycle and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the values of **a** and **b** from the previous clock cycle. For example, at time t = `2s`, **dataOut** = 1*1 + 8 = 9.

To continue accumulation, make **startIn** low at the next clock cycle and keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. At each time step, **dataOut** computes the product of the inputs from the previous clock cycle and sums the result with the **dataOut** value from the previous clock cycle. For example, at time t = 3s, **dataOut** = 2*2 + 9 = 13.

When **validIn** becomes low, **dataOut** holds the output value as seen at time t = 5s. At t = 5s, **endIn** and **validIn** are high. Therefore, **endOut** becomes high in the next clock cycle, which indicates end of frame. Therefore the frame between t = 2s (when **startOut** is high) and t = 6s (when **endOut** is high) indicates a frame containing valid output.

If **startIn**, **validIn**, and **endIn** are both high at the same time, only the **dataOut** corresponding to those inputs are accumulated as seen at t = 8s. If **startIn** is high for multiple clock cycles, and if **validIn** is high, the accumulator is reset at each clock cycle as seen at t = 10s and t = 11s. The accumulation continues at t = 12s.

Streaming Mode Using Number of Samples

You can use the **Operation Mode** setting for the block to specify a streaming mode of operation. When you select `Streaming - using Number of Samples`, you see two additional settings enabled by default. The settings include:

- **Valid output port**
- **Count output port**

It is recommended that you leave these settings enabled. When you apply the settings, you have an additional input port **validIn** and three additional output ports appear:

- **endOut**
- **validOut**
- **countOut**

This figure illustrates the streaming mode of operation using the number of samples. In this example, the bias value is 8 and the **Number of Samples** is 5.
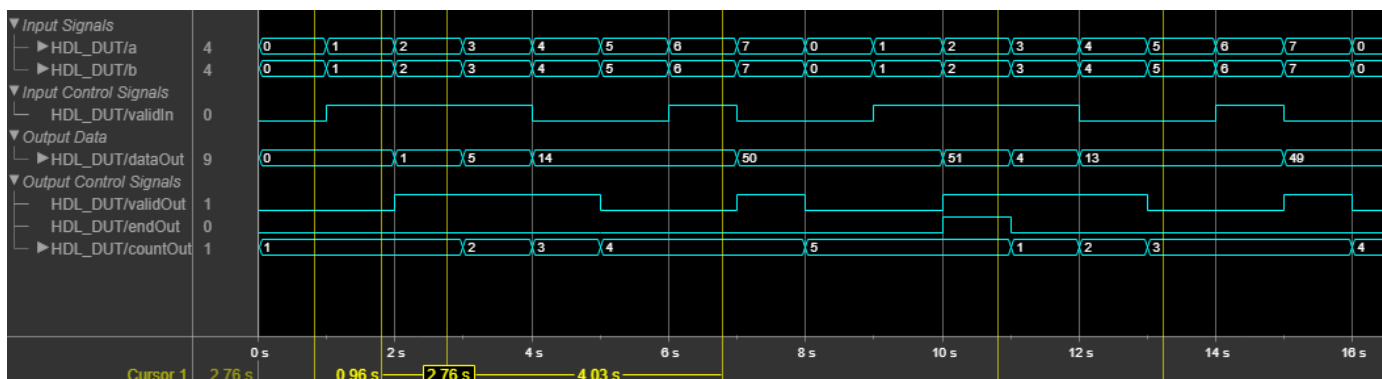
Initially, when **validIn** is low, **dataOut** is 0 and **countOut** is 1. At time 1s, **validIn** becomes high. Therefore, **validOut** becomes high in the next clock cycle, and **dataOut** starts producing valid accumulation output. During accumulation, **dataOut** takes the values of **a** and **b** from the previous clock cycle. For example, at time t = 2s, **dataOut** = 1*1 = 1. **countOut** increments by 1 at the next clock cycle, that is, at t = 3s, **countOut** becomes 2.

To continue accumulation, keep **validIn** high. **dataOut** continues accumulating the inputs until **validIn** becomes low. When five valid outputs are obtained from **dataOut**, **countOut** becomes 5 and **endOut** becomes high, which indicates the end of the frame. Therefore, the time between when **countOut** is 1 and when **countOut** is five indicates a frame containing valid output.

The accumulator counter is now reset and **countOut** starts from 1. When **validIn** becomes high again, **dataOut** starts accumulating a new set of values and **countOut** starts incrementing for each valid **dataOut**.

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

HDL Architecture Setting	Description
`Auto` (Default)	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.
`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 subsystem that contains the Multiply-Accumulate block, this architecture uses a single-rate implementation. This implementation runs the shared resources at the clock-rate and multiplexes the input vectors with a Multiply-Add block. This implementation introduces an additional latency of `N` at the clock rate. When you synthesize your design, depending on the input bit widths, this architecture maps to one DSP slice on the FPGA.

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`. For more details, see "ConstrainedOutputPipeline".

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

Complex Data Support

When you use complex signals, this block can generate HDL code, but does not map to DSP slices.

Reset Type Recommendation

For the mapping of the Multiply-Accumulate block to DSP slices, use these reset type settings:

- For Xilinx FPGA boards, set **Reset type** to Synchronous.
- For Altera FPGA boards, set **Reset type** to Asynchronous.

To set the reset type, select **HDL Code Generation** > **Global settings** > **Clock settings** > **Reset type**.

See Also
Multiply-Add | Dot Product

Topics
"Adaptive Pipelining"
"Clock-Rate Pipelining"

Multiply-Add

Multiply-add combined operation
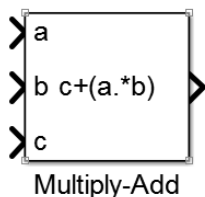
Libraries:
HDL Coder / HDL Operations

Multiply-Add

Description

The Multiply-Add block computes the product of the first two inputs, a and b, and adds the result to the third input, c. The inputs can be vectors or scalars.

The multiplication operation is full precision, regardless of the output type. The **Integer rounding mode**, **Output data type**, and **Saturate on integer overflow** settings apply only to the addition operation.

Use the Multiply-Add block to map a combined multiply-add or a multiply-subtract operation to a DSP unit in your target hardware. You can select the **Function** setting in the Block Parameters dialog box for the Multiply-Add block.

To map to a DSP unit, specify the `SynthesisTool` property for your model. When you generate HDL code for your model, HDL Coder configures the multiply-add operation so that your synthesis tool can map to a DSP unit.

Note Some DSP units do not have the multiply-add capability. To see if your hardware has the multiply-add capability, refer to the documentation for the hardware.

Limitations

The Multiply-Add block accepts and outputs signals of numeric data type that Simulink supports, including fixed-point data types.

You can use matrix data types with the Multiply-Add block. When you use these types, the port dimensions of the inputs a and b must match. For example, in MATLAB, you can perform these matrix operations:

```
a = [1 2; 3 4];
b = [5; 6];
c = 7;

c + (a.*b)

ans =

    12    17
    25    31
```

However, when you perform this multiplication in the Simulink environment, you see an error: `Error in port widths or dimensions`.

See "Data Types Supported by Simulink".

Ports

Input

a — Input signal
`scalar` | `vector` | `matrix`

Input signal to be multiplied with input to Port **b**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

b — Input signal
`scalar` | `vector` | `matrix`

Input signal to be multiplied with input to Port **a**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

c — Input signal
`scalar` | `vector` | `matrix`

Input signal that gets added or subtracted to the product of **a** and **b** depending on **Function** setting.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

Port_1 — Output signal
`scalar` | `vector` | `matrix`

Output data from the multiply-add operation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Function — Specify function

`c+(a.*b)` (default) | `c-(a.*b)` | `(a.*b)-c`

Specify the function to perform a combined multiply and add or a multiply and subtract operation.

Programmatic Use
Parameter: `Function`
Type: string scalar | character vector
Value: `"c+(a.*b)"` | `"c-(a.*b)"` | `"(a.*b)-c"`
Default: `'c+(a.*b)'`

Output data type — Specify output data type

`Inherit: Inherit via internal rule` (default)

Set the output data type to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button `>>` to display the Data Type Assistant dialog box, which helps you to set the **Output data type** parameter.

For more information, see "Control Data Types of Signals" in *Simulink User's Guide* .

Integer rounding mode — Specify rounding mode for fixed-point operations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Specify the rounding mode for fixed-point operations as either:

`Ceiling`

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

`Convergent`

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

`Floor`

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

`Nearest`

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

`Round`

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

`Simplest`

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

`Zero`

Rounds number toward zero. Equivalent to the MATLAB `fix` function.

Programmatic Use
Parameter: RndMeth
Type: string scalar | character vector
Value: "Ceiling" | "Convergent" | "Floor" | "Nearest" | "Round" | "Simplest" | "Zero"
Default: 'Floor'

See Also

For more information, see "Rounding".

Saturate on integer overflow — Specify whether overflows saturate
Off (default) | **On**

☑ On

    Overflows saturate to either the minimum or maximum value that the data type can represent.

    For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

☐ Off

    Overflows wrap to the appropriate value that the data type can represent.

    For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

Tips

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see "Troubleshoot Signal Range Errors".
- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Programmatic Use
Parameter: `SaturateOnIntegerOverflow`
Type: string scalar | character vector
Value: `"off"` | `"on"`
Default: `'off'`

Algorithms

Pipeline Depth

If you have fixed-point inputs to a Multiply-Add block, you can set the **PipelineDepth** for the block. For floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert pipeline registers.

The following diagrams show different configurations of pipeline registers for different synthesis tools and **PipelineDepth** settings. When you specify the **PipelineDepth** setting, HDL Coder inserts pipeline registers so that the configuration maps efficiently to DSP units.

Altera Hardware with PipelineDepth = 1
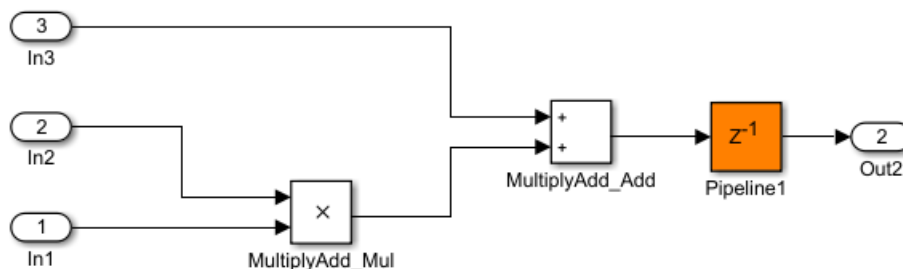
Altera Hardware with PipelineDepth = 2
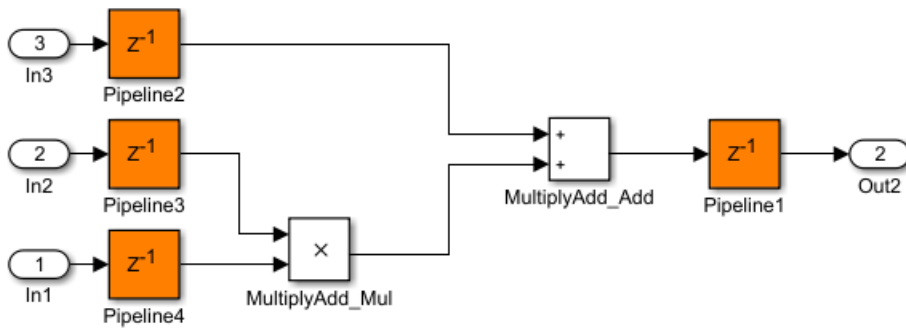
Xilinx Hardware with PipelineDepth = 1
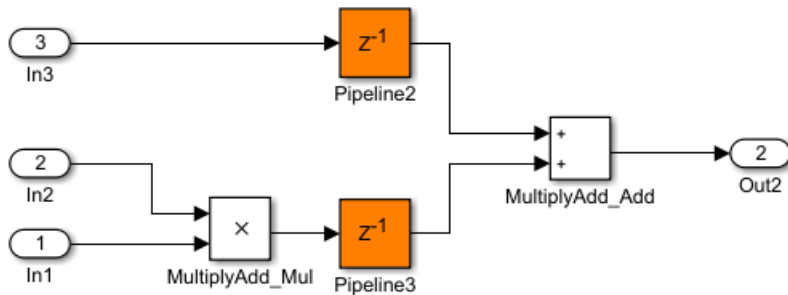
Xilinx Hardware with PipelineDepth = 2
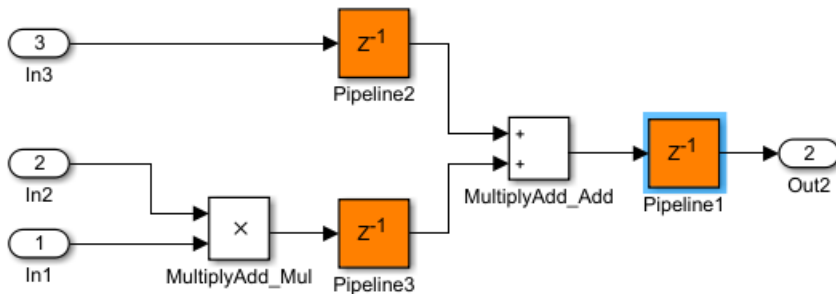
Xilinx Hardware with PipelineDepth = 3
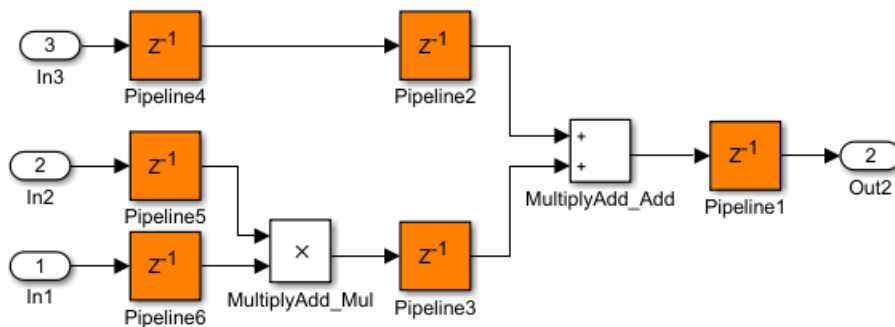
Version History
Introduced in R2015b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For adaptive pipelines to be inserted, when **PipelineDepth** is set to `auto`, you must specify the synthesis tool and enter a target frequency value greater than zero. You can enter an integer between 0 and 3. For Altera hardware targets, the maximum pipeline depth is 2.

General	
HandleDenormals	Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is `inherit`. See also "HandleDenormals".

General	
LatencyStrategy	Specify whether to map the blocks in your design to `inherit`, `Max`, `Min`, or `Zero` for the floating-point operator. The default is `inherit`. See also "LatencyStrategy".
MantissaMultiplyStrategy	Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is `inherit`. See also "MantissaMultiplyStrategy".

Complex Data Support

This block supports code generation for complex signals.

Restrictions

- When the block has floating-point inputs, HDL Coder ignores the **PipelineDepth** parameter and does not insert pipeline registers.

- If the block is in a feedback loop and you do not have sufficient delays at the block output, the coder reduces the **PipelineDepth** to prevent delay balancing failure. For sufficient delays, add Delay blocks at the output of the Multiply-Add block.

- To map the combined multiply-add operation to a DSP unit, the width of the third input *c* has to be less than 64 bits for Altera and 48 bits for Xilinx respectively.

- The subtraction operation in the **Function** setting `(a.*b)-c` does not map to a DSP unit in Altera FPGA libraries.

See Also
Multiply-Accumulate | Dot Product

Topics
"Adaptive Pipelining"
"Clock-Rate Pipelining"

Reciprocal

Implement control signal based reciprocal operation

Description
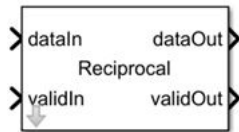
The Reciprocal block performs the reciprocal operation on the input data signal. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

`open_system('HDLMathLib')`

Latency Considerations

You can simulate the Sqrt block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on output data type. The latency of the block is calculated from word-length and fractional-length of fixed-point output.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

The block has following HDL code generation limitations:

* The block does not support floating-point data types, such as `half`, `single`, and `double`.
* Bus type inputs are not supported.

Ports

Input

dataIn — Data input signal
scalar | vector

Data input to calculate the reciprocal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: Boolean

Output

dataOut — Output data signal
scalar | vector

Output signal that is the reciprocal of the input signal.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: Boolean

Parameters

Latency strategy — Select the latency strategy

Max (default) | Custom | Zero

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: latencyMode
Type: character vector
Values: 'Max' | 'Custom' | 'Zero'
Default: 'Max'

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to Custom, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, L], where L is the maximum latency value of Reciprocal block. For more information, see "CustomLatency".

Dependency

To use this parameter, set **Latency strategy** to Custom.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: 0 to Max latency
Default: 0

Output data type — Select the output data type of the block

Inherit: Inherit via internal rule (default) | Inherit: Same as first input |
Inherit: Inherit via back propagation | int8 | uint8 | int16 | uint16 | int32 | uint32 |
int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the output data type. The data type can be inherited or specified directly.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back
propagation' | 'Inherit: Same as first input' | 'int8' | 'uint8' | int16 | 'uint16' |
'int32' | 'uint32' | 'int64' | 'uint64' | fixdt(1,16,0) | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional
configuration options that affect HDL implementation and synthesized logic. For more information,
see HDL Block Properties.

Version History
Introduced in R2020b

See Also
Math Function | Divide

Resettable Synchronous Subsystem

Represent resettable subsystem that has synchronous reset and enable behavior

Resettable Synchronous
Subsystem
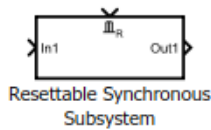
Libraries:
HDL Coder / HDL Subsystems

Description

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.
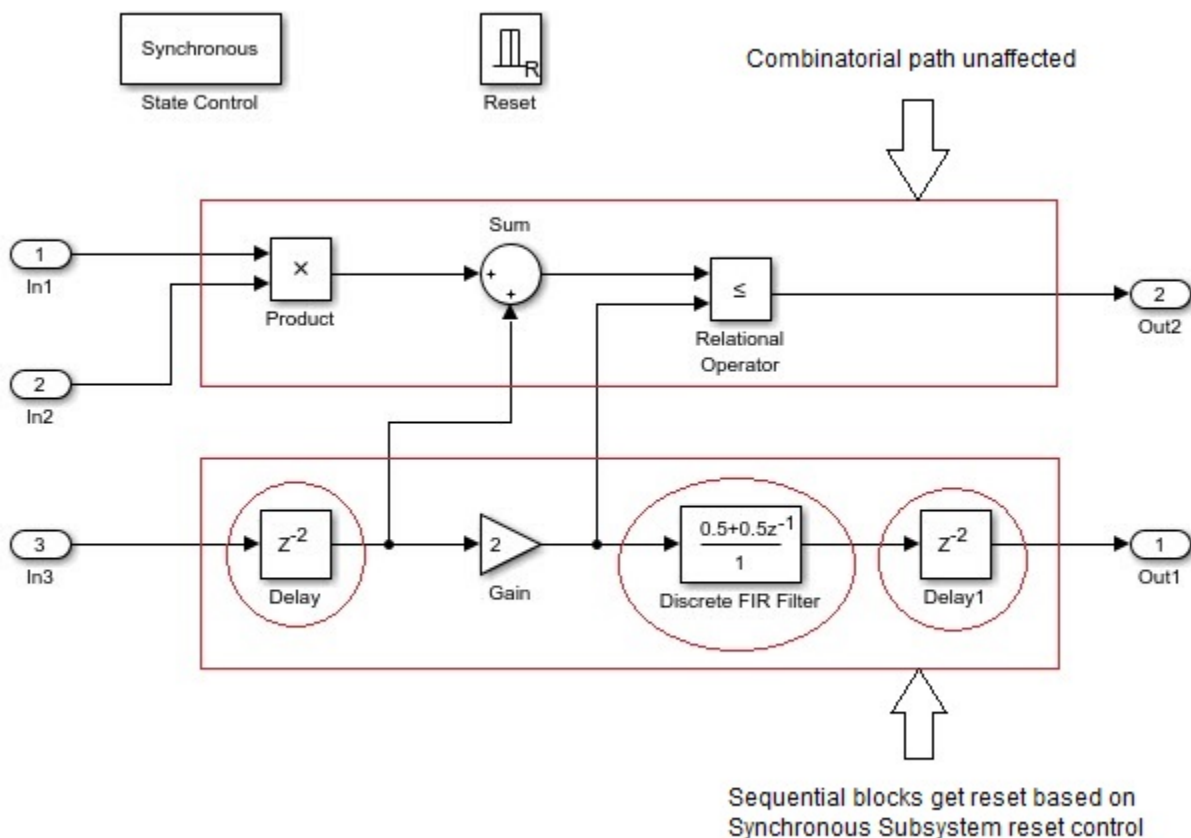
The reset port in the Resettable Synchronous Subsystem block adds reset capability to blocks inside the subsystem that have state. This includes blocks that need not have an external reset port capability, such as filters, Stateflow® Chart, and MATLAB Function blocks. For HDL code generation, the **Reset trigger type** of the Reset port is set to level hold by default.

Ports

Input

Port_1 — Input Signal to subsystem
scalar | vector

See Inport for information on the data types accepted by a subsystem's input ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

Port_1 — Output Signal from subsystem
scalar | vector

See Outport for information on the data types output by a subsystem's output ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Show port labels — Display options for port labels

`FromPortIcon` (default) | `FromPortBlockName` | `SignalName` | `none`

Select how to display port labels on the Synchronous Subsystem block icon.

`none`

> Do not display port labels.

`FromPortIcon`

> If the corresponding port icon displays a signal name, display the signal name on the Subsystem block. Otherwise, display the port block name or the port number if the block name is a default name.

`FromPortBlockName`

> Display the name of the corresponding port block on the Subsystem block.

`SignalName`

> If the signal connected to the port is named, display the name of the signal on the Subsystem block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: `ShowPortLabels`
Type: character vector
Value: `'FromPortIcon'` | `'FromPortBlockName'` | `'SignalName'` | `'none'`
Default: `'FromPortIcon'`

Read/Write permissions — Levels of access to contents of subsystem

ReadWrite (default) | ReadOnly | NoReadOrWrite

Control user access to the contents of the subsystem.

Settings

Default: ReadWrite

ReadWrite

Enables opening and modification of subsystem contents.

ReadOnly

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

NoReadOrWrite

Disables opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

Programmatic Use
Parameter: Permissions
Type: character vector
Value: 'ReadWrite' | 'ReadOnly' | 'NoReadOrWrite'
Default: 'ReadWrite'

Name of error callback function — Name of function to be called if error occurs

'' (default) | function name

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

Programmatic Use
Parameter: ErrorFcn
Type: character vector
Value: '' | '<function name>'
Default: ''

Permit hierarchical resolution — Resolution for workspace variable names

All (default) | ExplicitOnly | None

Select whether to resolve names of workspace variables referenced by this subsystem.

For more information, see "Symbol Resolution" and "Symbol Resolution Process".

All

Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, `Simulink.Signal` objects).

ExplicitOnly

Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as "must resolve".

None

Do not resolve workspace variable names.

Programmatic Use
Parameter: `PermitHierarchicalResolution`
Type: character vector
Value: `'All'` | `'ExplicitOnly'` | `'None'`
Default: `'All'`

Function packaging — Code format

Auto (default) | Inline | Nonreusable function | Reusable function

Select the code format to be generated for an atomic (nonvirtual) subsystem.

Auto

Simulink Coder chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

Inline

Simulink Coder inlines the subsystem unconditionally.

Nonreusable function

Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

Reusable function

Simulink Coder software generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

Programmatic Use
Parameter: `RTWSystemCode`
Type: character vector
Value: `'Auto'` | `'Inline'` | `'Nonreusable function'` | `'Reusable function'`
Default: `'Auto'`

Version History

Introduced in R2016b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

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. For more details, see "ConstrainedOutputPipeline".
DistributedPipelining	Pipeline register distribution, or register retiming. The default is inherit. 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".

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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".

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.

Target Specification	
AdditionalTargetInterfaces	Additional target interfaces, specified as a character vector. To save this block property on the model, in the **Set Target Interface** task of the **IP Core Generation** workflow, corresponding to the DUT ports that you want to add more interfaces, select **Add more...**. You can then add more interfaces in the Add New Target Interfaces dialog box. Specify the type of interface, number of additional interfaces, and a unique name for each additional interface. Values: `' '` (default) | cell array of character vectors Example: `'{{'AXI4-Stream','InterfaceID','AXI4-Stream1'}}'`
ProcessorFPGASynchronization	Processor/FPGA synchronization mode, specified as a character vector. To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `Free running` (default) | `Coprocessing - blocking` Example: `'Free running'`
TestPointMapping	To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `' '` (default) | cell array of character vectors Example: `'{{'TestPoint','AXI4-Lite','x"108"'}}'`

Target Specification	
TunableParameterMapping	To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `''` (default) | cell array of character vectors Example: `'{{'myParam','AXI4-Lite','x"108"'}}'`
AXI4RegisterReadback	To save this block property on the model, specify whether you want to enable readback on AXI4 subordinate write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'`
AXI4SlaveIDWidth	To save this block property on the model, specify the number of AXI manager interfaces that you want to connect the DUT IP core to by using the **AXI4 Slave ID Width** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface". Values: `'off'` (default) | `'on'`
AXI4SlavePortToPipelineRegisterRatio	To save this block property on the model, specify the number of AXI4 subordinate ports for which you want a pipeline register to be inserted by using the **AXI4 Slave port to pipeline register ratio** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'``'10'``'20'``'35'``'50'`
GenerateDefaultAXI4Slave	To save this block property on the model, specify whether you want to disable generation of default AXI4 subordinate interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. Values: `'on'` (default) | `'off'`
IPCoreAdditionalFiles	Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;). You can set this property in the HDL Workflow Advisor, in the **Additional source files** field. Values: `''` (default) | character vector Example: `'C:\myprojfiles\led_blinking_file1.vhd;C:\myprojfiles\led_blinking_file2.vhd;'`

Target Specification	
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'`
IPDataCaptureBuffer Size	FPGA Data Capture buffer size, specified as a character vector. Use FPGA Data Capture to observe signals in a design when running on an FPGA. The buffer size uses values that are 128*2^n, where n is an integer. By default, the buffer size is 128 (n=0). The maximum value of n is 13, which means that the maximum value for buffer size is 1048576 (=128*2^13). 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 code generation supports only boolean datatype at the Reset port.

- HDL Coder does not support these blocks inside a Resettable Synchronous Subsystem:

  - Stateflow Charts

  - All RAM blocks or blocks that infer a RAM in the generated HDL code. The RAM blocks include:

    - Single Port RAM

    - Simple Dual Port RAM

    - Dual Port RAM

    - Dual Rate Dual Port RAM

    - HDL FIFO

- `hdl.RAM` system object

DSP System Toolbox

- Biquad Filter
- NCO HDL Optimized

Communications Toolbox

- Convolutional Encoder
- Viterbi Decoder
- PN Sequence Generator
- Integer-Output RS Decoder HDL Optimized

Vision HDL Toolbox

- Demosaic Interpolator
- Edge Detector
- Histogram
- Image Filter, Median Filter, Bilateral Filter
- Line Memory
- Binary and Grayscale Morphology blocks
- Pixel Stream FIFO

Wireless HDL Toolbox

- LTE Turbo Decoder and LTE Turbo Encoder
- LTE Convolutional Encoder
- LTE OFDM Demodulator and LTE OFDM Modulator
- NR Polar Encoder and NR Polar Decoder
- Viterbi Decoder
- FFT 1536
- RS Decoder
- OFDM Channel Estimator
- NR LDPC Encoder and NR LDPC Decoder

See Also

State Control | Enabled Synchronous Subsystem | 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"

rSqrt

Implement control signal based reciprocal square-root function

Description
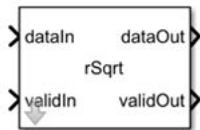
The rSqrt block performs the reciprocal square-root operation on the input data signal. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the Sqrt block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on **Number of iterations**.

The latency *L* of rSqrt block with `RecipSqrtNewtonSingleRate` architecture is given as,

$$L = (N * 4) + 5$$

where *N* is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

The rSqrt block has following HDL code generation limitations:

- The block does not support floating-point data types, such as `half`, `single`, and `double`.
- The block does not support bus inputs.

Ports

Input

dataIn — Input data signal
scalar | vector

Input signal to calculate the reciprocal square root.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

dataOut — Output data signal
scalar | vector

Output signal that is the reciprocal square root of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Architecture — Architecture used

`RecipSqrtNewtonSingleRate` (default)

Select the architecture for rSqrt block.

Programmatic Use
Block Parameter: `architecture`
Type: character vector
Values: `RecipSqrtNewtonSingleRate`
Default: `'RecipSqrtNewtonSingleRate'`

Number of iterations — Number of iterations used

3 (default) | `Integer`

Specify the number of iterations for rSqrt algorithm.

Programmatic Use
Block Parameter: `numOfIterations`
Type: character vector
Values: `Integer values`
Default: `'3'`

Latency strategy — Latency strategy

Max (default) | Custom | Zero

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: latencyMode
Type: character vector
Values: 'Max' | 'Custom' | 'Zero'
Default: 'Max'

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to Custom, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, *L*], where *L* is the maximum latency value of rSqrt block. For more information, see "CustomLatency".

Dependency

To use this parameter, set **Latency strategy** to Custom.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: 0 to Max latency
Default: 0

Output data type — Select the output data type of the block

Inherit: Inherit via internal rule (default) | Inherit: Same as first input | Inherit: Inherit via back propagation | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the output data type. The data type can be inherited or specified directly.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'Inherit: Same as first input' | 'int8' | 'uint8' | int16 | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | fixdt(1,16,0) | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Saturate on integer overflow — Choose the behavior when integer overflow occurs

off (default) | on

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Troubleshoot Signal Range Errors".	Overflows wrap to the value that is representable by the data type.	The maximum value that the `int8` (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as `int8`, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as `int8`, is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Programmatic Use
Block Parameter: `SaturateOnIntergerOverflow`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Integer rounding mode — Rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see "Rounding".

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Default: `'Floor'`

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Version History
Introduced in R2020b

See Also
Math Function | Sqrt

Serializer1D

Convert vector signal to scalar or smaller vectors
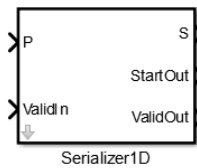
Libraries:
HDL Coder / HDL Operations

Description

The Serializer1D block converts a slower vector signal into a faster stream of scalar signals or smaller size vector signals based on the **Ratio** and **Idle Cycle** values. To match the faster serialized output, the sample time changes according to this equation:

Output Sample Time = Input Sample Time/(Ratio + Idle Cycles)

Consider this example where the input data is a vector of size 4 and the **Ratio** is set to 4.
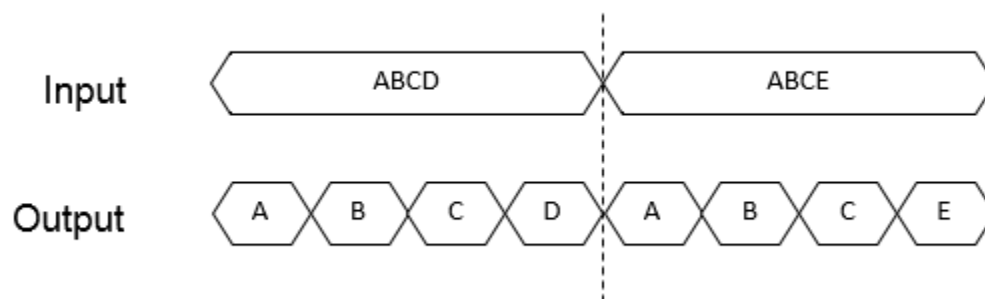
The output data serializes each of the vector signals into four scalar signals. The sample time at the output is: *Output Sample Time = Input Sample Time/4*.

To add idle cycles at the end of each output, for **Idle Cycles**, specify an integer greater than zero. Consider this example with **Ratio** set to 4 and **Idle Cycles** set to 2.
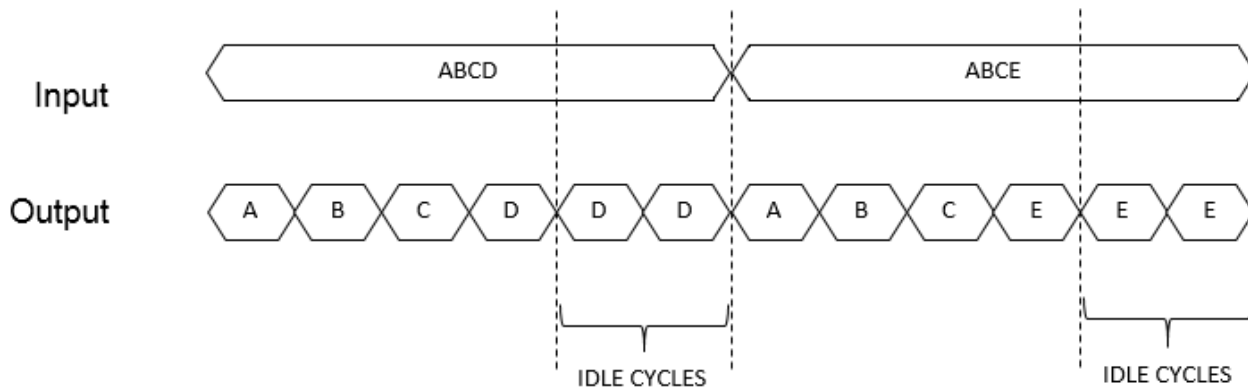
For each slow vector signal, the output has six fast cycles consisting of the four serialized scalar signals and two idle cycles. The sample time at the output is
Output Sample Time = Input Sample Time/6.

The Serializer1D block provides three control signals: **ValidIn**, **ValidOut**, and **StartOut**. You can use **ValidIn** to control **ValidOut** and **StartOut**. The serialized output does not depend on **ValidIn**. To determine whether the output serialized data is valid, use **ValidIn** and **ValidOut**. If you give a high input to **ValidIn**, and there are no idle cycles, **ValidOut** gives a high output, which indicates that the output serialized data is valid.

Consider an example that has input data as a vector of size 4, **Ratio** set to 4, **Idle Cycles** set to 2, and uses all three control signals.
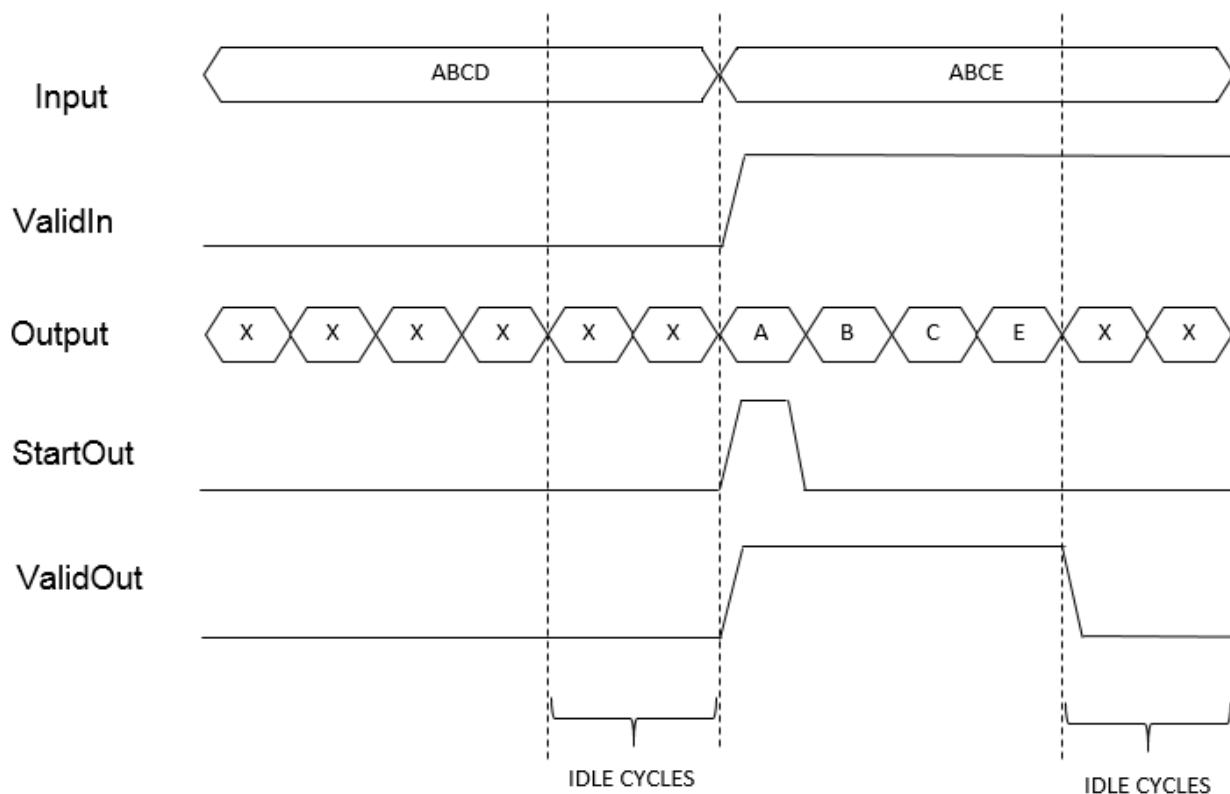
For the first input vector, ABCD, **ValidIn** is false. **StartOut** and **ValidOut** become false. This means that the output data values are not valid. In the waveform, the data values are represented as X, which correspond to *don't care* values.

For the second input vector, ABCE, **ValidIn** is true. The output data serializes the vector into four scalar signals. The control signal **StartOut** becomes true at output A to indicate the start of serialization. In the next cycle, the **StartOut** signal becomes false. **ValidOut** is true for all four output signals indicating valid output data for the four cycles. **ValidOut** becomes false for the idle cycles, and the output data values are *don't care* values.

Ports

Input

P — Parallel data input signal
scalar | vector

Input signal to serialize. Bus data types are not supported.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated

ValidIn — Valid input control signal
scalar

Input control signal that indicates valid input signal. This port is available when you select the **ValidIn** check box.

Data Types: `Boolean`

Output

S — Serialized data output signal
`scalar | vector`

Serialized output signal. Bus data types are not supported.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated`

StartOut — Start out control signal
`scalar`

Output control signal that indicates the start of serialization. You can use this signal as the **StartIn** input to the Deserializer1D block. To use this port, select the **StartOut** check box.

Data Types: `Boolean`

ValidOut — Valid output control signal
`scalar`

Output control signal that indicates valid output signal. You can use this signal as the **ValidIn** input to the Deserializer1D block. This port is available when you select the **ValidOut** check box.

Data Types: `Boolean`

Parameters

Ratio — Ratio of input vector size to output vector size
1 (default)

Serialization factor, specified as a positive scalar.

The ratio is equal to the size of the input vector divided by the size of the output vector. Input vector size must be divisible by the ratio.

Idle Cycles — Number of idle cycles
0 (default)

Number of idle cycles to add at the end of each output.

ValidIn

Activates the **ValidIn** port. Default is `off`.

StartOut

Activates the **StartOut** port. Default is `off`.

ValidOut

Activates the **ValidOut** port. Default is `off`.

Input data port dimensions (-1 for inherited) — Size of the input data signal
-1 (default)

Size of the input data signal. Input vector size must be divisible by the ratio. By default, the block inherits size based on the context within the model.

Input sample time (-1 for inherited) — Input sample time
-1 (default)

Time interval between sample time hits, or another appropriate sample time such as continuous. By default, the block inherits sample time based on context within the model. For more information, see "Sample Time".

Input signal type — Input signal type of the block
auto (default) | real | complex

Input signal type of the block, specified as `auto`, `real`, or `complex`. Default is `auto`.

Version History
Introduced in R2014b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

Note For simulation results that match the generated HDL code, in the Solver pane of the Configuration Parameters dialog box, clear the checkbox for **Treat each discrete rate as a separate task**. When the checkbox is cleared, single-tasking mode is enabled. If you simulate the block with this check box selected, the output data can update in the same cycle but in the generated HDL code, the output data is updated one cycle later.

This block has one 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. For more details, see "ConstrainedOutputPipeline".

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Deserializer1D

Simple Dual Port RAM

Dual port RAM with single output port

Libraries:
HDL Coder / HDL RAMs

Description

The Simple Dual Port RAM block models RAM that supports simultaneous read and write operations, and has a single output port for read data. You can use this block to generate HDL code that maps to RAM in most FPGAs.

The Simple Dual Port RAM is similar to the Dual Port RAM, but the Dual Port RAM has both a write data output port and a read data output port.

Read-During-Write Behavior

During a write operation, if a read operation occurs at the same address, old data appears at the output.

Ports

Input

wr_din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when wr_en is true. The data inherits the width and data type from the input signal. wr_din can be a `double`, `single`, `integer`, or a `fixed-point` (fi) object, and can be real or complex.

Data type: scalar fixed point, integer, or complex

Data Types: `int8` | `int16` | `int32` | `int64` | `fixed point`

wr_addr — Write address
Scalar (default) | Vector

Write address.

Address that you write the data into when wr_en is true. This value can be either `fixed-point(fi)` or `integer`, must be unsigned, and have a fraction length of 0.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

3-123

wr_en — Write enable
Scalar (default) | Vector

When wr_en is true, the RAM writes the data into the memory location that you specify.

Data Types: Boolean

rd_addr — Read address
Scalar (default) | Vector

Address that you read the data from. This value can be either fixed-point(fi) or integer, must be unsigned, and have a fraction length of 0.

Data Types: uint8 | uint16 | uint32 | uint64 | fixed point

Output

rd_dout — Output data from read address
Scalar (default) | Vector

Output data from read address, rd_addr.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Parameters

Address port width — Address bit width
8 (default)

Minimum bit width is 2, and maximum bit width is 29.

Programmatic Use
Block parameter: ram_size
Type: string scalar | character vector
Value: A minimum value of 2 and maximum value of 29
Default: '8'

Algorithms

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 3-41.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to distributed RAMs, block RAMs, or UltraRAM memory on the target FPGA. See also "RAMDirective".

Complex Data Support

This block supports code generation for complex signals.

See Also

Blocks
Single Port RAM | Dual Port RAM | Dual Rate Dual Port RAM

Simple Dual Port RAM System

Simple Dual Port RAM block based on the hdl.RAM system object with ability to provide initial value

Libraries:
HDL Coder / HDL RAMs

Description

The blocks are MATLAB System blocks that use the `hdl.RAM` System object. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Dual Port RAM System block behaves similar to the Dual Port RAM, the Single Port RAM System behaves similar to the Single Port RAM, and so on. With the MATLAB System blocks, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.

- Obtain faster simulation results when you use these blocks in your Simulink model.

- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.

- Obtain higher performance and support for large data memories.

Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.

- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

- The RAM write address can be either `fixed-point (fi)` or `integer`, must be unsigned, and must be between 2 and 31 bits long.

Ports

Input

din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

3-127

wr_addr — Write address
Scalar (default) | Vector

RAM address that you write the data into. This value can be either `fixed-point (fi)` or `integer`, must be unsigned, and must be between 2 and 31 bits long.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

wr_en — Write enable
Scalar (default) | Vector

When `wrEn` is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to `Single port`, the RAM reads the value in the memory location `addr` when `wrEn` is false.

Data Types: `Boolean`

rd_addr — Read address
Scalar (default) | Vector

Address that you read the data from the RAM. This value can be either `fixed-point (fi)` or `integer`, and must be real and unsigned.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Data Types: `uint8` | `uint16` | `fixed point`

Output

rd_dout — Read data
Scalar (default) | Vector

Old output data that the RAM reads from the memory location `rd_addr`.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to `Simple dual port` or `Dual port`.

Parameters

Specify the type of RAM — RAM type
Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- `Single port` — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.

- `Simple dual port` — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- `Dual port` — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

Specify the output data for a write operation — Write output behavior
`New data` (default) | `Old data`

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `Old data'` — Send out old data at the address to the output.

Specify the RAM initial value — Initial simulation output of RAM
`'0.0'` (default) | `Scalar` | `Vector`

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM` System object.

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. For more details, see "ConstrainedOutputPipeline".

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to RAM blocks on the target FPGA. For UltraRAM mapping, **Specify the RAM initial value** must be set to 0. For more details, see "RAMDirective".

Complex Data Support

This block supports code generation for complex signals.

See Also

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"

Single Port RAM

Single port RAM
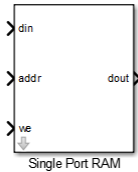
Libraries:
HDL Coder / HDL RAMs

Description

The Single Port RAM block models RAM that supports sequential read and write operations.

If you want to model RAM that supports simultaneous read and write operations, use the Dual Port RAM block or Simple Dual Port RAM block.

Ports

Input

din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when we is true. The data inherits the width and data type from the input signal. din can be a double, single, integer, or a fixed-point (fi) object, and can be real or complex.

Data Types: single | double | int8 | int16 | uint8 | uint16 | fixed point

addr — Write address
Scalar (default) | Vector

Address that you write the data into when we is true. This value can be either fixed-point(fi) or integer, must be unsigned, and have a fraction length of 0.

Data Types: uint8 | uint16 | fixed point

we — Write enable
Scalar (default) | Vector

When we is true, the RAM writes the data into the memory location that you specify.

Data Types: Boolean

Output

dout — Output data
Scalar (default) | Vector

Output data from address, `addr`.

Parameters

Address port width — Address bit width
8 (default)

Minimum bit width is 2, and maximum bit width is 29.

Programmatic Use
Block parameter: `ram_size`
Type: string scalar | character vector
Value: A minimum value of 2 and maximum value of 29
Default: `'8'`

Output data during write — Controls output data
New data (default) | Old data

Controls the output data, `dout`, during a write access. Specified as either:

- `New data`—During a write, new data appears at the output port, `dout`.
- `Old data`— During a write, old data appears at the output port, `dout`.

Programmatic Use
Block parameter: `dout_type`
Type: string scalar | character vector
Value: `"New data"` | `"Old data"`
Default: `'New data'`

Algorithms

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 3-132.

Version History
Introduced in R2014a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to distributed RAMs, block RAMs, or UltraRAM memory on the target FPGA. See also "RAMDirective".

Complex Data Support

This block supports code generation for complex signals.

See Also

Blocks
Simple Dual Port RAM | Dual Port RAM | Dual Rate Dual Port RAM

Single Port RAM System

Single Port RAM block based on hdl.RAM system object with ability to provide initial value
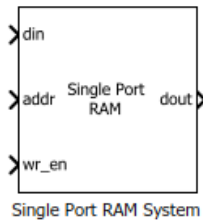
Libraries:
HDL Coder / HDL RAMs

Description

The blocks are MATLAB System blocks that use the `hdl.RAM` System object. You can specify the RAM type as `Dual port`, `Simple dual port`, or `Single port`. In terms of simulation behavior, the Single Port RAM System block behaves similar to the Single Port RAM.

By using the MATLAB System block implementation, you can:

- Specify an initial value for the RAM. In the Block Parameters dialog box, enter a value for **Specify the RAM initial value**.

- Obtain faster simulation results when you use these blocks in your Simulink model.

- Create parallel RAM banks when you use vector data by leveraging the `hdl.RAM` System object functionality.

- Obtain higher performance and support for large data memories.

Limitations

- The block does not support `boolean` inputs. Cast any `boolean` types to `ufix1` for input to the block.

- When you build the FPGA bitstream for the RAM, the global reset logic does not reset the RAM contents. To reset the RAM, make sure that you implement the reset logic.

- The RAM address can be either `fixed-point (fi)` or `integer`, must be unsigned, and must be between 2 and 31 bits long.

Ports

Input

din — Write data input
Scalar (default) | Vector

Data that you write into the RAM memory location when `wrEn` is true. This value can be `double`, `single`, `integer`, or a `fixed-point (fi)` object, and can be real or complex.

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fixed point`

addr — Write or Read address
Scalar (default) | Vector

Address that you write the data into when wrEn is true. The RAM reads the value in memory location **addr** when wrEn is false. This value can be either fixed-point (fi) or integer, must be unsigned, and must be between 2 and 31 bits long.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to Single port.

Data Types: uint8 | uint16 | fixed point

wr_en — Write enable
Scalar (default) | Vector

When wrEn is true, the RAM writes the data into the memory location that you specify. If you set the **Specify the type of RAM** to Single port, the RAM reads the value in the memory location addr when wrEn is false.

Data Types: Boolean

Output

dout — Output data
Scalar (default) | Vector

Output data that the RAM reads from the memory location addr when wrEn is false.

Dependencies

To enable this port, set the **Specify the type of RAM** parameter to Single port.

Parameters

Specify the type of RAM — RAM type
Dual port (default) | Simple dual port | Single port

Type of RAM, specified as either:

- Single port — Create a single port RAM with Write data, Address, and Write enable as inputs and Read data as the output.
- Simple dual port — Create a simple dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address as the output.
- Dual port — Create a dual port RAM with Write data, Write address, Write enable, and Read address as inputs and data from read address and write address as the outputs.

The code generator dynamically configures the input and output ports of the block based on the RAM type that you specify.

Specify the output data for a write operation — Write output behavior
New data (default) | Old data

Behavior for Write output, specified as either:

- `'New data'` — Send out new data at the address to the output.
- `Old data'` — Send out old data at the address to the output.

Specify the RAM initial value — Initial simulation output of RAM
`'0.0'` (default) | `Scalar` | `Vector`

Initial simulation output of the System object, specified as either:

- A scalar value.
- A vector with one-to-one mapping between the initial value and the RAM words.

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

The block has a `MATLABSystem` architecture which indicates that the block implementation uses the `hdl.RAM` System object.

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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
RAMDirective	Specify whether to map RAM blocks in your design to RAM blocks on the target FPGA. For UltraRAM mapping, **Specify the RAM initial value** must be set to 0. See also "RAMDirective".

Complex Data Support

This block supports code generation for complex signals.

See Also

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"

Sin

Implement control signal based sine function

Description
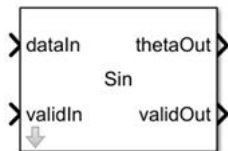
The Sin block computes the sine of input signal. The block uses the coordinate rotation digital computer (CORDIC) approximation method. For more information, see CORDIC approximation method in "Algorithms" on page 3-141. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

open_system('HDLMathLib')

Latency Considerations

You can simulate the Sin block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on the **Number of iterations**.

The maximum latency L of Sin block is given by this equation:

$L = N + 1$

where N is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

- The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dataIn — Input data signal
scalar | vector

Input data signal to compute sine function. The input value ranges from -2π to 2*π.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

thetaOut — Output data signal
scalar | vector

Output data signal that is the sine of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Number of iterations — Number of iterations for CORDIC algorithm

11 (default) | `Positive integer`

Specify the number of iterations for CORDIC algorithm.

Programmatic Use
Block Parameter: `iter`
Type: character vector
Values: `Integer values`
Default: `'11'`

Latency strategy — Latency strategy

Max (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: `latencyMode`
Type: character vector
Values: `'Max'` | `'Custom'` | `'Zero'`
Default: `'Max'`

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to `Custom`, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, *L*], where *L* is the maximum latency value of Sin block. For more information, see "CustomLatency".

Dependency

To enable this parameter, set **Latency strategy** to `Custom`.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: `0 to Max latency`
Default: `0`

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Version History
Introduced in R2020b

See Also
Trigonometric Function | Sine, Cosine

SinCos

Implement control signal based sine and cosine function

Description
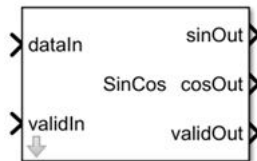
The SinCos block computes the sine and cosine of input signal. The block uses the coordinate rotation digital computer (CORDIC) approximation method. For more information, see CORDIC approximation method in "Algorithms" on page 3-144. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the SinCos block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on the **Number of iterations**.

The maximum latency L of SinCos block is given by this equation:

$L = N + 1$

where N is the value of the **Number of iterations** parameter.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

- The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dataIn — Input data signal
scalar | vector

Input data signal to compute sine and cosine function. The input value ranges from -2π to 2*π.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: `Boolean`

Output

sinOut — Sine output data signal
scalar | vector

Output data signal that is the sine of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

cosOut — Cosine output data signal
scalar | vector

Output data signal that is the cosine of the input signal.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: `Boolean`

Parameters

Number of iterations — Number of iterations for CORDIC algorithm

11 (default) | `Positive Integer`

Specify the number of iterations for CORDIC algorithm.

Programmatic Use
Block Parameter: `iter`
Type: character vector
Values: `Integer values`
Default: `'11'`

Latency strategy — Latency strategy

Max (default) | `Custom` | `Zero`

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: latencyMode
Type: character vector
Values: 'Max' | 'Custom' | 'Zero'
Default: 'Max'

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to Custom, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, *L*], where *L* is the maximum latency value of SinCos block. For more information, see "CustomLatency".

Dependency

To enable this parameter, set **Latency strategy** to Custom.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: 0 to Max latency
Default: 0

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Version History
Introduced in R2020b

See Also
Trigonometric Function | Sine, Cosine

Sqrt

Implement control signal based square root function

Description
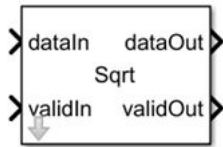
The Sqrt block performs the square-root operation of the input data signal. The block has control signals that indicate whether the input and output data are valid. You can also specify the number of iterations of the algorithm and the latency strategy.

To use this block in your Simulink model, open the `HDLMathLib` library by entering this command in the MATLAB Command Window:

```
open_system('HDLMathLib')
```

Latency Considerations

You can simulate the Sqrt block with latency. This block is a masked subsystem that contains a MATLAB Function block, `LumpLatency`. The subsystem uses this MATLAB Function block to compute the latency based on input and output data type. The latency of the block is calculated from word-length and fractional-length of fixed-point input and output.

The block supports three latency modes. You can specify a custom latency value by setting the **Latency Strategy** parameter to `Custom`.

To view the function that computes the latency of the block, open the `LumpLatency` block in the masked subsystem. To view inside the mask, click the ⇩ icon on the block.

Limitations

- The block does not support floating-point data types, such as `half`, `single`, and `double`.

Ports

Input

dataIn — Input data signal
scalar | vector

Input signal to calculate the square root.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `bus`

validIn — Indication of whether input control signal is valid
scalar

Input control signal that indicates whether the input signal is valid.

Data Types: Boolean

Output

dataOut — Output data signal
scalar | vector

Output signal that is the square root of the input signal.

Data Types: int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | bus

validOut — Indication of whether output control signal is valid
scalar

Output control signal that indicates whether output signal is valid.

Data Types: Boolean

Parameters

Latency strategy — Select the latency strategy

Max (default) | Custom | Zero

Specify whether to map the blocks in your design to maximum, custom, or zero latency. For more information, see "LatencyStrategy".

Programmatic Use
Block Parameter: latencyMode
Type: character vector
Values: 'Max' | 'Custom' | 'Zero'
Default: 'Max'

Custom latency — Specify the custom latency value

0 (default)

When you set **Latency strategy** to Custom, use this parameter to specify the custom latency value. The latency must be a nonnegative integer in the range [0, L], where L is the maximum latency value of Sqrt block. For more information, see "CustomLatency".

Dependency

To use this parameter, set **Latency strategy** to Custom.

Programmatic Use
Block Parameter: customLatencyValue
Type: Integer
Values: 0 to Max latency
Default: 0

Output data type — Select the output data type of the block

```
Inherit: Same as first input (default) | Inherit: Inherit via internal rule |
Inherit: Inherit via back propagation | int8 | uint8 | int16 | uint16 | int32 | uint32 |
int64 | uint64 | fixdt(1,16,0) | <data type expression>
```

Specify the output data type. The data type can be inherited or specified directly.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'Inherit: Same as first input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | fixdt(1,16,0) | '<data type expression>'
Default: 'Inherit: Same as first input'

Saturate on integer overflow — Choose the behavior when integer overflow occurs

off (default) | on

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow, and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Troubleshoot Signal Range Errors".	Overflows wrap to the value that is representable by the data type.	The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Programmatic Use
Block Parameter: SaturateOnIntergerOverflow
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations

Zero (default) | Ceiling | Floor | Nearest | Simplest

Specify the rounding mode for fixed-point operations. For more information, see "Rounding".

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Zero' | 'Ceiling' | 'Floor' | 'Nearest' | 'Simplest'
Default: 'Zero'

HDL Code Generation

The block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information, see HDL Block Properties.

Version History
Introduced in R2020b

See Also
Math Function | Sqrt

State Control

Specify synchronous reset and enable behavior for blocks with state
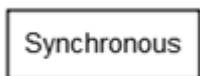
Synchronous	**Libraries:** HDL Coder / HDL Subsystems

Description

Use the State Control block to toggle subsystem behavior between the default Simulink simulation behavior and the synchronous hardware simulation behavior.

- For default Simulink simulation behavior, set **State control** to `Classic`. The simulation behavior in `Classic` mode is the same as when you do not use the State Control block inside the subsystem.

- For synchronous hardware simulation behavior, set **State control** to `Synchronous`. The State Control block in `Synchronous` mode improves the HDL simulation behavior of blocks with state, or blocks that have reset or enable ports. When use the `Synchronous` mode of the block, the generated HDL code uses fewer hardware resources and the Simulink simulation behavior closely matches that of the digital hardware.

See "Synchronous Subsystem Behavior with the State Control Block".

Limitations

Subsystem-level Limitations

- Conditional subsystems using classic semantics cannot have subsystems with synchronous semantics inside them.
- You cannot flatten a synchronous subsystem up into a classic system.
- Conditional subsystems must be single rate when you use the State Control block in synchronous mode.
- Synchronous Enabled Subsystem cannot contain reset subsystems or a reset parameter port. For example, you cannot have a Delay block with an external reset port inside the subsystem.
- All action subsystems connected to If and Switch Case blocks must have the same semantics, either classic or synchronous.
- These blocks are not supported in synchronous mode:

  - For Iterator Subsystem
  - While Iterator Subsystem
  - Function-Call Subsystem
  - Triggered Subsystem

Model-Level Limitations

- Variable-size signals are not supported with synchronous semantics.

- Synchronous semantics do not propagate across model boundaries. If your parent model has synchronous semantics, any referenced model must have synchronous semantics explicitly specified. At the root level of each referenced model, add a State Control block with the **State control** parameter set to `Synchronous`.

Supported Block Modes

The following restrictions apply to blocks in synchronous mode:

- Delay block: When you have an external reset port, set the **External reset** to `Level hold`.
- The method `ssSetStateSemanticsClassicAndSynchronous` must be set to `true`.
- Stateflow Chart: Set the **State Machine Type** to `Moore`.
- MATLAB Function block:

  - You cannot have System objects inside the MATLAB Function block.
  - If you use nondirect feedthrough in a MATLAB Function block, do not program the outputs to rely on inputs or updated persistent variables. The MATLAB Function block must drive the outputs from persistent variables.

    To use nondirect feedthrough, in the Property Inspector, clear the **Allow direct feedthrough** check box. See "Use Nondirect Feedthrough in a MATLAB Function Block".

Unsupported Blocks

The following blocks are not allowed in synchronous mode:

- The set of unit delay blocks in the **Additional Math & Discrete** > **Additional Discrete** sublibrary in Simulink, such as the Unit Delay Resettable and Unit Delay External IC blocks
- Simulink blocks with **Input processing** set to `Columns as channels (frame based)`, where this parameter applies.
- Continuous time blocks and blocks with continuous rate
- Discrete-Time Integrator with reset port
- From Workspace
- Trigger
- LMS Filter
- HDL Minimum Resource FFT
- DC Blocker
- PN Sequence Generator
- Convolutional Interleaver and Convolutional Deinterleaver
- General Multiplexed Interleaver and General Multiplexed Deinterleaver
- Convolutional Encoder and Viterbi Decoder
- Sample and Hold

Parameters

State control — Select the state control type
Synchronous (default) | Classic

Specify whether to use synchronous or classic semantics. The default is `Synchronous`.

Version History
Introduced in R2016a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

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.

See Also

Blocks
Enable | Enabled Subsystem | Enabled Synchronous Subsystem | Resettable Synchronous Subsystem

Topics
"Using the State Control Block to Generate More Efficient Code with HDL Coder"
"Resettable Subsystem Support in HDL Coder"
"Synchronous Subsystem Behavior with the State Control Block"

Synchronous Subsystem

Represent subsystem that has synchronous reset and enable behavior

Libraries:
HDL Coder / HDL Subsystems

Description

A Synchronous Subsystem is a subsystem that uses the `Synchronous` mode of the State Control block. If an **S** symbol appears in the subsystem, then it is synchronous.

To create a Synchronous Subsystem, add the block to your Simulink model from the HDL Subsystems block library. You can also add a State Control block with **State control** set to `Synchronous` inside a subsystem.

Ports

Input

in1 — Input Signal to subsystem
scalar | vector | matrix

See Inport for information on the data types accepted by subsystem input ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

out1 — Output Signal from subsystem
scalar | vector | matrix

See Outport for information on the data types output by subsystem output ports.

For more information, see "Data Types Supported by Simulink" in the Simulink documentation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Show port labels — Display options for port labels

`FromPortIcon` (default) | `FromPortBlockName` | `SignalName` | `none`

Select how to display port labels on the Synchronous Subsystem block icon.

Do not display port labels.

FromPortIcon

If the corresponding port icon displays a signal name, display the signal name on the Subsystem block. Otherwise, display the port block name or the port number if the block name is a default name.

FromPortBlockName

Display the name of the corresponding port block on the Subsystem block.

SignalName

If the signal connected to the port is named, display the name of the signal on the Subsystem block. Otherwise, display the name of the corresponding port block.

Programmatic Use
Parameter: ShowPortLabels
Type: character vector
Value: 'FromPortIcon' | 'FromPortBlockName' | 'SignalName' | 'none'
Default: 'FromPortIcon'

Read/Write permissions — Levels of access to contents of subsystem

ReadWrite (default) | ReadOnly | NoReadOrWrite

Control user access to the contents of the subsystem.

Settings

Default: ReadWrite

ReadWrite

Enables opening and modification of subsystem contents.

ReadOnly

Enables opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

NoReadOrWrite

Disables opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

Programmatic Use
Parameter: Permissions
Type: character vector
Value: 'ReadWrite' | 'ReadOnly' | 'NoReadOrWrite'
Default: 'ReadWrite'

Name of error callback function — Name of function to be called if error occurs

'' (default) | function name

Enter name of a function to be called if an error occurs while Simulink software is executing the subsystem.

Simulink software passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink software displays a generic error message if executing the subsystem causes an error.

Programmatic Use
Parameter: ErrorFcn
Type: character vector
Value: '' | '<function name>'
Default: ''

Permit hierarchical resolution — Resolution for workspace variable names

All (default) | ExplicitOnly | None

Select whether to resolve names of workspace variables referenced by this subsystem.

For more information, see "Symbol Resolution" and "Symbol Resolution Process".

All

  Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, Simulink.Signal objects).

ExplicitOnly

  Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as "must resolve".

None

  Do not resolve workspace variable names.

Programmatic Use
Parameter: PermitHierarchicalResolution
Type: character vector
Value: 'All' | 'ExplicitOnly' | 'None'
Default: 'All'

Function packaging — Code format
Auto (default) | Inline | Nonreusable function | Reusable function

Select the code format to be generated for an atomic (nonvirtual) subsystem.

Auto

  Simulink Coder chooses the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

Inline

  Simulink Coder inlines the subsystem unconditionally.

Nonreusable function

  Simulink Coder software explicitly generates a separate function in a separate file. Subsystems with this setting generate functions that might have arguments depending on the **Function interface** parameter setting. You can name the generated function and file using parameters **Function name** and **File name (no extension)**. These functions are not reentrant.

`Reusable function`

> Simulink Coder software generates a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.
>
> This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

Programmatic Use
Parameter: RTWSystemCode
Type: character vector
Value: `'Auto'` | `'Inline'` | `'Nonreusable function'` | `'Reusable function'`
Default: `'Auto'`

Version History
Introduced in R2016a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

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`. For more details, see "ConstrainedOutputPipeline".
DistributedPipelining	Pipeline register distribution, or register retiming. The default is `inherit`. 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`. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "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".

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.

Target Specification	
AdditionalTargetInterfaces	Additional target interfaces, specified as a character vector. To save this block property on the model, in the **Set Target Interface** task of the **IP Core Generation** workflow, corresponding to the DUT ports that you want to add more interfaces, select **Add more...**. You can then add more interfaces in the Add New Target Interfaces dialog box. Specify the type of interface, number of additional interfaces, and a unique name for each additional interface. Values: `''` (default) | cell array of character vectors Example: `'{{'AXI4-Stream','InterfaceID','AXI4-Stream1'}}'`
ProcessorFPGASynchronization	Processor/FPGA synchronization mode, specified as a character vector. To save this block property on the model, specify the **Processor/FPGA Synchronization** in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `Free running` (default) | `Coprocessing - blocking` Example: `'Free running'`
TestPointMapping	To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `''` (default) | cell array of character vectors Example: `'{{'TestPoint','AXI4-Lite','x"108"'}}'`
TunableParameterMapping	To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the **Set Target Interface** task of the **IP Core Generation** workflow. Values: `''` (default) | cell array of character vectors Example: `'{{'myParam','AXI4-Lite','x"108"'}}'`
AXI4RegisterReadback	To save this block property on the model, specify whether you want to enable readback on AXI4 subordinate write registers in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'`
AXI4SlaveIDWidth	To save this block property on the model, specify the number of AXI manager interfaces that you want to connect the DUT IP core to by using the **AXI4 Slave ID Width** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface". Values: `'off'` (default) | `'on'`

Target Specification	
AXI4SlavePortToPipelineRegisterRatio	To save this block property on the model, specify the number of AXI4 subordinate ports for which you want a pipeline register to be inserted by using the **AXI4 Slave port to pipeline register ratio** setting in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. To learn more, see "Model Design for AXI4 Slave Interface Generation". Values: `'off'` (default) | `'on'``'10'``'20'``'35'``'50'`
GenerateDefaultAXI4 Slave	To save this block property on the model, specify whether you want to disable generation of default AXI4 subordinate interfaces in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow. Values: `'on'` (default) | `'off'`
IPCoreAdditionalFiles	Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;). You can set this property in the HDL Workflow Advisor, in the **Additional source files** field. Values: `''` (default) | character vector Example: `'C:\myprojfiles``\led_blinking_file1.vhd;C:\myprojfiles``\led_blinking_file2.vhd;'`
IPCoreName	IP core name, specified as a character vector. You can set this property in the HDL Workflow Advisor, in the **IP core name** field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT. Values: `''` (default) | character vector Example: `'my_model_name'`
IPCoreVersion	IP core version number, specified as a character vector. You can set this property in the HDL Workflow Advisor, in the **IP core version** field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version. Values: `''` (default) | character vector Example: `'1.3'`

Target Specification	
IPDataCaptureBuffer Size	FPGA Data Capture buffer size, specified as a character vector. Use FPGA Data Capture to observe signals in a design when running on an FPGA. The buffer size uses values that are 128*2^n, where n is an integer. By default, the buffer size is 128 (n=0). The maximum value of n is 13, which means that the maximum value for buffer size is 1048576 (=128*2^13). 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

State Control | Enabled Synchronous Subsystem | Resettable Synchronous Subsystem | 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"

Tapped Delay Enabled Synchronous

Delay scalar signal multiple sample periods and output all delayed versions when external Enable signal is true

Libraries:
HDL Coder / Discrete

Description

The Tapped Delay Enabled Synchronous block delays an input by the specified number of sample periods. The block returns output signal for each delay when the external Enable signal **E** is true.

For example, when you specify **Number of delays** as 4 and **Order output starting with** as `Oldest`, the block returns four signals, the first delayed by four sample periods, the second delayed by three, and so on. Use this block to discretize a signal in time domain or resample a signal.

Output Vector Order

The block accepts one scalar input and generates an output vector that contains data for each incremental delay. Specify the order of the delayed signals in the output vector with the **Order output vector starting with** parameter:

- `Oldest` orders the output vector starting with the oldest delay version and ending with the newest delay version.
- `Newest` orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can mitigate unwanted output behavior.

Synchronous State Control

The block uses synchronous state control semantics. When you use this block in your model and generate HDL code by using HDL Coder, your model generates cleaner HDL code and uses fewer hardware resources due to the `Synchronous` behavior of the block.

Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use `Synchronous` semantics.

Ports

Input

u — Input signal
scalar

Input signal to delay.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

E — Enable signal
`Scalar`

The block accepts the Enable signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

When the Enable signal is false, the state and output signal hold the previous value. The Enable signal is true when E input is not zero and false when E input is zero.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

y — Delayed versions of input signal
scalar | vector

All versions of the delayed input signal. Use the **Order output vector starting with** parameter to specify the order of delayed signals in the output vector.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Parameters

Initial condition — Initial output

`0.0` (default) | scalar

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Limitations

The initial condition of this block cannot be `inf` or `NaN`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar
Default: `'0.0'`

Sample time — Time between samples

-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see "Specify Sample Time".

Programmatic Use
Block Parameter: samptime
Type: character vector
Values: scalar | vector
Default: '-1'

Number of delays — Number of discrete-time operators

4 (default) | positive scalar | -1 (for inherited)

Specify the number of discrete-time operators as a positive scalar, or -1 for inherited.

A value of -1 instructs the block to inherit the number of delays by backpropagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

Programmatic Use
Block Parameter: NumDelays
Type: character vector
Values: positive scalar | -1 (inherited)
Default: '4'

Order output vector starting with — Order of output

Oldest (default) | Newest

Specify whether to output the oldest delay version first, or the newest delay version first.

Programmatic Use
Block Parameter: DelayOrder
Type: character vector
Values: 'Oldest' | 'Newest'
Default: 'Oldest'

Include current input in output vector — Include current input in output vector

off (default) | on

Select this check box to include the current input in the output vector.

Programmatic Use
Block Parameter: includeCurrent
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Version History
Introduced in R2023a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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.

See Also
Tapped Delay Resettable Synchronous | Tapped Delay Enabled Resettable Synchronous

Tapped Delay Enabled Resettable Synchronous

Delay scalar signal multiple sample periods and output all delayed versions when external Enable signal is true and external Reset signal is false

Libraries:
HDL Coder / Discrete

Description

The Tapped Delay Enabled Resettable Synchronous block delays an input by the specified number of sample periods. The block returns an output signal for each delay when external Enable signal **E** is true and external Reset signal **R** is false.

For example, when you specify **Number of delays** as 4 and **Order output starting with** as Oldest, the block returns four signals, the first delayed by four sample periods, the second delayed by three, and so on. Use this block to discretize a signal in time domain or resample a signal.

Output Vector Order

The block accepts one scalar input and generates an output vector that contains data for each incremental delay. Specify the order of the delayed signals in the output vector with the **Order output vector starting with** parameter:

- Oldest orders the output vector starting with the oldest delay version and ending with the newest delay version.
- Newest orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can mitigate unwanted output behavior.

Synchronous State Control

The block uses synchronous state control semantics. When you use this block in your model and generate HDL code by using HDL Coder, your model generates cleaner HDL code and uses fewer hardware resources due to the Synchronous behavior of the block.

Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use Classic semantics. The Subsystem must use Synchronous semantics.

Ports

Input

u — Input signal
scalar

Input signal to delay.

Data Types: half | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

E — Enable signal
Scalar

The block accepts the Enable signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

When the Enable signal is false and Reset signal is false, the state and output signal hold the previous value. The Enable signal is true when E input is not zero and false when E input is zero.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

R — Reset signal
Scalar

The block accepts the Reset signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Reset signal is true when R is not zero and false when R is zero.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

y — Delayed versions of input signal
scalar | vector

All versions of the delayed input signal. Use the **Order output vector starting with** parameter to specify the order of delayed signals in the output vector.

Data Types: half | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point

Parameters

Initial condition — Initial output

0.0 (default) | scalar

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Limitations

The initial condition of this block cannot be `inf` or `NaN`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar
Default: `'0.0'`

Sample time — Time between samples

`-1` (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. For more information, see "Specify Sample Time".

Programmatic Use
Block Parameter: `samptime`
Type: character vector
Values: scalar | vector
Default: `'-1'`

Number of delays — Number of discrete-time operators

4 (default) | positive scalar | `-1` (for inherited)

Specify the number of discrete-time operators as a positive scalar, or `-1` for inherited.

A value of `-1` instructs the block to inherit the number of delays by backpropagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

Programmatic Use
Block Parameter: `NumDelays`
Type: character vector
Values: positive scalar | `-1` (inherited)
Default: `'4'`

Order output vector starting with — Order of output

`Oldest` (default) | `Newest`

Specify whether to output the oldest delay version first, or the newest delay version first.

Programmatic Use
Block Parameter: `DelayOrder`
Type: character vector
Values: `'Oldest'` | `'Newest'`
Default: `'Oldest'`

Include current input in output vector — Include current input in output vector

`off` (default) | on

Select this check box to include the current input in the output vector.

Programmatic Use
Block Parameter: includeCurrent
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Version History

Introduced in R2023a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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.

See Also
Tapped Delay Resettable Synchronous | Tapped Delay Enabled Synchronous

Tapped Delay Resettable Synchronous

Delay scalar signal multiple sample periods and output all delayed versions when external Reset signal is false

Libraries:
HDL Coder / Discrete

Description

The Tapped Delay Enabled Synchronous block delays an input by the specified number of sample periods. The block returns an output signal for each delay when the external Reset signal **R** is false.

For example, when you specify **Number of delays** as 4 and **Order output starting with** as `Oldest`, the block returns four signals, the first delayed by four sample periods, the second delayed by three, and so on. Use this block to discretize a signal in time domain or resample a signal.

Output Vector Order

The block accepts one scalar input and generates an output vector that contains data for each incremental delay. Specify the order of the delayed signals in the output vector with the **Order output vector starting with** parameter:

- `Oldest` orders the output vector starting with the oldest delay version and ending with the newest delay version.
- `Newest` orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can mitigate unwanted output behavior.

Synchronous State Control

The block uses synchronous state control semantics. When you use this block in your model and generate HDL code by using HDL Coder, your model generates cleaner HDL code and uses fewer hardware resources due to the `Synchronous` behavior of the block.

Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use `Synchronous` semantics.

Ports

Input

u — Input signal
scalar

Input signal to delay.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

R — Reset signal
`Scalar`

The block accepts the Reset signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Reset signal is true when R is not zero and false when R is zero.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

y — Delayed versions of input signal
scalar | vector

All versions of the delayed input signal. Use the **Order output vector starting with** parameter to specify the order of delayed signals in the output vector.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`

Parameters

Initial condition — Initial output

`0.0` (default) | scalar

Specify the initial output of the simulation. The **Initial condition** parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Limitations

The initial condition of this block cannot be `inf` or `NaN`.

Programmatic Use
Block Parameter: `vinit`
Type: character vector
Values: scalar
Default: `'0.0'`

Sample time — Time between samples

-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see "Specify Sample Time".

Programmatic Use
Block Parameter: samptime
Type: character vector
Values: scalar | vector
Default: '-1'

Number of delays — Number of discrete-time operators

4 (default) | positive scalar | -1 (for inherited)

Specify the number of discrete-time operators as a positive scalar, or -1 for inherited.

A value of -1 instructs the block to inherit the number of delays by backpropagation. Each delay is equivalent to the z^{-1} discrete-time operator, which the Unit Delay block represents.

Programmatic Use
Block Parameter: NumDelays
Type: character vector
Values: positive scalar | -1 (inherited)
Default: '4'

Order output vector starting with — Order of output

Oldest (default) | Newest

Specify whether to output the oldest delay version first, or the newest delay version first.

Programmatic Use
Block Parameter: DelayOrder
Type: character vector
Values: 'Oldest' | 'Newest'
Default: 'Oldest'

Include current input in output vector — Include current input in output vector

off (default) | on

Select this check box to include the current input in the output vector.

Programmatic Use
Block Parameter: includeCurrent
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Version History
Introduced in R2023a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "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.

See Also
Tapped Delay Enabled Resettable Synchronous | Tapped Delay Enabled Synchronous

Unit Delay Enabled Synchronous

Delay input signal by one sample period when external Enable signal is true

Libraries:
HDL Coder / Discrete

Description

The Unit Delay Enabled Synchronous block delays the input signal u by one sample period when the external Enable signal is true. When the Enable signal is false, the state and output signal hold the previous value. The Enable signal is true when E is not zero and false when E is zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in `Synchronous` mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the `Synchronous` behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Enable port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use `Synchronous` semantics.

Ports

Input

u — Input signal
Scalar | Vector | Matrix | Array | Bus

The Unit Delay Enabled Synchronous block accepts the input signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Input

E — Enable signal
Scalar

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

y — Output signal
`Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

Output data type matches input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Initial condition — Initial output of simulation
0.0 (default) | `Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with `NaN` or `Inf` as the **Initial condition**.

Programmatic Use
Block parameter: `InitialCondition`
Type: character vector
Value: `'0'` | `'[n]'` | `'[m n]'`
Default: `'0'`

Sample time — Time interval between samples
-1 (default) | `Scalar` | `Vector`

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use
Block parameter: `SampleTime`
Type: character vector
Value: `'-1'` | `'[n]'` | `'[m n]'`
Default: `'-1'`

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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.

Restrictions

HDL code generation supports only boolean datatype at the Enable port.

See Also

Blocks
Unit Delay | State Control | Unit Delay Resettable Synchronous | Unit Delay Enabled Resettable Synchronous

Unit Delay Resettable Synchronous

Delay input signal by one sample period when external Reset signal is false
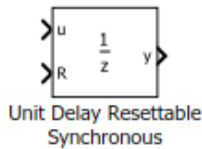
Libraries:
HDL Coder / Discrete

Description

The Unit Delay Resettable Synchronous block delays the input signal u by one sample period when the external Reset signal is false. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Reset signal is true when R is not zero and false when R is zero.

The Unit Delay Resettable Synchronous block implementation consists of a Synchronous Subsystem that contains a Resettable Delay block with a **Delay length** of one and a State Control block in `Synchronous` mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the `Synchronous` behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Reset port.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use `Synchronous` semantics.

Ports

Input

u — Input signal
Scalar | Vector | Matrix | Array | Bus

The Unit Delay Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Input

R — Reset signal
Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

y — Output signal
`Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

Output data type matches input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

Initial condition — Initial output of simulation
0.0 (default) | `Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with `NaN` or `Inf` as the **Initial condition**.

Programmatic Use
Block parameter: `InitialCondition`
Type: character vector
Value: `'0'` | `'[n]'` | `'[m n]'`
Default: `'0'`

Sample time — Time interval between samples
-1 (default) | `Scalar` | `Vector`

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use
Block parameter: `SampleTime`
Type: character vector
Value: `'-1'` | `'[n]'` | `'[m n]'`
Default: `'-1'`

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".
ResetType	Suppress reset logic generation. The default is default, which generates reset logic. See also "ResetType".

Complex Data Support

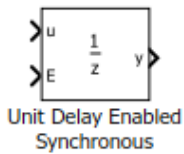This block supports code generation for complex signals.

Restrictions

HDL code generation supports only boolean datatype at the Reset port.

See Also

Blocks
Unit Delay | State Control | Unit Delay Enabled Synchronous | Unit Delay Enabled Resettable Synchronous

Unit Delay Enabled Resettable Synchronous

Delay input signal by one sample period when external Enable signal is true and external Reset signal is false

Unit Delay Enabled Resettable Synchronous
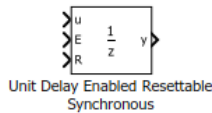
Libraries:
HDL Coder / Discrete

Description

The Unit Delay Enabled Resettable Synchronous block combines the functionality of the Unit Delay Enabled Synchronous block and the Unit Delay Resettable Synchronous block.

The Unit Delay Enabled Resettable Synchronous block delays the input signal u by one sample period when the external Enable signal is true and when the external Reset signal is false. When the Enable signal is false, the state and output signal hold the previous value. When the Reset signal is true, the state and output signal take the value of the **Initial condition** parameter. The Enable and Reset signals are true when E and R are nonzero and false when E and R equal zero.

The Unit Delay Enabled Synchronous block implementation consists of a Synchronous Subsystem that contains an Enabled Delay block with a **Delay length** of one and a State Control block in `Synchronous` mode. When you use this block in your model and have HDL Coder installed, your model generates cleaner HDL code and uses fewer hardware resources due to the `Synchronous` behavior of the State Control block.

Limitations

- The block does not support vector inputs on the Reset and Enable ports.
- You cannot use the block inside Enabled Subsystem, Triggered Subsystem, or Resettable Subsystem blocks that use `Classic` semantics. The Subsystem must use `Synchronous` semantics.

Ports

Input

u — Input signal
`Scalar` | `Vector` | `Matrix` | `Array` | `Bus`

The Unit Delay Enabled Resettable Synchronous block accepts the input signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Input

E — Enable signal
`Scalar`

The Unit Delay Enabled Synchronous block accepts the Enable signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

R — Reset signal
Scalar

The Unit Delay Resettable Synchronous block accepts the Reset signal of the data types listed below. For more information, see "Data Types Supported by Simulink".

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

y — Output signal
Scalar | Vector | Matrix | Array | Bus

Output data type matches input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Initial condition — Initial output of simulation
0.0 (default) | Scalar | Vector | Matrix | Array | Bus

The **Initial condition** can take a scalar input or use the same data type as the input signal. You cannot run the simulation with NaN or Inf as the **Initial condition**.

Programmatic Use
Block parameter: InitialCondition
Type: character vector
Value: '0' | '[n]' | '[m n]'
Default: '0'

Sample time — Time interval between samples
-1 (default) | Scalar | Vector

The **Sample time** must be a real double scalar that specifies the period or a real double vector of length two that specifies the period and offset. The period and offset must be finite and non-negative with offset less than the period.

Programmatic Use
Block parameter: SampleTime
Type: character vector
Value: '-1' | '[n]' | '[m n]'
Default: '-1'

Version History
Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "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.

Restrictions

HDL code generation supports only boolean datatype at the Reset and Enable ports.

See Also
Unit Delay | State Control | Unit Delay Resettable Synchronous | Unit Delay Enabled Synchronous

Word to Bits

Converts real numbers to vector of bits
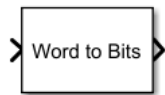
Libraries:
HDL Coder / Logic and Bit Operations

Description

The Word to Bits block converts a real number of word length *N* to a vector of bits. The output of the block is a vector whose length you specify in the **Maximum Word length** parameter.

The block treats the first element of the output vector as the least significant bit (LSB) and treats subsequent bits as the next significant bits in ascending order.

This figure shows the conversion of the unsigned integer 72 to its binary representation when you specify a maximum word length of 8. You can also convert a signed integer value to its corresponding binary vector.
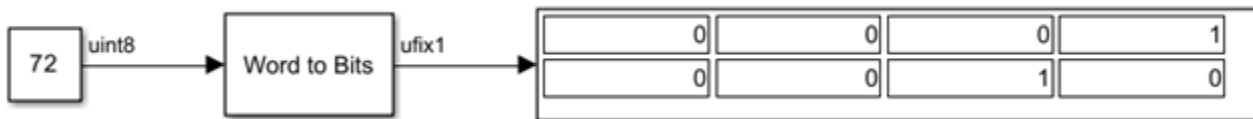
This figure shows the conversion of the non-integer value 5.5 to its binary representation. The output is a binary representation of the stored integer value of 5.5. To determine stored integer value of a fixed-point real number, use `storedInteger` function.
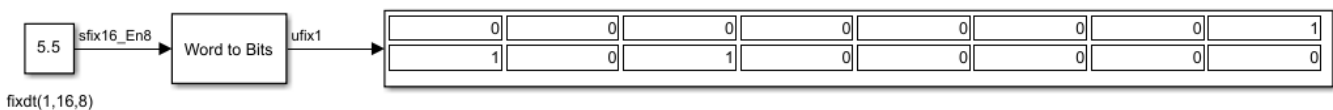
Limitations

The block has these limitations:

- Floating-point input is not supported.
- Bus, vector, and matrix inputs are not supported.
- Complex input is not supported.

Ports

Input

In — Input signal
scalar

Input signal, specified as a scalar. The word-length must be less than or equal to the **Maximum Word Length** parameter.

Example: 72

Example: 5.5

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Output

Out — Output signal
binary-valued scalar | binary-valued vector

Output signal, returned as a binary-valued scalar or vector. Set the length of this vector in the **Maximum Word length** parameter. The block treats the first element of the output vector as the LSB and treats subsequent bits as the next significant bits in ascending order.

Data Types: `ufix1`

Parameters

Maximum Word Length — Maximum output word length
64 (default) | integer in the range [1, 128]

Maximum output word length, specified as an integer in the range [1, 128]. The parameter value must be greater than or equal to the input signal word-length.

Programmatic Use
Block parameter: `maxWordLength`
Type: character vector
Value: Integer in the range [1, 128]
Default: `'8'`

Version History
Introduced in R2023a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one 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. For more details, see "ConstrainedOutputPipeline".
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline".
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline".

See Also

Bits to Word

Classes for HDL Code Generation from Simulink

hdlcoder.FloatingPointTargetConfig class

Package: hdlcoder

Floating-point target configuration for floating-point library

Description

Use objects of the `hdlcoder.FloatingPointTargetConfig` class to configure the floating-point target for a floating-point library. You can create either a native floating-point configuration object or a mixed native floating-point and vendor-specific floating point target configuration object for these vendor-specific floating-point libraries:

- Altera Megafunctions (ALTERA FP Functions)
- Altera Megafunctions (ALTFP)
- XilinxLogiCORE®

Creation

Create an `hdlcoder.FloatingPointTargetConfig` object by using the `createFloatingPointTargetConfig` method.

Properties

Vendor-Specific Floating-Point Library

VendorFloatingPointLibrary — Vendor-specific floating-point library name
`"None"` (default) | `"ALTERAFPFUNCTIONS"` | `"ALTFP"` | `"XILINXLOGICORE"`

Vendor-specific floating-point library name, specified as `"None"`, `"ALTERAFPFUNCTIONS"`, `"ALTFP"`, or `"XILINXLOGICORE"`. Specify the vendor-specific library to use in combination with the native floating-point (NFP) library.

To set properties of the vendor-specific floating-point library, use the `hdlcoder.FloatingPointTargetConfig` object.

Example: `"ALTFP"`

Data Types: `char` | `string`

Native Floating Point

HandleDenormals — Specify whether to handle denormal numbers in your design
`"Auto"` (default) | `"on"` | `"off"`

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a string array or character vector. Denormal numbers are nonzero numbers that are less than the smallest normal number.

Data Types: `char` | `string`

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 a maximum or a minimum latency setting for the floating-operators that your design uses. Specify this property as a string array or character vector.

Data Types: char | string

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. You can control the Digital Signal Processor (DSP) use on the target platform for your design. To learn more, see "Mantissa Multiplier Strategy".

Data Types: char | string

Altera FP Functions

InitializeIPPipelinesToZero — Whether to initialize pipeline registers in Altera Megafunction IP to zero
true (default) | false

Whether to initialize the pipeline registers in the Altera Megafunction IP to zero, specified as a logical. To avoid potential numeric mismatches in the HDL simulation, set InitializeIPPipelinesToZero to true. To specify this property, set VendorFloatingPointLibrary to "ALTERAFPFUNCTIONS".

Data Types: logical

ALTFP and Xilinx LogiCORE

LatencyStrategy — Whether to use maximum or minimum latency
"MIN" (default) | "MAX"

Whether to use minimum or a maximum latency when mapping your design to FPGA floating-point target libraries, specified as "MIN" or "MAX". To specify this property, set VendorFloatingPointLibrary to "ALTFP" or "XILINXLOGICORE".

Data Types: char | string

Objective — Whether to optimize design for speed or area
"SPEED" (default) | "AREA"

Whether you want to optimize the design for speed or area when mapping your design to FPGA floating-point target libraries, specified as "SPEED" or "AREA". To specify this property, set VendorFloatingPointLibrary to "ALTFP" or "XILINXLOGICORE".

Data Types: char | string

Methods

Public 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 native floating-point configuration, use the `hdlcoder.createFloatingPointTargetConfig` function.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint")


fpconfig =

  FloatingPointTargetConfig with properties:

                  Library: 'NATIVEFLOATINGPOINT'
          LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]
                 IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
            VendorLibrary: []
    VendorLibrarySettings: []
            VendorIPConfig: []
```

Load the model, `sfir_single`.

```
load_system('sfir_single');
```

Specify Custom NFP Library Settings

To customize the native 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'
    PartAddShiftMultiplierSize: '18x24'
```

Version: '3.0.0'

View Latency of Native Floating-Point Operators

The `IPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the native floating-point operators.

`fpconfig.IPConfig`

ans =

Name	DataType	MaxLatency	MinLatency	CustomLatency
{'ABS' }	{'DOUBLE' }	0	0	-1
{'ABS' }	{'SINGLE' }	0	0	-1
{'ACOS' }	{'SINGLE' }	23	17	-1
{'ACOSH' }	{'SINGLE' }	93	93	-1
{'ADDSUB' }	{'DOUBLE' }	11	6	-1
{'ADDSUB' }	{'HALF' }	8	4	-1
{'ADDSUB' }	{'SINGLE' }	11	6	-1
{'ASIN' }	{'SINGLE' }	23	17	-1
{'ASINH' }	{'SINGLE' }	94	94	-1
{'ATAN' }	{'SINGLE' }	36	36	-1
{'ATAN2' }	{'SINGLE' }	42	42	-1
{'ATANH' }	{'SINGLE' }	67	67	-1
{'CONVERT' }	{'DOUBLE_TO_NUMERICTYPE'}	6	3	-1
{'CONVERT' }	{'DOUBLE_TO_SINGLE' }	6	3	-1
{'CONVERT' }	{'HALF_TO_NUMERICTYPE' }	3	2	-1
{'CONVERT' }	{'HALF_TO_SINGLE' }	2	1	-1
{'CONVERT' }	{'NUMERICTYPE_TO_DOUBLE'}	6	3	-1
{'CONVERT' }	{'NUMERICTYPE_TO_HALF' }	4	2	-1
{'CONVERT' }	{'NUMERICTYPE_TO_SINGLE'}	6	6	-1
{'CONVERT' }	{'SINGLE_TO_DOUBLE' }	5	3	-1
{'CONVERT' }	{'SINGLE_TO_HALF' }	3	2	-1
{'CONVERT' }	{'SINGLE_TO_NUMERICTYPE'}	6	6	-1
{'COS' }	{'DOUBLE' }	48	48	-1
{'COS' }	{'HALF' }	14	9	-1
{'COS' }	{'SINGLE' }	27	27	-1
{'COSH' }	{'SINGLE' }	27	17	-1
{'DIV' }	{'DOUBLE' }	61	31	-1
{'DIV' }	{'HALF' }	19	10	-1
{'DIV' }	{'SINGLE' }	32	17	-1
{'EXP' }	{'HALF' }	16	9	-1
{'EXP' }	{'SINGLE' }	26	16	-1
{'FIX' }	{'DOUBLE' }	5	3	-1
{'FIX' }	{'SINGLE' }	5	3	-1
{'GAINPOW2' }	{'DOUBLE' }	2	1	-1
{'GAINPOW2' }	{'HALF' }	2	1	-1
{'GAINPOW2' }	{'SINGLE' }	2	1	-1
{'HDLRECIP' }	{'SINGLE' }	21	14	-1
{'HYPOT' }	{'SINGLE' }	33	17	-1
{'LOG' }	{'DOUBLE' }	44	34	-1
{'LOG' }	{'HALF' }	17	9	-1
{'LOG' }	{'SINGLE' }	27	20	-1
{'LOG10' }	{'HALF' }	18	10	-1

```
{'LOG10'   }    {'SINGLE'          }        27          17          -1
{'LOG2'    }    {'SINGLE'          }        26          16          -1
{'MINMAX'  }    {'SINGLE'          }         3           1          -1
{'MOD'     }    {'SINGLE'          }        26          16          -1
{'MUL'     }    {'DOUBLE'          }         9           6          -1
{'MUL'     }    {'HALF'            }         6           4          -1
{'MUL'     }    {'SINGLE'          }         8           6          -1
{'MULTADD' }    {'SINGLE'          }        14           8          -1
{'POW'     }    {'SINGLE'          }        54          33          -1
{'POW10'   }    {'SINGLE'          }        26          16          -1
{'POW2'    }    {'SINGLE'          }        23          14          -1
{'RECIP'   }    {'DOUBLE'          }        60          30          -1
{'RECIP'   }    {'HALF'            }        19          10          -1
{'RECIP'   }    {'SINGLE'          }        31          16          -1
{'RELOP'   }    {'DOUBLE'          }         3           1          -1
{'RELOP'   }    {'HALF'            }         2           1          -1
{'RELOP'   }    {'SINGLE'          }         3           1          -1
{'REM'     }    {'SINGLE'          }        24          15          -1
{'ROUNDING'}    {'DOUBLE'          }         5           3          -1
{'ROUNDING'}    {'SINGLE'          }         5           3          -1
{'RSQRT'   }    {'DOUBLE'          }        59          33          -1
{'RSQRT'   }    {'SINGLE'          }        30          16          -1
{'SIGNUM'  }    {'DOUBLE'          }         0           0          -1
{'SIGNUM'  }    {'SINGLE'          }         0           0          -1
{'SIN'     }    {'DOUBLE'          }        34          34          -1
{'SIN'     }    {'HALF'            }        14           8          -1
{'SIN'     }    {'SINGLE'          }        27          27          -1
{'SINCOS'  }    {'SINGLE'          }        27          27          -1
{'SINH'    }    {'SINGLE'          }        30          18          -1
{'SQRT'    }    {'DOUBLE'          }        58          36          -1
{'SQRT'    }    {'HALF'            }        12           6          -1
{'SQRT'    }    {'SINGLE'          }        28          16          -1
{'TAN'     }    {'SINGLE'          }        33          33          -1
{'TANH'    }    {'SINGLE'          }        43          25          -1
{'UMINUS'  }    {'DOUBLE'          }         0           0          -1
{'UMINUS'  }    {'HALF'            }         0           0          -1
{'UMINUS'  }    {'SINGLE'          }         0           0          -1
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by the `TargetDirectory` setting. In this example, the generated VHDL code is stored in the `C:/NativeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir', ...
                TargetDirectory='C:/NativeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
```

```
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 24 cycles.
### Output port 2: 24 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_single as C:\NativeFloatingPoint\hdlsrc\sfir_sin
### Working on sfir_single/symmetric_fir/nfp_mul_single as C:\NativeFloatingPoint\hdlsrc\sfir_sin
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_f
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/NativeFloatingPoint/hdlsrc/sfir_single/s
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Create and Customize Mixed-Mode Floating-Point Configuration and Generate Code

This example shows how to create a mixed-mode floating-point target configuration with the native floating point (NFP) library and a vendor-specific floating point library in HDL Coder™ and generate code. The vendor library in this example is the Altera® Megafunctions (ALTERAFPFUNCTIONS) library.

Create a Floating-Point Target Configuration

To create a floating-point configuration, set up the path to your synthesis tool by using the `hdlsetuptoolpath` function. For this example, use Altera Quartus II as your synthesis tool. To setup tools in your environment, run the `hdlsetuptoolpath` command with the unique synthesis tool path on your computer. For example, the function `quartuspath` returns the Altera Quartus II synthesis tool path.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', quartuspath);
```

```
Prepending following Altera Quartus II path(s) to the system path:
B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64
```

Load the model, `sfir_single`.

```
load_system('sfir_single')
```

Create a mixed-mode floating-point configuration, `fpconfig`, by using the function `hdlcoder.createFloatingPointTargetConfig`. The configuration `fpconfig` contains the NFP library and the Altera Megafunctions library configuration. Using NFP and vendor-specific IP together more efficiently uses resources on the FPGA, such as hardened DSP floating point adder or multiplier primitives, which allows you to fit a bigger design into the FPGA fabric.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",VendorFloatingPointLibr
```

```
fpconfig =

  FloatingPointTargetConfig with properties:
```

```
             Library: 'NATIVEFLOATINGPOINT'
     LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
            IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
       VendorLibrary: 'ALTERAFPFUNCTIONS'
VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
       VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom NFP Library Settings

To customize the native 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'
  PartAddShiftMultiplierSize: '18x24'
                    Version: '3.0.0'
```

Specify Custom Vendor-Specific Library Settings

To customize the vendor-specific floating-point configuration, specify custom vendor library settings.

```
fpconfig.VendorLibrarySettings.InitializeIPPipelinesToZero = true;
fpconfig.VendorLibrarySettings
```

```
ans =

  FrequencyDrivenMode with properties:

    InitializeIPPipelinesToZero: 1
```

View Latency of Floating-Point IPs

The `IPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the native floating-point operators.

```
fpconfig.IPConfig
```

```
ans =

     Name              DataType          MaxLatency    MinLatency    CustomLatency
   _____    _____    _____    _____    _____

   {'ABS'   }    {'DOUBLE'            }         0             0              -1
   {'ABS'   }    {'SINGLE'            }         0             0              -1
```

```
{'ACOS'    }    {'SINGLE'                   }    23    17    -1
{'ACOSH'   }    {'SINGLE'                   }    93    93    -1
{'ADDSUB'  }    {'DOUBLE'                   }    11     6    -1
{'ADDSUB'  }    {'HALF'                     }     8     4    -1
{'ADDSUB'  }    {'SINGLE'                   }    11     6    -1
{'ASIN'    }    {'SINGLE'                   }    23    17    -1
{'ASINH'   }    {'SINGLE'                   }    94    94    -1
{'ATAN'    }    {'SINGLE'                   }    36    36    -1
{'ATAN2'   }    {'SINGLE'                   }    42    42    -1
{'ATANH'   }    {'SINGLE'                   }    67    67    -1
{'CONVERT' }    {'DOUBLE_TO_NUMERICTYPE'}        6     3    -1
{'CONVERT' }    {'DOUBLE_TO_SINGLE'     }         6     3    -1
{'CONVERT' }    {'HALF_TO_NUMERICTYPE'  }         3     2    -1
{'CONVERT' }    {'HALF_TO_SINGLE'       }         2     1    -1
{'CONVERT' }    {'NUMERICTYPE_TO_DOUBLE'}        6     3    -1
{'CONVERT' }    {'NUMERICTYPE_TO_HALF'  }         4     2    -1
{'CONVERT' }    {'NUMERICTYPE_TO_SINGLE'}        6     6    -1
{'CONVERT' }    {'SINGLE_TO_DOUBLE'     }         5     3    -1
{'CONVERT' }    {'SINGLE_TO_HALF'       }         3     2    -1
{'CONVERT' }    {'SINGLE_TO_NUMERICTYPE'}        6     6    -1
{'COS'     }    {'DOUBLE'                   }    48    48    -1
{'COS'     }    {'HALF'                     }    14     9    -1
{'COS'     }    {'SINGLE'                   }    27    27    -1
{'COSH'    }    {'SINGLE'                   }    27    17    -1
{'DIV'     }    {'DOUBLE'                   }    61    31    -1
{'DIV'     }    {'HALF'                     }    19    10    -1
{'DIV'     }    {'SINGLE'                   }    32    17    -1
{'EXP'     }    {'HALF'                     }    16     9    -1
{'EXP'     }    {'SINGLE'                   }    26    16    -1
{'FIX'     }    {'DOUBLE'                   }     5     3    -1
{'FIX'     }    {'SINGLE'                   }     5     3    -1
{'GAINPOW2'}    {'DOUBLE'                   }     2     1    -1
{'GAINPOW2'}    {'HALF'                     }     2     1    -1
{'GAINPOW2'}    {'SINGLE'                   }     2     1    -1
{'HDLRECIP'}    {'SINGLE'                   }    21    14    -1
{'HYPOT'   }    {'SINGLE'                   }    33    17    -1
{'LOG'     }    {'DOUBLE'                   }    44    34    -1
{'LOG'     }    {'HALF'                     }    17     9    -1
{'LOG'     }    {'SINGLE'                   }    27    20    -1
{'LOG10'   }    {'HALF'                     }    18    10    -1
{'LOG10'   }    {'SINGLE'                   }    27    17    -1
{'LOG2'    }    {'SINGLE'                   }    26    16    -1
{'MINMAX'  }    {'SINGLE'                   }     3     1    -1
{'MOD'     }    {'SINGLE'                   }    26    16    -1
{'MUL'     }    {'DOUBLE'                   }     9     6    -1
{'MUL'     }    {'HALF'                     }     6     4    -1
{'MUL'     }    {'SINGLE'                   }     8     6    -1
{'MULTADD' }    {'SINGLE'                   }    14     8    -1
{'POW'     }    {'SINGLE'                   }    54    33    -1
{'POW10'   }    {'SINGLE'                   }    26    16    -1
{'POW2'    }    {'SINGLE'                   }    23    14    -1
{'RECIP'   }    {'DOUBLE'                   }    60    30    -1
{'RECIP'   }    {'HALF'                     }    19    10    -1
{'RECIP'   }    {'SINGLE'                   }    31    16    -1
{'RELOP'   }    {'DOUBLE'                   }     3     1    -1
{'RELOP'   }    {'HALF'                     }     2     1    -1
{'RELOP'   }    {'SINGLE'                   }     3     1    -1
{'REM'     }    {'SINGLE'                   }    24    15    -1
```

```
{'ROUNDING'}    {'DOUBLE'           }          5             3          -1
{'ROUNDING'}    {'SINGLE'           }          5             3          -1
{'RSQRT'   }    {'DOUBLE'           }         59            33          -1
{'RSQRT'   }    {'SINGLE'           }         30            16          -1
{'SIGNUM'  }    {'DOUBLE'           }          0             0          -1
{'SIGNUM'  }    {'SINGLE'           }          0             0          -1
{'SIN'     }    {'DOUBLE'           }         34            34          -1
{'SIN'     }    {'HALF'             }         14             8          -1
{'SIN'     }    {'SINGLE'           }         27            27          -1
{'SINCOS'  }    {'SINGLE'           }         27            27          -1
{'SINH'    }    {'SINGLE'           }         30            18          -1
{'SQRT'    }    {'DOUBLE'           }         58            36          -1
{'SQRT'    }    {'HALF'             }         12             6          -1
{'SQRT'    }    {'SINGLE'           }         28            16          -1
{'TAN'     }    {'SINGLE'           }         33            33          -1
{'TANH'    }    {'SINGLE'           }         43            25          -1
{'UMINUS'  }    {'DOUBLE'           }          0             0          -1
{'UMINUS'  }    {'HALF'             }          0             0          -1
{'UMINUS'  }    {'SINGLE'           }          0             0          -1
```

The `VendorIPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the vendor-specific floating-point operators.

```
fpconfig.VendorIPConfig
```

```
ans =

      Name                  DataType              Latency     ExtraArgs

    _____      _____    _____    _____

    {'ABS'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'ABS'    }     {'SINGLE'               }        -1       {0×0 char}
    {'ADDSUB' }     {'DOUBLE'               }        -1       {0×0 char}
    {'ADDSUB' }     {'SINGLE'               }        -1       {0×0 char}
    {'CONVERT'}     {'DOUBLE_TO_NUMERICTYPE'}        -1       {0×0 char}
    {'CONVERT'}     {'NUMERICTYPE_TO_DOUBLE'}        -1       {0×0 char}
    {'CONVERT'}     {'NUMERICTYPE_TO_SINGLE'}        -1       {0×0 char}
    {'CONVERT'}     {'SINGLE_TO_NUMERICTYPE'}        -1       {0×0 char}
    {'COS'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'COS'    }     {'SINGLE'               }        -1       {0×0 char}
    {'DIV'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'DIV'    }     {'SINGLE'               }        -1       {0×0 char}
    {'EXP'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'EXP'    }     {'SINGLE'               }        -1       {0×0 char}
    {'LOG'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'LOG'    }     {'SINGLE'               }        -1       {0×0 char}
    {'MUL'    }     {'DOUBLE'               }        -1       {0×0 char}
    {'MUL'    }     {'SINGLE'               }        -1       {0×0 char}
    {'MULTADD'}     {'SINGLE'               }        -1       {0×0 char}
    {'RECIP'  }     {'DOUBLE'               }        -1       {0×0 char}
    {'RECIP'  }     {'SINGLE'               }        -1       {0×0 char}
    {'RELOP'  }     {'DOUBLE'               }        -1       {0×0 char}
    {'RELOP'  }     {'SINGLE'               }        -1       {0×0 char}
    {'RSQRT'  }     {'DOUBLE'               }        -1       {0×0 char}
    {'RSQRT'  }     {'SINGLE'               }        -1       {0×0 char}
    {'SIN'    }     {'DOUBLE'               }        -1       {0×0 char}
```

```
{'SIN'   }    {'SINGLE'                }    -1    {0×0 char}
{'SQRT'  }    {'DOUBLE'                }    -1    {0×0 char}
{'SQRT'  }    {'SINGLE'                }    -1    {0×0 char}
```

Customize Latency of ADDSUB Vendor IP

Using the `customize` method of either of the `IPConfig` objects, you can customize the latency of the floating-point IP and specify any additional arguments. In this example, customize the latency of the vendor IP by using the `VendorIPConfig.customize` method.

```
fpconfig.VendorIPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.VendorIPConfig
```

```
ans =

        Name              DataType              Latency      ExtraArgs

    _____    _____    _____    _____

    {'ABS'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'ABS'    }    {'SINGLE'                }    -1    {0×0 char}
    {'ADDSUB' }    {'DOUBLE'                }    -1    {0×0 char}
    {'ADDSUB' }    {'SINGLE'                }     6    {0×0 char}
    {'CONVERT'}    {'DOUBLE_TO_NUMERICTYPE'}    -1    {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_DOUBLE'}    -1    {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_SINGLE'}    -1    {0×0 char}
    {'CONVERT'}    {'SINGLE_TO_NUMERICTYPE'}    -1    {0×0 char}
    {'COS'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'COS'    }    {'SINGLE'                }    -1    {0×0 char}
    {'DIV'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'DIV'    }    {'SINGLE'                }    -1    {0×0 char}
    {'EXP'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'EXP'    }    {'SINGLE'                }    -1    {0×0 char}
    {'LOG'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'LOG'    }    {'SINGLE'                }    -1    {0×0 char}
    {'MUL'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'MUL'    }    {'SINGLE'                }    -1    {0×0 char}
    {'MULTADD'}    {'SINGLE'                }    -1    {0×0 char}
    {'RECIP'  }    {'DOUBLE'                }    -1    {0×0 char}
    {'RECIP'  }    {'SINGLE'                }    -1    {0×0 char}
    {'RELOP'  }    {'DOUBLE'                }    -1    {0×0 char}
    {'RELOP'  }    {'SINGLE'                }    -1    {0×0 char}
    {'RSQRT'  }    {'DOUBLE'                }    -1    {0×0 char}
    {'RSQRT'  }    {'SINGLE'                }    -1    {0×0 char}
    {'SIN'    }    {'DOUBLE'                }    -1    {0×0 char}
    {'SIN'    }    {'SINGLE'                }    -1    {0×0 char}
    {'SQRT'   }    {'DOUBLE'                }    -1    {0×0 char}
    {'SQRT'   }    {'SINGLE'                }    -1    {0×0 char}
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Set the simulation library path in order to compile and simulate the generated code with your specified simulation tool.

```
hdlset_param('sfir_single','SimulationLibPath',alterasimulationlibpath);
```

Set the `SynthesisToolChipFamily` property to `Arria10` and generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by `TargetDirectory` property. In this example, the generated VHDL code is stored in the `C:/MixedModeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir',SynthesisToolChipFamily='Arria10',...
    TargetDirectory='C:/MixedModeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
### Using B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-ge
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 200 MHz.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sf:
### alterafpf_mul_single takes 3 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 21 cycles.
### Output port 2: 21 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generating Altera(R) megafunction: alterafpf_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sf:
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetri
### Generating package file C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/MixedModeFloatingPoint/hdlsrc/sfir_sing
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

Version History
Introduced in R2016b

R2023a: `createFloatingPointTargetConfig` no longer supports creating objects with only vendor-specific floating-point libraries
Behavior changed in R2023a

You cannot create `hdlcoder.FloatingPointTargetConfig` objects that use only vendor-specific floating-point libraries. You can create objects that use the native floating-point library or that use both the native floating-point library and a vendor-specific floating-point library.

- If you load a model created before R2023a or a command-line script that creates an vendor-specific only floating-point IP configuration object, the object is converted to use both the vendor-specific library and the native floating-point library. The native floating-point settings are the default settings.

- If you export an R2023a Simulink model that uses both native floating-point and vendor-specific floating-point libraries to an earlier version of Simulink, the object uses only the native floating-point library and the vendor-specific settings are removed, such as the `VendorLibrary`, `VendorLibrarySettings`, and `VendorIPConfig` settings.

See Also

`hdlcoder.FloatingPointTargetConfig.IPConfig` | `customize`

Topics
"FPGA Floating-Point Library IP Mapping"
"Floating Point Support: Field-Oriented Control Algorithm"
"Share Floating-Point IPs"
"Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries"
"Customize Floating-Point IP Configuration"
"Generate Target-Independent HDL Code with Native Floating-Point"

createFloatingPointTargetConfig

Class: `hdlcoder.FloatingPointTargetConfig`
Package: `hdlcoder`

Create floating-point target configuration for floating-point library that you specify

Syntax

```
fpconfig = hdlcoder.createFloatingPointTargetConfig()
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint")
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",
Name,Value)
fpconfig = hdlcoder.createFloatingPointTargetConfig(Name,Value)
```

Description

To create a floating-point target configuration object for a floating-point library, use the `hdlcoder.createFloatingPointTargetConfig` function. You can create either a native floating-point configuration object or a mixed native-floating point and vendor-specific floating-point target configuration object with these libraries as the vendor-specific floating point, vendor library options:

* Altera Megafunctions (ALTERA FP Functions)
* Altera Megafunctions (ALTFP)
* Xilinx LogiCORE

`fpconfig = hdlcoder.createFloatingPointTargetConfig()` creates an `hdlcoder.FloatingPointTargetConfig` object for a native floating-point library with default settings.

`fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint")` is an alternative way to create an `hdlcoder.FloatingPointTargetConfig` object for a native floating-point library with default settings.

`fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint", Name,Value)` creates an `hdlcoder.FloatingPointTargetConfig` object for a native floating-point library with additional options specified by one or more `Name,Value` arguments. `Name` can also be a property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`fpconfig = hdlcoder.createFloatingPointTargetConfig(Name,Value)` is an alternate way to create an `hdlcoder.FloatingPointTargetConfig` object for a native floating-point library with additional options specified by one or more `Name,Value` arguments. `Name` can also be a property name and `Value` is the corresponding value.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The name-value pair arguments that you can specify depend on the library that you select for creating the floating-point configuration.

Vendor Floating Point Library

`VendorFloatingPointLibrary` — Vendor floating point library name
`"None"` (default) | `"ALTERAFPFUNCTIONS"` | `"ALTFP"` | `"XILINXLOGICORE"`

Vendor-specific floating-point library name, specified as `"None"`, `"ALTERAFPFUNCTIONS"`, `"ALTFP"`, or `"XILINXLOGICORE"`. Specify the vendor-specific library to use in combination with a native floating-point (NFP) library. Using NFP and vendor-specific floating-point together in a mixed design more efficiently uses resources on the FPGA, which allows you to fit a bigger design into the FPGA fabric.

To set the properties of the vendor floating-point library, use the `hdlcoder.FloatingPointTargetConfig` object.

Example: `"ALTFP"`

Data Types: `char` | `string`

Native Floating Point

`HandleDenormals` — Specify whether to handle denormal numbers in your design
`"Auto"` (default) | `"on"` | `"off"`

Specify whether you want HDL Coder to handle denormal numbers in your design. Specify this property as a string array or character vector. Denormal numbers are non-zero numbers that are smaller than the smallest normal number.

Data Types: `char` | `string`

`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 string array or character vector.

Data Types: `char` | `string`

`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".

Data Types: char | string

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 native floating-point configuration, use the
`hdlcoder.createFloatingPointTargetConfig` function.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint")


fpconfig =

  FloatingPointTargetConfig with properties:

                   Library: 'NATIVEFLOATINGPOINT'
           LibrarySettings: [1x1 fpconfig.NFPLatencyDrivenMode]
                  IPConfig: [1x1 hdlcoder.FloatingPointTargetConfig.IPConfig]
             VendorLibrary: []
     VendorLibrarySettings: []
            VendorIPConfig: []
```

Load the model, `sfir_single`.

```
load_system('sfir_single');
```

Specify Custom NFP Library Settings

To customize the native 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'
    PartAddShiftMultiplierSize: '18x24'
                      Version: '3.0.0'
```

View Latency of Native Floating-Point Operators

The `IPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the native floating-point operators.

`fpconfig.IPConfig`

```
ans =
```

Name	DataType	MaxLatency	MinLatency	CustomLatency
{'ABS' }	{'DOUBLE' }	0	0	-1
{'ABS' }	{'SINGLE' }	0	0	-1
{'ACOS' }	{'SINGLE' }	23	17	-1
{'ACOSH' }	{'SINGLE' }	93	93	-1
{'ADDSUB' }	{'DOUBLE' }	11	6	-1
{'ADDSUB' }	{'HALF' }	8	4	-1
{'ADDSUB' }	{'SINGLE' }	11	6	-1
{'ASIN' }	{'SINGLE' }	23	17	-1
{'ASINH' }	{'SINGLE' }	94	94	-1
{'ATAN' }	{'SINGLE' }	36	36	-1
{'ATAN2' }	{'SINGLE' }	42	42	-1
{'ATANH' }	{'SINGLE' }	67	67	-1
{'CONVERT' }	{'DOUBLE_TO_NUMERICTYPE'}	6	3	-1
{'CONVERT' }	{'DOUBLE_TO_SINGLE' }	6	3	-1
{'CONVERT' }	{'HALF_TO_NUMERICTYPE' }	3	2	-1
{'CONVERT' }	{'HALF_TO_SINGLE' }	2	1	-1
{'CONVERT' }	{'NUMERICTYPE_TO_DOUBLE'}	6	3	-1
{'CONVERT' }	{'NUMERICTYPE_TO_HALF' }	4	2	-1
{'CONVERT' }	{'NUMERICTYPE_TO_SINGLE'}	6	6	-1
{'CONVERT' }	{'SINGLE_TO_DOUBLE' }	5	3	-1
{'CONVERT' }	{'SINGLE_TO_HALF' }	3	2	-1
{'CONVERT' }	{'SINGLE_TO_NUMERICTYPE'}	6	6	-1
{'COS' }	{'DOUBLE' }	48	48	-1
{'COS' }	{'HALF' }	14	9	-1
{'COS' }	{'SINGLE' }	27	27	-1
{'COSH' }	{'SINGLE' }	27	17	-1
{'DIV' }	{'DOUBLE' }	61	31	-1
{'DIV' }	{'HALF' }	19	10	-1
{'DIV' }	{'SINGLE' }	32	17	-1
{'EXP' }	{'HALF' }	16	9	-1
{'EXP' }	{'SINGLE' }	26	16	-1
{'FIX' }	{'DOUBLE' }	5	3	-1
{'FIX' }	{'SINGLE' }	5	3	-1
{'GAINPOW2'}	{'DOUBLE' }	2	1	-1
{'GAINPOW2'}	{'HALF' }	2	1	-1
{'GAINPOW2'}	{'SINGLE' }	2	1	-1
{'HDLRECIP'}	{'SINGLE' }	21	14	-1
{'HYPOT' }	{'SINGLE' }	33	17	-1
{'LOG' }	{'DOUBLE' }	44	34	-1
{'LOG' }	{'HALF' }	17	9	-1
{'LOG' }	{'SINGLE' }	27	20	-1
{'LOG10' }	{'HALF' }	18	10	-1
{'LOG10' }	{'SINGLE' }	27	17	-1
{'LOG2' }	{'SINGLE' }	26	16	-1
{'MINMAX' }	{'SINGLE' }	3	1	-1

```
{'MOD'      }   {'SINGLE'         }          26          16          -1
{'MUL'      }   {'DOUBLE'         }           9           6          -1
{'MUL'      }   {'HALF'           }           6           4          -1
{'MUL'      }   {'SINGLE'         }           8           6          -1
{'MULTADD'  }   {'SINGLE'         }          14           8          -1
{'POW'      }   {'SINGLE'         }          54          33          -1
{'POW10'    }   {'SINGLE'         }          26          16          -1
{'POW2'     }   {'SINGLE'         }          23          14          -1
{'RECIP'    }   {'DOUBLE'         }          60          30          -1
{'RECIP'    }   {'HALF'           }          19          10          -1
{'RECIP'    }   {'SINGLE'         }          31          16          -1
{'RELOP'    }   {'DOUBLE'         }           3           1          -1
{'RELOP'    }   {'HALF'           }           2           1          -1
{'RELOP'    }   {'SINGLE'         }           3           1          -1
{'REM'      }   {'SINGLE'         }          24          15          -1
{'ROUNDING'}   {'DOUBLE'         }           5           3          -1
{'ROUNDING'}   {'SINGLE'         }           5           3          -1
{'RSQRT'    }   {'DOUBLE'         }          59          33          -1
{'RSQRT'    }   {'SINGLE'         }          30          16          -1
{'SIGNUM'   }   {'DOUBLE'         }           0           0          -1
{'SIGNUM'   }   {'SINGLE'         }           0           0          -1
{'SIN'      }   {'DOUBLE'         }          34          34          -1
{'SIN'      }   {'HALF'           }          14           8          -1
{'SIN'      }   {'SINGLE'         }          27          27          -1
{'SINCOS'   }   {'SINGLE'         }          27          27          -1
{'SINH'     }   {'SINGLE'         }          30          18          -1
{'SQRT'     }   {'DOUBLE'         }          58          36          -1
{'SQRT'     }   {'HALF'           }          12           6          -1
{'SQRT'     }   {'SINGLE'         }          28          16          -1
{'TAN'      }   {'SINGLE'         }          33          33          -1
{'TANH'     }   {'SINGLE'         }          43          25          -1
{'UMINUS'   }   {'DOUBLE'         }           0           0          -1
{'UMINUS'   }   {'HALF'           }           0           0          -1
{'UMINUS'   }   {'SINGLE'         }           0           0          -1
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by the `TargetDirectory` setting. In this example, the generated VHDL code is stored in the `C:/NativeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir', ...
                TargetDirectory='C:/NativeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
```

```
### Output port 1: 24 cycles.
### Output port 2: 24 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir/nfp_add_single as C:\NativeFloatingPoint\hdlsrc\sfir_sin
### Working on sfir_single/symmetric_fir/nfp_mul_single as C:\NativeFloatingPoint\hdlsrc\sfir_sin
### Working on sfir_single/symmetric_fir as C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_f
### Generating package file C:\NativeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/NativeFloatingPoint/hdlsrc/sfir_single/s
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Create and Customize Mixed-Mode Floating-Point Configuration and Generate Code

This example shows how to create a mixed-mode floating-point target configuration with the native floating point (NFP) library and a vendor-specific floating point library in HDL Coder™ and generate code. The vendor library in this example is the Altera® Megafunctions (ALTERAFPFUNCTIONS) library.

Create a Floating-Point Target Configuration

To create a floating-point configuration, set up the path to your synthesis tool by using the hdlsetuptoolpath function. For this example, use Altera Quartus II as your synthesis tool. To setup tools in your environment, run the hdlsetuptoolpath command with the unique synthesis tool path on your computer. For example, the function quartuspath returns the Altera Quartus II synthesis tool path.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', quartuspath);
```

```
Prepending following Altera Quartus II path(s) to the system path:
B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64
```

Load the model, sfir_single.

```
load_system('sfir_single')
```

Create a mixed-mode floating-point configuration, fpconfig, by using the function hdlcoder.createFloatingPointTargetConfig. The configuration fpconfig contains the NFP library and the Altera Megafunctions library configuration. Using NFP and vendor-specific IP together more efficiently uses resources on the FPGA, such as hardened DSP floating point adder or multiplier primitives, which allows you to fit a bigger design into the FPGA fabric.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",VendorFloatingPointLibr
```

```
fpconfig =

  FloatingPointTargetConfig with properties:

               Library: 'NATIVEFLOATINGPOINT'
       LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
              IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

```
             VendorLibrary: 'ALTERAFPFUNCTIONS'
     VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
             VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom NFP Library Settings

To customize the native 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'
   PartAddShiftMultiplierSize: '18x24'
                     Version: '3.0.0'
```

Specify Custom Vendor-Specific Library Settings

To customize the vendor-specific floating-point configuration, specify custom vendor library settings.

```
fpconfig.VendorLibrarySettings.InitializeIPPipelinesToZero = true;
fpconfig.VendorLibrarySettings
```

```
ans =

  FrequencyDrivenMode with properties:

    InitializeIPPipelinesToZero: 1
```

View Latency of Floating-Point IPs

The `IPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the native floating-point operators.

```
fpconfig.IPConfig
```

```
ans =

      Name              DataType          MaxLatency    MinLatency    CustomLatency
   _____    _____    _____    _____    _____

   {'ABS'    }     {'DOUBLE'         }          0             0              -1
   {'ABS'    }     {'SINGLE'         }          0             0              -1
   {'ACOS'   }     {'SINGLE'         }         23            17              -1
   {'ACOSH'  }     {'SINGLE'         }         93            93              -1
   {'ADDSUB' }     {'DOUBLE'         }         11             6              -1
```

```
{'ADDSUB'  }    {'HALF'                    }       8         4        -1
{'ADDSUB'  }    {'SINGLE'                  }      11         6        -1
{'ASIN'    }    {'SINGLE'                  }      23        17        -1
{'ASINH'   }    {'SINGLE'                  }      94        94        -1
{'ATAN'    }    {'SINGLE'                  }      36        36        -1
{'ATAN2'   }    {'SINGLE'                  }      42        42        -1
{'ATANH'   }    {'SINGLE'                  }      67        67        -1
{'CONVERT' }    {'DOUBLE_TO_NUMERICTYPE'}         6         3        -1
{'CONVERT' }    {'DOUBLE_TO_SINGLE'        }       6         3        -1
{'CONVERT' }    {'HALF_TO_NUMERICTYPE'  }          3         2        -1
{'CONVERT' }    {'HALF_TO_SINGLE'          }       2         1        -1
{'CONVERT' }    {'NUMERICTYPE_TO_DOUBLE'}         6         3        -1
{'CONVERT' }    {'NUMERICTYPE_TO_HALF'  }          4         2        -1
{'CONVERT' }    {'NUMERICTYPE_TO_SINGLE'}         6         6        -1
{'CONVERT' }    {'SINGLE_TO_DOUBLE'        }       5         3        -1
{'CONVERT' }    {'SINGLE_TO_HALF'          }       3         2        -1
{'CONVERT' }    {'SINGLE_TO_NUMERICTYPE'}         6         6        -1
{'COS'     }    {'DOUBLE'                  }      48        48        -1
{'COS'     }    {'HALF'                    }      14         9        -1
{'COS'     }    {'SINGLE'                  }      27        27        -1
{'COSH'    }    {'SINGLE'                  }      27        17        -1
{'DIV'     }    {'DOUBLE'                  }      61        31        -1
{'DIV'     }    {'HALF'                    }      19        10        -1
{'DIV'     }    {'SINGLE'                  }      32        17        -1
{'EXP'     }    {'HALF'                    }      16         9        -1
{'EXP'     }    {'SINGLE'                  }      26        16        -1
{'FIX'     }    {'DOUBLE'                  }       5         3        -1
{'FIX'     }    {'SINGLE'                  }       5         3        -1
{'GAINPOW2'}    {'DOUBLE'                  }       2         1        -1
{'GAINPOW2'}    {'HALF'                    }       2         1        -1
{'GAINPOW2'}    {'SINGLE'                  }       2         1        -1
{'HDLRECIP'}    {'SINGLE'                  }      21        14        -1
{'HYPOT'   }    {'SINGLE'                  }      33        17        -1
{'LOG'     }    {'DOUBLE'                  }      44        34        -1
{'LOG'     }    {'HALF'                    }      17         9        -1
{'LOG'     }    {'SINGLE'                  }      27        20        -1
{'LOG10'   }    {'HALF'                    }      18        10        -1
{'LOG10'   }    {'SINGLE'                  }      27        17        -1
{'LOG2'    }    {'SINGLE'                  }      26        16        -1
{'MINMAX'  }    {'SINGLE'                  }       3         1        -1
{'MOD'     }    {'SINGLE'                  }      26        16        -1
{'MUL'     }    {'DOUBLE'                  }       9         6        -1
{'MUL'     }    {'HALF'                    }       6         4        -1
{'MUL'     }    {'SINGLE'                  }       8         6        -1
{'MULTADD' }    {'SINGLE'                  }      14         8        -1
{'POW'     }    {'SINGLE'                  }      54        33        -1
{'POW10'   }    {'SINGLE'                  }      26        16        -1
{'POW2'    }    {'SINGLE'                  }      23        14        -1
{'RECIP'   }    {'DOUBLE'                  }      60        30        -1
{'RECIP'   }    {'HALF'                    }      19        10        -1
{'RECIP'   }    {'SINGLE'                  }      31        16        -1
{'RELOP'   }    {'DOUBLE'                  }       3         1        -1
{'RELOP'   }    {'HALF'                    }       2         1        -1
{'RELOP'   }    {'SINGLE'                  }       3         1        -1
{'REM'     }    {'SINGLE'                  }      24        15        -1
{'ROUNDING'}    {'DOUBLE'                  }       5         3        -1
{'ROUNDING'}    {'SINGLE'                  }       5         3        -1
{'RSQRT'   }    {'DOUBLE'                  }      59        33        -1
```

```
{'RSQRT'  }    {'SINGLE'       }        30          16           -1
{'SIGNUM' }    {'DOUBLE'       }         0           0           -1
{'SIGNUM' }    {'SINGLE'       }         0           0           -1
{'SIN'    }    {'DOUBLE'       }        34          34           -1
{'SIN'    }    {'HALF'         }        14           8           -1
{'SIN'    }    {'SINGLE'       }        27          27           -1
{'SINCOS' }    {'SINGLE'       }        27          27           -1
{'SINH'   }    {'SINGLE'       }        30          18           -1
{'SQRT'   }    {'DOUBLE'       }        58          36           -1
{'SQRT'   }    {'HALF'         }        12           6           -1
{'SQRT'   }    {'SINGLE'       }        28          16           -1
{'TAN'    }    {'SINGLE'       }        33          33           -1
{'TANH'   }    {'SINGLE'       }        43          25           -1
{'UMINUS' }    {'DOUBLE'       }         0           0           -1
{'UMINUS' }    {'HALF'         }         0           0           -1
{'UMINUS' }    {'SINGLE'       }         0           0           -1
```

The `VendorIPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the vendor-specific floating-point operators.

`fpconfig.VendorIPConfig`

```
ans =

     Name              DataType              Latency      ExtraArgs
  _____    _____    _____    _____

  {'ABS'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'ABS'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'ADDSUB'  }    {'DOUBLE'                 }      -1      {0×0 char}
  {'ADDSUB'  }    {'SINGLE'                 }      -1      {0×0 char}
  {'CONVERT' }    {'DOUBLE_TO_NUMERICTYPE'  }      -1      {0×0 char}
  {'CONVERT' }    {'NUMERICTYPE_TO_DOUBLE'  }      -1      {0×0 char}
  {'CONVERT' }    {'NUMERICTYPE_TO_SINGLE'  }      -1      {0×0 char}
  {'CONVERT' }    {'SINGLE_TO_NUMERICTYPE'  }      -1      {0×0 char}
  {'COS'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'COS'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'DIV'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'DIV'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'EXP'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'EXP'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'LOG'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'LOG'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'MUL'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'MUL'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'MULTADD' }    {'SINGLE'                 }      -1      {0×0 char}
  {'RECIP'   }    {'DOUBLE'                 }      -1      {0×0 char}
  {'RECIP'   }    {'SINGLE'                 }      -1      {0×0 char}
  {'RELOP'   }    {'DOUBLE'                 }      -1      {0×0 char}
  {'RELOP'   }    {'SINGLE'                 }      -1      {0×0 char}
  {'RSQRT'   }    {'DOUBLE'                 }      -1      {0×0 char}
  {'RSQRT'   }    {'SINGLE'                 }      -1      {0×0 char}
  {'SIN'     }    {'DOUBLE'                 }      -1      {0×0 char}
  {'SIN'     }    {'SINGLE'                 }      -1      {0×0 char}
  {'SQRT'    }    {'DOUBLE'                 }      -1      {0×0 char}
```

```
{'SQRT'   }    {'SINGLE'                  }      -1      {0×0 char}
```

Customize Latency of ADDSUB Vendor IP

Using the `customize` method of either of the `IPConfig` objects, you can customize the latency of the floating-point IP and specify any additional arguments. In this example, customize the latency of the vendor IP by using the `VendorIPConfig.customize` method.

```
fpconfig.VendorIPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.VendorIPConfig
```

```
ans =

    Name                DataType              Latency     ExtraArgs
 _____    _____    _____    _____

 {'ABS'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'ABS'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'ADDSUB' }    {'DOUBLE'                 }      -1      {0×0 char}
 {'ADDSUB' }    {'SINGLE'                 }       6      {0×0 char}
 {'CONVERT'}    {'DOUBLE_TO_NUMERICTYPE'}      -1      {0×0 char}
 {'CONVERT'}    {'NUMERICTYPE_TO_DOUBLE'}      -1      {0×0 char}
 {'CONVERT'}    {'NUMERICTYPE_TO_SINGLE'}      -1      {0×0 char}
 {'CONVERT'}    {'SINGLE_TO_NUMERICTYPE'}      -1      {0×0 char}
 {'COS'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'COS'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'DIV'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'DIV'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'EXP'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'EXP'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'LOG'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'LOG'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'MUL'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'MUL'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'MULTADD'}    {'SINGLE'                 }      -1      {0×0 char}
 {'RECIP'  }    {'DOUBLE'                 }      -1      {0×0 char}
 {'RECIP'  }    {'SINGLE'                 }      -1      {0×0 char}
 {'RELOP'  }    {'DOUBLE'                 }      -1      {0×0 char}
 {'RELOP'  }    {'SINGLE'                 }      -1      {0×0 char}
 {'RSQRT'  }    {'DOUBLE'                 }      -1      {0×0 char}
 {'RSQRT'  }    {'SINGLE'                 }      -1      {0×0 char}
 {'SIN'    }    {'DOUBLE'                 }      -1      {0×0 char}
 {'SIN'    }    {'SINGLE'                 }      -1      {0×0 char}
 {'SQRT'   }    {'DOUBLE'                 }      -1      {0×0 char}
 {'SQRT'   }    {'SINGLE'                 }      -1      {0×0 char}
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Set the simulation library path in order to compile and simulate the generated code with your specified simulation tool.

```
hdlset_param('sfir_single','SimulationLibPath',alterasimulationlibpath);
```

Set the `SynthesisToolChipFamily` property to `Arria10` and generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by `TargetDirectory` property. In this example, the generated VHDL code is stored in the `C:/MixedModeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir',SynthesisToolChipFamily='Arria10',...
    TargetDirectory='C:/MixedModeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
### Using B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-ge
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 200 MHz.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sfi
### alterafpf_mul_single takes 3 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipel
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 21 cycles.
### Output port 2: 21 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generating Altera(R) megafunction: alterafpf_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sfi
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetri
### Generating package file C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/MixedModeFloatingPoint/hdlsrc/sfir_singl
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

Version History

R2023a: `createFloatingPointTargetConfig` no longer supports creating objects with only vendor-specific floating-point libraries
Behavior changed in R2023a

You cannot create `hdlcoder.FloatingPointTargetConfig` objects that use only vendor-specific floating-point libraries. You can create objects that use the native floating-point library or that use both the native floating-point library and a vendor-specific floating-point library.

- If you load a model created before R2023a or a command-line script that creates an vendor-specific only floating-point IP configuration object, the object is converted to use both the vendor-specific library and the native floating-point library. The native floating-point settings are the default settings.

- If you export an R2023a Simulink model that uses both native floating-point and vendor-specific floating-point libraries to an earlier version of Simulink, the object uses only the native floating-point library and the vendor-specific settings are removed, such as the `VendorLibrary`, `VendorLibrarySettings`, and `VendorIPConfig` settings.

See Also

`hdlcoder.FloatingPointTargetConfig` |
`hdlcoder.FloatingPointTargetConfig.IPConfig` | `customize`

Topics
"FPGA Floating-Point Library IP Mapping"
"Floating Point Support: Field-Oriented Control Algorithm"
"Share Floating-Point IPs"
"Generate HDL Code for Vendor-Specific 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

IP settings for selected floating-point configuration

Description

Use objects of the `hdlcoder.FloatingPointTargetConfig.IPConfig` class to list the 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. To see the native floating-point and vendor-specific floating point library IP settings:

1 Create a mixed-mode floating-point target configuration object and specify your desired vendor-specific floating-point library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",...
    VendorFloatingPointLibrary="ALTFP");
```

2 To see the native floating-point library IP settings, use the `IPConfig` object.

```
fpconfig.IPConfig
```

3 To see the vendor-specific floating-point library IP settings, use the `VendorIPConfig` object.

```
fpconfig.VendorIPConfig
```

To customize the IP configurations, use the `customize` method of either of the floating-point IP configuration objects.

Creation

Create objects by using the `createFloatingPointTargetConfig` method.

`fpconfig.IPConfig` shows the IP settings for the `fpconfig` mixed-mode floating-point target configuration that you create for the native floating-point library.

`fpconfig.VendorIPConfig` shows the IP settings for the `fpconfig` mixed-mode floating-point target configuration that you create for the vendor-specific floating-point library.

Methods

Public Methods

customize   Customize IP configuration for floating-point library

Examples

Create and Customize Mixed-Mode Floating-Point Configuration and Generate Code

This example shows how to create a mixed-mode floating-point target configuration with the native floating point (NFP) library and a vendor-specific floating point library in HDL Coder™ and generate

code. The vendor library in this example is the Altera® Megafunctions (ALTERAFPFUNCTIONS) library.

Create a Floating-Point Target Configuration

To create a floating-point configuration, set up the path to your synthesis tool by using the `hdlsetuptoolpath` function. For this example, use Altera Quartus II as your synthesis tool. To setup tools in your environment, run the `hdlsetuptoolpath` command with the unique synthesis tool path on your computer. For example, the function `quartuspath` returns the Altera Quartus II synthesis tool path.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', quartuspath);
```

```
Prepending following Altera Quartus II path(s) to the system path:
B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64
```

Load the model, `sfir_single`.

```
load_system('sfir_single')
```

Create a mixed-mode floating-point configuration, `fpconfig`, by using the function `hdlcoder.createFloatingPointTargetConfig`. The configuration `fpconfig` contains the NFP library and the Altera Megafunctions library configuration. Using NFP and vendor-specific IP together more efficiently uses resources on the FPGA, such as hardened DSP floating point adder or multiplier primitives, which allows you to fit a bigger design into the FPGA fabric.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",VendorFloatingPointLib
```

```
fpconfig =

  FloatingPointTargetConfig with properties:

                  Library: 'NATIVEFLOATINGPOINT'
          LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
                 IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
            VendorLibrary: 'ALTERAFPFUNCTIONS'
    VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
           VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom NFP Library Settings

To customize the native 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'
```

4-27

```
              PartAddShiftMultiplierSize: '18x24'
                                 Version: '3.0.0'
```

Specify Custom Vendor-Specific Library Settings

To customize the vendor-specific floating-point configuration, specify custom vendor library settings.

```
fpconfig.VendorLibrarySettings.InitializeIPPipelinesToZero = true;
fpconfig.VendorLibrarySettings
```

```
ans =

  FrequencyDrivenMode with properties:

    InitializeIPPipelinesToZero: 1
```

View Latency of Floating-Point IPs

The IPConfig property stores an IPConfig object that displays the maximum and minimum latency values of the native floating-point operators.

```
fpconfig.IPConfig
```

```
ans =

        Name                     DataType              MaxLatency    MinLatency    CustomLatency
    _____    _____    _____    _____    _____

    {'ABS'      }    {'DOUBLE'                    }        0             0              -1
    {'ABS'      }    {'SINGLE'                    }        0             0              -1
    {'ACOS'     }    {'SINGLE'                    }       23            17              -1
    {'ACOSH'    }    {'SINGLE'                    }       93            93              -1
    {'ADDSUB'   }    {'DOUBLE'                    }       11             6              -1
    {'ADDSUB'   }    {'HALF'                      }        8             4              -1
    {'ADDSUB'   }    {'SINGLE'                    }       11             6              -1
    {'ASIN'     }    {'SINGLE'                    }       23            17              -1
    {'ASINH'    }    {'SINGLE'                    }       94            94              -1
    {'ATAN'     }    {'SINGLE'                    }       36            36              -1
    {'ATAN2'    }    {'SINGLE'                    }       42            42              -1
    {'ATANH'    }    {'SINGLE'                    }       67            67              -1
    {'CONVERT'  }    {'DOUBLE_TO_NUMERICTYPE'     }        6             3              -1
    {'CONVERT'  }    {'DOUBLE_TO_SINGLE'          }        6             3              -1
    {'CONVERT'  }    {'HALF_TO_NUMERICTYPE'       }        3             2              -1
    {'CONVERT'  }    {'HALF_TO_SINGLE'            }        2             1              -1
    {'CONVERT'  }    {'NUMERICTYPE_TO_DOUBLE'     }        6             3              -1
    {'CONVERT'  }    {'NUMERICTYPE_TO_HALF'       }        4             2              -1
    {'CONVERT'  }    {'NUMERICTYPE_TO_SINGLE'     }        6             6              -1
    {'CONVERT'  }    {'SINGLE_TO_DOUBLE'          }        5             3              -1
    {'CONVERT'  }    {'SINGLE_TO_HALF'            }        3             2              -1
    {'CONVERT'  }    {'SINGLE_TO_NUMERICTYPE'     }        6             6              -1
    {'COS'      }    {'DOUBLE'                    }       48            48              -1
    {'COS'      }    {'HALF'                      }       14             9              -1
    {'COS'      }    {'SINGLE'                    }       27            27              -1
    {'COSH'     }    {'SINGLE'                    }       27            17              -1
    {'DIV'      }    {'DOUBLE'                    }       61            31              -1
```

```
{'DIV'     }    {'HALF'      }              19          10          -1
{'DIV'     }    {'SINGLE'    }              32          17          -1
{'EXP'     }    {'HALF'      }              16           9          -1
{'EXP'     }    {'SINGLE'    }              26          16          -1
{'FIX'     }    {'DOUBLE'    }               5           3          -1
{'FIX'     }    {'SINGLE'    }               5           3          -1
{'GAINPOW2'}    {'DOUBLE'    }               2           1          -1
{'GAINPOW2'}    {'HALF'      }               2           1          -1
{'GAINPOW2'}    {'SINGLE'    }               2           1          -1
{'HDLRECIP'}    {'SINGLE'    }              21          14          -1
{'HYPOT'   }    {'SINGLE'    }              33          17          -1
{'LOG'     }    {'DOUBLE'    }              44          34          -1
{'LOG'     }    {'HALF'      }              17           9          -1
{'LOG'     }    {'SINGLE'    }              27          20          -1
{'LOG10'   }    {'HALF'      }              18          10          -1
{'LOG10'   }    {'SINGLE'    }              27          17          -1
{'LOG2'    }    {'SINGLE'    }              26          16          -1
{'MINMAX'  }    {'SINGLE'    }               3           1          -1
{'MOD'     }    {'SINGLE'    }              26          16          -1
{'MUL'     }    {'DOUBLE'    }               9           6          -1
{'MUL'     }    {'HALF'      }               6           4          -1
{'MUL'     }    {'SINGLE'    }               8           6          -1
{'MULTADD' }    {'SINGLE'    }              14           8          -1
{'POW'     }    {'SINGLE'    }              54          33          -1
{'POW10'   }    {'SINGLE'    }              26          16          -1
{'POW2'    }    {'SINGLE'    }              23          14          -1
{'RECIP'   }    {'DOUBLE'    }              60          30          -1
{'RECIP'   }    {'HALF'      }              19          10          -1
{'RECIP'   }    {'SINGLE'    }              31          16          -1
{'RELOP'   }    {'DOUBLE'    }               3           1          -1
{'RELOP'   }    {'HALF'      }               2           1          -1
{'RELOP'   }    {'SINGLE'    }               3           1          -1
{'REM'     }    {'SINGLE'    }              24          15          -1
{'ROUNDING'}    {'DOUBLE'    }               5           3          -1
{'ROUNDING'}    {'SINGLE'    }               5           3          -1
{'RSQRT'   }    {'DOUBLE'    }              59          33          -1
{'RSQRT'   }    {'SINGLE'    }              30          16          -1
{'SIGNUM'  }    {'DOUBLE'    }               0           0          -1
{'SIGNUM'  }    {'SINGLE'    }               0           0          -1
{'SIN'     }    {'DOUBLE'    }              34          34          -1
{'SIN'     }    {'HALF'      }              14           8          -1
{'SIN'     }    {'SINGLE'    }              27          27          -1
{'SINCOS'  }    {'SINGLE'    }              27          27          -1
{'SINH'    }    {'SINGLE'    }              30          18          -1
{'SQRT'    }    {'DOUBLE'    }              58          36          -1
{'SQRT'    }    {'HALF'      }              12           6          -1
{'SQRT'    }    {'SINGLE'    }              28          16          -1
{'TAN'     }    {'SINGLE'    }              33          33          -1
{'TANH'    }    {'SINGLE'    }              43          25          -1
{'UMINUS'  }    {'DOUBLE'    }               0           0          -1
{'UMINUS'  }    {'HALF'      }               0           0          -1
{'UMINUS'  }    {'SINGLE'    }               0           0          -1
```

The `VendorIPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the vendor-specific floating-point operators.

`fpconfig.VendorIPConfig`

```
ans =

     Name                    DataType                Latency      ExtraArgs
  _____    _____    _____    _____

  {'ABS'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'ABS'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'ADDSUB' }      {'DOUBLE'                  }        -1       {0×0 char}
  {'ADDSUB' }      {'SINGLE'                  }        -1       {0×0 char}
  {'CONVERT'}      {'DOUBLE_TO_NUMERICTYPE'}          -1       {0×0 char}
  {'CONVERT'}      {'NUMERICTYPE_TO_DOUBLE'}          -1       {0×0 char}
  {'CONVERT'}      {'NUMERICTYPE_TO_SINGLE'}          -1       {0×0 char}
  {'CONVERT'}      {'SINGLE_TO_NUMERICTYPE'}          -1       {0×0 char}
  {'COS'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'COS'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'DIV'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'DIV'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'EXP'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'EXP'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'LOG'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'LOG'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'MUL'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'MUL'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'MULTADD'}      {'SINGLE'                  }        -1       {0×0 char}
  {'RECIP'  }      {'DOUBLE'                  }        -1       {0×0 char}
  {'RECIP'  }      {'SINGLE'                  }        -1       {0×0 char}
  {'RELOP'  }      {'DOUBLE'                  }        -1       {0×0 char}
  {'RELOP'  }      {'SINGLE'                  }        -1       {0×0 char}
  {'RSQRT'  }      {'DOUBLE'                  }        -1       {0×0 char}
  {'RSQRT'  }      {'SINGLE'                  }        -1       {0×0 char}
  {'SIN'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'SIN'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'SQRT'   }      {'DOUBLE'                  }        -1       {0×0 char}
  {'SQRT'   }      {'SINGLE'                  }        -1       {0×0 char}
```

Customize Latency of ADDSUB Vendor IP

Using the `customize` method of either of the `IPConfig` objects, you can customize the latency of the floating-point IP and specify any additional arguments. In this example, customize the latency of the vendor IP by using the `VendorIPConfig.customize` method.

```
fpconfig.VendorIPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.VendorIPConfig
```

```
ans =

     Name                    DataType                Latency      ExtraArgs
  _____    _____    _____    _____

  {'ABS'    }      {'DOUBLE'                  }        -1       {0×0 char}
  {'ABS'    }      {'SINGLE'                  }        -1       {0×0 char}
  {'ADDSUB' }      {'DOUBLE'                  }        -1       {0×0 char}
  {'ADDSUB' }      {'SINGLE'                  }         6       {0×0 char}
  {'CONVERT'}      {'DOUBLE_TO_NUMERICTYPE'}          -1       {0×0 char}
  {'CONVERT'}      {'NUMERICTYPE_TO_DOUBLE'}          -1       {0×0 char}
```

```
{'CONVERT'}    {'NUMERICTYPE_TO_SINGLE'}       -1      {0×0 char}
{'CONVERT'}    {'SINGLE_TO_NUMERICTYPE'}       -1      {0×0 char}
{'COS'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'COS'    }    {'SINGLE'                 }      -1      {0×0 char}
{'DIV'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'DIV'    }    {'SINGLE'                 }      -1      {0×0 char}
{'EXP'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'EXP'    }    {'SINGLE'                 }      -1      {0×0 char}
{'LOG'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'LOG'    }    {'SINGLE'                 }      -1      {0×0 char}
{'MUL'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'MUL'    }    {'SINGLE'                 }      -1      {0×0 char}
{'MULTADD'}    {'SINGLE'                 }      -1      {0×0 char}
{'RECIP'  }    {'DOUBLE'                 }      -1      {0×0 char}
{'RECIP'  }    {'SINGLE'                 }      -1      {0×0 char}
{'RELOP'  }    {'DOUBLE'                 }      -1      {0×0 char}
{'RELOP'  }    {'SINGLE'                 }      -1      {0×0 char}
{'RSQRT'  }    {'DOUBLE'                 }      -1      {0×0 char}
{'RSQRT'  }    {'SINGLE'                 }      -1      {0×0 char}
{'SIN'    }    {'DOUBLE'                 }      -1      {0×0 char}
{'SIN'    }    {'SINGLE'                 }      -1      {0×0 char}
{'SQRT'   }    {'DOUBLE'                 }      -1      {0×0 char}
{'SQRT'   }    {'SINGLE'                 }      -1      {0×0 char}
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Set the simulation library path in order to compile and simulate the generated code with your specified simulation tool.

```
hdlset_param('sfir_single','SimulationLibPath',alterasimulationlibpath);
```

Set the `SynthesisToolChipFamily` property to `Arria10` and generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by `TargetDirectory` property. In this example, the generated VHDL code is stored in the `C:/MixedModeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir',SynthesisToolChipFamily='Arria10',...
    TargetDirectory='C:/MixedModeFloatingPoint/hdlsrc')

### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
### Using B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-ge
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 200 MHz.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sf:
### alterafpf_mul_single takes 3 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipe:
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
```

```
### Output port 1: 21 cycles.
### Output port 2: 21 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generating Altera(R) megafunction: alterafpf_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sfi
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetri
### Generating package file C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/MixedModeFloatingPoint/hdlsrc/sfir_sing
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

Version History
Introduced in R2016b

See Also
`hdlcoder.FloatingPointTargetConfig`

Topics
"FPGA Floating-Point Library IP Mapping"
"Floating Point Support: Field-Oriented Control Algorithm"
"Share Floating-Point IPs"
"Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries"
"Customize Floating-Point IP Configuration"
"Generate Target-Independent HDL Code with Native Floating-Point"

customize

Class: hdlcoder.FloatingPointTargetConfig.IPConfig
Package: hdlcoder

Customize IP configuration for floating-point library

Syntax

```
fpconfig.IPConfig.customize(Name,DataType,Name,Value)
fpconfig.VendorIPConfig.customize(Name,DataType,Name,Value)
```

Description

`fpconfig.IPConfig.customize(Name,DataType,Name,Value)` customizes the `fpconfig` floating-point configuration for the native floating point library with additional options specified by one or more `Name,Value` pair arguments.

`fpconfig.VendorIPConfig.customize(Name,DataType,Name,Value)` customizes the `fpconfig` floating-point configuration for the vendor-specific floating point library chosen 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

CustomLatency — Latency of native floating-point IP
`-1` (default) | positive integer

Latency of the native floating-point IP, specified as -1 or a positive integer. This is only available for the `IPConfig` object for the native floating point library.

Example: `fpconfig.IPConfig.customize('ADDSUB','Double','CustomLatency', 6)` specifies a custom latency value of 6 for the ADDSUB IP for the native floating point library.

Latency — Latency of vendor-specific floating-point IP
`-1` (default) | positive integer

Latency of the vendor-specific floating-point IP, specified as -1 or a positive integer. This is only available for the `VendorIPConfig` object for the vendor-specific floating point library.

Example: `fpconfig.VendorIPConfig.customize('ADDSUB','Double','Latency', 6)` specifies a custom latency value of 6 for the ADDSUB IP for the vendor-specific floating point library.

ExtraArgs — Additional arguments of vendor-specific floating-point IP
`''` (default) | character vector

Additional arguments of the vendor-specific floating-point IP, specified as a character vector. This is only available for the `VendorIPConfig` object for the vendor-specific floating point library.

Example: `fpconfig.VendorIPConfig.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 Mixed-Mode Floating-Point Configuration and Generate Code

This example shows how to create a mixed-mode floating-point target configuration with the native floating point (NFP) library and a vendor-specific floating point library in HDL Coder™ and generate code. The vendor library in this example is the Altera® Megafunctions (ALTERAFPFUNCTIONS) library.

Create a Floating-Point Target Configuration

To create a floating-point configuration, set up the path to your synthesis tool by using the `hdlsetuptoolpath` function. For this example, use Altera Quartus II as your synthesis tool. To setup tools in your environment, run the `hdlsetuptoolpath` command with the unique synthesis tool path on your computer. For example, the function `quartuspath` returns the Altera Quartus II synthesis tool path.

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II','ToolPath', quartuspath);
```

```
Prepending following Altera Quartus II path(s) to the system path:
B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64
```

Load the model, `sfir_single`.

```
load_system('sfir_single')
```

Create a mixed-mode floating-point configuration, `fpconfig`, by using the function `hdlcoder.createFloatingPointTargetConfig`. The configuration `fpconfig` contains the NFP library and the Altera Megafunctions library configuration. Using NFP and vendor-specific IP together more efficiently uses resources on the FPGA, such as hardened DSP floating point adder or multiplier primitives, which allows you to fit a bigger design into the FPGA fabric.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",VendorFloatingPointLibr

fpconfig =

  FloatingPointTargetConfig with properties:

                   Library: 'NATIVEFLOATINGPOINT'
           LibrarySettings: [1×1 fpconfig.NFPLatencyDrivenMode]
                  IPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
             VendorLibrary: 'ALTERAFPFUNCTIONS'
     VendorLibrarySettings: [1×1 fpconfig.FrequencyDrivenMode]
            VendorIPConfig: [1×1 hdlcoder.FloatingPointTargetConfig.IPConfig]
```

Specify Custom NFP Library Settings

To customize the native 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'
    PartAddShiftMultiplierSize: '18x24'
                       Version: '3.0.0'
```

Specify Custom Vendor-Specific Library Settings

To customize the vendor-specific floating-point configuration, specify custom vendor library settings.

```
fpconfig.VendorLibrarySettings.InitializeIPPipelinesToZero = true;
fpconfig.VendorLibrarySettings

ans =

  FrequencyDrivenMode with properties:

    InitializeIPPipelinesToZero: 1
```

View Latency of Floating-Point IPs

The `IPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the native floating-point operators.

```
fpconfig.IPConfig

ans =
```

Name	DataType	MaxLatency	MinLatency	CustomLatency
{'ABS' }	{'DOUBLE' }	0	0	-1
{'ABS' }	{'SINGLE' }	0	0	-1
{'ACOS' }	{'SINGLE' }	23	17	-1
{'ACOSH' }	{'SINGLE' }	93	93	-1
{'ADDSUB' }	{'DOUBLE' }	11	6	-1
{'ADDSUB' }	{'HALF' }	8	4	-1
{'ADDSUB' }	{'SINGLE' }	11	6	-1
{'ASIN' }	{'SINGLE' }	23	17	-1
{'ASINH' }	{'SINGLE' }	94	94	-1
{'ATAN' }	{'SINGLE' }	36	36	-1
{'ATAN2' }	{'SINGLE' }	42	42	-1
{'ATANH' }	{'SINGLE' }	67	67	-1
{'CONVERT' }	{'DOUBLE_TO_NUMERICTYPE'}	6	3	-1
{'CONVERT' }	{'DOUBLE_TO_SINGLE' }	6	3	-1
{'CONVERT' }	{'HALF_TO_NUMERICTYPE' }	3	2	-1
{'CONVERT' }	{'HALF_TO_SINGLE' }	2	1	-1
{'CONVERT' }	{'NUMERICTYPE_TO_DOUBLE'}	6	3	-1
{'CONVERT' }	{'NUMERICTYPE_TO_HALF' }	4	2	-1
{'CONVERT' }	{'NUMERICTYPE_TO_SINGLE'}	6	6	-1
{'CONVERT' }	{'SINGLE_TO_DOUBLE' }	5	3	-1
{'CONVERT' }	{'SINGLE_TO_HALF' }	3	2	-1
{'CONVERT' }	{'SINGLE_TO_NUMERICTYPE'}	6	6	-1
{'COS' }	{'DOUBLE' }	48	48	-1
{'COS' }	{'HALF' }	14	9	-1
{'COS' }	{'SINGLE' }	27	27	-1
{'COSH' }	{'SINGLE' }	27	17	-1
{'DIV' }	{'DOUBLE' }	61	31	-1
{'DIV' }	{'HALF' }	19	10	-1
{'DIV' }	{'SINGLE' }	32	17	-1
{'EXP' }	{'HALF' }	16	9	-1
{'EXP' }	{'SINGLE' }	26	16	-1
{'FIX' }	{'DOUBLE' }	5	3	-1
{'FIX' }	{'SINGLE' }	5	3	-1
{'GAINPOW2'}	{'DOUBLE' }	2	1	-1
{'GAINPOW2'}	{'HALF' }	2	1	-1
{'GAINPOW2'}	{'SINGLE' }	2	1	-1
{'HDLRECIP'}	{'SINGLE' }	21	14	-1
{'HYPOT' }	{'SINGLE' }	33	17	-1
{'LOG' }	{'DOUBLE' }	44	34	-1
{'LOG' }	{'HALF' }	17	9	-1
{'LOG' }	{'SINGLE' }	27	20	-1
{'LOG10' }	{'HALF' }	18	10	-1
{'LOG10' }	{'SINGLE' }	27	17	-1
{'LOG2' }	{'SINGLE' }	26	16	-1
{'MINMAX' }	{'SINGLE' }	3	1	-1
{'MOD' }	{'SINGLE' }	26	16	-1
{'MUL' }	{'DOUBLE' }	9	6	-1
{'MUL' }	{'HALF' }	6	4	-1
{'MUL' }	{'SINGLE' }	8	6	-1
{'MULTADD' }	{'SINGLE' }	14	8	-1
{'POW' }	{'SINGLE' }	54	33	-1
{'POW10' }	{'SINGLE' }	26	16	-1
{'POW2' }	{'SINGLE' }	23	14	-1
{'RECIP' }	{'DOUBLE' }	60	30	-1

```
{'RECIP'   }    {'HALF'     }              19          10          -1
{'RECIP'   }    {'SINGLE'   }              31          16          -1
{'RELOP'   }    {'DOUBLE'   }               3           1          -1
{'RELOP'   }    {'HALF'     }               2           1          -1
{'RELOP'   }    {'SINGLE'   }               3           1          -1
{'REM'     }    {'SINGLE'   }              24          15          -1
{'ROUNDING'}    {'DOUBLE'   }               5           3          -1
{'ROUNDING'}    {'SINGLE'   }               5           3          -1
{'RSQRT'   }    {'DOUBLE'   }              59          33          -1
{'RSQRT'   }    {'SINGLE'   }              30          16          -1
{'SIGNUM'  }    {'DOUBLE'   }               0           0          -1
{'SIGNUM'  }    {'SINGLE'   }               0           0          -1
{'SIN'     }    {'DOUBLE'   }              34          34          -1
{'SIN'     }    {'HALF'     }              14           8          -1
{'SIN'     }    {'SINGLE'   }              27          27          -1
{'SINCOS'  }    {'SINGLE'   }              27          27          -1
{'SINH'    }    {'SINGLE'   }              30          18          -1
{'SQRT'    }    {'DOUBLE'   }              58          36          -1
{'SQRT'    }    {'HALF'     }              12           6          -1
{'SQRT'    }    {'SINGLE'   }              28          16          -1
{'TAN'     }    {'SINGLE'   }              33          33          -1
{'TANH'    }    {'SINGLE'   }              43          25          -1
{'UMINUS'  }    {'DOUBLE'   }               0           0          -1
{'UMINUS'  }    {'HALF'     }               0           0          -1
{'UMINUS'  }    {'SINGLE'   }               0           0          -1
```

The `VendorIPConfig` property stores an `IPConfig` object that displays the maximum and minimum latency values of the vendor-specific floating-point operators.

`fpconfig.VendorIPConfig`

```
ans =

     Name              DataType              Latency    ExtraArgs
    _____    _____    _____    _____

    {'ABS'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'ABS'     }    {'SINGLE'                }     -1      {0×0 char}
    {'ADDSUB'  }    {'DOUBLE'                }     -1      {0×0 char}
    {'ADDSUB'  }    {'SINGLE'                }     -1      {0×0 char}
    {'CONVERT'}    {'DOUBLE_TO_NUMERICTYPE'}     -1      {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_DOUBLE'}     -1      {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_SINGLE'}     -1      {0×0 char}
    {'CONVERT'}    {'SINGLE_TO_NUMERICTYPE'}     -1      {0×0 char}
    {'COS'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'COS'     }    {'SINGLE'                }     -1      {0×0 char}
    {'DIV'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'DIV'     }    {'SINGLE'                }     -1      {0×0 char}
    {'EXP'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'EXP'     }    {'SINGLE'                }     -1      {0×0 char}
    {'LOG'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'LOG'     }    {'SINGLE'                }     -1      {0×0 char}
    {'MUL'     }    {'DOUBLE'                }     -1      {0×0 char}
    {'MUL'     }    {'SINGLE'                }     -1      {0×0 char}
    {'MULTADD'}    {'SINGLE'                }     -1      {0×0 char}
    {'RECIP'   }    {'DOUBLE'                }     -1      {0×0 char}
```

```
{'RECIP'  }    {'SINGLE'                  }    -1    {0×0 char}
{'RELOP'  }    {'DOUBLE'                  }    -1    {0×0 char}
{'RELOP'  }    {'SINGLE'                  }    -1    {0×0 char}
{'RSQRT'  }    {'DOUBLE'                  }    -1    {0×0 char}
{'RSQRT'  }    {'SINGLE'                  }    -1    {0×0 char}
{'SIN'    }    {'DOUBLE'                  }    -1    {0×0 char}
{'SIN'    }    {'SINGLE'                  }    -1    {0×0 char}
{'SQRT'   }    {'DOUBLE'                  }    -1    {0×0 char}
{'SQRT'   }    {'SINGLE'                  }    -1    {0×0 char}
```

Customize Latency of ADDSUB Vendor IP

Using the `customize` method of either of the `IPConfig` objects, you can customize the latency of the floating-point IP and specify any additional arguments. In this example, customize the latency of the vendor IP by using the `VendorIPConfig.customize` method.

```
fpconfig.VendorIPConfig.customize('ADDSUB','Single','Latency',6);
fpconfig.VendorIPConfig
```

```
ans =

     Name               DataType               Latency    ExtraArgs
    _____    _____    _____    _____

    {'ABS'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'ABS'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'ADDSUB' }    {'DOUBLE'                  }    -1    {0×0 char}
    {'ADDSUB' }    {'SINGLE'                  }     6    {0×0 char}
    {'CONVERT'}    {'DOUBLE_TO_NUMERICTYPE'}    -1    {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_DOUBLE'}    -1    {0×0 char}
    {'CONVERT'}    {'NUMERICTYPE_TO_SINGLE'}    -1    {0×0 char}
    {'CONVERT'}    {'SINGLE_TO_NUMERICTYPE'}    -1    {0×0 char}
    {'COS'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'COS'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'DIV'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'DIV'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'EXP'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'EXP'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'LOG'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'LOG'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'MUL'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'MUL'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'MULTADD'}    {'SINGLE'                  }    -1    {0×0 char}
    {'RECIP'  }    {'DOUBLE'                  }    -1    {0×0 char}
    {'RECIP'  }    {'SINGLE'                  }    -1    {0×0 char}
    {'RELOP'  }    {'DOUBLE'                  }    -1    {0×0 char}
    {'RELOP'  }    {'SINGLE'                  }    -1    {0×0 char}
    {'RSQRT'  }    {'DOUBLE'                  }    -1    {0×0 char}
    {'RSQRT'  }    {'SINGLE'                  }    -1    {0×0 char}
    {'SIN'    }    {'DOUBLE'                  }    -1    {0×0 char}
    {'SIN'    }    {'SINGLE'                  }    -1    {0×0 char}
    {'SQRT'   }    {'DOUBLE'                  }    -1    {0×0 char}
    {'SQRT'   }    {'SINGLE'                  }    -1    {0×0 char}
```

Generate Code

For the `sfir_single` model, set the `FloatingPointTargetConfiguration` property to use the floating-point target configuration object, `fpconfig`.

```
hdlset_param('sfir_single',FloatingPointTargetConfiguration=fpconfig);
```

Set the simulation library path in order to compile and simulate the generated code with your specified simulation tool.

```
hdlset_param('sfir_single','SimulationLibPath',alterasimulationlibpath);
```

Set the `SynthesisToolChipFamily` property to `Arria10` and generate HDL code by using the `makehdl` command. The generated code files are stored in the directory path specified by `TargetDirectory` property. In this example, the generated VHDL code is stored in the `C:/MixedModeFloatingPoint/hdlsrc` folder.

```
makehdl('sfir_single/symmetric_fir',SynthesisToolChipFamily='Arria10',...
   TargetDirectory='C:/MixedModeFloatingPoint/hdlsrc')
```

```
### Generating HDL for 'sfir_single/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_single', {
### Running HDL checks on the model 'sfir_single'.
### Begin compilation of the model 'sfir_single'...
### Working on the model 'sfir_single'...
### Using B:\share\apps\HDLTools\Altera\21.1-mw-0\Windows\quartus\bin64\..\sopc_builder\bin\ip-ge
### Generating Altera(R) megafunction: alterafpf_mul_single for target frequency of 200 MHz.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sf:
### alterafpf_mul_single takes 3 cycles.
### Done.
### The code generation and optimization options you have chosen have introduced additional pipe
### The delay balancing feature has automatically inserted matching delays for compensation.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi
### Output port 1: 21 cycles.
### Output port 2: 21 cycles.
### Working on... <a href="matlab:configset.internal.open('sfir_single', 'GenerateModel')">Genera
### Begin model generation 'gm_sfir_single' ....
### Rendering DUT with optimization related changes (IO, Area, Pipelining)...
### Model generation complete.
### Generating Altera(R) megafunction: alterafpf_add_single for latency of 6.
### Found an existing generated file in a previous session: (C:\MixedModeFloatingPoint\hdlsrc\sf:
### Done.
### Begin VHDL Code Generation for 'sfir_single'.
### Working on sfir_single/symmetric_fir as C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetri
### Generating package file C:\MixedModeFloatingPoint\hdlsrc\sfir_single\symmetric_fir_pkg.vhd.
### Code Generation for 'sfir_single' completed.
### Creating HDL Code Generation Check Report file:///C:/MixedModeFloatingPoint/hdlsrc/sfir_sing:
### HDL check for 'sfir_single' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The latency of the ADDSUB IP is 6 and not the maximum latency value of 14.

Tips

Before using this function, create an object that uses native floating-point and vendor-specific floating-point libraries for your desired vendor library. Select the vendor floating-point library as

`Altera Megafunctions (ALTERA FP FUNCTIONS)`, `Altera Megafunctions (ALTFP)`, or `Xilinx LogiCORE`.

This example creates an object that uses the native floating-point library and the vendor-specific floating-point `Altera Megafunctions (ALTFP)` library.

```
fpconfig = hdlcoder.createFloatingPointTargetConfig("NativeFloatingPoint",...
    VendorFloatingPointLibrary="ALTFP");
```

Version History
Introduced in R2016b

See Also
`hdlcoder.FloatingPointTargetConfig.IPConfig` | `hdlcoder.FloatingPointTargetConfig`

Topics
"FPGA Floating-Point Library IP Mapping"
"Share Floating-Point IPs"
"Generate HDL Code for Vendor-Specific FPGA Floating-Point Target Libraries"
"Customize Floating-Point IP Configuration"

hdlcoder.WorkflowConfig class

Package: `hdlcoder`

Configure HDL code generation and deployment workflows

Description

Use the `hdlcoder.WorkflowConfig` object to set HDL workflow options for the `hdlcoder.runWorkflow` function. You can customize the `hdlcoder.WorkflowConfig` object for these workflows:

- Generic ASIC/FPGA
- FPGA-in-the-Loop (requires HDL Verifier)
- FPGA Turnkey
- IP Core Generation
- Simulink Real-Time FPGA I/O (requires Simulink Real-Time™)

A best practice is to use the HDL Workflow Advisor to configure the workflow, and then export a workflow script. The commands in the workflow script create and configure the `hdlcoder.WorkflowConfig` object. See "Run HDL Workflow with a Script".

HDL Workflow Advisor is not available in Simulink Online.

Creation

`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` arguments.

Input Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

SynthesisTool — Synthesis tool name
`'Xilinx Vivado'` (default) | `'Altera QUARTUS II'` | `'Xilinx ISE'` | `'Microchip Libero SoC'`

Name of the synthesis tool, specified as a character vector.

Example: `hdlcoder.WorkflowConfig(SynthesisTool = 'Altera QUARTUS II')` creates a workflow configuration object with `'Altera QUARTUS II'` as the synthesis tool and `'Generic ASIC/FPGA'` as the target workflow.

TargetWorkflow — Specify the target workflow
`'Generic ASIC/FPGA'` (default) | `'FPGA Turnkey'` | `'IP Core Generation'` | `'FPGA-in-the-Loop'` | `'Simulink Real-Time FPGA I/O'`

Target workflow for HDL code generation, specified as a `character vector`.

Example: `hdlcoder.WorkflowConfig(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.

`GenerateTestbench` — Generate HDL test bench
`false` (default) | `true`

Option to generate an HDL test bench in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`GenerateValidationModel` — Generate validation model
`false` (default) | `true`

Generate a validation model, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`AdditionalProjectCreationTclFiles` — Additional project creation Tcl files to include in your synthesis project
`''` (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: `'L:\file1.tcl;L:\file2.tcl;'`

`SkipPreRouteTimingAnalysis` — Skip pre-route timing analysis `logical`
`false` (default) | `true`

Skip pre-route timing analysis, specified as a `logical`. If your tool does not support early timing estimation, set to `true`.

When you enable this option, `CriticalPathSource` is set to `'post-route'`

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`IgnorePlaceAndRouteErrors` — Ignore place and route errors
false (default) | true

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

`CriticalPathSource` — Critical path source
`'pre-route'` (default) | `'post-route'`

Critical path source, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`CriticalPathNumber` — Number of critical paths to annotate
1 (default) | 2 | 3

Number of critical paths to annotate, specified as a positive integer from 1 to 3.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

`AnnotateModel` — Choose the model to annotate
original (default) | generated

Select the model that you want to annotate, specified as a character vector. You can perform the annotation on the original as well as generated model.

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.

`AllowUnsupportedToolVersion` — Allow unsupported synthesis tool version
`false` (default) | `true`

Allows you to use an unsupported synthesis tool version in the HDL Workflow Advisor, specified as a `logical`. You can set this parameter to `true` if you want to continue creating the project with the unsupported tool version. By default, HDL Coder generates an error if an unsupported tool version is detected. If you set this parameter to `true`, HDL Coder generates a warning instead. When you are using the supported synthesis tool version, this parameter value is ignored. You do not have to specify the parameter value in an HDL workflow script.

In the HDL Workflow Advisor, this option is in the **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool** task.

`ShowEndsOnly` — Show only endpoints of each critical path
`false` (default) | `true`

Show the endpoints of each critical path, omitting connecting signal lines, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Annotate Model with Synthesis Result** task.

FPGA-in-the-Loop

`ProjectFolder` — Folder for generated project files
`''` (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: `'project_file_folder'`

`RunTaskGenerateRTLCodeAndTestbench` — Enable task to generate code and test bench
`true` (default) | `false`

Enable or disable workflow task to generate code and test bench, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

`RunTaskVerifyWithHDLCosimulation` — Enable task to verify generated code with HDL cosimulation
`true` (default) | `false`

Enable or disable task to verify the generated code with HDL cosimulation, specified as a `logical`. This option takes effect only when `GenerateCosimulationModel` is `true`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Verify with HDL Cosimulation** task.

`RunTaskBuildFPGAInTheLoop` — Enable task to generate a model that contains a FIL block and a testbench around the FIL block
`true` (default) | `false`

Enable or disable task to generate a model that contains a FIL block and a testbench around the FIL block specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Build FPGA-in-the-Loop** task.

GenerateRTLCode — Generate HDL code
`true` (default) | `false`

Option to generate HDL code in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

GenerateTestbench — Generate HDL test bench
`false` (default) | `true`

Option to generate an HDL test bench in the target language, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

GenerateValidationModel — Generate validation model
`false` (default) | `true`

Generate a validation model, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and Testbench** task.

IPAddress — IP address of FPGA board
`'192.168.0.2'` (default) | character vector

IP address of the FPGA board, specified as a character vector. Default address is `'192.168.0.2'`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

MACAddress — MAC address of FPGA board
`'00-0A-35-02-21-8A'` (default) | character vector

MAC address of the FPGA board, specified as a character vector, for example `'00-0A-35-02-21-8A'`. In most cases, you do not have to change the Board MAC address. If you want to connect more than one FPGA board to a single computer, specify a unique MAC address for each additional board.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

SourceFiles — Additional HDL source files for verification
`''` (default) | character vector

Additional source files for the HDL design that you want to verify on the FPGA board, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

Connection — JTAG or Ethernet connection
'JTAG' (default) | 'Ethernet'

Ethernet or JTAG connection type to the FPGA development board, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA-in-the-Loop Implementation > Set FPGA-in-the-Loop Options** task.

RunExternalBuild — Run build process externally
true (default) | false

Option to run build process in parallel with MATLAB, specified as a logical. If this option is disabled, you cannot use MATLAB until the build is finished.

AllowUnsupportedToolVersion — Allow unsupported synthesis tool version
false (default) | true

Allows you to use an unsupported synthesis tool version in the HDL Workflow Advisor, specified as a logical. You can set this parameter to true if you want to continue creating the project with the unsupported tool version. By default, HDL Coder generates an error if an unsupported tool version is detected. If you set this parameter to true, HDL Coder generates a warning instead. When you are using the supported synthesis tool version, this parameter value is ignored. You do not have to specify the parameter value in an HDL workflow script.

In the HDL Workflow Advisor, this option is in the **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool** 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`.

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.

AllowUnsupportedToolVersion — Allow unsupported synthesis tool version
`false` (default) | `true`

Allows you to use an unsupported synthesis tool version in the HDL Workflow Advisor, specified as a `logical`. You can set this parameter to `true` if you want to continue creating the project with the unsupported tool version. By default, HDL Coder generates an error if an unsupported tool version is detected. If you set this parameter to `true`, HDL Coder generates a warning instead. When you are using the supported synthesis tool version, this parameter value is ignored. You do not have to specify the parameter value in an HDL workflow script.

In the HDL Workflow Advisor, this option is in the **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool** task.

IgnorePlaceAndRouteErrors — Ignore place and route errors
`false` (default) | `true`

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

IP Core Generation Workflow

ProjectFolder — Folder for generated project files
`''` (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: `'project_file_folder'`

ReferenceDesignToolVersion — Current reference design tool version
character vector

Current reference design tool version, specified as a character vector, for example `'2017.4'`. By default, the code generator selects a reference design tool version that is compatible with the current supported tool version. It is change this default reference design tool version, HDL Coder generates an error.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

IgnoreToolVersionMismatch — Ignore mismatch in reference design tool version
`false` (default) | `true`

Whether you want the code generator to ignore a mismatch between the reference design tool version and the supported tool version, specified as a `logical`. By default, if there is a tool version mismatch, HDL Coder generates an error. If you set this option to `true`, HDL Coder generates a warning instead.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

RunTaskGenerateRTLCodeAndIPCore — Enable task to generate code and IP core
`true` (default) | `false`

Enable or disable workflow task to generate code and IP core for embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and IP Core** task.

`AllowUnsupportedToolVersion` — Allow unsupported synthesis tool version
`false` (default) | `true`

Allows you to use an unsupported synthesis tool version in the HDL Workflow Advisor, specified as a `logical`. You can set this parameter to `true` if you want to continue creating the project with the unsupported tool version. By default, HDL Coder generates an error if an unsupported tool version is detected. If you set this parameter to `true`, HDL Coder generates a warning instead. When you are using the supported synthesis tool version, this parameter value is ignored. You do not have to specify the parameter value in an HDL workflow script.

In the HDL Workflow Advisor, this option is in the **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool** 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.

`RunTaskGenerateSoftwareInterface` — Enable task to generate software interface
`true` (default) | `false`

Enable or disable workflow task to generate a Simulink software interface model with the IP core driver blocks for embedded C code generation, specified as a `logical`. In this task, you can also verify the IP core functionality and connect to the onboard memory locations by generating a host interface model, host interface script, or both with the AXI Manager driver.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Generate Software Interface** task.

`GenerateSoftwareInterfaceModel` — Generate Simulink software interface model for SoC device
`true` (default) | `false`

Specify whether to generate a Simulink software interface model for an SoC device, specified as a `logical`. The software interface model is your original model with the AXI driver blocks replacing the parts you want to run on hardware. To use this property, you must set the `RunTaskGenerateSoftwareInterface` task to `true`. This property is not available for standalone FPGA boards.

After you generate the Simulink software interface model, you can generate C code from it by using Embedded Coder. If you do not have the Embedded Coder hardware support package for the target board installed, this property is not available. For example, if the target hardware board is a Zynq device, you must have the Embedded Coder Support Package for Xilinx Zynq® Platform installed.

`OperatingSystem` — Specify target operating system
`'Linux'` (default) | character vector

Specify the operating system for embedded processor, specified as a character vector. The operating system is board-specific. To use this property, you must set the RunTaskGenerateSoftwareInterface task to `true`.

HostTargetInterface — Specify target interface
`'Ethernet'` (default) | `'JTAG AXI Manager (HDL Verifier)'` | `'Ethernet AXI Manager (HDL Verifier)'`

Specify an interface that communicates between your host machine and the target hardware. Use one of these options.

- `'JTAG AXI Manager (HDL Verifier)'` — Use the JTAG interface to access AXI4 and AXI4-Lite registers on the target hardware. To enable this option, set `'HDLVerifierAXI'` reference design parameter to `'JTAG'` and map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interface.
- `'Ethernet AXI Manager (HDL Verifier)'` — Use the Ethernet interface to access AXI4 and AXI4-Lite registers on the target hardware. To enable this option, set `'HDLVerifierAXI'` reference design parameter to `'Ethernet'` and map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interface.
- `'Ethernet'` — Use the Ethernet interface to access the generated IP core deployed on your target hardware. This option is not available for standalone FPGA boards.

To use this property, you must set the RunTaskGenerateSoftwareInterface task to `true`.

GenerateHostInterfaceModel — Generate host interface model
`false` (default) | `true`

Specify whether to generate a host interface model, specified as a `logical`. The host interface model enables you to write to or read from the memory-mapped locations on the target hardware over a JTAG or Ethernet cable by using the AXI Manager Write and AXI Manager Read blocks. To use this property, you must set the RunTaskGenerateSoftwareInterface task to `true`.

To enable this property, specify HostTargetInterface as `'JTAG AXI Manager (HDL Verifier)'` or `'Ethernet AXI Manager (HDL Verifier)'`.

GenerateHostInterfaceScript — Generate host interface script
`true` (default) | `false`

Specify whether to generate a host interface script, specified as a `logical`. The host interface script contains commands that enable you to connect to the target hardware and to write to or read from the generated IP core by using the AXI driver blocks or the AXI Manager. To use this property, you must set the RunTaskGenerateSoftwareInterface task to `true`.

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. The `IP Core Generation` workflow uses an out-of-context (OOC) workflow. This workflow synthesizes the IP in the reference design out of context from the top-level design. You can reuse this cache in subsequent project runs, which reduces reference design synthesis time. To learn more, see "IP Caching for Faster Reference Design Synthesis".

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.

EnableDesignCheckpoint — Use routed design checkpoint file
true (default) | false

Option to expedite bitstream build process by using a routed design checkpoint file from a previous build. specified as a logical. If this option is not selected, you cannot use **DefaultCheckpointFile**.

Example: hWC.EnableDesignCheckPoint = true;

DefaultCheckpointFile — Option to use default or custom routed design checkpoint file
Default (default) | Custom

Option to specify whether to use the default checkpoint file location or use a custom checkpoint file.

Example: hWC.DefaultCheckPointFile = 'Custom'

RoutedDesignCheckpointFilePath — Option to specify file path for routed design checkpoint file
'hdl_prj\checkpoint\system_routed.dcp' (default) | 'C:\example_project\checkpoint\custom_name.dcp'

Option to specify the path to the custom routed design checkpoint file. If **DefaultCheckpointFile** is set to Default, you cannot specify a custom path.

Example: hWC.RoutedDesignCheckFilePath = 'c:\example_project\checkpoint\example_file.dcp'

MaxNumOfCoresForBuild — Maximum number of PC cores to use for bitstream build
'synthesis tool default' (default) | positive integer between 2 and 32 both included

Option to expedite the bitstream build process by using specified number of PC cores during bitstream build. If you set the option to 'synthesis tool default', the number of cores specified in the synthesis tool will be used during the bitstream build.

Example: hWC.MaxNumOfCoresForBuild = '12';

ReportTimingFailure — Report timing failures as warnings or errors
hdlcoder.ReportTimingFailure.Error (default) | hdlcoder.ReportTimingFailure.Warning

Select whether to report timing failures when generating the FPGA bitstream, specified as one of these values:

hdlcoder.ReportTimingFailure.Error (default)	Report timing failures as errors by default.
hdlcoder.ReportTimingFailure.Warning	Report timing failures as errors instead of warnings. Use this option if you have implemented the custom logic to resolve timing violations in your design.

TclFileForSynthesisBuild — Use custom or default synthesis tool build script
hdlcoder.BuildOption.Default (default) | hdlcoder.BuildOption.Custom

Select whether to use a custom or default synthesis tool build script, specified as one of these values:

hdlcoder.BuildOption.Default (default)	Use the default build script.
hdlcoder.BuildOption.Custom	Use a custom build script instead of the default build script.

CustomBuildTclFile — Custom synthesis tool build script file
'' (default) | character vector

Full path to a custom synthesis tool build Tcl script file, specified as a character vector. The contents of your custom Tcl file are inserted between the Tcl commands that open and close the project. If TclFileForSynthesisBuild is set to hdlcoder.BuildOption.Custom, you must specify a file.

If you want to generate a bitstream, the bitstream generation Tcl command must refer to the top file wrapper name and location either directly or implicitly. For example, this Xilinx Vivado Tcl command generates a bitstream and implicitly refers to the top file name and location:

```
launch_runs impl_1 -to_step write_bitstream
```

Example: 'C:\Temp\work\build.tcl'

Simulink Real-Time FPGA I/O

ProjectFolder — Folder for generated project files
'' (default) | character vector

Path to the folder where your generated project files are saved, specified as a character vector.

Example: 'project_file_folder'

ReferenceDesignToolVersion — Current reference design tool version
character vector

Current reference design tool version, specified as a character vector, for example '2017.4'. By default, the code generator selects a reference design tool version that is compatible with the current supported tool version. It is change this default reference design tool version, HDL Coder generates an error.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

IgnoreToolVersionMismatch — Ignore mismatch in reference design tool version
`false` (default) | `true`

Whether you want the code generator to ignore a mismatch between the reference design tool version and the supported tool version, specified as a `logical`. By default, if there is a tool version mismatch, HDL Coder generates an error. If you set this option to `true`, HDL Coder generates a warning instead.

In the HDL Workflow Advisor, this setting is in the **HDL Workflow Advisor > Set Target > Set Target Reference Design** task.

RunTaskGenerateRTLCodeAndIPCore — Enable task to generate code and IP core
`true` (default) | `false`

Enable or disable workflow task to generate code and IP core for embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code and IP Core** task.

AllowUnsupportedToolVersion — Allow unsupported synthesis tool version
`false` (default) | `true`

Allows you to use an unsupported synthesis tool version in the HDL Workflow Advisor, specified as a `logical`. You can set this parameter to `true` if you want to continue creating the project with the unsupported tool version. By default, HDL Coder generates an error if an unsupported tool version is detected. If you set this parameter to `true`, HDL Coder generates a warning instead. When you are using the supported synthesis tool version, this parameter value is ignored. You do not have to specify the parameter value in an HDL workflow script.

In the HDL Workflow Advisor, this option is in the **HDL Workflow Advisor > Set Target > Set Target Device and Synthesis Tool** task.

RunTaskGenerateRTLCode — Enable task to generate RTL code and HDL top-level wrapper
`true` (default) | `false`

Enable or disable workflow task to generate RTL code and an HDL top-level wrapper, specified as a `logical`. When enabled, this task also generates a constraint file that contains pin mapping information and clock constraints.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > HDL Code Generation > Generate RTL Code** task.

RunTaskCreateProject — Enable task to create embedded system tool project
`true` (default) | `false`

Enable or disable workflow task to create an embedded system tool project, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Create Project** task.

RunTaskPerformLogicSynthesis — Enable task to launch synthesis tool and run logic synthesis
`true` (default) | `false`

Enable or disable task to launch the synthesis tool and run logic synthesis, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Logic Synthesis** task.

`RunTaskPerformMapping` — Enable task to map synthesized logic to target device
`true` (default) | `false`

Enable or disable task to map the synthesized logic to the target device, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

`RunTaskPerformPlaceAndRoute` — Enable task to run place and route process
`true` (default) | `false`

Enable or disable task to run the place and route process, specified as a `logical`. This task is available only when your synthesis tool is Xilinx ISE or Altera Quartus II.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and Route** task.

`RunTaskGenerateProgrammingFile` — Enable task to generate FPGA programming file
`true` (default) | `false`

Enable or disable task to generate an FPGA programming file, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Programming File** task.

`RunTaskGenerateSimulinkRealTimeInterface` — Enable task to generate a model that contains an interface subsystem that you can plug into a Simulink Real-Time model
`true` (default) | `false`

Enable or disable task to generate a Simulink Real-Time model that contains an interface subsystem, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Download to Target > Generate Simulink Real-Time Interface** task.

`Objective` — Synthesis tool objective
`hdlcoder.Objective.None` (default) | `hdlcoder.Objective.SpeedOptimized` | `hdlcoder.Objective.AreaOptimized` | `hdlcoder.Objective.CompileOptimized`

High-level synthesis tool objective, specified as one of these values.

`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".

AdditionalProjectCreationTclFiles — Additional project creation Tcl files to include in your synthesis project
`' '` (default) | character vector

Additional project creation Tcl files that you want to include in your synthesis project, specified as a character vector.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Create Project** task.

Example: `'L:\file1.tcl;L:\file2.tcl;'`

SkipPreRouteTimingAnalysis — Skip pre-route timing analysis `logical`
`false` (default) | `true`

Skip pre-route timing analysis, specified as a `logical`. If your tool does not support early timing estimation, set to `true`.

When you enable this option, `CriticalPathSource` is set to `'post-route'`

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Mapping** task.

IgnorePlaceAndRouteErrors — Ignore place and route errors
`false` (default) | `true`

Ignore place and route errors, specified as a `logical`.

In the HDL Workflow Advisor, this option is part of the **HDL Workflow Advisor > FPGA Synthesis and Analysis > Perform Synthesis and P/R > Perform Place and route** task.

RunTaskBuildFPGABitstream — Enable task to generate bitstream for embedded system
`true` (default) | `false`

Enable or disable workflow task to generate a bitstream for the embedded system, specified as a `logical`.

In the HDL Workflow Advisor, this task is the **HDL Workflow Advisor > Embedded System Integration > Build FPGA Bitstream** task.

EnableDesignCheckpoint — Use routed design checkpoint file
`true` (default) | `false`

Option to expedite bitstream build process by using a routed design checkpoint file from a previous build. specified as a `logical`. If this option is not selected, you cannot use **DefaultCheckpointFile**.

Example: `hWC.EnableDesignCheckPoint = true;`

DefaultCheckpointFile — Option to use default or custom routed design checkpoint file
Default (default) | Custom

Option to specify whether to use the default checkpoint file location or use a custom checkpoint file.

Example: hWC.DefaultCheckPointFile = 'Custom'

RoutedDesignCheckpointFilePath — Option to specify file path for routed design checkpoint file
'hdl_prj\checkpoint\system_routed.dcp' (default) | 'C:\example_project\checkpoint\custom_name.dcp'

Option to specify the path to the custom routed design checkpoint file. If **DefaultCheckpointFile** is set to Default, you cannot specify a custom path.

Example: hWC.RoutedDesignCheckFilePath = 'c:\example_project\checkpoint\example_file.dcp'

MaxNumOfCoresForBuild — Maximum number of PC cores to use for bitstream build
'synthesis tool default' (default) | positive integer between 2 and 32 both included

Option to expedite the bitstream build process by using specified number of PC cores during bitstream build. If you set the option to 'synthesis tool default', the number of cores specified in the synthesis tool will be used during the bitstream build.

Example: hWC.MaxNumOfCoresForBuild = '12';

ReportTimingFailure — Report timing failures as warnings or errors
hdlcoder.ReportTimingFailure.Error (default) |
hdlcoder.ReportTimingFailure.Warning

Select whether to report timing failures when generating the FPGA bitstream, specified as one of these values:

hdlcoder.ReportTimingFailure.Error (default)	Report timing failures as errors by default.
hdlcoder.ReportTimingFailure.Warning	Report timing failures as errors instead of warnings. Use this option if you have implemented the custom logic to resolve timing violations in your design.

Methods

Public Methods

clearAllTasks	Disable all tasks in workflow
export	Generate MATLAB script that recreates the workflow configuration
setAllTasks	Enable all tasks in workflow
validate	Check property values in HDL Workflow CLI configuration object

Examples

Configure and Run Generic ASIC/FPGA Workflow with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a generic ASIC/FPGA workflow script that targets a Xilinx Virtex® 7 device and uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
% Export Workflow Configuration Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 14:42:37 on 29/03/2018
% This script was generated using the following parameter values:
%     Filename  : 'S:\generic_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('sfir_fixed', 'GenerateHDLTestBench', 'off');
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Virtex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7vx485t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', '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 Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskRunSynthesis = true;
hWC.RunTaskRunImplementation = false;
hWC.RunTaskAnnotateModelWithSynthesisResult = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskRunSynthesis' Task
hWC.SkipPreRouteTimingAnalysis = false;

% Set properties related to 'RunTaskRunImplementation' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Set properties related to 'RunTaskAnnotateModelWithSynthesisResult' Task
```

```
hWC.CriticalPathSource = 'pre-route';
hWC.CriticalPathNumber =  1;
hWC.AnnotateModel = 'original';
hWC.ShowAllPaths = false;
hWC.ShowDelayData = true;
hWC.ShowUniquePaths = false;
hWC.ShowEndsOnly = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `generic_workflow_example.m`, at the command line, enter:

```
generic_workflow_example
```

Configure and Run FPGA-in-the-Loop with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is an FPGA-in-the-Loop workflow script that targets a Xilinx Virtex 5 development board and uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
%-------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 15:11:23 on 04/05/2018
% This script was generated using the following parameter values:
%     Filename  : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow_
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
```

```matlab
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 25);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Xilinx Kintex-7 KC705 development board');
hdlset_param('sfir_fixed', 'Workflow', 'FPGA-in-the-Loop');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','FPGA-in-the-Loop

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';

%Set Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndTestbench = true;
hWC.RunTaskVerifyWithHDLCosimulation = false;
hWC.RunTaskBuildFPGAInTheLoop = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndTestbench' Task
hWC.GenerateRTLCode = true;
hWC.GenerateTestbench = false;
hWC.GenerateValidationModel = false;

% Set properties related to 'RunTaskBuildFPGAInTheLoop' Task
hWC.IPAddress = '192.168.0.2';
hWC.MACAddress = '00-0A-35-02-21-8A';
hWC.SourceFiles = '';
hWC.Connection = 'Ethernet';
hWC.RunExternalBuild = true;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
hdlcoder.runWorkflow('hdlcoderUARTServoControllerExample/UART_Servo_on_FPGA', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `FIL_workflow_example.m`, at the command line, enter:

`fil_workflow_example`

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.

```
%---------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.14 (R2023a) at 17:54:39 on 04/01/2023
% This script was generated using the following parameter values:
%     Filename  : 'S:\turnkey_workflow_example.m'
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'hdlcoder_led_blinking/led_counter'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','hdlcoder_led_blinking/led_counter');
%---------------------------------------------------------------------------

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('hdlcoder_led_blinking/led_counter');

%% Model HDL Parameters
%% Set Model 'hdlcoder_led_blinking' 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', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_led_blinking', 'ResourceReport', 'on');
hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolChipFamily', 'Virtex5');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolDeviceName', 'xc5vsx50t');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolPackageName', 'ff1136');
hdlset_param('hdlcoder_led_blinking', 'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoder_led_blinking', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_led_blinking', 'TargetFrequency', 100);
hdlset_param('hdlcoder_led_blinking', 'TargetPlatform', 'Xilinx Virtex-5 ML506 development board');
hdlset_param('hdlcoder_led_blinking', 'Traceability', 'on');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'FPGA Turnkey');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency',...
 'IOInterface', 'DIP Switches [0:7]');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency',...
 'IOInterfaceMapping', '[0:3]');

% Set Inport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction',...
 'IOInterface', 'DIP Switches [0:7]');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction',...
 'IOInterfaceMapping', '[4]');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/LED',...
 'IOInterface', 'LEDs General Purpose [0:7]');
hdlset_param('hdlcoder_led_blinking/led_counter/LED',...
 'IOInterfaceMapping', '[0:7]');

% Set Outport HDL parameters
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back',...
 'IOInterface', 'No Interface Specified');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back',...
 'IOInterfaceMapping', '');


%% 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.RunTaskGenerateRTLCode = true;
```

```
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskProgramTargetDevice = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskPerformMapping' Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set properties related to 'RunTaskPerformPlaceAndRoute' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('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 `turnkey_workflow_example.m`, at the command line, enter:

```
turnkey_workflow_example
```

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');
```

```matlab
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"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', ...
    'IOInterfaceOptions', {'RegisterInitialValue', 5});

% 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"');
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_direction, ...
    'IOInterfaceOptions', {'RegisterInitialValue', 1});

% 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"');
hdlset_param('hdlcoder_led_blinking/led_counter/Read_back, ...
    'IOInterfaceOptions', {'RegisterInitialValue', 3});

%% 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 Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskGenerateSoftwareInterface = 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
```

Configure and Run Simulink Real-Time FPGA I/O Workflow for ISE-Based Boards with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a `Simulink Real-Time FPGA I/O` workflow script that targets the `Speedgoat IO331` board that uses the Xilinx ISE synthesis tool.

Open and view your exported HDL workflow script.

```
%-------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:14 on 08/05/2018
% This script was generated using the following parameter values:
%     Filename  : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow_
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
```

```
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx ISE');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Spartan6');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc6slx150');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fgg676');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-3');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 75);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat IO331');
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx ISE','TargetWorkflow','Simulink Real-Time I

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '';
hWC.IgnoreToolVersionMismatch = false;

%Set Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCode = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskPerformLogicSynthesis = true;
hWC.RunTaskPerformMapping = true;
hWC.RunTaskPerformPlaceAndRoute = true;
hWC.RunTaskGenerateProgrammingFile = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';

% Set properties related to 'RunTaskPerformMapping' Task
hWC.SkipPreRouteTimingAnalysis = true;

% Set properties related to 'RunTaskPerformPlaceAndRoute' Task
hWC.IgnorePlaceAndRouteErrors = false;

% Validate the Workflow Configuration Object
hWC.validate;
```

```
%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

```
slrt_workflow_example
```

Configure and Run Simulink Real-Time FPGA I/O Workflow for Vivado-Based Boards with a Script

This example shows how to configure and run an exported HDL workflow script.

To generate an HDL workflow script, configure and run the HDL Workflow Advisor with your Simulink design, then export the script.

This script is a `Simulink Real-Time FPGA I/O` workflow script that targets the `Speedgoat IO333-325K` board that uses the Xilinx Vivado synthesis tool.

Open and view your exported HDL workflow script.

```
%-------------------------------------------------------------------------
% HDL Workflow Script
% Generated with MATLAB 9.5 (R2018b Prerelease) at 18:14:33 on 08/05/2018
% This script was generated using the following parameter values:
%     Filename  : 'C:\Users\ggnanase\Desktop\R2018b\18b_models\ipcore_timing_failure\hdlworkflow_
%     Overwrite : true
%     Comments  : true
%     Headers   : true
%     DUT       : 'sfir_fixed/symmetric_fir'
% To view changes after modifying the workflow, run the following command:
% >> hWC.export('DUT','sfir_fixed/symmetric_fir');
%-------------------------------------------------------------------------

%% Load the Model
load_system('sfir_fixed');

%% Restore the Model to default HDL parameters
%hdlrestoreparams('sfir_fixed/symmetric_fir');

%% Model HDL Parameters
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Kintex7');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7k325t');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'ffg900');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-2');
hdlset_param('sfir_fixed', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('sfir_fixed', 'TargetFrequency', 100);
hdlset_param('sfir_fixed', 'TargetPlatform', 'Speedgoat IO333-325K');
```

```matlab
hdlset_param('sfir_fixed', 'Workflow', 'Simulink Real-Time FPGA I/O');


%% Workflow Configuration Settings
% Construct the Workflow Configuration Object with default settings
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','Simulink Real-Tir

% Specify the top level project directory
hWC.ProjectFolder = 'hdl_prj';
hWC.ReferenceDesignToolVersion = '2017.4';
hWC.IgnoreToolVersionMismatch = false;

%Set Properties related to synthesis tool version
hWC.AllowUnsupportedToolVersion  = true;

% Set Workflow tasks to run
hWC.RunTaskGenerateRTLCodeAndIPCore = true;
hWC.RunTaskCreateProject = true;
hWC.RunTaskBuildFPGABitstream = true;
hWC.RunTaskGenerateSimulinkRealTimeInterface = true;

% Set properties related to 'RunTaskGenerateRTLCodeAndIPCore' Task
hWC.IPCoreRepository = '';
hWC.GenerateIPCoreReport = true;
hWC.GenerateIPCoreTestbench = false;
hWC.CustomIPTopHDLFile = '';
hWC.AXI4RegisterReadback = false;
hWC.IPDataCaptureBufferSize = '128';

% Set properties related to 'RunTaskCreateProject' Task
hWC.Objective = hdlcoder.Objective.None;
hWC.AdditionalProjectCreationTclFiles = '';
hWC.EnableIPCaching = true;

% Set properties related to 'RunTaskBuildFPGABitstream' Task
hWC.RunExternalBuild = false;
hWC.TclFileForSynthesisBuild = hdlcoder.BuildOption.Default;
hWC.CustomBuildTclFile = '';
hWC.ReportTimingFailure = hdlcoder.ReportTiming.Error;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('sfir_fixed/symmetric_fir', hWC);
```

Optionally, edit the script.

For example, enable or disable tasks in the `hdlcoder.WorkflowConfig` object, `hWC`.

Run the HDL workflow script.

For example, if the script file name is `slrt_workflow_example.m`, at the command line, enter:

slrt_workflow_example

Version History
Introduced in R2015b

See Also

Functions
hdlcoder.runWorkflow

Topics
"Run HDL Workflow with a Script"

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 pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

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'

Version History
Introduced in R2015b

See Also

Classes
hdlcoder.WorkflowConfig

Topics
"Run HDL Workflow with a Script"

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.

Version History
Introduced in R2015b

See Also

Functions
clearAllTasks

Classes
hdlcoder.WorkflowConfig

Topics
"Run HDL Workflow with a Script"

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.

Version History
Introduced in R2015b

See Also

Functions
setAllTasks

Classes
hdlcoder.WorkflowConfig

Topics
"Run HDL Workflow with a Script"

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 the required properties, and that the property values have valid data types. If validation fails, you get an error message.

Version History
Introduced in R2015b

See Also
hdlcoder.WorkflowConfig

Topics
"Run HDL Workflow with a Script"

hdlcoder.runWorkflow

Run HDL code generation and deployment workflow

Syntax

```
hdlcoder.runWorkflow(DUT)
hdlcoder.runWorkflow(DUT,workflow_config)
hdlcoder.runWorkflow(DUT,workflow_config, Name,Value)
```

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".

`hdlcoder.runWorkflow(DUT,workflow_config, Name,Value)` runs the HDL code generation and deployment workflow according to the specified workflow configuration, `workflow_config`, with additional options specified by one or more `Name,Value` arguments.

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');
```
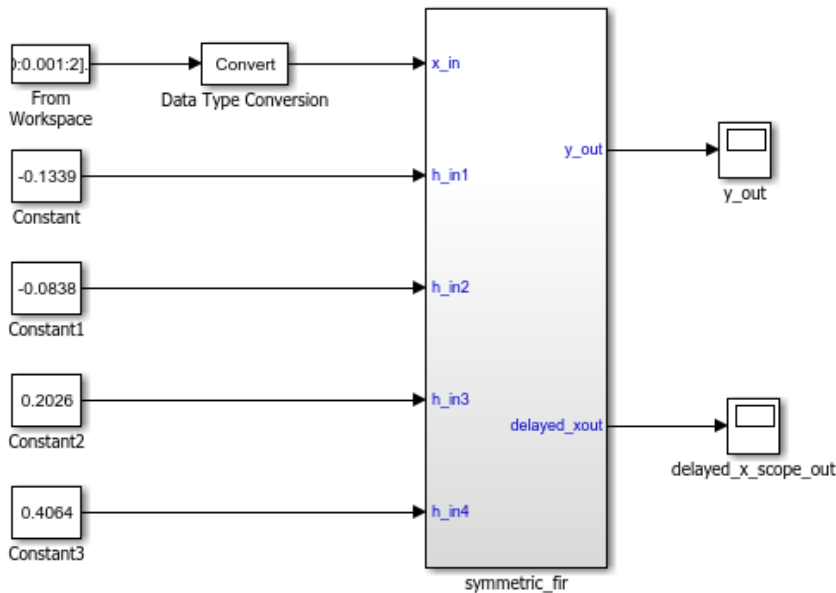

Copyright 2007 The MathWorks, Inc.

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_fir.vhd.
### Creating HDL Code Generation Check Report file://C:\Temp\hdl_prj\hdlsrc\sfir_fixed\symmetric_
### 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.openTargetTool('L:\X
### Generated logfile: <a href="matlab:edit('C:\Temp\hdl_prj\hdlsrc\sfir_fixed\workflow_task_Crea
### Task "Create Project" successful.
### ++++++++++++++ Task Run Synthesis ++++++++++++++
### Generated logfile: <a href="matlab:edit('C:\Temp\hdl_prj\hdlsrc\sfir_fixed\workflow_task_RunS
### 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_fir/y_out
### Click <a href="matlab:hdlannotatepath('reset')">here</a> to reset highlighting.
### Workflow complete.
```
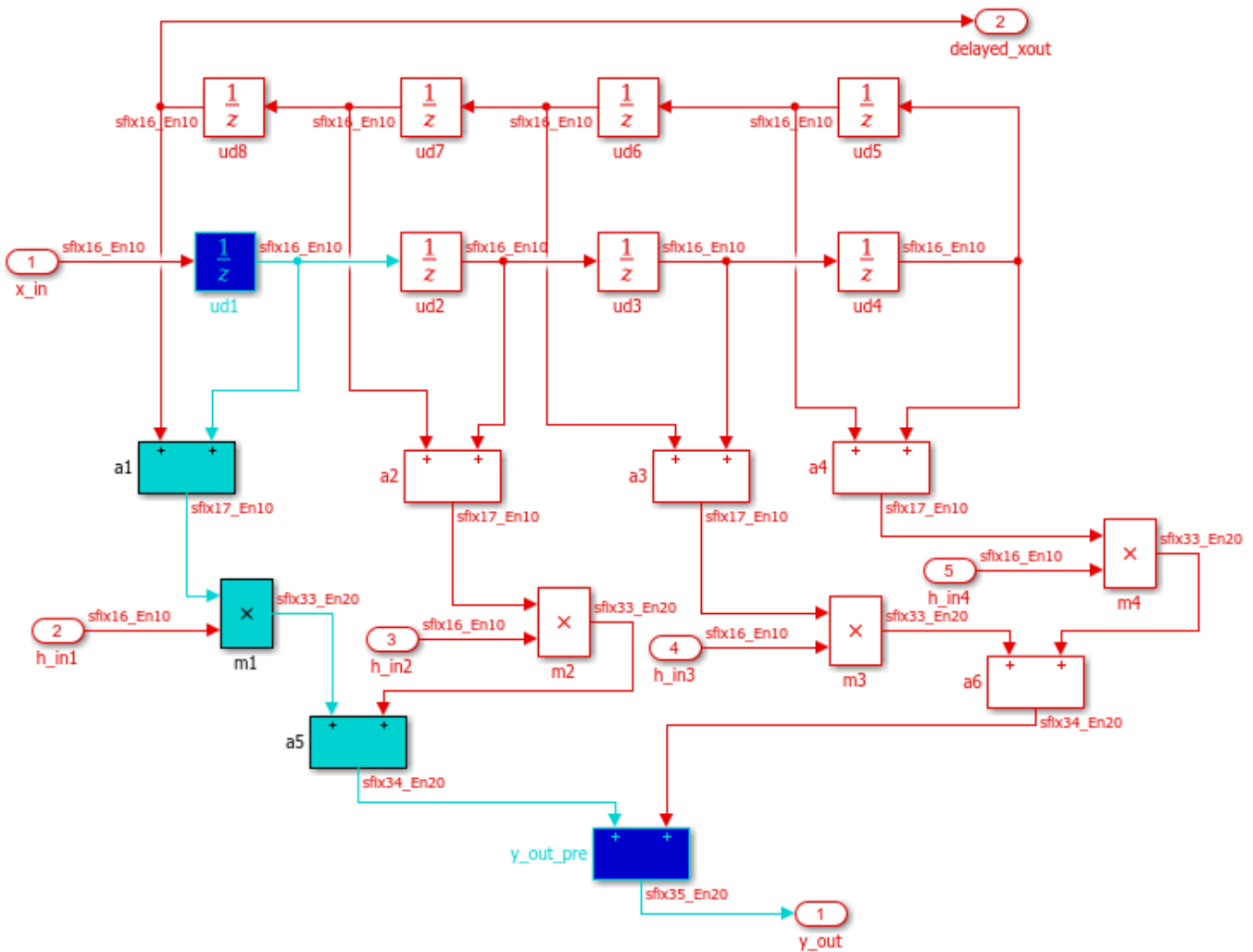
Input Arguments

DUT — Full path to DUT
'' (default) | character vector

Full path to the DUT, specified as a character vector.

Example: `'hdlcoder_led_blinking/led_counter'`

workflow_config — Workflow configuration
hdlcoder.WorkflowConfig

HDL code generation and deployment workflow configuration, specified as an
hdlcoder.WorkflowConfig object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Verbosity','on'`

Verbosity — Level of detail
`'off'` (default) | `'on'`

When `Verbosity` is left to default value of `'off'`, minimal code generation progress messages are displayed as the code generation and deployment workflow runs. When `Verbosity` is set to `'on'`, more detailed progress messages are displayed.

Version History
Introduced in R2015b

See Also

Functions
setAllTasks | clearAllTasks

Classes
hdlcoder.WorkflowConfig

Topics
"Run HDL Workflow with a Script"

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 TimingStrategy 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 TimingStrategy 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.

Creation

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.Exploratio nMode.BestFrequency`	Optimizes the design to try to achieve the maximum clock frequency `hdlcoder.OptimizationConfig.Exploratio nMode.BestFrequency` is the default.
`hdlcoder.OptimizationConfig.Exploratio nMode.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`.

TimingStrategy — Timing strategy used to calculate the critical path
'Synthesis' (default) | 'CriticalPathEstimation'

Timing strategy used to calculate the estimated or actual critical path of your design during automatic iterative HDL design optimization. The timing strategy can be set to either `Synthesis` or `CriticalPathEstimation`.

`Synthesis`	Uses synthesis as the timing strategy to determine the actual critical path of your design. Using `Synthesis` as your timing strategy can cause a long run time for `hdlcoder.OptimizeDesign`.

`CriticalPathEstimation`	Use critical path estimation as the timing strategy to determine the estimated critical path of your design. Critical path estimation does not run synthesis nor does it generate HDL code, which significantly shortens the runtime of `hdlcoder.OptimizeDesign`, while still maintaining optimization accuracy for calculating the critical path.

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 Vivado', ...
                   '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 Vivado');
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 Vivado', ...
                   '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 Vivado');
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 Vivado', ...
                  '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 Vivado');
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.

Configure hdlcoder.optimizeDesign for Critical Path Estimation

Use CriticalPathEstimation as the TimingStrategy for the optimization configuration object to shorten the run time for hdlcoder.optimizeDesign.

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 Vivado', ...
                   'SynthesisToolChipFamily','Zynq', ...
                   'SynthesisToolDeviceName','xc7z030', ...
                   'SynthesisToolPackageName','fbg484', ...
                   'SynthesisToolSpeedValue','-1')
```

Create an optimization configuration object oc.

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

Set the iteration limit to 10.

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

Set the timing strategy to CriticalPathEstimation.

```
oc.TimingStrategy = 'CriticalPathEstimation';
```

Optimize the model by using critical path estimation as the timing strategy.

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

```
%% Set Model 'sfir_fixed' HDL parameters
hdlset_param('sfir_fixed', 'HDLSubsystem', 'sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-1');

Iteration 0
Generate and synthesize HDL code ...
...
(CP ns) 9.94    (Constraint ns) 5.46    (Elapsed s) 4.80
Iteration 9
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 6.21 ns        Elapsed : 31.33 s
Iteration 0: (CP ns) 9.94    (Constraint ns) 5.46    (Elapsed s) 4.80
Iteration 1: (CP ns) 8.81    (Constraint ns) 5.46    (Elapsed s) 7.90
Iteration 2: (CP ns) 8.77    (Constraint ns) 7.14    (Elapsed s) 11.14
Iteration 3: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 14.74
Iteration 4: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 17.99
Iteration 5: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 21.25
Iteration 6: (CP ns) 7.34    (Constraint ns) 7.14    (Elapsed s) 24.75
Iteration 7: (CP ns) 6.21    (Constraint ns) 7.14    (Elapsed s) 27.95
Iteration 8: (CP ns) 6.21    (Constraint ns) 5.96    (Elapsed s) 31.33
```

For comparison to the previous results, set the timing strategy to Synthesis (its default) and optimize the model by using hdlcoder.optimizeDesign.

```
oc.TimingStrategy = 'Synthesis';
hdlcoder.optimizeDesign(model,oc)
```

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

```
hdlset_param('sfir_fixed', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('sfir_fixed', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('sfir_fixed', 'SynthesisToolDeviceName', 'xc7z030');
hdlset_param('sfir_fixed', 'SynthesisToolPackageName', 'fbg484');
hdlset_param('sfir_fixed', 'SynthesisToolSpeedValue', '-1');

Iteration 0
Generate and synthesize HDL code ...
(CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 252.66
Iteration 3
Generate and synthesize HDL code ...
Exiting because critical path cannot be further improved.
Summary report: summary.html
Achieved Critical Path (CP) Latency : 5.49 ns        Elapsed : 704.51 s
Iteration 0: (CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 252.66
Iteration 1: (CP ns) 9.74    (Constraint ns) 5.21    (Elapsed s) 479.88
Iteration 2: (CP ns) 5.49    (Constraint ns) 7.95    (Elapsed s) 704.51
```

	Synthesis Timing Strategy	Critical Path Estimation Timing Strategy
Total elapsed time, in seconds	704.51	31.33

When you use `CriticalPathEstimation` as the `TimingStrategy`, the time to run `hdlcoder.optimizeDesign` is shorter.

The critical path calculated by using `CriticalPathEstimation` as the `TimingStrategy` is the estimated critical path delay of the optimized model. The critical path calculated by using `Synthesis` as the `TimingStrategy` is the actual critical path delay of the optimized model.

Version History
Introduced in R2014a

See Also

Functions
`hdlcoder.optimizeDesign`

Topics
"Automatic Iterative Optimization"

Functions for HDL Code Generation from MATLAB

codegen

Generate HDL or SystemC 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 or SystemC code from MATLAB code.

`codegen -config hdlcfg -float2fixed fixptcfg matlab_design_name` converts floating-point MATLAB code to fixed-point code, then generates HDL or SystemC 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 or SystemC code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Input Arguments

`hdlcfg` — HDL code generation configuration
`coder.HdlConfig`

HDL code generation configuration options, specified as a `coder.HdlConfig` object.

Create a `coder.HdlConfig` object using the HDL `coder.config` function.

`matlab_design_name` — MATLAB design function name
character vector

Name of top-level MATLAB function for which you want to generate HDL code.

`fixptcfg` — Floating-point to fixed-point conversion configuration
`coder.FixptConfig`

Floating-point to fixed-point conversion configuration options, specified as a `coder.FixptConfig` object.

Use `fixptcfg` when generating HDL code from floating-point MATLAB code. Create a `coder.FixptConfig` object using the HDL `coder.config` function.

Version History
Introduced in R2013a

See Also
`coder.FixPtConfig` | `coder.HdlConfig` | `coder.config`

Topics
"Generate HDL Code from MATLAB Code Using the Command Line Interface"
"Get Started with MATLAB to SystemC Workflow Using the Command Line Interface"

coder.approximation

Create function replacement configuration object

Syntax

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

Description

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the`codegen` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

Examples

Replace `log` Function with Default Lookup Table

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```

Replace `log` Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...
'FunctionNamePrefix','log_replace_');
```

Replace `log` Function with Optimized Lookup Table

Create a function replacement configuration object using the `'OptimizeLUTSize'` option to specify to replace the `log` function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
    logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...
'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, saturateExp, with an optimized lookup table.

Create a custom function, saturateExp.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the saturateExp function with an optimized lookup table. Because the saturateExp function is not listed as a function for which coder.approximation can generate an approximation automatically, you must specify the CandidateFunction property.

```
saturateExp = @(x) 1/(1+exp(-x));
custAppx = coder.approximation('Function','saturateExp',...
'CandidateFunction', saturateExp,...
'NumberOfPoints',50,'InputRange',[0,10]);
```

Input Arguments

function_name — Name of the function to replace
'acos' | 'acosd' | 'acosh' | 'acoth' | 'asin' | 'asind' | 'asinh' | 'atan' | 'atand' | 'atanh' | 'cos' | 'cosd' | 'cosh' | 'erf ' | 'erfc' | 'exp' | 'log' | 'normcdf' | 'reallog' | 'realsqrt' | 'reciprocal' | 'rsqrt' | 'sin' | 'sinc' | 'sind' | 'sinh' | 'sqrt' | 'tan' | 'tand'

Name of function to replace, specified as a string. The function must be one of the listed functions.

Example: 'sqrt'

Data Types: char

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: '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

Property	Default Value
OptimizeLUTSize	false
OptimizeIterations	25

Version History
Introduced in R2014b

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"

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 or SystemC 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 or SystemC code. In this example, the MATLAB design function name is `mlhdlc_dti`.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Version History
Introduced in R2013a

See Also
coder.HdlConfig | coder.FixPtConfig | codegen

Topics
"Generate HDL Code from MATLAB Code Using the Command Line Interface"

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

Add 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;
```

Derive ranges and generate fixed-point code

```
fiaccel -float2fixed cfg dti;
```

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
`coder.FixPtConfig` | `coder.config` | `codegen`

clearDesignRangeSpecifications

Class: `coder.FixPtConfig`
Package: `coder`

Clear all design range specifications

Syntax

```
clearDesignRangeSpecifications()
```

Description

`clearDesignRangeSpecifications()` clears all design range specifications.

Examples

Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

getDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Get design range specifications for parameter

Syntax

`[designMin, designMax] = getDesignRangeSpecification(fcnName,paramName)`

Description

`[designMin, designMax] = getDesignRangeSpecification(fcnName,paramName)` gets the minimum and maximum values specified for the parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Function name
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Output Arguments

`designMin` — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Get Design Range Specifications

```matlab
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the  design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')

designMin =

    -1


designMax =

     1
```

hasDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Determine whether parameter has design range

Syntax

```
hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)
```

Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)` returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

Input Arguments

`fcnName` — Name of function
string

Function name, specified as a string.

Example: 'dti'

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Example: 'dti'

Data Types: `char`

Output Arguments

`hasDesignRange` — Parameter has design range
true | false

Parameter has design range, returned as a boolean.

Data Types: `logical`

Examples

Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
```

```
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')

hasDesignRanges =

    1
```

removeDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Remove design range specification from parameter

Syntax

`removeDesignRangeSpecification(fcnName,paramName)`

Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Name of function
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Examples

Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

Classes 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.

Creation

`fixptcfg = coder.config('fixpt')` creates a `coder.FixPtConfig` object for floating-point to fixed-point conversion.

Properties

ComputeDerivedRanges — Enable derived range analysis
`false` (default) | `true`

Enable derived range analysis, specified as `true` or `false`.

ComputeSimulationRanges — Enable collection and reporting of simulation range data
`true` (default) | `false`

Enable collection and reporting of simulation range data, specified as `true` or `false`. 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.

DefaultFractionLength — Default fixed-point fraction length
`4` (default) | positive integer

Default fixed-point fraction length, specified as a positive integer.

DefaultSignedness — Default signedness of variables in the generated code
`'Automatic'` (default) | `'Signed'` | `'Unsigned'`

Default signedness of variables in the generated code, specified as `'Automatic'`, `'Signed'`, or `'Unsigned'`.

DefaultWordLength — Default fixed-point word length
`14` (default) | positive integer

Default fixed-point word length, specified as a positive integer.

DetectFixptOverflows — Enable detection of overflows using scaled doubles
`false` (default) | `true`

Enable detection of overflows using scaled doubles, specified as `true` or `false`.

fimath — fimath properties to use for conversion
fimath('RoundingMethod','Floor','OverflowAction','Wrap','ProductMode','FullPr
ecision','SumMode','FullPrecision') (default) | fimath object

fimath properties to use for conversion, specified as a fimath object.

FixPtFileNameSuffix — Suffix for fixed-point file names
'_fixpt' (default) | string

Suffix for fixed-point file names, specified as a string.

LaunchNumericTypesReport — View the numeric types report after the software has proposed fixed-point types
true (default) | false

View the numeric types report after the software has proposed fixed-point types, specified as true or false.

LogIOForComparisonPlotting — Enable simulation data logging to plot the data differences introduced by fixed-point conversion
true (default) | false

Enable simulation data logging to plot the data differences introduced by fixed-point conversion, specified as true or false.

OptimizeWholeNumber — Optimize the word lengths of variables that are always whole numbers
true (default) | false

Optimize the word lengths of variables whose simulation min/max logs indicate that they are always whole numbers, specified as true or false.

PlotFunction — Name of function to use for comparison plots
'' (default) | string

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.

PlotWithSimulationDataInspector — Use the Simulation Data Inspector for comparison plots
false (default) | true

Use the Simulation Data Inspector for comparison plots, specified as true or false.

LogIOForComparisonPlotting must be set to true to enable comparison plotting. The PlotFunction option takes precedence over PlotWithSimulationDataInspector.

ProposeFractionLengthsForDefaultWordLength — Propose fixed-point types based on DefaultWordLength
true (default) | false

Propose fixed-point types based on DefaultWordLength, specified as true or false.

ProposeTargetContainerTypes — Whether to propose target container types
false (default) | true

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.

ProposeWordLengthsForDefaultFractionLength — Propose fixed-point types based on DefaultFractionLength
false (default) | true

Propose fixed-point types based on DefaultFractionLength, specified as true or false.

ProposeTypesUsing — Propose data types based on simulation range data, derived ranges, or both
'BothSimulationAndDerivedRanges' (default) | 'SimulationRanges' | 'DerivedRanges'

Propose data types based on simulation range data, derived ranges, or both, specified as 'BothSimulationAndDerivedRanges', 'SimulationRanges', or 'DerivedRanges'.

SafetyMargin — Safety margin percentage by which to increase the simulation range when proposing fixed-point types
0 (default) | real number greater than -100

Safety margin percentage by which to increase the simulation range when proposing fixed-point types, specified as a real number greater than -100.

Data Types: double

StaticAnalysisQuickMode — Perform faster static analysis
false (default) | true

Perform faster static analysis, specified as true or false.

StaticAnalysisTimeoutMinutes — Abort analysis if timeout is reached
'' (default) | positive integer

Abort analysis if timeout is reached, specified as a positive integer.

TestBenchName — Test bench function name or names
'' (default) | string | cell array of strings

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.

Data Types: string | cell

TestNumerics — Enable numerics testing
false (default) | true

Enable numerics testing, specified as true or false.

Examples

Generate HDL Code from Floating-Point MATLAB Code

Create a coder.FixPtConfig object, fixptcfg, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is mlhdlc_dti_tb.

```
fixptcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Create a coder.HdlConfig object, hdlcfg, with default settings.

```
hdlcfg = coder.config('hdl');
```

Convert your floating-point MATLAB design to fixed-point, and generate HDL code. In this example, the MATLAB design function name is mlhdlc_dti.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_dti
```

Alternatives

You can also generate HDL code from MATLAB code using the **HDL Workflow Advisor**. For more information, see "Basic HDL Code Generation and FPGA Synthesis from MATLAB".

Version History
Introduced in R2014b

See Also
coder.HdlConfig | coder.config | codegen

Topics
"Generate HDL Code from MATLAB Code Using the Command Line Interface"

coder.HdlConfig

HDL `codegen` configuration object

Description

A `coder.HdlConfig` object contains the configuration parameters that the HDL `codegen` function requires to generate HDL code. To pass this object to the `codegen` function, use the `-config` option.

Creation

Syntax

```
hdlcfg = coder.config("hdl")
```

Description

`hdlcfg = coder.config("hdl")` creates a `coder.HdlConfig` object for HDL code generation.

Properties

General

HDLCodingStandard — HDL coding standard to follow
'None' (default) | 'Industry'

HDL coding standard to follow when generating code, specified as a character vector or string array. Generates a compliance report showing errors, warnings, and messages.

Example: 'Industry' "Industry"

Data Types: char | string

HDLCodingStandardCustomizations — HDL coding standard rules and report customizations
none (default) | HDL coding standard customization object

HDL coding standard rules and report customizations, specified by using HDL Coding Standard Customization Properties. For more information, see HDL Coding Standard Customization Properties. If you want to customize the coding standard rules and report, you must set `HDLCodingStandard` to "Industry".

TargetLanguage — Target language of the generated code
'VHDL' (default) | 'Verilog' | 'SystemC'

Target language of the generated code, specified as a character vector or string array.

Example: 'Verilog' "Verilog"

Data Types: char | string

TestBenchName — Test bench function name
'' (default) | *'Testbench name'* | *"Testbench name"*

Test bench function name, specified as a character vector or string array. You must specify a test bench.

Data Types: char | string

Workflow

GenerateCosimTestBench — Option to generate a cosimulation test bench
false or 0 (default) | true or 1

Option to generate a cosimulation test bench, specified as a numeric or logical 1 (true) or 0 (false).

Data Types: logical

GenerateFILTestBench — Option to generate FIL test bench
false or 0 (default) | true or 1

Option to generate a FIL test bench, specified as a numeric or logical 1 (true) or 0 (false).

Data Types: logical

GenerateHDLTestBench — Option to generate an HDL test bench
false or 0 (default) | true or 1

Option to generate an HDL test bench, specified as a numeric or logical 1 (true) or 0 (false).

Data Types: logical

SimulateCosimTestBench — Option to simulate generated cosimulation test bench
false or 0 (default) | true or 1

Option to simulate a generated cosimulation test bench, specified as a numeric or logical 1 (true) or 0 (false). This option is ignored if GenerateCosimTestBench is false.

Data Types: logical

SimulateFILTestBench — Option to simulate generated cosimulation test bench
false or 0 (default) | true or 1

Option to simulate a generated cosimulation test bench, specified as a numeric or logical 1 (true) or 0 (false). This option is ignored if GenerateCosimTestBench is false.

Data Types: logical

SimulateGeneratedCode — Option to simulate generated code
false or 0 (default) | true or 1

Option to simulate generated code, specified as a numeric or logical 1 (true) or 0 (false).

Data Types: logical

SynthesizeGeneratedCode — Option to synthesize generated code
false or 0 (default) | true or 1

Option to synthesize generated code, specified as a numeric or logical 1 (true) or 0 (false).

Data Types: `logical`

SystemCTestBenchStimulus — SystemC test bench function name
`'HDL Test bench stimulus'` (default) | `'Test bench with random input stimulus'`

SystemC test bench function name, specified as a character vector or string array. Specify the test bench for SystemC code generation.

Example: `'Test bench with random input stimulus'`

Data Types: `char` | `string`

Target Tool Selection

CosimClockEnableDelay — Time delay before clock enable
`0` (default)

Time (in clock cycles) between deassertion of reset and assertion of clock enable.

Data Types: `int32`

CosimClockHighTime — Time that clock is high
`5` (default)

The number of nanoseconds the clock is high.

Data Types: `int32`

CosimClockLowTime — Time that clock is low
`5` (default)

The number of nanoseconds the clock is low.

Data Types: `int32`

CosimHoldTime — Hold time for input signals and forced reset signals
`2` (default)

The hold time for input signals and forced reset signals, specified in nanoseconds.

Data Types: `int32`

CosimLogOutputs — Option to log and plot outputs of the reference design function and HDL simulator
`false` or `0` (default) | `true` or `1`

Option to log and plot outputs of the reference design function and HDL simulator, specified as a numeric or logical `1` (true) or `0` (false).

Data Types: `logical`

CosimResetLength — Cosimulation reset length
`2` (default)

Specify the time (in clock cycles) between assertion and deassertion of reset.

Data Types: `int32`

CosimRunMode — HDL simulator run mode during simulation
`'Batch'` (default) | `'GUI'`

HDL simulator run mode during simulation, specified as a character vector or string array. When in `Batch` mode, you do not see the HDL simulator UI. The HDL simulator shuts down after simulation.

Example: `'GUI' "GUI"`

Data Types: `char` | `string`

CosimTool — HDL Simulator
`'ModelSim'` (default) | `'Incisive'` | `'Vivado Simulator'`

HDL simulator for the generated cosim test bench, specified as a character vector or string array.

Example: `'Incisive'"Incisive"`

Data Types: `char` | `string`

FILAdditionalFiles — List of additional source files to include
`''` (default) | `'Additional source files'`

List of additional source files to include, specified as a character vector or string array. Separate file names by using a semicolon (;).

Data Types: `char` | `string`

FILBoardIPAddress — IP address of the FPGA board
`'192.168.0.2'` (default)

IP address of the FPGA board, specified as a character vector or string array. You must enter a valid IP address.

Data Types: `char` | `string`

FILBoardMACAddress — MAC address of the FPGA board
`'00-0A-35-02-21-8A'` (default)

MAC address of the FPGA board, specified as a character vector or string array. You must enter a valid MAC address.

Data Types: `char` | `string`

FILBoardName — FPGA board name
`'Choose a board'` (default) | `'A board name'`

FPGA board name, specified as a character vector or string array. You must override the default value and specify a valid board name.

Data Types: `char` | `string`

FILLogOutputs — Option to log and plot outputs of the reference design function and FPGA
`false` or `0` (default) | `true` or `1`

Option to log and plot outputs of the reference design function and FPGA, specified as a numeric or logical `1` (true) or `0` (false).

Data Types: `logical`

SimulationTool — Simulation tool name
`'ModelSim'` (default) | `'ISIM'`

Simulation tool name, specified as a character vector or string array.

Example: `'ISIM' "ISIM"`

Data Types: `char | string`

SynthesisTool — Synthesis tool name
`''` (default) | `'Xilinx Vivado'` | `'Altera Quartus II'` | `'Microsemi Libero SoC'` | `'Intel Quartus Pro'` | `'Xilinx ISE'` | `'Cadence Stratus'`

Synthesis tool name, specified as a character vector or string array.

Example: `'Xilinx Vivado' "Xilinx Vivado"`

Data Types: `char | string`

SynthesisToolChipFamily — Synthesis target chip family name
`''` (default) | `'Family name'`

Synthesis target chip family name, specified as a character vector or string array.

Data Types: `char | string`

SynthesisToolDeviceName — Synthesis target device name
`''` (default) | `'Device name'`

Synthesis target device name, specified as a character vector or string array.

Data Types: `char | string`

SynthesisToolPackageName — Synthesis target package name
`''` (default) | `'Package name'`

Synthesis target package name, specified as a character vector or string array.

Data Types: `char | string`

SynthesisToolSpeedValue — Synthesis target speed
`''` (default) | `'Speed value'`

Synthesis target speed, specified as a character vector or string array.

Data Types: `char | string`

TargetFrequency — Target clock frequency
0 (default)

Specify the target frequency, in MHz, of the clock wired to the clock input of the generated HDL design. This frequency is the same as the output clock frequency of the clock module. Adaptive pipelining takes into account the target frequency that you set to improve the frequency of your design.

Data Types: `double`

Code Style

InitializeBlockRAM — Option to initialize block RAM
`true` or 1 (default) | `false` or 0

Option to initialize block RAM to `0` for simulation, specified as a numeric or logical `1` (true) or `0` (false).

Data Types: `logical`

InlineConfigurations — Option to include inline configurations in generated VHDL code
`true` or `1` (default) | `false` or `0`

Option to include inline configurations in generated VHDL code, specified as a numeric or logical `1` (true) or `0` (false).

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.

Data Types: `logical`

TimingControllerPostfix — Timing controller postfix
`'_tc'` (default) | `'Postfix'`

Postfix to append to the design name to form the name of the timing controller, specified as a character vector or string array.

Data Types: `char` | `string`

VHDLLibraryName — Target library name for generated VHDL code
`'work'` (default) | `'Library name'`

Target library name for generated VHDL code, specified as a character vector or string array.

Data Types: `char` | `string`

Clocks

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

Active clock edge, specified as a character vector or string array.

Example: `'Rising'` `"Rising"`

Data Types: `char` | `string`

Oversampling — Oversampling factor
`1` (default) | integer greater than or equal to 1

Oversampling factor, specified as an integer greater than or equal to 1. Specify the frequency of global oversampling clock as a multiple of the design base clock rate.

Data Types: `int32`

TimingControllerArch — Timing controller architecture
`'default'` (default) | `'resettable'`

Timing controller architecture, specified as one of these options:

TimingControllerArch Value	Description
`'default'`	Generate a timing controller without a reset. This setting generates a timing controller code file as its own HDL file and instantiates the timing controller in the DUT at the top level.
`'resettable'`	Generate a timing controller in the DUT with a reset. This setting generates a timing controller code file as its own HDL file and instantiates the timing controller in the DUT at the top level.

Example: `'resettable'`

Data Types: `char` | `string`

Ports

IOThreshold — Maximum number of I/O pins for FPGA deployment
5000 (default) | positive integer

Maximum number of I/O pins for FPGA deployment, specified as an integer. Specify the maximum number of I/O pins for your target FPGA. If the DUT pin count in the generated code exceeds the value of this parameter, HDL Coder generates the message specified by the `TreatIOThresholdAs` parameter in the HDL Conformance Report.

Data Types: `int32`

TreatIOThresholdAs — Message to return if DUT pin count exceeds I/O threshold
`'Error'` (default) | `'Warning'` | `'None'`

Message to return if DUT pin count exceeds I/O threshold, specified as a character vector or string array. Specify the type of message generated when the DUT pin count in the generated code exceeds the maximum number of I/O pins threshold set by the `IOThreshold` parameter.

Example: `'Warning' "Warning"`

Data Types: `char` | `string`

Optimizations

AdaptivePipelining — Option to insert adaptive pipeline registers in your design
`false` or `0` (default) | `true` or `1`

Option to insert adaptive pipeline registers in your design, specified as a numeric or logical `1` (true) or `0` (false). Enable adaptive pipelining to insert pipeline registers to the blocks in your design, reduce the area usage, and maximize the achievable clock frequency on the target FPGA device.

When you specify this parameter, specify the **Synthesis Tool** in the **Select Code Generation Target** task. If your design has multipliers, specify the **Synthesis Tool** and the **Target Frequency (MHz)** for adaptive pipeline insertion.

Data Types: `logical`

AdderSharingMinimumBitwidth — Minimum bit width for shared adders
0 (default) | integer greater than or equal to 2

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`.

Data Types: `int32`

`AggressiveDataflowConversion` — Option to use dataflow representation for MATLAB functions
`false` or `0` (default) | `true` or `1`

Option to use dataflow representation for MATLAB functions, specified as a numeric or logical `1` (true) or `0` (false). Transform the control flow algorithm of the MATLAB code inside a MATLAB function to a dataflow representation. Set this property to `true` for optimizations and options that use dataflow representation, such as frame-to-sample conversion and native floating-point optimizations.

If you set both `AggressiveDataflowConversion` and `GenerateMLFcnBlock` to `true`, you can generate a functionally equivalent Simulink model of your MATLAB function design that contains Simulink blocks, which is comparable to designing the algorithm with Simulink.

Data Types: `logical`

`BalanceClockRateOutputPorts` — Option to balance clock-rate pipelined DUT output ports
`false` or `0` (default) | `true` or `1`

Option to balance clock-rate pipelined DUT output ports, specified as a numeric or logical `1` (true) or `0` (false). Synchronize the DUT outputs while satisfying the highest-latency requirements of the outputs. Apply this option when interfacing your logic with a valid signal interface to align the output of the logic path and valid signal path.

Data Types: `logical`

`ClockRatePipelineOutputPorts` — Option to allow clock-rate pipelining of DUT output ports
`false` or `0` (default) | `true` or `1`

Option to allow clock-rate pipelining of DUT output ports, specified as a numeric or logical `1` (true) or `0` (false). Set this property to `true` to produce the DUT outputs as soon as possible by passing the outputs from the DUT at the clock rate rather than the data rate.

Data Types: `logical`

`ClockRatePipelining` — Option to insert pipeline registers at clock rate
`true` or `1` (default) | `false` or `0`

Option to insert pipeline registers at the clock rate, specified as a numeric or logical `1` (true) or `0` (false). Use clock-rate pipelining to insert pipeline registers at a clock rate that is faster than the data rate. This optimization improves the clock frequency and reduces the area usage without introducing additional latency. Clock-rate pipelining does not affect existing design delays in your design.

Data Types: `logical`

`DistributedPipelining` — Option to distribute pipeline registers
`false` or `0` (default) | `true` or `1`

Option to distribute pipeline registers, specified as a numeric or logical `1` (true) or `0` (false). When enabled, HDL Coder moves registers within your design to reduce the critical path.

Data Types: `logical`

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

Priority for distributed pipelining algorithm, specified as a character vector or string array.

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.

Example: `'NumericalIntegrity'` `"NumericalIntegrity"`

Data Types: `char` | `string`

InputPipeline — Number of input pipeline register stages
`0` (default) | integer greater than `0`

Specify the number of input pipeline register stages. When `DistributedPipelining` is enabled, these registers can be distributed through the design.

Data Types: `int32`

LoopOptimization — Loop optimization
`'LoopNone'` (default) | `'StreamLoops'` | `'UnrollLoops'`

Loop optimization in generated code, specified as a character vector or string array. See "Optimize MATLAB Loops".

LoopOptimization Value	Description
`LoopNone`	Do not optimize loops in generated code.
`StreamLoops`	Stream loops.
`UnrollLoops`	Unroll loops.

Example: `'StreamLoops'` `"StreamLoops"`

Data Types: `char` | `string`

MapPersistentVarsToRAM — Map persistent array variables to RAM
`true` or `1` (default) | `false` or `0`

Map persistent array variables to RAM, specified as a numeric or logical `1` (true) or `0` (false). Use this property to optimize area and save resources on your target device by mapping persistent arrays to block RAM.

To map these persistent arrays to block RAMs, the RAM size must be greater than or equal to the RAM mapping threshold. See RAMThreshold.

Data Types: `logical`

MapPipelineDelaysToRAM — Map pipeline registers in generated HDL code to RAM
`false` or `0` (default) | `true` or `1`

Map pipeline registers in generated HDL code to RAM, specified as a numeric or logical `1` (true) or `0` (false). Use this property to map the pipeline registers inserted by pipelining and resource sharing optimizations and block implementations to RAM. You can save area on your target device by mapping pipeline registers to RAM.

To map these registers to block RAMs, the RAM size must be greater than or equal to the RAM mapping threshold. See RAMThreshold.

Data Types: `logical`

MinimizeClockEnables — Option to omit generation of clock enable logic
`false` or `0` (default) | `true` or `1`

Option to omit generation of clock enable logic, specified as a numeric or logical `1` (true) or `0` (false).

When `false` (default), generate clock enable logic.

When `true`, omit generation of clock enable logic wherever possible.

Data Types: `logical`

MultiplierPartitioningThreshold — Maximum input bit width for hardware multipliers
none (default) | integer greater than or equal to 2

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 digital signal processor (DSP) or multiplier hardware on your target device.

Data Types: `int32`

MultiplierSharingMinimumBitwidth — Minimum bit width for shared multipliers
none (default) | integer greater than or equal to 2

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`.

Data Types: `int32`

OutputPipeline — Number of output pipeline register stages
0 (default) | integer greater than 0

Specify the number of output pipeline register stages. When `DistributedPipelining` is enabled, these registers can be distributed through the design.

Data Types: `int32`

PreserveDesignDelays — Preserve design delays
`false`or 0 (default) | `true` or 1

Prevent distributed pipelining from moving design delays or allow distributed pipelining to move design delays, specified as a numeric or logical 1 (true) or 0 (false).

Persistent variables and `dsp.Delay` System objects are design delays.

Data Types: `logical`

RAMThreshold — Minimum RAM size for mapping persistent array variables and pipeline delays to block RAM
`'256'` (default) | string array | character vector

Minimum RAM size for mapping persistent array variables and pipeline delays to block RAM, specified as a string array or character vector.

Specify the RAM mapping threshold by using either:

- A string array or character vector of a single positive integer to map persistent array variables to RAM if the RAM size is greater than this threshold. The unit is bits. To calculate the total RAM size for persistent arrays, use this formula:

  `RAMSize =  Array size * Word length * Complexity`

  `Complexity` is 2 for a complex data type or 1 for a real datatype. To calculate the total RAM size for delays, use this formula:

  `RAMSize = Delay length * Word length * Vector length * Complexity`

- A string array or character vector of format `MxN` that specifies two thresholds to define the shape of the data to map to RAM, where `M` is for array size (for persistent arrays) or delay length (for delays) and `N` is for word length or bit width of the data type.

Example: `"256"`

Example: `"500x50"`

Data Types: `string` | `char`

RegisterInputs — Option to insert pipeline register at each DUT input
`false` or 0 (default) | `true` or 1

Option to insert a pipeline register at each DUT input, specified as a numeric or logical 1 (true) or 0 (false). Distributed pipelining does not move these registers.

Data Types: `logical`

RegisterOutputs — Option to insert pipeline register at each DUT output
`false` or 0 (default) | `true` or 1

Option to insert a register at each DUT output, specified as a numeric or logical `1` (true) or `0` (false). Distributed pipelining does not move these registers.

Data Types: `logical`

ShareAdders — Option to share adders
`false` or `0` (default) | `true` or `1`

Option to share adders, specified as a numeric or logical `1` (true) or `0` (false). If `true`, share adders when `ResourceSharing` is greater than 1 and adder bit width is greater than or equal to `AdderSharingMinimumBitwidth`.

Data Types: `logical`

ShareMultipliers — Option to share multipliers
`true` or `1` (default) | `false` or `0`

Option to share multipliers, specified as a numeric or logical `1` (true) or `0` (false). If `true`, share multipliers when `ResourceSharing` is greater than 1 and multiplier bit width is greater than or equal to `MultiplierSharingMinimumBitwidth`.

Data Types: `logical`

Floating Point

FloatingPointLibrary — Floating point library name
`'None'` (default) | `'NativeFloatingPoint'`

Floating-point library name, specified as a character vector or string array.

Example: `'NativeFloatingPoint'`

Data Types: `char` | `string`

FloatingPointTargetConfiguration — Floating point target configuration
`[ ]` (default) | `hdlcoder.FloatingPointTargetConfig` object

Floating point target configuration when using native floating point for HDL code generation, specified as an `hdlcoder.FloatingPointTargetConfig` object. To configure the floating point target, see `hdlcoder.FloatingPointTargetConfig`. To generate HDL code from a MATLAB function, you can only set the native floating point properties `HandleDenormals`, `LatencyStategy`, and `MantissaMultiplyStrategy` in the `hdlcoder.FloatingPointTargetConfig` class.

Example: `hdlcfg.FloatingPointTargetConfiguration = hdlcoder.createFloatingPointTargetConfig('NATIVEFLOATINGPOINT')`, where `hdlcfg` is a `coder.HdlConfig` object. See `createFloatingPointTargetConfig`.

Data Types: `hdlcoder.FloatingPointTargetConfig`

Advanced Coding

GenerateMLFcnBlock — Option to generate Simulink model from MATLAB function
`false` or `0` (default) | `true` or `1`

Option to generate a Simulink model from your MATLAB function, specified as a numeric or logical `1` (true) or `0` (false). You must have a Simulink license.

If you set `AggressiveDataflowConversion` to `false`, this parameter generates a MATLAB Function block to use in a Simulink model.

If you set `AggressiveDataflowConversion` to `true`, this parameter generates a functionally equivalent Simulink model of your MATLAB function design that contains Simulink blocks that perform the algorithm designed in your MATLAB function. This model is similar to a generated model in the Simulink-to-HDL workflow.

Data Types: `logical`

InstantiateFunctions — Option to generate instantiable HDL code modules
`false` or `0` (default) | `true` or `1`

Option to generate instantiable HDL code modules from functions, specified as a numeric or logical `1` (true) or `0` (false).

Data Types: `logical`

TreatRealsInGeneratedCodeAs — Check for reals in generated HDL code
`'Warning'` (default) | `'None'` | `'Error'`

Check for reals in the generated HDL code, specified as a character vector or string array.

Value	Description
None	Do not check for reals in the generated HDL code.
Warning	Checks and warns of presence of real data types in the generated HDL code. Real data types in the generated HDL code are not synthesizable on target FPGA devices.
Error	Checks and generates an error if the generated HDL code uses real data types. If you are generating code for simulation purposes and not for synthesizing your design, you can change this setting to `Warning` or `None`. To generate synthesizable HDL code, set the **Floating Point IP Library** to `Native Floating Point`.

Example: `'None'` `"None"`

Data Types: `char` | `string`

Test Bench

SimulationIterationLimit — Simulation iteration limit
`-1` (default) | positive integer

Maximum number of simulation iterations during test bench generation. This property affects only test bench generation, not simulation during fixed-point conversion. When the value is `-1` (default), no maximum number of simulation iterations is set.

Data Types: `int32`

UseFileIOInTestBench — Option to use data files for test bench input and output data
`true` or `1` (default) | `false` or `0`

Option to create and use data files for reading and writing test bench input and output data, specified as a numeric or logical `1` (true) or `0` (false).

Data Types: `logical`

Lint Script

`HDLLintCmd` — HDL lint script command
`''` (default)

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 string array or character vector by using this format:

`custom_lint_tool_command -option1 -option2 %s`

Data Types: `char` | `string`

`HDLLintInit` — HDL lint script initialization name
`''` (default)

HDL lint script initialization name, specified as a character vector or string array.

Data Types: `char` | `string`

`HDLLintTerm` — HDL lint script termination name
`''` (default)

HDL lint script termination name, specified as a character vector or string array.

Data Types: `char` | `string`

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

HDL lint tool script to generate, specified as a character vector or string array.

Example: `'SpyGlass' "SpyGlass"`

Data Types: `char` | `string`

Frame to Sample Conversion

`FrameToSampleConversion` — Option to enable frame to sample conversion
`false` or `0` (default) | `true` or `1`

Option to enable frame-to-sample conversion, specified as a numeric or logical `1` (true) or `0` (false). If your MATLAB function is not synthesizable because it requires a large amount of I/O, this optimization can reduce the I/O in the design and generate synthesizable HDL code. The frame-to-sample conversion converts matrix inputs to smaller samples by streaming the input signal for HDL code generation to reduce the I/O that handles a large input signal. For more information, see "HDL Code Generation from Frame-Based Algorithms".

To enable this property, set `AggressiveDataflowConversion` to `true`.

Data Types: `logical`

`DelaySizeThreshold` — Delay size threshold for external memory
`100000000` (default) | positive integer

Delay size threshold for external memory, specified as a positive integer. Specify a threshold size in bits to map large integer delays to input and output DUT ports. Use this property to offload large

delays to external memory outside of your FPGA. If you map large delays to external memory, you cannot generate a test bench.

To enable this property, set `FrameToSampleConversion` to `true`.

Data Types: `int32`

InputFIFOSize — Register size of generated input FIFOs
10 (default) | integer greater than or equal to 4

Register size of generated input FIFOs, specified as an integer greater than or equal to 4. Use this parameter to specify the register size of the generated input FIFOs around the streaming matrix partitions. The frame-to-sample conversion generates an input FIFO for every input to the MATLAB function that has a sampled-based input signal.

To enable this property, set `FrameToSampleConversion` to `true`.

Data Types: `int32`

InputProcessingOrder — Order to process incoming frame data
`'RowMajor'` (default) | `'ColumnMajor'`

Order to process incoming frame data, specified as `'RowMajor'` or `'ColumnMajor'`. Choose between row-major and column-major ordering for the frame inputs that the frame-to-sample conversion optimization converts to sample inputs. This setting affects how the data is streamed into the device under test (DUT), but does not change the behavior of the generated DUT.

Set this property to `'RowMajor'` to traverse the input frame data for the frame-to-sample conversion using row-major ordering, which traverses the data from left to right and then top to bottom across the frame-based input matrix.

Set this property to `'ColumnMajor'` to traverse the input frame data for the frame-to-sample conversion using column-major ordering, which traverses the data from top to bottom and then left to right across the frame-based input matrix.

To enable this property, set `FrameToSampleConversion` to `true`.

Data Types: `char` | `string`

OutputFIFOSize — Register size of generated output FIFOs
10 (default) | integer greater than or equal to 4

Register size of the generated output FIFOs, specified as an integer greater than or equal to 4. Use this parameter to specify the register size of the generated output FIFOs around the streaming matrix partitions. The frame-to-sample conversion generates an output FIFO for every output of the MATLAB function that has a sampled-based output signal.

To enable this property, set `FrameToSampleConversion` to `true`.

Data Types: `int32`

SamplesPerCycle — Samples per cycle
1 (default) | integer greater than or equal to 1

Samples per cycle, specified as an integer greater than or equal to 1. Use this parameter to specify the size of the signals after the frame-to-sample conversion streams them. The streamed input signal is either a scalar (one sample per cycle) or 1-D vectors with N elements (N samples per cycle).

To enable this property, set `FrameToSampleConversion` to `true`.

Data Types: `int32`

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 that has default settings and provide a test bench name.

```
fixptcfg = coder.config("fixpt");
fixptcfg.TestBenchName = "mlhdlc_sfir_tb";
```

Create a `coder.HdlConfig` object that has 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 an FPGA board name.

```
hdlcfg.GenerateCosimTestBench = true;
hdlcfg.FILBoardName = "Xilinx Virtex-5 XUPV5-LX110T development board";
hdlcfg.GenerateFILTestBench = true;
```

Perform code generation, Cosim test bench generation, and FIL test bench generation.

```
codegen -float2fixed fixptcfg -config hdlcfg mlhdlc_sfir
```

Alternatives

You can also generate HDL code from MATLAB code using the HDL Workflow Advisor. For more information, see "Basic HDL Code Generation and FPGA Synthesis from MATLAB".

Version History
Introduced in R2014b

See Also

Functions
coder.config | codegen | hdlcoder.CodingStandard

Classes
coder.FixPtConfig

Properties
HDL Coding Standard Customization

Topics
"Generate HDL Code from MATLAB Code Using the Command Line Interface"

Shared Classes and Functions 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

Create a coding standard customization object *cso*.

```
cso = hdlcoder.CodingStandard('Industry');
```

Customize the Coding Standard Options

- 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 HDL Codegen Configuration and Coding Standard Customization Object

```
hdlcfg = coder.config('hdl');
```

Specify the coding standard and coding standard customization object.

```
hdlcfg.HDLCodingStandard = 'Industry';
hdlcfg.HDLCodingStandardCustomizations = cso;
```

Create a Temporary Folder and Copy the MATLAB Files

In this case, the design function is `mlhdlc_dti.m` and the test bench function is `mlhdlc_dti_tb.m`.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', ...
                                  'hdlcoderdemos', 'matlabhdlcoderdemos');
```

```
mlhdlc_temp_dir = [tempdir 'mlhdlc_sfir'];

cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_dti.m'), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, 'mlhdlc_dti_tb.m'), mlhdlc_temp_dir);
```

Generate HDL Code and Test Bench

Specify your test bench function name.

```
hdlcfg.TestBenchName = 'mlhdlc_dti_tb';
```

Generate HDL code for the design and check the code according to the customized HDL coding standard rules.

```
codegen -config hdlcfg mlhdlc_dti
```

```
### Begin VHDL Code Generation
### Working on mlhdlc_dti as <a href="matlab:edit('C:\TEMP\Bdoc23a_2213998_7084\ib967011\3\mlhdl
### Generating Resource Utilization Report <a href="matlab:web('C:\TEMP\Bdoc23a_2213998_7084\ib96
### Industry Compliance report with 0 errors, 1 warnings, 4 messages.
### Generating Industry Compliance Report <a href="matlab:web('C:\TEMP\Bdoc23a_2213998_7084\ib967
### Generating HDL Conformance Report <a href="matlab:web('C:\TEMP\Bdoc23a_2213998_7084\ib967011\
### HDL Conformance check complete with 0 errors, 26 warnings, and 0 messages.
Code generation successful.
```

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/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:\coding_standard\hdlsrc\sfir_fixed\symmetric_fir.vhd
### Industry Compliance report with 4 errors, 77 warnings, 6 messages.
### Generating Industry Compliance Report <a href="matlab:web('C:\coding_standard\hdlsrc\sfir_fi
### Creating HDL Code Generation Check Report file://C:\coding_standard\hdlsrc\sfir_fixed\symmetr
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

Input Arguments

`standardName` — HDL coding standard name
`'Industry'`

Specify the HDL coding standard to customize. The `standardName` value must match the `HDLCodingStandard` property value.

Example: `'Industry'`

Output Arguments

`cso` — HDL coding standard customizations
HDL coding standard customization object

HDL coding standard customizations, returned as an HDL coding standard customization object.

Version History
Introduced in R2014b

See Also

Properties
HDL Coding Standard Customization

Topics
"Generate HDL Coding Standard Report from Simulink"
"Generate an HDL Coding Standard Report from MATLAB"
"Choose Coding Standard and Report Option Parameters"
"HDL Coding Standard Report"

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.

To refer to a particular object and property, use dot notation:

```
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. If following a coding standard rule causes 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 this 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 this 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 this 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`.
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. 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 this 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 this 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 this 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 this 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) that 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 that are described separately within a process block (VHDL) or an always block (`Verilog`). The default is 1.

CascadedConditionalAssignmentCheck — Check if there are assignments to the same variable in multiple cascaded control regions
structure

Check if there are assignments to the same variable in multiple cascaded control regions within the same process block. This check corresponds to CGSL-2.F.B.1.a of the industry standard guidelines, specified as a structure with this field.

Field	Description
enable	Set to `true` to check for assignments to the same variable in multiple cascaded control regions.
	Set to `false` if you do not want to check for assignments to the same variable in multiple cascaded control regions.
	The default is `false`.

`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 to check 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`.

Version History
Introduced in R2014b

See Also
`hdlcoder.CodingStandard`

Topics
"Generate an HDL Coding Standard Report from MATLAB"
"Generate HDL Coding Standard Report from Simulink"
"HDL Coding Standard Report"
"Basic Coding Practices"
"RTL Description Rules and Checks"
"RTL Design Methodology Guidelines"

hdl.BlackBox

Package: hdl

Black box for including custom HDL code

Description

`hdl.BlackBox` provides a way to include custom HDL code, such as legacy or handwritten HDL code, in a MATLAB design intended for HDL code generation.

When you create a user-defined System object that inherits from `hdl.BlackBox`, you specify a port interface and simulation behavior that matches your custom HDL code.

HDL Coder simulates the design in MATLAB using the behavior you define in the System object. During code generation, instead of generating code for the simulation behavior, the coder instantiates a module with the port interface you specify in the System object.

To use the generated HDL code in a larger system, you include the custom HDL source files with the rest of the generated code.

To include custom HDL code:

1   Create the `hdl.BlackBox` 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?

Creation

Syntax

`hdl.BlackBox`

Description

Create a System object that inherits from `hdl.BlackBox` to create a black box for HDL code generation. See "Integrate Custom HDL Code Into MATLAB Design".

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.

> **Note** You cannot specify clock, reset, and clock enable signals explicitly in your Simulink model by using the **AddClockEnablePort**, **AddClockPort**, and **AddResetPort** properties. Instead, use these properties to add a clock, reset, or clock enable port in the generated HDL code.

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.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects
step   Run System object algorithm

release   Release resources and allow changes to System object property values and input characteristics
reset   Reset internal states of System object

Version History
Introduced in R2015a

Extended Capabilities

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`coder.HdlConfig`

Topics
"Integrate Custom HDL Code Into MATLAB Design"
"Generate Board-Independent IP Core from MATLAB Algorithm"
"Generate Black Box Interface for Subsystem"

hdl.Delay

Package: hdl

Delay input signal by number of samples

Description

The `hdl.Delay` System object delays the input to the System object by a specified number of samples. You can delay the input along each column for frame-based processing or delay the input for each element for sample-based processing. To select the processing mode, use the `InputProcessing` property. You can specify the initial output of the object through the `InitialConditions` property.

To delay the input of the `hdl.Delay` System object:

1   Create the `hdl.Delay` 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?

Creation

Syntax

```
delay = hdl.Delay
delay = hdl.Delay(Name,Value)
delay = hdl.Delay(len,Name,Value)
```

Description

`delay = hdl.Delay` creates a delay System object that delays the input by 1 sample.

`delay = hdl.Delay(Name,Value)` sets properties using one or more name-value pairs. For example, `delay = hdl.Delay('InitialConditions',1);` creates a delay object that has an initial output of one.

`delay = hdl.Delay(len,Name,Value)` creates a delay System object with the `Length` property set to `len`, and sets properties using one or more name-value pairs. For example, `delay = hdl.Delay(10,'InitialConditions',1,'InputProcessing','SampleBasedProcessing')` `;` creates a delay object that has an initial output of one and uses sampled-based processing.

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.

Length — Number of samples by which to delay input
1 (default) | scalar positive integer

Number of samples by which to delay the input signal, specified as a scalar positive integer.

InitialConditions — Initial output of object
0 (default) | scalar

Initial output of the System object, specified as a scalar.

InputProcessing — Input processing mode for object
"FrameBasedProcessing" (default) | "SampleBasedProcessing"

Input processing mode for the System object, specified as "FrameBasedProcessing" or "SampleBasedProcessing". For more information on frame- and sample-based processing, see "Sample- and Frame-Based Concepts" (DSP System Toolbox).

Usage

Syntax

delayOut = delay(dataInput)

Description

delayOut = delay(dataInput) adds delay to the data input and returns the delayed output.

Input Arguments

dataInput — Input signal to delay
scalar | vector | matrix

Input signal to delay, specified as a scalar, vector, or matrix.

Example: [1;2;3;4;5]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

Output Arguments

delayOut — Delayed output
scalar | vector | matrix

Delayed output, returned as a scalar, vector, or matrix. The size and data type of the output match the size and data type of the data input.

Example: [0;0;1;2;3]

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step   Run System object algorithm
release   Release resources and allow changes to System object property values and input
          characteristics
reset   Reset internal states of System object

Examples

Use Frame-Based Processing to Delay Input

Delay input by five samples by using the `hdl.Delay` System object. By default, `hdl.Delay` uses frame-based input processing and initial outputs of 0.

```
delay = hdl.Delay(5);
input = [(1:10)' (11:20)'];
step1 = delay(input)

step1 =

     0     0
     0     0
     0     0
     0     0
     0     0
     1    11
     2    12
     3    13
     4    14
     5    15
```

Use the object function `reset` to reset the delay states.

```
reset(delay);
step1 = delay(input)
step2 = delay(input)

step1 =

     0     0
     0     0
     0     0
     0     0
     0     0
     1    11
     2    12
     3    13
     4    14
     5    15
```

```
step2 =

      6    16
      7    17
      8    18
      9    19
     10    20
      1    11
      2    12
      3    13
      4    14
      5    15
```

Use Sample-Based Processing to Delay Input

Delay input by five samples by using the `hdl.Delay` System object with sample-based input processing. By default, `hdl.Delay` uses initial outputs of 0.

With sample-based processing, each element in the input is a channel and is delayed by five samples. As a result, the first five steps each output 0, which is the default initial condition. Step six generates the first non-zero output.

```
delay = hdl.Delay(5, 'InputProcessing', 'SampleBasedProcessing');
input = [(1:10)' (11:20)'];
step1 = delay(input)
step2 = delay(input)
step3 = delay(input)
step4 = delay(input)
step5 = delay(input)
step6 = delay(input)

step1 =

      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0


step2 =

      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
      0     0
```

```
         0     0
         0     0


step3 =

         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0


step4 =

         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0


step5 =

         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0
         0     0


step6 =

         1    11
         2    12
         3    13
         4    14
         5    15
         6    16
         7    17
         8    18
```

```
     9    19
    10    20
```

Version History
Introduced in R2021a

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 systems using Fixed-Point Designer™.

See Also

Objects
`dsp.Delay` | `hdl.RAM` | `hdl.BlackBox`

Blocks
Delay

Topics
"HDL Code Generation from System Objects"

hdl.RAM

Package: hdl

Single, simple dual, or dual-port RAM for memory read/write access

Description

`hdl.RAM` reads from and writes to memory locations for a single, simple dual, or dual-port RAM. The output data is delayed one step. If your input data is scalar, the address and write enable inputs must be scalar, and HDL Coder infers a single RAM block. If your data is a vector, HDL Coder infers an array of parallel RAM banks. With vector data input, the address and write enable inputs can be both scalars or vectors. When you specify scalar inputs for the write enable and address ports, the system object applies the same operation to each RAM bank.

The `hdl.RAM` System object can have 2^{31} bytes of internal storage. The RAM size takes into account the address width, the number of bytes that are used to store each word, and the number of RAM banks.

To read from or write to memory locations in the RAM:

1 Create the `hdl.RAM` object and set its properties.
2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

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.

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, half, integer, or a fixed-point (fi) object, and can be real or complex. The hdl.RAM
block uses fixed-point data type internally for address calculations. This results in a check out of the
Fixed-Point Designer product license.

Data Types: single | double | half | 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, must be unsigned, and must be between 2 and 31 bits long. Specify this address when you
create a single port RAM object.

Note Even if the input data is a built-in integer data type, the hdl.RAM block uses fixed-point data
type internally for address calculations when RAM banks are used. This results in a check out of the
Fixed-Point Designer product license.

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, must be unsigned, and
must be between 2 and 31 bits long.

Note Even if the input data is a built-in integer data type, the hdl.RAM block uses fixed-point data
type internally for address calculations when RAM banks are used. This results in a check out of the
Fixed-Point Designer product license.

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`, must be unsigned, and must be between 2 and 31 bits long.

> **Note** Even if the input data is a built-in `integer` data type, the hdl.RAM block uses fixed-point data type internally for address calculations when RAM banks are used. This results in a check out of the Fixed-Point Designer product license.

Data Types: `uint8` | `uint16` | `fi`

Output Arguments

dataOut — Output data
Scalar (default) | Vector

Output data that the System object reads from the memory location `rwAddress` a single port RAM object when `wrEn` is false.

rdDataOut — Data from Read address
Scalar (default) | Vector

Old output data that the System object reads from the memory location `rdAddress` of a simple dual port RAM or dual port RAM System object.

wrDataOut — Data from Write address
Scalar (default) | Vector

New or old output data that the System object reads from the memory location `wrAddress` of a simple dual port RAM or dual port RAM System object.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step   Run System object algorithm
release   Release resources and allow changes to System object property values and input characteristics
reset   Reset internal states of System object

Examples

Observe Previous Data at Write Time

Construct System object to read from or write to a memory location in RAM. Set `WriteOutputValue` to `Old data` to return the previous value stored at the write address.

The output data port corresponds to the read/write address passed in. During a write operation, the old data at the write address is sent out as the output.

```
ram_1p = hdl.RAM('RAMType','Single port',...
                        'WriteOutputValue','Old data')

ram_1p =
  hdl.RAM with properties:

            RAMType: 'Single port'
    WriteOutputValue: 'Old data'
     RAMInitialValue: 0
```

```
dataLength   = 10;
dataIn = 1:10;
dataOut = zeros(1,dataLength);
```

Write a count pattern to the memory. Previous values on the first writes are all zero.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
```

```
dataOut = 1×10

     0     0     0     0     0     0     0     0     0     0
```

Read the data back.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = false;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut
```

```
dataOut = 1×10

     0     1     2     3     4     5     6     7     8     9
```

Now, write the count in reverse order. The previous values are the original count.

```
for ii = 1:dataLength
  addressIn   = uint8(ii-1);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(dataLength-ii+1),addressIn,writeEnable);
```

```
end
dataOut

dataOut = 1×10

   10    1    2    3    4    5    6    7    8    9
```

Read/Write Single-Port RAM

Create System object that writes to a single port RAM and reads the newly written value.

Construct single-port RAM System object. When you write a location, the object returns the new value. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

```
ram_1p = hdl.RAM('RAMType','Single port','WriteOutputValue','New data');
dataLength       = 16;
[dataIn,dataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and then read data back out again.

```
for ii = 1:dataLength
  dataIn(ii)  = randi([0 63],1,1,'uint8');
  addressIn   = fi((ii-1),0,4,0);
  writeEnable = true;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut

dataOut = 1x16 uint8 row vector

   0   52   57    8   58   40    6   17   35   61   61   10   62   61   31   51
```

```
for ii = 1:dataLength
  addressIn   = fi((ii-1),0,4,0);
  writeEnable = false;
  dataOut(ii) = ram_1p(dataIn(ii),addressIn,writeEnable);
end
dataOut

dataOut = 1x16 uint8 row vector

   9   52   57    8   58   40    6   17   35   61   61   10   62   61   31   51
```

Create Simple Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

The output data port corresponds to the read address. If a read operation is performed at the same address as the write operation, old data at that address is read out as the output. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

```
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;
  dataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);
end
dataOut
```

```
dataOut = 1x16 uint8 row vector

   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Write and read from different addresses. The object returns the read result after one cycle delay.

```
for ii = 1:dataLength
  wrAddr   = fi((ii-1),0,4,0);
  rdAddr   = fi(dataLength-ii+1,0,4,0);
  writeEnable = true;
  dataOut(ii) = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
dataOut
```

```
dataOut = 1x16 uint8 row vector

   0    9    9   51   31   61   62   10   61   61   35   17    6   40   58    8
```

Create Dual-Port RAM System Object

Construct System object to read from and write to different memory locations in RAM.

There are two output ports: a write output data port and a read output data port. The write output data port sends out the new data at the write address. The read output data port sends out the old data at the read address. The size of the RAM is inferred from the bitwidth of the address and write data on the first call to the object.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
dataLength       = 16;
[dataIn,wrDataOut,rdDataOut] = deal(uint8(zeros(1,dataLength)));
```

Write randomly generated data to the System object, and read the old data from the same address.

```
for ii = 1:dataLength
  dataIn(ii)  = randi([0 63],1,1,'uint8');
  wrAddr  = fi((ii-1),0,4,0);
```

```
  writeEnable = true;
  [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,wrAddr);
end
wrDataOut
```

wrDataOut = 1x16 uint8 row vector

```
    0    52    57     8    58    40     6    17    35    61    61    10    62    61    31    51
```

```
rdDataOut
```

rdDataOut = 1x16 uint8 row vector

```
    0     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
```

Write and read from different addresses. The object returns the read result after one cycle delay.

```
for ii = 1:dataLength
  wrAddr   = fi((ii-1),0,4,0);
  rdAddr   = fi(dataLength-ii+1,0,4,0);
  writeEnable = true;
  [wrDataOut(ii),rdDataOut(ii)] = ram_2p(dataIn(ii),wrAddr,writeEnable,rdAddr);
end
wrDataOut
```

wrDataOut = 1x16 uint8 row vector

```
    9    52    57     8    58    40     6    17    35    61    61    10    62    61    31    51
```

```
rdDataOut
```

rdDataOut = 1x16 uint8 row vector

```
    0     9     9    51    31    61    62    10    61    61    35    17     6    40    58     8
```

Create Dual-Port RAM with Multiple Banks

Create a System object that can write vector data to a dual-port RAM and read vector data out. Each element of the vector corresponds to a separate bank of RAM. This example creates 4 16-bit banks. Each bank has eight entries.

Construct dual-port RAM System object.

```
ram_2p = hdl.RAM('RAMType','Dual port','WriteOutputValue','New data');
```

Create vector write data and addresses. Use a 3-bit address (for 8 locations), and write 16-bit data. Read and write addresses are independent. Allocate memory for the output data.

```
ramDataIn = fi(randi((2^16)-1,1,4),0,16,0);
ramReadAddr = fi([1,1,1,1],0,3,0);
ramWriteAddr = fi([1,1,1,1],0,3,0);
[wrOut,rdOut] = deal(fi(zeros(1,4),0,16,0));
```

First, write locations in bank 1 and 4, then read all banks. The write data is echoed in the `wrOut` output argument. The object returns read results after one cycle delay.

```
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[true,false,false,true],ramReadAddr);
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr);
[wrOut,rdOut] = ram_2p(ramDataIn,ramWriteAddr,[false,false,false,false],ramReadAddr)

wrOut =
      53393            0            0        59859

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Unsigned
          WordLength: 16
      FractionLength: 0

rdOut =
      53393            0            0        59859

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Unsigned
          WordLength: 16
      FractionLength: 0
```

Algorithms

In your Simulink model, you can use the `hdl.RAM` inside a MATLAB System or a MATLAB Function block. If you log the output of a MATLAB System block, the output data has at least three dimensions because the MATLAB System block has at least two dimensions, and the time data adds a third dimension. For example, if you input scalar data to the block, the logged output data has the dimension `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
`uintN`	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.

Version History
Introduced in R2015a

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 systems using Fixed-Point Designer™.

See Also

Blocks
Dual Port RAM | Simple Dual Port RAM | Single Port RAM | Dual Rate 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"

coder.hdl.loopspec

Unroll or stream loops in generated HDL and SystemC code

Syntax

```
coder.hdl.loopspec('unroll')
coder.hdl.loopspec('unroll',unroll_factor)
coder.hdl.loopspec('stream')
coder.hdl.loopspec('stream',stream_factor)

coder.hdl.loopspec('unroll')
coder.hdl.loopspec('unroll',unroll_factor)
coder.hdl.loopspec('pipeline')
coder.hdl.loopspec('pipeline',initiation_interval)
```

Description

For HDL Code Generation: `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.

For HDL Code Generation: `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.

For HDL Code Generation: `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.

For HDL Code Generation: `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.

For SystemC Code Generation: `coder.hdl.loopspec('unroll')` fully unrolls the `for`-loop in the generated SystemC code during synthesis. You must insert this pragma at the first line of the `for`-loop body in your MATLAB code.

This pragma does not affect MATLAB simulation behavior.

For SystemC Code Generation: `coder.hdl.loopspec('unroll',unroll_factor)` unrolls a `for`-loop by the specified unrolling factor, `unroll_factor`, in the generated SystemC code during synthesis. You must insert this pragma at the first line of the `for`-loop body in your MATLAB code.

This pragma does not affect MATLAB simulation behavior.

For SystemC Code Generation: `coder.hdl.loopspec('pipeline')` pipelines the `for`-loop with the initiation interval of 1, in the generated SystemC code. You must insert this pragma before the `for`-loop to be pipelined.

This pragma does not affect MATLAB simulation behavior.

For SystemC Code Generation: `coder.hdl.loopspec('pipeline',initiation_interval)` pipelines the `for`-loop by the specified initiation interval, `initiation_interval`, in the generated SystemC code. You must insert the pragma before the `for`-loop to be pipelined. The `initiation_interval` represents the number of clock cycles before the start of the next iteration of the `for`-loop.

This pragma does not affect MATLAB simulation behavior.

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);
```

```matlab
    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.

```matlab
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.

```matlab
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
```

Completely unroll MATLAB loop in generated SystemC code

Unroll loop in generated SystemC code.

```matlab
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 SystemC code

Generate a loop statement in the SystemC 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
```

Pipeline `for`-loop in generated SystemC code

In the generated SystemC code, the `for`-loop is pipelined during synthesis.

```
function y = myFun(a, b, c)
    y = int16(zeros(1,10));
    coder.hdl.loopspec('pipeline');
    for i = 1:10
        y(i) = i + a*b + c;
    end
end
```

Pipeline `for`-loop with `initiation_interval` in generated SystemC code

In the generated SystemC code, the `for`-loop is pipelined with the `initiation_interval` as 2 during synthesis.

```
function y = myFun(a, b, c)
    y = int16(zeros(1,10));
    coder.hdl.loopspec('pipeline',2);
    for i = 1:10
        y(i) = i + a*b + c;
    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

initiation_interval — Initiation interval
positive integer

Initiation interval to pipeline `for`-loops, specified as a positive integer.

The default value of initiation interval is `1`.

Example: 3

Version History
Introduced in R2015a

R2022a: Support for SystemC Code Generation

This function supports SystemC code generation.

See Also

Topics
"Optimize MATLAB Loops"

coder.hdl.pipeline

Insert pipeline registers at output of MATLAB expression

Syntax

```
out = coder.hdl.pipeline(expr)
out = coder.hdl.pipeline(expr,num)
```

Description

`out = coder.hdl.pipeline(expr)` inserts one pipeline register at the output of `expr` in the generated HDL code. This pragma does not affect MATLAB simulation behavior.

Use this pragma to specify exactly where to insert pipeline registers. For example, in a MATLAB assignment statement, you can specify the coder.hdl.pipeline pragma:

- On the entire right side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see "Port Registers".

`out = coder.hdl.pipeline(expr,num)` inserts `num` pipeline registers at the output of `expr` in the generated HDL code. This pragma does not affect MATLAB simulation behavior.

Use this pragma to specify exactly where to insert pipeline registers. For example, in a MATLAB assignment statement, you can specify the coder.hdl.pipeline pragma:

- On the entire right side of the assignment statement.
- On a subexpression.
- By nesting multiple pragmas.
- On a call to a subfunction, if the subfunction returns a single value. You cannot specify the pragma for a subfunction that returns multiple values.

If you enable distributed pipelining, HDL Coder can move the pipeline registers to break the critical path.

HDL Coder cannot insert a pipeline register at the output of a MATLAB expression if any of the variables in the expression are:

- In a loop.
- A persistent variable that maps to a state element, like a state register or RAM.
- An output of a function. For example, in the following code, you cannot add a pipeline register for an expression containing `y`:

```
function [y] = myfun(x)
y = x + 5;
end
```

- In a data feedback loop. For example, in the following code, you cannot pipeline an expression containing the `t` or `pvar` variables:

```
persistent pvar;
t = u + pvar;
pvar = t + v;
```

You cannot use `coder.hdl.pipeline` to insert a pipeline register for a single variable or other no-op expression. To learn how to insert a pipeline register for a function input variable, see "Port Registers".

Examples

Insert one pipeline register at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert a single pipeline register.

```
y = coder.hdl.pipeline(a + b * c);
```

Insert multiple pipeline registers at output of MATLAB expression

At the output of a MATLAB expression, `a + b * c`, insert three pipeline registers.

```
    y = coder.hdl.pipeline(a + b * c, 3);
```

Insert pipeline registers at intermediate stage of MATLAB expression

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers.

```
y = a + coder.hdl.pipeline(b * c, 5);
```

Insert pipeline registers at intermediate stage and at output of MATLAB expression

At an intermediate stage and at the output of a MATLAB expression, use nested `coder.hdl.pipeline` pragmas to insert pipeline registers.

For a MATLAB expression, `a + b * c`, after the computation of `b * c`, insert five pipeline registers, and insert two pipeline registers at the output of the whole expression.

```
y = coder.hdl.pipeline(a + coder.hdl.pipeline(b * c, 5),2);
```

Input Arguments

expr — MATLAB expression to pipeline
MATLAB expression

MATLAB expression to pipeline. At the output of this expression in the generated HDL code, insert pipeline registers.

Example: `a + b`

num — Number of registers
MATLAB expression

Number of pipeline registers to insert at the output of `expr` in the generated HDL code, specified as a positive integer.

Example: 3

Version History
Introduced in R2015a

See Also

Topics
"Pipeline MATLAB Expressions"
"Pipelining MATLAB Code"

coder.hdl.constrainlatency

Specify the minimum and maximum acceptable hardware latency for a MATLAB function for SystemC code generation

Syntax

```
coder.hdl.constrainlatency(minimum_latency, maximum_latency)
```

Description

`coder.hdl.constrainlatency(minimum_latency, maximum_latency)` pragma is created that enables you to specify the desired hardware latency range for a block of code for SystemC code generation.

Examples

Constrain the latency of the block of code within the specified range

Consider the MATLAB functions `hdltest` and `region1`. The latency of the `region1` function is constrained between 0 and 2 clock cycles in the generated SystemC code.

```
function y = hdltest(a,b)
    u = a+8;
    v = b+6;
    y = region1(u,v);
end

function z = region1(a,b)
    % Optional comment between function defination and pragma
    coder.hdl.constrainlatency(0,2);
    m = a+1;
    n = b+2;
    z = m*n;
end
```

Input Arguments

`minimum_latency` — Minimum latency of the MATLAB function
positive integer

Minimum latency, specified as a positive integer.

Example: 4

`maximum_latency` — Maximum latency of the MATLAB function
positive integer

Maximum latency, specified as a positive integer.

Example: 10

Version History
Introduced in R2022a

See Also

Topics
"Optimize MATLAB Loops"

hdlsetuphlstoolpath

Set up system environment to access High Level Synthesis (HLS) software

Syntax

```
hdlsetuphlstoolpath('ToolName',TOOLNAME,'ToolPath',
TOOLPATH,'SimulationToolPath',SIMULATIONTOOLPATH)
```

Description

`hdlsetuphlstoolpath('ToolName',TOOLNAME,'ToolPath',`
`TOOLPATH,'SimulationToolPath',SIMULATIONTOOLPATH)` adds a third-party HLS tool to your system path. It sets up the system environment variables for the synthesis tool. To configure one or more supported third-party HLS tools to use with HDL Coder, use the `hdlsetuphlstoolpath` function. You cannot set up third-party HLS tools in Simulink Online.

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". HDL Workflow Advisor is not available in Simulink Online.

Examples

Set Up Cadence Stratus

The following command sets the synthesis tool path to point to an installed Cadence Stratus executable file. You must have already installed Cadence Stratus version 21.2 or above.

```
hdlsetuphlstoolpath('ToolName','Cadence Stratus','ToolPath','/usr/cadence/stratus/bin/',...
 'SimulationToolPath', '/usr/cadence/xcel/tools/bin');
```

Check Synthesis Tool Setup

To check your Cadence Stratus synthesis tool setup in MATLAB, try launching the tool with the following command:

```
!stratus_ide
```

Input Arguments

TOOLNAME — Synthesis tool name
character vector

Synthesis tool name, specified as a character vector.

Example: `'Cadence Stratus'`

Note `Cadence Stratus` is the only supported synthesis tool for SystemC.

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: `'/usr/cadence/stratus/bin/'`

SIMULATIONTOOLPATH — Full path to the simulation tool executable
character vector

Full path to the simulation tool executable, specified as a character vector.

Example: `'/usr/cadence/xcelium/tools/bin'`

Note For Cadence Stratus supported simulation tool is Xcelium™.

Tips

- The `hdlsetuphlstoolpath` function changes the system path and system environment variables for only the current MATLAB session. To execute `hdlsetuphlstoolpath` programmatically when MATLAB starts, add `hdlsetuphlstoolpath` to your `startup.m` script.

Version History
Introduced in R2022a

See Also
`startup` | `setenv`

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

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.

Creation

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'`

Data Types: `char` | `string`

FPGAVendor — Vendor name
'' (default) | `'Altera'` | `'Xilinx'` | `'Microchip'`

FPGA vendor name, specified as a character vector.

Example: `'Xilinx'`

Data Types: `char` | `string`

FPGAFamily — FPGA family name
'' (default) | character vector

FPGA family name, specified as a character vector.

Example: `'Zynq'`

Data Types: `char` | `string`

FPGADevice — FPGA device identifier
'' (default) | character vector

FPGA device identifier, specified as a character vector.

Example: `'xc7z020'`

Data Types: `char` | `string`

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'`

Data Types: `char` | `string`

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'`

Data Types: `char` | `string`

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'}`

Example: `{'Microchip Libero SoC'}`

Data Types: `char` | `string`

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

Data Types: `int32`

Methods

Public Methods

addDeviceTree	Add device tree for board object
addDeviceTreeIncludeDirectory	Specify the path of an include file to compile the device tree against
addEthernetMACInterface	Define Ethernet MAC interface for board object

addExternalIOInterface	Define external IO interface for board object
addExternalPortInterface	Define external port interface for board object
validateBoard	Check property values in board object

Version History
Introduced in R2015a

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Define Custom Board and Reference Design for Microchip Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addDeviceTree

Class: hdlcoder.Board
Package: hdlcoder

Add device tree for board object

Syntax

addDeviceTree(dtFile)

Description

addDeviceTree(dtFile) registers a new device tree file to an hdlcoder.Board object.

Specify the device tree by using one of these extensions: ".dtb",".dts", or ".dtsi". Use ".dtb" when the device tree is specified as a binary file. Use ".dts" or ".dtsi" when the device tree is specified as a source file.

Input Arguments

dtFile — Device Tree File Name
" " (default) | string | character vector

Device tree file name, specified as a string or character vector.

Example: "board.dtsi"

Limitations

- HDL Coder supports specifying only a single device tree file for a plugin file.
- You cannot specify a binary device tree file for custom board and reference design simultaneously. When specifying device tree files for custom board and reference designs, specify the device trees as source files.

Version History
Introduced in R2021b

See Also
addDeviceTreeIncludeDirectory | hdlcoder.Board

addDeviceTreeIncludeDirectory

Class: `hdlcoder.Board`
Package: `hdlcoder`

Specify the path of an include file to compile the device tree against

Syntax

`addDeviceTreeIncludeDirectory(dirPath)`

Description

`addDeviceTreeIncludeDirectory(dirPath)` specifies a folder path of an include file to compile the device tree against.

Input Arguments

dirPath — Folder path
" " (default) | string | character vector

Absolute path or relative path of the device tree include folder, specified as a string or character vector. The include folder path can be either an absolute path or a path relative to the board plugin folder. You do not need to specify the board plugin folder as an include folder because it is automatically added to the include folder list. The include folders exist on the host computer. If a folder cannot be found, a warning is produced and the include folder is skipped.

Example: `"C:\work\devicetrees\zynq"` specifies an absolute path.

Example: `"devicetrees\zynq"` specifies a path relative to the board plugin folder.

Version History
Introduced in R2021b

See Also
`addDeviceTree` | `hdlcoder.Board`

addEthernetMACInterface

Class: `hdlcoder.Board`
Package: `hdlcoder`

Define Ethernet MAC interface for board object

Syntax

```
addEthernetMACInterface('InterfaceType',interfacetype,'MACAddress',
macaddress,'IPAddress',ipaddress,'NumChannels',channels,'PortAddresses',
portaddresses,'EthernetMACConstraintFile',constraintfile)
```

Description

`addEthernetMACInterface('InterfaceType',interfacetype,'MACAddress',` `macaddress,'IPAddress',ipaddress,'NumChannels',channels,'PortAddresses',` `portaddresses,'EthernetMACConstraintFile',constraintfile)` adds an Ethernet media access controller (MAC) Hub intellectual property (IP) to an `hdlcoder.Board` object.

Use this function if your board has the Ethernet physical layer (PHY) and you want to use the `Ethernet` option for the **Insert AXI Manager (HDL Verifier required)** and **FPGA Data Capture (HDL Verifier required)** reference design parameters in the **Set Target Reference Design** task of the HDL Workflow Advisor.

Note

- AXI manager and FPGA data capture in the HDL Workflow Advisor support programmable logic (PL) Ethernet only. The processing system (PS) Ethernet is not supported.
- FPGA data capture in the HDL Workflow Advisor does not support SGMII interface.

Input Arguments

`interfacetype` — Type of Ethernet interface for target board
`'GMII'` | `'MII'` | `'SGMII'`

Type of Ethernet interface for the target board, specified as `'GMII'`, `'MII'`, or `'SGMII'` according to the interface of your target board.

Example: `'GMII'`

Data Types: `char`

`macaddress` — MAC address of target board
character vector

MAC address of the target board, specified as a character vector containing a six-octet hexadecimal value. The MAC address is the physical address of the target board.

Example: `'0x000A3502218A'`

Data Types: `char`

`ipaddress` — IP address of Ethernet port on target board
character vector

Internet protocol (IP) address of the Ethernet port on the target board, specified as a character vector containing a dotted-quad value. The target IP address must be a set of four integers in the range [0, 255] that are separated by three dots.

Example: `'192.168.0.2'`

Data Types: `char`

`channels` — Number of AXI-stream channels in Ethernet MAC Hub IP
integer in the range [1, 8]

Number of AXI-stream channels in the Ethernet MAC Hub IP, specified as an integer in the range [1, 8].

Example: 2

Data Types: `uint8`

`portaddresses` — UDP port numbers for AXI-stream channels
row vector of integers

User datagram protocol (UDP) port numbers for each AXI-stream channel in the Ethernet MAC Hub IP, specified as a row vector of integers in the range [255, 65535].

Example: [50101 50102]

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`constraintfile` — Ethernet MAC constraint file path for target board
character vector

Ethernet MAC constraint file path for the target board, specified as a character vector containing the path of the constraint file relative to the path of the reference design.

Example: `'../kc705_ethernet_gmii.xdc'`

Data Types: `char`

Examples

Define Ethernet MAC Interface for Kintex-7 KC705 Development Board

Define an Ethernet MAC interface for an `hdlcoder.Board` object that you create for the Xilinx Kintex-7 KC705 development board.

Create an `hdlcoder.Board` object for the Kintex-7 KC705 board.

```
hB = hdlcoder.Board;
hB.BoardName = 'Xilinx Kintex-7 KC705 development board';
```

Define an Ethernet MAC interface for the created board object.

```
hB.addEthernetMACInterface( ...
        'InterfaceType','GMII', ...
        'MACAddress','0x000A3502218A', ...
        'IPAddress','192.168.0.2', ...
        'NumChannels',2,...
        'PortAddresses',[50101 50102], ...
        'EthernetMACConstraintFile','../kc705_ethernet_gmii.xdc');
```

Tips

*   For details about the Ethernet interface type, MAC and IP addresses, number of AXI-stream channels in the Ethernet MAC Hub IP and their UDP port numbers, and constraints for your board, view the board documentation.

Version History

Introduced in R2022b

See Also

`hdlcoder.Board`

Topics

"Define Custom Parameters and Callback Functions for Custom Reference Design"

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. You can use DUT ports that have flattened word lengths greater than 128 bits. To model DUT ports that have word lengths greater than 128 bits, use vector data types. For example, to model a 512-bit Data port, use a vector port with four 128-bit scalar ports.

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.

Version History
Introduced in R2015a

See Also
hdlcoder.Board | addExternalPortInterface

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Board and Reference Design Registration System"

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 `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.

Version History
Introduced in R2015a

See Also

addExternalIOInterface | hdlcoder.Board

Topics

"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Board and Reference Design Registration System"

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.

Version History
Introduced in R2015a

See Also
`hdlcoder.Board`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Board and Reference Design Registration System"

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.

Creation

`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` | `Microchip Libero SoC`

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** drop-down 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: {'2020.2'}

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 supports only 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` method.

Example: {'my_ip_core'}

Example: {fullfile('data', 'ps7_system_prj.xml')}

Example: {'ip'}

Example: {'ipcore'}

Example: {'pcores'}

DeviceTreeName — Linux device tree name
character vector

Specify the device tree file name. For an example that shows how to use different device tree file names when mapping the DUT ports to different AXI4-Stream channels, see "Dynamically Create Reference Design with Master Only or Slave Only AXI4-Stream Interface".

Example: `'devicetree_axistream_iio.dtb'`

AddMATLABAXIManagerParameter — Control visibility of Insert AXI Manager parameter
`'true'` (default) | `'false'` | logical data type

Control visibility of the **Insert AXI Manager (HDL Verifier required)** parameter in the **Set Target Reference Design** task of the HDL Workflow Advisor. By default, the property value is `'true'`, which means that the parameter is visible in the **Set Target Reference Design** task. To disable the parameter, set the property value to `'false'`.

After you enable this property, to specify whether you want the code generator to insert the AXI manager IP, use the `MATLABAXIManagerDefaultValue` property.

This property is optional.

Example: `'false'`

MATLABAXIManagerDefaultValue — Specify whether to insert AXI manager IP
`'off'` (default) | `'JTAG'` | `'Ethernet'` | character vector

Specify whether you want the code generator to insert the AXI manager IP. The values that you specify are the choices for the **Insert AXI Manager (HDL Verifier required)** drop-down in the **Set Target Reference Design** task of the HDL Workflow Advisor. To specify insertion of the AXI manager IP automatically, before you set this property, set the `AddMATLABAXIManagerParameter` property to `'true'`.

This property is optional. Set this property to one of these values.

- `'off'` — Disable insertion of the AXI manager IP.
- `'JTAG'` — Enable AXI manager IP insertion for the JTAG connection. This value inserts the AXI Manager IP into your reference design.
- `'Ethernet'` — Enable AXI manager IP insertion for the Ethernet connection. This value inserts the UDP AXI Manager IP into your reference design.

Example: `'JTAG'`

IPCacheZipFile — IP cache file to include in the project
`''` (default) | `'ipcache.zip'` | character vector

Specify the IP cache zip file to include in your project. When you run the `IP Core Generation` workflow in the HDL Workflow Advisor, the code generator extracts this file in the **Create Project** task. The **Build FPGA Bitstream** task reuses the IP cache, which accelerates reference design synthesis.

This property is optional.

Example: `'ipcache.zip'`

ReportTimingFailure — Report timing failures as warnings or errors
`'hdlcoder.ReportTiming.Warning'` (default) | `'hdlcoder.ReportTiming.Error'`

Specify whether you want the code generator to report timing failures in the **Build FPGA Bitstream** task as warnings or errors. When you run the `IP Core Generation` workflow in the HDL Workflow Advisor, by default, the code generator reports any timing failures as error. If you have implemented the custom logic to resolve timing failures, you can specify these failures to be reported as warning instead of error. To learn more, see "Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows".

This property is optional.

Example: `'hdlcoder.ReportTiming.Warning'`

HasProcessingSystem — Specify if reference design has existing Processing System (PS)
`true` (default) | `false` | logical data type

Specify if the reference design has an existing PS.

Example: `'false'`

GenerateIPCoreDeviceTreeNodes — Enable generation of device tree nodes for HDL Coder IP core
`false` (default) | `true` | logical data type

Enable generation of device tree nodes for an HDL Coder generated IP core, and then insert the nodes into the device tree. To enable the generation of device tree nodes for the IP core, `HasProcessingSystem` must be set to `true`.

Do not enable this property if you do not need any additional device tree nodes to be inserted into the registered device tree for the generated IP core.

Example: `'true'`

ResourcesUsed — Board resources used by reference design
structure

Board resources used by reference design, returned as a structure with the fields:

LogicElements — Reference design resources utilized by FPGA lookup tables (LUTs)
0 (default)

Reference design resources utilized by FPGA lookup tables (LUTs), specified as a number.

Example: `hRD.ResourcesUsed.LogicElements = 100`

DSP — Reference design resources utilized by FPGA DSP slices
0 (default)

Reference design resources utilized by FPGA DSP slices, specified as a number.

Example: `hRD.ResourcesUsed.DSP = 3`

RAM — Reference design resources utilized by FPGA board RAM resources
0 (default)

Reference design resources utilized by FPGA board RAM resources, specified as a number.

Example: `hRD.ResourcesUsed.RAM = 32000`

Methods

Public Methods

CallbackCustomProgrammingMethod	Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor
CustomizeReferenceDesignFcn	Function handle for callback function that gets executed before Set Target Interface task in the HDL 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 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
addAXI4MasterInterface	Add and define AXI4 Master interface
addAXI4SlaveInterface	Add and define AXI4 slave interface
addAXI4StreamInterface	Add AXI4-Stream interface
addAXI4StreamVideoInterface	Add AXI4-Stream Video interface
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
addCustomLiberoDesign	Specify Microchip Libero SoC exported block design Tcl file
addDeviceTree	Add device tree for reference design object
addDeviceTreeIncludeDirectory	Specify the path of an include file to compile the device tree against
addIPRepository	Include IP modules from your IP repository folder in your custom reference design
addInternalIOInterface	Add and define internal IO interface between generated IP core and existing IP cores
addParameter	Add and define custom parameters for your reference design
validateReferenceDesign	Check property values in reference design object

Version History
Introduced in R2015a

See Also
`hdlcoder.Board`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"

"Define Custom Board and Reference Design for Intel SoC Workflow"
"Define Custom Board and Reference Design for Microchip 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"

addAXI4MasterInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add and define AXI4 Master interface

Syntax

```
addAXI4MasterInterface(InterfaceConnection',Interface_Connection)
addAXI4MasterInterface(InterfaceConnection',
Interface_Connection,'TargetAddressSegments',Target_Address_Segments)
addAXI4MasterInterface(InterfaceConnection',Interface_Connection, Name,Value)
addAXI4MasterInterface(InterfaceConnection',
Interface_Connection,'TargetAddressSegments',Target_Address_Segments,
Name,Value)
```

Description

addAXI4MasterInterface(InterfaceConnection',Interface_Connection) adds and defines an AXI4 Master interface for an Intel Qsys reference design.

addAXI4MasterInterface(InterfaceConnection', Interface_Connection,'TargetAddressSegments',Target_Address_Segments) adds and defines an AXI4 Master interface for a Xilinx Vivado reference design.

addAXI4MasterInterface(InterfaceConnection',Interface_Connection, Name,Value) adds and defines an AXI4 Master interface for an Intel Qsys reference design, with additional options specified by one or more Name,Value pair arguments.

addAXI4MasterInterface(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_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. The format of the target address segment is {'SegmentName', low address, range}. You must use a power of 2 value for range.

Example: '{{'mig_7series_0/memmap/
memaddr',hex2dec('40000000'),hex2dec('40000000')}}'

Tip To add more than one AXI4 Master interface to your reference design, call the
`addAXI4MasterInterface` method multiple times depending on the number of interfaces you want
to add. For each additional interface, specify a unique `InterfaceID`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the
argument name and `Value` is the corresponding value. Name-value arguments must appear after
other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

InterfaceID — 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'

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.

HasMemoryConnection — Indicate AXI4 Master connection to memory
'false' (default) | 'true'

Indicate if the AXI4 Master interface is connected to a memory region, specified as `true` or `false`. Set this value to `true` to enable device tree generation for the IP core AXI4 Master interfaces.

Example: `'HasMemoryConnection'=true`

Data Types: `logical`

ProcessorAccessibleMemoryRegion — Regions of memory accessible to the AXI4 Master interface and processor
numeric array

Regions of memory accessible to the AXI4 Master interface and the processor, specified as a numeric array. Use these regions to exchange data with the AXI4 Master interface. Specify the memory region as an address in range tuple. If there are multiple accessible memory regions, specify an N-by-two matrix, where each row is an address in range tuple.

Example: `ProcessorAccessibleMemoryRegion=[0x80000000 0x200000]`

Example: `ProcessorAccessibleMemoryRegion=[0x80000000 0x200000;0xA0000000 0x200000]`

DeviceTreeMemoryRegionNode — Reference to the processor-accessible memory regions in device tree
char | string | string array | cell array of character vectors

Reference to the processor-accessible memory regions in the registered device tree, specified as a character vector, string, string array, or a cell array of character vectors. Set the value to match the name of the corresponding memory region node in the registered device tree. The memory region node in the device tree should have the same address space as defined in the `ProcessorAccessibleMemoryRegion`. If there are multiple memory regions, specify a list of reference nodes. References to device tree nodes must start with "&". To reference a node by its label, specify "&" before the label. For example, "&myLabel". To reference a node by its path, specify the path inside "&{"and"}". For example, "&{/myNode/childNode}".

Example: `DeviceTreeMemoryRegionNode="&plmem"`

Example: `DeviceTreeMemoryRegionNode=["&plmem","&sharedmem"]`

Version History
Introduced in R2017b

See Also
addClockInterface | addAXI4StreamInterface | hdlcoder.ReferenceDesign

Topics
"Perform Matrix Operation Using External Memory"
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Model Design for AXI4 Master Interface Generation"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

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 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 arguments.

Input Arguments

ref_design_port — Reference design port name
'' (default) | character vector

Reference design port that is connected to the AXI4 or AXI4-Lite interface, specified as a character vector. For reference designs based on Intel Qsys, when you want to connect multiple AXI Master IPs to the AXI4 or AXI4-Lite interface, specify each of the AXI Master instance names and the corresponding port names as a cell array of character vectors.

Example: 'axi_interconnect_0/M00_AXI',
{'hps_0.h2f_axi_master','master_0.master'},...

base_addr — Base address
'' (default) | character vector

Base address for AXI4 or AXI4-Lite slave interface, specified as a character vector.

Example: `'0x40010000'`

master_addr_space — Master interface address space (Vivado only)
'' (default) | character vector

Address space of the master interface connected to this slave interface, specified as a character vector. For Vivado reference designs only. When you want to connect more than one AXI Master IP, specify each of the AXI Master instance names and the corresponding address spaces.

Example: `'processing_system7_0/Data', {'processing_system7_0/Data','hdlverifier_axi_master_0/axi4m'}`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

InterfaceType — Interface type
`{'AXI4-Lite','AXI4'}` (default) | `'AXI4'` | `'AXI4-Lite'`

Type of interface connection, specified as a `character vector` or a `cell array` of character vectors.

Example: `'InterfaceType','AXI4-Lite'` specifies an `'AXI4—Lite'` interface type connection.

InterfaceID — Interface name
`{'AXI4-Lite','AXI4'}` (default) | character vector

Name of the interface, specified as a character vector. When you provide the `InterfaceID`, `InterfaceType` must be set to either `'AXI4'` or `'AXI4—Lite'`.

Example: `'InterfaceID','MyAXI4','InterfaceType','AXI4'` specifies interface name as `'MyAXI4'` and interface type as `'AXI4'`.

IDWidth — Width of ID signals
12 (default) | positive integer

Width of all ID signals, such as `AWID`, `WID`, `ARID`, and `RID`, specified as a positive integer. This property enables you to specify the number of AXI Master interfaces that you want the AXI4 slave interface in the HDL DUT IP core to connect to. The default value is `12`, which enables you to connect the HDL IP core to one AXI Master interface. To connect the IP core to multiple AXI Master interfaces, increase the `IDWidth`. The ID width is tool-specific.

Example: `'IDWidth','13'` might indicate that you want the IP core to connect to two AXI Master interfaces in the reference design.

HasProcessorConnection — Indicate AXI4 slave connection to processor
`true` (default) | `false` | logical data type

Indicate if the processor is one of the masters to the IP core AXI4 slave interface. To enable device tree generation for the IP core AXI4 slave interface, keep this value set to `true`.

Example: `'HasProcessorConnection','false'`

DeviceTreeNodes — Reference to processor AXI4 master bus node in the device tree
" " (default) | string | character vector

Reference to the processor AXI4 master bus node in the device tree. Set this value to match the name of the corresponding bus node in the registered device tree. References to device tree nodes must start with "&". To reference a node by its label, specify "&" before the label, such as "&myLabel". To reference a node by its path, specify the path inside "&{" and "}", such as "&{/myNode/childNode}".

Example: `'DeviceTreeNodes','&fpga_axi'`

Tips

- Before running this method, you must run the `addClockInterface` method.
- The `addAXI4SlaveInterface` method is optional. You can define your own custom reference design without the AXI4 slave interface.
- To connect the HDL IP core for your DUT to multiple AXI Master interfaces in the reference design, use the `IDWidth` property of this method. To learn more, see "Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface".

Version History
Introduced in R2015a

See Also
`addClockInterface` | `hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Define Multiple AXI Master Interfaces in Reference Designs to Access DUT AXI4 Slave Interface"
"Board and Reference Design Registration System"

addAXI4StreamInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add AXI4-Stream interface

Syntax

```
addAXI4StreamInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port)
addAXI4StreamInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port), Name,Value
```

Description

addAXI4StreamInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port) adds an AXI4-Stream interface to an hdlcoder.ReferenceDesign object.

addAXI4StreamInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port), Name,Value adds and defines an AXI4-Stream interface, with additional options specified by one or more Name,Value pair arguments.

Input Arguments

Master_Channel_Port — Reference design port connected to IP core AXI4-Stream master interface
character vector

Reference design port connected to the IP core AXI4-Stream master interface, specified as a character vector. This port must be an AXI4-Stream slave interface. If master_channel_enable is true, you must specify master_channel_port.

Example: 'axi_dma_0/S_AXIS_S2MM'

Slave_Channel_Port — Reference design port connected to IP core AXI4-Stream slave interface
character vector

Reference design port connected to the IP core AXI4-Stream slave interface, specified as a character vector. This port must be an AXI4-Stream master interface. If slave_channel_number is true, you must specify slave_channel_port and slave_channel_data_width.

Example: 'axi_dma_0/M_AXIS_MM2S'

Tip To add more than one AXI4-Stream interface to your reference design, call the addAXI4StreamInterface method multiple times depending on the number of interfaces that you want to add. For each additional interface, specify a unique InterfaceID.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

`InterfaceID` — AXI4-Stream interface name
`'AXI4-Stream'` (default) | character vector

Name of the AXI4-Stream interface that you add to the reference design, specified as a character vector. To create multiple AXI4-Stream interfaces, make sure that you use unique names for each interface.

Example: `'InterfaceID','AXI4-Stream1'`

`MasterChannelEnable` — Master channel enable signal
`'true'` (default) | `'false'`

Master channel enable signal, specified as either `true` or `false`. By default, `MasterChannelEnable` is `true`, so you must specify `master_channel_port`. To use a reference design that has only a Slave channel, set `MasterChannelEnable` to `false` and do not specify the `master_channel_port`.

Example: `'MasterChannelEnable','false'`

`SlaveChannelEnable` — Slave channel enable
`'true'` (default) | `'false'`

Slave channel enable signal, specified as either `true` or `false`. By default, `SlaveChannelEnable` is `true`, so you must specify `slave_channel_port`. To use a reference design that has only a Master channel, set `SlaveChannelEnable` to `false` and do not specify the `slave_channel_port`.

Example: `'SlaveChannelEnable','false'`

`MasterChannelDataWidth` — Master channel data width
32 (default) | `positive integer`

Reference design master channel TDATA bit width, specified as a positive integer. By default, the `master_channel_data_width` is 32 bits.

Example: `'MasterChannelDataWidth',32`

`SlaveChannelDataWidth` — Slave channel data width
32 (default) | `positive integer`

Reference design slave channel TDATA bit width, specified as a positive integer. By default, `slave_channel_data_width` is 32 bits.

Example: `'SlaveChannelDataWidth',32`

`HasDMAConnection` — Indicate AXI4-Stream connection to Direct Memory Access (DMA)
`false` (default) | `true` | logical data type

Indicates if the AXI4-Stream interface is connected to the processor through a DMA. Set this value to `true` to enable device tree generation for the IP core AXI4-Stream interfaces.

Example: `'HasDMAConnection','true'`

DeviceTreeMasterChannelDMANode — Reference to stream to memory-mapped (S2MM) DMA node in the device tree
" " (default) | string | character vector

Reference to the corresponding S2MM DMA node in the registered device tree. Set the value to match the name of the corresponding S2MM DMA node in the registered device tree. References to device tree nodes must start with "&". To reference a node by its label, specify "&" before the label, such as "&myLabel". To reference a node by its path, specify the path inside "&{" and "}", such as "&{/myNode/childNode}".

Set `MasterChannelEnable` to `true` to specify this value.

Example: `'DeviceTreeMasterChannelDMANode','&axi4stream_s2mm'`

DeviceTreeSlaveChannelDMANode — Reference to memory-mapped to stream (MM2S) DMA node in the device tree
" " (default) | string | character vector

Reference to the corresponding MM2S DMA node in the registered device tree. Set the value to match the name of the corresponding MM2S DMA node in the registered device tree. References to device tree nodes must start with "&". To reference a node by its label, specify "&" before the label, such as "&myLabel". To reference a node by its path, specify the path inside "&{" and "}", such as "&{/myNode/childNode}".

Set `SlaveChannelEnable` to `true` to specify this value.

Example: `'DeviceTreeSlaveChannelDMANode','&axi4stream_m2ss'`

Version History
Introduced in R2020a

See Also
`hdlcoder.ReferenceDesign`

Topics
"Deploy Model with AXI-Stream Interface in Zynq Workflow"
"Define Custom Board and Reference Design for Zynq Workflow"
"Model Design for AXI4-Stream Interface Generation"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addAXI4StreamVideoInterface

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add AXI4-Stream Video interface

Syntax

```
addAXI4StreamVideoInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port)
addAXI4StreamVideoInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port), Name,Value
```

Description

addAXI4StreamVideoInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port) adds an AXI4-
Stream Video interface to an hdlcoder.ReferenceDesign object.

addAXI4StreamVideoInterface('MasterChannelConnection',
Master_Channel_Port,'SlaveChannelConnection',Slave_Channel_Port), Name,Value
adds and defines an AXI4-Stream Video interface, with additional options specified by one or more
Name,Value pair arguments.

Input Arguments

Master_Channel_Port — Reference design port connected to IP core AXI4-Stream Video master interface
character vector

Reference design port connected to the IP core AXI4-Stream Video master interface, specified as a
character vector. This port should be an AXI4-Stream Video slave interface. If
master_channel_enable is true, you must specify master_channel_port.

Example: 'RGBtoYCbCr_0/AXI4_Stream_Video_Slave'

Slave_Channel_Port — Reference design port connected to IP core AXI4-Stream Video slave interface
character vector

Reference design port connected to the IP core AXI4-Stream Video slave interface, specified as a
character vector. This port should be an AXI4-Stream Video master interface. If
slave_channel_number is true, you must specify slave_channel_port.

Example: 'YCbCrtoRGB_0/AXI4_Stream_Video_Master'

Tip To add more than one AXI4-Stream Video interface to your reference design, call the
addAXI4StreamVideoInterface method multiple times depending on the number of interfaces
you want to add. For each additional interface, specify a unique InterfaceID.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

`InterfaceID` — AXI4-Stream Video interface name
`'AXI4-Stream Video'` (default) | character vector

Name of the AXI4-Stream Video interface that you add to the reference design, specified as a character vector. To create multiple AXI4-Stream Video interfaces, make sure that you use unique names for each interface.

Example: `'AXI4-Stream Video1','AXI4-Stream Video2'`

`MasterChannelEnable` — Master channel enable signal
`'true'` (default) | `'false'`

Master channel enable signal, specified as either `true` or `false`. By default, `MasterChannelEnable` is `true`, and you must specify `master_channel_port`. To use a reference design that has only a Slave channel, set `MasterChannelEnable` to `false` and do not specify the `master_channel_port`.

Example: `'false'`

`SlaveChannelEnable` — Slave channel enable
`'true'` (default) | `'false'`

Slave channel enable signal, specified as either `true` or `false`. By default, `SlaveChannelEnable` is `true`, and you must specify `slave_channel_port`. To use a reference design that has only a Master channel, set `SlaveChannelEnable` to `false` and do not specify the `slave_channel_port`.

Example: `'false'`

`MasterChannelDataWidth` — Master channel data width
32 (default) | positive integer

Reference design master channel TDATA bit width, specified as a positive integer. By default, the `master_channel_data_width` is 32 bits.

Example: 32

`SlaveChannelDataWidth` — Slave channel data width
32 (default) | positive integer

Reference design slave channel TDATA bit width, specified as a positive integer. By default, `slave_channel_data_width` is 32 bits.

Example: 32

Version History
Introduced in R2020a

See Also

`hdlcoder.ReferenceDesign`

Topics
"Deploy Model with AXI4-Stream Video Interface in Zynq Workflow"
"Define Custom Board and Reference Design for Zynq Workflow"
"Model Design for AXI4-Stream Video Interface Generation"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addDeviceTree

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Add device tree for reference design object

Syntax

addDeviceTree(dtFile)

Description

addDeviceTree(dtFile) registers a new device tree file to an hdlcoder.ReferenceDesign object.

Specify the device tree by using one of these extensions: ".dtb",".dts", or ".dtsi". Use ".dtb" when the device tree is specified as a binary file. Use ".dts" or ".dtsi" when the device tree is specified as a source file.

Input Arguments

dtFile — Device Tree File Name
" " (default) | string | character vector

Device tree file name, specified as a string or character vector.

Example: "refdesign.dtsi"

Limitations

*   HDL Coder supports specifying only a single device tree file for a plugin file.
*   You cannot specify a binary device tree file for custom board and reference design simultaneously. When specifying device tree files for custom board and reference designs, specify the device trees as source files.

Version History
Introduced in R2021b

See Also
addDeviceTreeIncludeDirectory | hdlcoder.ReferenceDesign

addDeviceTreeIncludeDirectory

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify the path of an include file to compile the device tree against

Syntax

addDeviceTreeIncludeDirectory(dirPath)

Description

addDeviceTreeIncludeDirectory(dirPath) specifies a folder path of an include file to compile the device tree against.

Input Arguments

dirPath — Folder path
" " (default) | string | character vector

Absolute path or relative path of the device tree include folder, specified as a string or character vector. The include folder path can be either an absolute path or a path relative to the reference design plugin folder. You do not need to specify the reference design plugin folder as an include folder because it is automatically added to the include folder list. All include folders exist on the host computer. If a folder cannot be found, a warning is produced and the include folder is skipped.

Example: "C:\work\devicetrees\zynq" specifies an absolute path.

Example: "devicetrees\zynq" specifies a path relative to the reference design plugin folder.

Version History
Introduced in R2021b

See Also
addDeviceTree | hdlcoder.Board

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. You can use DUT ports that have flattened word lengths greater than 128 bits. To model DUT ports that have word lengths greater than 128 bits, use vector data types. For example, to model a 512-bit Data port, use a vector port with four 128-bit scalar ports.

`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'`

Version History
Introduced in R2015b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

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 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).

The HDL Workflow Advisor adjusts the **Set Target Frequency** task in different ways, depending on
how you specify these name-value arguments:

- When DefaultFrequencyMHz, MinFrequencyMHz, and MaxFrequencyMHz are not specified or
  all set to zero, the HDL Workflow Advisor does not have the **Set Target Frequency** task.
- When DefaultFrequencyMHz, MinFrequencyMHz, and MaxFrequencyMHz are all equal, the
  HDL Workflow Advisor adds the **Set Target Frequency** task, but the target frequency is not
  adjustable.
- When MinFrequencyMHz is not equal to MaxFrequencyMHz, the HDL Workflow Advisor adds the
  Set Target Frequency task and the target frequency is adjustable. During the **Create Project**
  task, HDL Coder sets the target frequency on the specified clock module IP
  ClockModuleInstance and its specified clock output number ClockNumber in the reference
  design. If the ClockNumber and ClockModuleInstance arguments are not specified, the default
  values are used.

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

`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. If you have a Vivado reference design, use a one-based index to specify the clock port number. If you have a Quartus reference design, use a zero-based index to specify the clock port number.

Example: `'ClockNumber',1` specifies the first output port in the clock module IP to use as the clock port for a Vivado reference design.

Example: `'ClockNumber',0` specifies the first output port in the clock module IP to use as the clock port for a Quartus reference design.

`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.

Version History
Introduced in R2015a

See Also
addAXI4SlaveInterface | hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

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.

Version History
Introduced in R2015a

See Also
addCustomQsysDesign | addCustomVivadoDesign | hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addCustomQsysDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify Altera Qsys project file

Syntax

addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file)

Description

addCustomQsysDesign('CustomQsysPrjFile',qsys_project_file) specifies the Qsys project file that contains the Altera Qsys embedded system design. Use this method if your synthesis tool is Altera Quartus II.

Input Arguments

qsys_project_file — Qsys project file
character vector

Qsys project file for Altera Qsys embedded system design, specified as a character vector.

Example: 'system_soc.qsys'

Tips

- If you have more than one AXI Master IP, in the custom qsys project file, you must make sure that the AXI Master IPs connect to the same AXI Interconnect IP. The AXI4 slave interfaces in the HDL IP core also connect to this Interconnect.

- If your synthesis tool is Xilinx Vivado, use the addCustomVivadoDesign method.

- If your synthesis tool is Xilinx ISE, use the addCustomEDKDesign method.

Version History
Introduced in R2015a

See Also
addCustomEDKDesign | addCustomVivadoDesign | hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addCustomVivadoDesign

Class: `hdlcoder.ReferenceDesign`
Package: `hdlcoder`

Specify Xilinx Vivado exported block design Tcl file

Syntax

`addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)`

Description

`addCustomVivadoDesign('CustomBlockDesignTcl',bd_tcl_file)` specifies the exported block design Tcl file that contains the Xilinx Vivado embedded system design. Use this method if your synthesis tool is Xilinx Vivado.

Input Arguments

`bd_tcl_file` — Block design Tcl file
character vector

Block design Tcl file that you exported from your Xilinx Vivado embedded system design project, specified as a character vector. The Tcl file name must be the same as the Vivado block diagram name.

Example: `'system_top.tcl'`

Tips

- If you have more than one AXI Master IP, in the custom block design Tcl file, you must make sure that the AXI Master IPs connect to the same AXI Interconnect IP. The AXI4 slave interfaces in the HDL IP core also connect to this Interconnect.

- If your synthesis tool is Xilinx ISE, use the `addCustomEDKDesign` method.

- If your synthesis tool is Altera Quartus II, use the `addCustomQsysDesign` method.

Version History
Introduced in R2015a

See Also
`addCustomQsysDesign` | `addCustomEDKDesign` | `hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addCustomLiberoDesign

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Specify Microchip Libero SoC exported block design Tcl file

Syntax

addCustomLiberoDesign('CustomBlockDesignTcl',bd_tcl_file)

Description

addCustomLiberoDesign('CustomBlockDesignTcl',bd_tcl_file) specifies the exported
block design Tcl file that contains the Microchip Libero SoC embedded system design. Use this
method if your synthesis tool is Microchip Libero SoC.

Input Arguments

bd_tcl_file — Block design Tcl file
character vector

Block design Tcl file that you exported from your Microchip Libero SoC embedded system design
project, specified as a character vector. The Tcl file name must be the same as the Libero block
diagram name.

Example: 'design1_led.tcl'

Tips

- If you have more than one AXI Master IP, in the custom block design Tcl file, you must make sure
  that the AXI Master IPs connect to the same AXI Interconnect IP. The AXI4 slave interfaces in the
  HDL IP core also connect to this Interconnect.

- If your synthesis tool is Xilinx ISE, use the addCustomEDKDesign method.

- If your synthesis tool is Altera Quartus II, use the addCustomQsysDesign method.

Version History
Introduced in R2022b

See Also

addCustomQsysDesign | addCustomEDKDesign | hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Microchip Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

addCustomMSSConfig

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Import microcontroller subsystem (MSS) in Microchip Libero Smart design

Syntax

addCustomMSSConfig('CustomMSSConfig',MSS_config)

Description

addCustomMSSConfig('CustomMSSConfig',MSS_config) specifies the microcontroller subsystem (MSS) configuration file that contains the processor information such as peripherals, interfaces and clock. Use this method if your synthesis tool is Microchip Libero SoC.

Input Arguments

MSS_config — MSS Configuration file
character vector

MSS configuration file that is used to import MSS IP in Microchip Libero SoC Smart design.

Example: 'ICICLE_MSS.cfg'

Version History
Introduced in R2022b

See Also
addCustomLiberoDesign | hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Microchip Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

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,'NotExistMessage',
Not_Exist_Message)
```

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,'NotExistMessage',
Not_Exist_Message)` adds IP modules that are in the IP repository folder to your reference design project and displays a message if the modules do not exist.

Before you use this method, define the IP list function that points to the IP modules in the repository folder. You can also specify an optional root directory in the IP list function. This enables you to add IP repositories that do not have to be located relative to the IP list function. 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'`

Not_Exist_Message — 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'`

Version History
Introduced in R2017a

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

CallbackCustomProgrammingMethod

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for custom callback function that gets executed during Program Target Device task in the Workflow Advisor

Syntax

CallbackCustomProgrammingMethod

Description

CallbackCustomProgrammingMethod registers a function handle for the callback function that gets executed when running the **Program Target Device** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, then use this syntax to register the function handle:

hRD.CallbackCustomProgrammingMethod = @my_reference_design.callback_CustomProgrammingMethod;

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is callback_PostBuildBitstream, located in the reference design package folder, +my_reference_design.

With this callback function, you can specify a custom programming method to program the target device. This example code shows how to create the callback function.

```
function [status, log] = callback_CustomProgrammingMethod(infoStruct)
% Reference design callback function for custom programming method
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.BitstreamPath: the path to the generated FPGA bitstream file
% infoStruct.ToolProjectFolder: the path to synthesis tool project folder
% infoStruct.ToolProjectName: the synthesis tool project name
% infoStruct.ToolCommandString: the command for running a tcl file
%
% status: process run status
%         status == true means process run successfully
%         status == false means process run failed
% log:    output log string
status = true;
log = sprintf('Run custom programming method callback...\n');


% Enter your commands for custom programming here
% ...
% ...
```

```
end
```

In the HDL Workflow Advisor, HDL Coder selects the custom programming method to program the target SoC device. If you do not specify the custom programming method, HDL Coder provides `JTAG` and `Download` as the options to program the target device.

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

Version History
Introduced in R2016a

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

CustomizeReferenceDesignFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed before Set Target Interface task in the HDL Workflow Advisor

Syntax

CustomizeReferenceDesignFcn

Description

CustomizeReferenceDesignFcn registers a function handle for the callback function that gets executed before **Set Target Interface** task in the HDL Workflow Advisor. By using this callback function, you can customize the block design Tcl file, reference design interfaces, reference design interface properties, and IP repositories in your reference design.

Tip Before you use the callback function, in the plugin_rd.m file, you must define a reference design parameter by using the addParameter method of the hdlcoder.ReferenceDesign class.

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.CustomizeReferenceDesignFcn = @my_reference_design.customcallback_aximaster;
```

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 customcallback_aximaster, and is located in the reference design package folder +my_reference_design.

For example, you can specify whether you want an AXI4 Master or AXI4 only reference design. In the plugin_rd.m file, use the addParameter method to specify the different parameter choices as a dropdown.

```
% ...

% Parameter For  calling AXI4 master interface from Callback function

hRD.addParameter('ParameterID',     'interface_name', ...
                 'DisplayName',     'Reference_design_interface_name', ...
                 'DefaultValue',    'AXI4 Master',...
                 'ParameterType',   hdlcoder.ParameterType.Dropdown, ...
                 'Choice',          {'AXI4 Master','AXI4 Only'});

hRD.CustomizeReferenceDesignFcn = @my_reference_design.customcallback_aximaster;

% ...
```

In the callback function, you can dynamically change the reference design interfaces depending on the interface types. 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.

```matlab
% Callback function to control AXI Master and AXI4 interface information dynamically

function customcallback_aximaster(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 struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ReferenceDesignToolVersion: Reference design Tool Version set in 1.2 Task

hRD = infoStruct.ReferenceDesignObject;
paramStruct = infoStruct.ParameterStruct;
interface_name = paramStruct.interface_name;

    if strcmp(interface_name, 'AXI4 Only')
        hRD.addAXI4SlaveInterface( ...
            'InterfaceConnection', 'axi_interconnect_2/M01_AXI', ...
            'BaseAddress',          '0x00000000', ...
            'MasterAddressSpace',   'hdlverifier_axi_master_0/axi4m', ...
            'InterfaceType',        'AXI4');

    % To validate IP repository and AXI4 master interface signals
    elseif strcmp(interface_name,'AXI4 Master')
      hRD.addAXI4SlaveInterface( ...
        'InterfaceConnection', 'axi_interconnect_2/M01_AXI', ...
        'BaseAddress',          '0x00000000', ...
        'MasterAddressSpace',   'hdlverifier_axi_master_0/axi4m', ...
        'InterfaceType',        'AXI4');

    hRD.addAXI4MasterInterface(...
        'InterfaceID',          'AXI4 Master', ...
        'ReadSupport',          true, ...
        'WriteSupport',         true, ...
        'MaxDataWidth',         128, ...
        'AddrWidth',            32, ...
        'DefaultReadBaseAddr',  hex2dec('40000000'), ...
        'DefaultWriteBaseAddr', hex2dec('41000000'), ...
        'InterfaceConnection', 'axi_interconnect_1/S01_AXI',...
        'TargetAddressSegments', {{'mig_7series_0/memmap/memaddr',hex2dec('40000000'),hex2dec('40000000')}})

    end
end
```

In the HDL Workflow Advisor, on the **Set Target Reference Design** task, when you select the custom reference design that you want to customize for the target board, HDL Coder populates the reference design parameters. Depending on the parameter choices such as the interface types you specify, the callback function is evaluated. After you run this task, when you select the **Set Target Interface** task, the target interfaces get populated in the Target platform interface table.

Version History
Introduced in R2020a

See Also
hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Reference Design"

"Define Custom Parameters and Callback Functions for Custom Reference Design"
"Customize Reference Design Dynamically Based on Reference Design Parameters"
"Board and Reference Design Registration System"

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 displays the **Generate Software Interface** task in the HDL Workflow Advisor, but has the **Generate Simulink software interface model** check box greyed out. If you have HDL Verifier installed, on the **Set Target Reference Design** task, set **Insert AXI Manager (HDL Verifier required)** to `JTAG` or `Ethernet` based on the interface that communicates between your host machine and the FPGA board. For Ethernet, specify the IP address of the FPGA board using the **Board IP Address** parameter.

Note By default, the `Ethernet` option is available for only the Artix-7 35T Arty, Kintex-7 KC705, and Virtex-7 VC707 boards. To enable this option for other Xilinx boards that have the Ethernet physical layer (PHY), manually add the Ethernet media access controller (MAC) Hub IP in the `plugin_board` file using the `addEthernetMACInterface` method before you launch the **HDL Workflow Advisor** tool.

You can then generate a host interface model or host interface script or both in the **Generate Software Interface** task to rapidly prototype and test the HDL IP core functionality by using the AXI Manager. See "Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core".

Note To generate a host interface model, you must map each DUT signal that you want to capture to the `AXI4` or `AXI4-Lite` interfaces in the **Set Target Interface** task.

If you are targeting SoC devices, use this syntax depending on whether you are using an Intel SoC or a Xilinx Zynq platform.

```
hRD.EmbeddedCoderSupportPackage = ...
                   hdlcoder.EmbeddedCoderSupportPackage.Zynq;
```

```
hRD.EmbeddedCoderSupportPackage = ...
                        hdlcoder.EmbeddedCoderSupportPackage.AlteraSoC;
```

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow"
"IP Core Generation Workflow for Standalone FPGA Devices"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

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. When this function is called, you cannot run the build process externally. Before you run the **Build FPGA Bitstream** task, clear the **Run build process externally** check box to build the FPGA bitstream within 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 struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.BitstreamPath: the path to generated FPGA bitstream file
%
% status: process run status
%        status == true means process run successfully
%        status == false means process run failed
% log:    output log string


status = false;
log = sprintf('Run post build bitstream callback\n%s\n%s\n', infoStruct.HDLModelDutPath, infoStruct.BitstreamPath);

% Exporting the InfoStruct Contents
% ...
% ...

end
```

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. Use this information to specify custom settings for the build process and bitstream generation.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

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.

```matlab
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 struct format
% 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.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

PostSWInterfaceFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for custom callback function that gets executed after Generate Software Interface 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** 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 struct format
% 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', swDutPath), 'Position', [50, 50, 80, 80]);

end
```

In the HDL Workflow Advisor, when HDL Coder runs the **Generate Software Interface** 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.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

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 struct format
% 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.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

PostTargetReferenceDesignFcn

Class: hdlcoder.ReferenceDesign
Package: hdlcoder

Function handle for callback function that gets executed after Set Target Reference Design task in the HDL Workflow Advisor

Syntax

PostTargetReferenceDesignFcn

Description

PostTargetReferenceDesignFcn registers a function handle for the callback function that gets called at the end of the **Set Target Reference Design** task in the HDL Workflow Advisor. If hRD is the reference design object that you construct with the hdlcoder.ReferenceDesign class, use this syntax to register the function handle:

hRD.PostTargetReferenceDesignFcn = @my_reference_design.callback_PostTargetReferenceDesign

To define your callback function, create a file that defines a MATLAB function and add it to your MATLAB path. You can use any name for the callback function. In this example, the function name is callback_PostTargetReferenceDesign, and is located in the reference design package folder +my_reference_design.

With the callback function, you can enable custom validations for your design. This example code shows how to create the callback function and validate that the reset type is synchronous.

```
function callback_PostTargetReferenceDesign(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% 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.

Version History
Introduced in R2016b

See Also
`hdlcoder.ReferenceDesign`

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel 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"

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.

Version History

Introduced in R2015a

See Also

hdlcoder.ReferenceDesign

Topics
"Define Custom Board and Reference Design for Zynq Workflow"
"Define Custom Board and Reference Design for Intel SoC Workflow"
"Register a Custom Board"
"Register a Custom Reference Design"
"Board and Reference Design Registration System"

fpga

Access target FPGA or SoC device from MATLAB

Description

This object represents a connection from MATLAB to the target FPGA or SoC device. To interact with the target device, use this object with the functions listed in "Object Functions" on page 7-111.

Creation

Syntax

```
hFPGA = fpga(Vendor)
hFPGA= fpga(p)
```

Description

`hFPGA = fpga(Vendor)` creates a hardware object that you can use to connect to your target Intel or Xilinx pure-FPGA device.

`hFPGA= fpga(p)` creates a hardware object by taking in a processor connection object `intelsoc` or `xilinxsoc` that you can use to connect to your target Intel or Xilinx SoC device.

Input Arguments

Vendor — Name of vendor
`"Xilinx"` | `"Intel"`

Connect to an Intel or Xilinx pure-FPGA target device from MATLAB.

When connecting to an SoC board, specify a processor connection object as the input.

Example: `hFPGA = fpga("Intel")`

Data Types: `string` | `char`

p — Connection to processor on SoC board
`xilinxsoc` object | `intelsoc` object

For SoC boards, specify a processor connection object as the input when creating the FPGA connection object. Pass in a `xilinxsoc` object for Xilinx SoC board. Pass in an `intelsoc` object for an Intel SoC board.

Example: `hFPGA = fpga(p)`

Output Arguments

hFPGA — Connection to FPGA on a Xilinx or Intel board
`fpga` object

Connection from MATLAB to the FPGA on the Xilinx or Intel board. You can use this connection to interact with the hardware board and attached devices.

Object Functions

Use the object functions to interact with your FPGA or SoC device.

addAXI4SlaveInterface	Write data to IP core or read data from IP core using AXI4 or AXI4-Lite interface
addAXI4StreamInterface	Write data to IP core or read data from IP core using AXI4-Stream interface
mapPort	Maps a DUT port to specified AXI4 interface in HDL IP core
writePort	Write data to a DUT port from MATLAB
readPort	Reads output data and returns it with the port data type and dimension
release	Release the hardware resources associated with the fpga object

Examples

Connect to Xilinx Target

Create an `fpga` object to connect to a Xilinx target device.

Create an `fpga` object with `Vendor` as `Xilinx`.

```
hFPGA = fpga("Xilinx")


hFPGA =

  fpga with properties:

      Vendor: "Xilinx"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Connect to Intel Target

Create an `fpga` object to connect to an Intel target.

Create an `fpga` object with `Intel` as `Vendor`.

```
hFPGA = fpga("Intel")


hFPGA =

  fpga with properties:

      Vendor: "Intel"
   Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Program FPGA on Xilinx SoC Board

Create a hardware processor object to connect to a Xilinx SoC board. You can use the object to program your FPGA on-board the SoC board at the MATLAB command-line and to create an `fpga` object that connects to the FPGA through the processor on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Program the FPGA and set the corresponding device tree from processor on the SoC board by using the `xilinxsoc` object function `programFPGA`.

```
programFPGA(p, "myBitstream.bit", "myDeviceTree.dtb");
```

For more information on device trees, see "Generate Device Tree for IP Core".

Connect to the FPGA on-board the SoC board by using the `fpga` function. Use the processor hardware object `p` as an input to the `fpga` function.

```
f = fpga(p);
```

Version History
Introduced in R2020b

See Also

Classes
`hdlcoder.Board | hdlcoder.ReferenceDesign`

Objects
`hdlcoder.DUTPort | xilinxsoc | intelsoc`

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

xilinxsoc

Connection to processor on Xilinx SoC board

Description

The `xilinxsoc` object represents a connection from MATLAB to the processor on a Xilinx SoC board . Use this object to perform basic Linux shell operations, program your FPGA at the MATLAB command-line, and create an `fpga` object that connects to the FPGA through the processor on the Xilinx SoC board.

Creation

Syntax

```
p = xilinxsoc
p = xilinxsoc(ipaddress, username, password)
p = xilinxsoc(hostname, username, password)
```

Description

`p = xilinxsoc` creates a connection to the Xilinx SoC hardware board. This connection reuses the IP address or host name, user name, and password from the most recent connection to a specified Xilinx board. When you connect MATLAB to a Xilinx board for the first time, enter the host name or IP address, user name, and password of the Xilinx board.

`p = xilinxsoc(ipaddress, username, password)` creates a connection to the Xilinx board by using the IP address, user name, and password that you specify.

`p = xilinxsoc(hostname, username, password)` creates a connection to the Xilinx board by using the host name of the board, user name, and password that you specify.

Input Arguments

`ipaddress` — IP address of Xilinx SoC hardware board
character vector | string scalar

IP address of the Xilinx board connected to the network, specified as a character vector or string scalar. If no IP address is specified, the most recently specified IP address is used.

Example: `'192.168.1.101'`

`username` — User name to log into the Xilinx SoC board
character vector | string scalar

User name to log into the Xilinx board over SSH, specified as a character vector or string scalar. If no user name is specified, the most recently specified user name is used. The default user name is `"root"`.

Example: `'root'`

password — Password to log into Xilinx SoC board
character vector | string scalar

Password used to log into the Xilinx board over SSH, specified as a character vector or string scalar. If no password is specified, the most recently specified password is used. The default password is "root".

Example: 'root'

hostname — Host name of Xilinx SoC hardware board
character vector | string scalar

Host name of the Xilinx board connected to the network, specified as a character vector or string scalar.

Example: 'myhost.mydomain.com'

Output Arguments

p — Connection to Xilinx SoC board
xilinxsoc object

Connection from MATLAB to the processor on a Xilinx SoC hardware board, returned as a xilinxsoc object. You can use this connection to interact with the hardware board and attached devices, such as the FPGA on-board the SoC board.

Properties

DeviceAddress — IP address or host name of Xilinx SoC hardware board
character vector

IP address or host name of the Ethernet port on the Xilinx board.

This property is read-only and cannot be changed after you create the xilinxsoc object.

Example: '192.168.1.101'

Data Types: char

Username — User name to log into the Xilinx SoC board
character vector

User name to log into the Xilinx board over SSH. If no user name is specified, the most recently specified user name is used. The default user name is 'root'.

Example: 'root'

Data Types: char

Object Functions

Use the object functions to interact with your SoC device.

system	Run command in Linux shell on SoC board
getFile	Transfer file from SoC board to host computer
putFile	Transfer file from host computer to SoC board
deleteFile	Delete file on SoC board

dir	List directory contents on SoC board
programFPGA	Program FPGA and set corresponding device tree from processor on SoC board

Examples

Connect to Xilinx SoC Board from MATLAB

Create a hardware processor object to connect to a Xilinx SoC board. You can use the object to perform basic Linux shell operations on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Run the `uname` command in a Linux shell on the SoC board by using the `xilinxsoc` object function `system`.

```
system(p, 'uname')

ans =

    'Linux'
```

Program FPGA on Xilinx SoC Board

Create a hardware processor object to connect to a Xilinx SoC board. You can use the object to program your FPGA on-board the SoC board at the MATLAB command-line and to create an `fpga` object that connects to the FPGA through the processor on the SoC board.

Create a Xilinx processor hardware object and connect to the processor on-board the Xilinx SoC board.

```
p = xilinxsoc('192.168.1.101', 'root', 'root');
```

Program the FPGA and set the corresponding device tree from processor on the SoC board by using the `xilinxsoc` object function `programFPGA`.

```
programFPGA(p, "myBitstream.bit", "myDeviceTree.dtb");
```

For more information on device trees, see "Generate Device Tree for IP Core".

Connect to the FPGA on-board the SoC board by using the `fpga` function. Use the processor hardware object `p` as an input to the `fpga` function.

```
f = fpga(p);
```

Version History
Introduced in R2022a

See Also
`intelsoc` | `fpga`

intelsoc

Connection to processor on Intel SoC board

Description

The `intelsoc` object represents a connection from MATLAB to the processor on a Intel SoC board. Use this object to perform basic Linux shell operations, program your FPGA at the MATLAB command-line, and create an `fpga` object that connects to the FPGA through the processor on the Intel SoC board.

Creation

Syntax

```
p = intelsoc
p = intelsoc(ipaddress, username, password)
p = intelsoc(hostname, username, password)
```

Description

`p = intelsoc` creates a connection to the Intel SoC hardware board. This connection reuses the IP address or host name, user name, and password from the most recent connection to a specified Intel board. When you connect MATLAB to a Intel board for the first time, enter the host name or IP address, user name, and password of the Intel board.

`p = intelsoc(ipaddress, username, password)` creates a connection to the Intel board by using the IP address, user name, and password that you specify.

`p = intelsoc(hostname, username, password)` creates a connection to the Intel board by using the host name of the board, user name, and password that you specify.

Input Arguments

`ipaddress` — IP address of Intel SoC hardware board
character vector | string scalar

IP address of the Intel board connected to the network, specified as a character vector or string scalar. If no IP address is specified, the most recently specified IP address is used.

Example: `'192.168.1.101'`

`username` — User name to log into the Intel SoC board
character vector | string scalar

User name to log into the Intel board over SSH, specified as a character vector or string scalar. If no user name is specified, the most recently specified user name is used. The default user name is `"root"`.

Example: `'root'`

password — Password to log into Intel SoC board
character vector | string scalar

Password used to log into the Intel board over SSH, specified as a character vector or string scalar. If no password is specified, the most recently specified password is used. The default password is `"cyclonevsoc"`.

Example: `'cyclonevsoc'`

hostname — Host name of Intel SoC hardware board
character vector | string scalar

Host name of the Intel board connected to the network, specified as a character vector or string scalar.

Example: `'myhost.mydomain.com'`

Output Arguments

p — Connection to Intel SoC board
`intelsoc` object

Connection from MATLAB to the processor on an Intel SoC hardware board, returned as a `intelsoc` object. You can use this connection to interact with the hardware board and attached devices, such as the FPGA on-board the SoC board.

Properties

DeviceAddress — IP address or host name of Intel SoC hardware board
character vector

IP address or host name of the Ethernet port on the Intel board.

This property is read-only and cannot be changed after you create the `intelsoc` object.

Example: `'192.168.1.11'`

Data Types: `char`

Username — User name to log into the Intel SoC board
character vector

User name used to log into the Intel board over SSH. If no user name is specified, the most recently specified user name is used. The default user name is `'root'`.

Example: `'root'`

Data Types: `char`

Object Functions

Use the object functions to interact with your SoC device.

system	Run command in Linux shell on SoC board
getFile	Transfer file from SoC board to host computer
putFile	Transfer file from host computer to SoC board
deleteFile	Delete file on SoC board

dir   List directory contents on SoC board
programFPGA   Program FPGA and set corresponding device tree from processor on SoC board

Examples

Connect to Intel SoC Board from MATLAB

Create a hardware processor object to connect to an Intel SoC board. You can then use the object to perform basic Linux shell operations on the SoC board.

Create an Intel processor hardware object and connect to the processor on-board the Intel SoC board.

```
p = intelsoc('192.168.1.101', 'root', 'root');
```

Run the `uname` command in a Linux shell on the SoC board by using the `intelsoc` object function `system`.

```
system(p, 'uname')

ans =

    'Linux'
```

Program FPGA on Intel SoC Board

Create a hardware processor object to connect to an Intel SoC board. You can use the object to program your FPGA on-board the SoC board at the MATLAB command-line and to create an `fpga` object that connects to the FPGA through the processor on the SoC board.

Create an Intel processor hardware object and connect to the processor on-board the Intel SoC board.

```
p = intelsoc('192.168.1.101', 'root', 'root');
```

Program the FPGA and set the corresponding device tree from processor on the SoC board by using the `intelsoc` object function `programFPGA`.

```
programFPGA(p, "myBitstream.bit", "myDeviceTree.dtb");
```

For more information on device trees, see "Generate Device Tree for IP Core".

Connect to the FPGA on-board the SoC board by using the `fpga` function. Use the processor hardware object `p` as an input to the `fpga` function.

```
f = fpga(p);
```

Version History
Introduced in R2022a

See Also

xilinxsoc | fpga

hdlcoder.DUTPort

DUT port from an HDL Coder generated IP core, saved as an object array

Description

This object represents each DUT port name from an HDL Coder generated IP core. The object represents the ports of your DUT on the target hardware. When you generate an HDL IP core by running the `IP Core Generation` workflow, you map the ports to AXI4 slave or AXI4-Stream interfaces. The port object contains information about these DUT ports and the interfaces it is mapped to, based on the interface mapping information in the Target platform interface table of the "Set Target Interface" task. After you use the `mapPort` function, you can write to or read from the DUT ports by using the `writePort` and `.readPort` functions.

Creation

Syntax

`hPort = hdlcoder.DUTPort(Name)`

Description

`hPort = hdlcoder.DUTPort(Name)` creates a DUT port object as an object array, with additional properties specified by name-value pair arguments.

Properties

Name — Port name
string

Name of the input or output port of the DUT subsystem in your original model. When you run the `IP Core Generation` workflow, you obtain this information from the **Port Name** section of the Target platform interface table.

Example: `hPort = hdlcoder.DUTPort("h_in", ...)`

Data Types: `string` | `char`

Direction — Port direction
`"IN"` | `"OUT"` | `"INOUT"`

Direction of the DUT port, specified as `IN`, `OUT`, or `INOUT`. The port direction `INOUT` is supported for only the AXI4 and AXI4-Lite interface.

Example: `hPort = hdlcoder.DUTPort(..., "Direction", "OUT", ...)`

Data Types: `char`

IsComplex — Complex data on the port
`"true"` | `"false"`

Complex data on the DUT port, specified as `true` or `false`. HDL Coder supports complex data streaming on the AXI4-Stream interface. This setting is supported for only the AXI4-Stream interface.

Example: `hPort = hdlcoder.DUTPort(..., "IsComplex", "true", ...)`

Data Types: `char`

DataType — Port data type
`numeric type` | `numerictype`

Data type of the DUT port that is mapped to the AXI interface, specified as a MATLAB numeric type such as `uint32` or a `numerictype` object. When you run the `IP Core Generation` workflow, you obtain this information from the **Data Type** section of the Target platform interface table.

Example: `hPort = hdlcoder.DUTPort(..., "DataType", numerictype(1,16,10), ... )`

Data Types: `uint8` | `uint16` | `uint32` | `uint64`

Dimension — Port dimensions
`integer array`

Dimensions of the DUT port that is mapped to the AXI interface, specified as an integer array or any other data type. The dimensions depend on whether the port mapped is a scalar or vector. For a scalar port, the `Dimension` is `[1 1]`, and for an N-dimensional vector port, it is `[1 N]`. When you run the `IP Core Generation` workflow, you obtain this information from the **Data Type** section of the Target platform interface table.

Example: `hPort = hdlcoder.DUTPort(..., "Dimension", [1 6], ...)`

Data Types: `int8` | `int16` | `int32` | `int64` | `single` | `double`

IOInterface — Target platform interface
`"AXI4"` | `"AXI4-Lite"` | `"AXI4-Stream Master"` | `"AXI4-Stream Slave"` | `string array` | `character array`

Target platform interface that the DUT port is mapped to, specified as a string or character array. When you run the `IP Core Generation` workflow, you obtain this information from the **Target Platform Interfaces** section of the Target platform interface table.

Example: `hPort = hdlcoder.DUTPort(..., "IOInterface", "AXI4-Lite")`

Data Types: `string` | `char`

IOInterfaceMapping — Target interface mapping
`character array` | `string array` | `numeric type`

Target platform interface mapping information, specified as a character array, string array, or numeric type.

Example: `hPort = hdlcoder.DUTPort(..., "IOInterfaceMapping", "0x100")`

Data Types: `string` | `char`

Examples

Create a DUT Port Object Mapped to AXI4 Slave Interface

Create an `fpga` object to connect to a target device and then use `hdlcoder.DUTPort` object to specify the DUT port.

Create an `fpga` object for the target device.

```
hFPGA = fpga("Xilinx")


hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4 slave interface to the `hFPGA` object by using the `addAXI4SlaveInterface` function.

```
%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000);
```

Create a `hdlcoder.DUTPort` object for the AXI4-Lite Interface. After you create the object, you can map the port to the IO interface by using the `mapPort` function.

```
% ...
hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");

mapPort(hFPGA, hPort_h_in1);


hPort_h_in1 =

  DUTPort with properties:

                  Name: "h_in1"
             Direction: IN
              DataType: [1×1 embedded.numerictype]
             Dimension: [1 1]
           IOInterface: "AXI4-Lite"
    IOInterfaceMapping: "0x100"
```

Create a DUT Port Object Mapped to AXI4-Stream Interface

Create an `fpga` object to connect to a target device and then use `hdlcoder.DUTPort` object to specify the DUT port.

Create an `fpga` object.

```
hFPGA = fpga("Xilinx")

hFPGA =

  fpga with properties:

        Vendor: "Xilinx"
    Interfaces: [0x0 fpgaio.interface.InterfaceBase]
```

Add the AXI4-Stream interface to the `hFPGA` object by using the `addAXI4StreamInterface` function.

```
%% AXI4-Stream
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
        "ReadEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadFrameLength", 1024);
```

Create a `hdlcoder.DUTPort` object for an AXI4-Stream Interface. After you create the object, you can map the port to the IO interface by using the `mapPort` function.

```
hPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream")

hPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,20), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream")

mapPort(hFPGA, [hPort_x_in_data, hPort_y_out_data]);

 hPort_x_in_data =

  DUTPort with properties:

                Name: "x_in_data"
           Direction: IN
            DataType: [1×1 embedded.numerictype]
           Dimension: [1 1]
         IOInterface: "AXI4-Stream"
```

```
    IOInterfaceMapping: ""

hPort_y_out_data =

  DUTPort with properties:

                Name: "y_out_data"
           Direction: OUT
            DataType: [1×1 embedded.numerictype]
           Dimension: [1 1]
         IOInterface: "AXI4-Stream"
    IOInterfaceMapping: ""
```

Version History
Introduced in R2020b

See Also

Objects
fpga

Functions
mapPort | writePort | readPort

Topics
"Generate Host Interface Script to Probe and Rapidly Prototype HDL IP Core"
"Create Host Interface Script to Control and Rapidly Prototype HDL IP Core"

hdl.npufun

Apply neighborhood processing and element-wise operations to an incoming image or matrix for frame-to-sample conversion

Syntax

```
output_data = hdl.npufun(kernelFun,kernelSize,input_data)
output_data = hdl.npufun(kernelFun,kernelSize,input_data,Name,Value)
```

Description

`output_data = hdl.npufun(kernelFun,kernelSize,input_data)` applies the `kernelFun` function to each sliding window of the input data, `input_data`, using a sliding window determined by `kernelSize`. The function outputs one argument for each output argument in the function `kernelFun`.

Use `hdl.npufun` to process neighborhood and element-wise operations performed on frame-based inputs, such as filtering with a kernel.

Note `hdl.npufun` is a utility function that applies a neighborhood processing operation from another function to incoming data.

`output_data = hdl.npufun(kernelFun,kernelSize,input_data,Name,Value)` applies the function using optional name-value arguments.

Examples

Apply Image Processing Algorithm

Apply image blurring to the input image A by using `hdl.npufun`.

Create the blurring kernel function, `blurringKernel`, that takes a sample of the image A as the input `in`, blurs it with an average filter, and outputs the result `y`.

```
function y = blurringKernel(in)
  y = sum(in(:),'native')/9;
end
```

Apply image blurring to the input image A with a kernel window for the blurring algorithm that is a 3-by-3 matrix.

```
A_blurred = hdl.npufun(@blurringKernel, [3 3], A);
```

Apply Neighborhood Processing With Custom Boundary Constant

Apply image blurring to an input image with a custom boundary constant.

Create the blurring kernel function, `blurringKernel`, that takes a sample of the image A as the input `in`, blurs it with an average filter, and outputs the result `y`.

```
function y = blurringKernel(in)
  y = sum(in(:),'native')/9;
end
```

Apply image blurring to the input image A with a custom boundary constant of five and a kernel window that is a 3-by-3 matrix.

```
A_blurred = hdl.npufun(@blurringKernel, [3 3], A, 'BoundaryConstant', 5);
```

Apply Neighborhood Processing with Custom Coefficient for Input Data

Apply a neighborhood processing algorithm to an input array with a custom coefficient that is used in the kernel function.

Create the kernel function, `kernelSum`, that takes a sample of the input array A as the input `in`, multiples it with the input `coeff`, and sums the output.

```
function out = kernelSum(in, coeff)
  out = hdl.treesum(in .* coeff, 'all');
end
```

Apply the summation algorithm, `kernelSum`, to the streamed input from the input array A. Supply `kernelSum` with the input `coeff`.

```
A_summed = hdl.npufun(@kernelSum, kernelSize, A, 'KernelArg', coeff);
```

The argument order of `hdl.npufun` for the streamed input data A and non-streamed input data `coeff` must match the input argument order of the kernel function `kernelSum`. In this example the streamed input data `in` is first and the non-streamed input data `coeff` is second. If, for example, the `kernelSum` function has the syntax `kernelSum(coeff, in)`, then `hdl.npufun` must match the argument order and have the syntax `hdl.npufun(@kernelSum, kernelSize, 'KernelArg', coeff, A)`.

Input Arguments

kernelFun — Kernel operator
function handle

Kernel operator, specified as a function handle. `hdl.npufun` applies the `kernelFun` function to each sliding window of `kernelSize` of the input data. The function calls `kernelFun` for each `kernelSize` window of the input and computes an element of the output.

Example: `@blurringKernel`

kernelSize — Size of kernel
2-D positive integer array

Size of the kernel or sliding window for the neighborhood processing algorithm in `kernelFun`, specified as a 2-D positive integer array.

Example: `[3,3]`

input_data — Input data for neighborhood processing algorithm
2-D numeric array

Input data for the neighborhood processing algorithm to convert in the frame-to-sample conversion, specified as a 2-D numeric array. Frame-to-sample conversion converts the input signal you specify for `input_data` from a frame input to single values streamed as sampled inputs. The `hdl.npufun` function applies the `kernelFun` function to each element of the streaming input data `input_data`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `'BoundaryConstant', 5`

BoundaryConstant — Boundary constant around streamed input data
0 (default) | integer scalar

Boundary constant to apply around the streamed input data, specified as an integer. Custom boundary constant that is applied around the streamed input array boundary. You can specify only one value for `BoundaryConstant`.

KernelArg — Kernel input value used in neighborhood processing algorithm
numeric scalar

Kernel input value used in the neighborhood processing algorithm in the `kernelFun` function, specified as a numeric scalar. This value can be any input that is not streamed from frame to samples by the frame-to-sample conversion. You can pass multiple non-streamed kernel inputs by using this name-value argument more than once. When you specify `KernelArg`, ensure the input arguments in the `hdl.npufun` function are in the same order as the streamed and non-streamed input arguments in the `kernelFun` function.

Example: `hdl.npufun(@kernelFun, kernelSize, input_data, 'KernelArg', 3, 'KernelArg', 7.8);`

Output Arguments

output_data — Output of kernel operation
2-D numeric array

Output of the kernel operation in the `kernelFun` function, returned as a 2-D numeric array. The size of `output_data` is the size of `input_data`.

You can specify multiple output arguments for `hdl.npufun` if the `kernelFun` function has multiple output arguments. For example, you can specify multiple outputs for `hdl.npufun` with syntax like `[a, b, c] = hdl.npufun(@multiOutKernel, [3 3], in)`, where the kernel has the function syntax `[a_pix, b_pix, c_pix] = multiOutKernel(in_window)`.

Version History
Introduced in R2022b

R2022b: Renamed NonSampleInput argument

The name-value input argument `NonSampleInput` is now called `KernelArg`. The behavior remains the same.

See Also

`hdl.iteratorfun`

Topics
"HDL Code Generation from Frame-Based Algorithms"

hdl.iteratorfun

Apply iterative operation to an incoming image or matrix for frame-to-sample conversion

Syntax

```
out = hdl.iteratorfun(iterFun,I,outputData)
out = hdl.iteratorfun(iterFun,I,outputData, KernelArg)
```

Description

`out = hdl.iteratorfun(iterFun,I,outputData)` applies the iterative operation, `iterFun`, to each element of the input data `I`. `outputData` is the initial value of `out` and stores the output of each iteration of `iterFun`. The final value of `outputData` is the returned function output `out`.

`hdl.iteratorfun` supports iterative operations in `iterFun`, such as looping over arrays to produce a single output to an incoming image or matrix for histogram equalization and to compute statistics such as `min` and `max`.

Note `hdl.iteratorfun` is a utility function that applies a iterative operation from another function to incoming data.

`out = hdl.iteratorfun(iterFun,I,outputData, KernelArg)` passes the non-streamed kernel input `KernelArg` to `iterFun`.

Examples

Compute Histogram

Compute a histogram from matrix `data`.

Create the histogram kernel function, `hist_kernel_fcn`, that takes a histogram of the input array `data` by binning each sample from the input array and storing the output in `hist`. The element index input `idx` must be defined as an input argument to the iterator function when using `hdl.iteratorfun`, even if `idx` is not used in the iterator function.

```
function hist = hist_kernel_fcn(sample, hist, idx)
  hist(sample +1) = hist(sample +1) + 1;
end
```

Initialize the size of the output data `hist_init`, which serves as the initialized output for the `hist_kernel_fcn` function, `hist`.

```
hist_init = zeros(1, 256, 'uint32');
```

Generate histogram data and apply the histogram kernel function, `hist_kernel_fcn` to the streamed input data `data`.

```
out = hdl.iteratorfun(@hist_kernel_fcn, data, hist_init);
```

Compute Maximum and Maximum Index of Input Data

Compute the maximum value of an input vector `data` and find the index of that maximum value.

Create an input vector `data` by using the `randi` function.

```
data = randi(100,1,10);
```

Create the iterator function kernel, `max_fun`, that finds the first maximum value and the maximum value index of the input array.

```
function max_val= max_fun(sample, max_val, idx)
  if sample > max_val(1)
    max_val(1) = sample;
    max_val(2) = idx;
  end
end
```

Initialize the size of the output data `max_init`, which serves as the initialized output for the `max_fun` function, `max_val`.

```
max_init = [-inf 0];
```

Calculate the first maximum value and the corresponding index of the input array `data` by applying the max function, `max_fun`, to the streamed input from the input array `data`.

```
out = hdl.iteratorfun(@max_fun, data, max_init);
```

Threshold Input Data

Count values above a certain threshold from an input array `data`.

Create the iterator function kernel, `count_kernel`, that counts each element of the input array `data` that is greater than the threshold value specified by the input `threshold`.

```
function count = count_kernel(sample, count, idx, threshold)
  if sample > threshold
    count = count + 1;
  end
end
```

Initialize the size of the output data `count_init`, which serves as the initialized output for the `count_kernel` function, `count`.

```
count_init = 0;
```

Count the values from the input `data` that are greater than the input value `threshold` by applying the count function, `count_kernel` to the streamed input from the input array `data`.

```
out = hdl.iteratorfun(@count_kernel, data, count_init, threshold);
```

Input Arguments

`iterFun` — Iterator function
function handle

Iterator function, specified as a function handle. The function you specify for `iterFun` must have the syntax `out_data = iterFun(element,outputData,idx)`. The `hdl.iteratorfun` function passes to `iterFun`, in the specified input order, an element from the input `I`, the initialized output data `outputData`, and the element index `idx` in the input array `I`. You must specify each of these input arguments for `iterFun`, regardless of whether you use them in the iterative operation.

`I` — Input data for the iterative operation
2-D numeric array

Input data for the iterative operation, specified as a 2-D numeric array. The `hdl.iteratorfun` function applies the `iterFun` function to each element of the input data `I`. If this input is a frame-based value, then frame-to-sample conversion converts the input signal you specify for `I` from a frame input to single values streamed as sampled inputs.

`outputData` — Initial value for the output
scalar | vector | 2-D matrix

Initial value for the output of `hdl.iteratorfun`, specified as a scalar, vector, or 2-D matrix. You must initialize the size of `outputData` to match the size of the output of `hdl.iteratorfun`, `out`, and the size of the output of the `iterFun` function.

`KernelArg` — Additional kernal input data
scalar | vector | 2-D matrix

Additional kernel input data, specified as a scalar, vector, or 2-D matrix. This value can be any input that is not streamed from frame to samples by the frame-to-sample conversion. For example, you can use `KernelArg` in `iterFun` for to provide a threshold for the input data `I`.

Output Arguments

`out` — Output of the iterative operation
scalar | vector | 2-D matrix

Output of the iterative operation in `iterFun`, returned as a scalar, vector, or 2-D matrix. The size of `out` must match the size of `outputData`.

Version History
Introduced in R2022b

R2022b: Renamed NonSampleInput argument

The input argument `NonSampleInput` is now called `KernelArg`. The behavior remains the same.

See Also

`hdl.npufun`

Topics
"HDL Code Generation from Frame-Based Algorithms"

Block Reference Page Examples

Convert Input Vector of Bits to Integer

This example shows how to use the Bits to Word block in a Simulink® Model that converts vector of bits to an unsigned integer. You also generate HDL code for the model.

Simulate Model

Load and open the `hdlcoder_bits_to_word` model. The `DUT` subsystem contains Bits to Word block that converts a vector of bits to an unsigned integer.

```
load_system("hdlcoder_bits_to_word");
open_system("hdlcoder_bits_to_word/DUT");
```
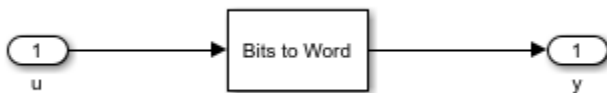

Simulate the `hdlcoder_bits_to_word` model. The input is a vector of length 5. The block treats first element of the input vector as the least significant bit (LSB) of the output. The output of the block is an unsigned integer with the `ufix5` type.

```
sim('hdlcoder_bits_to_word');
```

Generate HDL code

You can generate the HDL code for a model by using HDL Coder™. To generate HDL code for `DUT` subsystem, run this `makehdl` command:

```
makehdl("hdlcoder_bits_to_word/DUT")
```

Convert Integer to Vector of Bits

This example shows how to implement the Word to Bits block in a Simulink® Model that converts an integer to a vector of bits. You also generate HDL code for the model.

Simulate Model

Load and open the `hdlcoder_word_to_bits` model. The `DUT` subsystem contains the Word to Bits block, which converts an integer to a vector of bits.

```
load_system("hdlcoder_word_to_bits");
open_system("hdlcoder_word_to_bits/DUT");
```
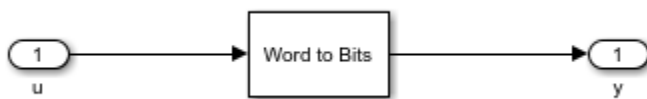

Simulate the `hdlcoder_word_to_bits` model. The input is an integer with a value of 72. The block uses maximum word length of 8 to determine the output vector size. The output is a binary representation of 72. The block treats first element of the output vector as the least significant bit (LSB) of the output.

```
sim('hdlcoder_word_to_bits');
```

Generate HDL code

You can generate the HDL code for a model by using HDL Coder™. To generate HDL code for `DUT` subsystem, run this `makehdl` command:

```
makehdl("hdlcoder_word_to_bits/DUT")
```

Simulate and Generate HDL Code for Tapped Delay Enabled Resettable Synchronous Block

This example shows how to implement the Tapped Delay Enabled Resettable Synchronous block in a Simulink® model that a delays scalar signal multiple sample periods and output all delayed versions when external Enable signal is true and external Reset signal is false. You also generate HDL code for the model.

Open and Load Model

Load and open the `TappedDelayEnableResetModel` model. The `DUT` subsystem contains a Tapped Delay Enabled Resettable Synchronous block. The **Number of delay** parameter corresponds to the number of delayed versions of the input signal the block returns.

```
load_system("TappedDelayEnableResetModel");
open_system("TappedDelayEnableResetModel/DUT");
```
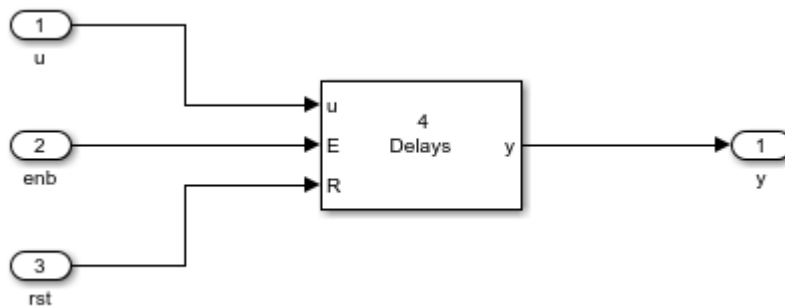

Simulate Design

Simulate the `TappedDelayEnableResetModel` model. The output of the block is a vector. The block returns four signals, the first delayed by four sample periods, the second delayed by three, and so on.

```
sim('TappedDelayEnableResetModel');
```

Generate HDL code

You can generate the HDL code for a model by using HDL Coder™. Generate the HDL code for `DUT` subsystem using the `makehdl` function.

```
makehdl("TappedDelayEnableResetModel/DUT")
```