



Include FOR and GENERATE loops in the generated VHDL code.

Tip

- If you are using an electronic design automation (EDA) tool that does not support GENERATE loops, select this option to omit loops from your generated VHDL code.
- Setting this option does not affect results obtained from simulation or synthesis of generated VHDL code.

Dependency

This option is enabled when the target language (specified by the **Language** option) is VHDL.

Command-Line Information

Property: LoopUnrolling

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

Generate parameterized HDL code from masked subsystem

Generate reusable HDL code for subsystems with the same tunable mask parameters, but with different values.

Settings

Default: Off



Generate one reusable HDL file for multiple masked subsystems with different values for the mask parameters. HDL Coder automatically detects subsystems with tunable mask parameters that are sharable.

Inside the subsystem, you can use the mask parameter only in the following blocks and parameters.

Block	Parameter	Limitation
Constant	Constant value on the Main tab of the dialog box	None
Gain	Gain on the Main tab of the dialog box	Parameter data type must be the same for all Gain blocks.



Generate a separate HDL file for each masked subsystem.

Command-Line Information

Property: MaskParameterAsGeneric

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

See Also

"Generate Reusable Code for Atomic Subsystems" on page 27-17

Enumerated Type Encoding Scheme

Specify the encoding scheme to represent enumeration types in the generated HDL code.

Settings

Default: default

Use `default`, `onehot`, `twohot`, or `binary` encoding scheme to represent enumerated types in the generated HDL code.

default

The code generator uses decimal encoding in Verilog and VHDL-native enumerated types in VHDL. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 2'd0,
is_Chart_IN_s_rx = 2'd1,
is_Chart_IN_s_wait_0 = 2'd2,
is_Chart_IN_s_wait_tb = 2'd3;
```

onehot

The code generator uses a one-hot encoding scheme where a single bit is high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0001,
is_Chart_IN_s_rx = 4'b0010,
is_Chart_IN_s_wait_0 = 4'b0100,
is_Chart_IN_s_wait_tb = 4'b1000;
```

This encoding scheme does not support more than 64 enumeration values or number of states.

twohot

The code generator uses a two-hot encoding scheme where two bits are high to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 4'b0011,
```

```
is_Chart_IN_s_rx = 4'b0101,
is_Chart_IN_s_wait_0 = 4'b0110,
is_Chart_IN_s_wait_tb = 4'b1001;
```

binary

The code generator uses a binary encoding scheme to represent each enumeration value. This example shows the verilog code snippet of this encoding scheme for a Stateflow Chart that has four states.

```
parameter
is_Chart_IN_s_idle = 2'b00,
is_Chart_IN_s_rx = 2'b01,
is_Chart_IN_s_wait_0 = 2'b10,
is_Chart_IN_s_wait_tb = 2'b11;
```

In VHDL, the generated code uses **CONSTANT** types to encode nondefault enumeration values in the generated code. For example, this code snippet shows the generated VHDL code when you use the two-hot state encoding for a Stateflow Chart that has four states.

```
PACKAGE s_pkg IS
-- Constants
-- Two-hot encoded enumeration values for type state_type_is_Chart
CONSTANT IN_s_idle      : std_logic_vector(3 DOWNTO 0) :=
"0011";
CONSTANT IN_s_rx        : std_logic_vector(3 DOWNTO 0) :=
"0101";
CONSTANT IN_s_wait_0    : std_logic_vector(3 DOWNTO 0) :=
"0110";
CONSTANT IN_s_wait_tb   : std_logic_vector(3 DOWNTO 0) :=
"1001";

END s_pkg;
```

Command-Line Information

Property: EnumEncodingScheme

Type: character vector

Value: 'default' | 'onehot' | 'twohot' | 'binary'

Default: 'default'

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

Timing Controller Settings

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

Optimize timing controller

Optimize timing controller entity for speed and code size by implementing separate counters per rate.

Settings

Default: On



On

HDL Coder generates multiple counters (one counter for each rate in the model) in the timing controller code. The benefit of this optimization is that it generates faster logic, and the size of the generated code is usually much smaller.



Off

The coder generates a timing controller that uses one counter to generate all rates in the model.

Tip

A timing controller code file is generated if required by the design, for example:

- When code is generated for a multirate model
- When a cascade block implementation for certain blocks is specified

This file contains a module defining timing signals (clock, reset, external clock enable inputs and clock enable output) in a separate entity or module. In a multirate model, the timing controller entity generates the required rates from a single master clock using one or more counters and multiple clock enables.

The timing controller name derives from the name of the subsystem that is selected for code generation (the DUT), and the current value of the property `TimingControllerPostfix`. For example, if the name of your DUT is `my_test`, in the default case the coder adds the `TimingControllerPostfix_tc` to form the timing controller name `my_test_tc`.

Command-Line Information

Property: `OptimizeTimingController`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

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

Timing controller architecture

Specify whether to generate a reset for the timing controller.

Settings**Default:** default**resettable**

Generate a reset for the timing controller. If you select this option, the **Clock inputs** value must be Single.

default

Do not generate a reset for the timing controller.

Command-Line Information**Property:** TimingControllerArch**Type:** character vector**Value:** 'resettable' | 'default'**Default:** 'default'

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

File Comment Customization Parameters

This page describes parameters that reside in the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box.

Custom File Header Comment

Specify a custom file header comment in the generated HDL code.

Default: ''

With **Custom File Header Comment**, you can enter custom comments to appear as header in the generated HDL file for your design.

For example, you can specify arguments such as title, author, modified date, and so on.

```
// =====
// Title      : <%Title%>
// Project    : <%Project%>
// Author     : <%Author%>
//
// Revision   : $Revision$
// Date Modified : $Date$
// =====
```

Command-Line Information

Property: CustomFileHeaderComment

Type: character vector

Default: ''

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

Custom File Footer Comment

Specify a custom file header comment in the generated HDL code.

Default: ''

With **Custom File Footer Comment**, you can enter custom comments to appear as footer in the generated HDL file for your design.

For example, you can specify arguments such as revision, generated log file, revision number, and so on.

```
//=====
// xxxxxx
//=====
// $Log$
// Revision 1.2 2009/12/14 04:38:51 sxxxxxx
// Initial revision
//=====
//=====
```

Command-Line Information

Property: CustomFileFooterComment

Type: character vector

Default: ''

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

Choose Coding Standard and Report Option Parameters

This section contains parameters in the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to generate HDL code that adheres to the guidelines recommended by Industry coding standards.

HDL coding standard

Specify whether to enable the Industry coding standard guidelines that the generated HDL code must conform to.

Settings

Default: None

None

Generate generic synthesizable HDL code. The generated code need not conform with the Industry standard guidelines.

Industry

Generate synthesizable HDL code that follows the industry standard rules supported by HDL Coder. When you specify the **Industry** setting, the code generator enables the **Report options** check box and rules that you can customize in the **Coding Standards** tab.

When you specify the **Industry** setting and generate code, HDL Coder generates a standards compliance report. The report displays errors, warnings, messages, and lists the corresponding rules. To filter the report such that the passing rules do not appear, clear the **Report options** check box.

Command-Line Information

Property: `HDLCodingStandard`

Type: character vector

Value: `'None'` | `'Industry'`

Default: `'None'`

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

For example, you can enable the Industry standard guidelines compliance for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HDLCodingStandard','Industry')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry')
```

See Also

- `makehdl`
- “HDL Coding Standards” on page 26-4

- HDL Coding Standard Customization Properties

Do not show passing rules in coding standard report

Specify whether to filter the coding standard report such that the passing rules do not appear. By default, the report displays pass, errors, warnings, messages, and lists the corresponding rules.

Settings

Default: Off



On

Show only rules with errors or warnings. The code generator filters out messages and passing rules from the report.



Off

Show all rules in the report including the messages and passing rules.

Dependency

To clear the **Report options** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **ShowPassingRules** property of the HDL coding standard customization object.

For example, to omit passing rules from the report, enter:

```
cso.ShowPassingRules.enable = false;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**
- “HDL Coding Standards” on page 26-4
- **hdlcoder.CodingStandard**
- HDL Coding Standard Customization

Basic Coding Practices Parameters

These parameters belong to the **Basic coding rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize basic coding rules that are specified by the Industry standard guidelines. These rules correspond to naming conventions that your design uses.

Check for duplicate names

Specify whether to check for duplicate names in the design. This check corresponds to CGSL-1.A.A.5 of the Industry standard guidelines.

Settings

Default: On



Check for duplicate names.



Do not check for duplicate names.

Dependency

To clear the **Check for duplicate names** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **DetectDuplicateNamesCheck** property of the HDL coding standard customization object.

For example, to disable the check for duplicate names, enter:

```
cso.DetectDuplicateNamesCheck.enable = false;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**
- “Basic Coding Practices” on page 26-9
- **hdlcoder.CodingStandard**

- HDL Coding Standard Customization

Check for HDL keywords in design names

Specify whether to check for HDL keywords in design names. This check corresponds to CGSL-1.A.A.3 of the Industry standard guidelines.

Settings

Default: On



On

Check for HDL keywords in design names.



Off

Do not check for HDL keywords in design names.

Dependency

To clear the **Check for HDL keywords in design names** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **HDLKeywords** property of the HDL coding standard customization object.

For example, to disable the check for HDL keywords in design names, enter:

```
cso.HDLKeywords.enable = false;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “Basic Coding Practices” on page 26-9
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check module, instance, entity name length

Specify whether to check module, instance, and entity name length. This check corresponds to CGSL-1.A.C.3 of the Industry standard guidelines.

Settings

Default: On

On

Check module, instance, and entity name length.

Minimum

Minimum name length, specified as a positive integer. The default is 2.

Maximum

Maximum name length, specified as a positive integer. The default is 32.

Off

Do not check module, instance, and entity name length.

Dependency

To clear the **Check module, instance, entity name length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **ModuleInstanceEntityNameLength** property of the HDL coding standard customization object.

For example, to enable the check for module, instance, and entity name length, with 5 as the minimum length and 30 as the maximum length, enter:

```
cso.ModuleInstanceEntityNameLength.enable = true;
cso.ModuleInstanceEntityNameLength.length = [5 30];
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “Basic Coding Practices” on page 26-9
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check signal, port, and parameter name length

Specify whether to check signal, port, and parameter name length. This check corresponds to CGSL-1.A.B.1 of the Industry standard guidelines.

Settings

Default: On



On
Check signal, port, and parameter name length.

Minimum

Minimum name length, specified as a positive integer. The default is 2.

Maximum

Maximum name length, specified as a positive integer. The default is 40.



Off
Do not check signal, port, and parameter name length.

Dependency

To clear the **Check signal, port, and parameter name length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **SignalPortParamNameLength** property of the HDL coding standard customization object.

For example, to enable the check for signal, port, and parameter name length, with 5 as the minimum length and 30 as the maximum length, enter:

```
cso.SignalPortParamNameLength.enable = true;
cso.SignalPortParamNameLength.length = [5 30];
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**
- “Basic Coding Practices” on page 26-9
- **hdlcoder.CodingStandard**

- HDL Coding Standard Customization

RTL Description Rules for clock enables and resets Parameters

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules for clock enable and reset signals that are specified by the Industry standard guidelines.

Check for clock enable signals

Specify whether to check for clock enable signals in the generated code. This check corresponds to CGSL-2.C.C.4 of the Industry standard guidelines.

Settings

Default: Off



On

Minimize clock enables during code generation, then check for clock enable signals in the generated code.



Off

Do not check for clock enable signals in the generated code.

Dependency

To select the **Check for clock enable signals** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **MinimizeClockEnableCheck** property of the HDL coding standard customization object.

For example, to minimize clock enables and check for clock enable signals in the generated code, enter:

```
cso.MinimizeClockEnableCheck.enable = true;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**

- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Detect usage of reset signals

Specify whether to check for reset signals in the generated code. This check corresponds to CGSL-2.C.C.5 of the Industry standard guidelines.

Settings

Default: Off



On

Minimize reset signals in the generated code, then check for reset signals after code generation.



Off

Do not check for reset signals in the generated code.

Dependency

To select the **Detect usage of reset signals** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the `RemoveResetCheck` property of the HDL coding standard customization object.

For example, to check for reset signals, enter:

```
cso.RemoveResetCheck.enable = true;
```

- 3 Set the `HDLCodingStandardCustomizations` property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is `sfir_fixed/symmetric_fir`, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Detect usage of asynchronous reset signals

Specify whether to check for asynchronous reset signals in the generated code. This check corresponds to CGSL-2.C.C.6 of the Industry standard guidelines.

Settings

Default: Off

On

Check for asynchronous reset signals in the generated code.

Off

Do not check for asynchronous reset signals in the generated code.

Dependency

To clear the **Detect usage of asynchronous reset signals** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **AsynchronousResetCheck** property of the HDL coding standard customization object.

For example, to minimize use of variables, enter:

```
cso.AsynchronousResetCheck.enable = true;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

RTL Description Rules for Conditional Parameters

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules for conditional and if-else statements that are specified by the Industry standard guidelines.

Check for conditional statements in processes

Specify whether to check for length of conditional statements that are described separately within a process. This check corresponds to CGSL-2.F.B.1 of the Industry standard guidelines.

Settings

Default: On

On

Check for length of conditional statements in a process. The default length is 1.

Off

Do not check for length of conditional statements in a process.

Dependency

To clear the **Check for conditional statements in processes** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **ConditionalRegionCheck** property of the HDL coding standard customization object.

For example, to check for four conditional statements in a process, enter:

```
cso.ConditionalRegionCheck.enable = true;
cso.ConditionalRegionCheck.length = 4;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`

- HDL Coding Standard Customization

Check if-else statement chain length

Specify whether to check if-else statement chain length. This check corresponds to CGSL-2.G.C.1c of the Industry standard guidelines.

Settings

Default: On



On

Check if-else statement chain length.

Length

Maximum if-else statement chain length, specified as a positive integer. The default is 7.



Off

Do not check if-else statement chain length.

Dependency

To clear the **Check if-else statement chain length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **IfElseChain** property of the HDL coding standard customization object.

For example, to check for if-else statement chains with length greater than 5, enter:

```
cso.IfElseChain.enable = true;
cso.IfElseChain.length = 5;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Check if-else statement nesting depth

Specify whether to check if-else statement nesting depth. This check corresponds to CGSL-2.G.C.1a of the Industry standard guidelines.

Settings

Default: On

On

Check if-else statement nesting depth.

Depth

Maximum if-else statement nesting depth, specified as a positive integer. The default is 3.

Off

Do not check if-else statement nesting depth.

Dependency

To clear the **Check if-else statement nesting depth** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **IfElseNesting** property of the HDL coding standard customization object.

For example, to enable the check for if-else statement nesting depth with a maximum depth of 5, enter:

```
cso.IfElseNesting.enable = true;
cso.IfElseNesting.depth = 5;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Description Techniques” on page 26-18
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Other RTL Description Rule Parameters

These parameters belong to the **RTL description rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to customize RTL description rules of the Industry standard guidelines. These rules pertain to checking the multiplier width, whether to minimize use of variables, and initial statements to provide initial value for RAMs.

Minimize use of variables

Specify whether to minimize use of variables. This check corresponds to CGSL-2.G of the Industry standard guidelines.

Settings

Default: Off



On
Minimize use of variables.



Off
Do not minimize use of variables.

Dependency

To select the **Minimize use of variables** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **MinimizeVariableUsage** property of the HDL coding standard customization object.

For example, to minimize use of variables, enter:

```
cso.MinimizeVariableUsage.enable = true;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**
- “RTL Description Techniques” on page 26-18
- **hdlcoder.CodingStandard**

- HDL Coding Standard Customization

Check for initial statements that set RAM initial values

Specify whether to check for initial statements that set RAM initial values. This check corresponds to CGSL-2.C.D.1 of the Industry standard guidelines.

Settings

Default: On



On

Check for initial statements that set RAM initial values



Off

Do not check for initial statements that set RAM initial values.

Dependency

To clear the **Check for initial statements that set RAM initial values** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **InitialStatements** property of the HDL coding standard customization object.

For example, to disable the check for initial statements that set RAM initial values, enter:

```
cso.InitialStatements.enable = false;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- **makehdl**
- “RTL Description Techniques” on page 26-18
- **hdlcoder.CodingStandard**
- HDL Coding Standard Customization

Check multiplier width

Specify whether to check multiplier bit width. This check corresponds to CGSL-2.J.F.5 of the Industry standard guidelines.

Settings

Default: On



On
Check multiplier width.

Maximum

Maximum multiplier bit width, specified as a positive integer. The default is 16.



Off
Do not check multiplier width.

Dependency

To clear the **Check multiplier width** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **MultiplierBitWidth** property of the HDL coding standard customization object.

For example, to enable the check for multiplier width with a maximum bit width of 32, enter:

```
cso.MultiplierBitWidth.enable = true;
cso.MultiplierBitWidth.width = 32;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard', 'Industry', ...
'HDLCodingStandardCustomizations', cso);
```

See Also

- **makehdl**
- “RTL Description Techniques” on page 26-18
- **hdlcoder.CodingStandard**
- HDL Coding Standard Customization

RTL Design Rule Parameters

This section contains configuration parameters in the **RTL design rules** section of the **Coding standards** tab of the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. Use these parameters to check for presence of non-integer constants and the line wrap length in the generated HDL code.

Check for non-integer constants

Specify whether to check for non-integer constants. This check corresponds to CGSL-3.B.D.1 of the Industry standard guidelines.

Settings

Default: On



On

Check for non-integer constants.



Off

Do not check for non-integer constants.

Dependency

To clear the **Check for non-integer constants** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **NonIntegerTypes** property of the HDL coding standard customization object.

For example, to disable the check for non-integer constants, enter:

```
cso.NonIntegerTypes.enable = false;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Design Rule Parameters” on page 17-80
- `hdlcoder.CodingStandard`

- HDL Coding Standard Customization

Check line length

Specify whether to check line lengths in the generated HDL code. This check corresponds to CGSL-3.A.D.5 of the Industry standard guidelines.

Settings

Default: On



On
Check line length.

Maximum

Maximum number of characters in a line, specified as a positive integer. The default is 110.



Off
Do not check line length.

Dependency

To clear the **Check line length** check box, set the **HDL coding standard** parameter to **Industry**.

Command-Line Information

To set this property:

- 1 Create an HDL coding standard customization object.

```
cso = hdlcoder.CodingStandard('Industry');
```

- 2 Set the **LineLength** property of the HDL coding standard customization object.

For example, to enable the check line length with a maximum character length of 80, enter:

```
cso.HDLKeywordsLineLength.enable = true;
cso.HDLKeywordsLineLength.length = 80;
```

- 3 Set the **HDLCodingStandardCustomizations** property to the HDL coding standard customization object, specify the coding standard, and generate code.

For example, if your DUT is **sfir_fixed/symmetric_fir**, enter:

```
makehdl('sfir_fixed/symmetric_fir', 'HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso);
```

See Also

- `makehdl`
- “RTL Design Rule Parameters” on page 17-80
- `hdlcoder.CodingStandard`
- HDL Coding Standard Customization

Model Generation Parameters for HDL Code

In the Configuration Parameter dialog box, you can select the types of the model that you want to generate. Select **HDL Code Generation > Global Settings > Model Generation**.

You can customize the name and layout of the generated model and the validation model by using “Naming and Layout Options for Model Generation” on page 17-85.

Generated model

Enable or disable generation of the generated model that shows latency and numeric differences between your Simulink DUT and the generated HDL code. Delays that the coder inserts are highlighted in the generated model.

Note When you select **Generated model**, the **Naming options** and **Layout options** become available.

Settings

Default: On



On

Select this setting to generate the generated model. By default, HDL Coder generates code and the generated model. To generate only the generated model, clear the **Generate HDL code** check box.



Off

Clear this setting when you do not want to generate the generated model. When you click the **Generate** button, HDL Coder generates code for the model.

Command-Line Information

Property: `GenerateModel`

Type: character vector

Value: `'on'` | `'off'`

Default: `'off'`

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

By default, the `GenerateHDLCode` property is enabled. You can use this property in conjunction with the `GenerateModel` property to specify whether to generate the generated model and the HDL code. To generate the code and the generated model, run `makehdl`.

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

If you want to generate only the generated model, disable the `GenerateHDLCode` property and run `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateModel', 'on');
hdlset_param('sfir_fixed', 'GenerateHDLCode', off');
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Balance delays” on page 15-3
- “Generated Model and Validation Model” on page 24-10
- “Prefix for generated model name” on page 17-85

Validation model

Enable or disable generation of a validation model that verifies the functional equivalence of the original model with the generated model. The validation model contains the original and the generated DUT models. You can use the generated DUT model to observe the effect of block settings and optimizations such as resource sharing, streaming, and delay balancing.

If you enable generation of a validation model, make sure that delay balancing is enabled on the model. In the **HDL Code Generation > Optimization > General** tab, select the **Balance delays** check box. Delay balancing keeps the generated DUT model synchronized with the original DUT model. Validation fails when there is a mismatch between delays in the original DUT model and delays in the generated DUT model.

Settings

Default: Off



On

Select this setting to generate the validation model. By default, HDL Coder generates code and the validation model. To generate only the validation model, clear the **Generate HDL code** check box.



Off

Clear this setting when you do not want to generate the validation model. When you click the **Generate** button, HDL Coder generates code for the model.

Command-Line Information

Property: `GenerateValidationModel`

Type: character vector

Value: ‘on’ | ‘off’

Default: ‘off’

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

By default, the `GenerateHDLCode` property is enabled. You can use this property in conjunction with the `GenerateValidationModel` property to specify whether to generate the validation model and the HDL code. To generate the code and the validation model, enable the `GenerateValidationModel` property with `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateValidationModel', 'on');
makehdl('sfir_fixed/symmetric_fir')
```

If you want to generate only the validation model, disable the `GenerateHDLCode` property and enable the `GenerateValidationModel` property with `makehdl`.

```
hdlset_param('sfir_fixed', 'GenerateValidationModel','on');
hdlset_param('sfir_fixed', 'GenerateHDLCode',off');
makehdl('sfir_fixed/symmetric_fir'
```

See Also

- “Balance delays” on page 15-3
- “Generated Model and Validation Model” on page 24-10
- “Suffix for validation model name” on page 17-85

Naming and Layout Options for Model Generation

This section contains different naming options available in **HDL Code Generation > Global Settings** pane under **Model Generation** tab. You can control the prefix for the generated model name and the suffix for the validation model name.

Prefix for generated model name

Specify the prefix to the generated model name.

Settings

Default: 'gm_'

Specify the prefix as a character vector. HDL Coder appends the prefix to name of generated model.

Command-Line Information

Property: GeneratedmodelNamePrefix

Type: character vector

Default: 'gm_'

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

For example, to indicate that you are using the generated model as a software interface model, you can use the prefix `sm_`. Specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model by using either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir',...
    'GeneratedmodelNamePrefix','sm_')
```

- When you use `hdlset_param`, set the parameter on the model, and then generate HDL code by using `makehdl`.

```
hdlset_param('sfir_fixed','GeneratedmodelNamePrefix','sm_')
makehdl('sfir_fixed/symmetric_fir')
```

Dependency

To specify **Prefix for generated model**, select **Generated model**.

See Also

- “Generated Model” on page 24-10
- “Suffix for validation model name” on page 17-85
- “Generated Model and Validation Model” on page 24-10

Suffix for validation model name

Specify the suffix for the validation model name.

Settings

Default: '_vnl'

Specify the suffix as a character vector. HDL Coder appends the suffix to name of validation model.

Command-Line Information

Property: ValidationModelNameSuffix

Type: character vector

Default: '_vnl'

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

For example, to indicate that you are using the generated model as a software interface model, you can use the suffix `_sm` for the validation model name. Specify this property when you generate HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model by using either of these methods:

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir',...
    'ValidationModelNameSuffix','_sm')
```

- When you use `hdlset_param`, set the parameter on the model, and then generate HDL code by using `makehdl`.

```
hdlset_param('sfir_fixed','ValidationModelNameSuffix','_sm')
makehdl('sfir_fixed/symmetric_fir')
```

Dependency

To specify **Suffix for validation model**, select **Generated model** and **Validation model**.

See Also

- “Validation model” on page 17-83
- “Prefix for generated model name” on page 17-85
- “Generated Model and Validation Model” on page 24-10

Auto block placement

Specify automatic placement of blocks in the HDL model.

Settings

Default: On

Command-Line Information

Property: autoplace

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param`:

```
hdlset_param(gcs,'autoplacement','on')
```

Dependency

To select **Auto block placement**, first select **Generated model**.

See Also

"Model Generation Parameters for HDL Code" on page 17-82

Auto signal routing

Specify automatic routing of signals in the generated HDL model.

Settings

Default: On

Command-Line Information

Property: autoroute

Type: character vector

Value: 'on' | 'off'

Default: 'on'

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'autoroute', 'on')
```

Dependency

To select **Auto signal routing**, first select **Auto block placement**.

See Also

"Model Generation Parameters for HDL Code" on page 17-82

Inter-block horizontal scaling

Scale the generated model horizontally. You can use this setting with **Inter-block vertical scaling** depending on how tightly packed or loosely packed you want the model to appear.

Settings

Default: 1.7

Command-Line Information

Property: InterBlkHorzScale

Type: positive integer | positive double

Default: 1.7

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'InterBlkHorzScale', 1.7)
```

Dependency

To select **Inter-block horizontal scaling**, first select **Auto block placement**.

See Also

"Model Generation Parameters for HDL Code" on page 17-82

Inter-block vertical scaling

Scale the generated model vertically. You can use this setting with **Inter-block horizontal scaling** depending on how tightly packed or loosely packed you want the model to appear.

Settings

Default: 1.2

Command-Line Information

Property: InterBlkVertScale

Type: positive integer | positive double

Default: 1.2

To set this property, use `hdlset_param`:

```
hdlset_param(gcs, 'InterBlkVertScale', 1.2)
```

Dependency

To select **Inter-block vertical scaling**, first select **Auto block placement**.

See Also

“Model Generation Parameters for HDL Code” on page 17-82

Diagnostic Parameters for Optimizations

This section contains parameters in the **Diagnostics** option under **Advanced** tab in the Configuration Parameters dialog box. Select **HDL Code Generation > Global Settings**. To highlight blocks and feedback loops that inhibit delay balancing, distributed pipelining, clock-rate pipelining, and other optimizations, use these parameters.

Highlight feedback loops inhibiting delay balancing and optimizations

Feedback loops in your Simulink model can inhibit delay balancing and optimizations such as resource sharing and streaming. Use this setting to generate a script that highlights feedback loops.

When you generate the feedback loop highlighting script, HDL Coder generates another script that clears the highlighting of feedback loops in your model. To turn off highlighting, click the link to the [clearhighlighting](#) script.

Settings

Default: On



On

Generate a MATLAB script that highlights feedback loops in the original model and the generated model. When you run the script, the code generator highlights the feedback loops using different colors. The highlighting script is saved in the same target folder as the generated HDL code.

It is recommended that you leave this setting enabled so that you can identify the feedback loops and further optimize your design.



Off

Do not generate a script to highlight feedback loops.

Command-Line Information

Property: `HighlightFeedbackLoops`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

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

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HighlightFeedbackLoops','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HighlightFeedbackLoops','off')
```

See Also

- “Find Feedback Loops” on page 24-90
- “Delay Balancing” on page 24-63
- “Generated Model and Validation Model” on page 24-10

Highlight blocks inhibiting clock-rate pipelining

Certain blocks in your Simulink model can inhibit clock-rate pipelining and therefore delimit clock-rate pipelining regions. Use this setting to generate a script to highlight the blocks.

When you generate the clock-rate pipelining highlighting script, HDL Coder generates another script that clears the highlighting. To turn off highlighting, click the link to the `clearhighlighting` script.

Settings

Default: On



On

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

It is recommended that you leave this setting enabled so that you can identify the blocks that delimit the clock-rate pipelining regions and further optimize your design.



Off

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

Command-Line Information

Property: `HighlightClockRatePipeliningDiagnostic`

Type: character vector

Value: ‘on’ | ‘off’

Default: ‘on’

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

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','HighlightClockRatePipeliningDiagnostic','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','HighlightClockRatePipeliningDiagnostic','off')
```

See Also

- “Pipelining Parameters” on page 15-9
- “Diagnostic Parameters for Reals and Black Box Interfaces” on page 17-92
- “Generated Model and Validation Model” on page 24-10

Highlight blocks inhibiting distributed pipelining

Certain blocks in your Simulink model can act as barriers for the distributed pipelining optimization. Use this setting to generate a script to highlight the blocks that are inhibiting distributed pipelining.

When you generate the highlighting script that displays distributed pipelining barriers, HDL Coder generates another script that clears the highlighting. To turn off highlighting, click the link to the [clearhighlighting](#) script.

Settings

Default: On



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

It is recommended that you leave this setting enabled so that you can identify the blocks that are barriers for distributed pipelining and further optimize your design.



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

Command-Line Information

Property: `DistributedPipeliningBarriers`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DistributedPipeliningBarriers','off')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','DistributedPipeliningBarriers','off')
```

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

See Also

- “Pipelining Parameters” on page 15-9
- “Diagnostic Parameters for Reals and Black Box Interfaces” on page 17-92
- “Generated Model and Validation Model” on page 24-10

Diagnostic Parameters for Reals and Black Box Interfaces

This section contains parameters in the **Diagnostics** option under **Advanced** tab in the Configuration Parameters dialog box. Select **HDL Code Generation > Global Settings**. To check for name conflicts in black box interfaces and for the presence of reals in the generated HDL code, use these parameters.

Check for name conflicts in black box interfaces

Specify whether to check for duplicate module or entity names in generated HDL code and black box interface HDL code.

Settings

Default: Warning

None

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

Warning

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

Error

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

Command-Line Information

Property: DetectBlackBoxNameCollision

Type: character vector

Value: 'None' | 'Warning' | 'Error'

Default: 'Warning'

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

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed','DetectBlackBoxNameCollision','None')
makehdl('sfir_fixed/symmetric_fir')
```

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir','DetectBlackBoxNameCollision','None')
```

See Also

- `makehdl`
- “Generate Black Box Interface for Subsystem” on page 27-4
- “Diagnostic Parameters for Optimizations” on page 17-89

Check for presence of reals in generated HDL code

Specify whether to check for reals in the generated HDL code.

Settings

Default: Error

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.

Command-Line Information

Property: TreatRealsInGeneratedCodeAs

Type: character vector

Value: 'None' | 'Warning' | 'Error'

Default: 'Error'

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

For example, you can specify this property while generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Use `hdlset_param` to set the parameter on the model. Then generate HDL code using `makehdl`.


```
hdlset_param('sfir_fixed','TreatRealsInGeneratedCodeAs','Warning')
makehdl('sfir_fixed/symmetric_fir')
```
- Pass the property as an argument to the `makehdl` function.


```
makehdl('sfir_fixed/symmetric_fir','TreatRealsInGeneratedCodeAs','Warning')
```

See Also

- `makehdl`
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- “Diagnostic Parameters for Optimizations” on page 17-89

Code Generation Output Parameter

You can specify whether or not to generate HDL code by using the **Generate HDL code** parameter. In the Configuration Parameters dialog box, select **HDL Code Generation > Global Settings > Advanced > Code generation output**.

Generate HDL code

Enable or disable HDL code generation for the model or Subsystem. To specify the Subsystem that you want to generate HDL code for, use the **Generate HDL for** parameter. Then, click the **Generate** button in the **HDL Code Generation** pane. By default, the HDL code is generated in VHDL language and put into the `hdlsrc` folder.

Settings

Default: On

On

Select this setting to generate HDL code.

Off

When you clear this setting, you cannot generate HDL code for the model.

Command-Line Information

Property: `GenerateHDLCode`

Type: character vector

Value: `'on'` | `'off'`

Default: `'on'`

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

By default, the `GenerateHDLCode` property is selected. To generate code, use the `makehdl` function. For example, this command generates HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model.

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

Control Code Generation Output

Property: `CodeGenerationOutput`

Type: character vector

Value: `'GenerateHDLCode'` |
`'GenerateHDLCodeAndDisplayGeneratedModel'` | `'DisplayGeneratedModelOnly'`

Default: `'GenerateHDLCode'`

By default, HDL Coder creates a model called the generated model when you generate HDL code. The generated model uses HDL-specific block implementations, and it implements the area and speed optimizations that you specify in your Simulink model. The code generator creates the generated model but does not display the model by default. To control display of the generated model, use the `CodeGenerationOutput` property.

This example shows how to generate HDL code, and then display the generated model by using `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'CodeGenerationOutput','GenerateHDLCodeAndDisplayGeneratedModel')
```

If you specify **DisplayGeneratedModelOnly**, the code generator displays the generated model but does not proceed to code generation.

See Also

- [makehdl](#)
- “Generated Model and Validation Model” on page 24-10
- “Model Generation Parameters for HDL Code” on page 17-82

HDL Code Generation Pane: Report

- “Report Pane Overview” on page 18-2
- “Code Generation Report Parameters” on page 18-3

Report Pane Overview

When you use the parameters in the **Report** pane, HDL Coder creates a Code Generation Report when generating HDL code for your model or Subsystem. The Code Generation Report contains a **Summary**, a **Code Interface Report**, and one or more of these reports.

- A traceability report that you can use to trace from the generated HDL code to the model and from the model to HDL code.
- A resource utilization report that contains the number of hardware resources used in the HDL code.
- An optimization report that displays the result of optimizations such as streaming, sharing, distributed pipelining, and floating-point target-specific information that was implemented in the generated code.
- A web view of the model that you can use to navigate between the generate code and your Simulink model.

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

Code Generation Report Parameters

In this section...

- “Generate traceability report” on page 18-3
- “Traceability style” on page 18-4
- “Generate model Web view” on page 18-5
- “Generate resource utilization report” on page 18-6
- “Generate high-level timing critical path report” on page 18-7
- “Generate optimization report” on page 18-8

This page describes configuration parameters that reside in the **HDL Code Generation > Report** pane of the Configuration Parameters dialog box. Enable these parameters to see the **Summary**, **Code Interface Report**, and reports that display traceability information, resource utilization, and effect of optimizations on your design.

Generate traceability report

Enable or disable generation of an HTML code generation report with hyperlinks from code to model and model to code. The report provides line-level traceability for each block in your Simulink model. When you click the hyperlink beside a certain line of code in the report, HDL Coder highlights the corresponding block in your Simulink model. When you select a certain block in your model, the report highlights all lines of code corresponding to that block.

Settings

Default: Off

On

Create and display a traceability report section in the HTML code generation report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the traceability report.

Off

Do not create an HTML code generation report.

Dependency

When you select this check box, you can select the **Traceability style**. By default, the **Traceability style** is **Line Level**.

Command-Line Information

Property: `Traceability`

Type: character vector

Value: `'on'` | `'off'`

Default: `'off'`

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

For example, you can generate a traceability report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `Traceability` property as an argument to `makehdl`.

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

- Enable the `Traceability` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','Traceability','on')
makehdl('sfir_fixed/symmetric_fir')
```

You can use the `RequirementComments` property to generate hyperlinked requirements comments within the HTML code generation report. The requirements comments link to the corresponding requirements documents for your model.

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-4
- `makehdl`

Traceability style

You can use **Traceability style** to specify whether you want to generate line-level or comment-based hyperlinks in the traceability report.

Settings

Default: Line Level

The options are:

Line Level

By default, HDL Coder generates a line-level traceability report that contains hyperlinks from each line of HDL code to the corresponding block in your Simulink model. The traceability report that is generated by using this style does not contain hyperlinked comments above the HDL code corresponding to a certain block. When you select a certain block and navigate to the HDL code, the code generator highlights all lines of code corresponding to that block.

Comment Based

If you specify generation of a comment-based traceability report, the report contains hyperlinked comments above a block of HDL code. The comments contain a traceability tag that contains a searchable pattern of the format `<system>/blockname`. `<system>` is the root model or a Subsystem inside the model, and `blockname` is the name of the block inside that model or Subsystem.

For example, if you have a model, `foo`, that has a Subsystem, `outer`, and a nested Subsystem, `Inner`, then the `<System>` tag is:

- `<Root>: foo`
- `<S1>: foo/outer`
- `<S2>: foo/outer/inner`

Dependency

To specify this setting, select the **Generate traceability report** check box.

Command-Line Information

Property: TraceabilityStyle

Type: character vector

Value: 'LineLevel' | 'CommentBased'

Default: 'LineLevel'

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

For example, when you generate a traceability report for the `symmetric_fir` subsystem inside the `sfir_fixed` model, specify the `TraceabilityStyle` by using either of these methods:

- Pass in the `TraceabilityStyle` property as an argument to `makehdl`.

```
makehdl('sfir_fixed/symmetric_fir','Traceability','on',...
        'TraceabilityStyle','CommentBased')
```

- Enable the `TraceabilityStyle` property using `hdlset_param`, and then use `makehdl`.

```
hdlset_param('sfir_fixed','Traceability','on')
hdlset_param(gcs,'TraceabilityStyle','CommentBased')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-4
- `makehdl`

Generate model Web view

Include the model Web view in the HDL Code Generation report to navigate between the code and model within the same window. With a model Web view, you can click a link in the generated code to highlight the corresponding block in the model. Using this capability, you can review, analyze, and debug the generated HDL code. You can share your model and generated code outside of the MATLAB environment.

Settings

Default: Off

On

Include model Web view in the Code Generation report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the model web view.

Off

Do not include model Web view in the Code Generation report.

Dependencies

To include a Web view (Simulink Report Generator) of the model in the Code Generation report, you must have Simulink Report Generator™ installed.

Command-Line Information

Parameter: HDLGenerateWebview

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

For example, you can generate a model web view when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `HDLGenerateWebview` property as an argument to `makehdl`.

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

- Enable the `HDLGenerateWebview` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','HDLGenerateWebview','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Web View of Model in Code Generation Report” on page 25-10
- `makehdl`

Generate resource utilization report

Enable or disable generation of an HTML resource utilization report. The report contains a summary and detailed information about the number of hardware resources, such as multipliers, adders, and registers that are used in the generated HDL code. If you have floating-point data types in your model, you can generate HDL code with native floating point support or map your design to Intel or Xilinx FPGA floating-point libraries. The resource utilization report displays a target-specific report corresponding to FPGA floating-point library mapping and a resource report corresponding to HDL code in native floating-point mode.

Settings

Default: Off

On

Create and display an HTML resource utilization report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the resource utilization report.

Off

Do not create an HTML resource utilization report.

Command-Line Information**Property:** ResourceReport**Type:** character vector**Value:** 'on' | 'off'**Default:** 'off'

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

For example, you can generate a resource utilization report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `ResourceReport` property as an argument to `makehdl`.

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

- Enable the `ResourceReport` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','ResourceReport','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

Generate high-level timing critical path report

Specify whether to generate a highlighting script that shows the estimated critical path. The report displays the critical path delay and generates a highlighting script as a link that you can click to highlight the estimated critical path in the generated model. If your design contains blocks without timing information, the report displays the link to another highlighting script that is generated to highlight those blocks.

Settings**Default:** Off

On

Generate a highlighting script that shows the estimated critical path. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the critical path estimation report.

To estimate the critical path for single-precision floating-point models, use the **Native Floating Point** mode. In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Floating Point Target** tab, set **Library** to **Native Floating Point**



Off

Do not calculate the estimated critical path.

Command-Line Information**Property:** CriticalPathEstimation

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

For example, you can generate a critical path estimation report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `CriticalPathEstimation` property as an argument to `makehdl`.

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

- Enable the `CriticalPathEstimation` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','CriticalPathEstimation','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- “Critical Path Estimation Without Running Synthesis” on page 24-137
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80
- `makehdl`

Generate optimization report

Enable or disable generation of an HTML optimization report. The report contains information about the results of distributed pipelining, streaming, sharing, delay balancing, and adaptive pipelining optimizations that are implemented in the generated code. The report includes hyperlinks back to referenced blocks, subsystems, or validation models. If you have floating-point data types in your model, you can generate HDL code with native floating point support or map your design to Intel or Xilinx FPGA floating-point libraries. When you map to FPGA floating-point libraries, the optimization report displays a target code generation section that displays the target device summary and a link to the generated model.

Settings

Default: Off



Create and display an HTML optimization report. To generate the report, after you enable this setting, click the **Generate** button. The code generation report contains a summary section and a code interface report along with the optimization report.



Do not create an HTML optimization report.

Command-Line Information

Property: `OptimizationReport`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

For example, you can generate an optimization report when generating HDL code for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass in the `OptimizationReport` property as an argument to `makehdl`.

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

- Enable the `OptimizationReport` property using `hdlset_param` and then use `makehdl`.

```
hdlset_param('sfir_fixed','OptimizationReport','on')
makehdl('sfir_fixed/symmetric_fir')
```

See Also

- “Create and Use Code Generation Reports” on page 25-2
- `makehdl`

HDL Code Generation Pane: Test Bench

- “Test Bench Overview” on page 19-2
- “Test Bench Generation Output Parameters” on page 19-3
- “Test Bench Postfix Parameters” on page 19-8
- “Clock and Reset Input Parameters for Testbench” on page 19-10
- “Setup and Hold Time Parameters for Testbench” on page 19-16
- “Test Bench Stimulus and Output Parameters” on page 19-18
- “Multi-File Testbench and Simulation Library Path Parameters” on page 19-23
- “Floating-Point Tolerance Parameters” on page 19-26

Test Bench Overview

The **Test Bench** pane lets you set options that determine characteristics of generated test bench code.

Generate Test Bench Button

The **Generate Test Bench** button initiates test bench generation for the system selected in the **Generate HDL for** menu on the parent HDL Code Generation pane. Make sure that the system selected is the DUT. Testbench generation is disabled if you select the entire model. See also `makehdltb`.

Test Bench Generation Output Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench > Test Bench Generation Output** section of the Configuration Parameters dialog box. Using the parameters in this section, you can specify the type of test bench to generate for verifying the HDL code, and the simulation tool.

HDL test bench

Enable or disable HDL test bench generation.

Settings

Default: selected



On

Enable generation of HDL test bench code. The code generator creates a HDL test bench by running a Simulink simulation to capture input vectors and expected output data for your DUT.

This test bench is the default test bench that HDL Coder generates for your model. If you have not already generated code for your model, running HDL test bench generation also generates code for your DUT.

Specify your HDL simulator in the **Simulation tool** menu. HDL Coder generates build-and-run scripts for the simulator that you specify.



Off

Suppress generation of HDL test bench code. You can use this option when you use an alternate test bench.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

This check box enables the options in the **Configuration** section of the **Test Bench** pane. Select a **Simulation tool** to generate scripts to build and run the test bench.

Command-Line Information

Property: GenerateHDLTestBench

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

For example, to generate a HDL test bench for the `sfir_fixed/symmetric_fir` Subsystem, pass the DUT as an argument to the `makehdltb` function.

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

Cosimulation model

Enable or disable generation of a model including a HDL Cosimulation block. This option requires an HDL Verifier license. After you select this check box, specify your **Simulation tool**. You can select Mentor Graphics ModelSim or Cadence Incisive® for cosimulation. Custom script settings are not supported with this test bench.

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

The coder appends the character vector that the **CosimLibPostfix** property specifies to the names of the generated HDL Cosimulation blocks.

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: `GenerateCoSimBlock`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

Property: `GenerateCoSimModel`

Type: character vector

Value: 'ModelSim' | 'Incisive' | 'None'

Default: 'ModelSim'

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

For example, you can enable the `GenerateCoSimModel` property when you generate a testbench for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'GenerateCoSimModel','ModelSim')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed','GenerateCoSimModel','ModelSim')
makehdltb('sfir_fixed/symmetric_fir')
```

See Also

- “Simulation tool” on page 19-6
- “Generate a Cosimulation Model” on page 27-41

SystemVerilog DPI test bench

Enable or disable generation of SystemVerilog DPI test bench. Select your HDL simulator at **Simulation tool**. For SystemVerilog DPI test bench you can select Mentor Graphics ModelSim, Cadence Incisive, SynopsysVCS®, or Xilinx Vivado. Custom script settings are not supported with this test bench.

When you set this property, the code generator 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 code generator generates a SystemVerilog test bench that compares the output of the DPI component with the output of the HDL implementation of your DUT. The coder also builds shared libraries and generates a simulation script for the simulator you select.

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

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

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Limitations

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.

Command-Line Information

Property: GenerateSVDPITestBench

Type: character vector

Value: 'ModelSim' | 'Incisive'|'Custom'|'VCS'||'Vivado'

Default: 'ModelSim'

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

For example, you can enable the `GenerateCosimModel` property when you generate a testbench for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'GenerateSVDPITestBench','ModelSim')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed','GenerateSVDPITestBench','ModelSim')
makehdltb('sfir_fixed/symmetric_fir')
```

See Also

- “Generate a SystemVerilog DPI Component” (HDL Verifier)
- “Simulation tool” on page 19-6
- “Verify HDL Design Using SystemVerilog DPI Test Bench” on page 27-79

Simulation tool

Simulator where you will run the generated test benches. The tool generates a script to build and run your HDL code and test bench.

Settings

- **Mentor Graphics ModelSim**: This option is the default. HDL Coder generates the selected types of test benches for use with Mentor Graphics ModelSim.
- **Cadence Incisive**: The coder generates the selected types of test benches for use with Cadence Incisive.
- **Custom**: Selecting this option enables the custom script options on the **EDA Tool Scripts** pane.
- **VCS**: This simulator is supported only for **SystemVerilog DPI test bench**.
- **Vivado**: This simulator is supported only for **SystemVerilog DPI test bench**.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

For HDL test bench, use the `SimulationTool` property. For cosimulation, use the `GenerateCosimModel` property. For SystemVerilog DPI test bench, use the `GenerateSVDPITestbench` property.

Property: `SimulationTool`

Type: character vector

Value: 'Mentor Graphics ModelSim' | 'Cadence Incisive'|'Custom'

Default: 'Mentor Graphics ModelSim'

Property: `GenerateCosimModel`

Type: character vector

Value: 'ModelSim' | 'Incisive' |None

Default: 'ModelSim'

Property: `GenerateSVDPITestbench`

Type: character vector

Value: 'ModelSim' | 'Incisive' | 'Custom' | 'VCS' | 'Vivado'

Default: 'ModelSim'

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

HDL code coverage

Enable or disable HDL code coverage flags in the generated simulator scripts

With this option enabled, when you run the HDL simulation, code coverage is collected for your generated test bench. Specify your HDL simulator in the `SimulationTool` property. The coder generates build-and-run scripts for the simulator you specify.

Settings

Default: not selected

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: `HDLCodeCoverage`

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

For example, you can enable the `HDLCodeCoverage` property when you generate a testbench for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'HDLCodeCoverage', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed','HDLCodeCoverage','on')
makehdltb('sfir_fixed/symmetric_fir')
```

Test Bench Postfix Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can customize the postfix for the test bench name, data file, and test bench reference.

Test bench name postfix

Specify a suffix appended to the test bench name.

Settings

Default: '_tb'

For example, if the name of your DUT is `my_test`, HDL Coder adds the default postfix `_tb` to form the name `my_test_tb`.

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Property: `TestBenchPostFix`

Type: character vector

Default: '`_tb`'

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

Test bench reference postfix

Specify a character vector to be appended to names of reference signals generated in test bench code.

Settings

Default: '`_ref`'

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

Dependencies

Make sure that the system selected is the DUT. This option is disabled if you select the entire model.

Command-Line Information

Parameter: `TestBenchReferencePostFix`

Type: character vector

Default: '`_ref`'

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

Test bench data file name postfix

Specify suffix added to test bench data file name when generating multi-file test bench.

Settings

Default: '_data'

HDL Coder applies the **Test bench data file name postfix** character vector only when generating a multi-file test bench (i.e., when **Multi-file test bench** is selected).

For example, if the name of your DUT is `my_test`, and **Test bench name postfix** has the default value `_tb`, the coder adds the postfix `_data` to form the test bench data file name `my_test_tb_data`.

Dependency

This parameter is enabled by **Multi-file test bench**.

Command-Line Information

Property: `TestBenchDataPostFix`

Type: character vector

Default: '_data'

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

Clock and Reset Input Parameters for Testbench

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can specify the clock high time, clock low time, and whether you want the test bench to force clock, reset, and clock enable input signals.

Force clock

Specify whether the test bench forces clock input signals.

Settings

Default: On

On

The test bench forces the clock input signals. When this option is selected, the clock high and low time settings control the clock waveform.

Off

A user-defined external source forces the clock input signals.

Dependencies

This property enables the **Clock high time** and **Clock low time** options. This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceClock

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'ForceClock', 'off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'ForceClock', 'off')
makehdltb('sfir_fixed/symmetric_fir')
```

Clock high time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals high (1).

Settings

Default: 5

Specify a positive integer value. The **Clock high time** and **Clock low time** properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Dependency

This parameter is enabled when **Force clock** is selected.

Command-Line Information

Property: ClockHighTime

Type: integer

Value: positive integer

Default: 5

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
    'ClockHighTime', 2)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'ClockHighTime', 2)
makehdl('sfir_fixed/symmetric_fir')
```

Clock low time (ns)

Specify the period, in nanoseconds, during which the test bench drives clock input signals low (0).

Settings

Default: 5

The **Clock high time** and **Clock low time** properties define the period and duty cycle for the clock signal. Using the defaults, the clock signal is a square wave (50% duty cycle) with a period of 10 ns.

Dependency

This parameter is enabled when **Force clock** is selected.

Command-Line Information

Property: ClockLowTime

Type: integer

Value: positive integer

Default: 5

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'ClockLowTime', 2)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'ClockLowTime', 2)
makehdltb('sfir_fixed/symmetric_fir')
```

Force clock enable

Specify whether the test bench forces clock enable input signals.

Settings

Default: On



The test bench forces the clock enable input signals to active-high (1) or active-low (0), depending on the setting of the clock enable input value.



A user-defined external source forces the clock enable input signals.

Dependencies

This property enables the **Clock enable delay (in clock cycles)** option.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceClockEnable

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'ForceClockEnable', 'off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'ForceClockEnable', 'off')
makehdltb('sfir_fixed/symmetric_fir')
```

Clock enable delay (in clock cycles)

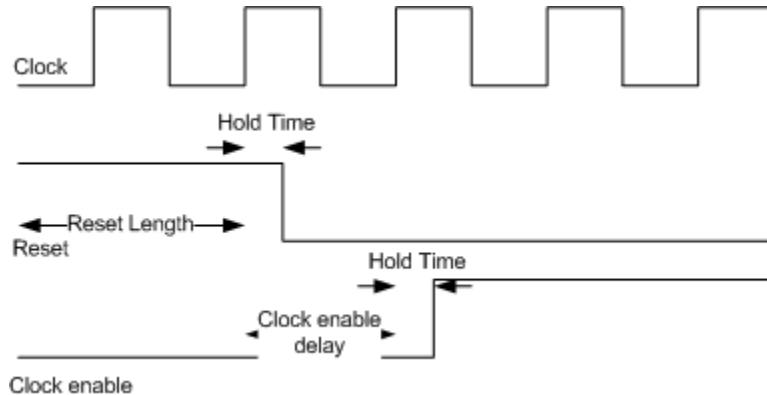
Define elapsed time (in clock cycles) between deassertion of reset and assertion of clock enable.

Settings

Default: 1

The **Clock enable delay (in clock cycles)** property defines the number of clock cycles elapsed between the time the reset signal is deasserted and the time the clock enable signal is first asserted. In the figure below, the reset signal (active-high) deasserts after 2 clock cycles and the clock enable asserts after a clock enable delay of 1 cycle (the default).

In the figure below, the reset signal (active-high) de-asserts after the interval labelled **Hold Time**. The clock enable asserts after a further interval labelled **Clock enable delay**.



Dependency

This parameter is enabled when **Force clock enable** is selected.

Command-Line Information

Property: `TestBenchClockEnableDelay`

Type: integer

Default: 1

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'TestBenchClockEnableDelay', 2)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'TestBenchClockEnableDelay', 2)
makehdltb('sfir_fixed/symmetric_fir')
```

Force reset

Specify whether the test bench forces reset input signals.

Settings

Default: On



On

The test bench forces the reset input signals.



Off

A user-defined external source forces the reset input signals.

Tips

If you select this option, you can use the **Hold time** option to control the timing of a reset.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: ForceReset

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'ForceReset', 'off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'ForceReset', 'off')
makehdltb('sfir_fixed/symmetric_fir')
```

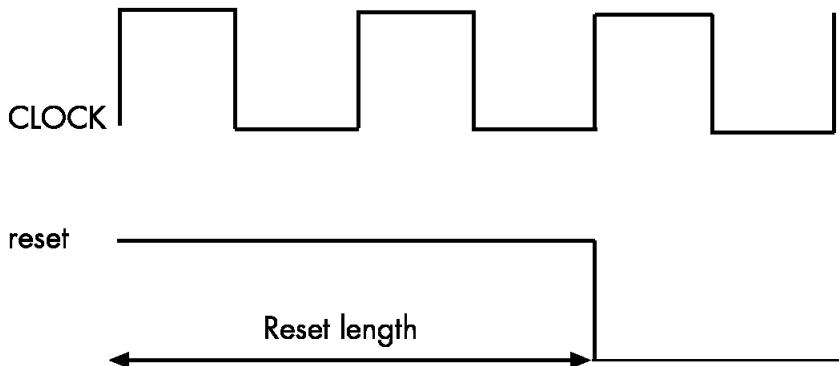
Reset length (in clock cycles)

Define length of time (in clock cycles) during which reset is asserted.

Settings

Default: 2

The **Reset length (in clock cycles)** property defines the number of clock cycles during which reset is asserted. **Reset length (in clock cycles)** must be an integer greater than or equal to 0. The following figure illustrates the default case, in which the reset signal (active-high) is asserted for 2 clock cycles.



Dependency

This parameter is enabled when **Force reset** is selected.

Command-Line Information

Property: Resetlength

Type: integer

Default: 2

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'Resetlength', 4)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'Resetlength', 4)
makehdltb('sfir_fixed/symmetric_fir')
```

Setup and Hold Time Parameters for Testbench

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can specify the setup time for data input and hold time for data input and forced reset signal signals.

Hold time (ns)

Specify a hold time, in nanoseconds, for input signals and forced reset input signals.

Settings

Default: 2 (given the default clock period of 10 ns)

The hold time defines the number of nanoseconds that reset input signals and input data are held past the clock rising edge. The hold time is expressed as a positive integer or double (with a maximum of 6 significant digits after the decimal point).

Tips

- The specified hold time must be less than the clock period (specified by the **Clock high time** and **Clock low time** properties).
- This option applies to reset input signals only if **Force reset** is selected.

Usage Notes

Hold Time for Reset Input Signals

The hold time is the amount of time that reset input signals and input data are held past the clock rising edge. The following figures show the application of a hold time (t_{hold}) for reset and data input signals when the signals are forced to active high and active low.

Hold Time for Data Input Signals

Note A reset signal is always asserted for two cycles plus t_{hold} .

Dependencies

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: HoldTime

Type: integer

Value: positive integer

Default: 2

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'HoldTime', 4)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'HoldTime', 4)
makehdltb('sfir_fixed/symmetric_fir')
```

Setup time (ns)

Display setup time for data input signals.

Settings

Default: None

This is a display-only field, showing a value computed as (clock period - HoldTime) in nanoseconds.

Dependency

The value displayed in this field depends on the clock rate and the values of the **Hold time** property.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Because this is a display-only field, a corresponding command-line property does not exist.

Test Bench Stimulus and Output Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can specify whether to ignore data checking, and hold input data between samples.

Hold input data between samples

Specify how long substrate signal values are held in valid state.

Settings

Default: On

On

Data values for substrate signals are held in a valid state across N base-rate clock cycles, where N is the number of base-rate clock cycles that elapse per substrate sample period. (N ≥ 2 .)

Off

Data values for substrate signals are held in a valid state for only one base-rate clock cycle. For the subsequent base-rate cycles, data is in an unknown state (expressed as 'X') until leading edge of the next substrate sample period.

Tip

In most cases, the default (On) is the best setting for **Hold input data between samples**. This setting matches the behavior of a Simulink simulation, in which substrate signals are held valid through each base-rate clock period.

In some cases (for example modeling memory or memory interfaces), it is desirable to clear **Hold input data between samples**. In this way you can obtain diagnostic information about when data is in an invalid ('X') state.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: HoldInputDataBetweenSamples

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'HoldInputDataBetweenSamples', 'off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'HoldInputDataBetweenSamples', 'off')
makehdltb('sfir_fixed/symmetric_fir')
```

Initialize test bench inputs

Specify initial value driven on test bench inputs before data is asserted to DUT.

Settings

Default: Off



On
Initial value driven on test bench inputs is '0'.



Off
Initial value driven on test bench inputs is 'X' (unknown).

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: InitializeTestBenchInputs

Type: character vector

Value: 'on' | 'off'

Default: 'off'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'InitializeTestBenchInputs', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'InitializeTestBenchInputs', 'on')
makehdltb('sfir_fixed/symmetric_fir')
```

Ignore output data checking (number of samples)

Specify number of samples during which output data checking is suppressed.

Settings

Default: 0

The value must be a positive integer.

When the value of **Ignore output data checking (number of samples)**, N, is greater than zero, the test bench suppresses output data checking for the first N output samples after the clock enable output (`ce_out`) is asserted.

When using pipelined block implementations, output data may be in an invalid state for some number of samples. To avoid spurious test bench errors, determine this number and set **Ignore output data checking (number of samples)** accordingly.

Be careful to specify N as a number of samples, not as a number of clock cycles. For a single-rate model, these are equivalent, but they are not equivalent for a multirate model.

You should use **Ignore output data checking (number of samples)** in cases where there is a state (register) initial condition in the HDL code that does not match the Simulink state, including the following specific cases:

- When you set the **DistributedPipelining** property to 'on' for the MATLAB Function block (see "Distributed Pipeline Insertion for MATLAB Function Blocks" on page 29-37)
- When you set the **ResetType** property to 'None' for the following blocks:
 - `commcnvinrlv2/Convolutional Deinterleaver`
 - `commcnvinrlv2/Convolutional Interleaver`
 - `commcnvinrlv2/General Multiplexed Deinterleaver`
 - `commcnvinrlv2/General Multiplexed Interleaver`
 - `dpsigops/Delay`
 - `simulink/Additional Math & Discrete/Additional Discrete/Unit Delay Enabled`
 - `simulink/Commonly Used Blocks/Unit Delay`
 - `simulink/Discrete/Delay`
 - `simulink/Discrete/Memory`
 - `simulink/Discrete/Tapped Delay`
 - `simulink/User-Defined Functions/MATLAB Function`
 - `sflib/Chart`
 - `sflib/Truth Table`
- When generating a black box interface to existing manually written HDL code

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: `IgnoreDataChecking`

Type: integer

Default: 0

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'IgnoreDataChecking', 2)
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'IgnoreDataChecking', 2)
makehdl('sfir_fixed/symmetric_fir')
```

Use file I/O to read/write test bench data

Create and use data files for reading and writing test bench input and output data.

Settings

Default: On

On

Create and use data files for reading and writing test bench input and output data.

Off

Use constants in the test bench for DUT stimulus and reference data.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: `UseFileIOInTestBench`

Type: character vector

Value: '`on`' | '`off`'

Default: '`on`'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdl` function.

```
makehdl('sfir_fixed/symmetric_fir', ...
        'UseFileIOInTestBench', 'off')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdl`.

```
hdlset_param('sfir_fixed', 'UseFileIOInTestBench', 'off')
makehdltb('sfir_fixed/symmetric_fir')
```

Multi-File Testbench and Simulation Library Path Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can specify the simulation library path and whether to generate a multi-file testbench.

Multi-file test bench

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

Description

You can use this setting to specify how you want to divide files that contain the test bench code, data, and helper functions.

The file names are derived from the name of the DUT, the **Test bench name postfix** property, and the **Test bench data file name postfix** property as:

DUTname_TestBenchPostfix_TestBenchDataPostfix

For example, if the DUT name is `symmetric_fir`, and the target language is VHDL, the default test bench file names are:

- `symmetric_fir_tb.vhd`: test bench code
- `symmetric_fir_tb_pkg.vhd`: helper functions package
- `symmetric_fir_tb_data.vhd`: data package

If the DUT name is `symmetric_fir` and the target language is Verilog, the default test bench file names are:

- `symmetric_fir_tb.v`: test bench code
- `symmetric_fir_tb_pkg.v`: helper functions package
- `symmetric_fir_tb_data.v`: test bench data

Settings

Default: Off

On

Write three separate HDL files. There is a separate file for test bench code, helper functions, and test bench data.

Off

Write two separate HDL files. One file contains the HDL test bench code. The other file contains the helper functions package and test bench data.

Dependency

When this property is selected, **Test bench data file name postfix** is enabled.

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information**Property:** MultifileTestBench**Type:** character vector**Value:** 'on' | 'off'**Default:** 'off'

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

For example, you can specify this parameter for the `symmetric_fir` subsystem inside the `sfir_fixed` model using either of these methods.

- Pass the property as an argument to the `makehdltb` function.

```
makehdltb('sfir_fixed/symmetric_fir', ...
          'MultifileTestBench', 'on')
```

- When you use `hdlset_param`, you can set the parameter on the model and then generate HDL code using `makehdltb`.

```
hdlset_param('sfir_fixed', 'MultifileTestBench', 'on')
makehdltb('sfir_fixed/symmetric_fir')
```

Simulation library path

Specify the path to your compiled Altera or Xilinx simulation libraries.

Settings

Default: ''

Specify the path to the compiled Altera or Xilinx simulation libraries. Altera provides the simulation model files in `\quartus\eda\sim_lib` folder.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information**Property:** SimulationLibPath**Type:** character vector**Default:** ''

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

For example, if you want to set the path to the compiled Xilinx Simulation library, enter:

```
myDUT = gcb;

libpath = '/apps/Xilinx_ISE/XilinxISE-13.4/Linux/ISE_DS/ISE/vhdl/
           mti_se/6.6a/lin64/xilinxcorelib';

hdlset_param (myDUT, 'SimulationLibPath', libpath);
```

```
makehdltb(myDUT)
```

Floating-Point Tolerance Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > Test Bench** tab of the Configuration Parameters dialog box. Using the parameters in this tab, you can specify the floating-point tolerance strategy and the tolerance value.

Floating point tolerance check based on

When you map your design to the native floating-point libraries or the floating-point target libraries, specify the floating-point tolerance check option.

Settings

Default: relative error

Select one of these options from the dropdown menu:

- **relative error**: This is the default option. When you verify the generated code by using HDL Testbench, HDL Coder checks for the floating-point tolerance of the native floating-point library or the floating-point target library that your design mapped to based on the relative error.
- **ulp error**: When you verify the generated code by using HDL Testbench, HDL Coder checks for the floating-point tolerance of the native floating-point library or the floating-point target library that your design mapped to based on the ULP error.

Dependency

This option is disabled if you select the entire model. Select the DUT instead for **Generate HDL for** setting.

Command-Line Information

Property: FPToleranceStrategy

Type: character vector

Value: 'relative' | 'ULP'

Default: 'relative'

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

For example, to specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% Check for floating-point tolerance based on ULP
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');

% When using ULP, optionally enter tolerance value >= 0
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);

% Generate HDL testbench with specified tolerance setting
makehdltb('sfir_single/symmetric_fir')
```

Tolerance Value

Enter the tolerance value based on the floating-point tolerance check setting that you specify.

Settings

Default: 1e-07

The value must be a positive integer or a double data type.

The default tolerance value depends on the floating-point tolerance check setting that you specify. When you set the **Floating point tolerance check based on** to:

- **relative error**, the default is a **Tolerance Value** of 1e-07. When you use this floating-point tolerance check setting, specify the tolerance value as a double data type. You can specify a **Tolerance Value**, N, that is less than or equal to 1e-07.
- **ulp error**, the default is a **Tolerance Value** of 0. When you use this floating-point tolerance check setting, specify the tolerance value as an integer. You can specify a **Tolerance Value**, N, that is greater than or equal to 0.

Command-Line Information

Property: FPToleranceValue

Type: double | integer

Default: 1e-07

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

For example, to specify the floating-point tolerance value for a model, use the `hdlset_param` function to specify the tolerance strategy, and then enter the tolerance value. For example, to check the floating-point tolerance based on ULP error and enter the tolerance value:

```
% Check for floating-point tolerance based on ULP
hdlset_param('sfir_single', 'FPToleranceStrategy', 'ULP');

% When using ULP, optionally enter tolerance value >= 0
hdlset_param('FP_test_16a', 'FPToleranceValue', 1);

% Generate HDL testbench with specified tolerance setting
makehdl('sfir_single/symmetric_fir')
```


HDL Code Generation Pane: EDA Tool Scripts

- “EDA Tool Scripts Overview” on page 20-2
- “Generate EDA scripts” on page 20-3
- “Compilation Script Parameters” on page 20-4
- “Simulation Script Parameters” on page 20-7
- “Synthesis Script Parameters” on page 20-11
- “Lint Script Parameters” on page 20-16

EDA Tool Scripts Overview

The **EDA Tool Scripts** pane lets you set the options that control generation of script files for third-party HDL simulation and synthesis tools.

Generate EDA scripts

Enable generation of script files for third-party electronic design automation (EDA) tools. These scripts let you compile and simulate generated HDL code and/or synthesize generated HDL code.

Settings

Default: On



On
Generation of script files is enabled.



Off
Generation of script files is disabled.

Command-Line Information

Parameter: EDAScriptGeneration

Type: character vector

Value: 'on' | 'off'

Default: 'on'

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Compilation Script Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > EDA Tool Scripts > Compilation Script** tab of the Configuration Parameters dialog box.

Compile file postfix

Specify a postfix to append to the DUT or test bench name to form the compilation script file name.

Settings

Default: `_compile.do`

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_compile.do` to form the name `my_design_compile.do`.

Command-Line Information

Property: `HDLCompileFilePostfix`

Type: character vector

Default: `'_compile.do'`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Compile initialization

Format name passed to `fprintf` to write the `Init` section of the compilation script.

Settings

Default: `vlib %s\n`

The `Init` phase of the script performs required setup actions, such as creating a design library or a project file.

The implicit argument, `%s`, is the contents of the '`VHDLLibNameName`' property, which defaults to '`work`'. You can override the default `Init` string (`'vlib work\n'`) by changing the value of '`VHDLLibNameName`'.

Command-Line Information

Property: `HDLCompileInit`

Type: character vector

Default: `'vlib %s\n'`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Compile command for VHDL

Format name passed to `fprintf` to write the `Cmd` section of the compilation script for VHDL files.

Settings

Default: `vcom %s %s\n`

The command-per-file phase (`Cmd`) of the script is called iteratively, once per generated HDL file. On each call, a different file name is passed in.

The two implicit arguments in the compile command are the contents of the `SimulatorFlags` property and the file name of the current entity or module. To omit the flags, set `SimulatorFlags` to '' (the default).

Command-Line Information

Property: `HDLCompileVHDLCmd`

Type: character vector

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

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Compile command for Verilog

Format name passed to `fprintf` to write the `Cmd` section of the compilation script for Verilog files.

Settings

Default: `vlog %s %s\n`

The command-per-file phase (`Cmd`) of the script is called iteratively, once per generated HDL file. On each call, a different file name is passed in.

The two implicit arguments in the compile command are the contents of the `SimulatorFlags` property and the file name of the current entity or module. To omit the flags, set `SimulatorFlags` property to '' (the default).

Command-Line Information

Property: `HDLCompileVerilogCmd`

Type: character vector

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

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Compile termination

Format name passed to `fprintf` to write the termination portion of the compilation script.

Settings

Default: empty character vector

The termination phase (`Term`) is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase does not take arguments.

Command-Line Information

Property: `HDLCompileTerm`

Type: character vector

Default: ''

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulation Script Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > EDA Tool Scripts > Simulation Script** tab of the Configuration Parameters dialog box.

Simulation file postfix

Specify a postfix to append to the DUT or test bench name to form the simulation script file name.

Settings

Default: `_sim.do`

For example, if the name of the device under test or test bench is `my_design`, HDL Coder adds the postfix `_sim.do` to form the name `my_design_sim.do`.

Command-Line Information

Property: `HDLSimFilePostfix`

Type: character vector

Default: `'_sim.do'`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulation initialization

Format name passed to `fprintf` to write the initialization section of the simulation script.

Settings

Default: The default is

```
[ 'onbreak resume\nnonerror resume\n' ]
```

The `Init` phase of the script performs required setup actions, such as creating a design library or a project file.

Command-Line Information

Property: `HDLSimInit`

Type: character vector

Default: `['onbreak resume\nnonerror resume\n']`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2

- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulation command

Format name passed to `fprintf` to write the simulation command.

Settings

Default: `vsim -voptargs+=acc %s.%s\n`

The first implicit argument, `%s`, is the library name. The second implicit argument is the top-level module or entity name. If your target language is VHDL, the library name is the value of “VHDL library name” on page 17-30. If your target language is Verilog, the library name is ‘`work`’ and cannot be changed.

If you compile your filter design with code from other libraries, update **VHDL library name** to avoid library name conflicts.

Note Prior to R2020b, the default simulation command was `vsim -novopt %s.%s\n`. Mentor Graphics ModelSim versions prior to 10.7 support the former syntax. If you use a more recent Mentor Graphics ModelSim version, use the `vsim -voptargs+=acc %s.%s\n` syntax.

Command-Line Information

Property: HDLSimCmd

Type: character vector

Default: ‘`vsim -novopt %s.%s\n`’

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulation waveform viewing command

Specify the waveform viewing command written to simulation script.

Settings

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

The implicit argument, `%s`, adds the signal paths for the DUT top-level input, output, and output reference signals.

Command-Line Information

Property: HDLSimViewWaveCmd

Type: character vector

Default: ‘`add wave sim:%s\n`’

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulation termination

Format name passed to `fprintf` to write the termination portion of the simulation script.

Settings

Default: `run -all\n`

The termination phase (`Term`) is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase does not take arguments.

Command-Line Information

Property: `HDLSimTerm`

Type: character vector

Default: '`run -all\n`'

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Simulator flags

Specify simulator flags to apply to generated compilation scripts.

Settings

Default: '' (no simulator flags)

Specify simulator flags to apply to generated compilation scripts as a character vector. The simulator flags are specific to your application and the simulator you are using. For example, if you must use the 1076-1993 VHDL compiler, specify the flag `-93`.

The flags you specify with this option are added to the compilation command in generated compilation scripts. The simulation command is specified by the `HDLCompileVHDLCmd` or `HDLCompileVerilogCmd` properties.

Command-Line Information

Property: `SimulatorFlags`

Type: character vector

Default: ''

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Synthesis Script Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > EDA Tool Scripts > Synthesis Script** tab of the Configuration Parameters dialog box.

Choose synthesis tool

Enable or disable generation of synthesis scripts, and select the synthesis tool for which HDL Coder generates scripts.

Settings

Default: None

None

When you select None, HDL Coder does not generate a synthesis script. The coder clears and disables the fields in the **Synthesis script** pane.

Xilinx ISE

Generate a synthesis script for Xilinx ISE. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_ise.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Microsemi Libero

Generate a synthesis script for Microsemi Libero. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_libero.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Mentor Graphics Precision

Generate a synthesis script for Mentor Graphics Precision. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_precision.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Altera Quartus II

Generate a synthesis script for Altera Quartus II. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_quartus.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Synopsys Synplify Pro

Generate a synthesis script for Synopsys Synplify Pro. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_synplify.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Xilinx Vivado

Generate a synthesis script for Xilinx Vivado. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_vivado.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with TCL script code for the tool.

Custom

Generate a custom synthesis script. When you select this option, the coder:

- Enables the fields in the **Synthesis script** pane.
- Sets **Synthesis file postfix** to `_custom.tcl`
- Fills in the **Synthesis initialization**, **Synthesis command** and **Synthesis termination** fields with example TCL script code.

Command-Line Information

Property: HDLSynthTool

Type: character vector

Value: 'None' | 'ISE' | 'Libero' | 'Precision' | 'Quartus' | 'Synplify' | 'Vivado' | 'Custom'

Default: 'None'

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Synthesis file postfix

Specify a postfix to append to file name for generated synthesis scripts.

Settings

Default: None.

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the postfix for generated synthesis file names to one of the following:

`_ise.tcl`

```
_libero.tcl
_precision.tcl
_quartus.tcl
_synplify.tcl
_vivado.tcl
_custom.tcl
```

For example, if the DUT name is `my_design` and the choice of synthesis tool is `Synopsys Synplify Pro`, HDL Coder adds the postfix `_synplify.tcl` to form the name `my_design_synplify.tcl`.

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information

Property: `HDLSynthFilePostfix`

Type: character vector

Default: `none`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Synthesis initialization

Format name passed to `fprintf` to write the initialization section of the synthesis script.

Settings

Default: `none`.

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the **Synthesis initialization** string. The content of the string is specific to the selected synthesis tool.

The default is a synthesis project creation command passed as a format string to `fprintf` to write the `Init` section of the synthesis script. The implicit argument, `%s`, is the top-level module or entity name.

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information

Property: `HDLSynthInit`

Type: character vector

Default: `none`

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Synthesis command

Format name passed to `fprintf` to write the synthesis command.

Settings

Default: none.

Your choice of synthesis tool (from the **Choose synthesis tool** menu) sets the **Synthesis command** string. The content of the string is specific to the selected synthesis tool.

The default is a format string passed to `fprintf` to write the `Cmd` section of the synthesis script. The implicit argument, `%s`, is the file name of the entity or module. The command is iterated for each generated file.

To avoid issues when generating synthesis scripts for various tools, retain both format specifiers (`%s`).

Dependency

To use this setting, the **Choose synthesis tool** or `HDLSynthTool` property must be set to a value other than `None`.

Command-Line Information

Property: `HDLSynthCmd`

Type: character vector

Default: none

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Synthesis termination

Specify a format name that is passed to `fprintf` to write the termination portion of the synthesis script.

Settings

Default: none

Your choice of synthesis tool (from the **Choose synthesis tool** pulldown menu) sets the **Synthesis termination** string. The content of the string is specific to the selected synthesis tool.

The default is a format name passed to `fprintf` to write the `Term` section of the synthesis script. The termination string does not take arguments.

Dependency

To use this setting, the **Choose synthesis tool** or **HDLSynthTool** property must be set to a value other than None.

Command-Line Information

Property: HDLSynthTerm

Type: character vector

Default: none

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

See Also

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Additional files to add to synthesis project

Include additional HDL or constraint files in synthesis project.

Settings

Default: '' (no files added)

Additional project files, such as HDL source files (.v, .vhdl) or constraint files (.ucf), that you want to include in your synthesis project, specified as a character vector. Separate file names with a semicolon (;).

You cannot use this setting to include Tcl files. To specify synthesis project Tcl files, use the `AdditionalProjectCreationTclFiles` property of the `hdlcoder.WorkflowConfig` object.

Command-Line Information

Property: SynthesisProjectAdditionalFiles

Type: character vector

Default: ''

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

To include a source file, `src_file.vhd`, and a constraint file, `constraint_file.ucf`, in the synthesis project for a DUT subsystem, `myDUT`:

```
hdlset_param (myDUT, 'SynthesisProjectAdditionalFiles', ...
              'L:\src_file.vhd;L:\constraint_file.ucf;')
```

See Also

- `hdlcoder.WorkflowConfig`
- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7

Lint Script Parameters

This page describes configuration parameters that reside in the **HDL Code Generation > EDA Tool Scripts > Lint Script** tab of the Configuration Parameters dialog box.

Choose HDL lint tool

Enable or disable generation of an HDL lint script, and select the HDL lint tool for which HDL Coder generates a script.

After you select an HDL lint tool, the **Lint initialization**, **Lint command** and **Lint termination** fields are enabled.

Dependencies

If you set the HDL lint tool to one of the supported third-party tools, you can generate a Tcl script without setting **Lint initialization**, **Lint command**, and **Lint termination** to nondefault values. If the **Lint initialization**, **Lint command**, and **Lint termination** have default values, HDL Coder automatically writes tool-specific default initialization, command, and termination strings to the Tcl script.

Settings

Default: None

None

When you select None, the coder does not generate a lint script. The coder clears and disables the fields in the **Lint script** pane.

Ascent Lint

Generate a lint script for Real Intent Ascent Lint.

HDL Designer

Generate a lint script for Mentor Graphics HDL Designer.

Leda

Generate a lint script for Synopsys Leda.

SpyGlass

Generate a lint script for Atrenta SpyGlass.

Custom

Generate a custom synthesis script.

Command-Line Information

Property: `HDLLintTool`

Type: character vector

Value: 'None' | 'AscentLint' | 'Leda' | 'SpyGlass' | 'Custom'

Default: 'None'

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

See Also

"Generate an HDL Lint Tool Script" on page 26-45

Lint initialization

Enter an initialization text for your HDL lint script.

Dependencies

If **Lint initialization** is set to the default value, ' ', and you set **HDLLintCmd** to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default termination string in the Tcl script.

Command-Line Information

Property: `HDLLintInit`

Type: character vector

Default: none

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

See Also

"Generate an HDL Lint Tool Script" on page 26-45

Lint command

Enter the command for your HDL lint script.

Command-Line Information

Property: `HDLLintCmd`

Type: character vector

Default: none

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

If you set **HDLLintTool** to `Custom`, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script. Specify **HDLLintCmd** using the following format:

```
custom_lint_tool_command -option1 -option2 %s
```

See Also

"Generate an HDL Lint Tool Script" on page 26-45

Lint termination

Enter a termination character vector for your HDL lint script.

Dependencies

If **Lint termination** is set to the default value, ' ', and you set **HDLLintCmd** to one of the supported third-party tools, HDL Coder automatically inserts a tool-specific default termination string in the Tcl script.

Command-Line Information

Property: `HDLLintTerm`

Type: character vector

Default: none

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

See Also

"Generate an HDL Lint Tool Script" on page 26-45

Modeling Guidelines

- “HDL Modeling Guidelines Severity Levels” on page 21-2
- “Model Design and Compatibility Guidelines - By Numbered List” on page 21-3
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 21-6
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 21-10
- “Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 21-12
- “Guidelines for Model Setup and Checking Model Compatibility” on page 21-18
- “Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 21-22
- “Terminate Unconnected Block Outputs and Usage of Commenting Blocks” on page 21-25
- “Identify and Programmatically Change and Display HDL Block Parameters” on page 21-29
- “DUT Subsystem Guidelines” on page 21-34
- “Hierarchical Modeling Guidelines” on page 21-38
- “Design Considerations for Matrices and Vectors” on page 21-44
- “Use Bus Signals to Improve Readability of Model and Generate HDL Code” on page 21-49
- “Guidelines for Clock and Reset Signals” on page 21-55
- “Modeling with Native Floating Point” on page 21-62
- “Design Considerations for RAM Blocks and Blocks in HDL Operations Library” on page 21-65
- “Usage of Blocks in Logic and Bit Operations Library” on page 21-69
- “Generate FPGA Block RAM from Lookup Tables” on page 21-74
- “Usage of Different Subsystem Types” on page 21-77
- “Usage of Rate Change and Constant Blocks” on page 21-82
- “Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation” on page 21-85
- “Modeling Efficient Multiplication and Division Operations for FPGA Targeting” on page 21-87
- “Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation” on page 21-92
- “Guidelines for HDL Code Generation Using Stateflow Charts” on page 21-98
- “Simulink Data Type Considerations” on page 21-103
- “Resource Sharing Settings for Various Blocks” on page 21-105
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 21-109
- “Distributed Pipelining and Clock-Rate Pipelining Guidelines” on page 21-114
- “Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs” on page 21-117

HDL Modeling Guidelines Severity Levels

Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. This table illustrates what each severity level indicates.

Severity Levels

Category	Mandatory	Strongly Recommended	Recommended	Informative
Definition	Guidelines that are absolutely essential to follow. Models created must conform to these guidelines to 100%.	Guidelines that are agreed upon to be a good practice. Models created should conform to these guidelines to the greatest extent possible, but does not have to be 100%.	Guidelines that are recommended to improve the generated code and optimize the code on the target device, but are not critical	Guidelines that are meant to understand some modeling recommendations and best practices.
Impact	If you violate these guidelines, you cannot generate code and synthesize your design on the target hardware.	If you violate these guidelines, you get poor quality of results.	Violating these guidelines may impact the efficiency or ease of using the generated code with downstream synthesis tools	None

See Also

More About

- “Model Design and Compatibility Guidelines - By Numbered List” on page 21-3
- “Guidelines for Supported Blocks and Data Types - By Numbered List” on page 21-6
- “Guidelines for Speed and Area Optimizations - By Numbered List” on page 21-10

Model Design and Compatibility Guidelines - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. The model design and compatibility guidelines consist of guidelines for basic block usage, clock and reset signals, buses and vectors, and subsystem and hierarchical designing. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

These tables list the model design and compatibility guidelines in HDL Coder. These guidelines start from 1.1 and are divided into subsections. In the table, you see that certain guidelines have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

Guidelines 1.1: Basic Settings

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.1.1	“Use HDL-Supported Blocks” on page 21-12	Recommended	None
1.1.2	“Partition Model into DUT and Test Bench” on page 21-13	Recommended	None
1.1.3	“Avoid Using Double-Byte Characters” on page 21-15	Mandatory	None
1.1.4	“Document Model Features and Attributes” on page 21-15	Recommended	None
1.1.5	“Customize <code>hdlsetup</code> Function Based on Target Application” on page 21-18	Recommended	Model Check: “Check for model parameters suited for HDL code generation” on page 38-5
1.1.6	“Check Subsystem for HDL Compatibility” on page 21-19	Recommended	None
1.1.7	“Run Model Checks for HDL Coder” on page 21-19	Recommended	None
1.1.8	“Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 21-22	Informative	None
1.1.9	“Terminate Unconnected Block Outputs” on page 21-25	Mandatory	None
1.1.10	“Using Comment Out and Comment Through of Blocks” on page 21-26	Informative	None
1.1.11	“Adjust Sizes of Constant and Gain Blocks for Identifying Parameters” on page 21-29	Recommended	None

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.1.12	"Display Parameters that Affect HDL Code Generation" on page 21-29	Recommended	None
1.1.13	"Change Block Parameters by Using <code>find_system</code> and <code>set_param</code> " on page 21-33	Informative	None

Guidelines 1.2: DUT Subsystem and Hierarchical Modeling

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.2.1	"DUT Subsystem Considerations" on page 21-34	Strongly Recommended	Model Check: "Check for invalid top level subsystem" on page 38-13
1.2.2	"Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks" on page 21-34	Strongly Recommended	None
1.2.3	"Insert Handwritten Code into Simulink Modeling Environment" on page 21-36	Informative	None
1.2.4	"Avoid Constant Block Connections to Subsystem Port Boundaries" on page 21-38	Mandatory	None
1.2.5	"Generate Parameterized HDL Code for Constant and Gain Blocks" on page 21-39	Recommended	None
1.2.6	"Place Physical Signal Lines Inside a Subsystem" on page 21-41	Mandatory	None

Guidelines 1.3: Guidelines for Vectors and Buses

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.3.1	"Modeling Requirements for Matrices" on page 21-44	Mandatory	Model Check: "Check for large matrix operations" on page 38-18
1.3.2	"Avoid Generating Ascending Bit Order in HDL Code From Vector Signals" on page 21-46	Strongly Recommended	None

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.3.3	"Use Bus Signals to Improve Readability of Model and Generate HDL Code" on page 21-49	Informative	None

Guidelines 1.4: Guidelines for Clock Bundle Signals

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.4.1	"Use Global Oversampling to Create Frequency-Divided Clock" on page 21-55	Informative	Model Check: "Check for invalid top level subsystem" on page 38-13
1.4.2	"Create Multirate Model with Integer Clock Multiples by Clock Division" on page 21-55	Mandatory	None
1.4.3	"Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times" on page 21-57	Mandatory	None
1.4.4	"Asynchronous Clock Modeling in HDL Coder" on page 21-58	Recommended	None
1.4.5	"Use Global Reset Type Setting Based on Target Hardware" on page 21-60	Strongly Recommended	Model check: "Use Global Reset Type Setting Based on Target Hardware" on page 21-60

Guidelines 1.5: Modeling Guidelines for Native Floating Point

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
1.5.1	"Modeling with Native Floating Point" on page 21-62	Recommended	None

See Also

More About

- "Guidelines for Supported Blocks and Data Types - By Numbered List" on page 21-6
- "Guidelines for Speed and Area Optimizations - By Numbered List" on page 21-10

Guidelines for Supported Blocks and Data Types - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. The guidelines for supported blocks and data types consist of guidelines for using various blocks in the HDL Coder block library, and about the supported data types. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

These tables list the guidelines for supported data types in HDL Coder and for various blocks in the HDL Coder block library. The guidelines start from 2.1 and are divided into subsections. In the table, you see that certain guidelines have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

Guidelines 2.1: Blocks in HDL RAMs and HDL Operations Library

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.1.1	“RAM Block Access Considerations” on page 21-65	Recommended	None
2.1.2	“Serial to Parallel Conversion” on page 21-67	Recommended	None

Guidelines 2.2: Blocks in Logic and Bit Operations Library

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.2.1	“Logical and Arithmetic Bit Shift Operations” on page 21-69	Informative	None
2.2.2	“Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks” on page 21-71	Informative	None
2.2.3	“Use Boolean Output for Compare to Constant and Relational Operator Blocks” on page 21-73	Strongly Recommended	Model Check: “Check for Relational Operator block usage” on page 38-29

Guidelines 2.3: Lookup Table Blocks

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.3.1	“Generate FPGA Block RAM from Lookup Tables” on page 21-74	Strongly Recommended	None

Guidelines 2.4: Ports and Subsystems

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.4.1	"Virtual Subsystem: Use as DUT" on page 21-77	Mandatory	Model Check: "Check for invalid top level subsystem" on page 38-13
2.4.2	"Atomic Subsystem: Generate Reusable HDL Files" on page 21-77	Recommended	None
2.4.3	"Variant Subsystem: Using Variant Subsystems for HDL Code Generation" on page 21-78	Mandatory	None
2.4.4	"Model References: Build Model Design Using Smaller Partitions" on page 21-79	Recommended	None
2.4.5	"Block Settings of Enabled and Triggered Subsystems" on page 21-80	Mandatory	Model check: "Check initial conditions of enabled and triggered subsystems" on page 38-14

Guideline 2.5: Rate Change and Constant Blocks

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.5.1	"Usage of Rate Conversion Blocks" on page 21-82	Recommended	None
2.5.2	"Use Discrete and Finite Sample Time for Constant Block" on page 21-83	Mandatory	Model Check: "Check for infinite and continuous sample time sources" on page 38-16

Guideline 2.6: Delay Blocks

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.6.1	"Appropriate Usage of Delay Blocks as Registers" on page 21-85	Recommended	"Check for obsolete Unit Delay Enabled/Resettable Blocks" on page 38-21
2.6.2	"Required HDL Settings for Goto and From Blocks" on page 21-85	Mandatory	None

Guideline 2.7: Blocks for Multiplication and Accumulation Operations

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.7.1	"Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA" on page 21-87	Strongly Recommended	None
2.7.2	"Use ShiftAdd Architecture of Divide Block for Fixed-Point Types" on page 21-91	Recommended	None

Guideline 2.8: MATLAB Function Blocks

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.8.1	"Update Persistent Variables at End of MATLAB Function" on page 21-92	Strongly Recommended	None
2.8.2	"Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks" on page 21-93	Mandatory	None
2.8.3	"Use hdlfimath Setting and Specify fi Objects inside MATLAB Function Block" on page 21-95	Strongly Recommended	"Check for MATLAB Function block settings" on page 38-19

Guideline 2.9: Stateflow Charts

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.9.1	"Choose State Machine Type based on HDL Implementation Requirements" on page 21-98	Strongly Recommended	None
2.9.2	"Specify Block Configuration Settings of Stateflow Chart" on page 21-98	Strongly Recommended	"Check for Stateflow chart settings" on page 38-20
2.9.3	"Insert Unconditional Transition State for Else Statement in HDL Code" on page 21-99	Recommended	None

Guidelines 2.10: Data Types

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
2.10.1	"Use Boolean for Logical Data and Ufix1 for Numerical Data" on page 21-103	Mandatory	None
2.10.2	"Specify Data Type of Gain Blocks" on page 21-103	Recommended	None
2.10.3	"Enumerated Data Type Restrictions" on page 21-104	Mandatory	None

See Also

More About

- "Model Design and Compatibility Guidelines - By Numbered List" on page 21-3
- "Guidelines for Speed and Area Optimizations - By Numbered List" on page 21-10

Guidelines for Speed and Area Optimizations - By Numbered List

The HDL modeling guidelines are a set of recommended guidelines that you can follow when creating Simulink model for code generation with HDL Coder. In addition to providing architectural guidance, because the generated code targets hardware platforms such as FPGAs, ASICs, and SoCs, you can use these guidelines to optimize your design for speed or area on the target hardware.. Each modeling guideline for HDL code generation has a different level of severity that indicates the levels of compliance requirements. To learn more about these severity levels, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

These tables list the guidelines for speed and area optimizations in HDL Coder. The guidelines start from 3.1 and are divided into subsections. These guidelines do not have an associated model check. You can follow the modeling pattern recommended for these guidelines by running that check in the HDL Code Advisor. To learn more about the HDL Code Advisor, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

Guidelines 3.1: Resource Sharing

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
3.1.1	“Resource Sharing of Add Blocks” on page 21-105	Recommended	None
3.1.2	“Resource Sharing of Gain Blocks” on page 21-106	Recommended	None
3.1.3	“Resource Sharing of Product Blocks” on page 21-107	Recommended	None
3.1.4	“Resource Sharing of Multiply-Add Blocks” on page 21-107	Recommended	None
3.1.5	“General Considerations for Sharing of Subsystems” on page 21-109	Recommended	None
3.1.6	“Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks” on page 21-110	Recommended	None
3.1.7	“Sharing of Atomic Subsystems” on page 21-110	Recommended	None
3.1.8	“Resource Sharing of Floating-Point IPs” on page 21-112	Recommended	None

Guidelines 3.2: Clock Rate Pipelining and Distributed Pipelining

Guideline ID	Title	Severity	Associated Model Check/Coding Standard Rule
3.2.1	"Clock-Rate Pipelining Guidelines" on page 21-114	Informative	None
3.2.2	"Recommended Distributed Pipelining Settings" on page 21-114	Recommended	None
3.2.3	"Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs" on page 21-117	Informative	None

See Also

More About

- "Model Design and Compatibility Guidelines - By Numbered List" on page 21-3
- "Guidelines for Supported Blocks and Data Types - By Numbered List" on page 21-6

Basic Guidelines for Modeling HDL Algorithm in Simulink

In this section...

- “Use HDL-Supported Blocks” on page 21-12
- “Partition Model into DUT and Test Bench” on page 21-13
- “Avoid Using Double-Byte Characters” on page 21-15
- “Document Model Features and Attributes” on page 21-15

Use these guidelines to develop your HDL algorithm in Simulink. The guidelines include using HDL-supported blocks when modeling your design and how to partition your design when developing the algorithm.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Use HDL-Supported Blocks

Guideline ID

1.1.1

Severity

Strongly Recommended

Description

When you create your Simulink model, use blocks from the **Simulink Library Browser > HDL Coder** library. Several blocks in this library are pre-configured for HDL code generation. Blocks in this library are available with Simulink. If you do not have HDL Coder, you can simulate the blocks in your model, but cannot generate HDL code.

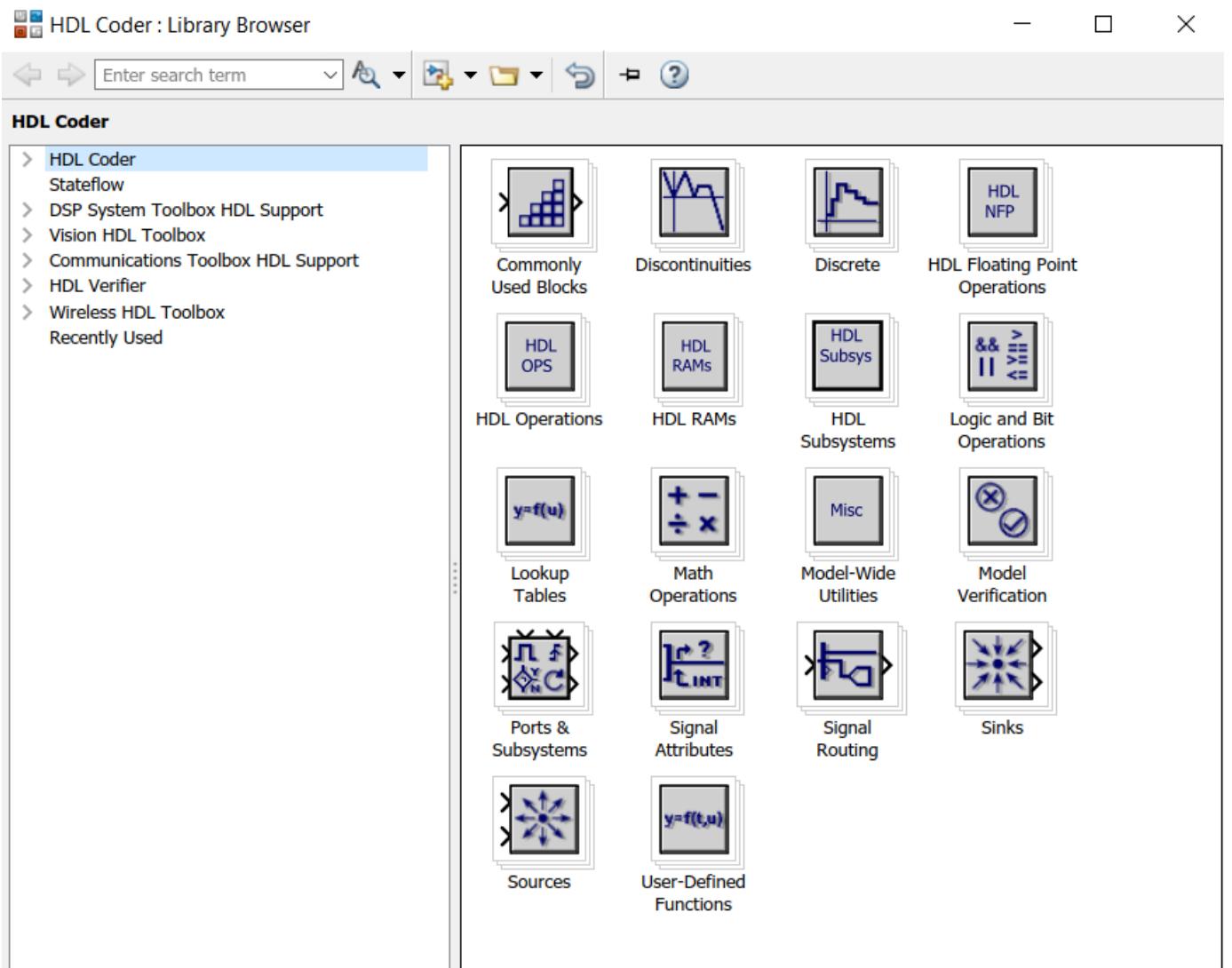
You can find additional HDL-supported blocks in these Simulink block libraries:

- **DSP System Toolbox HDL Support**
- **Communications Toolbox HDL Support**
- **Vision HDL Toolbox**
- **Wireless HDL Toolbox**

To display only HDL-supported blocks in the Library Browser:

- in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **HDL Block Properties > Open HDL Block Library**.
- Alternatively, at the MATLAB Command Window, enter `hdllib`.

`hdllib`



To restore the library browser to the default view, enter this command:

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

Note The set of supported blocks will change in future releases, so you should rebuild your supported blocks library each time you install a new version of this product.

Partition Model into DUT and Test Bench

Guideline ID

1.1.2

Severity

Recommended

Description

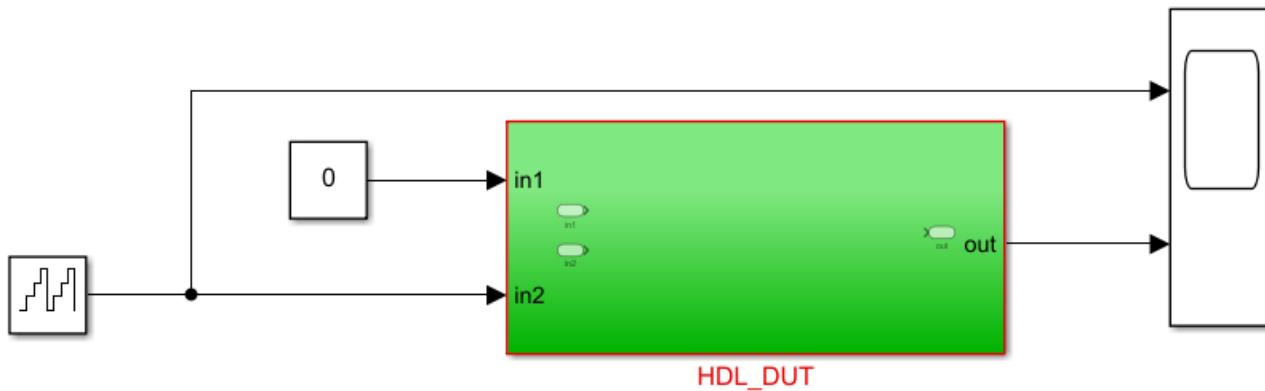
When you create your Simulink model for HDL code generation, the Subsystem that you want to generate HDL code for is the Design-Under-Test (DUT). This Subsystem contains Simulink blocks that can be implemented on your target FPGA or ASIC device. You can further partition the logic inside the DUT into smaller subsystems based on functionality, sample rates in your design, and so on. When you generate HDL code, the DUT becomes the top-level module or entity, and the Subsystems inside the DUT become submodules or smaller entities.

Blocks outside the DUT Subsystem become part of the test bench. The test bench can consist of blocks that are not supported for HDL code generation. Simulate the test bench to:

- Verify the functionality of the DUT in your Simulink model.
- Verify functional equivalence of the generated model with your original model.

For example, if you open the Simulink model template **Blank_DUT**, this model opens in the Simulink Editor.

Note: This model is configured with '[hdlsetup](#)'



Add your design targeted for ASIC/FPGA inside **HDL_DUT** and then run the following command:
[makehdl\('HDL_DUT'\)](#)

In this model, **HDL_DUT** Subsystem is the DUT and blocks outside this Subsystem form the test bench. You can develop your HDL algorithm inside the **HDL_DUT** Subsystem. This template model is preconfigured for HDL code generation.

Note You can also generate HDL code for the entire model instead of the DUT Subsystem. Replace the input signals and Constant blocks with Import blocks. Replace the output signals and Scope blocks with Outport blocks.

Avoid Using Double-Byte Characters

Guideline ID

1.1.3

Severity

Strongly Recommended

Description

Downstream synthesis and simulation tools do not support double-byte characters such as Japanese and Chinese characters. HDL Coder does not support using:

- Double-byte characters in model and block names.
- Reserved words of your Operating System in model and block names such as CR, con, prn, aux, ptr, null, ipt1, ipt2, ipt3, and ipt4, com1, com2, com3, and com4.
- Double-byte characters in comments because the comments are propagated to the generated code. Use English comments instead.

Document Model Features and Attributes

Guideline ID

1.1.4

Severity

Recommended

Description

To make the generated HDL code easier to manage, you can document reference information as part of your model settings in these ways:

- **Custom File Headers and Footer Comments in HDL Code for Design and Testbench**

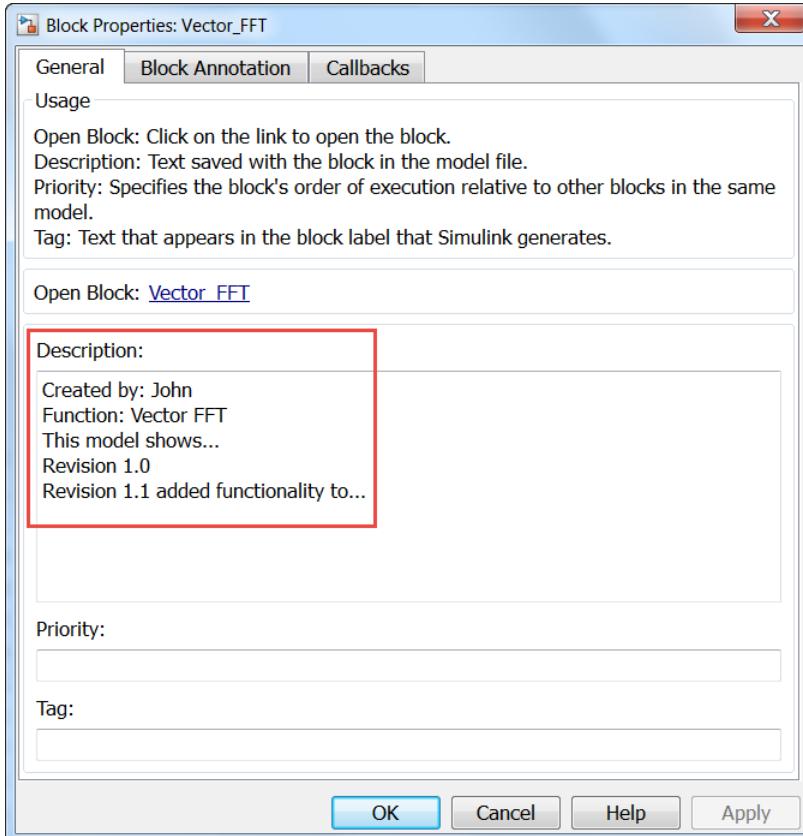
In the **HDL Code Generation > Global Settings > Coding Style** tab of the Configuration Parameters dialog box, by using the **Custom File Header Comment** and **Custom File Footer Comment** parameters, you can enter your own custom comments to appear as headers or footers in all generated HDL files. To learn more, see “File Comment Customization Parameters” on page 17-62.

- **Model and Block Annotations, Text Comments, and Requirement Comments**

You can add annotations in the form of model annotations, text comments, or requirement comments to the generated code. For example, you can enter text directly on the block diagram as Simulink annotations, or insert text comments by placing a DocBlock in your model. To relate annotations in the block diagram to blocks in your model, use lines to connect the annotations to those blocks. These annotations appear as comments beside the blocks in the generated code. To learn more, see “Generate Code with Annotations or Comments” on page 25-13.

- **Block Features and Attributes as Custom Header Comments for Each File**

In the **Description** section of the Block Properties for subsystems that you use in your design. This information appears as comment headers in the HDL code. For example, this figure illustrates block comments added for a **Vector FFT** Subsystem in your design.



The block comments appear as headers in the generated HDL code.

```
-- Simulink subsystem description for vector_fft_implementation_example/Vector_FFT:
--
-- Created by: John
-- Function: Vector FFT
-- This model shows...
-- Revision 1.0
-- Revision 1.1 added functionality to...
--
--
-----
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Vector_FFT IS
```

See Also

Functions

`checkhdl` | `hdllib` | `hdlmodelchecker`

Modeling Guidelines

“Guidelines for Model Setup and Checking Model Compatibility” on page 21-18 | “Modeling with Simulink, Stateflow, and MATLAB Function Blocks” on page 21-22

More About

- “Use Simulink Templates for HDL Code Generation” on page 10-7
- “Create HDL-Compatible Simulink Model”
- “Show Supported Blocks in Library Browser” on page 25-18

Guidelines for Model Setup and Checking Model Compatibility

In this section...

- “Customize `hdlsetup` Function Based on Target Application” on page 21-18
- “Check Subsystem for HDL Compatibility” on page 21-19
- “Run Model Checks for HDL Coder” on page 21-9

Use these guidelines to setup your Simulink model for HDL code generation compatibility and verify that your design is ready to generate code.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Customize `hdlsetup` Function Based on Target Application

Guideline ID

1.1.5

Severity

Strongly Recommended

Description

Before generating code, you must configure the model. To configure the model, you can use the `hdlsetup` function. The `hdlsetup` function uses the `set_param` function to set up models for HDL code generation. The settings include using a fixed-step discrete solver, specifying ASIC/FPGA as the hardware type, and so on. To see the settings that `hdlsetup` function saves on the model, run this command:

```
edit hdlsetup.m
```

Some of the settings that the `hdlsetup` function saves on the model may not be suitable for your target application. In such cases, you can customize the `hdlsetup.m` file such that it runs only those commands required for your target application. For example, you can disable some of the solver settings in the Configuration Parameters and instead enable certain model parameters such as displaying port data types, and so on.

```
% following config parameters are disabled.
%     'Solver',           'fixedstepdiscrete', ...
%     'SaveTime',         'off', ...
%     'SaveOutput',       'off', ...
%     'DataTypeOverride', 'ForceOff', ...

% Following model parameters are enabled.
set_param(model, 'ShowLineDimensions', 'on')
set_param(model, 'ShowPortDataTypes', 'on')
set_param(model, 'SampleTimeColors', 'on')
set_param(model, 'WideLines', 'on')
```

To see a custom `hdlsetup` function that consists of these commands and specifies some of the HDL-specific settings required for HDL code generation, open the file `myhdlsetup.m`.

```
edit myhdlsetup.m
```

You see that this custom `myhdlsetup` file also saves some HDL-specific parameters by using `hdlset_param` on the model.

Check Subsystem for HDL Compatibility

Guideline ID

1.1.6

Severity

Strongly Recommended

Description

The compatibility checker generates a report specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, and so on.

To run the check for HDL compatibility:

- From the UI, right-click the DUT Subsystem and select **HDL Code > Check Subsystem for HDL compatibility**.
- At the command line, use the `checkhdl` function. Select the DUT Subsystem and then enter this command:

```
checkhdl(gcb)
```

See also “Check Your Model for HDL Compatibility” on page 25-16.

When you run this command, the HDL compatibility checker generates an HDL Code Generation Check Report. The report is stored in the target `hdlsrc` folder. If the report does not display any errors, it indicates that your model is compatible for HDL code generation.

```
### Starting HDL Check.
### HDL Check Complete with 0 errors, warnings and messages.
```

Note `checkhdl` does not detect all compatibility issues. Even if HDL check completes without any errors or warnings, HDL Coder can generate errors during code generation.

Run Model Checks for HDL Coder

Guideline ID

1.1.7

Severity

Strongly Recommended

Description

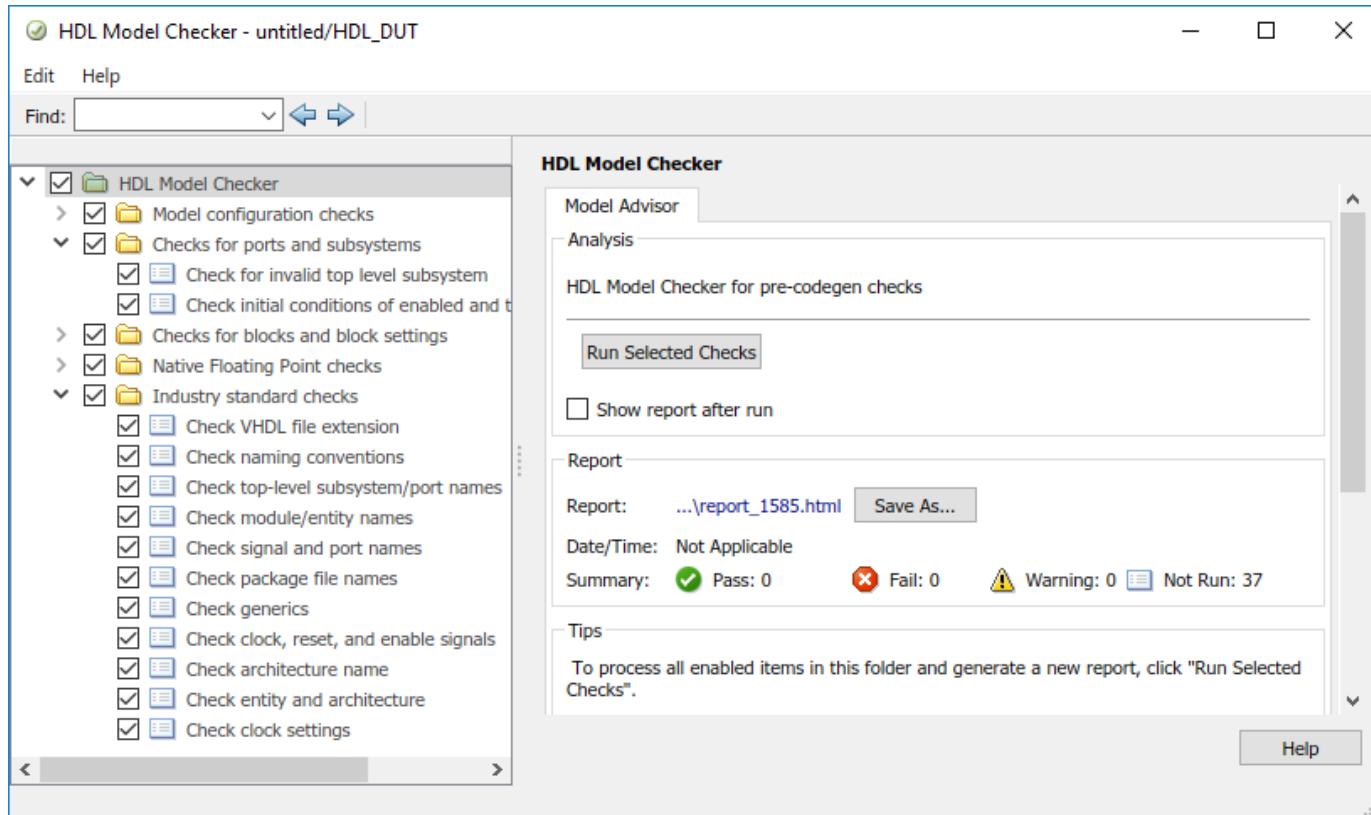
To see whether your DUT Subsystem is compatible for HDL code generation, run the checks in the HDL Code Advisor or the Simulink Model Advisor checks for **HDL Coder**.

To open the HDL Code Advisor:

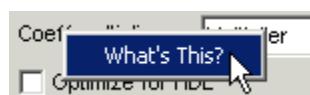
- From the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the DUT Subsystem and then click **HDL Code Advisor**.
- To run the model checks for the Subsystem you want to analyze, right-click that Subsystem, and in the context menu, select **HDL Code > Check Model Compatibility**.
- At the command line, use the `hdlmodelchecker` function:

```
hdlmodelchecker(gcb)
```

When you run this command, the HDL Code Advisor appears.



You may not have to run all checks in the HDL Code Advisor. For example, if your model does not have single or double data types, you do not have to run the checks in the **Native Floating Point checks** folder. To learn more about each check and whether to run the check for your model, right-click that check and select **What's This?**.



See Also

Functions

`checkhdl | hdllib | hdlmodelchecker`

Modeling Guidelines

“Basic Guidelines for Modeling HDL Algorithm in Simulink” on page 21-12

More About

- “Use Simulink Templates for HDL Code Generation” on page 10-7
- “Create HDL-Compatible Simulink Model”
- “Show Supported Blocks in Library Browser” on page 25-18

Modeling with Simulink, Stateflow, and MATLAB Function Blocks

In this section...

[“Guideline ID” on page 21-22](#)

[“Severity” on page 21-22](#)

[“Description” on page 21-22](#)

You can follow this guideline as a general practice for modeling your design with various blocks in the Simulink Library Browser.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see [“HDL Modeling Guidelines Severity Levels” on page 21-2](#).

Guideline ID

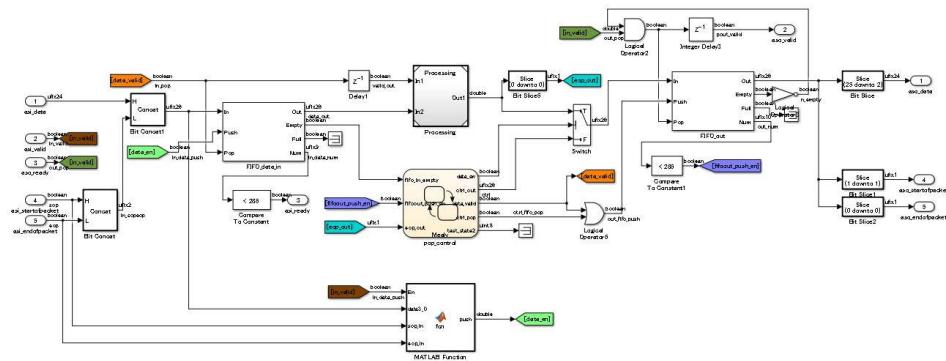
1.1.8

Severity

Informative

Description

When you create a Simulink model for HDL code generation, use Simulink blocks, MATLAB Function blocks, and Stateflow blocks based on the application. This figure shows an example of how you can use the various blocks inside your DUT.



Simulink Blocks

Use Simulink blocks to model arithmetic algorithms that perform numerical processing or contains feedback loops.

MATLAB Function Blocks

Use MATLAB Function blocks to model the control logic, conditional branches such as if-else statements, and simple state machines. You can also use MATLAB Function blocks to model an IP that is written using MATLAB code.

Stateflow Blocks

Use these Stateflow blocks to model your algorithm:

- State Transition Table: Use these blocks to model state machines that control the output using knowledge of the past and the present.
- Chart: Use these blocks to model flow charts using conditional if-else branches and state machines that control the output using knowledge of the past and the present.
- Truth Table: Use these blocks to model conditional if-else branches.

You can model combinational logic using Stateflow blocks. For more complex operations and operations that change timing such as pipeline insertion and processing, use Simulink blocks. You can then use the Stateflow logic to process the result calculated from the Simulink blocks

Model References

For significantly large algorithms that have complex computations, you can partition the design into a hierarchy of smaller designs. Use this partitioning for reuse, modular development, and accelerated simulation. You can reuse models by including them as Model blocks inside a top model. The model that reuses this block is called the top model and the block that is reused or included in the top model is called the referenced model.

Note When you generate HDL code for a Subsystem that is not at the top level of the model, HDL Coder converts the Subsystem to a model reference.

A referenced model is treated similar to an Atomic Subsystem. In some cases, an algebraic loop can potentially occur, and can prevent HDL code generation. To generate code, either remove the algebraic loop in your design, or, in the Configuration Parameters dialog box, specify the **Minimize algebraic loop occurrences** setting.

BlackBox Subsystems

For subsystems that you want to simulate in your design and to include the HDL code that you authored, use BlackBox subsystems. To create a **BlackBox** Subsystem, set the HDL Architecture of a Subsystem or Model reference to **BlackBox**. You can use this architecture to incorporate handwritten HDL code into a Simulink model. For more information, see “Verify the Combination of Hand-Written and Generated HDL Code” (HDL Verifier).

If you generate a Simulink model using the HDL code that you authored, use HDL import. To learn more, see “Import Verilog Code and Generate Simulink Model” on page 10-127.

HDL Cosimulation Blocks

If you have a HDL simulator such as Mentor Graphics ModelSim or Cadence Incisive, you can use HDL Cosimulation blocks to simulate the HDL code for the DUT and instantiate this HDL code in the generated code.

See Also

Modeling Guidelines

"Basic Guidelines for Modeling HDL Algorithm in Simulink" on page 21-12

More About

- "Verify with HDL Cosimulation" on page 37-13
- "Show Supported Blocks in Library Browser" on page 25-18
- "Design Guidelines for the MATLAB Function Block" on page 29-29
- "Introduction to Stateflow HDL Code Generation" on page 28-2
- "Generate Black Box Interface for Subsystem" on page 27-4
- "Generate Black Box Interface for Referenced Model" on page 27-8

Terminate Unconnected Block Outputs and Usage of Commenting Blocks

You can follow these guidelines as recommended modeling practices such as making sure that block outputs are terminated and how you can comment out blocks for HDL code generation.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Terminate Unconnected Block Outputs

Guideline ID

1.1.9

Severity

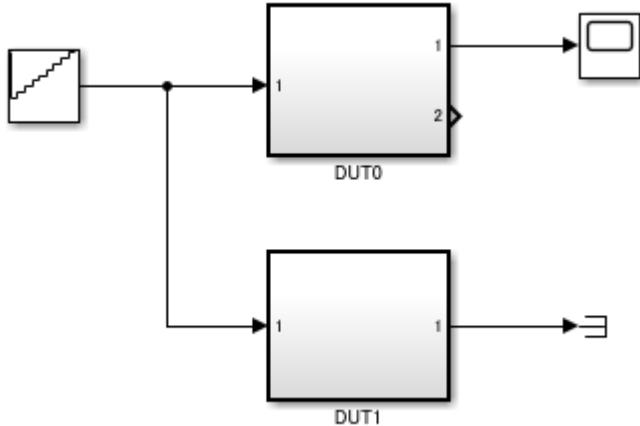
Mandatory

Description

If you generate HDL code for a Subsystem that has unconnected output ports, HDL Coder™ generates an error. For output ports that are not connected to downstream logic, connect them to a Terminator block.

This model illustrates a DUT0 Subsystem that has an unconnected output port Out2.

```
open_system('hdlcoder_terminateout')
```



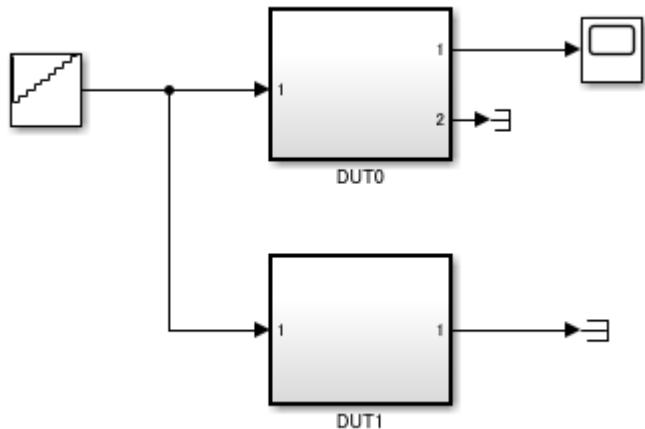
If you generate HDL code for this Subsystem, HDL Coder™ generates this error:

```
error in validation model generation: Failed to find source for outport 2 on
'DUT0' Please create a fully connected subsystem when generating the
cosimulation model.
```

```
close_system('hdlcoder_terminateout')
```

You can use the `addterms` function to add Terminator blocks to unconnected ports in your model.

```
load_system('hdlcoder_terminateout')
addterms('hdlcoder_terminateout')
open_system('hdlcoder_terminateout')
```



Using Comment Out and Comment Through of Blocks

Guideline ID

1.1.10

Severity

Informative

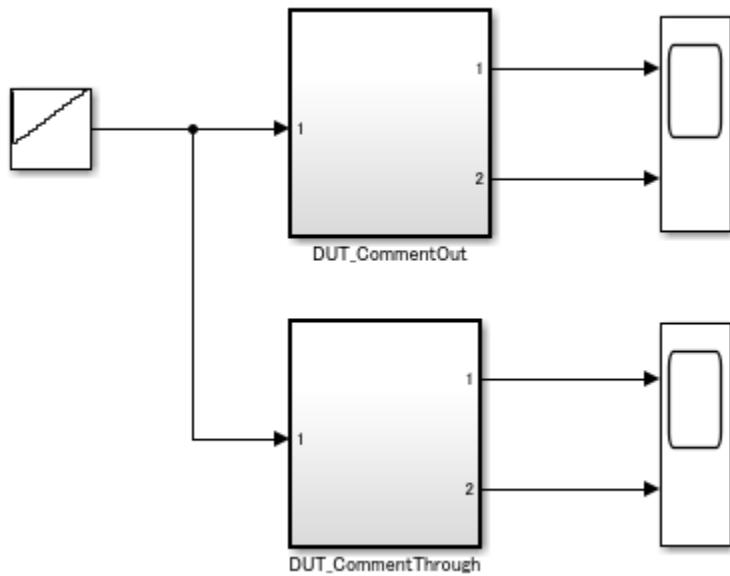
Description

To exclude blocks in your model from simulation without physically removing the blocks from your model, use **Comment Out** or **Comment Through**. When you use **Comment Out**, the signals are terminated and grounded. When you use **Comment Through**, the signals are passed through.

When you generate HDL code, you can use this capability to exclude certain blocks such as blocks that are not supported for HDL code generation.

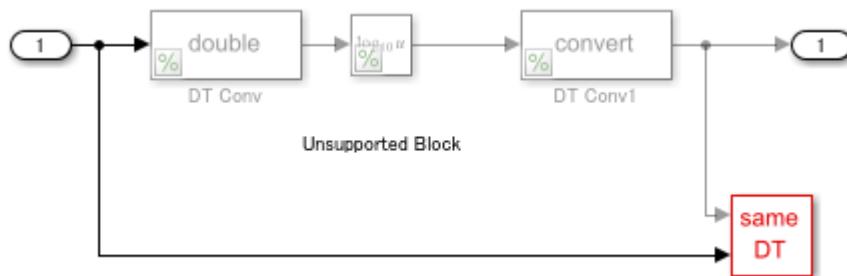
Open the model `hdlcoder_comment_through_out`.

```
open_system('hdlcoder_comment_through_out')
```



The code generator supports blocks that are comment out when the output signals are unused. The generated code assigns a constant value of 0 to the signal at the output. The Dut_CommentOut subsystem contains blocks that are commented out.

```
open_system('hdlcoder_comment_through_out/DUT_CommentOut/Generated_CommentOut')
```



When you generate code, this VHDL code generated for the DUT_CommentOut subsystem indicates a constant zero value assigned to Out1.

```
ARCHITECTURE rtl OF Generated IS
```

```
-- Signals
SIGNAL TmpGroundAtData_Type_DuplicateInport1_out1 : signed(15 DOWNTO 0); -- sfix16_En6

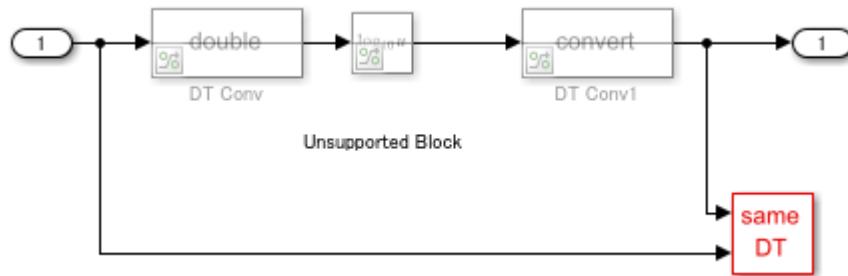
BEGIN
-- Unsupported Block

TmpGroundAtData_Type_DuplicateInport1_out1 <= to_signed(16#0000#, 16);
Out1 <= std_logic_vector(TmpGroundAtData_Type_DuplicateInport1_out1);

END rtl;
```

The code generator supports blocks that are comment through. The generated code passes the input signal through to the output. The `Dut_CommentThrough` subsystem contains blocks that are comment through.

```
open_system('hdlcoder_comment_through_out/DUT_CommentThrough/Generated_CommentThrough')
```



When you generate code for `Dut_CommentThrough` subsystem, the VHDL code shows `In1` passed through to `Out1`.

```
ARCHITECTURE rtl OF Generated_CommentThrough IS
BEGIN
-- Unsupported Block
  Out1 <= In1;
END rtl;
```

See Also

Modeling Guidelines

"Basic Guidelines for Modeling HDL Algorithm in Simulink" on page 21-12

More About

- "Comment Out and Comment Through"

Identify and Programmatically Change and Display HDL Block Parameters

You can follow these guidelines to learn how you can identify block parameters in your design and programmatically update some of the parameters so that the model is compatible for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Adjust Sizes of Constant and Gain Blocks for Identifying Parameters

Guideline ID

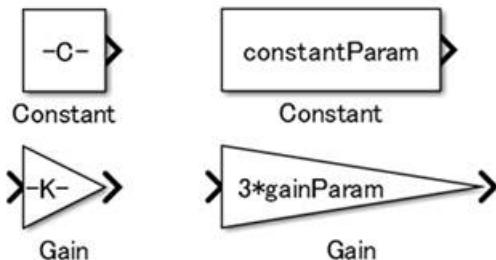
1.1.11

Severity

Recommended

Description

For Constant blocks and Gain blocks that have significantly large values or use parameter values, the **Constant** or **Gain** values may not be visible in the block mask. To increase readability, adjust the size of the block so that the parameter value can be displayed as shown in figure.



Display Parameters that Affect HDL Code Generation

Guideline ID

1.1.12

Severity

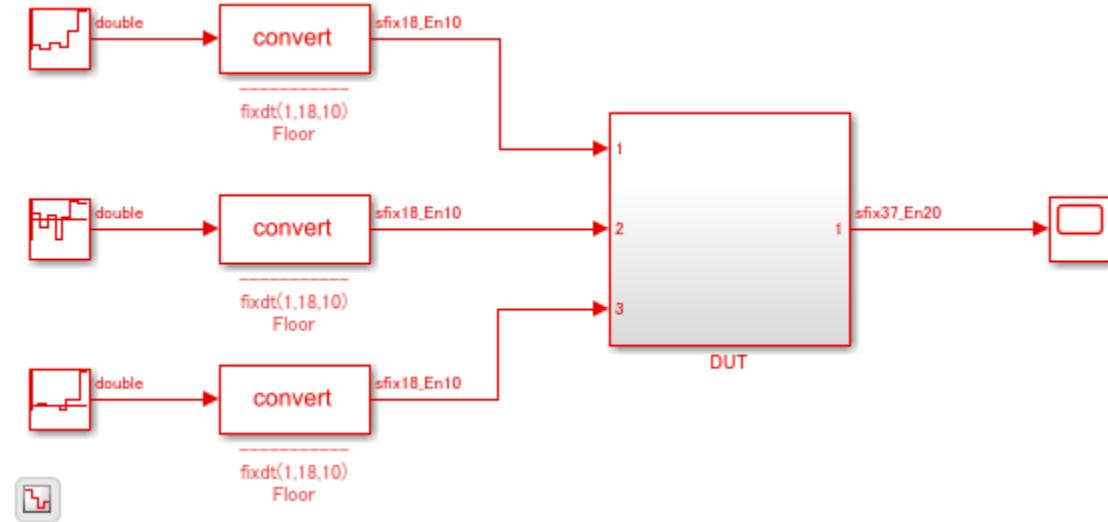
Recommended

Description

Certain HDL block properties such as **Distributed Pipelining** and **Sharing Factor** can significantly affect HDL code generation. If the block properties are enabled for a certain block or Subsystem, it is recommended that you annotate the block properties beside that block in the Simulink™ diagram. When you annotate the model, use delimiters such as `--HDL--` to separate the annotation from the block name.

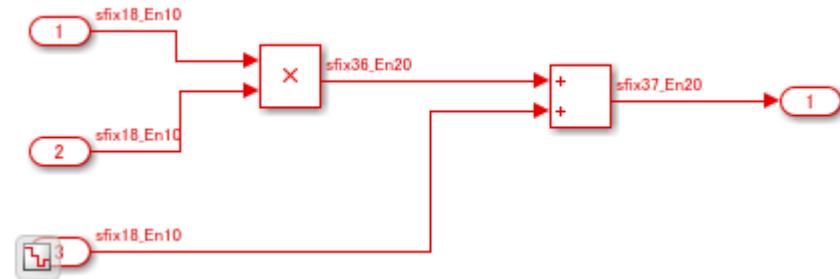
For example, open the model `hdlcoder_block_annotation_HDL_params.slx`.

```
open_system('hdlcoder_block_annotation_HDL_params')
set_param('hdlcoder_block_annotation_HDL_params','SimulationCommand','Update')
```



The DUT Subsystem performs a simple multiply-add operation.

```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



There are HDL block parameters saved on the model. To see the parameters, use the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_block_annotation_HDL_params/DUT')
```

```
% Set Model 'hdlcoder_block_annotation_HDL_params' HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'HierarchicalDistPipelining', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MaskParameterAsGeneric', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'ResourceReport', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_block_annotation_HDL_params', 'Traceability', 'on');
```

```
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'DistributedPipelining', 'on');
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT', 'OutputPipeline', 3);

% Set Sum HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'InputPipeline', 1);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Add', 'OutputPipeline', 1);

% Set Product HDL parameters
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'InputPipeline', 2);
hdlset_param('hdlcoder_block_annotation_HDL_params/DUT/Product', 'OutputPipeline', 1);
```

To annotate the model with the HDL block parameters saved on the model, use the `showHdlBlockParams` script attached with the example.

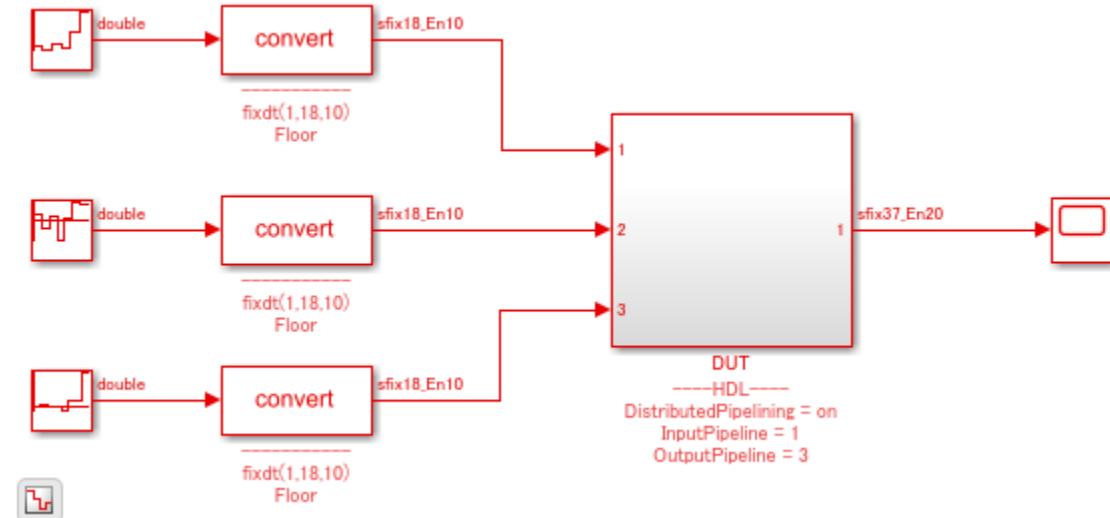
```
showHdlBlockParams('hdlcoder_block_annotation_HDL_params/DUT', 'on')
```

```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT.
----HDL----\nDistributedPipelining = on\nInputPipeline = 1\nOutputPipeline = 3
```

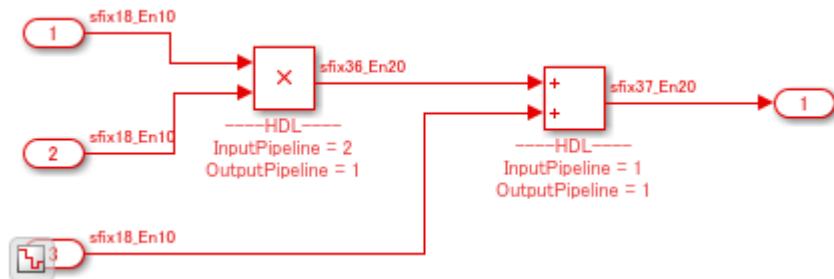
```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT/Add.
----HDL----\nInputPipeline = 1\nOutputPipeline = 1
```

```
Add block annotation for hdlcoder_block_annotation_HDL_params/DUT/Product.
----HDL----\nInputPipeline = 2\nOutputPipeline = 1
```

```
open_system('hdlcoder_block_annotation_HDL_params')
```



```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```

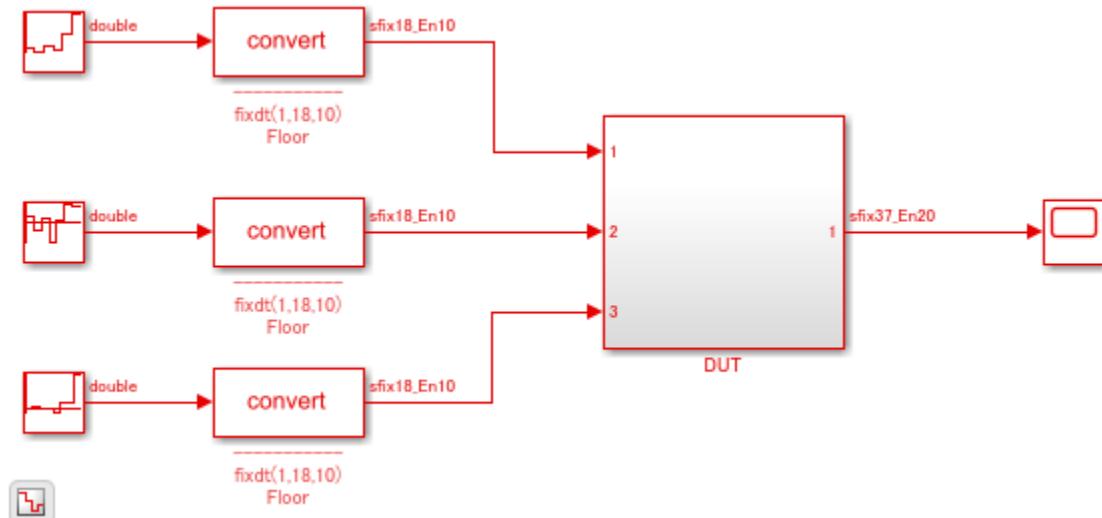


To remove the HDL block parameters annotation from the model, run the `showHdlBlockParams` set to `off`.

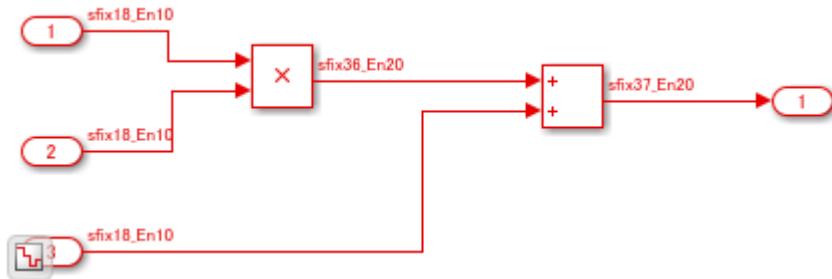
```
showHdlBlockParams('hdlcoder_block_annotation_HDL_params/DUT', 'off')
```

HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In1 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In2 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/In3 are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Add are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Product are removed
HDL block annotations for hdlcoder_block_annotation_HDL_params/DUT/Out1 are removed

```
open_system('hdlcoder_block_annotation_HDL_params')
```



```
open_system('hdlcoder_block_annotation_HDL_params/DUT')
```



Change Block Parameters by Using `find_system` and `set_param`

Guideline ID

1.1.13

Severity

Informative

Description

To modify the parameters of certain blocks, you can use the function `find_system` with the function `set_param`. For example, this script that detects all Constant blocks with a **Sample time** of `inf` and modifies it to `-1`:

```
modelname = 'sfir_fixed';
open_system (modelName)

% Detect all Constant blocks in the model
blockConstant = find_system(bdroot, 'blocktype', 'Constant')

% Detect the Constant blocks with sample time [inf], and change to [-1]
for n = 1:numel(blockConstant)
    sTime = get_param(blockConstant{n}, 'SampleTime')
    if strcmp(lower(sTime), 'inf')
        set_param(blockConstant{n}, 'SampleTime', '-1')
    end
end
```

See Also

Functions

`find_system` | `hdlsaveparams` | `set_param`

More About

- “Specify Block Properties”

DUT Subsystem Guidelines

You can follow these guidelines to learn some best practices on how you can model the DUT for HDL code and testbench generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

DUT Subsystem Considerations

Guideline ID

1.2.1

Severity

Strongly Recommended

Description

The DUT is the Subsystem that contains the algorithm for which you want to generate code. Generally, you specify the top-level Subsystem as the DUT. See also “Partition Model into DUT and Test Bench” on page 21-13.

Consider using these recommended settings when you design the DUT Subsystem for HDL code generation.

- Make sure that the DUT is not a conditionally-executed subsystem, such as an Enabled Subsystem or a Triggered Subsystem. To verify that you are using a valid top-level Subsystem as the DUT, you can run this HDL model check “Check for invalid top level subsystem” on page 38-13.
- Make sure that the **HDL Architecture** of the DUT is not specified as a **BlackBox**. See “BlackBox Subsystems” on page 21-23.
- Connect output signals that are unconnected to a Terminator block. To learn more, see “Terminate Unconnected Block Outputs” on page 21-25.
- For a nontop DUT, specify the DUT as a nonvirtual Subsystem before generating HDL code to avoid numerical mismatches in the simulation results. To learn more, see “Usage of Different Subsystem Types” on page 21-77.

Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks

Guideline ID

1.2.2

Severity

Strongly Recommended

Description

In some cases, parts of the Simulink™ testbench can contain Simscape™ blocks or other blocks from the Simulink library that operate at a continuous sample time. To simulate these blocks, you must

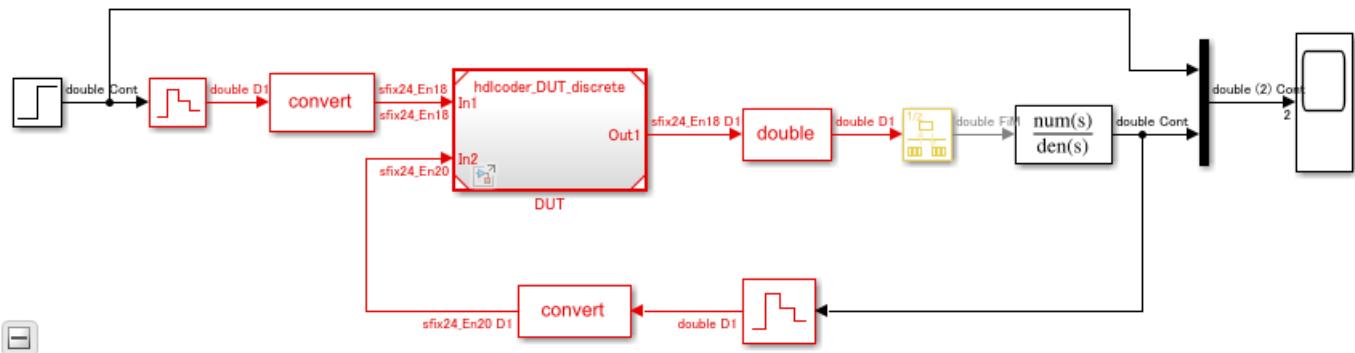
specify a continuous solver setting for your model. The solver settings that you specify applies to all blocks in your model. This means that the DUT Subsystem uses a continuous solver, which is not supported for HDL code generation. To generate HDL code, convert the DUT Subsystem to a model reference, and then use a fixed-step discrete solver for the referenced model. As the parent model and the referenced model use different solver settings, you must convert the sample time by inserting Zero-Order Hold and Rate Transition blocks at the DUT boundary.

For example, open the model `hdlcoder_testbench_continuous.slx`. The model uses `ode45`, which is a continuous solver setting. You see that the DUT is a model reference block. Zero-Order Hold and Rate Transition blocks at the boundary convert the sample time.

```
open_system('hdlcoder_testbench_continuous')
set_param('hdlcoder_testbench_continuous','SimulationCommand','Update')
get_param('hdlcoder_testbench_continuous','Solver')
```

ans =

```
'ode45'
```



To see the referenced model `hdlcoder_DUT_discrete`, double-click the DUT block. You see that the DUT uses a discrete solver setting.

```
open_system('hdlcoder_testbench_continuous/DUT')
get_param('hdlcoder_DUT_discrete','Solver')
```

ans =

```
'FixedStepDiscrete'
```



Insert Handwritten Code into Simulink Modeling Environment

Guideline ID

1.2.3

Severity

Informative

Description

You can reuse a pre-verified RTL IP or insert your handwritten HDL code into the Simulink modeling environment by using these methods:

- **Verilog HDL Import**

If you have handwritten Verilog code, you can import the code into the Simulink environment. The import process generates a Simulink model that is functionally equivalent to your handwritten HDL code.

HDL import supports a subset of Verilog constructs that you can use for importing your design to create the Simulink model. To learn more, see:

- “Supported Verilog Constructs for HDL Import” on page 10-130
- “Limitations of Verilog HDL Import” on page 10-128

- **BlackBox Subsystems**

You can use BlackBox subsystems to insert your handwritten HDL code for a block in your Simulink model. You can then integrate BlackBox subsystems with other blocks in your Simulink model and then generate HDL code.

To make the BlackBox Subsystem compatible with other blocks for HDL code generation and to include this block in your model, create the block in Simulink:

- Name the block by using the same name as the VHDL entity or Verilog module.
- Define the same inputs and outputs, including the same types, sizes, and names.
- Define the same clock, reset, and clock enable signals. A single block can have not more than one clock, reset, and clock enable signal.
- Use a single sample rate for the block.
- Specify the **Architecture** of the block as **BlackBox** in the HDL Block Properties.

To learn more, see “Generate Black Box Interface for Subsystem” on page 27-4 .

- **DocBlock in BlackBox Subsystems**

To keep the HDL code with your model, instead of as a separate file, use a DocBlock to integrate custom HDL code. You can use your own handwritten VHDL or Verilog code as the text in the DocBlock.

You include each DocBlock that contains custom HDL code by placing it in a black box subsystem, and including the black box subsystem in your DUT. One HDL file is generated per black box subsystem. For more information, see “Integrate Custom HDL Code Using DocBlock” on page 27-10.

- **HDL Cosimulation Blocks**

If you have a HDL simulator such as Mentor Graphics ModelSim or Cadence Incisive, you can use HDL Cosimulation blocks to simulate the HDL code for the DUT by using that HDL simulator.

You can simulate the HDL code for the DUT in Simulink and instantiate the HDL code in the generated code for the DUT.

See Also

Functions

`importhdl` | `makehdl` | `makehdltb`

More About

- “Model Referencing for HDL Code Generation” on page 27-2
- “Customize Black Box or HDL Cosimulation Interface” on page 27-12
- “Generate a Cosimulation Model” on page 27-41

Hierarchical Modeling Guidelines

Consider using these recommended settings when you build your model hierarchically and generate HDL code for your design. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Avoid Constant Block Connections to Subsystem Port Boundaries

Guideline ID

1.2.4

Severity

Mandatory

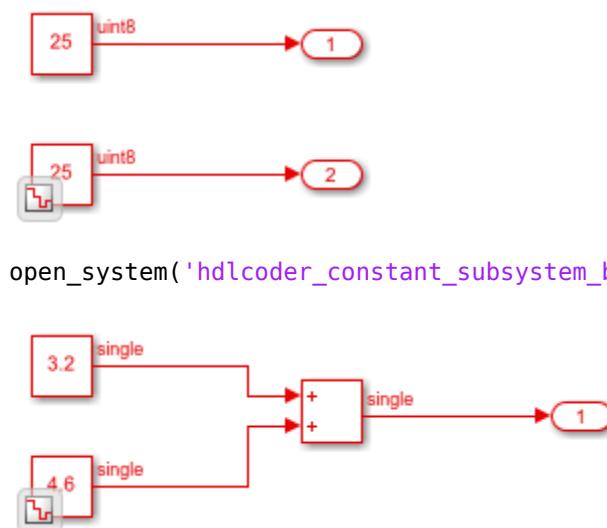
Description

It is recommended that you avoid directly connecting Constant blocks to the output ports of a Subsystem. Synthesis tools may optimize and remove the constants and create unconnected ports.

If you use floating-point data types with the Native Floating Point mode enabled, and input constant values to an arithmetic operator such as an Add block, HDL Coder™ replaces the Add block with a Constant block when generating code. This optimization can result in a Constant block directly connected to the output port. Therefore, it is recommended that you avoid such modeling constructs. See also Simplify Constant Operations and Reduce Design Complexity in HDL Coder.

For example, open the model `hdlcoder_constant_subsystem_boundary.slx`. The DUT contains two subsystems `Constant_subsys1` and `Constant_subsys2`, the outputs of which are inputs to a third Subsystem. `Constant_subsys1` contains Constant blocks directly connected to the output ports, and `Constant_subsys2` contains Constant blocks that have single data types as inputs to an Add block.

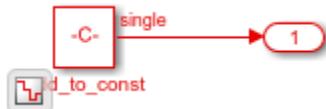
```
load_system('hdlcoder_constant_subsystem_boundary.slx')
set_param('hdlcoder_constant_subsystem_boundary','SimulationCommand','Update')
open_system('hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys1')
```



```
open_system('hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys2')
```

As Constant_subsys2 uses single data types and the model has Native Floating Point mode enabled, when you generate HDL code for the DUT, the Constant_subsys2 becomes a candidate for the optimization that simplifies constant operations. When you open the generated model, you see a Constant block directly connected to the output port.

```
open_system('gm_hdlcoder_constant_subsystem_boundary.slx')
set_param('gm_hdlcoder_constant_subsystem_boundary','SimulationCommand','Update')
open_system('gm_hdlcoder_constant_subsystem_boundary/DUT/Constant_subsys2')
```



Generate Parameterized HDL Code for Constant and Gain Blocks

Guideline ID

1.2.5

Severity

Recommended

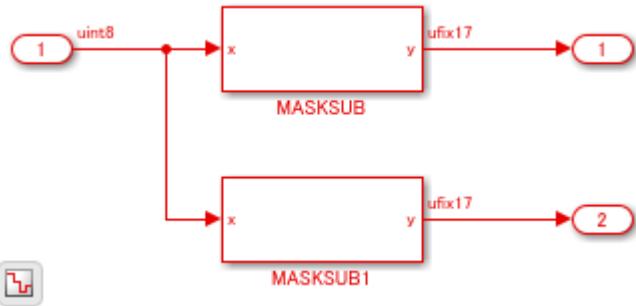
Description

To generate parameterized HDL code for Gain and Constant blocks:

- The Subsystem that contains the Gain and Constant blocks must be a masked subsystem. The Gain and Constant blocks use these mask parameter values. You define mask parameters of the Subsystem in the Mask Editor dialog box.
- The Subsystem that contains the Gain and Constant blocks must be an Atomic Subsystem. To make a Subsystem an Atomic Subsystem, right-click that Subsystem and select **Treat as atomic unit**.
- Enable the Generate parameterized HDL code from masked subsystem setting in the Configuration Parameters dialog box or set **MaskParameterAsGeneric** to on at the command line using **makehdl** or **hdlset_param**.

For an example, open the model **hdlcoder_masked_subsystems**. The Top Subsystem contains two atomic masked subsystems MASKSUB and MASKSUB1 that are similar but for the masked parameter values.

```
load_system('hdlcoder_masked_subsystems')
set_param('hdlcoder_masked_subsystems','SimulationCommand','Update')
open_system('hdlcoder_masked_subsystems/TOP')
```



The model has the **MaskParameterAsGeneric** setting enabled. This setting corresponds to the **Generate parameterized HDL code from masked subsystem** setting that is enabled at the command line.

```
hdlsaveparams('hdlcoder_masked_subsystems')
```

```
%% Set Model 'hdlcoder_masked_subsystems' HDL parameters
hdlset_param('hdlcoder_masked_subsystems', 'HDLSubsystem', 'hdlcoder_masked_subsystems/TOP');
hdlset_param('hdlcoder_masked_subsystems', 'MaskParameterAsGeneric', 'on');
```

To generate VHDL code for the Top Subsystem, run this command:

```
makehdl('hdlcoder_masked_subsystems/TOP')
```

In the generated code, you see that HDL Coder™ generates one HDL file MaskedSub with the different masked parameters mapped to generic ports.

```
-----
-- File Name: hdlsrc\hdlcoder_masked_subsystems\TOP.vhd
-- Created: 2018-10-08 13:30:02
--
-- Generated by MATLAB 9.6 and HDL Coder 3.13
--
-- 
ARCHITECTURE rtl OF TOP IS
    -- Component Declarations
COMPONENT MASKSUB
    GENERIC(
        m          : integer;
        b          : integer
    );
    PORT(
        x          : IN  std_logic_vector(7 DOWNTO 0); -- uint8
        y          : OUT std_logic_vector(16 DOWNTO 0) -- ufix17
    );
END COMPONENT;
    -- Component Configuration Statements
FOR ALL : MASKSUB
```

```

USE ENTITY work.MASKSUB(rtl);

-- Signals
SIGNAL MASKSUB_out1          : std_logic_vector(16 DOWNTO 0); -- ufix17
SIGNAL MASKSUB1_out1          : std_logic_vector(16 DOWNTO 0); -- ufix17

BEGIN
  u_MASKSUB : MASKSUB
    GENERIC MAP( m => 5,
                  b => 2
                )
    PORT MAP( x => In1, -- uint8
              y => MASKSUB_out1 -- ufix17
            );

  u_MASKSUB1 : MASKSUB
    GENERIC MAP( m => 6,
                  b => 4
                )
    PORT MAP( x => In1, -- uint8
              y => MASKSUB1_out1 -- ufix17
            );

  Out1 <= MASKSUB_out1;
  Out2 <= MASKSUB1_out1;

END rtl;

```

Place Physical Signal Lines Inside a Subsystem

Guideline ID

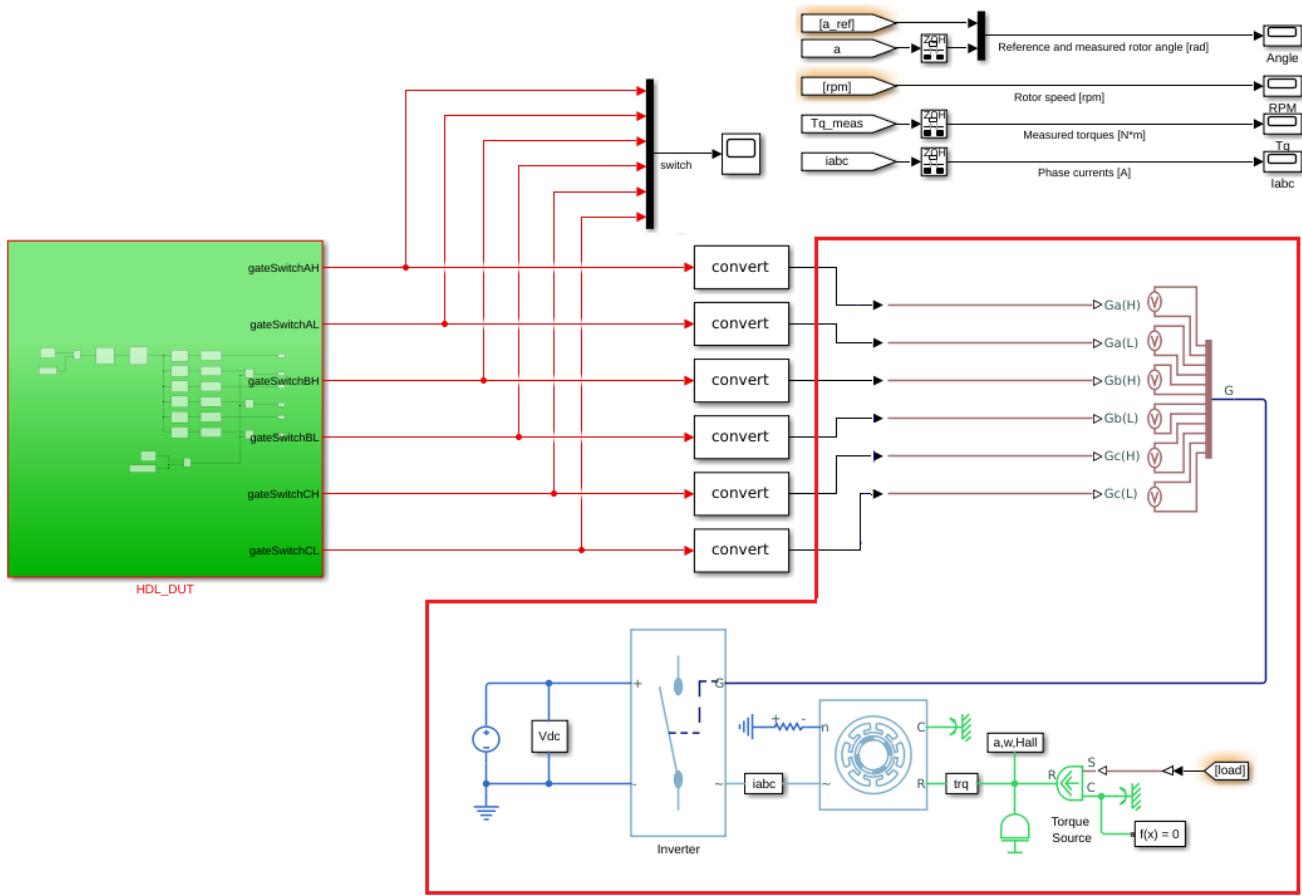
1.2.6

Severity

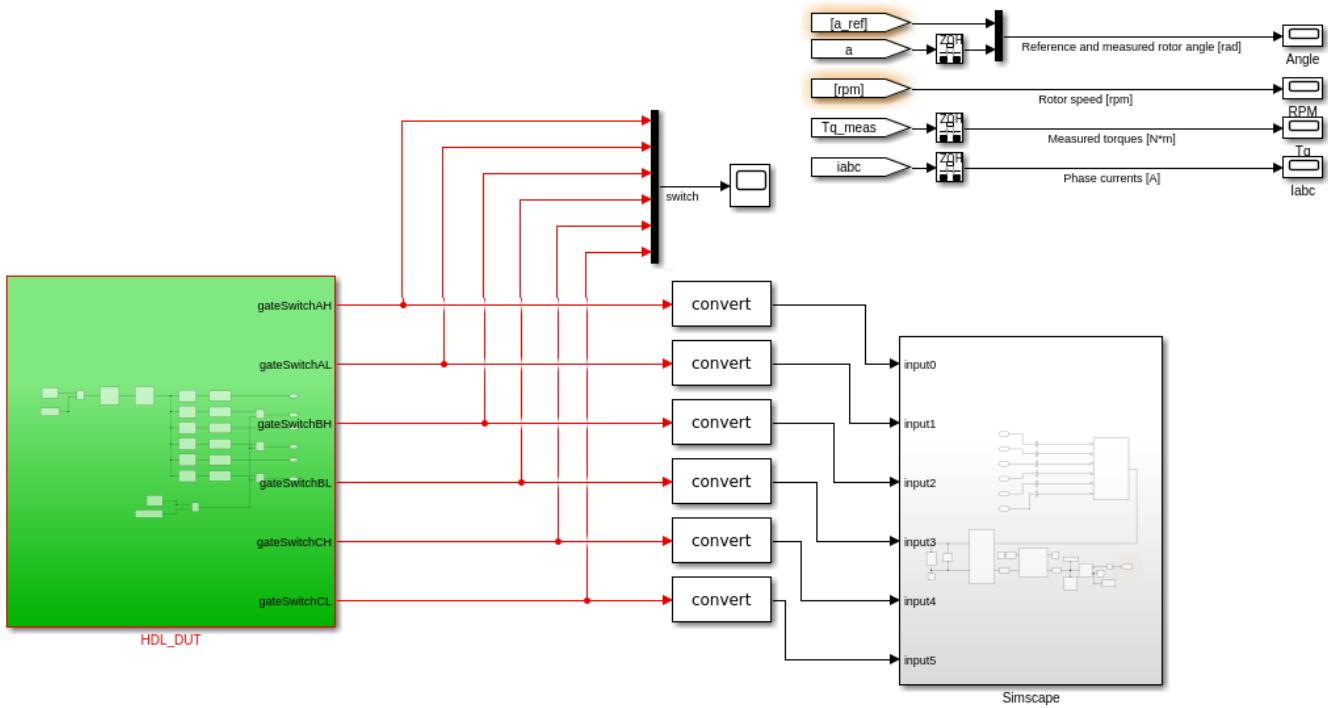
Mandatory

Description

To avoid errors when generating HDL test bench, physical signal lines that are present at the same level as the DUT subsystem must be placed inside a Subsystem block. For example, consider this Simulink model that has physical signal lines outside the DUT subsystem, `HDL_DUT`.



Place the physical signal lines and the blocks connected to it that are highlighted inside a Subsystem block. You can then generate HDL code and test bench for the DUT subsystem.



See Also

Functions

`makehdl` | `makehdltb`

More About

- “Generate Parameterized Code for Referenced Models” on page 10-20
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-17
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166

Design Considerations for Matrices and Vectors

These guidelines recommend how you can use matrix and vector signals when modeling your design for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Modeling Requirements for Matrices

Guideline ID

1.3.1

Severity

Mandatory

Description

HDL Coder™ does not support matrix data types at the DUT interfaces. Before the 2-D matrix signals enter the DUT Subsystem, convert the signals to 1-D vectors by using a Reshape block. Inside the DUT Subsystem, you can convert the vectors back to matrices by using Reshape blocks, and then perform matrix computations. After performing computations, you must convert the matrices back to vector signals at the DUT output interface. Outside the DUT interface, you can convert the vector signals back to matrices.

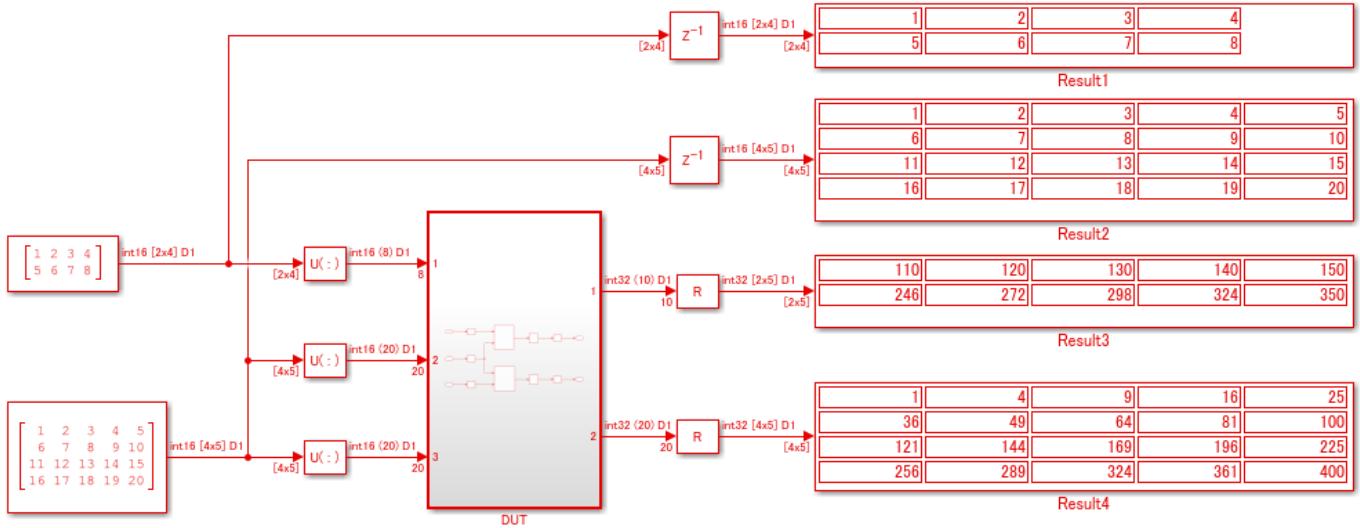
Modeling Considerations

- When you use the Reshape block to convert vectors to 2D matrices, make sure that you specify the right **Output dimensions**.
- When you use the Product block, use the right **Multiplication** mode. By using this mode, you can perform either matrix multiplication or element-wise multiplication. The multiplied output can have different dimensions depending on the **Multiplication** mode.
- When you use the Product block to perform matrix multiplication, place the Matrix Multiply block inside a Subsystem block. When you generate code and open the generated model, you see that HDL Coder expands the matrix multiplication to multiple Product and Add blocks. Placing the Matrix Multiply block inside a subsystem makes the generated model easier to understand. In addition, make sure that you do not provide more than two inputs to the Matrix Multiply block.
- When you extract matrix data, use Selector and Assignment blocks. Make sure that you do not use fixed-point data types for the index input ports for the blocks.

Example

This example shows how to use matrix types in HDL Coder™. Open this model `hdlcoder_matrix_multiply`. The model contains a Reshape block that converts the matrix input to a 1-D vector at the DUT Subsystem interface.

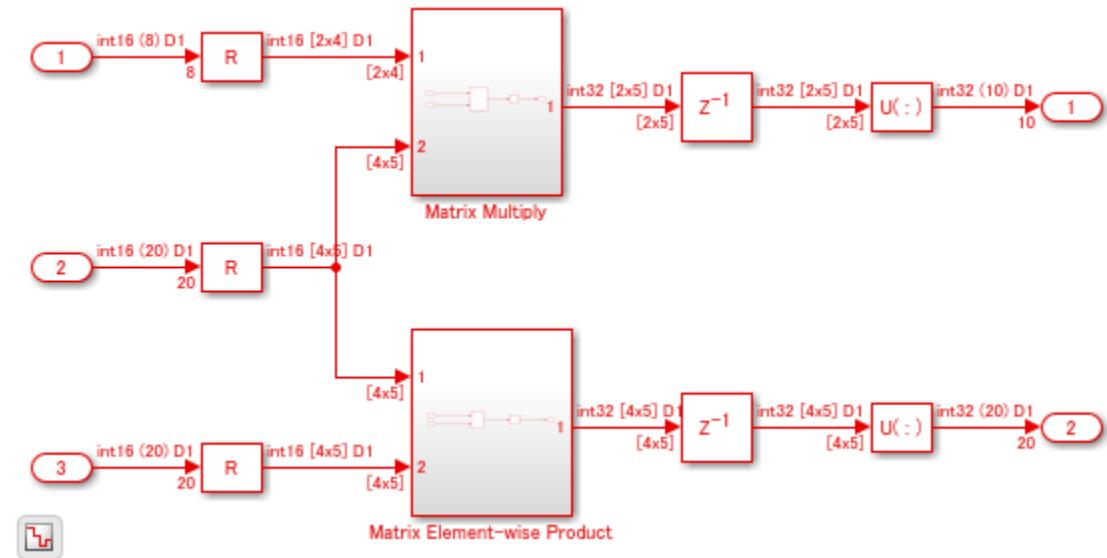
```
open_system('hdlcoder_matrix_multiply')
set_param('hdlcoder_matrix_multiply', 'SimulationCommand', 'update')
sim('hdlcoder_matrix_multiply')
```



Copyright 2018 The MathWorks, Inc.

If you open the DUT Subsystem, you see two subsystems. The Reshape blocks convert the 1-D array back to the 2-by-2 matrices for input to the subsystems. One subsystem uses a Matrix Multiply block and the other subsystem performs element-wise multiplication. The output result is converted back to vectors.

```
open_system('hdlcoder_matrix_multiply/DUT')
```



If you generate HDL code for the DUT Subsystem and open the generated model, you see how the multiplication operation is performed.

Avoid Generating Ascending Bit Order in HDL Code From Vector Signals

Guideline ID

1.3.2

Severity

Strongly Recommended

Description

In MATLAB, the default bit ordering for arrays is ascending. The generated VHDL code in such cases uses a declaration of `std_logic_vector (0 to n)`. This signal declaration generates warnings by violating certain HDL coding standard rules. These are some scenarios:

Ascending Bit Order Scenarios

Scenario	Problem Example	Workaround
Delay block with a Delay length greater than 1.	<p>This example illustrates the generated code for a Delay block with a Delay length of 5.</p> <pre> ENTITY Subsystem1 IS PORT(clk : IN std_logic; reset : IN std_logic; enb : IN std_logic; In1 : IN std_logic; Out1 : OUT std_logic); END Subsystem1; ARCHITECTURE rtl OF Subsystem1 IS -- Signals SIGNAL Delay_reg : std_logic_vector(Signal4); -- ufix1 [5] SIGNAL Delay_out1 : std_logic; </pre>	<p>Instead of using a Delay block with a Delay length of 5, you can connect five Delay blocks that have a Delay length of 1 in series.</p> <pre> ENTITY Subsystem1 IS PORT(clk : IN std_logic; reset : IN std_logic; enb : IN std_logic; In1 : IN std_logic; Out1 : OUT std_logic); END Subsystem1; ARCHITECTURE rtl OF Subsystem1 IS -- Signals -SIGNAL1 Delay_out1 : std_logic; -- ufix1 [5] SIGNAL Delay1_out1 : std_logic; SIGNAL Delay2_out1 : std_logic; SIGNAL Delay3_out1 : std_logic; SIGNAL Delay4_out1 : std_logic; </pre>
Combining multiple input signals to a vector signal using the Mux block.	<p>This example illustrates the generated code when you use a Mux block to combine 4 input signals.</p> <pre> ENTITY Subsystem IS PORT(In1 : IN std_logic; Out1 : OUT std_logic_vector(TO(4));); END Subsystem; ARCHITECTURE rtl OF Subsystem IS -- Signals SIGNAL Mux_out1 : std_logic_vector(0 TO 3); -- ufix1 [4] </pre>	<p>Use a Bit Concat block to combine the input signals. This example illustrates the generated code for this block by concatenating 4 input signals.</p> <pre> ENTITY Subsystem IS PORT(In1 : IN std_logic; Out1 : OUT std_logic_vector(3 DOWNTO 0);); END Subsystem; ARCHITECTURE rtl OF Subsystem IS SIGNAL Bit_Concat_out1 : unsigned(3 DOWNTO 0); </pre>

Scenario	Problem Example	Workaround
Using a Constant block to generate vector signals.	<p>This example illustrates the generated code when you use a Constant block to generate a vector of 4 scalar boolean signals.</p> <pre> ENTITY Subsystem2 IS PORT(Out1 : OUT std_logic_vector(0 TO 3) -- boolean [4]); END Subsystem; ARCHITECTURE rtl OF Subsystem2 IS -- Signals SIGNAL Constant_out1 : std_logic_vector(0 TO 3); -- boolean [4] </pre>	<p>Use a Demux block followed by a Bit Concat block after the Constant block. This example illustrates the generated code when you apply this modeling technique to the vector of 4 Constant block.</p> <pre> ENTITY Subsystem2 IS PORT(Out1 : OUT std_logic_vector(3 DOWNTO 0)); END Subsystem2; ARCHITECTURE rtl OF Subsystem2 IS -- Signals SIGNAL Constant_out1 : std_logic_vector(3 DOWNTO 0); SIGNAL Constant_out1_0 : std_logic; SIGNAL Constant_out1_1 : std_logic; SIGNAL Constant_out1_2 : std_logic; SIGNAL Constant_out1_3 : std_logic; SIGNAL Bit_Concat_out1 : unsigned(3 DOWNTO 0) </pre>

See Also

Functions

`makehdl | makehdltb`

More About

- “Signal and Data Type Support” on page 10-2
- “RTL Description Techniques” on page 26-18

Use Bus Signals to Improve Readability of Model and Generate HDL Code

You can follow these guidelines to learn about bus signals, how to model your design by using these signals, and generate HDL code. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Guideline ID

1.3.3

Severity

Informative

Description

When to Use Buses?

If your DUT or other blocks in your model have many input or output signals, you can create bus signals to improve the readability of your model. A bus signal or bus is a composite signal that consists of other signals that are called elements. The bus signal can have a structure of different data types or a vector signal with the same data types. If all signals have the same data type, you generally use a Mux block. The constituent signals or elements of a bus can be:

- Mixed data type signals such as double, integer, and fixed-point
- Mixed scalar and vector elements
- Mixed real and complex signals
- Other buses nested to any level
- Multidimensional signals

HDL Coder™ Support for Buses

You can generate HDL code for designs that have:

- DUT subsystem ports connected to buses.
- Simulink® and Stateflow® blocks supported for HDL code generation.

HDL Coder supports code generation for bus-capable blocks in the **HDL Coder** block library. Bus-capable blocks are blocks that can accept bus signals as input and produce bus signals as outputs. For a list of bus-capable blocks that Simulink supports, see “Bus-Capable Blocks”.

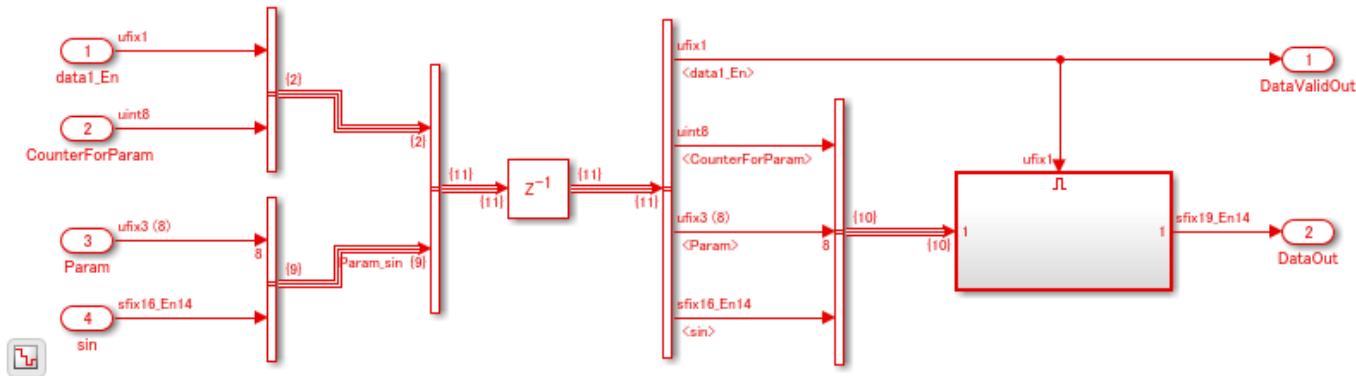
See “Signal and Data Type Support” on page 10-2 for blocks that support HDL code generation with buses.

Create Bus Signals

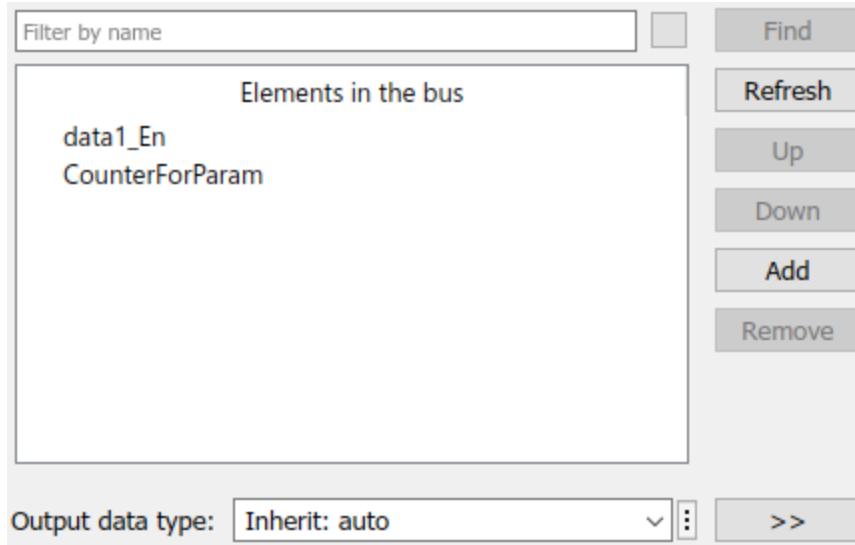
You can create bus signals by using Bus Creator blocks. A Bus Creator block assigns a name to each signal that it creates. You can then refer to signals by name when you search for their sources.

For an example that illustrates how to model with buses, open `hdlcoder_bus_nested.slx`. Double-click the **HDL_DUT** Subsystem.

```
open_system('hdlcoder_bus_nested')
set_param('hdlcoder_bus_nested','SimulationCommand','Update')
open_system('hdlcoder_bus_nested/HDL_DUT')
```



In this model, the Bus Creator blocks create two bus signals. One bus signal contains `data1_En` and `CounterForParam` signals. The other bus signal contains `Param` and `sin` signals. By default, each signal on the bus inherits the name of the signal connected to the bus. This figure shows the **Elements in the bus** block parameter for the Bus Creator block that takes inputs `data1_En` and `CounterForParam`.



Nest Buses

You see another Bus Creator block that combines these two bus signals. When one or more inputs to a Bus Creator block is a bus, the output is a nested bus.

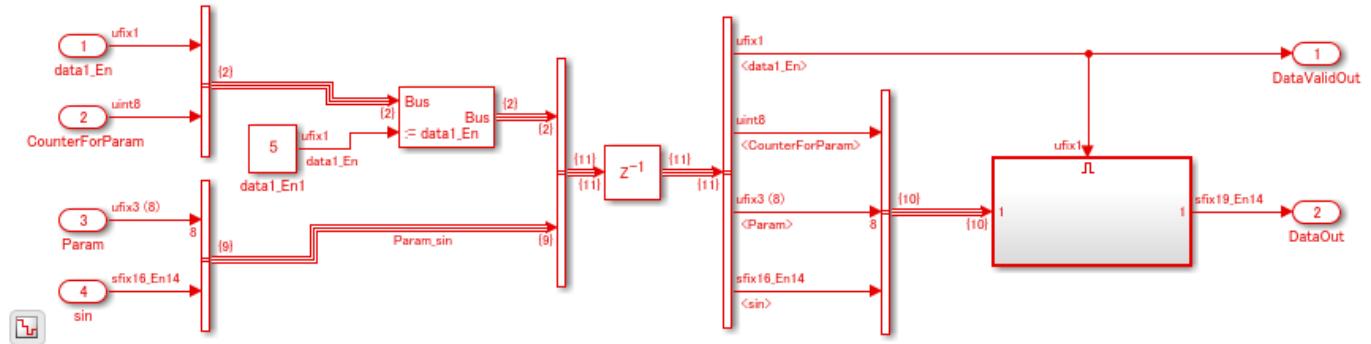
The Bus Creator block generates names for bus signals whose corresponding inputs do not have names. The names are in the form `signaln`, where n is the number of the port the input signal connects to. For example, if you open the Block Parameters dialog box for the second Bus Creator block, you see **Elements in the bus** as `signal1` and `Param_sin`.

Assign Signal Values to Bus

To change bus element values, use a Bus Assignment block. Use a Bus Assignment block to change bus element values without adding Bus Selector and Bus Creator blocks that select bus elements and reassemble them into a bus.

For example, open the model `hdlcoder_bus_nested_assignment`.

```
open_system('hdlcoder_bus_nested_assignment')
set_param('hdlcoder_bus_nested_assignment','SimulationCommand','Update')
open_system('hdlcoder_bus_nested_assignment/HDL_DUT')
```



In the model, you see a Bus Assignment block that assigns the value 5 to the `data1_En` signal in the bus.

Select Bus Outputs

To extract the signals from a bus that includes nested buses, use Bus Selector blocks. By default, the block outputs the specified bus elements as separate signals. You can also output the signals as another bus. You can use the `OutputSignals` block property to see the **Elements in the bus** that the block outputs. By using this property, you can track which signals are entering a Bus Selector block deep within your model hierarchy.

```
get_param('hdlcoder_bus_nested/HDL_DUT/Bus Selector5', 'OutputSignals')
```

```
ans =
'signal1.data1_En,signal1.CounterForParam,Param_sin.Param,Param_sin.sin'
```

Generate HDL Code

To generate HDL code for this model, run this command:

```
makehdl('hdlcoder_bus_nested/HDL_DUT')
```

You see that the code generator expands the bus signals to scalar signals in the generated code. For example, if you open the generated Verilog file for the `HDL_DUT` Subsystem, for the Delay block that takes the two nested bus signals `signal1` and `Param_sin`, you see four always blocks created for each signal in the bus. For example, you see an always block for the `data1_En` signal that is part of `signal1`. This figure displays the scalar signals created for each bus signal in the module definition.

```

`timescale 1 ns / 1 ns

module HDL_DUT
  (clk, reset, clk_enable,
   data1_En, alphaCounterForParam,
   Param_0, Param_1, Param_2, Param_3,
   Param_4, Param_5, Param_6, Param_7,
   sin, ce_out, DataValidOut, DataOut);

  .....

  assign ce_out = clk_enable;

endmodule // HDL_DUT

```

Simplify Subsystem Bus Interfaces

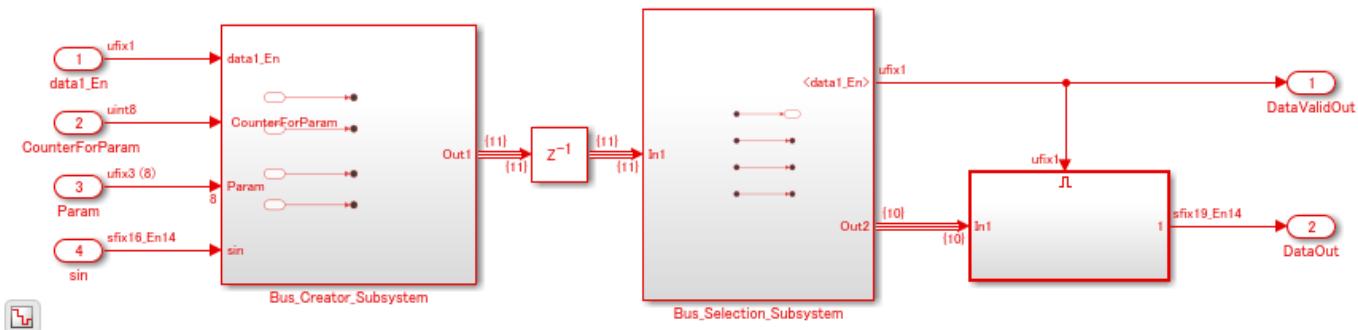
You can simplify the Subsystem bus interfaces by using Bus Element blocks. The In Bus Element and Out Bus Element blocks provide a simplified and flexible way to use bus signals as inputs and outputs to subsystems. The In Bus Element block is equivalent to an Import block combined with a Bus Selector block. The Out Bus Element block is equivalent to an Outport block combined with a Bus Creator block. To refactor an existing model that uses Import, Bus Selector, Bus Creator, and Outport blocks to use In Bus Element and Out Bus Element blocks, you can use Simulink® Editor action bars.

For example, open the model `hdlcoder_bus_nested_simplified`. This model is functionally equivalent to the `hdlcoder_bus_nested` model but is a more simplified version.

```

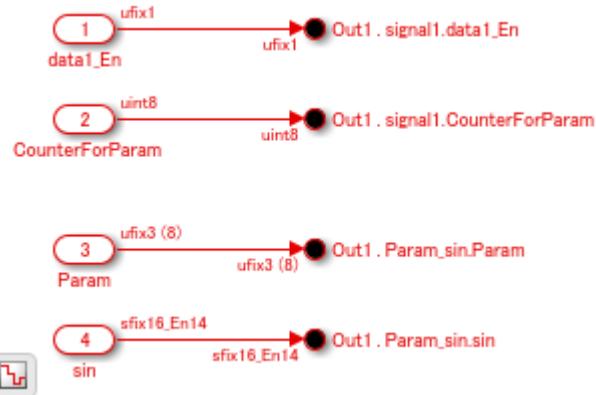
open_system('hdlcoder_bus_nested_simplified')
set_param('hdlcoder_bus_nested_simplified','SimulationCommand','Update')
open_system('hdlcoder_bus_nested_simplified/HDL_DUT')

```



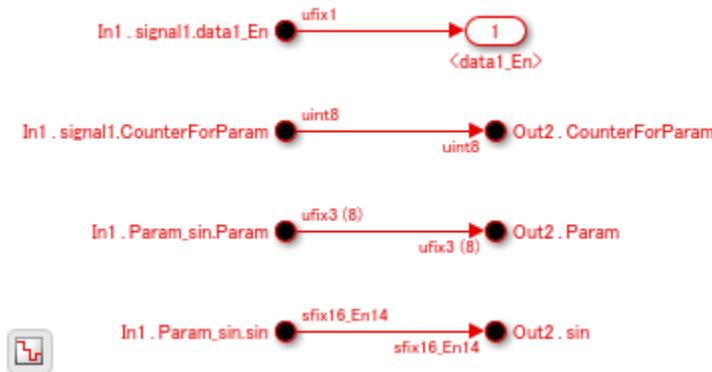
The model has two Subsystems that perform bus creation and bus selection by using Bus Element blocks. The `Bus_Creator_Subsystem` combines the Outport blocks with the Bus Creator blocks to create Out Bus Element blocks.

```
open_system('hdlcoder_bus_nested_simplified/HDL_DUT/Bus_Creator_Subsystem')
```



The `Bus_Selection_Subsystem` combines the Import blocks with the Bus Selector blocks to create In Bus Element blocks.

```
open_system('hdlcoder_bus_nested_simplified/HDL_DUT/Bus_Selection_Subsystem')
```



To learn more, see “Simplify Subsystem and Model Interfaces with Buses”.

Virtual and Nonvirtual Buses

The bus signals in the model `hdlcoder_bus_nested` created earlier by using Bus Creator and Bus Selector blocks are virtual buses. Each bus element signal is stored in memory, but the bus signal is not stored. The bus simplifies the graph but has no functional effect. In the generated HDL code, you see the constituent signals but not the bus signal.

To more easily track the correspondence between a bus signal in the model and the generated HDL code, use nonvirtual buses. Nonvirtual buses generate clean HDL code because it uses a structure to hold the bus signals. To convert a virtual bus to a nonvirtual bus, in the Block Parameters of the Bus Creator blocks, you specify the **Output data type** as `Bus: object_name` by replacing `object_name` with the name of the bus object and then select **Output as nonvirtual bus**.

See “Convert Virtual Bus to a Nonvirtual Bus”.

Array of Buses

An array of buses is an array whose elements are buses. Each element in an array of buses must be nonvirtual and must have the same data type.

To learn more about modeling with array of buses, see “Generating HDL Code for Subsystems with Array of Buses” on page 10-21.

See Also

Functions

`makehdl` | `makehdltb`

More About

- “Signal and Data Type Support” on page 10-2
- “Use Arrays of Buses in Models”
- “Convert Models to Use Arrays of Buses”

Guidelines for Clock and Reset Signals

In the Simulink modeling environment, You do not create global signals such as clock, reset, and clock enable. These signals are created when you generate HDL code for your model. You can specify the clock cycle by using the sample time in Simulink.

If your model is single rate, it means all blocks operate at the same sample time. The synthesis tools infer that the registers or Delay blocks you add to your model run at the clock rate. For the synthesis tools, data propagates from the source register to the destination register in one clock cycle.

You can follow these guidelines to learn about generating clock signals in the HDL code. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Use Global Oversampling to Create Frequency-Divided Clock

Guideline ID

1.4.1

Severity

Informative

Description

You can assign a frequency-divided clock rate for HDL code generation to be a multiple of the Simulink base sample rate. For example, if the Simulink base rate is 1 MHz and you want the clock frequency of your target hardware to run at 50 MHz, you can assign an **Oversampling factor** of 50. You specify the “Oversampling factor” on page 17-15 in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box. To learn more, see “Generate a Global Oversampling Clock” on page 23-8.

Create Multirate Model with Integer Clock Multiples by Clock Division

Guideline ID

1.4.2

Severity

Mandatory

Description

You can generate a multirate model by using clock-rate division or by using clock multiples. For a multirate model, the fastest sample time in your Simulink™ model corresponds to the master clock rate. A timing controller entity is created to control the clocking for blocks operating at slower sample rates. Clock enable signals that have the necessary rate and phase information control the clocking for these blocks in your design.

Multirate models are created when you use certain blocks in your Simulink™ model, specify certain block architectures, or use operations such as resource sharing. For example, these block-architecture combinations generate a multirate model:

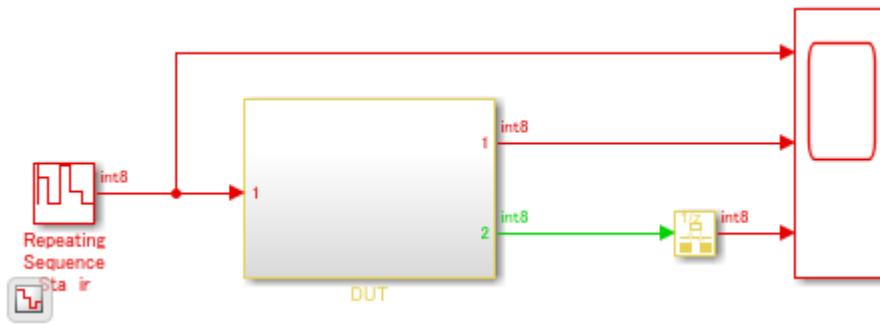
- Divide block with Newton-Raphson implementation.
- Reciprocal block with ReciprocalSqrtBasedNewton implementation.
- Sum of Elements and Product of Elements blocks with Cascade architecture.
- Sqrt with SqrtBasedNewton and Reciprocal Sqrt with ReciprocalRsqrtBasedNewton implementation.

In addition, to model multirate designs in Simulink™, use these blocks:

- In the **Simulink > Signal Attributes** Library, use the Rate Transition block.
- In the **DSP System Toolbox > Signal Operations** Library, you can use Upsample, Downsample, and Repeat blocks.
- In the **HDL Coder > HDL RAMs** Library, use the HDL FIFO block.

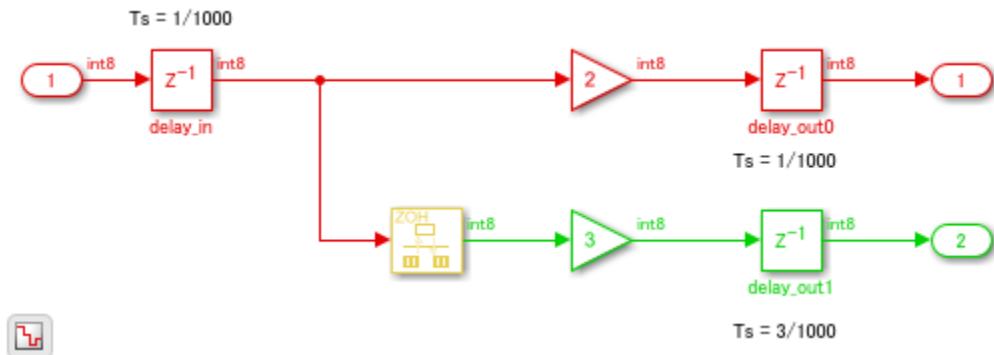
This model illustrates how to create a multirate design by using a Rate Transition block.

```
load_system('hdlcoder_multiclock')
set_param('hdlcoder_multiclock','SimulationCommand','Update')
open_system('hdlcoder_multiclock')
```



The different colors in the model indicate that the model is multirate and has a faster rate D1 and a slower rate D2. To see the Rate Transition block that produces the different sample rates, double-click the DUT Subsystem.

```
open_system('hdlcoder_multiclock/DUT')
```



To see the sample times in your model, run this command:

```

ts = Simulink.BlockDiagram.getSampleTimes('hdlcoder_multiclock');
sampletime_D1 = ts(1)
sampletime_D2 = ts(2)

sampletime_D1 =
    SampleTime with properties:
        Value: [1.0000e-03 0]
        Description: 'Discrete 1'
        ColorRGBValue: [0.9000 0 0]
        Annotation: 'D1'
        OwnerBlock: []
    ComponentSampleTimes: [0x0 Simulink.SampleTime]

sampletime_D2 =
    SampleTime with properties:
        Value: [0.0030 0]
        Description: 'Discrete 2'
        ColorRGBValue: [0 0.8200 0]
        Annotation: 'D2'
        OwnerBlock: []
    ComponentSampleTimes: [0x0 Simulink.SampleTime]

```

When you use a Rate Transition block in your model for multirate design, select the block parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)**. Make sure the output sample rate is an integer multiple of the input sample rate.

For a multirate design, you can generate a single clock signal or multiple clock signals to control the clocking to blocks that operate at various sample rates. To specify this setting, in the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings** pane, specify the **Clock inputs** setting.

By default, **Clock inputs** is specified as **single**. A single clock is generated to control the clocking for all registers or Delay blocks in your model. The timing controller enable signals control the clocking to various blocks in your design. This mode can increase the power dissipation as a single, fastest clock is connected to all registers in your design.

If you specify **Clock inputs** as **multiple**, a clock signal is generated for each sample rate in your design. However, this mode requires you to connect each of the clock, clock enable, and reset ports externally. This mode reduces power as the HDL design contains registers connected to slower clock signals. For more information, see “Using Multiple Clocks in HDL Coder” on page 23-12.

Use Dual Rate Dual Port RAM for Noninteger Multiple Sample Times

Guideline ID

1.4.3

Severity

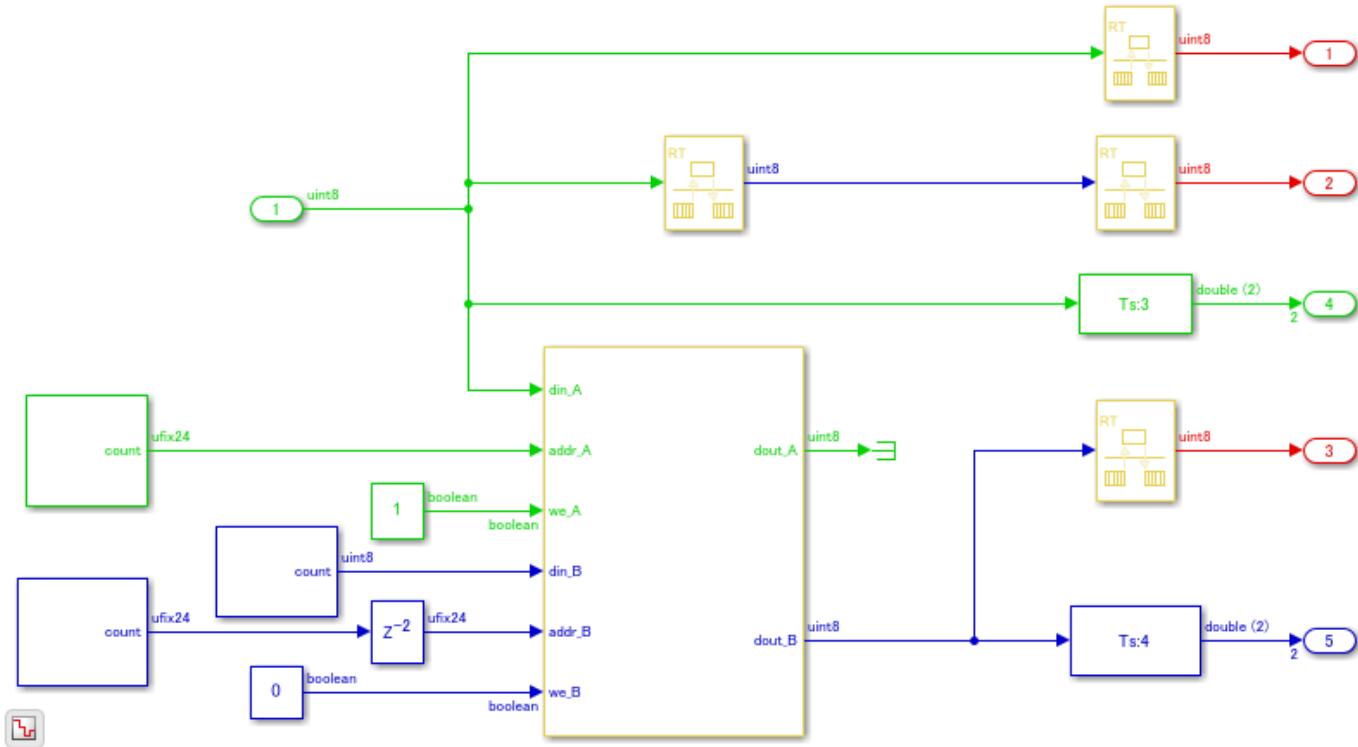
Mandatory

Description

When you use Rate Transition, Upsample, or Downsample blocks to create multirate models, the clock rates must be integer multiples of the base rate. To create a multirate model with clocks that are noninteger multiples, use a Dual Rate Dual Port RAM block. For integer clock multiplies, you can use the HDL FIFO or the Dual Rate Dual Port RAM block.

This model illustrates how you can create noninteger multiples of sample rates.

```
load_system('hdlcoder_dual_rate_dual_port_RAM')
set_param('hdlcoder_dual_rate_dual_port_RAM','SimulationCommand','Update')
open_system('hdlcoder_dual_rate_dual_port_RAM/DUT')
```



You cannot generate HDL code for this model because the Rate Transition blocks have the block parameter **Ensure data integrity during data transfer** cleared. To learn how you can manage address control when you use the RAM block, see Design considerations for RAM block access.

Asynchronous Clock Modeling in HDL Coder

Guideline ID

1.4.4

Severity

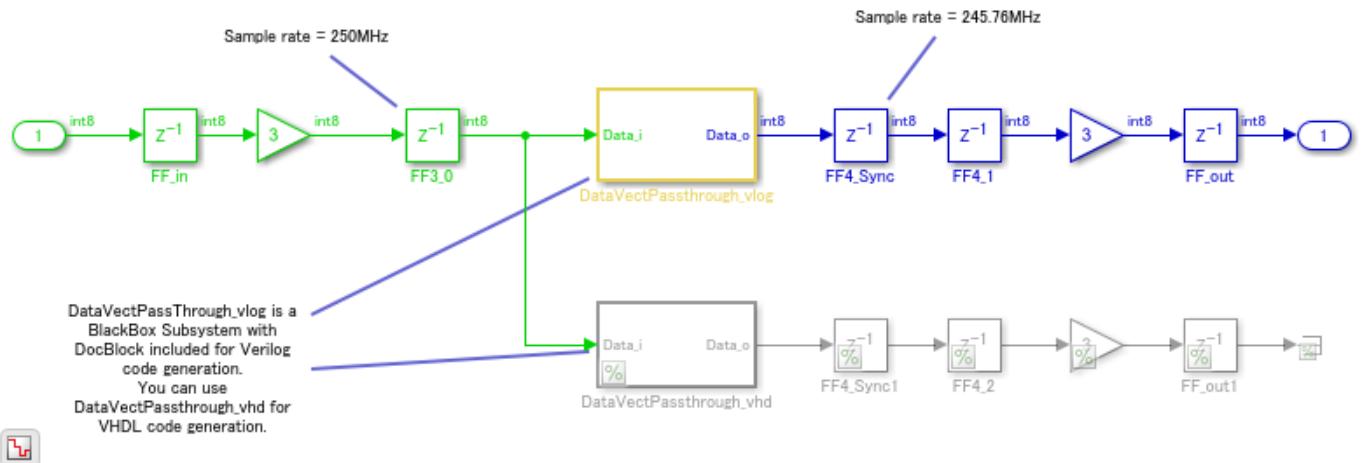
Recommended

Description

Most FPGA designs must have more than one clock domain with multiple parts of the design operating at various frequencies. You can model the various clock domains in Simulink™ by using a pass-through implementation for transitioning between different sample rates. These sample rates correspond to the clock rates on the FPGA device.

For an example, open the model `hdlcoder_multi_clock_domain` and then open the DUT Subsystem.

```
load_system('hdlcoder_multi_clock_domain')
set_param('hdlcoder_multi_clock_domain', 'SimulationCommand', 'Update')
open_system('hdlcoder_multi_clock_domain/DUT')
```



You see a BlackBox Subsystem that contains a DocBlock, which is a text file that corresponds to the Verilog code for a passthrough implementation. You can open the DocBlock to see the Verilog code. You see that the output of this Subsystem operates at a different sample rate or is in a clock domain that is different from the sample rate at the input of the Subsystem. The Subsystem also contains a commented out path that contains the VHDL equivalent of the passthrough implementation. To generate VHDL code, uncomment this path and comment out the path that contains the Verilog BlackBox implementation.

To generate Verilog code for this model, run this command:

```
makehdl('hdlcoder_multi_clock_domain/DUT')
```

In the generated Verilog header file, you see the different clock domains in the model.

```
// -----
// 
// File Name: hdlsrc\hdlcoder_multi_clock_domain\DUT.v
// Created: 2018-10-05 11:30:21
//
// Generated by MATLAB 9.6 and HDL Coder 3.13
//
// 
// -- -----
// -- Rate and Clocking Details
```

```
// -- -----
// Model base rate: 1.30208e-12
// Target subsystem base rate: 2.65428e-12
//
//
// Clock      Domain  Description
// -----
// clk_1_3072 1      3072x slower than base rate clock
// clk_1_3125 2      3125x slower than base rate clock
//
//
// Output Signal          Clock      Domain  Sample Time
// -----
// Output1              (no clock) 0      4.06901e-09
// -----
//
```

Use Global Reset Type Setting Based on Target Hardware

Guideline ID

1.4.5

Severity

Recommended

Description

Matching the reset type to the FPGA architecture can improve resource utilization and the speed at which your design runs on the target hardware. To control this setting, in the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings** settings, specify the **Reset type**.

When you target Xilinx devices, set **Reset type** to **Synchronous**. For Intel or Altera devices, set **Reset type** to **Asynchronous**.

To make sure that you use the correct reset type for the hardware that you are targeting, in the HDL Code Advisor, run the model check “Check for global reset setting for Xilinx and Altera devices” on page 38-7.

Note Some Intel devices recommend using synchronous reset. For recommended reset settings, see the Intel or Xilinx documentation for that device.

See Also

Functions

`makehdl | makehdltb`

Simulink Configuration Parameters

“Timing Controller Settings” on page 17-60 | “Clock Settings and Timing Controller Postfix Parameters” on page 17-4

More About

- “Multirate Model Requirements for HDL Code Generation” on page 23-6
- “Code Generation from Multirate Models” on page 23-2
- “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2

Modeling with Native Floating Point

HDL Coder native floating-point technology can generate HDL code from your floating-point design. These are some of the key features:

- Generation of target-independent HDL code that you can deploy on any FPGA or ASIC.
- Support for the full range of IEEE-754 features including denormal numbers, exceptions, and rounding modes.
- Extensive support for math and trigonometric blocks.

You can follow these guidelines as best practices when modeling your design for native floating-point code generation.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Guideline ID

1.5.1

Severity

Recommended

Description

Native floating-point support in HDL Coder generates code from your floating-point design. If your design has complex math and trigonometric operations or has data with a large dynamic range, use native floating-point. The generated HDL code is target-independent and complies with the IEEE-754 standard of floating-point arithmetic. To learn more, see “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80.

You can use these modeling guidelines when using the native floating-point support in HDL Coder.

Use Blocks from HDL Floating Point Operations Library

The **HDL Floating Point Operations** block library consists of math and trigonometric functions and certain Simulink blocks that are configured for HDL code generation in native floating-point mode. For example, Discrete FIR Filter with **Architecture** set to **Fully Parallel**.

Use Floating-Point Types Based on Accuracy and Hardware Resource Usage Requirements

You can generate HDL code for models that contain floating-point and fixed-point data types in native floating-point mode. Floating point types have higher dynamic range but can potentially occupy more area on the target hardware. To design for these trade-offs, in your Simulink model, it is recommended to use floating-point data types to model the algorithm data path and fixed-point types to model the control logic. To switch between floating-point and fixed-point data types, use Data Type Conversion blocks.

See also “Data Type Considerations” on page 10-82.

Enable Optimizations such as Resource Sharing on Model

By enabling optimizations on the model, you can improve area and timing of your design on the target FPGA device. For example, to save area on the target FPGA device, use the resource sharing optimization. To share:

- Floating-point adders, set “Share Adders” on page 15-15 to on.
- Floating-point multipliers, make sure “Share Multipliers” on page 15-17 is set to on.
- Other floating-point resources, set “Share Floating-Point IPs” on page 15-25 to on.

See also “Resource Sharing” on page 24-32.

Simulate Latency of Blocks in Model

Floating-point designs have an inherent latency by default. This latency is added when generating HDL code for your model. It is recommended that you simulate latency in your model by adding this latency information to your original Simulink model. The code generator absorbs this latency during HDL code generation. To learn more, see “Latency Values of Floating Point Operators” on page 10-91.

Customize Latency of Model or Blocks

You can customize the latency of an entire model, or selectively for certain blocks in your design. Using custom settings, you can specify a custom latency and design for trade-offs between latency and throughput.

To learn more, see “Latency Considerations with Native Floating Point” on page 10-96.

Use sincos Block Instead of Separate sin and cos Blocks

Certain modeling patterns that you use can optimize your model when you generate code with native floating-point technology. For example, if you are computing the trigonometric sine and cosine of the same input, in the **HDL Floating Point Operations** block library, use the Sincos block instead of separate Sin and Cos blocks. The Sincos block shares some of the logic that is used for computing the sine and cosine of the input. This implementation reduces the area footprint on the target FPGA device.

See also Trigonometric Function.

Use Tree as the HDL Architecture

To obtain a lower latency implementation, use Tree as the **HDL Architecture** for blocks such as the Sum of Elements and Product of Elements.

See also Sum of Elements and Product of Elements.

See Also

Functions

`makehdl | makehdltb`

Simulink Configuration Parameters

“Floating Point IP Library” on page 16-3 | “Native Floating Point Parameters” on page 16-4

More About

- “Numeric Considerations with Native Floating-Point” on page 10-84
- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103
- “Verify the Generated Code from Native Floating-Point” on page 10-116

Design Considerations for RAM Blocks and Blocks in HDL Operations Library

Follow these guidelines to learn how you can use RAM blocks and blocks in the **HDL Operations** library when modeling your design.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

RAM Block Access Considerations

Guideline ID

2.1.1

Severity

Recommended

Description

In the **HDL RAMs** block library, there are seven different RAM blocks and a HDL FIFO block. If you see a RAM block that has the term **System** as part of the block name, such as Single Port RAM System, it is recommended that you use this block instead of the equivalent block that does not have **System** as part of the name, such as Single Port RAM. These blocks have **System** as part of the name because the block implementation is based on the `hdl.RAM System` object. The system blocks support vector inputs and yield much faster simulation results when used in your Simulink model.

When you use these blocks, make sure that the input sample time and output sample time are the same. This table illustrates the various RAM blocks that you can use and their purpose. Each row in the table describes a RAM block that is larger in circuit size than the RAM block in the previous row. The generated HDL code for these blocks maps to RAM in most FPGAs.

Block Name	Recommended Usage
Single Port RAM System	<p>Use this block to replace the Single Port RAM block in your model. You obtain faster simulation results when using this block in your model.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> object. Use this block to perform sequential read and write operations. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. To perform simultaneous read and write operations to different addresses, use the Simple Dual Port RAM System or the Dual Port RAM System block instead.</p> <p>The block does not support boolean inputs. Cast boolean types to <code>ufix1</code> for input to the block.</p>

Block Name	Recommended Usage
Simple Dual Port RAM System	<p>Use this block to replace the Simple Dual Port RAM block in your model. You obtain faster simulation results when using this block in your model.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> object. Use this block to perform simultaneous read and write operations. It has a single output port to read data. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM.</p> <p>The block does not support boolean inputs. Cast boolean types to <code>ufix1</code> for input to the block.</p>
Dual Port RAM System	<p>Use this block to replace the Dual Port RAM block in your model. You obtain faster simulation results when using this block in your model.</p> <p>The block implementation uses a MATLAB System block that uses the <code>hdl.RAM System</code> object. Use this block to perform simultaneous read and write operations. It has a read data output port and a write data output port. In the Block Parameters dialog box of the block, you can specify an initial value for the RAM. If you do not want to use the write data output port, to achieve better RAM inference, use the Simple Dual Port RAM System block instead.</p> <p>The block does not support boolean inputs. Cast boolean types to <code>ufix1</code> for input to the block.</p>
Dual Rate Dual Port RAM	<p>This block does not have an equivalent System object-based implementation.</p> <p>Use this block to perform simultaneous read and write operations to two different addresses that operate at different clock rates. You cannot perform concurrent access to the same address of the RAM.</p> <p>To run the RAM ports at multiple clock rates, set Clock Inputs to Multiple. You can access this RAM twice in one clock cycle.</p>

Block Name	Recommended Usage
HDL FIFO	<p>The HDL FIFO block stores a sequence of samples in a first in, first out (FIFO) register.</p> <p>The inputs, In and Push, and the outputs, Out and Pop can run at different sample times. Specify the ratio of output to input sample time as a positive integer or $1/N$ such that N is a positive integer. For example:</p> <ul style="list-style-type: none"> • If you specify the ratio as 2, the output sample time is twice the input sample time. The outputs run slower than the input. • If you specify the ratio as $1/2$, the output sample time is half the input sample time. The outputs run faster than the input. <p>The signals Full, Empty, and Num run at the fastest rate in your model. When you use the control output of the FIFO in an input, you may have to perform to a rate transition.</p> <p>The input and output rates of the FIFO block are synchronous to each other. For an example of asynchronous FIFO modeling by using the HDL FIFO block, open the model <code>hdlcoder_asynchronous_fifo</code>.</p> <pre>open_system('hdlcoder_asynchronous_fifo')</pre>

Serial to Parallel Conversion

Guideline ID

2.1.2

Severity

Informative

Description

You can use the Serializer1D and Deserializer1D blocks to perform serial to parallel and parallel to serial conversion.

See Also

Functions

`makehdl`

Related Examples

- “Getting Started with RAM and ROM in Simulink®” on page 22-58

More About

- “HDL Code Generation from `hdl.RAM` System Object” on page 1-37

- “Map Matrices to Block RAMs to Reduce Area” on page 8-3

Usage of Blocks in Logic and Bit Operations Library

These guidelines illustrate how to model your design for generating HDL-ready code from blocks in the **Logic and Bit Operations** Library. The Library contains blocks that perform logical and bitwise operations, bit reduction, and concatenation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Logical and Arithmetic Bit Shift Operations

Guideline ID

2.2.1

Severity

Informative

Description

You can use Simulink blocks to perform bit shifting operations. The blocks can perform logical and arithmetic bit shift. Left logical and arithmetic bit shift produce the same results but right logical shift and arithmetic shift operate differently as illustrated in this table.

Block/Function Name	Parameter/Operation	Verilog Code Equivalent	VHDL code equivalent	Comments
Bit Shift	Shift Left Logical	<<<	sll (sll and SHIFT_LEFT are the same in VHDL.)	This mode is the default mode for the block. The left shift operation does not preserve the sign bit. If the input uses a signed data type and has a positive value, the left shift operation shifts a 0 into the empty bit on the LSB (Least Significant Bit) side.
	Shift Right Logical	>>	srl	This mode does not preserve the sign bit. If the input uses a signed data type and has a positive value, the right shift operation shifts a 0 into the empty bit on the MSB (Most Significant Bit) side.
	Shift Right Arithmetic	>>>	SHIFT_RIGHT	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right.
Shift Arithmetic block bitshift function	Positive value/shift right arithmetic	>>>	SHIFT_RIGHT	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right.

Block/Function Name	Parameter/Operation	Verilog Code Equivalent	VHDL code equivalent	Comments
	Negative value/shift left arithmetic	<<<	sll	This mode does not preserve the sign bit. If the input uses a signed data type and has a positive value, the left shift operation shifts a 0 into the empty bit on the LSB side. This mode does not check underflows and overflows.
bitsll function	None/logical left shift	<<<	sll	This mode does not preserve the sign bit. The generated HDL code is the same as that of the Shift Left Logical mode of the Bit Shift block.
bitsrl function	None/logical right shift	>>	srl	This mode does not preserve the sign bit. The generated HDL code is the same as that of the Shift Right Logical mode of the Bit Shift block.
bitsra function	None/arithmetic right shift	>>>	SHIFT_RIGHT	When the input is a signed data type, the sign bit is preserved, and other bits shift to the right. The generated HDL code is the same as that of the Shift Right Arithmetic mode of the Bit Shift block.

The difference between a logical shift and an arithmetic shift is whether the sign bit is preserved. For signed data types, this bit is the MSB. In a logical right shift, the sign bit is shifted to the right and zero goes into the MSB side. In an arithmetic right shift, the MSB (sign bit) is preserved during the shift operation. For example, this code illustrates the difference between the functions.

```
A = fi([], 1, 4, 0, 'bin','1011');
B = bitsrl(A, 2)

B =
2

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 4
    FractionLength: 0

B.bin
ans =
'0010'
```

```
C = bitsra(A, 2)
C =
-2

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 4
    FractionLength: 0

C.bin
ans =
'1110'
```

Usage of Logical Operator, Bitwise Operator, and Bit Reduce Blocks

Guideline ID

2.2.2

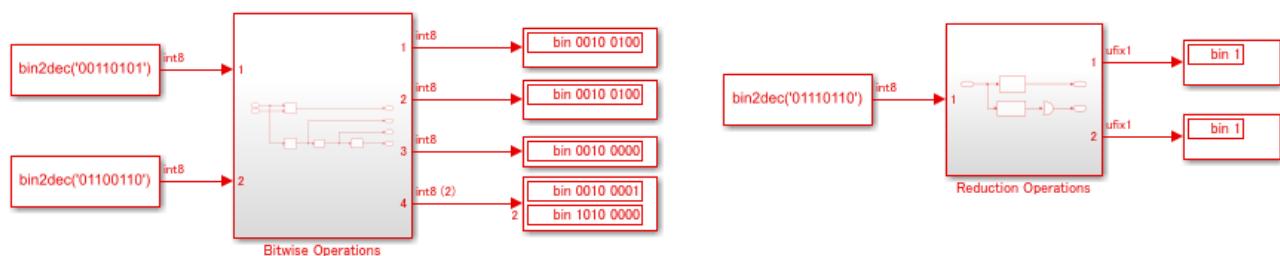
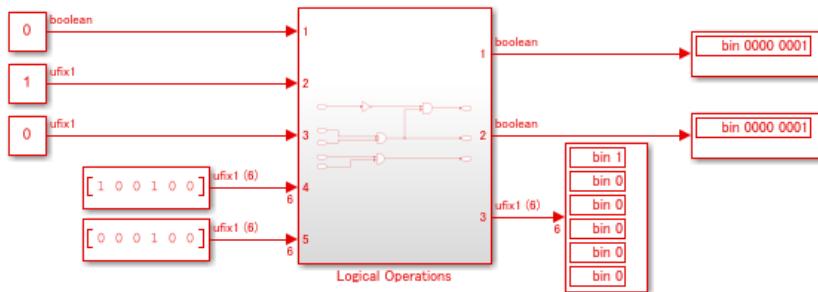
Severity

Informative

Description

To learn how you can use logical and bitwise operations, open the model `hdlcoder_logical_bitwise_operations.slx`.

```
load_system('hdlcoder_logical_bitwise_operations')
sim('hdlcoder_logical_bitwise_operations')
open_system('hdlcoder_logical_bitwise_operations')
```

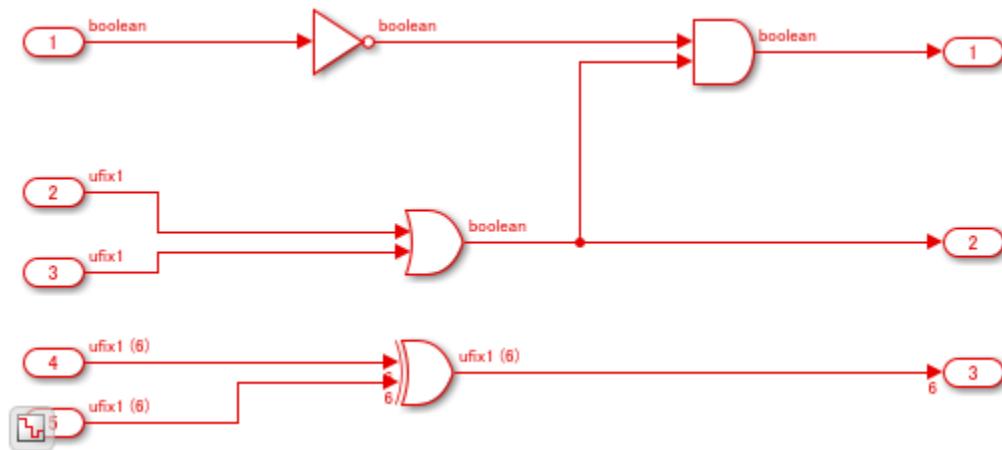


For single-bit operations that use Boolean or `ufix1` data types, use a Logical Operator block. To view the operation as a logical circuit symbol, in the Block Parameters dialog box of the block, specify the **Icon shape** as **Distinctive**. You can also input vectors that have Boolean or `ufix1` data types to the block.

Note: **Boolean** and **`ufix1`** are different data types. Mixing these data types within the same model or using them interchangeably is not recommended. For more information, see [Simulink Data Type Considerations](#).

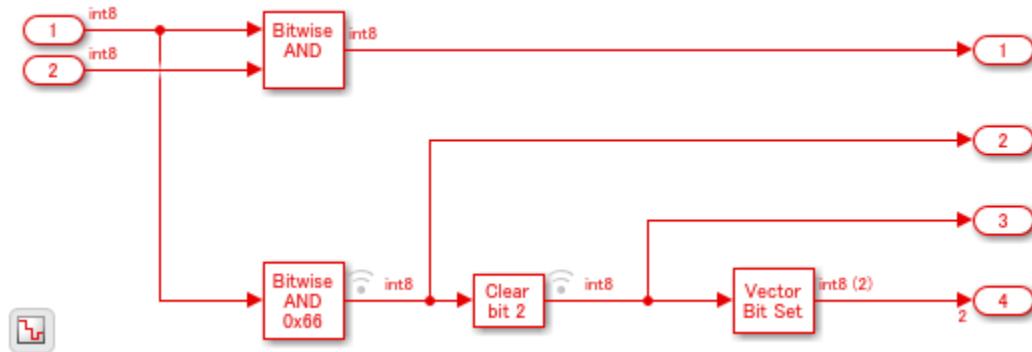
For an example of using the block, open the **Logical Operations Subsystem**.

```
open_system('hdlcoder_logical_bitwise_operations/Logical Operations')
```



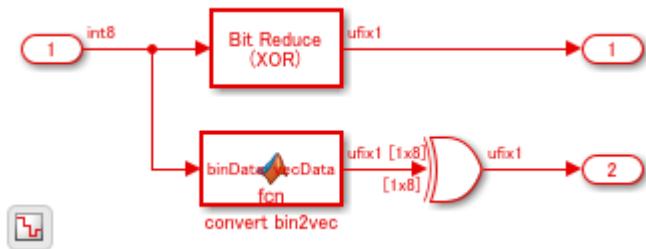
For bitwise operations on two or more bits that use integer or fixed-point data types, use the Bitwise Operator block. For an example, double-click the **Bitwise Operations Subsystem**.

```
open_system('hdlcoder_logical_bitwise_operations/Bitwise Operations')
```



To perform a bit-by-bit reduction operation on a vector that uses Boolean or `ufix1` and return a 1-bit value, use the Bit Reduce block. For an example, double-click the **Reduction Operations Subsystem**.

```
open_system('hdlcoder_logical_bitwise_operations/Reduction Operations')
```



The MATLAB Function block inside the Subsystem converts the 8-bit vector to a vector of 8 1-bit `ufix1` elements.

```
open_system('hdlcoder_logical_bitwise_operations/Reduction_Operations/convert_bin2vec')
```

Use Boolean Output for Compare to Constant and Relational Operator Blocks

Guideline ID

2.2.3

Severity

Strongly Recommended

Description

For Compare To Constant, Compare To Zero, and Relational Operator blocks, you can specify `uint8` or `boolean` as the **Output data type**. To generate efficient HDL code for models that contain these blocks, specify `boolean` as the **Output data type**, because the HDL code has to connect only the LSB.

For a Relational Operator block, make sure that both inputs are of the same data type. Using different data types for the inputs can result in unintended truncation of bits such as the sign bit, which can lead to simulation mismatches after HDL code generation.

To verify whether Relational Operator blocks in your model use the same input data type, and use `boolean` as the output data type, run the HDL model check “Check for Relational Operator block usage” on page 38-29.

See Also

Functions

`makehdl`

Blocks

[Bitwise Operator](#) | [Extract Bits](#)

More About

- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59

Generate FPGA Block RAM from Lookup Tables

You can follow these guidelines to learn about how you can map the lookup table blocks to RAM to save area on the target FPGA device.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Guideline ID

2.3.1

Severity

Strongly Recommended

Description

To map lookup tables to a block RAM, you can use the adaptive pipelining optimization. This optimization is enabled by default. The optimization inserts a Delay block that has a **Delay length** of 1 and **ResetType** set to none immediately following the Lookup Table block. This modeling pattern efficiently maps your design to a Block RAM on the FPGA. To use the adaptive pipelining optimization, you must:

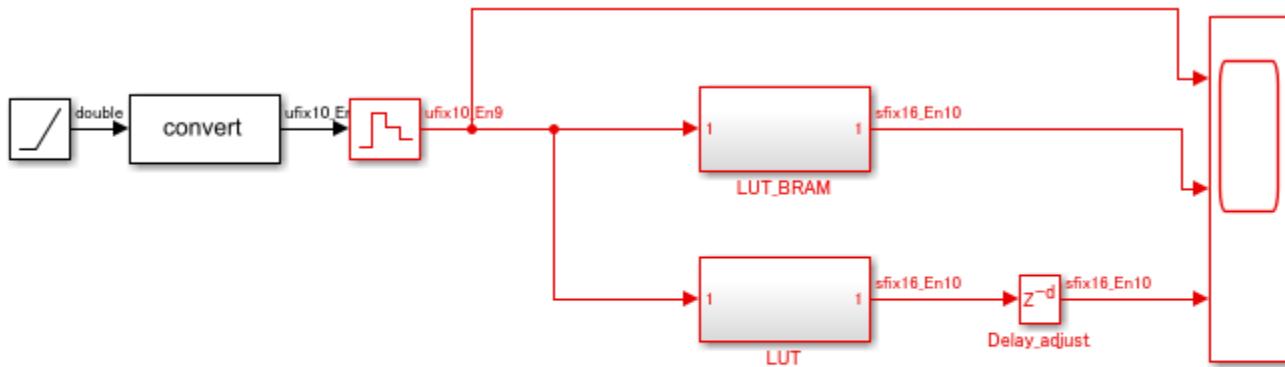
- Make sure that **AdaptivePipelining** is enabled on the model.
- Specify the synthesis tool.

To learn more about adaptive pipelining, see “Adaptive Pipelining” on page 24-130.

If you do not want to use the adaptive pipelining optimization for the entire design, you can selectively enable this optimization for certain Subsystems in your design, or create the same modeling pattern in your design that is otherwise generated by the optimization.

For an example, open the model `hdlcoder_LUT_BRAM_mapping.slx`.

```
open_system('hdlcoder_LUT_BRAM_mapping')
set_param('hdlcoder_LUT_BRAM_mapping','SimulationCommand','Update')
```



The adaptive pipelining optimization is disabled on this model.

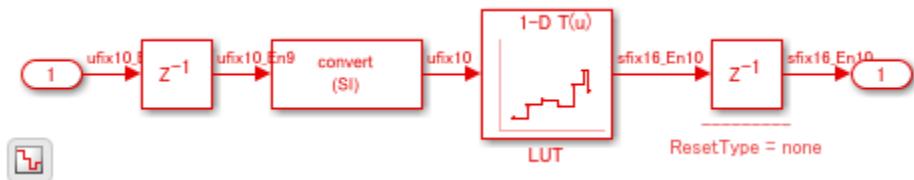
```
hdlget_param('hdlcoder_LUT_BRAM_mapping','AdaptivePipelining')
```

```
ans =
```

```
'off'
```

The LUT_BRAM Subsystem contains a 1-D Lookup Table block followed by a Delay block that has a **Delay length** of 1 and **ResetType** set to none.

```
open_system('hdlcoder_LUT_BRAM_mapping/LUT_BRAM')
```



When you generate HDL code and synthesize the design on an FPGA, it efficiently maps to Block RAM. This figure displays the synthesis results for the LUT_BRAM Subsystem.

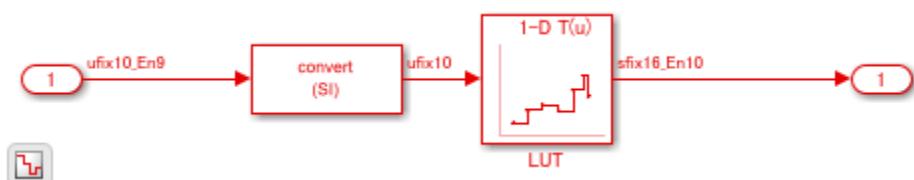
Parsed resource report file: [LUT_OK_utilization_synth.rpt](#).

Resource summary	
Resource	Usage
Slice LUTs	0
Slice Registers	0
DSPs	0
Block RAM Tile	0.5

Parsed timing report file: [timing_post_map.rpt](#).

The LUT Subsystem in this model does not use this modeling pattern.

```
open_system('hdlcoder_LUT_BRAM_mapping/LUT')
```



As adaptive pipelining is disabled on the model, synthesizing this Subsystem maps the logic to LUTs instead of utilizing the Block RAMs. This figure displays the synthesis results for the LUT Subsystem.

Parsed resource report file: [LUT_utilization_synth.rpt](#).

Resource summary	
Resource	Usage
Slice LUTs	138
Slice Registers	0
DSPs	0
Block RAM Tile	0

Parsed timing report file: [timing_post_map.rpt](#).

See Also

Functions

`makehdl`

Simulink Configuration Parameters

[“RAM Mapping Parameters” on page 15-7](#) | [“MapPersistentVarsToRAM” on page 22-16](#) | [“Adaptive pipelining” on page 15-13](#)

More About

- [“Adaptive Pipelining” on page 24-130](#)
- [“RAM Mapping for Simulink Models” on page 24-95](#)
- [“RAM Mapping With the MATLAB Function Block” on page 24-96](#)

Usage of Different Subsystem Types

You can follow these guidelines to learn how to use different types of subsystems in your design and model your algorithm hierarchically. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Virtual Subsystem: Use as DUT

Guideline ID

2.4.1

Severity

Strongly Recommended

Description

A virtual subsystem is a Subsystem that is not a conditionally-executed Subsystem or an Atomic Subsystem. By default, a regular Subsystem block that you add to your Simulink model is a virtual subsystem. Nonvirtual subsystem types include Atomic Subsystem, model reference, Variant Subsystem, and a variant model. You can learn about these subsystem types in the preceding sections.

To determine whether a subsystem is virtual, you can use the `get_param` function with the parameter `IsSubsystemVirtual`. For example:

```
get_param('sfir_fixed/symmetric_fir', 'IsSubsystemVirtual')
```

Atomic Subsystem: Generate Reusable HDL Files

Guideline ID

2.4.2

Severity

Recommended

Description

You can specify the DUT as an Atomic Subsystem. To specify a Subsystem as an Atomic Subsystem, in the Block Parameters dialog box of that Subsystem, select **Treat as atomic unit**. Use atomic subsystems to generate a single HDL file for identical instances of the subsystems that you use at lower levels of a hierarchy. To learn more, see “Generate Reusable Code for Atomic Subsystems” on page 27-17.

To enable resource sharing on a subsystem unit, specify all subsystems that you want to share as atomic subsystems. To learn more, see “General Considerations for Sharing Subsystems” on page 21-109.

Variant Subsystem: Using Variant Subsystems for HDL Code Generation

Guideline ID

2.4.3

Severity

Mandatory

Description

The Variant Subsystem, Variant Model block is a template preconfigured to contain two Subsystem blocks to use as Variant Subsystem choices. At simulation time, a variant control decides which among the two Subsystem blocks is active. Therefore, you can use the Variant Subsystem to create two different configurations or subsystem behaviors and then specify the active configuration at simulation time.

- You cannot use a Variant Subsystem as the DUT. To generate code, place the Variant Subsystem inside the Subsystem that you want to use as the DUT. The file name and instance name of the generated code is unique to the active configuration that is chosen at code generation time.
- You cannot share multiple Variant Subsystem blocks by using the Variant Subsystem optimization.
- You must make sure that when verifying the functionality of the generated code, the active variant that you used when simulating the model is the same as the active variant that you used for generating HDL code.

For an example, open the model `hdlcoder_variant_subsystem_design.slx`. If you open the DUT Subsystem, you see a Variant Subsystem block, `Divide`. The Variant Subsystem has two different subsystems, `Recip` and `Op`. If you open the Block Parameters dialog box for the `Divide` Subsystem, you see the **Variant control expression** and the **Condition** that determines which Subsystem to enable during simulation. In this case, `Recip` is 1, and the `Recip` Subsystem becomes active during simulation.

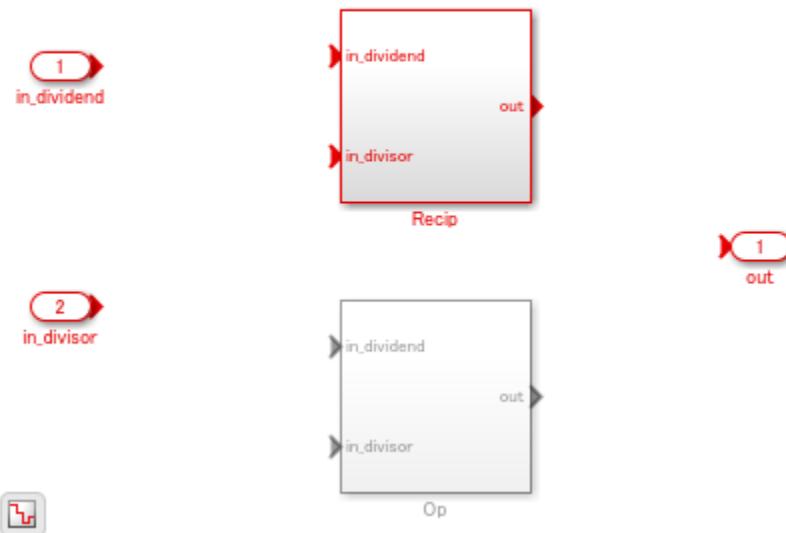
```
load_system('hdlcoder_variant_subsystem_design')
set_param('hdlcoder_variant_subsystem_design','SimulationCommand','Update')
open_system('hdlcoder_variant_subsystem_design/DUT/Divide')

variantRecip =
Simulink.Variant
    Condition: 'Recip == 1'

variantOp =
Simulink.Variant
    Condition: 'Recip == 0'

Recip =
1
```

- 1) Only subsystems can be added as variant choices at this level
- 2) Blocks cannot be connected at this level as connectivity is automatically determined at simulation, based on the active variant



To generate HDL code, run this command:

```
makehdl('hdlcoder_variant_subsystem_design/DUT')
```

A HDL file with the name Recip.vhd is generated because the code is generated for the Recip Subsystem which is active at code generation time.

Model References: Build Model Design Using Smaller Partitions

Guideline ID

2.4.4

Severity

Recommended

Description

Modeling a large design hierarchically can increase code generation time. If you specify generation of reports such as the traceability report, the code generation time can further increase significantly. To avoid such performance issues, it is recommended that you partition your design into smaller partitions. Use the Model block to unify a model that consists of smaller partitions. It also enables incremental code generation. You can generate HDL code for the parent model or the referenced model. To see the generated HDL code, in the `hdlsrc` folder, a folder is created for the parent model with a separate subfolder for the referenced model.

When generating the HDL test bench, if the test bench consists of blocks that operate with a continuous sample time, you can convert the DUT to a referenced model. This conversion enables the DUT to run at a fixed-step, discrete sample time. To learn more, see “Convert DUT Subsystem to Model Reference for Testbenches with Continuous Blocks” on page 21-34.

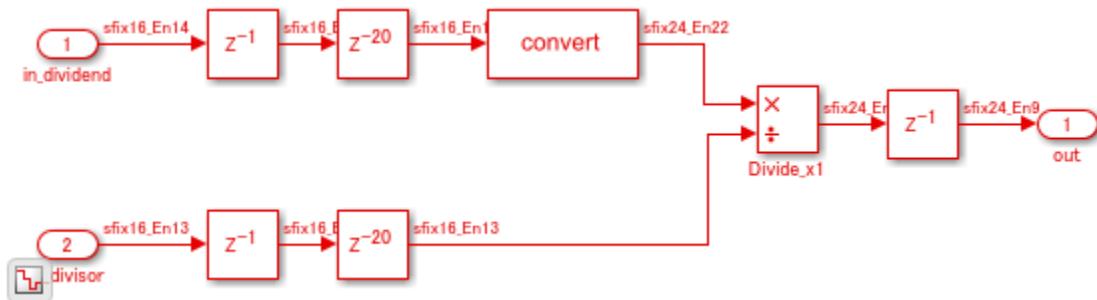
For an example, open the model `hdlcoder_divide_parentmodel.slx`. When you double-click the DUT Subsystem, you see a Model block that references the model `hdlcoder_divide_referencedmodel`.

```
load_system('hdlcoder_divide_parentmodel')
set_param('hdlcoder_divide_parentmodel', 'SimulationCommand', 'Update')
open_system('hdlcoder_divide_parentmodel/DUT')
```



To see the referenced model, double-click the Model block:

```
open_system('hdlcoder_divide_parentmodel/DUT/Model')
```



To generate HDL code, enter this command:

```
makehdl('hdlcoder_divide_parentmodel/DUT')
```

For more information, see “Model Referencing for HDL Code Generation” on page 27-2.

Block Settings of Enabled and Triggered Subsystems

Guideline ID

2.4.5

Severity

Mandatory

Description

A Triggered Subsystem is a subsystem that receives a control signal via a Trigger block. The Triggered Subsystem executes for one cycle each time a trigger event occurs. When you generate HDL code for a triggered subsystem:

- Do not use the Triggered Subsystem block as the DUT. Place the Triggered Subsystem inside another Subsystem block, and use that Subsystem as the DUT.
- Make sure that the initial condition of the Triggered Subsystem must be zero.
- You can add unit delays to the output signals of the Triggered Subsystem. The unit delays prevent the code generator from inserting additional bypass registers in the HDL code.
- Make sure that the **Use trigger signal as clock** setting does not result in timing mismatches when you simulate the testbench to verify the generated code. To learn more, see “Using Triggered Subsystems for HDL Code Generation” on page 23-16.

For other preferences when configuring the Triggered Subsystem block for HDL code generation, see “HDL Code Generation” on the Triggered Subsystem page.

An Enabled Subsystem is a subsystem that receives a control signal via an Enable block. The Enabled Subsystem executes at each simulation step where the control signal has a positive value. When you generate HDL code for an Enabled Subsystem:

- Do not use the Enabled Subsystem block as the DUT. Place the Enabled Subsystem inside another Subsystem block, and use that Subsystem as the DUT.
- Make sure that the initial condition of the Enabled Subsystem is zero.
- You can add unit delays to the output signals of the Enabled Subsystem. The unit delays prevent the code generator from inserting additional bypass registers in the HDL code.
- You can add a State Control block in **Synchronous** mode inside the Enabled Subsystem. The State Control block converts the Enabled Subsystem block to an Enabled Synchronous Subsystem block. This block generates more efficient and hardware-friendly HDL code. To learn more, see “Synchronous Subsystem Behavior with the State Control Block” on page 27-85.

For other preferences when configuring the Enabled Subsystem block for HDL code generation, see “HDL Code Generation” on the Enabled Subsystem page.

See Also

Functions

`makehdl`

Simulink Configuration Parameters

“Use trigger signal as clock” on page 17-41

Related Examples

- “Resettable Subsystem Support in HDL Coder™” on page 27-97

More About

- “Using Enabled and Triggered Subsystems”

Usage of Rate Change and Constant Blocks

You can follow these guidelines to learn how to use blocks that can perform rate conversions in your model and blocks from the Sources library such as Constant blocks in your design. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Usage of Rate Conversion Blocks

Guideline ID

2.5.1

Severity

Recommended

Description

There are various ways in which you can model rate transitions. How you model rate transitions determine the timing and resource requirements of your design. This guideline shows various approaches for modeling rate transitions.

Increasing the Sample Rate

This table illustrates the blocks that you can use to increase the sample rate of your design. When you use these blocks, leave the block parameters to the default settings.

Rate Conversion Approach

Block	Generates Bypass Register?	Generates Zero Padding?	Notes
Repeat	No	No	To use this block, you must have DSP System Toolbox installed.
Rate Transition	No	No	None
Upsample	Yes	Yes	To use this block, you must have DSP System Toolbox installed. When you use this block, consider the impact of the bypass register and the logic that inserts zero padding on hardware resource usage.

For the Rate Transition block, to upsample the input signal without incurring a unit delay, in the Block Parameters dialog box of the Rate Transition block:

- Clear the **Ensure data integrity during data transfer** check box. Clearing this check box makes the **Ensure deterministic data transfer (maximum delay)** check box to disappear.
- Configure the output port sample time of the block to be an integer multiple of the input port sample time. Specify a fractional value of $1/n$ for **Sample time multiple** where n is an integer.

You can choose any value for the block parameter **Output port sample time options** as long as **Sample time multiple** uses a value $1/n$.

When the input and output clocks are not synchronous to each other, to avoid insertion of a bypass register in the HDL code generated for the Repeat and Rate Transition blocks, insert one unit delay following the Repeat and Rate Transition blocks in your model.

Decreasing the Sample Rate

To reduce the sample rate, you can use a DownSample or a Rate Transition block. To use the Downsample block, you must have DSP System Toolbox installed. When you use these blocks, leave the block parameters to the default settings.

When downsampling the input signal, use the Rate Transition block because you can leave the block parameters to the default settings for HDL code generation. The **Ensure data integrity during data transfer** and **Ensure deterministic data transfer (maximum delay)** check boxes must be left selected. This mode generates an additional bypass register in the HDL code.

You use the **Initial Condition** parameter of the block when reducing the sample rate. This parameter setting is not used when the code generator increases the sample rate.

Use Discrete and Finite Sample Time for Constant Block

Guideline ID

2.5.2

Severity

Mandatory

Description

By default, the sample time of a Constant block is `inf`. When you connect a Constant block with sample time of `inf` to other blocks in your design, it hinders speed and area optimizations. Optimizations such as retiming, sharing, and streaming use the clock rate information to improve the speed and area of your design.

When you use the Constant block, set the sample time of the blocks to `-1`. To identify Constant blocks that have infinite sample time in your design, in the Simulink model window, In the **Debug** tab, on the **Information Overlays > Sample Time** section, select **Colors**. The Sample Time Legend then displays the Constant blocks that have `Inf` sample time.

You can identify and change the sample time of all Constant blocks to `-1` by using either of these approaches:

- Run a script that can programmatically change the sample time of the blocks to `-1`. For an example script, see “Identify and Programmatically Change and Display HDL Block Parameters” on page 21-29.
- Run the check “Check for infinite and continuous sample time sources” on page 38-16 in the HDL Code Advisor. If running the check fails, it displays sources such as Constant blocks that have infinite sample time. Select **Modify Settings** to update the sample time to `-1` or to inherit via backpropagation.

See Also

Functions

`makehdl`

More About

- “Multirate Model Requirements for HDL Code Generation” on page 23-6
- “Code Generation from Multirate Models” on page 23-2

Guidelines for Using Delays and Goto and From Blocks for HDL Code Generation

These guidelines illustrate the recommended settings for modeling delays in your model. You model delays by using blocks available in the **Discrete** Library. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Appropriate Usage of Delay Blocks as Registers

Guideline ID

2.6.1

Severity

Recommended

Description

For blocks in your model to be inferred as a flipflop on the target FPGA, use Delay blocks. You can specify a local reset and enable signal for each Delay block.

By default, the **Delay length** of the block is set to 2. In this case, the input to the block passes to the output after two time steps. If the **Delay length** is set to 0, the input passes to the output without any delay. The generated HDL code treats the block as a wire. To infer a flipflop or register on the target device, set the **Source** to dialog and specify a **Delay length** greater than zero.

Do not use the Unit Delay Enabled, Unit Delay Resettable, and Unit Delay Enabled Resettable blocks for HDL code generation. These blocks have been obsoleted. Instead, replace these blocks with the Unit Delay Enabled Synchronous, Unit Delay Resettable Synchronous, and Unit Delay Enabled Resettable Synchronous blocks. These blocks use the State Control block for synchronous hardware behavior. To perform this replacement in your model, run the model check “Check for obsolete Unit Delay Enabled/Resettable Blocks” on page 38-21.

Required HDL Settings for Goto and From Blocks

Guideline ID

2.6.2

Severity

Mandatory

Description

When you generate HDL code for the DUT Subsystem that uses From and Goto blocks:

- Do not use From and Goto blocks across the boundary of the DUT subsystem. To connect the input and output ports of the DUT, use Import and Outport blocks instead.
- Do not use From and Goto blocks across the boundary of an Atomic Subsystem. To connect the input and output ports of the DUT, use Import and Outport blocks instead.

- Scope of From and Goto blocks must be local to a subsystem hierarchy. Set **Tag Visibility** of the blocks to **local** or **Scoped**. HDL code generation does not support **Tag Visibility** of the blocks set to **global**.

Using From and Goto blocks across a subsystem hierarchy can impact the readability of the model. Before generating HDL code, it is recommended that you use From and Goto blocks in the same subsystem and use **local** or **Scoped** visibility. When you generate HDL code, in the generated model, each Goto and From block becomes a pair of From and Goto subsystems connected back to back.

See Also

Functions

`makehdl`

More About

- “Goto and From Blocks as a Signal Routing Alternative”

Modeling Efficient Multiplication and Division Operations for FPGA Targeting

These guidelines illustrate the recommended settings when using Divide and Product blocks in your model for improved area and timing on the target FPGA. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Designing Multipliers and Adders for Efficient Mapping to DSP Blocks on FPGA

Guideline ID

2.7.1

Severity

Strongly Recommended

Description

Digital signal processing (DSP) algorithms use several multipliers and accumulators. FPGA devices provided by vendors such as Xilinx® and Intel® contain dedicated DSP slices. These small size, high speed, DSP slices contain several multipliers and accumulators that make FPGA devices best suited for DSP applications.

The architecture of DSP slices varies widely across the different FPGA vendors and across different families of devices provided by the same vendor. To map your Simulink® model containing adders, multipliers, and delays to DSP slices, adapt your model to the DSP slice architecture by taking into consideration:

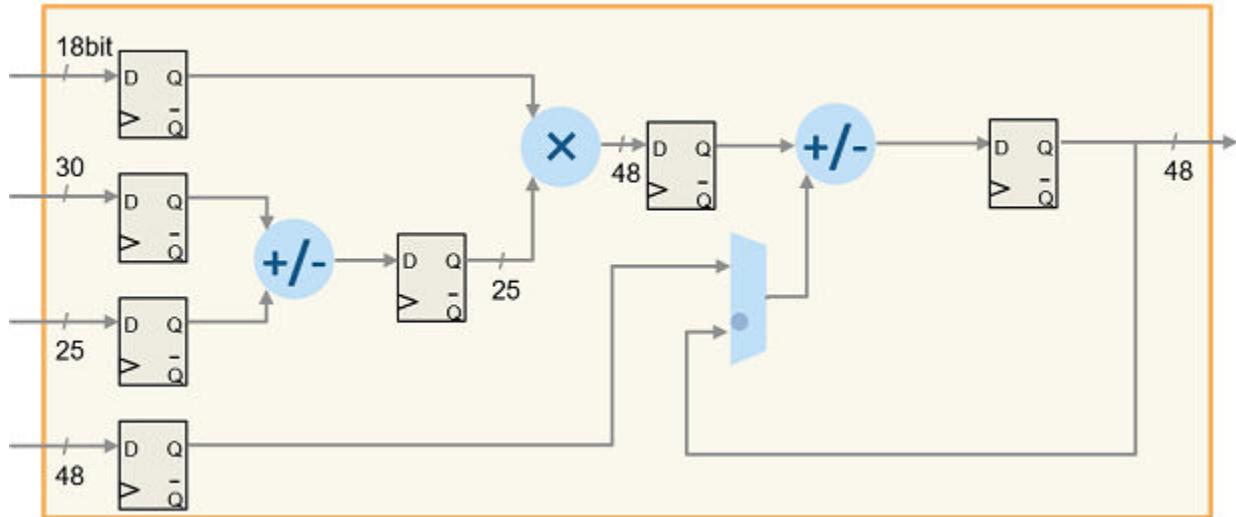
- Arrangement of flipflops, adders, and multipliers in the DSP slice.
- Rounding and saturation settings.
- Bit widths of the adders and multipliers. For efficient mapping, use bit widths in your model that are less than or equal to the bit widths of the DSP unit.

When the bit widths in your model become larger than the bit widths of the DSP, your design does not fit onto one DSP. In this case, multiple DSPs or additional logic is required.

You can map these blocks in your model to DSP blocks on an FPGA:

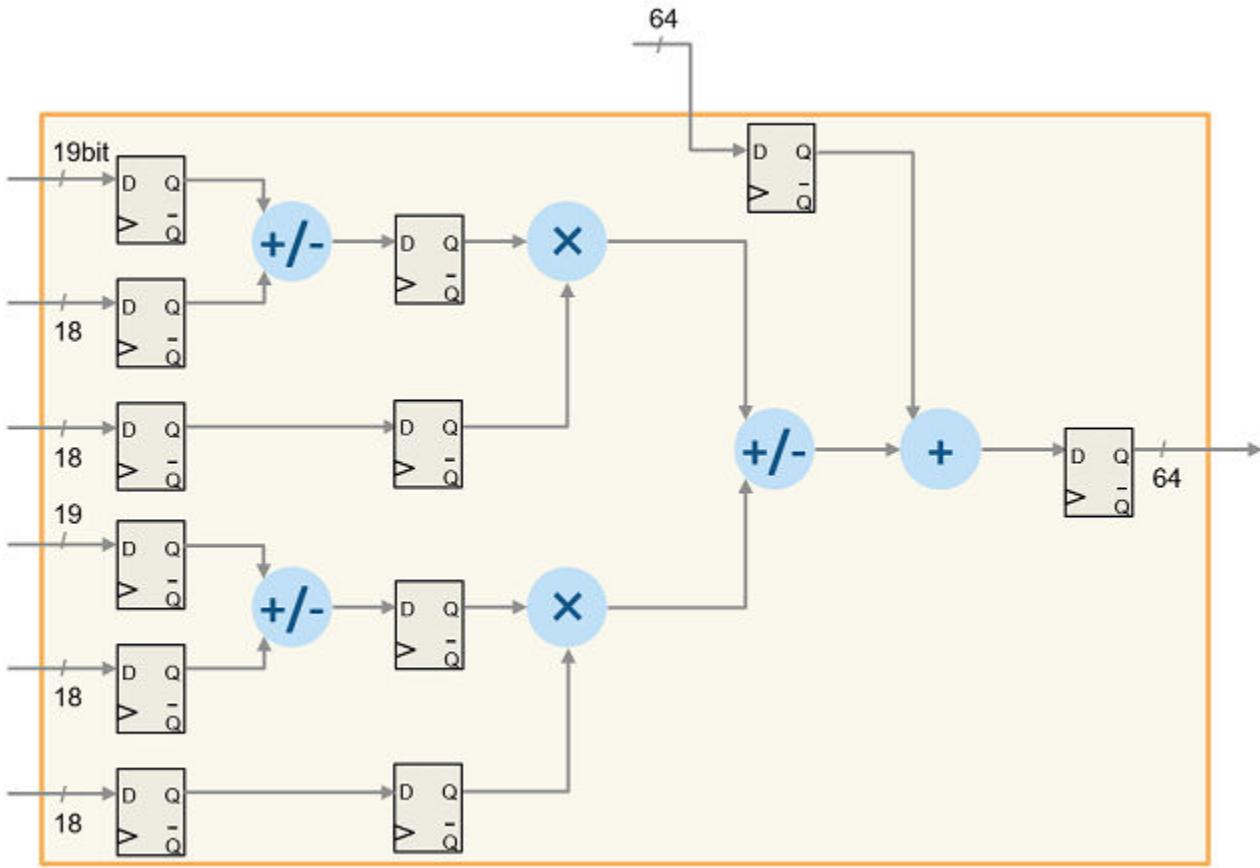
- Add and Sum
- Delay
- Product
- Multiply-Add
- Multiply-Accumulate

This figure illustrates the Xilinx DSP architecture. Xilinx 7 series FPGAs have dedicated DSP slices that use this architecture. The DSP architecture consists of input registers, pre-adder, 25x18 multiplier, intermediate registers, post-adder, and an output register.



For more information, see [DSP48E1 Slice Overview](#) in the Xilinx documentation.

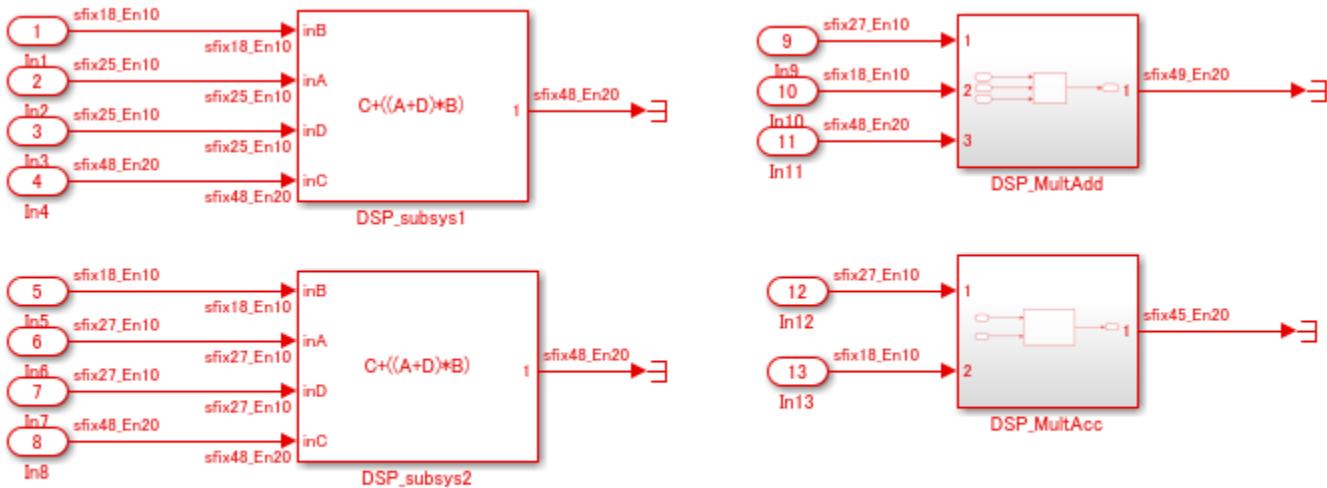
This figure illustrates the Intel DSP architecture. This DSP architecture for Stratix® V devices is a variable precision DSP architecture. The DSP blocks can have bit widths of 9, 18, 27, and 36 bits, and 18x25 complex multiplication for FFTs.



For more information, see **DSP Block Architecture** in the Intel documentation.

To learn how you can design your algorithm to map to this DSP unit, open the model `hdlcoder_multiplier_adder_DSP.slx`

```
open_system('hdlcoder_multiplier_adder_DSP')
set_param('hdlcoder_multiplier_adder_DSP', 'SimulationCommand', 'Update')
```



Copyright 2014–2019 The MathWorks, Inc.

The model consists of two subsystems **dsp_subsys1** and **dsp_subsys2** that implement the operation $C + ((A+D)*B)$. You can also implement this operation by using Multiply-Add or Multiply-Accumulate blocks as illustrated by subsystems **DSP_MultAdd** and **DSP_MultAcc**.

dsp_subsys1 implements the operation $C + ((A+D)*B)$ by using bit widths that equal the DSP on a Xilinx 7 series FPGA. If you open the HDL Workflow Advisor and deploy this Subsystem onto a Xilinx Virtex® 7 FPGA, the entire design fits exactly onto one DSP slice.

Passed Synthesis

Parsed resource report file: [DSP_subsys1_utilization_synth.rpt](#).

Resource summary	
Resource	Usage
Slice LUTs	0
Slice Registers	0
DSPs	1
Block RAM Tile	0
URAM	0

dsp_subsys2 implements the same operation by using bit widths that are larger than the DSP on a Xilinx FPGA. If you deploy this Subsystem onto a Xilinx Virtex 7 FPGA, you see that the entire design fits onto one DSP slice and uses additional slice logic.

Passed Synthesis

Parsed resource report file: [DSP_subsys2_utilization_synth.rpt](#).

Resource summary	
Resource	Usage
Slice LUTs	141
Slice Registers	210
DSPs	1
Block RAM Tile	0
URAM	0

Use ShiftAdd Architecture of Divide Block for Fixed-Point Types

Guideline ID

2.7.2

Severity

Recommended

Description

When you use fixed-point data types as inputs to the Divide block, specify the HDL Architecture of the block as **ShiftAdd** and then set the HDL block property “UsePipelines” on page 22-24 to on. In this architecture, the block computes the result by using multiple shift and add operations. The operations are pipelined to achieve higher clock frequencies on the target FPGA device.

When you use floating-point data types as inputs to the Divide block, leave the HDL Architecture to default value of **Linear** and set the **Floating Point IP Library** to **Native Floating Point**.

See Also

Functions

`makehdl`

Blocks

Multiply-Accumulate | Multiply-Add

More About

- “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103

Using Persistent Variables and fi Objects Inside MATLAB Function Blocks for HDL Code Generation

These guidelines illustrate the recommended settings when using persistent variables inside MATLAB Function blocks in your model. The MATLAB Function block is available in the **User-Defined Functions** block library. A persistent variable in a MATLAB Function block acts similar to a delay element in your Simulink model.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Update Persistent Variables at End of MATLAB Function

Guideline ID

2.8.1

Severity

Strongly Recommended

Description

To make sure that the persistent variables inside the MATLAB Function block map to a register on the target FPGA device, update the persistent variable at the end of the MATLAB code inside the MATLAB Function block. Do not update the persistent variable before its value is read or used by the function.

For example, this MATLAB code is not recommended because the function updates the persistent variable FF0 is updated before the value is read at the output.

```
function FF_out0 = fcn(FF_in)
 %#codegen

persistent FF0

if isempty(FF0)
    FF0 = zeros(1, 'like', FF_in);
end

% Incorrect order of FF update
FF0 = FF_in

% Output FF0. FF_out0 is NOT delayed
FF_out0 = FF0;
```

This MATLAB code is recommended because the value is written to FF0 at the end of the code.

```
function FF_out0 = fcn(FF_in)
 %#codegen

persistent FF0

if isempty(FF0)
    FF0 = zeros(1, 'like', FF_in);
```

```

end

% Output FF0
FF_out0 = FF0;

% Write FF update at the end of the code
FF0 = FF_in

```

Avoid Algebraic Loop Errors from Persistent Variables inside MATLAB Function Blocks

Guideline ID

2.8.2

Severity

Mandatory

Description

When your Simulink® model contains MATLAB Function blocks inside a feedback loop and uses persistent variables, compiling or simulating the model might generate algebraic loop errors. To simulate the model and generate HDL code, use nondirect feedthrough.

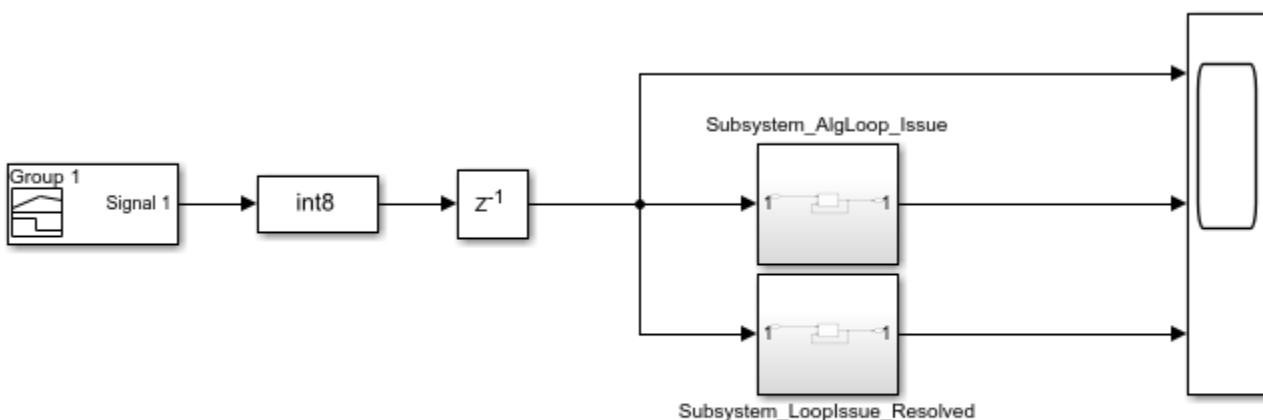
In certain cases, the persistent delay in the MATLAB Function block inside a feedback loop causes an algebraic loop error. When you use direct feedthrough, the output of the block directly depends on the input. When **Allow direct feedthrough** is cleared, the output of the block depends on the internal state and properties and does not depend on the input. The nondirect feedthrough semantics prevents algebraic loops errors by making the outputs depend only on the state.

For an example, open the model `hdlcoder_MLFB_avoid_algebraic_loops`.

```

modelname = 'hdlcoder_MLFB_avoid_algebraic_loops';
blkname = 'hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLop_Issue/MATLAB Function1';
open_system(modelname)

```



When you simulate the model, the algebraic loop error message is displayed. The MATLAB Function block `hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue/MATLAB Function` uses a persistent variable inside a MATLAB Function block.

```
open_system(blkname)

function y = fcn(u0, u1)
%#codegen

persistent tmp
if isempty(tmp)
    tmp = int8(0);
end

% There is a delay but algebraic loop is detected
y = tmp;

if u0 ~= 0
    tmp = int8(u1 + u0);
else
    tmp = int8(u1);
end
```

To avoid this error, use nondirect feedthrough. To specify nondirect feedthrough at the command line, create a `MATLABFunctionConfiguration` object by using `get_param` function, and then change the property value `AllowDirectFeedthrough`:

```
MLFBCfg = get_param(blkname, 'MATLABFunctionConfiguration');
MLFBCfg.AllowDirectFeedthrough = 0;
```

See also `MATLABFunctionConfiguration`.

To specify nondirect feedthrough from the UI:

- 1 Open the MATLAB Function block `MATLAB Function1`.
- 2 Opens the Ports and Data Manager dialog box. On the MATLAB® Editor, click **Edit Data**.
- 3 On the Ports and Data Manager dialog box, clear **Allow direct feedthrough** check box.

See also “Prevent Algebraic Loop Errors in MATLAB Function and Stateflow Blocks”.

The model now simulates without algebraic errors. You can now generate HDL code for the Subsystem block `Subsystem_AlgLoop_Issue`.

```
open_system(modelname)
set_param('hdlcoder_MLFB_avoid_algebraic_loops', 'SimulationCommand', 'Update')
makehdl('hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue')

### Generating HDL for 'hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue')">Subsystem_AlgLoop_Issue</a>.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_MLFB_avoid_algebraic_loops'.
```

```
### Working on hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue/MATLAB Function1 as h
### Working on hdlcoder_MLFB_avoid_algebraic_loops/Subsystem_AlgLoop_Issue as hdsrc\hdlcoder_ML
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\6\tpa
### HDL check for 'hdlcoder_MLFB_avoid_algebraic_loops' complete with 0 errors, 0 warnings, and 0
### HDL code generation complete.
```

Use `hdlfimath` Setting and Specify fi Objects inside MATLAB Function Block

Guideline ID

2.8.3

Severity

Strongly Recommended

Description

`fimath` properties define the rules for performing arithmetic operations on `fi` objects. To specify `fimath` properties that govern arithmetic operations, use a `fimath` object. To see the default `fimath` property settings, run this command:

```
F = fimath
```

```
F =
```

```
RoundingMethod: Nearest
OverflowAction: Saturate
ProductMode: FullPrecision
SumMode: FullPrecision
```

The default `fimath` settings reduce rounding errors and overflows. However, HDL code generation for a MATLAB Function block that uses these settings can incur additional resource usage on the target FPGA device. To avoid the additional logic, use `hdlfimath`. The `hdlfimath` function is a utility that defines `fimath` properties optimized for HDL code generation. To see the default `hdlfimath` settings, run this command:

```
H = hdlfimath
```

```
H =
```

```
RoundingMethod: Floor
OverflowAction: Wrap
ProductMode: FullPrecision
SumMode: FullPrecision
```

HDL code generation for a MATLAB Function block that uses these settings avoids the additional resource usage and saves area on the target FPGA device.

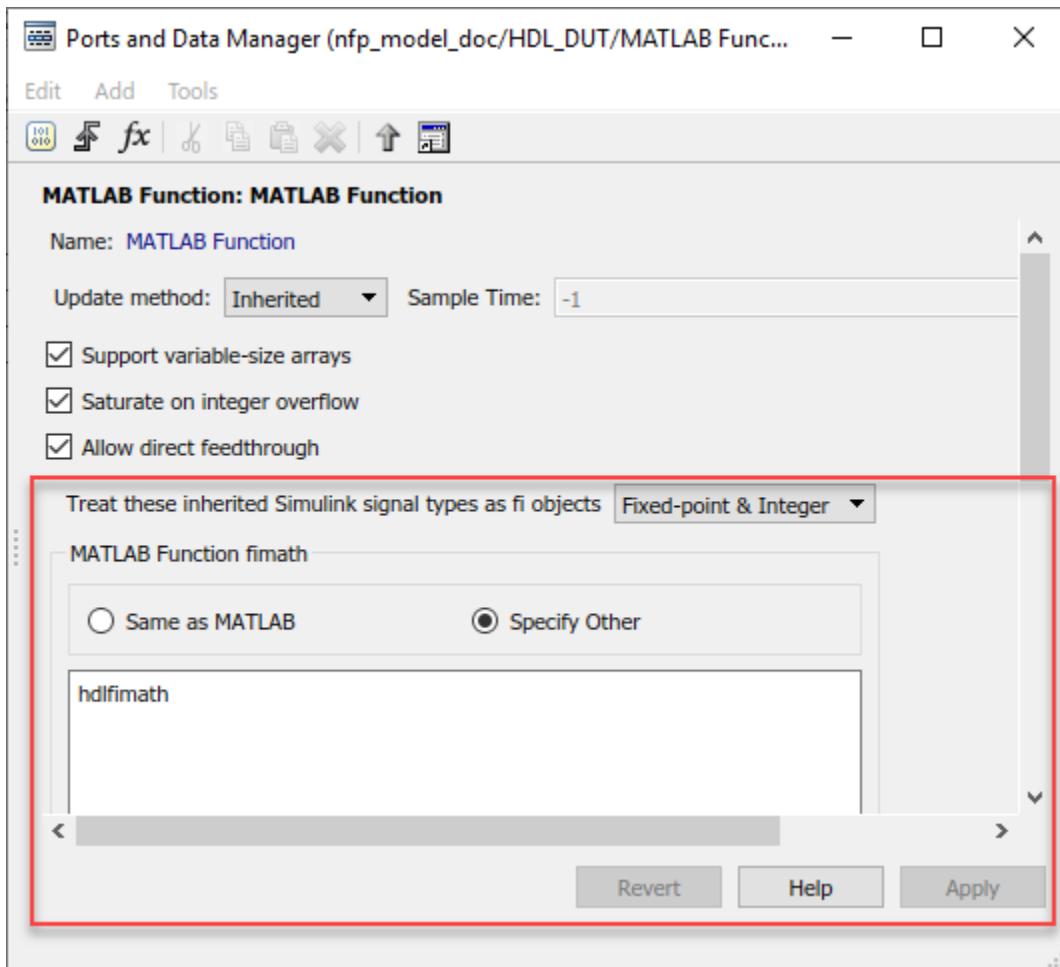
To specify these settings for a MATLAB Function block:

- Double click the MATLAB Function block and select **Edit Data** on the MATLAB Editor.

- In the Ports and Data Manager dialog box, for:
 - **Treat these inherited Simulink signal types as fi objects**, select Fixed-point & Integer.

If you use the default Fixed-point setting, fixed-point data types specified by using fi objects and built-in integer types such as int8 and int16 are treated differently. When you use built-in integer types, the output data type for integer type calculations becomes the same as the input data type. The bit width is not expanded to perform numeric calculation.

- **MATLAB Function fimath**, select **Specify Other** and then enter hdlfimath.



To perform rounding operations that are different from the default hdlfimath settings, specify these settings explicitly by using the fi object as illustrated below.

```
A = fi(4.9, 1, 8)
```

```
A =
```

```
4.8750
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
```

```
WordLength: 8
FractionLength: 4

B = fi(2.3, 1, 10)

B =
2.2969

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 10
    FractionLength: 7

C = fi(A+B, 'RoundingMethod', 'Nearest', 'OverflowAction', 'Saturate')

C =
7.1719

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 12
    FractionLength: 7

    RoundingMethod: Nearest
    OverflowAction: Saturate
    ProductMode: FullPrecision
    SumMode: FullPrecision
```

To make sure that the fimath settings are specified according to hdfimath for the MATLAB Function block, you can run the check “Check for MATLAB Function block settings” on page 38-19.

See Also

Blocks

MATLAB Function

Functions

fimath | makehdl

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “Initialize Persistent Variables in MATLAB Functions” on page 28-22
- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59

Guidelines for HDL Code Generation Using Stateflow Charts

These guidelines illustrate the recommended settings when using Stateflow charts in your model. The Stateflow Chart block is available in the **Stateflow** block library. By using Stateflow charts, you can model delays in your Simulink model.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Choose State Machine Type based on HDL Implementation Requirements

Guideline ID

2.9.1

Severity

Strongly Recommended

Description

HDL Coder supports code generation for Mealy and Moore Stateflow charts. Do not use MATLAB Function blocks to model either Mealy or Moore state machines.

To specify whether you want a Mealy or Moore state machine, in the Chart (Stateflow) properties, specify the **State Machine Type**. Do not use **Classic** because it affects readability of the generated HDL code. Choose the **State Machine Type** depending on how you want the Stateflow semantics to map to a hardware implementation. See “Hardware Realization of Stateflow Semantics” on page 28-6.

When you use Mealy charts, the outputs depend on the current state and inputs. By using Mealy charts, you can more easily define state transitions which makes these charts more flexible to use. The generated HDL code from Mealy charts may be less readable.

For Moore charts, the outputs depend only on the current state. The generated HDL code from Moore charts is more readable. Moore charts restrict flexibility in defining state transitions.

Specify Block Configuration Settings of Stateflow Chart

Guideline ID

2.9.2

Severity

Strongly Recommended

Description

When you use Stateflow Chart (Stateflow) blocks in your model for HDL code generation, use these recommended settings:

- For **Action Language**, use MATLAB
- For **Update method**, use Discrete or Inherited. Do not use Continuous.

Moore Chart

- Enable **Initialize Outputs Every Time Chart Wakes Up**
- Disable **Support Variable-Size Arrays**
- Disable **Support Variable-Size Arrays**

Mealy Chart

- Enable **Execute (Enter) Chart at Initialization**
- Enable **Initialize Outputs Every Time Chart Wakes Up**
- Disable **Enable Super Step Semantics**
- Disable **Support Variable-Size Arrays**

To make sure that these settings are specified for the Stateflow Chart, you can run the check “Check for Stateflow chart settings” on page 38-20.

Insert Unconditional Transition State for Else Statement in HDL Code

Guideline ID

2.9.3

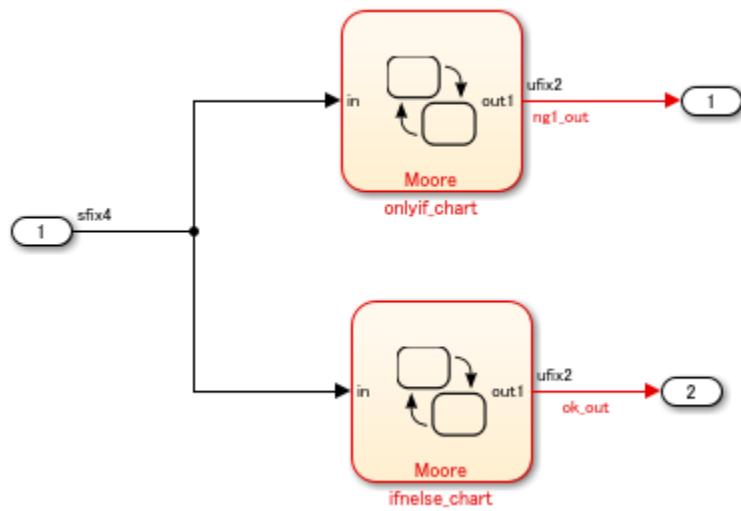
Severity

Recommended

Description

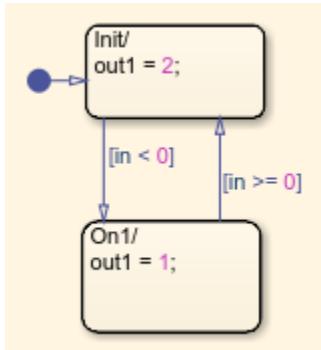
When you use Stateflow® charts for HDL code generation, insert unconditional states in the chart. The HDL code generated for such a chart contains an else branch with the if statement. The presence of an else branch prevents the third-party tool from inferring a latch when you deploy the HDL code. For example, open the model `hdlcoder_chart_ifnelsecond`.

```
open_system('hdlcoder_chart_ifnelsecond')
set_param('hdlcoder_chart_ifnelsecond', 'SimulationCommand', 'Update')
open_system('hdlcoder_chart_ifnelsecond/dut_chart')
```



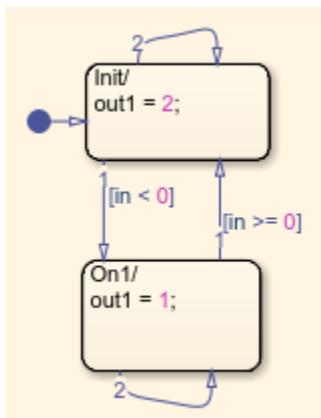
The model contains two Stateflow Moore Charts. The chart `onlyif_chart` implements a simple condition that outputs `out1` based on `in1`.

```
open_system('hdlcoder_chart_ifnelsecond/dut_chart/onlyif_chart')
```



The Chart block `ifnelse_chart` is the same as `onlyif_chart` and has an unconditional transition state.

```
open_system('hdlcoder_chart_ifnelsecond/dut_chart/ifnelse_chart')
```



To generate HDL code for the DUT, run this command:

```
makehdl('hdlcoder_chart_ifnelsecond/dut_chart')
```

The HDL code generated for the `onlyif_chart` does not contain an else condition. Do not deploy this code to a target device because synthesis tools might infer a latch.

```
case (is_onlyif_chart)
    is_onlyif_chart_IN_Init :
        begin
            if (in < 4'sb0000) begin
                is_onlyif_chart_temp = is_onlyif_chart_IN_0n1;
            end
        end
    default :
        begin
            //case IN_0n1:
            if (in >= 4'sb0000) begin
                is_onlyif_chart_temp = is_onlyif_chart_IN_Init;
            end
        end
endcase
is_onlyif_chart <= is_onlyif_chart_temp;
```

The HDL code generated for the `ifnelse_chart` contains an else statement for the unconditional transition state. This code is recommended for deployment to the target FPGA device.

```
case (is_ifnelse_chart)
    is_ifnelse_chart_IN_Init :
        begin
            if (in < 4'sb0000) begin
                is_ifnelse_chart_temp = is_ifnelse_chart_IN_On1;
            end
            else begin
                is_ifnelse_chart_temp = is_ifnelse_chart_IN_Init;
            end
        end
    default :
        begin
            //case IN_On1:
            if (in >= 4'sb0000) begin
                is_ifnelse_chart_temp = is_ifnelse_chart_IN_Init;
            end
            else begin
                is_ifnelse_chart_temp = is_ifnelse_chart_IN_On1;
            end
        end
    endcase
is_ifnelse_chart <= is_ifnelse_chart_temp;
```

See Also

Functions

`makehdl`

More About

- “Bitwise Operations in MATLAB for HDL Code Generation” on page 1-59

Simulink Data Type Considerations

You can follow these guidelines to learn the recommended data type settings that you want to use in your Simulink model for HDL code generation. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Use Boolean for Logical Data and Ufix1 for Numerical Data

Guideline ID

2.10.1

Severity

Mandatory

Description

Boolean and the fixed-point type, `ufix1`, are both 1-bit data types in MATLAB and Simulink. These types are treated differently.

- Use Boolean for control logic signals such as enable and local reset signals. If you want to calculate a Boolean signal with a fixed-point data type, use a Data Type Conversion to convert the signal to a `fixdt(0,1,0)` type.
- To perform numeric calculations, use `fixdt(0,1,0)`. Sometimes, the output bit width can become larger than the bitwidth. To perform such operations, use the `Inherit: Inherit via internal rule` setting, because of the `numerictype` property of `fixdt(0,1,0)`.

Specify Data Type of Gain Blocks

Guideline ID

2.10.2

Severity

Recommended

Description

Gain blocks have a **Gain** parameter and an **Output data type** setting. It is recommended that you use fixed-point data types for these settings. In the Block Parameters dialog box of the Gain block:

- Specify a `Simulink.NumericType` object, such as `fixdt(1, 16, 8)`.
- Make sure that the **Gain** parameter of the block does not use a round parameter value. To avoid rounding of the gain value, you can specify a `fi` object, such as `fi(3.44,0,8,4)`.
- Avoid using `Inherit:Inherit via internal rule`. This setting can result in an erroneous data type being assigned to the block, thereby resulting in an HDL code generation error.

Enumerated Data Type Restrictions

Guideline ID

2.10.3

Severity

Mandatory

Description

Certain optimizations such as pipelining and resource sharing do not work seamlessly in the presence of enumerated data types. It is recommended that you use enumerated types on an as needed basis. HDL code generation has certain restrictions when modeling with enumerated types.

- You cannot use an enumerated data type for the input or output port of the top-level DUT.
- You must use monotonically increasing enumeration values. For example, see this code:

```
classdef BasicColors < Simulink.IntEnumType
enumeration
    Red(0)
    Yellow(1)
    Blue(2)
end
methods (Static)
    function retVal = getDefaultValue()
        retVal = BasicColors.Blue;
    end
end
end
```

- You cannot perform arithmetic operations such as *, /, -, and + with enumeration values.
- You cannot perform comparison operations such as >, <, >=, <=, ==, and ~= with enumeration values. You can perform a <> operation or a conditional branch such as if or switch.

See Also

Functions

`makehdl`

More About

- “Signal and Data Type Support” on page 10-2

Resource Sharing Settings for Various Blocks

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations. To learn more about how resource sharing works, see “Resource Sharing” on page 24-32.

You can follow these guidelines to learn how to use the resource sharing optimization effectively with blocks such as Add and Product. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Resource Sharing of Add Blocks

Guideline ID

3.1.1

Severity

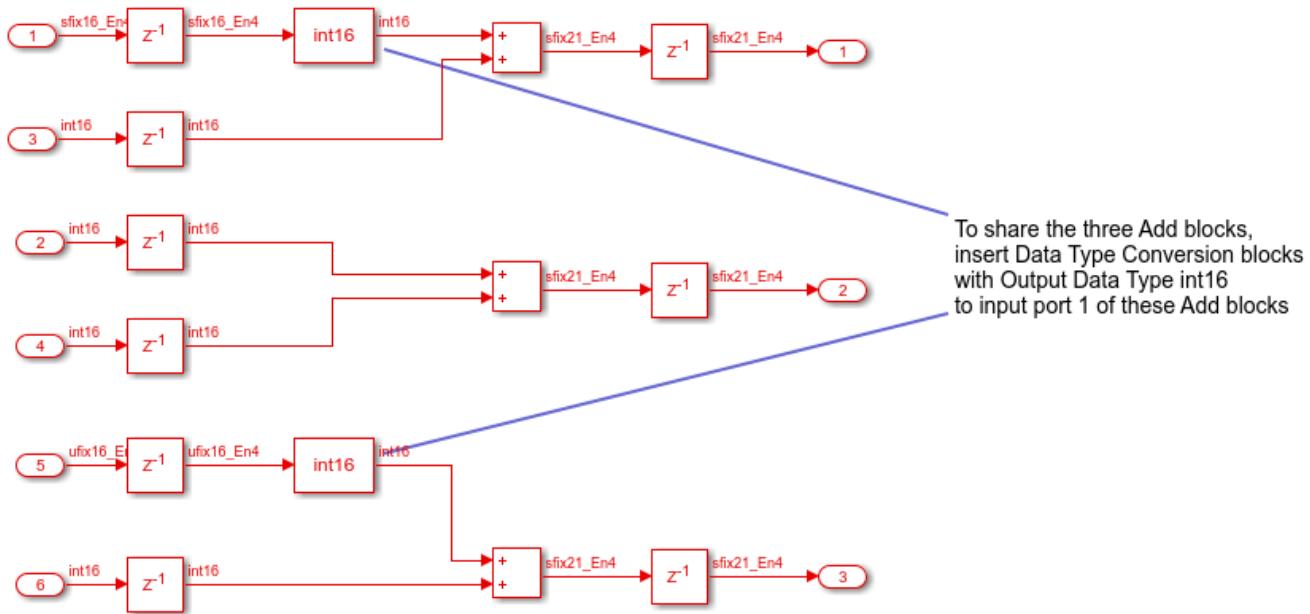
Recommended

Description

To share multiple Add blocks:

- Select the **Share Adders** setting.
- Leave the **Adder sharing minimum bitwidth** to 0.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than 1.
- Specify the “StreamingFactor” on page 22-24 for Add blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 22-23 for Add blocks with scalar inputs or outputs.
- Make sure that the input word-lengths of the Add blocks match.

For example, this figure illustrates a model containing three Add blocks placed inside a Subsystem with **SharingFactor** of 3. To share the Add blocks, you must insert Data Type Conversion blocks with **Output data type** set to **int16** so that the input word lengths match.



Resource Sharing of Gain Blocks

Guideline ID

3.1.2

Severity

Recommended

Description

When you share multiple Gain blocks in your design, the optimization inserts serialization and deserialization logic to share resources. This additional logic can become an area overhead if you are not sharing a large number of resources. Therefore, if your design does not contain a large number of Gain blocks to share, it is recommended that you disable the resource sharing optimization. To share multiple Gain blocks:

To share multiple Gain blocks:

- Determine how to implement the Gain block. HDL Coder does not share Gain blocks in either of these cases:
 - **ConstMultiplierOptimization** parameter set to `csd` or `fcsd`.
 - **Gain** parameter is a power of two.

In both these cases, the code generator uses a cast operation to replace the multiplier operations with shift and add or subtract operations, which causes sharing to be unsuccessful. In addition, if the **Gain** parameter is 0 or 1, then resource sharing requires no additional logic.

- Specify the “StreamingFactor” on page 22-24 for Gain blocks with vector inputs or outputs.

- Specify the “SharingFactor” on page 22-23 for Gain blocks with scalar inputs or outputs.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than 1.
- Use the same synthesis attribute settings if you specify the **DSPStyle** block property for the Gain blocks. HDL Coder does not share multipliers that have different synthesis attribute settings.

Resource Sharing of Product Blocks

Guideline ID

3.1.3

Severity

Recommended

Description

To share multiple Product blocks:

- Specify 18 as the **Multiplier partitioning threshold** when targeting Xilinx devices and 25 as the threshold when targeting Intel devices. This setting creates more resource sharing opportunities for multipliers with a wide bit width, which reduces the use of DSPs on the FPGA.
- Specify the **Multiplier promotion threshold** if you want to share Product blocks that have different word-lengths. The multiplier promotion threshold is the maximum word-length by which HDL Coder promotes a multiplier for sharing with other multipliers.
- Leave the **Share Multipliers** setting enabled and the **Multiplier sharing minimum bitwidth** to 0.
- Specify the “StreamingFactor” on page 22-24 for the subsystems that contain Product blocks with vector inputs or outputs.
- Specify the “SharingFactor” on page 22-23 for the subsystems that contain Product blocks with scalar inputs or outputs.
- Use a Gain block instead of a Product block when one of the inputs to the Product block is a constant. Use the constant value as the **Gain** parameter of the Gain block. If you use floating-point data types in the **Native Floating Point** mode, HDL Coder converts the Product block to a Gain block automatically during code generation. To learn more, see “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-17.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than 1.
- Use the same synthesis attribute settings if you specify the **DSPStyle** block property for the Product blocks. HDL Coder does not share multipliers that have different synthesis attribute settings.

Resource Sharing of Multiply-Add Blocks

Guideline ID

3.1.4

Severity*Recommended***Description**

To share multiple Multiply-Add blocks:

- Leave the **Share Multiply-Add blocks** setting enabled and the **Multiply-Add block sharing minimum bitwidth** set to 0.
- Determine whether to perform resource sharing at the existing clock rate or at a higher clock rate. To use a higher clock rate, specify an **Oversampling factor** greater than 1.
- Specify the “SharingFactor” on page 22-23.

See Also

[Simulink Configuration Parameters](#)

[Resource Sharing of Adders and Multipliers](#) | [Resource Sharing of Multiply-Add and Other Blocks](#)

Related Examples

- “Resource Sharing of Multipliers to Reduce Area” on page 8-24
- “Resource Sharing For Area Optimization” on page 24-40
- “Single-rate Resource Sharing Architecture” on page 24-50

More About

- “Streaming” on page 24-29
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 21-109

Resource Sharing of Subsystems and Floating-Point IPs

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations. To learn more about how resource sharing works, see “Resource Sharing” on page 24-32.

You can follow these guidelines to learn how to use the resource sharing optimization effectively for subsystems such as atomic subsystems and MATLAB Function blocks, and with floating-point IPs. Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

General Considerations for Sharing of Subsystems

Guideline ID

3.1.5

Severity

Recommended

Description

To share resources for identical subsystems, such as when grouping Product, Add, and Delay blocks to map to one DSP slice, the subsystems to be shared must be Atomic Subsystem blocks or MATLAB Function blocks.

- Determine whether you want to share resources at the existing clock rate or at a higher clock rate.
- Sharing of enabled subsystems is not supported. For sharing resources, use atomic subsystems without enable semantics.
- Specify a “SharingFactor” on page 22-23 that is greater than or equal to the number of subsystems that you want to share.

For example, if you have 10 atomic subsystems, and you set the **SharingFactor** to 5, HDL Coder cannot implement the resource sharing to 2 instances of the subsystem. To share the subsystems, divide the subsystems, and then share the instances of the smaller subsystems.

- Check the **SharingFactor** that you specify for various subsystems. The resource sharing optimization overclocks the shared resources by the LCM (Least Common Multiple) of the **SharingFactor** of various subsystems.

For example, if you specify a **SharingFactor** of 5 for one Subsystem, and a **SharingFactor** of 7 for another Subsystem, the resource sharing optimization overclocks the shared resources by 35. In such cases, it is recommended that you use the same **SharingFactor** for both subsystems, such as 5 or 7. To learn more about this calculation, see “How Resource Sharing Works” on page 24-32.

Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks

Guideline ID

3.1.6

Severity

Recommended

Description

HDL Coder shares MATLAB Function blocks that have:

- The same Simulink checksum. Use `Simulink.Subsystem.getChecksum` to determine the checksum.
- The same HDL block properties.

Make sure that the blocks do not use:

- Persistent variables
- Loop streaming
- Output pipelining

By using the MATLAB Datapath architecture, you can share resources inside the MATLAB Function block and across the MATLAB Function block with other blocks in your Simulink model. When you use this architecture, the code generator treats the MATLAB Function block like a regular Subsystem block. This capability enables you to more widely apply various speed and area optimizations with MATLAB Function blocks. See “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-146.

Sharing of Atomic Subsystems

Guideline ID

3.1.7

Severity

Recommended

Description

HDL Coder can share Atomic Subsystem blocks that have the same Simulink checksum and the same HDL block properties.

To share Atomic Subsystem blocks, the state elements that the blocks can contain are:

- Delay
- Unit Delay
- Unit Delay Enabled Synchronous

- Unit Delay Resettable Synchronous
- Unit Delay Enabled Resettable Synchronous

The state elements must have the **Initial condition** parameter set to 0.

Sharing of atomic subsystems inside enabled subsystems with synchronous semantics is not supported. To share resources, use enabled subsystems with classic semantics.

You cannot share atomic subsystems that contain the following blocks or block implementations:

- Detect Change
- Discrete Transfer Fcn
- HDL FFT
- HDL FIFO
- Math Function (conj, hermitian, transpose)
- MATLAB Function blocks that contain persistent variables
- Sqrt
- Cascade architecture (MinMax, Product, Sum)
- CORDIC architecture
- Reciprocal Newton architecture
- Filter blocks including Discrete FIR Filter
- Communications Toolbox blocks
- DSP System Toolbox blocks, except Discrete FIR Filter
- Stateflow blocks
- Blocks that are not supported for delay balancing. For details, see “Delay Balancing Limitations” on page 24-64.

HDL Coder can share Atomic Subsystem blocks that have the same Simulink checksum and the same HDL block properties.

If you want to share Atomic Subsystem blocks, the state elements that the blocks can contain are:

- Delay
- Unit Delay
- Unit Delay Enabled Synchronous
- Unit Delay Resettable Synchronous
- Unit Delay Enabled Resettable Synchronous

The state elements must have the **Initial condition** parameter set to 0.

You cannot share atomic subsystems that contain the following blocks or block implementations:

- Detect Change
- Discrete Transfer Fcn
- HDL FFT
- HDL FIFO

- Math Function (conj, hermitian, transpose)
- MATLAB Function blocks that contain persistent variables
- Sqrt
- Cascade architecture (MinMax, Product, Sum)
- CORDIC architecture
- Reciprocal Newton architecture
- Filter blocks including Discrete FIR Filter
- Communications Toolbox blocks
- DSP System Toolbox blocks, except Discrete FIR Filter
- Stateflow blocks
- Blocks that are not supported for delay balancing. For details, see “Delay Balancing Limitations” on page 24-64.

Resource Sharing of Floating-Point IPs

Guideline ID

3.1.8

Severity

Recommended

Description

To share multiple:

- Floating-point adders, set ShareAdders to on.
- Floating-point multipliers, make sure ShareMultipliers is set to on.
- Other floating-point resources, set ShareFloatingPointIP to on.

See also **Modeling with Native Floating Point**.

See Also

Simulink Configuration Parameters

[Resource Sharing of Adders and Multipliers](#) | [Resource Sharing of Multiply-Add and Other Blocks](#)

Related Examples

- “Resource Sharing of Multipliers to Reduce Area” on page 8-24
- “Resource Sharing For Area Optimization” on page 24-40
- “Single-rate Resource Sharing Architecture” on page 24-50

More About

- “Resource Sharing” on page 24-32

- “Streaming” on page 24-29

Distributed Pipelining and Clock-Rate Pipelining Guidelines

The code generator introduces registers when you specify certain block implementations or use certain settings. You can follow these guidelines to learn more about these registers and how you can use them to optimize the timing of your design.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see “HDL Modeling Guidelines Severity Levels” on page 21-2.

Clock-Rate Pipelining Guidelines

Guideline ID

3.2.1

Severity

Informative

Description

In most cases, the code generator introduces the registers in regions that run slower than the clock rate. To avoid or minimize additional latency, you can run these registers at the fast clock rate by using clock-rate pipelining. You can use clock-rate pipelining with these optimizations:

- Input and output pipelining
- Multi-cycle block implementations, such as complex math operations like Sqrt and Reciprocal.
- Floating-point library mapping
- Delay balancing
- Resource sharing and streaming

In addition, for designs with multiple hierarchies, to improve opportunities for clock-rate pipelining, it is recommended that you have the HDL block property **FlattenHierarchy** enabled on the top-level Subsystem.

To learn more about clock-rate pipelining and blocks that act as barriers to this optimization, see “Clock-Rate Pipelining” on page 24-114.

Recommended Distributed Pipelining Settings

Guideline ID

3.2.2

Severity

Recommended

Description

Distributed pipelining is a speed optimization that reduces the critical path by moving existing delays in your design while preserving the functional behavior.

To use this optimization for a Subsystem, set the **DistributedPipelining** HDL block property set to on.

To more effectively use this optimization, in the Configuration Parameters dialog box, on the **HDL Code Generation > Optimization** pane, you can specify these additional settings.

- “ConstrainedOutputPipeline” on page 22-8: Make sure that the total number of delays that are inserted including anyinput and output pipelining that you specify is greater than or equal to the value that you specify for **ConstrainedOutputPipeline** on the Subsystem.
- “Hierarchical distributed pipelining” on page 15-9: Select this option if you want to apply the distributed pipelining optimization across multiple subsystem hierarchy. Make sure that the top-level Subsystem and each subsystem in the hierarchy has the **DistributedPipelining** HDL block property set to on.

Note If you cannot enable **DistributedPipelining** on the top-level Subsystem, you can enable **FlattenHierarchy**, which enables pipelining with other blocks at a lower model hierarchy.

- “Clock-rate pipelining” on page 15-11: Select this option if you want the code generator to insert registers at the clock rate instead of the data rate.
- “Allow clock-rate pipelining of DUT output ports” on page 15-12: Select this option if you want the code generator to insert registers at the clock rate instead of the data rate at the DUT output ports.
- “Preserve design delays” on page 15-14: Select this option if you do not want the code generator to move the delays you added to your design. The optimization only moves pipeline registers.
- “Distributed pipelining priority” on page 15-10: Specify whether you want the priority to be **Numerical Integrity** or **Performance**. If you use **Performance**, make sure that the simulation results match. In some cases, this setting moves registers into blocks that have initial values such as constants, which can affect simulation results.

The Subsystem for which you want to apply the optimization must meet these requirements:

- Make sure that the Subsystem that you apply this optimization on does not contain any feedback loops.
- Use blocks that are supported for distributed pipelining. For a list of unsupported blocks, see “Limitations of Distributed Pipelining” on page 24-103. As a workaround:
 - Place some of the unsupported blocks such as Dot Product inside another Subsystem that does not have distributed pipelining enabled.
 - Change the **Distributed pipelining priority** to **Performance** for certain blocks such as Enabled Subsystem.
- The **Sample Time** of the blocks must be discrete. If you have blocks with **Sample Time** set to **Inf**, change them to -1. To identify and change the sample time programmatically, see “Change Block Parameters by Using `find_system` and `set_param`” on page 21-33.
- Remove any input ports on Scope blocks to avoid generation of infinite sample time.

See Also

Simulink Configuration Parameters

“Pipelining Parameters” on page 15-9

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 24-108
- “Distributed Pipelining for Clock Speed Optimization” on page 8-16

More About

- “Distributed Pipelining” on page 24-101

Insert Distributed Pipeline Registers for Blocks with Vector Data Type Inputs

Distributed pipelining is a speed optimization that reduces the critical path by moving existing delays in your design while preserving the functional behavior. This guidelines illustrates how you can use the optimization effectively for vector inputs.

Each guideline has a severity level that indicates the level of compliance requirements. To learn more, see "HDL Modeling Guidelines Severity Levels" on page 21-2.

Guideline ID

3.2.3

Severity

Informative

Description

Blocks that Participate in Distributed Pipelining with Vector Types

By specifying certain settings, you can apply the distributed pipelining optimization to insert pipeline registers for these blocks when you input vectors that are larger than 3 in size. For details, see the "HDL Code Generation" section of each block page.

- Adders: Add, Subtract, and Sum of Elements
- Multipliers: Gain, Product, and Product of Elements
- MinMax
- Dot Product

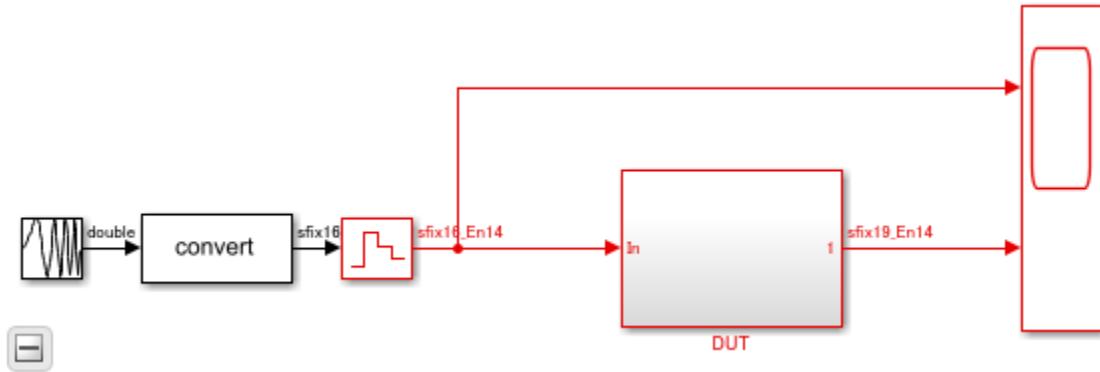
Block Settings and Requirements

- 1 In the HDL Block Properties for the blocks, set **Architecture** to:
 - Tree for adders, multipliers, and MinMax blocks. Distributed pipeline register insertion does not support **Linear** and **Cascade** architectures.
 - **Linear** for Dot Product. Distributed pipeline register insertion does not support Tree architecture for this block.
- 2 Specify the number of pipeline stages by using the **InputPipeline** and **OutputPipeline** properties in the HDL Block Properties dialog box, or by manually inserting Delay blocks.
- 3 Enable **DistributedPipelining** on the Subsystem for which you want to apply this optimization.
- 4 Open the Distributed Pipelining report.
- 5 Open and examine the generated model.

Distributed Pipelining Example for Vector Sum of Elements

This example shows how you can distribute pipeline registers at the output of a Sum of Elements block that accepts vector inputs.

```
open_system('hdlcoder_distributed_pipelining_soe')
set_param('hdlcoder_distributed_pipelining_soe','SimulationCommand','Update')
```



If you navigate the model, you see three pipeline stages for the Sum of Elements block.

```
open_system('hdlcoder_distributed_pipelining_soe/DUT/Add')
```



You see a Delay block of three added at the output of the Sum of Elements block. You can use distributed pipelining to distribute the delays.

1. Set **Architecture** to Tree for the Sum of Elements block.

```
hdlset_param('hdlcoder_distributed_pipelining_soe/DUT/Add/Add', ...
    'Architecture','Tree')
```

2. Enable **DistributedPipelining** on the Add Subsystem

```
hdlset_param('hdlcoder_distributed_pipelining_soe/DUT/Add', ...
    'DistributedPipelining','On')
```

3. Generate HDL code for the DUT Subsystem.

```
makehdl('hdlcoder_distributed_pipelining_soe/DUT')
```

4. Open the Code Generation Report to see the effect of the distributed pipelining optimization.

Detailed Report

Subsystem: [Add](#)

Implementation Parameters: *DistributedPipelining:* 'on'; *InputPipeline:* 0; *OutputPipeline:* 0

Status: Distributed Pipelining successful.

Before Distributed Pipelining : 3 registers (57 flip-flops)

Registers	Count
19-bit	3

After Distributed Pipelining : 7 registers (123 flip-flops)

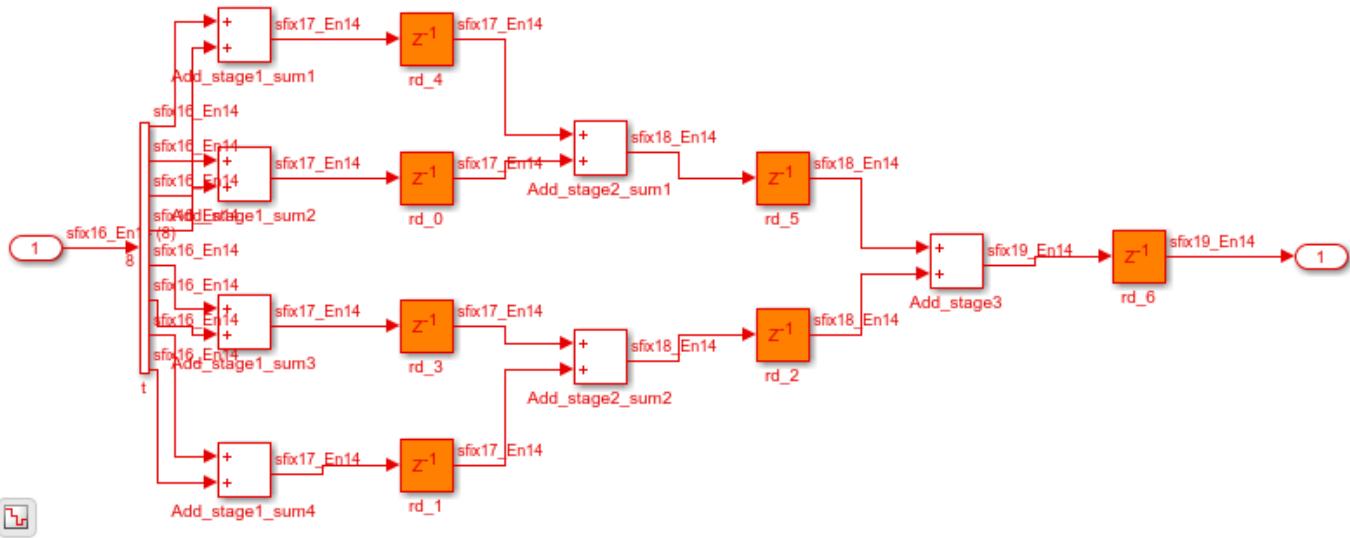
Registers	Count
17-bit	4
18-bit	2
19-bit	1

Generated Model

Generated model after the transformation: [gm_hdlcoder_distributed_pipelining_soe](#)

5. To see the effect of the transformation and how the pipeline registers are distributed, open the generated model and navigate to the Add Subsystem.

```
load_system('gm_hdlcoder_distributed_pipelining_soe')
set_param('gm_hdlcoder_distributed_pipelining_soe','SimulationCommand','Update')
open_system('gm_hdlcoder_distributed_pipelining_soe/DUT/Add')
```



See Also

Simulink Configuration Parameters
Pipelining Parameters

Related Examples

- “Distributed Pipelining: Speed Optimization” on page 24-108
- “Distributed Pipelining for Clock Speed Optimization” on page 8-16

More About

- “Distributed Pipelining” on page 24-101

Supported Blocks Library and Block Properties

- “View HDL-Supported Blocks and HDL-Specific Block Documentation” on page 22-2
- “HDL Block Properties: General” on page 22-3
- “HDL Block Properties: Native Floating Point” on page 22-29
- “HDL Filter Block Properties” on page 22-39
- “HDL Filter Architectures” on page 22-45
- “Distributed Arithmetic for HDL Filters” on page 22-50
- “Set and View HDL Model and Block Parameters” on page 22-52
- “Pass through, No HDL, and Cascade Implementations” on page 22-56
- “Build a ROM Block with Simulink Blocks” on page 22-57
- “Getting Started with RAM and ROM in Simulink®” on page 22-58
- “Wireless Communications Design for FPGAs and ASICs” on page 22-61

View HDL-Supported Blocks and HDL-Specific Block Documentation

View HDL-Supported Blocks and Documentation

You can generate efficient HDL code for a number of blocks in Simulink and other product libraries. To see the product libraries that support HDL code generation use the `hdllib` function. This function filters the library browser to show blocks that are supported for HDL code generation. To learn more, see “Show Blocks Supported for HDL Code Generation” on page 25-18.

This table shows blocks in various product libraries supported for HDL code generation. To view usage notes and limitations, in the corresponding reference page, scroll down to the **Extended Capabilities** section at the bottom and expand the **HDL Code Generation** section.

HDL code generation support for the blocks is summarized in the following tables.

- Blocks Supported for HDL Code Generation (Category List)
- Blocks Supported for HDL Code Generation (Alphabetical List)

View HDL-Specific Block Documentation

The HDL Block Properties dialog box contains HDL-specific properties for each block and subsystem in your model. On this dialog box, you can click the **Help** button to navigate to the documentation for that block. See also “HDL Block Properties: General” on page 22-3 and “HDL Block Properties: Native Floating Point” on page 22-29.

To view HDL-specific block documentation, either:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the block for which you want to see the help documentation and then select **HDL Block Properties**. To view the block documentation, click **Help**.
- Right-click the block and select **HDL Code > HDL Block Properties**. To view the block documentation, click **Help**.

You see the documentation in the **Extended Capabilities > HDL Code Generation** section of the block page in the product that owns the block.

See Also

`hdllib`

Related Examples

- “Set and View HDL Model and Block Parameters” on page 22-52
- “Show Blocks Supported for HDL Code Generation” on page 25-18

More About

- “HDL Block Properties: General” on page 22-3
- “HDL Filter Block Properties” on page 22-39

HDL Block Properties: General

In this section...

[“Overview” on page 22-3](#)
[“AdaptivePipelining” on page 22-4](#)
[“BalanceDelays” on page 22-5](#)
[“ClockRatePipelining” on page 22-5](#)
[“CodingStyle” on page 22-6](#)
[“ConstMultiplierOptimization” on page 22-7](#)
[“ConstrainedOutputPipeline” on page 22-8](#)
[“DistributedPipelining” on page 22-8](#)
[“DotProductStrategy” on page 22-9](#)
[“DSPStyle” on page 22-10](#)
[“FlattenHierarchy” on page 22-12](#)
[“InputPipeline” on page 22-13](#)
[“InstantiateFunctions” on page 22-13](#)
[“InstantiateStages” on page 22-14](#)
[“LoopOptimization” on page 22-14](#)
[“LUTRegisterResetType” on page 22-15](#)
[“MapPersistentVarsToRAM” on page 22-16](#)
[“OutputPipeline” on page 22-17](#)
[“RAMDirective” on page 22-18](#)
[“ResetType” on page 22-21](#)
[“SerialPartition” on page 22-22](#)
[“SharingFactor” on page 22-23](#)
[“SoftReset” on page 22-23](#)
[“StreamingFactor” on page 22-24](#)
[“UsePipelines” on page 22-24](#)
[“UseRAM” on page 22-25](#)
[“VariablesToPipeline” on page 22-28](#)

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Model and Block Parameters” on page 22-52 to learn how to select block implementations and parameters in the GUI or the command line.

Property names are specified as character vectors. The data type of a property value is specific to the property. This section describes the syntax of each block implementation parameter and how the parameter affects generated code.

HDL Block Properties of Library Blocks

HDL block properties of library blocks are treated similar to mask parameters. When you instantiate library blocks in your model, the current HDL block properties of that library block are copied to instances of that block in your model. The HDL block properties of these instances are not synchronized with the HDL block properties of the library block. That is, if you change the HDL block property of the library block, the change does not get propagated to instances of the library block that you already added to your Simulink model. If you want the HDL block properties of a library block to be synchronized with its instances in the model, create a Subsystem and then place this block inside that Subsystem. The HDL block properties of blocks that reside inside the library block are synchronized with the corresponding instances in your model.

Suppose a library contains a Subsystem block with HDL architecture set to **Module**. When you instantiate this block in your model, the block instance uses **Module** as the HDL architecture. If you change the HDL architecture of the Subsystem block in the library to **BlackBox**, existing instances of that Subsystem block in your model still use **Module** as the HDL architecture. If you now add instances of the Subsystem block from the library in your model, the new block instances get a copy of the current HDL block properties, and therefore use **BlackBox** as the HDL architecture. If you want the HDL architecture of the Subsystem block in the library to be synchronized with its instances in the model, create a wrapper subsystem with the HDL architecture that you want inside this Subsystem.

Adaptive Pipelining

The **AdaptivePipelining** subsystem parameter enables you to set adaptive pipelining on a subsystem within a model.

Adaptive Pipelining Setting	Description
'inherit' (default)	Use the adaptive pipelining setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the adaptive pipelining setting for the model.
'on'	Insert adaptive pipelines for this subsystem.
'off'	Do not insert adaptive pipelines for this subsystem, even if the parent subsystem has adaptive pipelining enabled.

To disable adaptive pipelining for a subsystem within a model, set the adaptive pipelining parameter, **AdaptivePipelining**, to 'off' for that subsystem.

To learn how to set model-level adaptive pipelining, see "Adaptive pipelining" on page 15-13.

Set Adaptive Pipelining For a Subsystem

To set adaptive pipelining for a subsystem from the HDL Block Properties dialog box:

- 1 Right-click the subsystem and select **HDL Code > HDL Block Properties**.
- 2 For **AdaptivePipelining**, select **inherit**, **on**, or **off**.

To set adaptive pipelining for a subsystem from the command line, use `hdlset_param`. For example, to turn off adaptive pipelining for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'AdaptivePipelining', 'off')
```

See also `hdlset_param`.

BalanceDelays

The `BalanceDelays` subsystem parameter enables you to set delay balancing on a subsystem within a model.

BalanceDelays Setting	Description
'inherit' (default)	Use the delay balancing setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the delay balancing setting for the model.
'on'	Balance delays for this subsystem.
'off'	Do not balance delays for this subsystem, even if the parent subsystem has delay balancing enabled.

To disable delay balancing for any subsystem within a model, you must set the model-level delay balancing parameter, `BalanceDelays`, to '`off`'. When delay balancing is enabled on the model, the delay balancing setting on individual subsystems is ignored.

To learn how to set model-level delay balancing, see "Balance delays" on page 15-3.

Set Delay Balancing For a Subsystem

To set delay balancing for a subsystem using the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **BalanceDelays**, select **inherit**, **on**, or **off**.

To set delay balancing for a subsystem from the command line, use `hdlset_param`. For example, to turn off delay balancing for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'BalanceDelays', 'off')
```

See also `hdlset_param`.

ClockRatePipelining

The `ClockRatePipelining` subsystem parameter enables you to set clock-rate pipelining on a subsystem within a model.

Clock-Rate Pipelining Setting	Description
'inherit' (default)	Use the clock-rate pipelining setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the clock-rate pipelining setting for the model.

Clock-Rate Pipelining Setting	Description
'on'	Insert clock-rate pipelines for this subsystem.
'off'	Do not insert clock-rate pipelines for this subsystem, even if the parent subsystem has clock-rate pipelining enabled.

To disable clock-rate pipelining for a subsystem within a model, set the clock-rate pipelining parameter, **ClockRatePipelining**, to 'off' for that subsystem.

To learn how to set model-level clock-rate pipelining, see "Clock-rate pipelining" on page 15-11.

Set Clock-Rate Pipelining For a Subsystem

To set clock-rate pipelining for a subsystem using the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **ClockRatePipelining**, select **inherit**, **on**, or **off**.

To set clock-rate pipelining for a subsystem from the command line, use `hdlset_param`. For example, to turn off clock-rate pipelining for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'ClockRatePipelining', 'off')
```

See also `hdlset_param`.

CodingStyle

When you use Multiport Switch blocks, use the **CodingStyle** parameter to specify whether you want to generate HDL code with if-else or case statements. By default, HDL Coder generates if-else statements. If you have several Multiport Switch blocks in your model, you can choose to specify a different **CodingStyle** for each block.

CodingStyle Setting	Description
'ifelse_stmt'(Default)	Generate if-else statements in the Verilog code or when-else statements in the VHDL code for a Multiport Switch block.
'case_stmt'	Generate case statements in the Verilog code or case-when statements in the VHDL code for a Multiport Switch block.

Set CodingStyle For Multiport Switch Block

To set CodingStyle for a Multiport Switch using the HDL Block Properties dialog box:

- 1 Right-click the Multiport Switch block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **CodingStyle**, select **ifelse_stmt** or **case_stmt**.

To see the **CodingStyle** specified for a subsystem from the command line, use `hdlget_param`. For example, to see the settings specified for a Multiport Switch block inside a subsystem, `my_dut`:

```
hdlget_param('my_dut/Multiport Switch', 'CodingStyle')
```

ans =

```
'case_stmt'
```

See also `hdlset_param`.

ConstMultiplierOptimization

The `ConstMultiplierOptimization` implementation parameter lets you specify use of canonical signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in the generated code.

The following table shows the `ConstMultiplierOptimization` parameter values.

ConstMultiplierOptimization Setting	Description
'none' (Default)	By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.
'CSD'	When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.
'FCSD'	This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction. This option lets you achieve a greater area reduction than CSD, at the cost of decreasing clock speed.
'auto'	When you specify this option, HDL Coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. When you specify 'auto', the coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).

The `ConstMultiplierOptimization` parameter is available for the following blocks:

- Gain
- Stateflow chart
- Truth Table

- MATLAB Function
- MATLAB System

ConstrainedOutputPipeline

Use the **ConstrainedOutputPipeline** parameter to specify a nonnegative number of registers to place at the block outputs.

HDL Coder moves existing delays within your design to try to meet your constraint. New registers are not added. If there are fewer registers than the coder needs to satisfy your constraint, the coder reports the difference between the number of desired and actual output registers. You can add delays to your design using input or output pipelining.

Distributed pipelining does not redistribute registers you specify with constrained output pipelining.

How to Specify Constrained Output Pipelining

To specify constrained output pipelining for a block using the GUI:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
- 2 For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.

To specify constrained output pipelining, at the command line, enter:

```
hdlset_param(path_to_block,  
            'ConstrainedOutputPipeline', number_of_output_registers)
```

For example, to constrain 6 registers at the output ports of a subsystem, `subsys`, in your model, `mymodel`, enter:

```
hdlset_param('mymodel/subsys','ConstrainedOutputPipeline', 6)
```

See Also

- “Constrained Output Pipelining” on page 24-112

DistributedPipelining

The **DistributedPipelining** parameter enables pipeline register distribution, a speed optimization that enables you to increase your clock speed by reducing your critical path.

The following table shows the effect of the **DistributedPipelining** and **OutputPipeline** parameters.

DistributedPipelining	OutputPipeline, nStages	Result
'off' (default)	Unspecified (<i>nStages</i> defaults to 0)	HDL Coder does not insert pipeline registers.
	<i>nStages</i> > 0	The coder inserts <i>nStages</i> output registers at the output of the subsystem, MATLAB Function block, or Stateflow chart.

DistributedPipelining	OutputPipeline, nStages	Result
'on'	Unspecified (<i>nStages</i> defaults to 0)	The coder does not insert pipeline registers. DistributedPipelining has no effect.
	<i>nStages</i> > 0	The coder distributes <i>nStages</i> registers inside the subsystem, MATLAB Function block, or Stateflow chart, based on critical path analysis.

To achieve further optimization of code generated with distributed pipelining, perform retiming during RTL synthesis, if possible.

Tip Output data might be in an invalid state initially if you insert pipeline registers. To avoid test bench errors resulting from initial invalid samples, disable output checking for those samples. For more information, see “Ignore output data checking (number of samples)” on page 19-19.

See Also

- “Distributed Pipelining” on page 24-101
- “Specify Distributed Pipelining” on page 24-103
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37

DotProductStrategy

If you use the Product block for matrix multiplication in your design, use the **DotProductStrategy** to specify how you want to implement the matrix multiplication.

The **DotProductStrategy** options are listed in the following table.

DotProductStrategy Value	Description
'Fully Parallel' (default)	<p>Expands the matrix multiplication operation into multipliers and adders. For example, if you multiply two 2x2 matrices, the implementation uses eight multipliers and four adders to compute the result.</p> <p>Note The DotProductStrategy must be set to 'Fully Parallel' when you use the Native Floating Point mode.</p>

DotProductStrategy Value	Description
'Serial Multiply-Accumulate'	Uses the Serial architecture of the Multiply-Accumulate block to implement the matrix multiplication. In this architecture, the clock rate must be faster than the clock rate that you specify with Parallel architecture. You can see the clock rate in the Clock Summary information of the Code Generation report.
'Parallel Multiply-Accumulate'	Uses the Parallel architecture of the Multiply-Accumulate block to implement the matrix multiplication.

DSPStyle

DSPStyle enables you to generate code that includes synthesis attributes for multiplier mapping in your design. You can choose whether to map a particular block's multipliers to DSPs or logic in hardware.

For Xilinx targets, the generated code uses the `use_dsp` attribute. For Altera targets, the generated code uses the `multstyle` attribute.

The **DSPStyle** options are listed in the following table.

DSPStyle Value	Description
'none' (default)	Do not insert a DSP mapping synthesis attribute.
'on'	Insert synthesis attribute that directs the synthesis tool to map to DSPs in hardware.
'off'	Insert synthesis attribute that directs the synthesis tool to map to logic in hardware.

The **DSPStyle** parameter is available for the following blocks:

- Gain
- Product
- Product of Elements with Architecture set to Tree
- Subsystem
- Atomic Subsystem
- Variant Subsystem
- Enabled Subsystem
- Triggered Subsystem
- Model with Architecture set to **ModelReference**

Hierarchy Flattening Behavior

If you specify hierarchy flattening for a subsystem that also has a nondefault **DSPStyle** setting, HDL Coder propagates the **DSPStyle** setting to the parent subsystem.

If the flattened subsystem contains Gain, Product, or Product of Elements blocks, the coder keeps their nondefault **DSPStyle** settings, and replaces default **DSPStyle** settings with the flattened subsystem **DSPStyle** setting.

Synthesis Attributes in Generated Code

The generated code for synthesis attributes depends on:

- Target language
- **DSPStyle** value
- **SynthesisTool** value

The following table shows examples of synthesis attributes in generated code.

DSPStyle Value	TargetLanguage Value	SynthesisTool Value	
		'Altera Quartus II'	'Xilinx ISE' 'Xilinx Vivado'
'none'	'Verilog'	wire signed [32:0] m4_out1;	wire signed [32:0] m4_out1;
	'VHDL'	m4_out1 : signal;	m4_out1 : signal;
'on'	'Verilog'	(* multstyle = "dsp" *) wire signed [32:0] m4_out1;	(* use_dsp = "yes" *) wire signed [32:0] m4_out1;
	'VHDL'	attribute multstyle : string ; attribute multstyle of m4_out1 : signal is "dsp" ;	attribute use_dsp : string ; attribute use_dsp of m4_out1 : signal is "yes" ;
'off'	'Verilog'	(* multstyle = "logic" *) wire signed [32:0] m4_out1;	(* use_dsp = "no" *) wire signed [32:0] m4_out1;
	'VHDL'	attribute multstyle : string ; attribute multstyle of m4_out1 : signal is "logic" ;	attribute use_dsp : string ; attribute use_dsp of m4_out1 : signal is "no" ;

Requirement For Synthesis Attribute Specification

You must specify a synthesis tool by using the **SynthesisTool** property.

How To Specify a Synthesis Attribute

To specify a synthesis attribute using the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **DSPStyle**, select **on**, **off**, or **none**.

To specify a synthesis attribute from the command line, use `hdlset_param`. For example, suppose you have a model, `my_model`, with a DUT subsystem, `my_dut`, that contains a . Gain block, `my_multiplier`. To insert a synthesis attribute to map `my_multiplier` to a DSP, enter:

```
hdlset_param('my_model/my_dut/my_multiplier', 'DSPStyle', 'on')
```

See also `hdlset_param`.

Limitations For Synthesis Attribute Specification

- When you specify a nondefault `DSPStyle` block property, the `ConstMultiplierOptimization` property must be set to '`none`'.
- Inputs to multiplier components cannot use the `double` data type.
- Gain constant cannot be a power of 2.

FlattenHierarchy

`FlattenHierarchy` enables you to remove subsystem hierarchy from the HDL code generated from your design.

FlattenHierarchy Setting	Description
'inherit' (default)	Use the hierarchy flattening setting of the parent subsystem. If this subsystem is the highest-level subsystem, do not flatten.
'on'	Flatten this subsystem.
'off'	Do not flatten this subsystem, even if the parent subsystem is flattened.

To flatten hierarchy, you must also have the `MaskParameterAsGeneric` global property set to '`off`'. For more information, see "Generate parameterized HDL code from masked subsystem" on page 17-57.

How To Flatten Hierarchy

To set hierarchy flattening using the HDL Block Properties dialog box:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.

To set hierarchy flattening from the command line, use `hdlset_param`. For example, to turn on hierarchy flattening for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'FlattenHierarchy', 'on')
```

See also `hdlset_param`.

Limitations For Hierarchy Flattening

A subsystem cannot be flattened if the subsystem is:

- A Synchronous Subsystem or uses the State Control block in Synchronous mode.
- A model reference implementation.
- A Triggered Subsystem when “Use trigger signal as clock” on page 17-41 is enabled.
- A masked subsystem that contains any of the following:
 - Bus.
 - Enumerated data type.
 - Lookup table blocks: 1-D Lookup Table, 2-D Lookup Table, Cosine HDL Optimized, Direct LookupTable (n-D), Prelookup, Sine HDL Optimized, n-D Lookup Table.
 - MATLAB System block.
 - Stateflow blocks: Chart, State Transition Table, Sequence Viewer.
 - Blocks with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 22-56.

Note This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

InputPipeline

InputPipeline lets you specify a implementation with input pipelining for selected blocks. The parameter value specifies the number of input pipeline stages (pipeline depth) in the generated code.

The following code specifies an input pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii}),'InputPipeline', 2), end;
```

Note The **InputPipeline** setting does not have any effect on blocks that do not have an input port.

When generating code for pipeline registers, HDL Coder appends a postfix string to names of input or output pipeline registers. The default postfix string is `_pipe`. To customize the postfix string, use the **Pipeline postfix** option in the **Global Settings / General** pane in the **HDL Code Generation** pane of the Configuration Parameters dialog box. Alternatively, you can pass the desired postfix as a character vector in the `makehdl` property `PipelinePostfix`. For an example, see “Pipeline postfix” on page 17-23.

InstantiateFunctions

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL entity or Verilog module for each function. HDL Coder generates code for each entity or module in a separate file.

The **InstantiateFunctions** options for the MATLAB Function block are listed in the following table.

InstantiateFunctions Setting	Description
'off' (default)	Generate code for functions inline.

InstantiateFunctions Setting	Description
'on'	Generate a VHDL entity or Verilog module for each function, and save each module or entity in a separate file.

How To Generate Instantiable Code for Functions

To set the **InstantiateFunctions** parameter using the HDL Block Properties dialog box:

- 1 Right-click the MATLAB Function block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **InstantiateFunctions**, select **on**.

To set the **InstantiateFunctions** parameter from the command line, use `hdlset_param`. For example, to generate instantiable code for functions in a MATLAB Function block, `myMatlabFcn`, in your DUT subsystem, `myDUT`, enter:

```
hdlset_param('my_DUT/my_MATLABFcnBlk', 'InstantiateFunctions', 'on')
```

Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or `for` loops.
- Any function is called with a nonconstant `struct` input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

InstantiateStages

For a Cascade architecture, you can use the **InstantiateStages** parameter to generate a VHDL entity or Verilog module for each computation stage. HDL Coder generates code for each entity or module in a separate file.

InstantiateStages Setting	Description
'off' (default)	Generate cascade stages in a single VHDL entity or Verilog module.
'on'	Generate a VHDL entity or Verilog module for each cascade stage, and save each module or entity in a separate file.

LoopOptimization

`LoopOptimization` enables you to stream or unroll loops in code generated from a MATLAB Function block. Loop streaming optimizes for area; loop unrolling optimizes for speed.

Note If you specify the MATLAB Datapath architecture of the MATLAB Function block, you can only unroll loops. To stream loops, you can use the streaming optimization by specifying a **StreamingFactor**. See “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-146.

LoopOptimization Setting	Description
'none' (default)	Do not optimize loops.
'Unrolling'	Unroll loops.
'Streaming'	Stream loops.

How to Optimize MATLAB Function Block For Loops

To select a loop optimization using the HDL Block Properties dialog box:

- 1 Right-click the MATLAB Function block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **LoopOptimization**, select none, Unrolling, or Streaming.

To select a loop optimization from the command line, use `hdlset_param`. For example, to turn on loop streaming for a MATLAB Function block, `my_mlfn`:

```
hdlset_param('my_mlfn', 'LoopOptimization', 'Streaming')
```

See also `hdlset_param`.

Limitations for MATLAB Function Block Loop Optimization

HDL Coder cannot stream a loop if:

- The loop index counts down. The loop index must increase by 1 on each iteration.
- There are 2 or more nested loops at the same level of hierarchy within another loop.
- Any particular persistent variable is updated both inside and outside a loop.

HDL Coder can stream a loop when the persistent variable is:

- Updated inside the loop and read outside the loop.
- Read within the loop and updated outside the loop.

LUTRegisterResetType

Use the LUTRegisterResetType block parameter to control synthesis of a LUT into a ROM structure on an FPGA.

LUTRegisterResetType Value	Description
default	LUT output register has default reset logic. When you generate HDL, the LUT will be synthesized as registers.

LUTRegisterResetType Value	Description
none	LUT output register has no reset logic. When you generate HDL, the LUT will be synthesized as a ROM.

You can specify `LUTRegisterResetType` for the following blocks:

- Gamma Correction
- Lookup Table

The NCO HDL Optimized block ignores this parameter.

MapPersistentVarsToRAM

With the `MapPersistentVarsToRAM` implementation parameter, you can use RAM-based mapping for persistent arrays of a MATLAB Function block instead of mapping to registers.

MapPersistentVarsToRAM Setting	Mapping Behavior
off	Persistent arrays map to registers in the generated HDL code.
on	Persistent array variables map to RAM. For restrictions, see "RAM Mapping Restrictions" on page 22-16.

RAM Mapping Restrictions

When you enable RAM mapping, a persistent array or user-defined System object private property maps to a block RAM when all of the following conditions are true:

- Each read or write access is for a single element only. For example, submatrix access and array copies are not allowed.
- Address computation logic is not read-dependent. For example, computation of a read or write address using the data read from the array is not allowed.
- Persistent variables or user-defined System object private properties are initialized to 0 if they have a cyclic dependency. For example, if you have two persistent variables, A and B, you have a cyclic dependency if A depends on B, and B depends on A.
- If an access is within a conditional statement, the conditional statement uses only simple logic expressions (`&&`, `||`, `~`) or relational operators. For example, in the following code, `r1` does not map to RAM:

```
if (mod(i,2) > 0)
    a = r1(u);
else
    r1(i) = u;
end
```

Rewrite complex conditions, such as conditions that call functions, by assigning them to temporary variables, and using the temporary variables in the conditional statement. For example, to map `r1` to RAM, rewrite the previous code as follows:

```
temp = mod(i,2);
if (temp > 0)
```

```

    a = r1(u);
else
    r1(i) = u;
end

```

- The persistent array or user-defined System object private property value depends on external inputs.

For example, in the following code, `bigarray` does not map to RAM because it does not depend on `u`:

```

function z = foo(u)

persistent cnt bigarray
if isempty(cnt)
    cnt = fi(0,1,16,10,hdlfimath);
    bigarray = uint8(zeros(1024,1));
end
z = u + cnt;
idx = uint8(cnt);
temp = bigarray(idx+1);
cnt(:) = cnt + fi(1,1,16,0,hdlfimath) + temp;
bigarray(idx+1) = idx;

```

- `RAMSize` is greater than or equal to the `RAMMappingThreshold` value. `RAMSize` is the product `NumElements * WordLength * Complexity`.
 - `NumElements` is the number of elements in the array.
 - `WordLength` is the number of bits that represent the data type of the array.
 - `Complexity` is 2 for arrays with a complex base type; 1 otherwise.

If any of the above conditions is false, the persistent array or user-defined System object private property maps to a register in the HDL code.

RAMMappingThreshold

The default value of `RAMMappingThreshold` is 256. To change the threshold, use `hdlset_param`. For example, the following command changes the mapping threshold for the `sfir_fixed` model to 128 bits:

```
hdlset_param('sfir_fixed', 'RAMMappingThreshold', 128);
```

You can also change the RAM mapping threshold in the Configuration Parameters dialog box. For more information, see **RAM mapping threshold (bits)** section in “RAM Mapping Parameters” on page 15-7.

Example

For an example that shows how to map persistent array variables to RAM in a MATLAB Function block, see “RAM Mapping With the MATLAB Function Block” on page 24-96.

OutputPipeline

`OutputPipeline` lets you specify a implementation with output pipelining for selected blocks. The parameter value specifies the number of output pipeline stages (pipeline depth) in the generated code.

The following code specifies an output pipeline depth of two stages for each Sum block in the model:

```
sblocks = find_system(gcb, 'BlockType', 'Sum');
for ii=1:length(sblocks),hdlset_param(sblocks{ii}),'OutputPipeline', 2), end;
```

Note The **OutputPipeline** setting does not have any effect on blocks that do not have an output port.

When generating code for pipeline registers, HDL Coder appends a postfix string to names of input or output pipeline registers. The default postfix string is `_pipe`. To customize the postfix string, use the **Pipeline postfix** option in the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > General** tab. Alternatively, you can use the **PipelinePostfix** property with `makehdl`. For an example, see “Pipeline postfix” on page 17-23.

See also “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37.

RAMDirective

RAMDirective lets you specify whether you want to map the RAM blocks in your Simulink model to distributed RAMs, block RAMs, or UltraRAM memory. When you select a value for this setting, HDL Coder generates a `ramstyle` attribute in the HDL code. This attribute specifies the type of RAM memory unit that you want the synthesis tool to use when inferring the RAM blocks in your design.

RAMDirective Value	Description
<code>none</code> (default)	Do not generate the <code>ramstyle</code> attribute in the HDL code. The synthesis tool determines the type of inferred RAM for mapping the RAM blocks in your model.
<code>distributed</code>	<p>Generate HDL attribute for mapping the RAM blocks in your model to distributed RAMs. Distributed RAMs are constructed with LUT. These RAMs are faster but occupy a larger number of LUT slices on the FPGA.</p> <p>This VHDL code shows the <code>ramstyle</code> attribute set to <code>distributed</code>:</p> <pre>attribute ram_style: string; attribute ram_style of ram : signal is "distributed";</pre> <p>This Verilog code shows the <code>ramstyle</code> attribute set to <code>distributed</code>:</p> <pre>(* ram_style = "distributed" *)</pre>

RAMDirective Value	Description
block	<p>Generate HDL attribute for mapping the RAM blocks in your model to block RAMs. A block RAM is a dedicated memory unit on the FPGA device. The number of block RAMs available depends on the FPGA device that you are deploying the HDL code to. Sizes of block RAMs can be 4kb, 8kb, 16kb, and 32kb.</p> <p>To map your RAM blocks to block RAM:</p> <ul style="list-style-type: none"> Specify the synthesis tool. You must target a Xilinx device that contains block RAM resources. <p>Note If the target device does not contain block RAMs, the synthesis tool ignores this attribute and might infer the RAM as distributed RAMs or LUT slices.</p> <ul style="list-style-type: none"> Enter a target frequency greater than zero. <p>This VHDL code shows the <code>ramstyle</code> attribute set to <code>block</code>:</p> <pre>attribute ram_style: string; attribute ram_style of ram : signal is "block";</pre> <p>This Verilog code shows the <code>ramstyle</code> attribute set to <code>block</code>:</p> <pre>(* ram_style = "block" *)</pre>

RAMDirective Value	Description
ultra	<p>Generate HDL attribute for mapping the RAM blocks in your model to UltraRAM memory. An UltraRAM is a dedicated memory block on the target FPGA. The number of UltraRAM memory units available depends on the FPGA device that you are deploying the HDL code to. UltraRAM units are larger than block RAMs and can be as large as 500Mb in size.</p> <p>To map your RAM blocks to UltraRAM:</p> <ul style="list-style-type: none"> Specify Xilinx Vivado as the synthesis tool. You must target a Xilinx device that contains UltraRAM resources, such as Virtex® UltraScale+™. <p>Note If the target device does not contain UltraRAM memory, the synthesis tool ignores this attribute and might infer the RAM as distributed RAMs or LUT slices. To map the RAM blocks to block RAMs instead, set RAMDirective to block.</p> <ul style="list-style-type: none"> Enter a target frequency greater than zero. The RAM blocks in your design must follow a fixed-read behavior and have a single clock interface. In the HDL RAMs library, except for Dual Port RAM and Dual Rate Dual Port RAM blocks, you can map all other RAM blocks to UltraRAM. The RAM blocks must not have an initial value specified. When you use the RAM System blocks such as Single Port RAM System, Specify the RAM initial value must be set to 0. On device reset, all memory locations in the UltraRAM are initialized to zero. <p>This VHDL code shows the <code>ramstyle</code> attribute set to <code>ultra</code>:</p> <pre>attribute ram_style: string; attribute ram_style of ram : signal is "ultra";</pre> <p>This Verilog code shows the <code>ramstyle</code> attribute set to <code>ultra</code>:</p> <pre>(* ram_style = "ultra" *)</pre>

Set RAMDirective for RAM Blocks

In the **HDL RAMs** library, except for the Dual Rate Dual Port RAM, you can specify the **RAMDirective** property for all other RAM blocks.

To set **RAMDirective** for a RAM block from the HDL Block Properties dialog box:

- 1 Right-click the RAM block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **RAMDirective**, select **none**, **distributed**, **block**, or **ultra**.

Note For the Dual Port RAM block, you cannot specify `ultra` as the **RAMDirective** because the block does not have a fixed read behavior.

To set RAMDirective for a block from the command line, use `hdlset_param`. For example, to set RAMDirective to `ultra` for a Single Port RAM block inside a subsystem, `my_dut`:

```
hdlset_param('my_dut/Single Port RAM', 'RAMDirective', 'ultra');
```

See also `hdlset_param`.

ResetType

Use the ResetType block parameter to suppress reset logic generation.

ResetType Value	Description
<code>default</code>	Generate reset logic.
<code>none</code>	<p>Do not generate reset logic.</p> <p>Reset is not applied to generated registers. Therefore, mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded.</p> <p>To avoid test bench errors during the initial phase, determine the number of samples required to fully load the registers. Then, set the Ignore output data checking (number of samples) option accordingly. See also Ignore output data checking (number of samples) in “Test Bench Stimulus and Output Parameters” on page 19-18.</p>

You can specify ResetType for the following blocks:

- Chart
- Convolutional Deinterleaver
- Convolutional Interleaver
- Delay
- Delay (DSP System Toolbox)
- General Multiplexed Deinterleaver
- General Multiplexed Interleaver
- MATLAB Function
- MATLAB System
- Memory
- Tapped Delay
- Truth Table
- Unit Delay Enabled
- Unit Delay

Reset Logic for Optimizations in the MATLAB Function Block

When you set **ResetType** to none for a MATLAB Function block, HDL Coder does not generate reset logic for persistent variables in the MATLAB code.

However, if you specify other optimizations for the block, the coder may insert registers that use reset logic. The coder does not suppress reset logic generation for these registers. Therefore, if you set **ResetType** to none along with other block optimizations, your generated code may have a reset port at the top level.

How to Suppress Reset Logic Generation

To suppress reset logic generation for a block using the UI:

- 1 Right-click the block and select **HDL Code > HDL Block Properties**.
- 2 For **ResetType**, select none.

To suppress reset logic generation, on the command line, enter:

```
hdlset_param(path_to_block,'ResetType','none')
```

For example, to suppress reset logic generation for a Unit Delay block, `UnitDelay1`, within a subsystem, `mySubsys`, on the command line, enter:

```
hdlset_param('mySubsys/UnitDelay1','ResetType','none');
```

Specify Synchronous or Asynchronous Reset

To specify a synchronous or asynchronous reset, use the **ResetType** model-level parameter. For details, see **Reset type** in “Reset Settings and Parameters” on page 17-8.

SerialPartition

Use this parameter on Min/Max blocks to specify partitions for a serial cascade architecture. The default setting uses the minimum number of partitions.

To Generate This Architecture...	Set SerialPartition to...
Cascade-serial with explicitly specified partitioning	[p1 p2 p3...pN]: a vector of N integers, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the input data vector. The values of the vector elements must be in descending order, except the last two elements can be equal. For example, for an input of 8 elements, partitions [5 3] or [4 2 2] are legal, but the partitions [2 2 2] or [3 2 3] raise an error at code generation time.
Cascade-serial with automatically optimized partitioning	0

This property is also used for serial filter architectures. For how to configure filter blocks, see “SerialPartition” on page 22-43.

SharingFactor

Use **SharingFactor** to specify the number of functionally equivalent resources to map to a single shared resource. The default is 0. See “Resource Sharing” on page 24-32.

SoftReset

Use the **SoftReset** block parameter to specify whether to generate hardware-friendly synchronous reset logic, or local reset logic that matches the Simulink simulation behavior. This property is available for the Unit Delay Resettable block or Unit Delay Enabled Resettable block.

SoftReset Value	Description
off (default)	Generate local reset logic that matches the Simulink simulation behavior.
on	Generate synchronous reset logic for the block. This option generates code that is more efficient for synthesis, but does not match the Simulink simulation behavior.

When **SoftReset** set to 'off', the following code is generated for a Unit Delay Resettable block :

```
always @(posedge clk or posedge reset)
begin : Unit_Delay_Resettable_process
  if (reset == 1'b1) begin
    Unit_Delay_Resettable_zero_delay <= 1'b1;
    Unit_Delay_Resettable_switch_delay <= 2'b00;
  end
  else begin
    if (enb) begin
      Unit_Delay_Resettable_zero_delay <= 1'b0;
      if (UDR_reset == 1'b1) begin
        Unit_Delay_Resettable_switch_delay <= 2'b00;
      end
      else begin
        Unit_Delay_Resettable_switch_delay <= In1;
      end
    end
  end
end

assign Unit_Delay_Resettable_1 =
  (UDR_reset ||
   Unit_Delay_Resettable_zero_delay ? 1'b1 : 1'b0);
assign out0 = (Unit_Delay_Resettable_1 == 1'b1 ? 2'b00 :
  Unit_Delay_Resettable_switch_delay);
```

When **SoftReset** set to 'on', the following code is generated for a Unit Delay Resettable block :

```
always @(posedge clk or posedge reset)
begin : Unit_Delay_Resettable_process
  if (reset == 1'b1) begin
    Unit_Delay_Resettable_reg <= 2'b00;
  end
  else begin
    if (enb) begin
```

```
if (UDR_reset != 1'b0) begin
    Unit_Delay_Resettable_reg <= 2'b00;
end
else begin
    Unit_Delay_Resettable_reg <= In1;
end
end
end
end

assign out0 = Unit_Delay_Resettable_reg;
```

StreamingFactor

Number of parallel data paths, or vectors, to transform into serial, scalar data paths by time-multiplexing serial data paths and sharing hardware resources. The default is 0, which implements fully parallel data paths. See also “Streaming” on page 24-29.

UsePipelines

You can use this mode with Product blocks in Divide and Reciprocal modes. This property becomes available when you set the HDL architecture for the blocks to **ShiftAdd**. This architecture uses a non-restoring division algorithm that performs multiple shift and add operations to compute the quotient. The **ShiftAdd** architecture provides improved accuracy compared to the Newton-Raphson approximation method.

When you use the **ShiftAdd** architecture, you can use the **UsePipelines** implementation parameter to specify whether to use a pipelined or non-pipelined implementation of the non-restoring division.

UsePipelines Setting	Mapping Behavior
on (default)	Use a pipelined implementation of the non-restoring shift and add operation for Divide and Reciprocal blocks. This setting adds more delays to your design but achieves a higher maximum clock frequency on the target FPGA device. The number of pipelines inserted matches the number of iterations that the algorithm requires to compute the quotient or reciprocal.
off	Use a non-pipelined implementation of the non-restoring shift and add operation for Divide and Reciprocal blocks. This setting does not add delays to your design. As division and reciprocal are resource-intensive operations, to achieve higher clock frequencies on the target FPGA, set UsePipelines to on.

Set UsePipelines for Divide and Reciprocal Blocks

To set **UsePipelines** for a subsystem from the HDL Block Properties dialog box:

- 1 Right-click the subsystem.
- 2 Select **HDL Code** > **HDL Block Properties**.
- 3 For **UsePipelines**, select **on** or **off**.

To set `UsePipelines` for a block from the command line, use `hdlset_param`. For example, to turn off `UsePipelines` for a Divide block inside a subsystem, `my_dut`:

```
hdlset_param('my_dut/divide', 'UsePipelines', 'off');
```

See also `hdlset_param`.

UseRAM

The `UseRAM` implementation parameter enables using RAM-based mapping for a block instead of mapping to a shift register.

UseRAM Setting	Mapping Behavior
<code>off</code>	The delay maps to a shift register in the generated HDL code, except in one case. For details, see “Effects of Streaming and Distributed Pipelining” on page 22-27.
<code>on</code>	<p>The delay maps to a dual-port RAM block when the following conditions are true:</p> <ul style="list-style-type: none"> Initial value of the delay is zero. The Delay block does not have an external reset or enable port. Delay length > 4. Delay has one of the following set of numeric and data type attributes: <ul style="list-style-type: none"> (a) Real scalar with a non-floating-point data type (such as signed integer, unsigned integer, fixed point, or Boolean) (b) Complex scalar with real and imaginary parts that use non-floating-point data type (c) Vector where each element is either (a) or (b) <code>RAMSize</code> is greater than or equal to the <code>RAMMappingThreshold</code> value. <code>RAMSize</code> is the product <code>DelayLength * WordLength * ComplexLength</code>. <ul style="list-style-type: none"> <code>DelayLength</code> is the number of delays that the Delay block specifies. <code>WordLength</code> is the number of bits that represent the data type of the delay. <code>ComplexLength</code> is 2 for complex signals; 1 otherwise. <p>If any condition is false, the delay maps to a shift register in the HDL code unless it merges with other delays to map to a single RAM. For more information, see “Mapping Multiple Delays to RAM” on page 22-25.</p>

This implementation parameter is available for the Delay block in the Simulink Discrete library and the Delay block in the DSP System Toolbox Signal Operations library.

Mapping Multiple Delays to RAM

HDL Coder can also merge several delays of equal length into one delay and then map the merged delay to a single RAM. This optimization provides the following benefits:

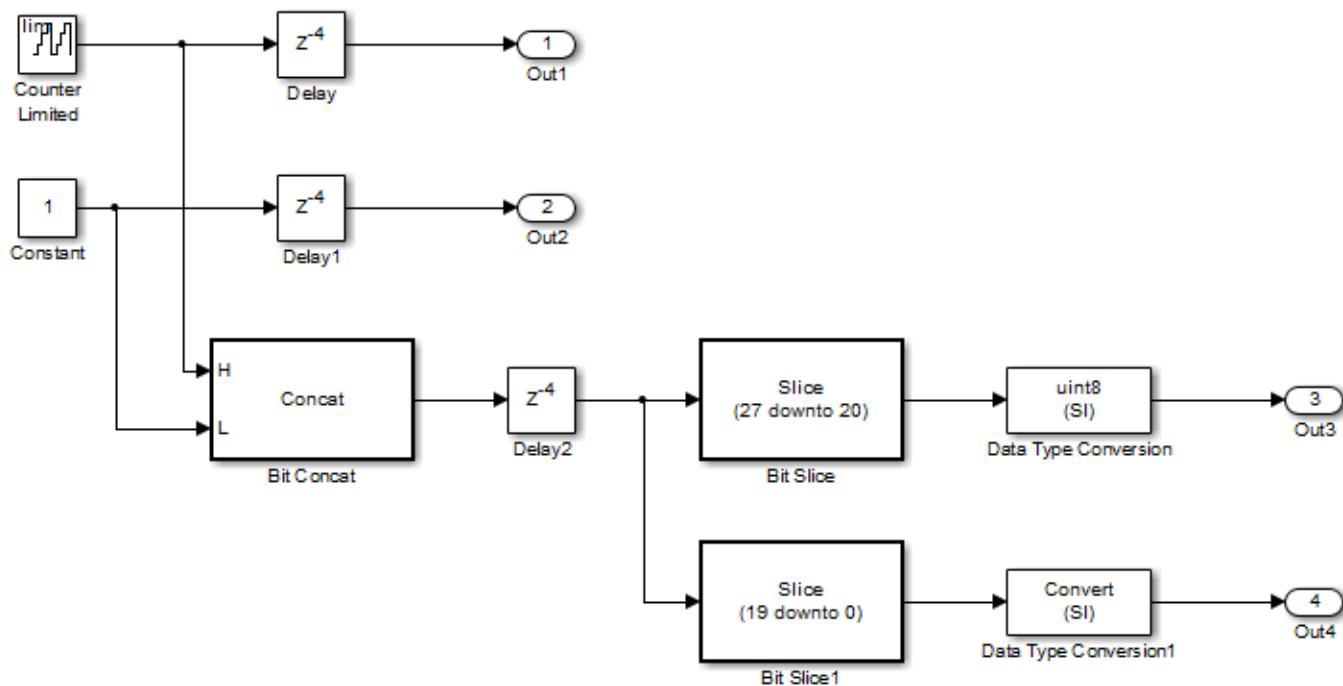
- Increased occupancy on a single RAM
- Sharing of address generation logic, which minimizes duplication of identical HDL code
- Mapping of delays to a RAM when the *individual* delays do not satisfy the threshold

The following rules control whether or not multiple delays can merge into one delay:

- The delays must:
 - Be at the same level of the subsystem hierarchy.
 - Use the same compiled sample time.
 - Have **UseRAM** set to **on**, or be generated by streaming or resource sharing.
 - Have the same **ResetType** setting, which cannot be **none**.
- The total word length of the merged delay cannot exceed 128 bits.
- The **RAMSize** of the merged delay is greater than or equal to the **RAMMappingThreshold** value. **RAMSize** is the product **DelayLength** * **WordLength** * **VectorLength** * **ComplexLength**.
 - DelayLength** is the total number of delays.
 - WordLength** is the number of bits that represent the data type of the merged delay.
 - VectorLength** is the number of elements in a vector delay. **VectorLength** is 1 for a scalar delay.
 - ComplexLength** is 2 for complex delays; 1 otherwise.

Example of Multiple Delays Mapping to a Block RAM

RAMMappingThreshold for the following model is 100 bits.



The Delay and Delay1 blocks merge and map to a dual-port RAM in the generated HDL code by satisfying the following conditions:

- Both delay blocks:
 - Are at the same level of the hierarchy.
 - Use the same compiled sample time.
 - Have **UseRAM** set to **on** in the HDL block properties dialog box.
 - Have the same **ResetType** setting of **default**.
- The total word length of the merged delay is 28 bits, which is below the 128-bit limit.
- The **RAMSize** of the merged delay is 112 bits (4 delays * 28-bit word length), which is greater than the mapping threshold of 100 bits.

When you generate HDL code for this model, HDL Coder generates additional files to specify RAM mapping. The coder stores these files in the same source location as other generated HDL files, for example, the `hdlsrc` folder.

Effects of Streaming and Distributed Pipelining

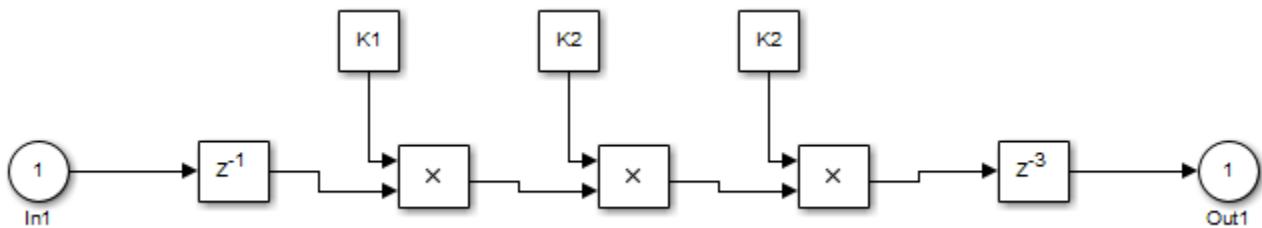
When **UseRAM** is **off** for a Delay block, HDL Coder maps the delay to a shift register by default. However, the coder changes the **UseRAM** setting to **on** and tries to map the delay to a RAM under the following conditions:

- Streaming is *enabled* for the subsystem with the Delay block.
- Distributed pipelining is *disabled* for the subsystem with the Delay block.

Suppose that distributed pipelining is *enabled* for the subsystem with the Delay block.

- When **UseRAM** is **off**, the Delay block participates in retiming.
- When **UseRAM** is **on**, the Delay block does not participate in retiming. HDL Coder does not break up a delay marked for RAM mapping.

Consider a subsystem with two Delay blocks, three Constant blocks, and three Product blocks:



When **UseRAM** is **on** for the Delay block on the right, that delay does not participate in retiming.

The following summary describes whether or not HDL Coder tries to map a delay to a RAM instead of a shift register.

UseRAM Setting for the Delay Block	Optimizations Enabled for Subsystem with Delay Block		
	Distributed Pipelining Only	Streaming Only	Both Distributed Pipelining and Streaming
On	Yes	Yes	Yes
Off	No	Yes, because mapping to a RAM instead of a shift register can provide an area-efficient design.	No

VariablesToPipeline

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

The VariablesToPipeline parameter enables you to insert a pipeline register at the output of one or more MATLAB variables. Specify a list of variables as a character vector, with spaces separating the variables.

See also “Pipeline MATLAB Expressions” on page 8-13.

HDL Block Properties: Native Floating Point

In this section...

- "Overview" on page 22-29
- "CheckResetToZero" on page 22-30
- "DivisionAlgorithm" on page 22-30
- "HandleDenormals" on page 22-31
- "InputRangeReduction" on page 22-32
- "LatencyStrategy" on page 22-33
- "CustomLatency" on page 22-34
- "NFPCustomLatency" on page 22-35
- "MantissaMultiplyStrategy" on page 22-36
- "MaxIterations" on page 22-37

Overview

Block implementation parameters enable you to control details of the code generated for specific block implementations. See “Set and View HDL Model and Block Parameters” on page 22-52 to learn how to select block implementations and parameters in the GUI or the command line.

Property names are specified as character vectors. The data type of a property value is specific to the property. This section describes the syntax of each block implementation parameter that you can specify in the **Native Floating Point** tab of the HDL Block Properties. You can see how specifying the parameter affects the generated code.

HDL Block Properties of Library Blocks

HDL block properties of library blocks are treated similar to mask parameters. When you instantiate library blocks in your model, the current HDL block properties of that library block are copied to instances of that block in your model. The HDL block properties of these instances are not synchronized with the HDL block properties of the library block. That is, if you change the HDL block property of the library block, the change does not get propagated to instances of the library block that you already added to your Simulink model. If you want the HDL block properties of a library block to be synchronized with its instances in the model, create a Subsystem and then place this block inside that Subsystem. The HDL block properties of blocks that reside inside the library block are synchronized with the corresponding instances in your model.

Suppose a library contains a Subsystem block with HDL architecture set to **Module**. When you instantiate this block in your model, the block instance uses **Module** as the HDL architecture. If you change the HDL architecture of the Subsystem block in the library to **BlackBox**, existing instances of that Subsystem block in your model still use **Module** as the HDL architecture. If you now add instances of the Subsystem block from the library in your model, the new block instances get a copy of the current HDL block properties, and therefore use **BlackBox** as the HDL architecture. If you want the HDL architecture of the Subsystem block in the library to be synchronized with its instances in the model, create a wrapper subsystem with the HDL architecture that you want inside this Subsystem.

CheckResetToZero

You can use the **CheckResetToZero** property for the mod and rem functions of the Math Function block in native floating-point mode. If you have numbers a and b such that the quotient a/b is close to an integer, this setting treats a as an integral multiple of b , and $\text{rem}(a,b)=0$. This result is numerically accurate and matches the Simulink simulation result. However, computing this result uses additional resources and increases the area footprint on the target FPGA device.

For example, for these sets of numbers, you get different simulation results when you enable and disable the **CheckResetToZero** setting.

CheckResetToZero Setting	Description
'on' (default)	When you compute mod or rem of two numbers whose quotient is closer to an integer, and has a precision greater than that of the floating point data type you use, HDL Coder adds the required logic to output the result of mod or rem as zero when the quotient of the numbers is close to an integer.
'off'	HDL Coder does not insert the additional logic to calculate the quotient, which saves area on the target FPGA device.

Set CheckResetToZero For the Math Function Block

To set **CheckResetToZero** for a block from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code HDL Block Properties**.
- 3 For **CheckResetToZero**, select **on** or **off**.

To set **CheckResetToZero** for the Math Function block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Math', 'CheckResetToZero', 'on')
```

See also `hdlset_param`.

DivisionAlgorithm

You can use the **DivisionAlgorithm** property when you enable **Native Floating Point** mode for the Divide block and the Math Function block in Reciprocal mode.

DivisionAlgorithm Setting	Description
Radix-2 (default)	The default Radix-2 mode performs repeated subtractions by computing one bit of the quotient in each iteration. To design for lower area usage while trading off for latency, use the Radix-2 mode in combination with the LatencyStrategy set to MAX.

DivisionAlgorithm Setting	Description
Radix-4	The Radix-4 mode performs repeated subtractions by computing two bits of the quotient in each iteration. To compute the result, the Radix-4 mode uses half the number of iterations that is required by the Radix-2 mode. To design for lower latency while trading off for area, use the Radix-4 mode in combination with the LatencyStrategy set to MAX.

Single-Precision Division Resource Utilization and Maximum Clock Frequency on Xilinx Virtex-7

DivisionAlgorithm Mode	LatencyStrategy	Latency	Fmax	LUTs	Registers
Radix-2	MIN	17	334.4MHz	1248	1011
	MAX	32	454.5MHz	1294	1797
Radix-4	MIN	11	245.5MHz	1956	865
	MAX	20	453.1MHz	1854	1522

Specify DivisionAlgorithm For the Math Function or Division Block

To specify **DivisionAlgorithm** for a block from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 In the **Native Floating Point** tab, specify the **DivisionAlgorithm**.

To specify **DivisionAlgorithm** for the block at the command line, use `hdlset_param`. For example, this command specifies Radix-4 mode for a Divide block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Divide', 'DivisionAlgorithm', 'Radix-4')
```

HandleDenormals

You can use the `HandleDenormals` property for certain blocks that support HDL code generation in Native Floating Point mode. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. With this setting, you can specify whether you want HDL Coder to insert additional logic to handle the denormal numbers in your design. For more information, see “Denormal Numbers” on page 10-84.

HandleDenormals Setting	Description
'inherit' (default)	Use the handle denormals setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the handle denormals setting for the model.

HandleDenormals Setting	Description
'on'	If you have denormal numbers at these block inputs, HDL Coder adds the logic to normalize the denormal numbers.
'off'	HDL Coder does not insert additional logic to handle denormal numbers in your design. The code generator treats the denormal value as zero before performing any computation.

To enable HandleDenormals for a block within a model, set the parameter, HandleDenormals, to 'on' for that block.

Set Handle Denormals For a Block

To set handle denormals for a block from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **HandleDenormals**, select **inherit**, **on**, or **off**.

To set handle denormals for a block from the command line, use `hdlset_param`. For example, to enable adaptive pipelining for a Product block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'HandleDenormals', 'on')
```

See also `hdlset_param`.

InputRangeReduction

You can use the **InputRangeReduction** property for the sin, cos, tan, sincos, and cos+jsin functions of the Trigonometric Function block in Native Floating Point mode. By default, this setting is enabled for the block, and it assumes that your input range is unbounded. If your input to the block is bounded in the range [-pi, pi], your design does not require the logic to reduce the input range. In that case, you can disable this setting, and the block implementation incurs a lower latency and uses fewer resources on the target hardware. When you disable the setting, the generated model contains a block that verifies whether the inputs are bounded in the range [-pi, pi]. If you have unbounded inputs, the generated model triggers an assertion during simulation.

InputRangeReduction Setting	Description
'on' (default)	Assumes that the input range is unbounded and inserts additional logic to reduce the input argument range to [-pi, pi] before computing the algorithm.
'off'	Assumes that the input argument is bounded in the range [-pi, pi] and does not insert the additional logic to reduce the input argument range. This implementation reduces the latency and saves area on the target platform.

Set InputRangeReduction For the Trigonometric Function Block

To set **InputRangeReduction** for a block from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code HDL Block Properties**.
- 3 In the **Native Floating Point** tab, for **InputRangeReduction**, select **on** or **off**.

To disable **InputRangeReduction** for the Trigonometric Function block inside a subsystem, `my_trigonometric` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/my_trigonometric', ...
    'InputRangeReduction', 'off')
```

See also `hdlset_param`.

LatencyStrategy

You can use the **LatencyStrategy** property for certain blocks that support HDL code generation for fixed-point and floating-point types. When you use floating-point types, set “Floating Point IP Library” on page 16-3 to **Native Floating Point**. For fixed-point types, the property specifies whether you want to use zero, maximum, or custom latency. For floating-point types, the property specifies whether you want the blocks in your design to map to minimum, maximum, or a custom latency for the operator.

LatencyStrategy Setting	Description
'inherit' (default)	Use the latency strategy setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the latency strategy setting for the model.
'Max'	During code generation, HDL Coder uses the maximum latency value for the native floating point operator.
'Min'	During code generation, HDL Coder uses the minimum latency value for the native floating point operator.
'Zero'	During code generation, HDL Coder does not add any latency for the native floating point operator.
'Custom'	During code generation, HDL Coder adds latency equal to the value that you specify for CustomLatency or NFPCustomLatency settings of the native floating point operator. You can use this setting for certain blocks in the native floating-point mode. To see the blocks for which you can specify the setting, see “NFPCustomLatency” on page 22-35.

To specify the minimum latency option for a block within a model, set the parameter, **LatencyStrategy**, to 'MIN' for that block.

To learn how to set model-level latency strategy setting, see “Latency Considerations with Native Floating Point” on page 10-96.

Set Latency Strategy for Fixed-Point Blocks Block

When you use fixed-point types, you can specify the **LatencyStrategy** for these blocks.

- Divide and Reciprocal blocks that have **ShiftAdd** as the HDL architecture.
- Sqrt block that has **SqrtFunction** as the HDL architecture.
- Trigonometric Function block that has **Function** set to **sin**, **cos**, **sincos**, **cos+jsin**, or **atan2** and **Approximation method** as **CORDIC**.

To set latency strategy for a subsystem from the HDL Block Properties dialog box:

- 1 In the Simulink toolbar, on the **Apps** tab, select **HDL Coder**.
- 2 Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
- 3 In the **General** tab, specify the **LatencyStrategy**. If you set **LatencyStrategy** to **Custom**, you must specify a value for the **CustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify the minimum latency for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'MAX')
```

See also `hdlset_param`.

Set Latency Strategy for Floating-Point Block

To set latency strategy for a subsystem from the HDL Block Properties dialog box:

- 1 In the Simulink toolbar, on the **Apps** tab, select **HDL Coder**.
- 2 Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
- 3 In the **Native Floating Point** tab, specify the **LatencyStrategy**. If you set **LatencyStrategy** to **Custom**, you must specify a value for the **NFPCustomLatency**.

For details, see the “HDL Code Generation” section of each block page. To learn about blocks for which you can specify a custom latency, see “`NFPCustomLatency`” on page 22-35.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify the minimum latency for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'MIN')
```

See also `hdlset_param`.

CustomLatency

You can specify a custom latency for certain blocks for fixed-point types. By using the custom latency strategy, you can trade-off between clock frequency and power consumption. To specify a custom latency strategy, set **LatencyStrategy** to **CUSTOM** and specify a value for **CustomLatency**. For details, see the “HDL Code Generation” section of each block page.

You can specify the **CustomLatency** setting for these blocks with fixed-point types.

- Divide and Reciprocal blocks that have **ShiftAdd** as the HDL architecture.
- Sqrt block that has **SqrtFunction** as the HDL architecture.
- Trigonometric Function block that has **Function** set to **sin**, **cos**, **sincos**, **cos+jsin**, or **atan2** and **Approximation method** as **CORDIC**.

Set Custom Latency Value For a Block

To set custom latency value for a subsystem from the HDL Block Properties dialog box:

- 1 In the Simulink toolbar, on the **Apps** tab, select **HDL Coder**.
- 2 Select the block, and on the **HDL Code** tab, click the **HDL Block Properties** button.
- 3 In the **General** tab, set **LatencyStrategy** to **Custom** and specify a value for **CustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify a custom latency of four for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'Custom')
hdlset_param('my_design/my_dut/Product', 'CustomLatency', 4)
```

See also `hdlset_param`.

NFPCustomLatency

You can specify a custom latency for certain blocks in the native floating-point mode. By using the custom latency strategy, you can trade-off between clock frequency and power consumption. To specify a custom latency strategy, set **LatencyStrategy** to **Custom** and specify a value for **NFPCustomLatency**. For details, see the "HDL Code Generation" section of each block page.

You can specify the **NFPCustomLatency** setting for these blocks with both **single** and **double** data types.

- Add
- Subtract
- Product
- Math Function in Reciprocal mode
- Gain
- Divide
- Relational Operator
- Data Type Conversion
- Rounding Function

You can also specify a **NFPCustomLatency** setting for these blocks with **single** data types.

- Sqrt
- Reciprocal Sqrt
- Sum of Elements

- Product of Elements

Set Custom Latency Value For a Block

To set custom latency value for a subsystem from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 In the **Native Floating Point** tab, for **LatencyStrategy**, select **Custom**.
- 4 Specify a value for the **NFPCustomLatency**.

To specify the latency strategy for a block from the command line, use `hdlset_param`. For example, to specify a custom latency of four for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'LatencyStrategy', 'Custom')
hdlset_param('my_design/my_dut/Product', 'NFPCustomLatency', 4)
```

See also `hdlset_param`.

MantissaMultiplyStrategy

You can use the **MantissaMultiplyStrategy** property for multipliers that support HDL code generation in native floating-point mode. Blocks that have this setting include Product, Divide, Math Function (in Reciprocal mode), and so on. By using this setting, you can specify how you want HDL Coder to implement the mantissa multiplication operation for the blocks.

MantissaMultiplyStrategy Setting	Description
'inherit' (default)	Use the mantissa multiply strategy setting of the parent subsystem. If this subsystem is the highest-level subsystem, use the mantissa multiply strategy setting for the model.
'FullMultiplier'	HDL Coder uses multipliers to perform the mantissa multiplication operation for the native floating point operator. The multipliers can utilize DSP units on the target device.
'PartMultiplierPartAddShift'	HDL Coder splits the implementation into two parts. One part is implemented with multipliers. The other part is implemented with a combination of adders and shifters. The multipliers can utilize the DSP units on the target device. The combination of adders and shifters does not utilize the DSP.
'NoMultiplierFullAddShift'	HDL Coder uses adders and shifters to implement the mantissa multiplication. This option does not utilize DSP units on the target device. You can also use this option if your target device does not contain DSP units.

To implement the mantissa multiplication with adders and shifters, set **MantissaMultiplyStrategy**, to '`NoMultiplierFullAddShift`' for that block.

Set Mantissa Multiply Strategy For a Block

To set adaptive pipelining for a subsystem from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 In the **Native Floating Point** tab, for **MantissaMultiplyStrategy**, select **inherit**, **FullMultiplier**, **PartMultiplierPartAddShift**, or **NoMultiplierFullAddShift**.

To specify the mantissa multiply strategy for a block from the command line, use `hdlset_param`. For example, to implement the mantissa multiplication using adders and shifters for a Product block inside a subsystem `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', ...
            'MantissaMultiplyStrategy', 'PartMultiplierPartAddShift')
```

See also `hdlset_param`.

MaxIterations

You can use the **MaxIterations** property for the mod and rem functions of the Math Function block in Native Floating Point mode. If you have numbers `a` and `b` that are significantly large integers, you can increase the **MaxIterations** setting to match the Simulink simulation result. However, computing this result uses additional resources and increases the area footprint on the target FPGA device.

MaxIterations Setting	Description
32 (default)	The default number of iterations to compute the result of mod and rem functions in Native Floating Point mode. This implementation can potentially result in a numerical mismatch with the Simulink simulation results for large integers.
64	Specify 64 as the number of iterations to compute the result of mod and rem functions in Native Floating Point mode. In this mode, the implementation has higher probability of matching the Simulink simulation result for significantly large integers but can use more hardware resources.
128	Specify 128 as the number of iterations to compute the result of mod and rem functions in Native Floating Point mode. In this mode, the implementation matches the Simulink simulation result for large integers but uses more hardware resources.

Set MaxIterations For the Math Function Block

To set **MaxIterations** for a block from the HDL Block Properties dialog box:

- 1 Right-click the block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 In the **Native Floating Point** tab, for **MaxIterations**, select **32**, **64**, or **128**.

To set handle denormals for a block from the command line, use `hdlset_param`. For example, to enable adaptive pipelining for a Product block inside a subsystem, `my_dut` in your Simulink model `my_design`:

```
hdlset_param('my_design/my_dut/Product', 'HandleDenormals', 'on')
```

See also `hdlset_param`.

See Also

Related Examples

- “Floating Point Support: Field-Oriented Control Algorithm” on page 10-109

More About

- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80

HDL Filter Block Properties

In this section...
"AdderTreePipeline" on page 22-39
"AddPipelineRegisters" on page 22-39
"ChannelSharing" on page 22-40
"CoeffMultipliers" on page 22-40
"DALUTPartition" on page 22-40
"DARadix" on page 22-41
"FoldingFactor" on page 22-42
"MultiplierInputPipeline" on page 22-42
"MultiplierOutputPipeline" on page 22-42
"NumMultipliers" on page 22-43
"ReuseAccum" on page 22-43
"SerialPartition" on page 22-43

AdderTreePipeline

This property applies to frame-based filters. It specifies how many pipeline registers the architecture includes between levels of the adder tree. These pipeline stages increase filter throughput while adding latency. The default value is `0`. To improve the speed of this architecture, the recommended setting is `2`.

Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For more information on the frame-based filter architecture, see “Frame-Based Architecture” on page 22-47.

AddPipelineRegisters

This property applies to scalar input filters. When you enable this property, the default linear adder of the filter is implemented as a pipelined tree adder instead. This architecture increases filter throughput while adding latency. The default value is `off`.

The following limitations apply to `AddPipelineRegisters`:

- If you use `AddPipelineRegisters`, the code generator forces full precision in the HDL and the generated filter model. This option implements a pipelined adder tree structure in the HDL code for which only full precision is supported. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.
- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

Note When you use this property with the CIC Interpolation block, delays in parallel paths are not automatically balanced. Manually add delays where needed by your design.

For filter architecture diagrams that indicate where the pipeline stages are added, see “HDL Filter Architectures” on page 22-45.

ChannelSharing

You can use the **ChannelSharing** implementation parameter with a multichannel filter to enable sharing a single filter implementation among channels for a more area-efficient design. This parameter is either 'on' or 'off'. The default is 'off', and a separate filter will be implemented for each channel.

See “Multichannel FIR Filter for FPGA” (DSP System Toolbox).

CoeffMultipliers

The **CoeffMultipliers** implementation parameter lets you specify use of canonical signed digit (CSD) or factored CSD optimizations for processing coefficient multiplier operations in code generated for certain filter blocks. Specify the **CoeffMultipliers** parameter using one of the following options:

- 'csd': Use CSD techniques to replace multiplier operations with shift-and-add operations. CSD techniques minimize the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits. This representation decreases the area used by the filter while maintaining or increasing clock speed.
- 'factored-csd': Use factored CSD techniques, which replace multiplier operations with shift-and-add operations on prime factors of the coefficients. This option lets you achieve a greater filter area reduction than CSD, at the cost of decreasing clock speed.
- 'multipliers' (default): Retain multiplier operations.

HDL Coder supports **CoeffMultipliers** for fully-parallel filter implementations. It is not supported for fully-serial and partly-serial architectures.

DALUTPartition

The size of the LUT grows exponentially with the order of the filter. For a filter with N coefficients, the LUT must have 2^N values. For higher order filters, LUT size must be reduced to reasonable levels. To reduce the size, you can subdivide the LUT into a number of LUTs, called LUT partitions. Each LUT partition operates on a different set of taps. The results obtained from the partitions are summed.

For example, for a 160-tap filter, the LUT size is $(2^{160}) * W$ bits, where W is the word size of the LUT data. Dividing this into 16 LUT partitions, each taking 10 inputs (taps), the total LUT size is reduced to $16 * (2^{10}) * W$ bits.

Although LUT partitioning reduces LUT size, more adders are required to sum the LUT data.

You can use **DALUTPartition** to enables DA code generation and specify the number and size of LUT partitions.

Specify LUT partitions as a vector of integers [p1 p2...pN] where:

- N is the number of partitions.
- Each vector element specifies the size of a partition. The maximum size for an individual partition is 12.

- The sum of all vector elements equals the filter length FL. FL is calculated differently depending on the filter type. You can find how FL is calculated for different filter types in the next section.

See “Distributed Arithmetic for HDL Filters” on page 22-50.

Specifying DALUTPartition for Single-Rate Filters

To determine the LUT partition for one of the supported single-rate filter types, calculate FL as shown in the following table. Then, specify the partition as a vector whose elements sum to FL.

Filter Type	Filter Length (FL) Calculation
Direct-form FIR	<code>FL = length(find(Hd.numerator ~= 0))</code>
Direct-form asymmetrical FIR, direct-form symmetrical FIR	<code>FL = ceil(length(find(Hd.numerator ~= 0))/2)</code>

You can also specify generation of DA code for your filter design without LUT partitioning. To do so, specify a vector of one element, whose value is equal to the filter length.

Specifying DALUTPartition for Multirate Filters

For supported multirate filters (FIR Decimation and FIR Interpolation), you can specify the LUT partition as

- A vector defining a partition for LUTs for all polyphase subfilters.
- A matrix of LUT partitions, where each row vector specifies a LUT partition for a corresponding polyphase subfilter. In this case, the FL is uniform for all subfilters. This approach provides fine control for partitioning each subfilter.

The following table shows the FL calculations for each type of LUT partition.

LUT Partition	Filter Length (FL) Calculation
Vector: Determine FL as shown in the Filter Length (FL) column to the right. Specify the LUT partition as a vector of integers whose elements sum to FL.	<code>FL = size(polyphase(Hm), 2)</code>
Matrix: Determine the subfilter length FLi based on the polyphase decomposition of the filter, as shown in the Filter Length (FL) column to the right. Specify the LUT partition for each subfilter as a row vector whose elements sum to FLi.	<code>p = polyphase(Hm);</code> <code>FLi = length(find(p(i,:)));</code> <p>where i is the index to the ith row of the polyphase matrix of the multirate filter. The ith row of the matrix p represents the ith subfilter.</p>

DARadix

The inherently bit-serial nature of DA can limit throughput. To improve throughput, the basic DA algorithm can be modified to compute more than one bit sum at a time. The number of simultaneously computed bit sums is expressed as a power of two called the DA radix. For example, a DA radix of 2 (2^1) indicates that one bit sum is computed at a time. A DA radix of 4 (2^2) indicates that two bit sums are computed at a time, and so on.

To compute more than one bit sum at a time, the LUT is replicated. For example, to perform DA on 2 bits at a time (radix 4), the odd bits are fed to one LUT and the even bits are simultaneously fed to an

identical LUT. The LUT results corresponding to odd bits are left-shifted before they are added to the LUT results corresponding to even bits. This result is then fed into a scaling accumulator that shifts its feedback value by 2 places.

Processing more than one bit at a time introduces a degree of parallelism into the operation, improving speed at the expense of area.

You can use **DARadix** to specify the number of bits processed simultaneously in DA. The number of bits is expressed as **N**, which must be:

- A nonzero positive integer that is a power of two
- Such that $\text{mod}(W, \log_2(N)) = 0$, where **W** is the input word size of the filter

The default value for **N** is 2, specifying processing of one bit at a time, or fully serial DA, which is slow but low in area. The maximum value for **N** is 2^W , where **W** is the input word size of the filter. This maximum specifies fully parallel DA, which is fast but high in area. Values of **N** between these extrema specify partly serial DA.

Note When setting a **DARadix** value for symmetrical and asymmetrical filters, see “Considerations for Symmetrical and Asymmetrical Filters” on page 22-51.

See “Distributed Arithmetic for HDL Filters” on page 22-50.

FoldingFactor

FoldingFactor specifies the total number of clock cycles taken for the computation of filter output in an IIR SOS filter with serial architecture. It is complementary with “**NumMultipliers**” on page 22-43. You must select one property or the other; you cannot use both. If you do not specify either **FoldingFactor** or **NumMultipliers**, HDL code for the filter is generated with fully parallel architecture.

MultiplierInputPipeline

You can use this parameter to generate a specified number of pipeline stages at multiplier inputs for FIR filter structures. The default value is 0.

The following limitation applies to **MultiplierInputPipeline**:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For diagrams of where these pipeline stages occur in the filter architecture, see “HDL Filter Architectures” on page 22-45.

MultiplierOutputPipeline

You can use this parameter to generate a specified number of pipeline stages at multiplier outputs for FIR filter structures. The default value is 0.

The following limitation applies to **MultiplierOutputPipeline**:

- Pipeline stages introduce delays along the path in the model that contains the affected filter. When you enable this pipeline option, the coder automatically adds balancing delays on parallel data paths.

For diagrams of where these pipeline stages occur in the filter architecture, see “HDL Filter Architectures” on page 22-45.

NumMultipliers

NumMultipliers specifies the total number of multipliers used for the filter implementation in an IIR SOS filter with serial architecture. It is complementary with “**FoldingFactor**” on page 22-42 property. You must select one property or the other; you cannot use both. If you do not specify either **FoldingFactor** or **NumMultipliers**, HDL code for the filter is generated with fully parallel architecture.

ReuseAccum

You can use this parameter to enable or disable accumulator reuse in a serial HDL architecture. The default is a fully parallel architecture.

To Generate This Architecture...	Set ReuseAccum to...
Fully parallel	Omit this property
Fully serial	Not specified, or 'off'
Partly serial	'off'
Cascade-serial with explicitly specified partitioning	'on'
Cascade-serial with automatically optimized partitioning	'on'

For more information on parallel and serial filter architectures, see “HDL Filter Architectures” on page 22-45

SerialPartition

Use this parameter to specify partitions for a serial filter architecture. The default is a fully parallel architecture.

To Generate This Architecture...	Set SerialPartition to...
Fully parallel	Omit this property
Fully serial	N, where N is the length of the filter

To Generate This Architecture...	Set SerialPartition to...
Partly serial	<p>[p1 p2 p3...pN]: A vector of integers having N elements, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. When you define the partitioning for a partly serial architecture, consider the following:</p> <ul style="list-style-type: none"> The filter length should be divided as uniformly as possible into a vector equal in length to the number of multipliers intended. For example, if your design requires a filter of length 9 with 2 multipliers, the recommended partition is [5 4]. If your design requires 3 multipliers, the recommended partition is [3 3 3] rather than some less uniform division such as [1 4 4] or [3 4 2]. If your design is constrained by the need to compute each output value (corresponding to each input value) in an exact number N of clock cycles, use N as the largest partition size and partition the other elements as uniformly as possible. For example, if the filter length is 9 and your design requires exactly 4 cycles to compute the output, define the partition as [4 3 2]. This partition executes in 4 clock cycles, at the cost of 3 multipliers.
Cascade-serial with explicitly specified partitioning	[p1 p2 p3...pN]: A vector of N integers, where N is the number of serial partitions. Each element of the vector specifies the length of the corresponding partition. The sum of the vector elements must be equal to the length of the filter. The values of the vector elements must be in descending order, except the last two elements, which can be equal. For example, for a filter length of 8, partitions [5 3] or [4 2 2] are valid, but the partitions [2 2 2 2] and [3 2 3] raise an error at code generation time.
Cascade-serial with automatically optimized partitioning	Omit this property.

For more information on parallel and serial filter architectures, see “HDL Filter Architectures” on page 22-45.

This property is also used for Min/Max blocks with cascade-serial architectures. For how to configure Min/Max cascades, see “SerialPartition” on page 22-22.

See Also

More About

- “Set and View HDL Model and Block Parameters” on page 22-52
- “HDL Block Properties: General” on page 22-3

HDL Filter Architectures

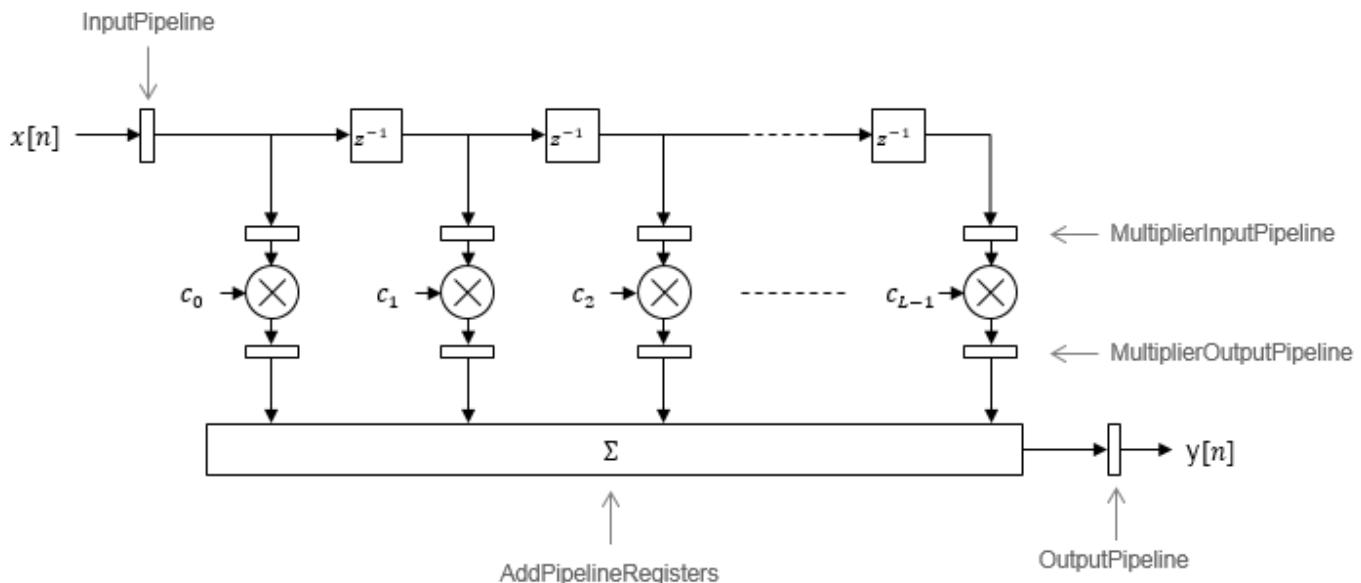
The HDL Coder software provides architecture options that extend your control over speed vs. area tradeoffs in the realization of filter designs. To achieve the desired tradeoff for generated HDL code, you can either specify a fully parallel architecture, or choose one of several serial architectures. Configure a serial architecture using the “SerialPartition” on page 22-43 and “ReuseAccum” on page 22-43 parameters. You can also choose a frame-based filter for increased throughput.

Use pipelining parameters to improve speed performance of your filter designs. Add pipelines to the adder logic of your filter using AddPipelineRegisters on page 22-39 for scalar input filters, and “AdderTreePipeline” on page 22-39 for frame-based filters. Specify pipeline stages before and after each multiplier with MultiplierInputPipeline on page 22-42 and MultiplierOutputPipeline on page 22-42. Set the number of pipeline stages before and after the filter using “InputPipeline” on page 22-13 and “OutputPipeline” on page 22-17. The architecture diagrams show the locations of the various configurable pipeline stages.

Fully Parallel Architecture

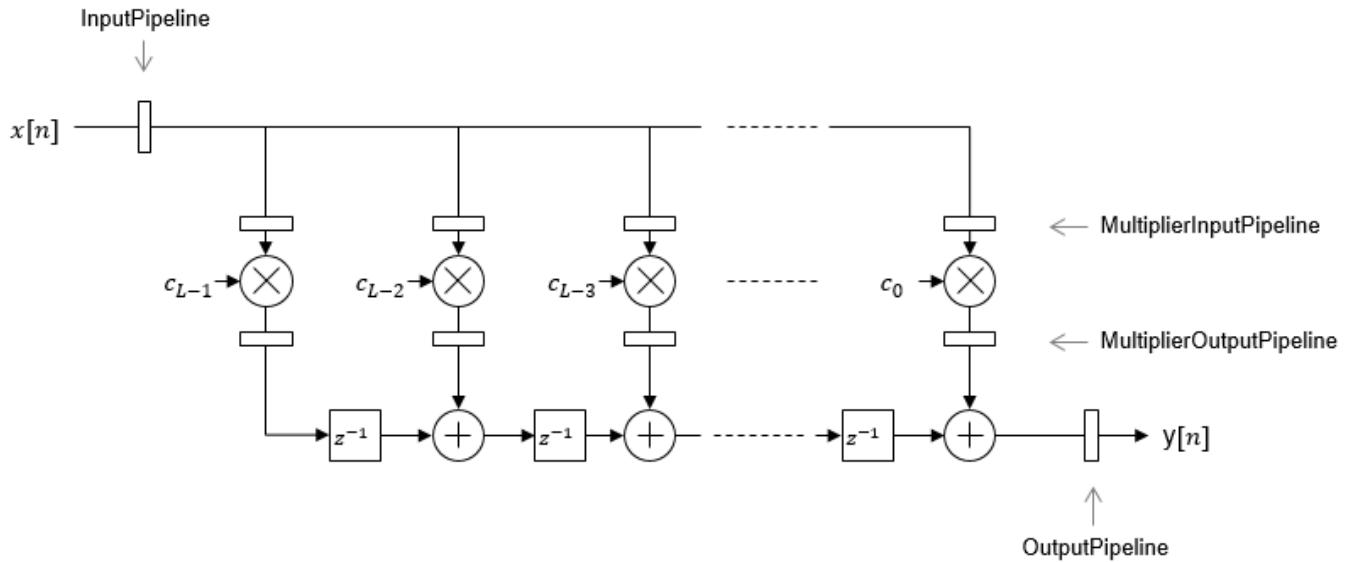
This option is the default architecture. A fully parallel architecture uses a dedicated multiplier and adder for each filter tap. The taps execute in parallel. A fully parallel architecture is optimal for speed. However, it requires more multipliers and adders than a serial architecture, and therefore consumes more chip area. The diagrams show the architectures for direct form and for transposed filter structures with fully parallel implementations, and the location of configurable pipeline stages.

Direct Form



By default, the block implements linear adder logic. When you enable AddPipelineRegisters, the adder logic is implemented as a pipelined adder tree. The adder tree uses full-precision data types. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Transposed



The AddPipelineRegisters parameter has no effect on a transposed filter implementation.

Serial Architectures

Serial architectures reuse hardware resources in time, saving chip area. Configure a serial architecture using the “SerialPartition” on page 22-43 and “ReuseAccum” on page 22-43 parameters. The available serial architecture options are *fully serial*, *partly serial*, and *cascade serial*.

Fully Serial

A fully serial architecture conserves area by reusing multiplier and adder resources sequentially. For example, a four-tap filter design uses a single multiplier and adder, executing a multiply-accumulate operation once for each tap. The multiply-accumulate section of the design runs at four times the filter's input/output sample rate. This design saves area at the cost of some speed loss and higher power consumption.

In a fully serial architecture, the system clock runs at a much higher rate than the sample rate of the filter. Thus, for a given filter design, the maximum speed achievable by a fully serial architecture is less than that of a parallel architecture.

Partly Serial

Partly serial architectures cover the full range of speed vs. area tradeoffs that lie between fully parallel and fully serial architectures.

In a partly serial architecture, the filter taps are grouped into a number of serial partitions. The taps within each partition execute serially, but the partitions execute in parallel with respect to one another. The outputs of the partitions are summed at the final output.

When you select a partly serial architecture, you specify the number of partitions and the length (number of taps) of each partition. Suppose you specify a four-tap filter with two partitions, each having two taps. The system clock runs at twice the filter's sample rate.

Cascade Serial

A cascade-serial architecture closely resembles a partly serial architecture. As in a partly serial architecture, the filter taps are grouped into a number of serial partitions that execute in parallel with respect to one another. However, the accumulated output of each partition is cascaded to the accumulator of the previous partition. The output of all partitions is therefore computed at the accumulator of the first partition. This technique is termed *accumulator reuse*. A final adder is not required, which saves area.

The cascade-serial architecture requires an extra cycle of the system clock to complete the final summation to the output. Therefore, the frequency of the system clock must be increased slightly with respect to the clock used in a noncascade partly serial architecture.

To generate a cascade-serial architecture, specify a partly serial architecture with accumulator reuse enabled. If you do not specify the serial partitions, HDL Coder automatically selects an optimal partitioning.

Latency in Serial Architectures

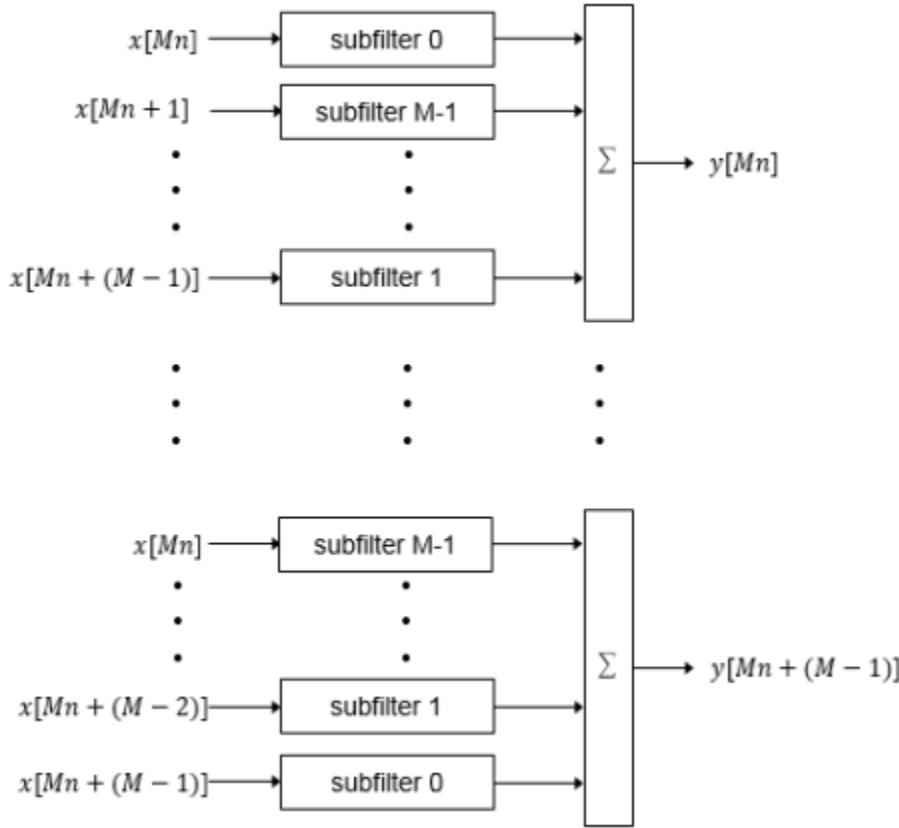
Serialization of a filter increases the total latency of the design by one clock cycle. The serial architectures use an accumulator (an adder with a register) to add the products sequentially. An additional final register is used to store the summed result of all the serial partitions, requiring an extra clock cycle for the operation. To model this latency, HDL Coder inserts a Delay block into the generated model after the filter block.

Full-Precision for Serial Architectures

When you choose a serial architecture, the code generator uses full precision in the HDL code. HDL Coder therefore forces full precision in the generated model. If you generate a validation model, you must use full precision in the original model to avoid validation mismatches.

Frame-Based Architecture

When you select a frame-based architecture and provide an M -sample input frame, the coder implements a fully parallel filter architecture. The filter includes M parallel subfilters for each input sample.



Each of the subfilters includes every M th coefficient. The subfilter results are added so that each output sample is the sum of each of the coefficients multiplied with one input sample.

subfilter 0 = c_0, c_M, \dots

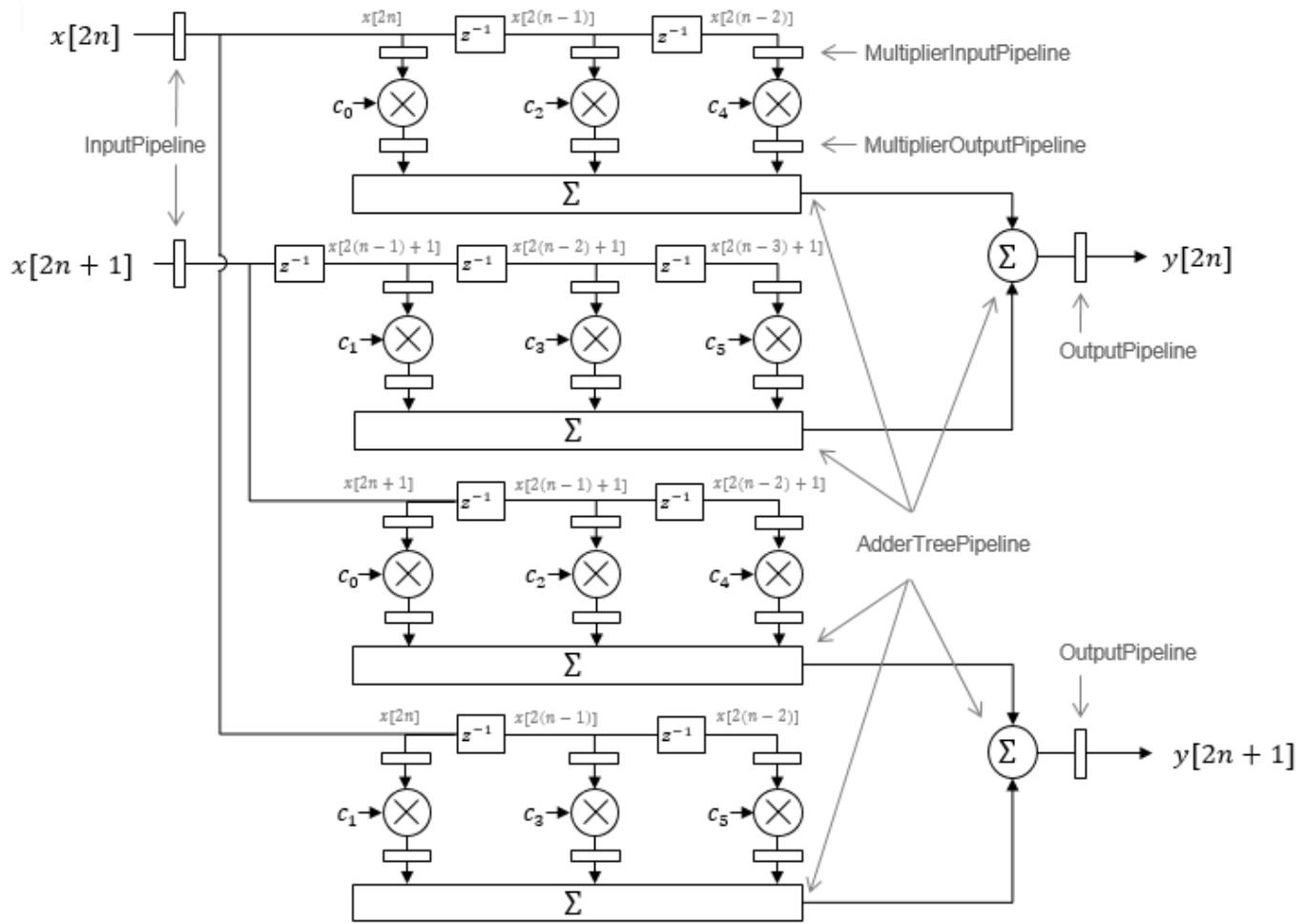
subfilter 1 = c_1, c_{M+1}, \dots

...

subfilter $M-1 = c_{M-1}, c_{2M-1}, \dots$

The diagram shows the filter architecture for a frame size of two samples ($M = 2$), and a filter length of six coefficients. The input is a vector with two values representing samples in time. The input samples, $x[2n]$ and $x[2n+1]$, represent the n th input pair. Every second sample from each stream is fed to two parallel subfilters. The four subfilter results are added together to create two output samples. In this way, each output sample is the sum of each of the coefficients multiplied with one of the input samples.

The sums are implemented as a pipelined adder tree. Set “AdderTreePipeline” on page 22-39 to specify the number of pipeline stages between levels of the adder tree. To improve clock speed, it is recommended that you set this parameter to 2. To fit the multipliers into DSP blocks on your FPGA, add pipeline stages before and after the multipliers using MultiplierInputPipeline on page 22-42 and MultiplierOutputPipeline on page 22-42.



For symmetric or antisymmetric coefficients, the filter architecture reuses the coefficient multipliers and adds design delay between the multiplier and summation stages as required.

See Also

More About

- “HDL Filter Block Properties” on page 22-39
- “Distributed Arithmetic for HDL Filters” on page 22-50

Distributed Arithmetic for HDL Filters

Distributed Arithmetic (DA) is a widely used technique for implementing sum-of-products computations without the use of multipliers. Designers frequently use DA to build efficient Multiply-Accumulate Circuitry (MAC) for filters and other DSP applications. The main advantage of DA is its high computational efficiency. DA distributes multiply and accumulate operations across shifters, lookup tables (LUTs) and adders in such a way that conventional multipliers are not required.

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

The delay line is tapped (based on the input word size W), to form a W -bit address that indexes into a lookup table (LUT). The LUT stores all possible sums of partial products over the filter coefficients space. The LUT is followed by a shift and adder (scaling accumulator) that adds the values obtained from the LUT sequentially.

A table lookup is performed sequentially for each bit (in order of significance starting from the LSB). On each clock cycle, the LUT result is added to the accumulated and shifted result from the previous cycle. For the last bit (MSB), the table lookup result is subtracted, accounting for the sign of the operand.

This basic form of DA is fully serial, operating on one bit at a time. If the input data sequence is W bits wide, then a FIR structure takes W clock cycles to compute the output. Symmetric and asymmetric FIR structures are an exception, requiring $W+1$ cycles, because one additional clock cycle is needed to process the carry bit of the preadders.

You can control how DA code is generated by using the `DALUTPartition` and `DARadix` implementation parameters. The `DALUTPartition` and `DARadix` parameters have certain requirements and restrictions that are specific to different filter types. These requirements are included in the discussions of each parameter.

- Reduce LUT Size: “`DALUTPartition`” on page 22-40
- Improve Performance with Parallelism: “`DARadix`” on page 22-41

For information on the theoretical foundations of DA, see “Further References” on page 22-51.

Requirements and Considerations for Generating Distributed Arithmetic Code

Fixed-Point Quantization Required

Generation of DA code is supported only for fixed-point filter designs.

Specifying Filter Precision

The data path in HDL code generated for the DA architecture is carefully optimized for full precision computations. The filter result is cast to the output data size only at the final stage when it is presented to the output.

Distributed arithmetic merges the product and accumulator operations and does computations at full precision. This approach ignores the **Product output** and **Accumulator** properties of the Digital Filter block and sets these properties to full precision.

Coefficients with Zero Values

DA ignores taps that have zero-valued coefficients and reduces the size of the DA LUT accordingly.

Considerations for Symmetrical and Asymmetrical Filters

For symmetrical and asymmetrical filters:

- A bit-level preadder or presubtractor is required to add tap data values that have coefficients of equal value and/or opposite sign. One extra clock cycle is required to compute the result because of the additional carry bit.
- HDL Coder takes advantage of filter symmetry where possible. This reduces the DA LUT size substantially, because the effective filter length for these filter types is halved.

Further References

Detailed discussions of the theoretical foundations of DA appear in the following publications:

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

Set and View HDL Model and Block Parameters

In this section...

- “Set HDL Block Parameters” on page 22-52
- “Set HDL Block Parameters for Multiple Blocks Programmatically” on page 22-52
- “View All HDL Block Parameters” on page 22-54
- “View Non-Default HDL Block Parameters” on page 22-54
- “View HDL Model Parameters” on page 22-54

You can view and set HDL-related block properties, such as implementation and implementation parameters, at the model level and at the individual block level.

Set HDL Block Parameters

To set the HDL Block parameters from the UI, open the HDL Block Properties dialog box, and modify the block properties as needed. To open the HDL Properties dialog box, either:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the block for which you want to see the HDL parameters and then select **HDL Block Properties**.
- Right-click the block and select **HDL Code > HDL Block Properties**.

To set the HDL-related parameters at the command line, use `hdlset_param`.

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

For example, to set the sharing factor to 2 and the architecture to Tree for a block in your model:

- 1 Open the model and select the block.
- 2 Enter the following at the command line:

```
hdlset_param (gcb, 'SharingFactor', 2, 'Architecture', 'Tree')
```

To view the HDL parameters specified for a block, use `hdlget_param`. For example, to see the HDL architecture setting for a block, at the command line, enter:

```
hdlget_param(gcb, 'Architecture')
```

You can also assign the returned HDL block parameters to a cell array. In the following example, `hdlget_param` returns all HDL block parameters and values to the cell array *p*.

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

```
p =
```

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

Set HDL Block Parameters for Multiple Blocks Programmatically

For models that contain a large number of blocks, using the **HDL Block Properties** dialog box to select block implementations or set implementation parameters for individual blocks may not be

practical. It is more efficient to set HDL-related model or block parameters for multiple blocks programmatically. You can use the `find_system` function to locate the blocks of interest. Then, use a loop to call `hdlset_param` to set the desired parameters for each block.

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

```
open_system('sfir_fixed')

% Find all Product blocks in the model
prodblocks = find_system('sfir_fixed/symmetric_fir', ...
    'BlockType', 'Product')

% Set the output pipeline to 2 for the blocks
for ii=1:length(prodblocks)
    hdlset_param(prodblocks{ii}, 'OutputPipeline', 2)
end

prodblocks =
4x1 cell array

{'sfir_fixed/symmetric_fir/m1'}
{'sfir_fixed/symmetric_fir/m2'}
{'sfir_fixed/symmetric_fir/m3'}
{'sfir_fixed/symmetric_fir/m4'}
```

To verify the settings, use `hdlget_param` to display the value of the `OutputPipeline` parameter for the blocks .

```
% Get the output pipeline to 2 for the blocks
for ii=1:length(prodblocks)
    hdlget_param(prodblocks{ii}, 'OutputPipeline')
end

ans =
2

ans =
2

ans =
2

ans =
2
```

View All HDL Block Parameters

`hdldispblkparams` displays the HDL block parameters available for a specified block.

The following example displays HDL block parameters and values for the currently selected block.

```
hdldispblkparams(gcb, 'all')
```

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

Implementation

Architecture : Linear

Implementation Parameters

```
InputPipeline : 0
OutputPipeline : 0
```

See also `hdldispblkparams`.

View Non-Default HDL Block Parameters

The following example displays only HDL block parameters that have non-default values for the currently selected block.

```
hdldispblkparams(gcb)
```

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

Implementation

Architecture : Linear

Implementation Parameters

```
OutputPipeline : 3
```

View HDL Model Parameters

To display the names and values of HDL-related properties in a model, use the `hdldispmdlparams` function.

The following example displays HDL-related properties and values of the current model, in alphabetical order by property name.

```
hdldispmdlparams(bdroot, 'all')
```

```
%%%%%%
```

```
HDL CodeGen Parameters
%%%%%%%%%%%%%
AddPipelineRegisters      : 'off'
Backannotation             : 'on'
BlockGenerateLabel         : '_gen'
CheckHDL                  : 'off'
ClockEnableInputPort       : 'clk_enable'
.
.
.
VerilogFileExtension      : '.v'
```

The following example displays only HDL-related properties that have non-default values.

```
hdldispmdlparams(bdroot)

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

CodeGenerationOutput      : 'GenerateHDLCodeAndDisplayGeneratedModel'
HDLSubsystem               : 'simplevectorsum/vsum'
ResetAssertedLevel        : 'Active-low'
Traceability                : 'on'
```

See Also

Functions

`hdldispblkparams` | `hdldispmdlparams` | `hdlget_param` | `hdlset_param`

More About

- “HDL Block Properties: General” on page 22-3
- “HDL Block Properties: Native Floating Point” on page 22-29

Pass through, No HDL, and Cascade Implementations

Pass-through and No HDL Implementations

Implementation	Description
Pass-through implementations	<p>Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. HDL Coder supports the following blocks with a pass-through implementation:</p> <ul style="list-style-type: none">• Convert 1-D to 2-D• Reshape• Signal Conversion• Signal Specification
No HDL	<p>The <code>NoHDL</code> implementation completely removes the block from the generated code. Thus, you can use the block in simulation but treat it as a “no-op” in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but would be meaningless in HDL code.</p> <p>You can also use this implementation as an alternative implementation for subsystems.</p>

For more information related to special-purpose implementations, see “External Component Interfaces”.

Cascade Architecture Best Practices

HDL Coder supports cascade implementations for the Sum of Elements, Product of Elements, and MinMax blocks. These implementations require multiple clock cycles to process their inputs; therefore, their inputs must be kept unchanged for their entire sample-time period. Generated test benches accomplish this by using a register to drive the inputs.

A recommended design practice, when integrating generated HDL code with other HDL code, is to provide registers at the inputs. While not strictly required, adding registers to the inputs improves timing and avoids problems with data stability for blocks that require multiple clock cycles to process their inputs.

Build a ROM Block with Simulink Blocks

HDL Coder does not provide a ROM block, but you can easily build one using basic Simulink blocks. The Getting Started with RAM and ROM example includes a ROM built using a 1-D Lookup Table block and a Unit Delay block. To open the example, type the following command at the MATLAB prompt:

```
hdlcoderramrom
```

Getting Started with RAM and ROM in Simulink®

This example shows how to utilize RAM resources in your FPGA design using HDL Coder™.

Introduction

Dedicated RAM blocks in FPGA are valuable resources for digital designs. It is easy to design with RAM and ROM in Simulink®, and utilize the dedicated RAM blocks available in your FPGA using HDL Coder.

RAM Blocks in the HDL Example Library

HDL Coder provides following types of RAM blocks in the HDL RAMs block library. Use `hdllib` to display blocks that are compatible with HDL Coder™, and then select the `HDL RAMs` library under `HDL Coder`:

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

```
% Run this command navigate to RAM blocks in HDL library
% hdllib
```

The RAM blocks are masked subsystems built using Simulink blocks for behavioral simulation. For code generation, HDL Coder generates predefined templates that describe RAM structures in HDL. Most synthesis tools recognize the RAM structures in the templates, and map them to RAM resources on the FPGA. For more information, see [HDL Coder Block Library - RAM Blocks](#) in the documentation.

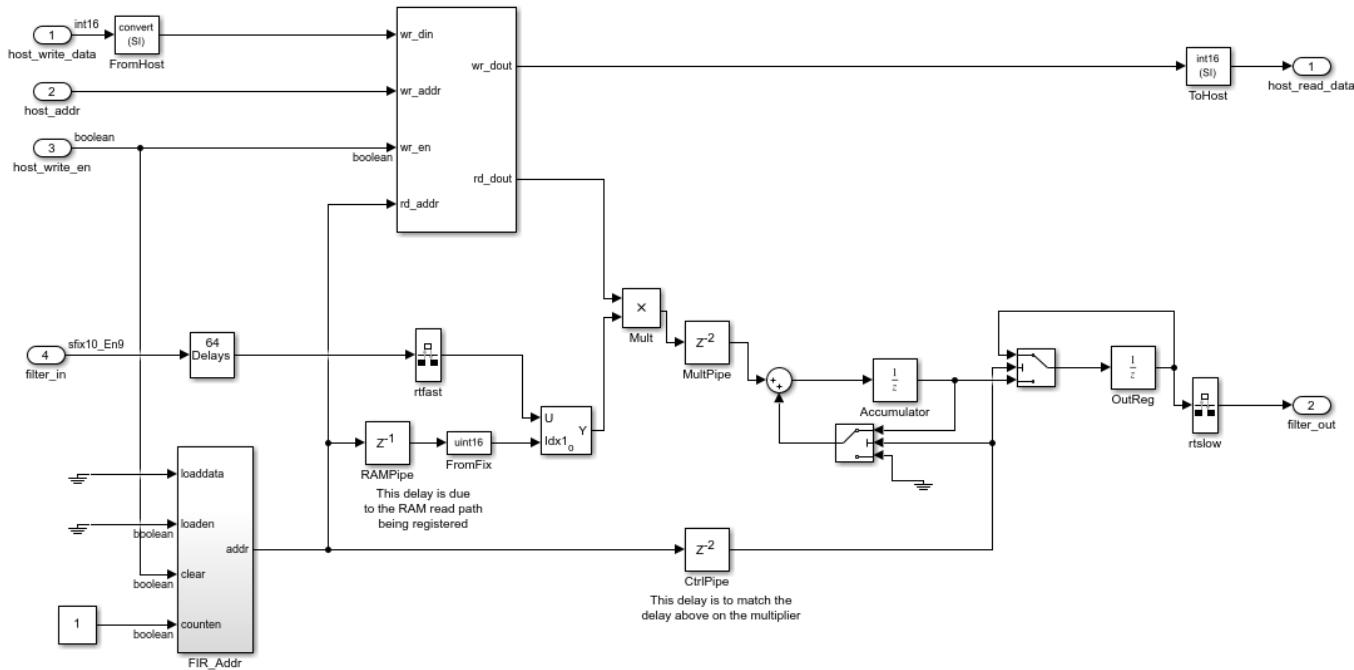
Using Generic RAM Coding Style for RAM Blocks

By default, HDL Coder provides RAM template that uses clock enable for the RAM structures. As an alternative, HDL Coder also provides a style of generic template that does not use clock enable. The generic RAM style template implements clock enable with logic in a wrapper around the RAM. You can control this using 'RAM Architecture' option in the HDL Coder global settings panel.

You may want to use the generic RAM style if your synthesis tool does not support RAM structures with a clock enable, and cannot map the HDL to FPGA RAM resources as a result.

The example `hdlcoderfirram` is an example of how to use the generic RAM style for your design.

```
open_system('hdlcoderfirram');
open_system('hdlcoderfirram/FIR_RAM');
```



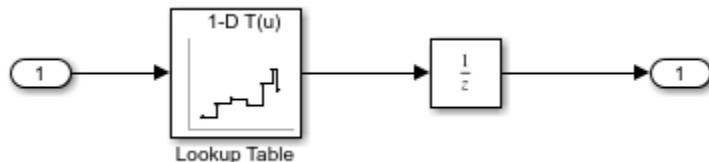
Selecting RAM Coding Style

The RAM Coding Style is selected by choosing the desired RAM Architecture on the "Coding Style" tab of the HDL Code Generation Global Settings configuration page.

Building a ROM Using Simulink Blocks

HDL Coder does not provide a ROM block, but you can easily build one using a Lookup Table block and a Unit Delay block from Simulink, as shown in the following example.

```
open_system('hdlcoderrom');
open_system('hdlcoderrom/ROM');
```



Follow these modeling guidelines when building a ROM from Simulink:

- For an n-bit address, specify all 2^n entries of the Lookup Table data. Otherwise, your synthesis tool may not map the generated code to RAM, and the code may not match your Simulink model.
- Place the Lookup Table and Unit Delay blocks in the same model hierarchy.
- Support of RAM reset logic varies among FPGA devices and synthesis tools. For best synthesis result, suppress the generation of reset logic for the Unit Delay block by setting its 'ResetType' property to 'none', in the HDL Block Properties dialog box. Also set the 'IgnoreDataChecking' property to 1 in the HDL Test Bench configuration parameters to ignore the initial simulation mismatch caused by suppressing the reset logic.

If you follow the preceding guidelines, most synthesis tools will implement the ROM using dedicated RAM blocks in an FPGA.

Minimum RAM Size Requirement

If the size of the RAM or ROM in your design is small, your synthesis tools may map the generated code to registers instead of dedicated RAM blocks for better speed performance. Check your synthesis tool for any minimum RAM size requirement, and if desired, how you may override that requirement.

Wireless Communications Design for FPGAs and ASICs

In this section...

- “From Mathematical Algorithm to Hardware Implementation” on page 22-61
- “HDL-Optimized Blocks” on page 22-63
- “Reference Applications” on page 22-63
- “Generate HDL Code and Prototype on FPGA” on page 22-64

Deploying algorithmic models to FPGA hardware makes it possible to do over-the-air testing and verification. However, designing wireless communications systems for hardware requires design tradeoffs between hardware resources and throughput. You can speed up hardware design and deployment by using HDL-optimized blocks that have hardware-suitable interfaces and architectures, reference applications that implement portions of the LTE and 5G NR physical layer, and automatic HDL code generation. You can also use hardware support packages to assist with deploying and verifying your design on real hardware.

MathWorks® HDL products, such as Wireless HDL Toolbox, allow you to start with a mathematical model, such as MATLAB code from LTE Toolbox™ or 5G Toolbox™, and design a hardware implementation of that algorithm that is suitable for FPGAs and ASICs.

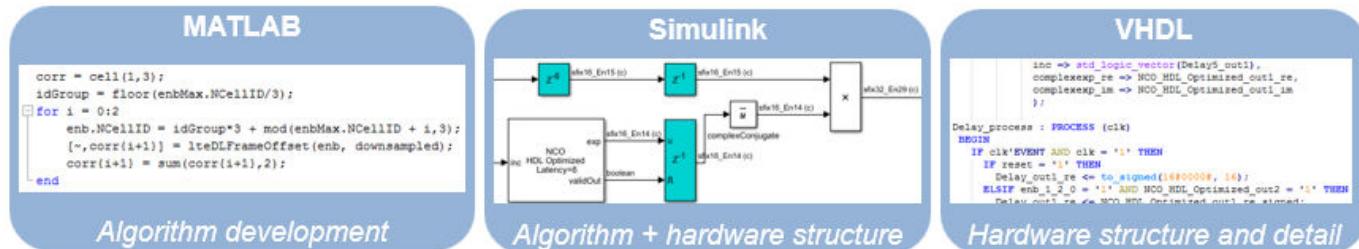
From Mathematical Algorithm to Hardware Implementation

Wireless communications design often starts with algorithm development and testing using MATLAB functions. MATLAB code, which usually operates on matrices of floating-point data, is good for developing mathematical algorithms, manipulating large data sets, and visualizing data.

Hardware engineers typically receive a mathematical specification from an algorithm team, and reimplement the algorithm for hardware. Hardware designs require tradeoffs of resource usage for clock speed and overall throughput. Usually this tradeoff means operating on streaming data, and using some logic to control the storage and flow of data. Hardware engineers usually work in hardware description languages (HDLs), like VHDL and Verilog, that provide cycle-based modeling and parallelism.

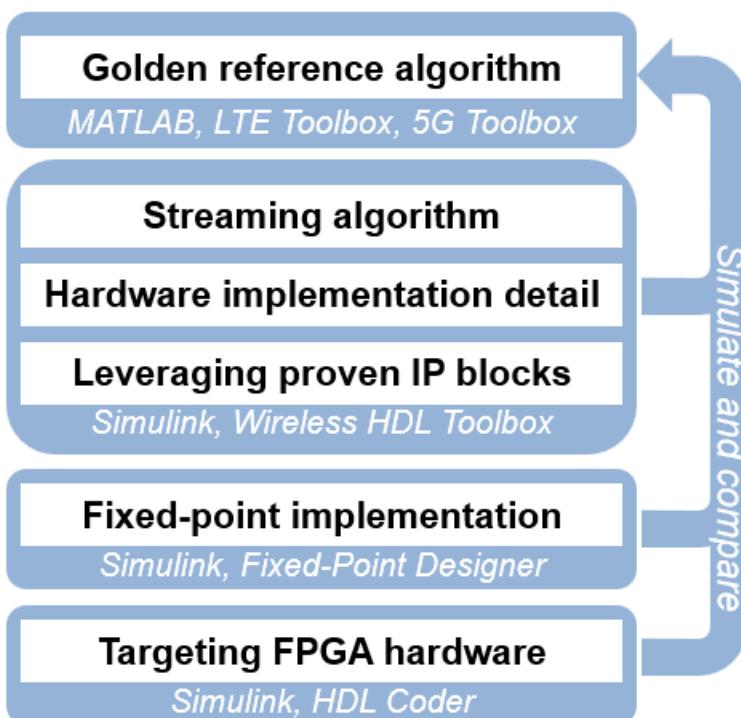
To bridge this gap between mathematical algorithm and hardware implementation, use the MATLAB algorithm model as a starting point for hardware implementation. Make incremental changes to the design to make it suitable for hardware, and progress towards a Simulink model that you can use to automatically generate HDL code by using HDL Coder.

This diagram shows the design progression from mathematical algorithm in MATLAB, to hardware-compatible implementation in Simulink, and then the generated VHDL code.

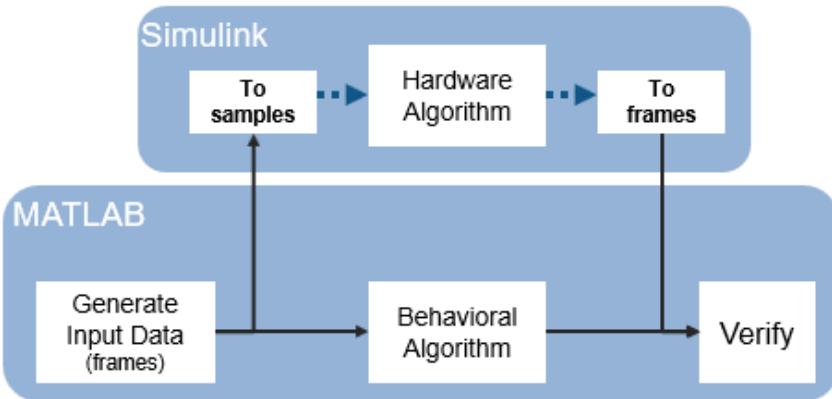


While both MATLAB and Simulink support automatic generation of HDL code, you must construct your design with hardware requirements in mind, and Simulink is better-suited for cycle-based modeling for hardware. It can represent parallel data paths and streaming data with control signals to manage the timing of the data stream. To aid in fixed-point type choices, it clearly visualizes data type propagation in the design. It also allows for easy pipelining of mathematical operations to improve maximum clock frequency in hardware.

While you create your hardware-ready design, use the MATLAB algorithm as a "golden reference" to verify that each version of the design still meets the mathematical requirements. The workflow shown in the diagram uses MATLAB and Simulink as collaboration and communication tools between the algorithm and hardware design teams.



For instance, when designing for LTE or 5G wireless standards, you can use LTE Toolbox and 5G Toolbox functions to create a golden reference in MATLAB. Then transition to Simulink and create a hardware-compatible implementation by using library blocks from Wireless HDL Toolbox and blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation. You can reuse test and data generation infrastructure from MATLAB by importing data from MATLAB to your Simulink model and returning the output of the model to MATLAB to verify it against the "golden reference".



HDL-Optimized Blocks

Library blocks from Wireless HDL Toolbox implement encoders, decoders, modulators, demodulators, and sequence generators for use in an LTE, 5G, or general wireless communications system. These blocks use a standard streaming data interface for hardware. This interface makes it easy to connect parts of the algorithm together, and includes control signals that manage the flow of data and mark frame boundaries. These blocks support automatic HDL code generation with HDL Coder. You can also use blocks from Communications Toolbox and DSP System Toolbox that support HDL code generation.

The blocks provide hardware-suitable architectures that optimize resource use, such as including adder and multiplier pipelining to fit well into FPGA DSP slices. They also support automatic and configurable fixed-point data types. Using predefined blocks also allows you to try different parameter configurations without changing the rest of the design.

For lists of blocks that support HDL code generation, see [Wireless HDL Toolbox Block List \(HDL Code Generation\)](#), [Communications Toolbox Block List \(HDL Code Generation\)](#), and [DSP System Toolbox Block List \(HDL Code Generation\)](#).

Reference Applications

Wireless HDL Toolbox provides reference applications that contain hardware-ready implementations of large parts of the LTE and 5G NR physical layer. These designs are verified against the "golden reference" functions provided by LTE Toolbox and 5G Toolbox. They have also been tested on FPGA boards to confirm that they encode and decode over-the-air waveforms and use a reasonable amount of hardware resources. They are designed to be modular, scalable, and extensible so you can insert additional physical channels. The receiver design was tested using waveforms captured off-the-air.

The suite of reference applications includes:

- LTE and 5G NR primary and secondary synchronization signal (PSS/SSS) generation and detection
- LTE downlink shared control channel detector and master information block (MIB) generation and recovery
- LTE first system information block (SIB1) decoder
- Hardware-software interface models for MIB and SIB1 bit parsing and channel estimation data indexing

- LTE waveform generation for multiple-antenna transmission
- Support for FDD and TDD for LTE transmitter and receiver applications

These reference applications can be used as-is to deliver packet information to your unique application and to generate synthesizable VHDL or Verilog with HDL Coder. They also serve as examples to illustrate recommended practices for implementing communications algorithms on FPGA or ASIC hardware.

Generate HDL Code and Prototype on FPGA

Wireless HDL Toolbox provides blocks that support HDL code generation. To generate HDL code from designs that use these blocks, you must have an HDL Coder license. HDL Coder produces device-independent code with signal names that correspond to the Simulink model. HDL Coder also provides a tool to drive the FPGA synthesis and targeting process, and enables you to generate scripts and test benches for use with third-party HDL simulators.

To assist with the setup and targeting of programmable logic on a prototype board, and to verify your wireless communications system design on hardware, download a hardware support package such as Communications Toolbox Support Package for Xilinx Zynq®-Based Radio.

See Also

External Websites

- Wireless HDL Toolbox
- HDL Coder

Generating HDL Code for Multirate Models

- “Code Generation from Multirate Models” on page 23-2
- “Timing Controller for Multirate Models” on page 23-4
- “Generate Reset for Timing Controller” on page 23-5
- “Multirate Model Requirements for HDL Code Generation” on page 23-6
- “Generate a Global Oversampling Clock” on page 23-8
- “Using Multiple Clocks in HDL Coder” on page 23-12
- “Using Triggered Subsystems for HDL Code Generation” on page 23-16
- “Generate Multicycle Path Information Files” on page 23-19
- “Meet Timing Requirements Using Enable-Based Multicycle Path Constraints” on page 23-26
- “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 23-32

Code Generation from Multirate Models

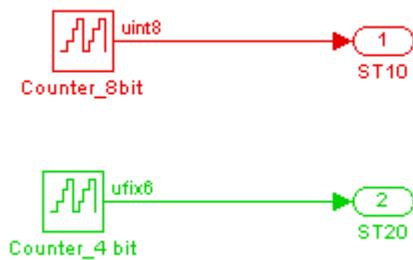
HDL Coder supports HDL code generation for single-clock and multiple clock multirate models. Your model can include blocks running at multiple sample rates:

- Within the device under test (DUT).
- In the test bench driving the DUT. In this case, the DUT inherits multiple sample rates from its inputs or outputs.
- In both the test bench and the DUT.

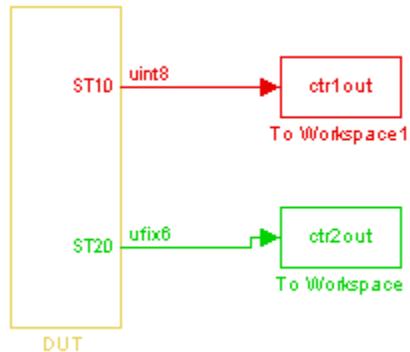
In general, generating HDL code for a multirate model does not differ greatly from generating HDL code for a single-rate model. However, there are a few requirements and restrictions on the configuration of the model and the use of specialized blocks (such as Rate Transitions) that apply to multirate models. For details, see “Multirate Model Requirements for HDL Code Generation” on page 23-6.

Clock Enable Generation for a Multirate DUT

The following block diagram shows the interior of a subsystem containing blocks that are explicitly configured with different sample times. The upper and lower Counter Free-Running blocks have sample times of 10 s and 20 s respectively. The counter output signals are routed to output ports ST10 and ST20, which inherit their sample times. The signal path terminating at ST10 runs at the base rate of the model; the signal path terminating at ST20 is a substrate signal, running at half the base rate of the model.



As shown in the next figure, the outputs of the multirate DUT drive To Workspace blocks in the test bench. These blocks inherit the sample times of the DUT outputs.

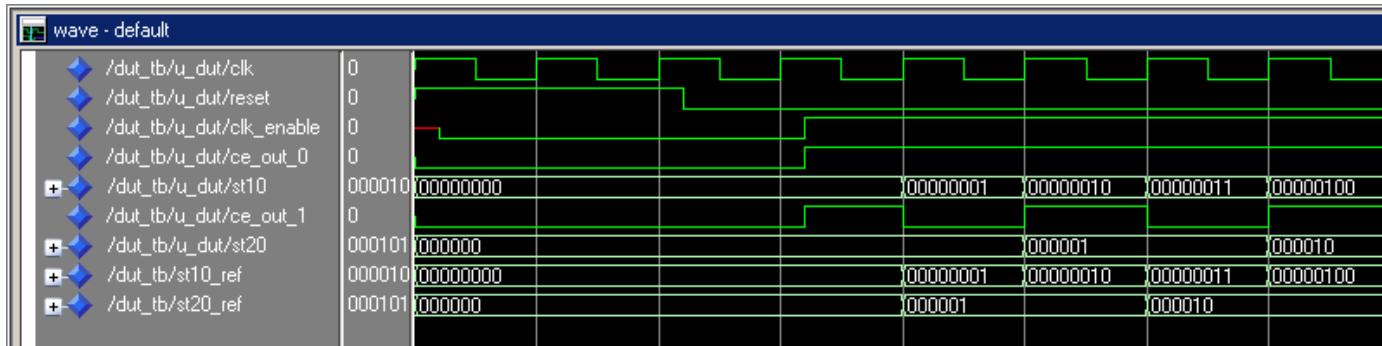


The following listing shows the VHDL entity declaration generated for the DUT.

```
ENTITY DUT IS
  PORT( clk           : IN  std_logic;
        reset         : IN  std_logic;
        clk_enable    : IN  std_logic;
        ce_out_0      : OUT std_logic;
        ce_out_1      : OUT std_logic;
        ST10          : OUT std_logic_vector(7 DOWNTO 0);  -- uint8
        ST20          : OUT std_logic_vector(5 DOWNTO 0)  -- ufix6
      );
END DUT;
```

The entity has the standard clock, reset, and clock enable inputs and data outputs for the ST10 and ST20 signals. In addition, the entity has two clock enable outputs (`ce_out_0` and `ce_out_1`). These clock enable outputs replicate internal clock enable signals maintained by the timing controller entity.

The following figure, showing a portion of a Mentor Graphics ModelSim simulation of the generated VHDL code, lets you observe the timing relationship of the base rate clock (`clk`), the clock enables, and the computed outputs of the model.



After the assertion of `clk_enable` (replicated by `ce_out_0`), a new value is computed and output to `ST10` for every cycle of the base rate clock.

A new value is computed and output for substrate signal `ST20` for every other cycle of the base rate clock. An internal signal, `enb_1_2_1` (replicated by `ce_out_1`) governs the timing of this computation.

Timing Controller for Multirate Models

A timing controller entity generates the required rates from a single master clock, using one or more counters to create multiple clock enables. The master clock rate is the fastest rate in the model in single clock mode. In multiple clock mode, it can be any clock in the DUT. The outputs of the timing controller are clock enable signals running at rates an integer multiple slower than the timing controller's master clock.

When using single clock mode, HDL code generated from multirate models employs a single master clock that corresponds to the base rate of the DUT. When using multiple clock mode, HDL code generated from multirate models employs one clock input for each rate in the DUT. The number of timing controllers generated in multiple clock mode depends on the design in the DUT.

Each timing controller entity definition is written to a separate code file. The timing controller file and entity names derive from the name of the subsystem that is selected for code generation (the DUT). To form the timing controller name, HDL Coder appends the value of the `TimingControllerPostfix` property to the DUT name.

See Also

Functions

`makehdl` | `makehdltb`

Simulink Configuration Parameters

[Optimize timing controller](#) | [Timing controller architecture](#) | [Timing controller postfix](#)

Related Examples

- “Using Multiple Clocks in HDL Coder” on page 23-12

Generate Reset for Timing Controller

In this section...

- "Requirements for Timing Controller Reset Port Generation" on page 23-5
- "How To Generate Reset for Timing Controller" on page 23-5
- "Limitations for Timing Controller Reset Port Generation" on page 23-5

You can generate a reset port for the timing controller, which generates the clock, clock enable, and reset signals in a multirate DUT. In the generated code, the reset for the timing controller is a DUT input port.

Requirements for Timing Controller Reset Port Generation

Your design must use single-clock mode. That is, the `ClockInputs` property value must be '`Single`'.

How To Generate Reset for Timing Controller

To generate a reset port for the timing controller, set the `TimingControllerArch` property to '`resettable`' using `makehdl` or `hdlset_param`.

To disable reset port generation for the timing controller, set the `TimingControllerArch` property to '`default`'.

For example, for a model, `sfir_fixed`, specify a reset port for the timing controller by entering:

```
hdlset_param('sfir_fixed','TimingControllerArch','resettable')
```

Limitations for Timing Controller Reset Port Generation

The following workflows are not compatible with timing controller reset port generation:

- FPGA Turnkey
- FPGA-in-the-Loop
- Custom IP core generation

Multirate Model Requirements for HDL Code Generation

In this section...

["Model Configuration Parameters" on page 23-6](#)

["Sample Rate" on page 23-6](#)

["Blocks To Use For Rate Transitions" on page 23-6](#)

Model Configuration Parameters

Before generating HDL code, configure the parameters of your model using the `hdlsetup` command. This sets up your multirate model for HDL code generation. This section summarizes settings applied to the model by `hdlsetup` that are relevant to multirate code generation. These include:

- **Solver** options that are recommended or required for HDL code generation:
 - **Type:** Fixed-step.
 - **Solver:** Discrete (no continuous states). Other fixed-step solvers could be selected, but this option is usually best for simulating discrete systems.
 - **Tasking mode:** Must be explicitly set to SingleTasking. Do not set **Tasking mode** to Auto.
- `hdlsetup` configures the following **Diagnostics / Sample time** options for all models:
 - **Multitask rate transition:** error
 - **Single task rate transition:** error

In multirate models intended for HDL code generation, Rate Transition blocks must be explicitly inserted when blocks running at different rates are connected. Set **Multitask rate transition** and **Single task rate transition** to **error** to detect illegal rate transitions before code is generated.

To learn more about the settings that `hdlsetup` configures, see “Check for model parameters suited for HDL code generation” on page 38-5.

Sample Rate

HDL Coder requires that at least one valid sample rate (sample time > 0) must exist in the model. If all rates are 0, -1, or -2, the code generator (`makehdl`) and compatibility checker (`checkhdl`) terminates with an error message.

Blocks To Use For Rate Transitions

Use Rate Transition blocks, rather than the following blocks, to create rate transitions in models intended for HDL code generation:

- Delay
- Tapped Delay
- Unit Delay
- Unit Delay Enabled
- Zero-Order Hold

The Delay blocks listed should be configured to have the same input and output sample rates.

Zero-Order Hold blocks must be configured with inherited (-1) sample times.

Generate a Global Oversampling Clock

In this section...

- “Why Use a Global Oversampling Clock?” on page 23-8
- “Requirements for the Oversampling Factor” on page 23-8
- “Specifying the Oversampling Factor From the GUI” on page 23-8
- “Specifying the Oversampling Factor From the Command Line” on page 23-9
- “Resolving Oversampling Rate Conflicts” on page 23-9

Why Use a Global Oversampling Clock?

In many designs, the DUT is not self-contained. For example, consider a DUT that is part of a larger system that supplies timing signals to its components under control of a global clock. The global clock typically runs at a higher rate than some of the components under its control. By specifying such a global oversampling clock, you can integrate your DUT into a larger system without using Upsample or Downsample blocks.

To generate global clock logic, you specify an oversampling factor. The oversampling factor expresses the desired rate of the global oversampling clock as a multiple of the base rate of your model.

When you specify an oversampling factor, HDL Coder generates the global oversampling clock and derives the required timing signals from clock signal. Generation of the global oversampling clock affects only generated HDL code. The clock does not affect the simulation behavior of your model.

Requirements for the Oversampling Factor

When you specify the oversampling factor for a global oversampling clock, note these requirements:

- The oversampling factor must be an integer greater than or equal to 1.
- The default value is 1. In the default case, HDL Coder does not generate a global oversampling clock.
- Some DUTs require multiple sampling rates for their internal operations. In such cases, the other rates in the DUT must divide evenly into the global oversampling rate. For more information, see “Resolving Oversampling Rate Conflicts” on page 23-9 .

Specifying the Oversampling Factor From the GUI

You can specify the oversampling factor for a global clock from the GUI as follows:

- 1 Select the **HDL Code Generation > Global Settings** pane in the Configuration Parameters dialog box.
- 2 For **Oversampling factor** in the **Clock settings** section, enter the desired oversampling factor. In the following figure, **Oversampling factor** specifies a global oversampling clock that runs at ten times the base rate of the model.
- 3 Click **Generate** on the **HDL Code Generation** pane to initiate code generation.

HDL Coder reports the oversampling clock rate:

```
### Begin VHDL Code Generation
### MESSAGE: The design requires 10 times faster clock with respect to the base rate = 1.
```

```
### Working on symmetric_fir_tc as hdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir as hdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
```

Specifying the Oversampling Factor From the Command Line

You can specify the oversampling factor for a global clock from the command line by setting the **Oversampling** property with **hdlset_param** or **makehdl**. The following example specifies an oversampling factor of 7:

```
makehdl(gcb, 'Oversampling', 7)

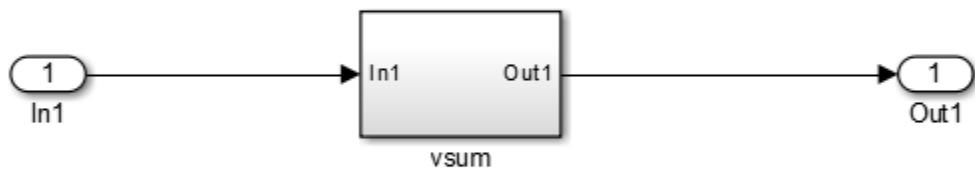
### Generating HDL for 'sfir_fixed/symmetric_fir'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin VHDL Code Generation
### MESSAGE: The design requires 7 times faster clock with respect to the base rate = 1.
### Working on symmetric_fir_tc as hdlsrc\symmetric_fir_tc.vhd
### Working on sfir_fixed/symmetric_fir as hdlsrc\symmetric_fir.vhd
### HDL Code Generation Complete.
```

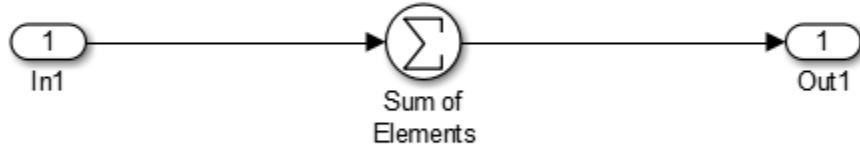
Resolving Oversampling Rate Conflicts

The HDL realization of some designs is inherently multirate, even though the original Simulink model is single-rate. As an example, consider the **simplevectorsum_cascade** model.

This model consists of a subsystem, **vsum**, driven by a vector input of width 10, with a scalar output. The following figure shows the root level of the model.



The device under test is the **vsum** subsystem, shown in the following figure. The subsystem contains a Sum block, configured for vector summation.



The `simplevectorsum_cascade` model specifies a cascaded implementation (`SumCascadeHDLImplementation`) for the Sum block. The generated HDL code for a cascaded vector Sum block implementation runs at two effective rates: a faster (oversampling) rate for internal computations and a slower rate for input/output. HDL Coder reports that the inherent oversampling rate for the DUT is five times the base rate:

```
dut = 'simplevectorsum_cascade/vsum';
makehdl(dut);

### Generating HDL for 'simplevectorsum_cascade/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The code generation and optimization options you have chosen have introduced
additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for
compensation.
### The DUT requires an initial pipeline setup latency. Each output port
experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### MESSAGE: The design requires 5 times faster clock with respect to the
base rate = 1.

...
```

In some cases, the clock requirements for such a DUT conflict with the global oversampling rate. To avoid oversampling rate conflicts, verify that subrates in the model divide evenly into the global oversampling rate.

For example, if you request a global oversampling rate of 8 for the `simplevectorsum_cascade` model, the coder displays a warning and ignores the requested oversampling factor. The coder instead respects the oversampling factor that the DUT requests:

```
dut = 'simplevectorsum_cascade/vsum';
makehdl(dut,'Oversampling',8);

### Generating HDL for 'simplevectorsum/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The code generation and optimization options you have chosen have introduced
additional pipeline delays.
```

```

### The delay balancing feature has automatically inserted matching delays for
compensation.
### The DUT requires an initial pipeline setup latency. Each output port
experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### WARNING: The design requires 5 times faster clock with respect to
the base rate = 1, which is incompatible with the oversampling
value (8). Oversampling value is ignored.

...

```

An oversampling factor of 10 works in this case:

```

dut = 'simplevectorsum_cascade/vsum';
makehdl(dut,'Oversampling',10);

### Generating HDL for 'simplevectorsum_cascade/vsum'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### The code generation and optimization options you have chosen have introduced
additional pipeline delays.
### The delay balancing feature has automatically inserted matching delays for
compensation.
### The DUT requires an initial pipeline setup latency. Each output port
experiences these additional delays
### Output port 0: 1 cycles

### Begin VHDL Code Generation
### MESSAGE: The design requires 10 times faster clock with respect to
the base rate = 1.

...

```

Using Multiple Clocks in HDL Coder

This example shows how to instantiate multiple top-level synchronous clock input ports in HDL Coder™.

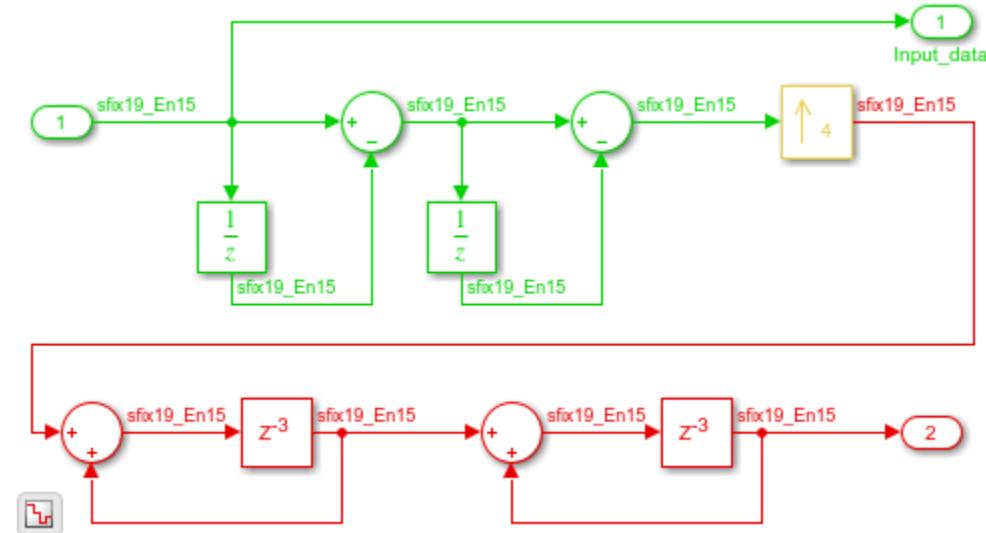
Overview of Clocking Modes

HDL Coder has two clocking modes. One mode generates a single clock input to the Device Under Test (DUT). The other mode generates a synchronous primary clock input for each Simulink® rate in the DUT. By default, HDL Coder creates an HDL design that uses a single clock port for the DUT. In single clock mode, if multiple rates exist in the Simulink model, a timing controller is created to control the clocking to the portions of the model that run at a slower rate. The timing controller generates a set of clock enables with the necessary rate and phase information to control the clocking for the design. Each generated clock enable is an integer multiple slower than the primary clock rate. Each output signal rate is associated with a clock enable output signal that indicates the correct timing to sample the output data.

In synchronous multiple clock mode, the generated code has a set of clock ports as primary inputs to the DUT. Each clock port corresponds to a separate rate in the model. Transitions between rates require clock enables at a given rate that are out of phase with that rate's clock. These out of phase signals are generated with a timing controller. A multiple clock model may require multiple timing controllers.

The first example uses a multirate CIC Interpolation filter in single clock mode. The filter's input is also presented as an output for this example to present a model with output signals running at different rates.

```
load_system('hdlcoder_clockdemo');
open_system('hdlcoder_clockdemo/DUT');
set_param('hdlcoder_clockdemo', 'SimulationCommand', 'update');
```



Single Clock Mode DUT Timing Interface

In single clock mode, the HDL code for the DUT has a set of three signals that do not appear in the Simulink diagram added to it. Collectively, these are a clock bundle that contains signals for clock,

master clock enable, and reset. These signals appear in the VHDL Entity declaration and are used throughout the generated code.

```
hdlset_param('hdlcoder_clockdemo', 'Traceability', 'on');
makehdl('hdlcoder_clockdemo/DUT');

### Generating HDL for 'hdlcoder_clockdemo/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockde...
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc as hdsrc\hdlcoder_clockdemo\DU...
### Working on hdlcoder_clockdemo/DUT as hdsrc\hdlcoder_clockdemo\DU...
### Generating package file hdsrc\hdlcoder_clockdemo\DU...
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_152...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Clock Summary Reporting in Single Clock Mode

The file comment block in the HDL DUT code contains Clock Summary information. In single clock mode, this report contains a table detailing the sample rates for each clock enable output signal. The report also contains a table listing each user output signal and its associated clock enable output signal. Any time a HTML report is generated, the Clock Summary Report is also generated.

Generating Synchronous Multiclock HDL Code

To generate multiple synchronous clocks for this design, the **ClockInputs** property must be set to **multiple**. Either change the **ClockInputs** property at the command line using **makehdl** or change the **Clock inputs** setting to **Multiple** on the **HDL Code Generation > Global Settings** tab of the Configuration Parameters dialog box.

```
makehdl('hdlcoder_clockdemo/DUT', 'ClockInputs', 'multiple');

### Generating HDL for 'hdlcoder_clockdemo/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_clockde...
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_clockdemo'.
### Working on DUT_tc_d1 as hdsrc\hdlcoder_clockdemo\DU...
### Working on hdlcoder_clockdemo/DUT as hdsrc\hdlcoder_clockdemo\DU...
### Generating package file hdsrc\hdlcoder_clockdemo\DU...
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_152...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_clockdemo' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Clock Summary Information in Multiclock Mode

The contents of the Clock Summary are different in multiple clock mode. The report now contains a clock table. This table has one entry for each primary DUT clock. It describes the relative clock ratio between each clock and the fastest clock in the model. As with single clock mode, this information is presented both in the HDL DUT file comment block and the HTML report.

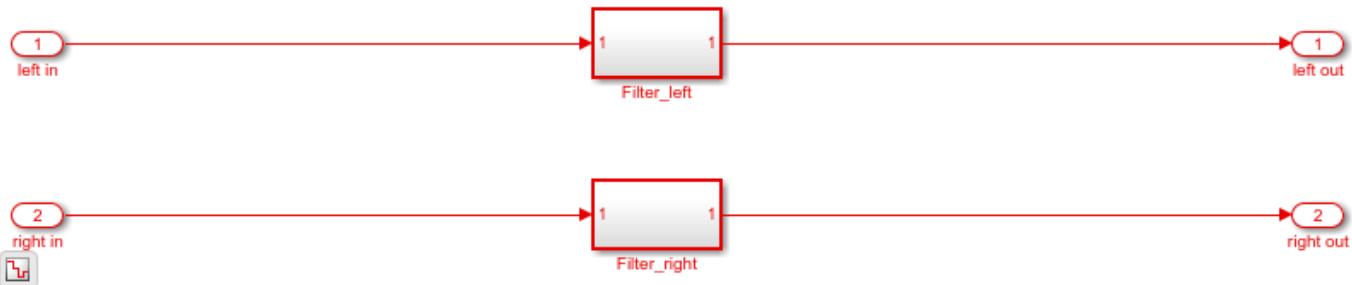
Multiclock Mode and HDL Coder Optimizations

Multiple synchronous clocks can be useful even for a design with only a single Simulink rate. Various optimizations can require clock rates faster than indicated in the original model. The following

example demonstrates an audio filtering model that applies the same filter on the left and right channels. By default, HDL Coder would generate two filter modules in hardware. With this configuration, multiple clock mode still only generates one clock, just as single clock mode does.

```
bdclose_hdlcoder_clockdemo;
load_system('hdlcoder_audiofiltering');
open_system('hdlcoder_audiofiltering/Audio filter');
hdlset_param('hdlcoder_audiofiltering', 'ClockInputs', 'Multiple');
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 0);
makehdl('hdlcoder_audiofiltering/Audio filter', 'Traceability', 'on');

### Generating HDL for 'hdlcoder_audiofiltering/Audio filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiofiltering')".
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### Working on hdlcoder_audiofiltering/Filter_left as hdsrc\hdlcoder_audiofiltering\Filter_left
### Working on hdlcoder_audiofiltering/Filter as hdsrc\hdlcoder_audiofiltering\Filter
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp...
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Using Multiple Clock Mode with Resource Sharing

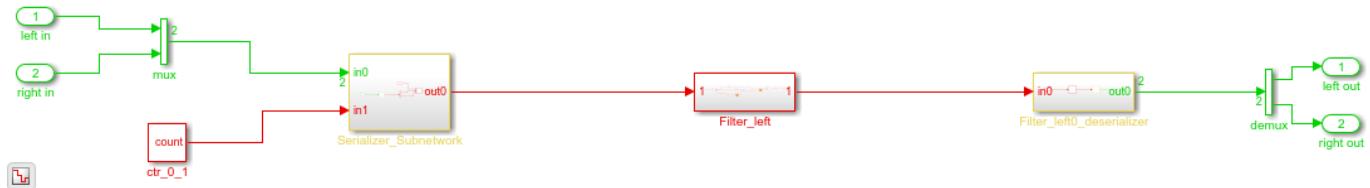
With resource sharing applied to the identical left and right channel atomic subsystems, only one filter is generated. To meet the Simulink timing requirements, the single filter is run at twice the clock rate as the original Simulink model, as is shown below. Since the resource sharing optimization creates a second clock rate, the user can use synchronous multiple clock mode to provide external clocks for both rates. In this configuration, multiple clock mode still only generates one clock. You see the message:

The design requires 2 times faster clock with respect to the base rate = 0.00012207.

```
bdclose_gm_hdlcoder_audiofiltering;
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
makehdl('hdlcoder_audiofiltering/Audio filter', 'Traceability', 'on');
open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');

### Generating HDL for 'hdlcoder_audiofiltering/Audio filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiofiltering')".
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit...
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
```

```
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0.00012207
### Working on hdlcoder_audiofiltering/Audio filter/Filter_left as hdlsrc\hdlcoder_audiofiltering
### Working on hdlcoder_audiofiltering/Audio filter as hdlsrc\hdlcoder_audiofiltering\Audio_filter
### Generating package file hdlsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Using Triggered Subsystems for HDL Code Generation

In this section...

- “Best Practices” on page 23-16
- “Using the Signal Builder Block” on page 23-16
- “Using Trigger As Clock” on page 23-16
- “Requirements” on page 23-17
- “Specify Trigger As Clock” on page 23-17
- “Limitations” on page 23-17

The Triggered Subsystem block is a Subsystem block that executes each time the control signal has a trigger value. To learn more about the block, see Triggered Subsystem.

Best Practices

When using triggered subsystems in models targeted for HDL code generation, consider the following:

- For synthesis results to match Simulink results, drive the trigger port with registered logic (with a synchronous clock) on the FPGA.
- It is good practice to put unit delays on Triggered Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
- The use of triggered subsystems can affect synthesis results in the following ways:
 - In some cases, the system clock speed can drop by a small percentage.
 - Generated code uses more resources, scaling with the number of triggered subsystem instances and the number of output ports per subsystem.

Using the Signal Builder Block

When you connect outputs from a Signal Builder block to a triggered subsystem, you might need to use a Rate Transition block. To run all triggered subsystem ports at the same rate:

- If the trigger source is a Signal Builder block, but the other triggered subsystem inputs come from other sources, insert a Rate Transition block into the signal path before the trigger input.
- If all inputs (including the trigger) come from a Signal Builder block, they have the same rate, so special action is not required.

Using Trigger As Clock

Using the trigger as clock in triggered subsystems enables you to partition your design into different clock regions in the generated code. Make sure that the **Clock edge** setting in the Configuration Parameters dialog box matches the **Trigger type** of the Trigger block inside the triggered subsystem.

For example, you can model:

- A design with clocks that run at the same rate, but out of phase.

- Clock regions driven by an external or internal clock divider.
- Clock regions driven by clocks whose rates are not integer multiples of each other.
- Internally generated clocks.
- Clock gating for low-power design.

Note Using the trigger as clock for triggered subsystems can result in timing mismatches of one cycle during testbench simulation.

Requirements

When you use the trigger as clock in triggered subsystems, each triggered subsystem input or output data signal must have delays immediately outside and immediately inside the subsystem. These delays act as a synchronization interface between the regions running at different rates.

Specify Trigger As Clock

- In **HDL Code Generation > Global Settings > Optimization** tab, select **Use trigger signal as clock**.
- Set the **TriggerAsClock** property using `makehdl` or `hdlset_param`. For example, to generate HDL code that uses the trigger signal as clock for triggered subsystems in a DUT subsystem, `myDUT`, in a model, `myModel`, enter:

```
makehdl ('myModel/myDUT', 'TriggerAsClock', 'on')
```

Limitations

HDL Coder supports HDL code generation for triggered subsystems that meet the following conditions:

- The triggered subsystem is not the DUT.
- The subsystem is not *both* triggered *and* enabled.
- The trigger signal is a scalar.
- Outputs of the triggered subsystem have an initial value of 0.
- All inputs and outputs of the triggered subsystem (including the trigger signal) run at the same rate.
- The **Show output port** parameter of the Trigger block is set to **Off**.
- The **Latch input by delaying outside signal** check box is not selected on the Inport block inside the Triggered Subsystem.
- If the DUT contains the following blocks, **RAMArchitecture** is set to **WithClockEnable**:
 - Dual Port RAM
 - Simple Dual Port RAM
 - Single Port RAM
- The triggered subsystem does not contain the following blocks:
 - Discrete-Time Integrator

- CIC Decimation
- CIC Interpolation
- FIR Decimation
- FIR Interpolation
- Downsample
- Upsample
- HDL Cosimulation blocks for HDL Verifier
- Rate Transition
- Pixel Stream FIFO (Vision HDL Toolbox)
- PN Sequence Generator, if the **Use trigger signal as clock** option is selected.

Generate Multicycle Path Information Files

In this section...

- “Overview” on page 23-19
- “Format and Content of a Multicycle Path Information File” on page 23-20
- “File Naming and Location Conventions” on page 23-23
- “Generating Multicycle Path Information Files Using the GUI” on page 23-23
- “Generating Multicycle Path Information Files Using the Command Line” on page 23-23
- “Limitations” on page 23-24

Overview

HDL Coder implements multirate systems in HDL by generating a master clock running at the model's base rate, and generating substrate timing signals from the master clock (see also “Code Generation from Multirate Models” on page 23-2). The propagation time between two substrate registers can be more than one cycle of the master clock. A multicycle path is a path between two such registers.

When synthesizing HDL code, it is often useful to provide an analysis of multicycle register-to-register paths to the synthesis tool. If the synthesis tool can identify multicycle paths, you may be able to:

- Realize higher clock rates from your multirate design.
- Reduce the area of your design.
- Reduce the execution time of the synthesis tool.

Using the **Generate multicycle path information** option (or the equivalent `MulticyclePathInfo` property for `makehdl`) you can instruct the coder to analyze multicycle paths in the generated code, and generate a multicycle path information file.

A multicycle path information file is a text file that describes one or more multicycle path constraints. A multicycle path constraint is a timing exception – it relaxes the default constraints on the system timing by allowing signals on a given path to have a longer propagation time. When using multiple clock mode, the file also contains clock definitions.

Typically a synthesis tool gives every signal a time budget of exactly 1 clock cycle to propagate from a source register to a destination register. A timing exception defines a *path multiplier*, N , that informs the synthesis tool that a signal has N clock cycles ($N > 1$) to propagate from the source to destination register. The path multiplier expresses some number of cycles of a *relative clock* at either the source or destination register. Where a timing exception is defined for a path, the synthesis tool has more flexibility in meeting the timing requirements for that path and for the system as a whole.

The generated multicycle path information file does not follow the native constraint file format of a particular synthesis tool. The file contains the multicycle path information required by popular synthesis tools. You can manually convert this information to multicycle path constraints in the format required by your synthesis tool, or write a script or tool to perform the conversion. The next section describes the format of a multicycle path constraint file in detail.

Format and Content of a Multicycle Path Information File

The following listing shows a simple multicycle path information file.

```
%%%%%%%%%%%%%%
% Constraints Report
%   Module: Sbs
%   Model: mSbs.mdl
%
%   File Name: hdlsrc/Sbs_constraints.txt
%   Created: 2009-04-10 09:50:10
%   Generated by MATLAB 7.9 and HDL Coder 1.6
%
%%%%%%%%%%%%%
%
%%%%%%%%%%%%%
% Multicycle Paths
%%%%%%%%%%%%%
FROM : Sbs.boolireg; T0 : Sbs.booloreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.boolireg_v<0>; T0 : Sbs.booloreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.doubireg; T0 : Sbs.douboreg; PATH_MULT : 2; RELATIVE_CLK : source,
      Sbs.clk;
FROM : Sbs.doubireg_v<0>; T0 : Sbs.douboreg_v<0>; PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg(7:0); T0 : Sbs.intoreg(7:0); PATH_MULT : 2;
      RELATIVE_CLK : source, Sbs.clk;
FROM : Sbs.intireg_v<0>(7:0); T0 : Sbs.intoreg_v<0>(7:0); PATH_MULT : 2
      RELATIVE_CLK : source, Sbs.clk;
```

The first section of the file is a header that identifies the source model and gives other information about how HDL Coder generated the file. This section terminates with the following comment lines:

```
%%%%%%%%%%%%%%
% Multicycle Paths
%%%%%%%%%%%%%
```

Note For a single-rate model or a model without multicycle paths, the coder generates only the header section of the file.

The main body of the file follows. This section contains a flat table, each row of which defines a multicycle path constraint.

Each constraint consists of four fields. The format of each field is one of the following:

- KEYWORD : field;
- KEYWORD : subfield₁,... subfield_N;

The keyword identifies the type of information contained in the field. The keyword string in each field terminates with a space followed by a colon.

The delimiter between fields is the semicolon. Within a field, the delimiter between subfields is the comma.

The following table defines the fields of a multicycle path constraint, in left-to-right order.

Keyword : field (or subfields)	Field Description
<code>FROM : src_reg_path;</code>	The source (or FROM) register of a multicycle path in the system. The value of <code>src_reg_path</code> is the HDL path of the source register's output signal. See also "Register Path Syntax for FROM : and TO : Fields" on page 23-21 .
<code>TO : dst_reg_path;</code>	The destination (or TO) register of a multicycle path in the system. The FROM register drives the TO register in the HDL code. The value of <code>dst_reg_path</code> is the HDL path of the destination register's output signal. See also "Register Path Syntax for FROM : and TO : Fields" on page 23-21.
<code>PATH_MULT : N;</code>	<p>The path multiplier defines the number of clock cycles that a signal has to propagate from the source to destination register. The RELATIVE_CLK field describes the clock associated with the path multiplier (the relative clock for the path).</p> <p>The path multiplier value <i>N</i> indicates that the signal has <i>N</i> clock cycles of its relative clock to propagate from source to destination register.</p> <p>The coder does not report register-to-register paths where <i>N</i> = 1, because this is the default path multiplier.</p>
<code>RELATIVE_CLK : relclock, sysclock;</code>	<p>The RELATIVE_CLK field contains two comma-delimited subfields. Each subfield expresses the location of the relative clock in a different form, for the use of different synthesis tools. The subfields are:</p> <ul style="list-style-type: none"> • <code>relclock</code>: Since HDL Coder currently generates only single-clock systems, this subfield takes the value <code>source</code>. In a multi-clock system, the relative clock associated with a multicycle path could be either the source or destination register of the path, and this subfield could take on either of the values <code>source</code> or <code>destination</code>. This usage is reserved for future release of the coder. • <code>sysclock</code>: This subfield is intended for use with synthesis tools that require the actual propagation time for a multicycle path. <code>sysclock</code> provides the path to the system's top-level clock (e.g., <code>Sbs.clk</code>) You can use the period of this clock and the path multiplier to calculate the propagation time for a given path.

Register Path Syntax for FROM : and TO : Fields

The FROM : and TO: fields of a multipath constraint provide the path to a source or destination register and information about the signal data type, size, and other characteristics.

Fixed Point Signals

For fixed point signals, the register path has the form

`reg_path<ps> (hb:lb)`

where:

- `reg_path` is the HDL hierarchical path of the signal. The delimiter between hierarchical levels is the period, for example: `Sbs.u_H1.initreg`.
- `<ps>`: Part select (zero-origin integer index) for vector signals. Angle brackets `<>` delimit the part select field

- (*hb*:*lb*): Bit select field, indicated from high-order bit to low-order bit. The signal width (*hb*:*lb*) is the same as the defined width of the signal in the HDL code. This representation does not necessarily imply that the bits of the FROM : register are connected to the corresponding bits of the TO : register. The actual bit-to-bit connections are determined during synthesis.

Boolean and Double Signals

For boolean and double signals, the register path has the form

`reg_path<ps>`

where:

- `reg_path` is the HDL hierarchical path of the signal. The delimiter between hierarchical levels is the period (.), for example: `Sbs.u_H1.intreg`.
- `<ps>`: Part select (zero-origin integer index) for vector signals. Angle brackets <> delimit the part select field

For boolean and double signals, no bit select field is present.

Note The format does not distinguish between boolean and double signals.

Examples

The following table gives several examples of register-to-register paths as represented in a multicycle path information file.

Path	Description
<code>FROM : Sbs.intireg(7:0); TO : Sbs.intoreg(7:0);</code>	Both signals are fixed point and eight bits wide.
<code>FROM : Sbs.intireg; TO : Sbs.intoreg;</code>	Both signals are either boolean or double.
<code>FROM : Sbs.intireg<0>(7:0); TO : Sbs.intoreg<1>(7:0);</code>	The FROM signal is the first element of a vector. The TO signal is the second element of a vector. Both signals are fixed point and eight bits wide.
<code>FROM : Sbs.u_H1.intireg(7:0); TO : Sbs.intoreg(7:0);</code>	The signal <code>intireg</code> is defined in the module <code>H1</code> , and <code>H1</code> is inside the module <code>Sbs</code> . <code>u_H1</code> is the instance name of <code>H1</code> in <code>Sbs</code> . Both signals are fixed point and eight bits wide.

Ordering of Multicycle Path Constraints

For a given model or subsystem, the ordering of multicycle path constraints within a multicycle path information file may vary depending on whether the target language is VHDL or Verilog, and on other factors. The ordering of constraints may also change in future versions of the coder. When you design scripts or other tools that process multicycle path information file, do not build in any assumptions about the ordering of multicycle path constraints within a file.

Clock Definitions

When you use multiple clock mode, the multicycle path information file also contains a "Clock Definitions" section, as shown in the following listing. This section is located after the header and before the "Multicycle Paths" section.

```
%%%%%%%%%%%%%
% Clock Definitions
%%%%%%%%%%%%%
CLOCK: Sbs.clk PERIOD: 0.05
CLOCK: Sbs.clk_1_2 BASE_CLOCK: Sbs.clk MULTIPLIER: 2 PERIOD: 0.1
```

The following table defines the fields for the clock definitions.

Keyword : field (or subfields)	Field Description
CLOCK: <code>clock_name</code>	Each clock in the design has a CLOCK definition line.
PERIOD: <code>float_value</code>	The Simulink rate (floating point value) associated with this CLOCK.
BASE_CLOCK: <code>base_clock_name</code>	Names the master clock. This field does not appear on the master clock.
MULTIPLIER: <code>int_value</code>	Gives the ratio of the period of this clock to the master clock. This field does not appear on the master clock.

File Naming and Location Conventions

The file name for the multicycle path information file derives from the name of the DUT and the postfix string '`_constraints`', as follows:

DUTname_constraints.txt

For example, if the DUT name is `symmetric_fir`, the name of the multicycle path information file is `symmetric_fir_constraints.txt`.

HDL Coder writes the multicycle path information file to the target .

Generating Multicycle Path Information Files Using the GUI

To enable generation of multicycle path information files, select **Register-to-register path info** in the **Multicycle Path Constraints** section of the **HDL Code Generation > Target and Optimizations** pane of the Configuration Parameters dialog box.

When you select this check box and generate code for your model, the code generator creates a multicycle path information file.

Generating Multicycle Path Information Files Using the Command Line

To generate a multicycle path information file from the command line, pass in the property/value pair '`MulticyclePathInfo`', '`on`' to `makehdl`, as in the following example.

```
>> dut = 'hdlfirtdecim_multicycle/Subsystem';
>> makehdl(dut, 'MulticyclePathInfo','on');
### Generating HDL for 'hdlfirtdecim_multicycle/Subsystem'
### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 1 message.

### MESSAGE: For the block 'hdlfirtdecim_multicycle/Subsystem/downsamp0'
The initial condition may not be used when the sample offset is 0.

### Begin VHDL Code Generation
### Working on Subsystem_tc as hdlsrc\Subsystem_tc.vhd
### Working on hdlfirtdecim_multicycle/Subsystem as hdlsrc\Subsystem.vhd
### Generating package file hdlsrc\Subsystem_pkg.vhd
```

```
### Finishing multicycle path connectivity analysis.  
### Writing multicycle path information in hdlsrc\Subsystem_constraints.txt  
### HDL Code Generation Complete.
```

Limitations

Unsupported Blocks and Implementations

The following table lists block implementations (and associated Simulink blocks) that will not contribute to multicycle path constraints information.

Implementation	Block(s)
SumCascadeHDLEmission	Add, Subtract, Sum, Sum of Elements
ProductCascadeHDLEmission	Product, Product of Elements
MinMaxCascadeHDLEmission	MinMax, Maximum, Minimum
ModelReferenceHDLInstantiation	Model
SubsystemBlackBoxHDLInstantiation	Subsystem
RamBlockDualHDLInstantiation	Dual Port RAM
RamBlockSimpDualHDLInstantiation	Simple Dual Port RAM
RamBlockSingleHDLInstantiation	Single Port RAM

Limitations on MATLAB Function Blocks and Stateflow Charts

Loop-Carried Dependencies

HDL Coder does not generate constraints for MATLAB Function blocks or Stateflow charts that contain a `for` loop with a loop-carried dependency.

Indexing Vector or Matrix Variables

In order to generate constraints for a vector or matrix index expression, the index expression must be one of the following:

- A constant
- A `for` loop induction variable

For example, in the following example of code for a MATLAB Function block, the index expression `reg(i)` does not generate constraints.

```
function y = fcn(u)  
 %#codegen  
  
N=length(u);  
persistent reg;  
if isempty(reg)  
    reg = zeros(1,N);  
end  
  
y = reg;  
  
for i = 1:N-1  
    reg(i) = u(i) + reg(i+1);  
end  
reg(N) = u(N);
```

File Generation Time

Tip Generation of constraint files for large models can be slow.

Meet Timing Requirements Using Enable-Based Multicycle Path Constraints

In this section...

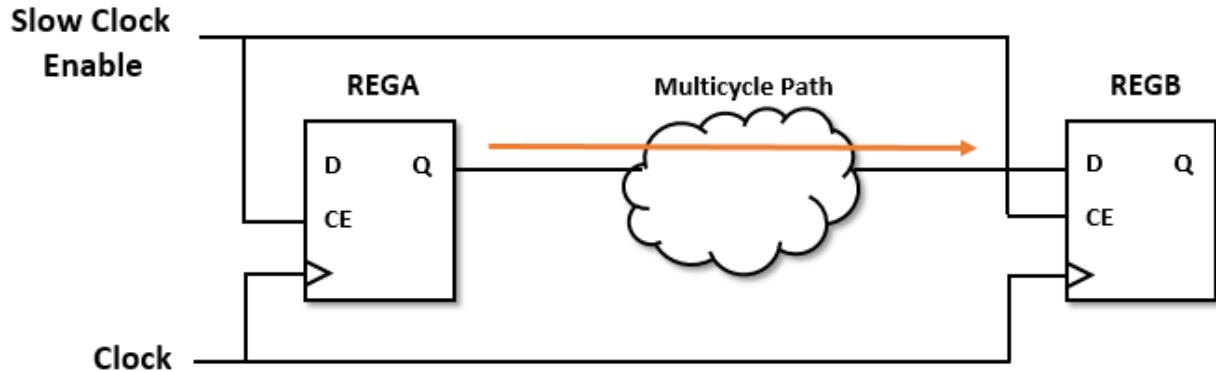
- "How Enable-Based Multicycle Path Constraints Work" on page 23-26
- "Specify Enable-Based Constraints" on page 23-27
- "Benefits of Using Enable-Based Constraints" on page 23-28
- "Modeling Guidelines" on page 23-29
- "Multicycle Path Constraints for Various Synthesis Tools" on page 23-29
- "Caveats and Limitations" on page 23-30

If your Simulink model contains multiple sample rates or uses speed and area optimizations that insert pipeline registers, your design can have multicycle paths. Multicycle paths are data paths between two registers that operate at a sample rate slower than the FPGA clock rate and therefore take multiple clock cycles to complete their execution. To synchronize the clock rate to the sample rates of various paths in your design, you can use a single clock mode or a multiple clock mode. By default, HDL Coder uses a single clock mode that generates a single master clock at the fastest sample rate and creates a timing controller entity to control the clock rate to the multicycle paths. The timing controller generates a set of clock enables with the required rate and phase information to control the sequential elements such as Delay blocks that operate at different sample rates.

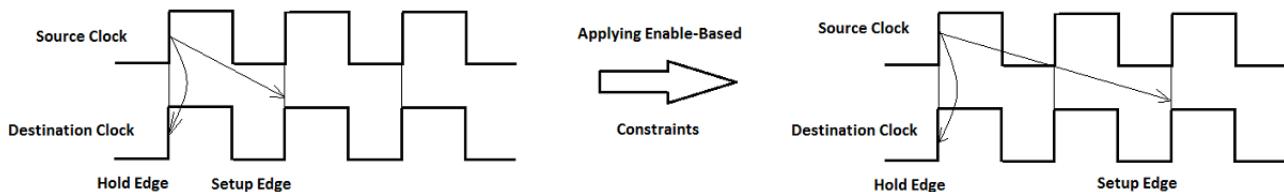
When you synthesize the generated HDL code, synthesis tools can fail to meet the timing requirements of multicycle paths. The timing failure occurs because synthesis tools cannot infer the various sample rates in your design from the generated HDL code. The synthesis tools assume that the registers in your design run at the master clock rate and require data to travel between the registers within one clock cycle. However, the multicycle paths are not required to complete their execution within one clock cycle and therefore cannot meet the timing requirements. To meet the timing requirements, specify generation of enable-based multicycle path constraints.

How Enable-Based Multicycle Path Constraints Work

Synthesis tools require that data propagates from a source register to a destination register within one clock cycle. Multicycle path constraints relax this timing requirement by allowing multiple clock cycles for data to propagate between the registers. The code generator uses the timing controller enable signals to create enable-based register groups, with registers in each group driven by the same clock enable. When you apply the enable-based constraints and generate HDL code, the code generator outputs a constraints file with the naming convention `dutname_constraints`. The file defines the timing requirements of multicycle paths and contains information about the setup and hold constraints that needs to be met.



This figure shows a multicycle path that takes a certain number of clock cycles, say N , for the data to propagate from REGA to REGB. By default, the synthesis tools define the setup edge at the next active clock edge and the hold edge at the same active clock edge with respect to the destination clock signal. For a multicycle path that takes N clock cycles, the constraints redefine the setup and hold edge to allow for the longer data propagation time.



For example, consider a multicycle path takes two clock cycles for data top propagate from the source to the destination register. This waveform shows how applying enable-based constraints redefines the setup and hold edges. This code snippet shows this setup and hold requirement in the constraints file that gets generated when you enable multicycle path constraints.

```
set_multicycle_path 2 -setup -from $REGA -to $REGB
set_multicycle_path 1 -hold   -from $REGA -to $REGB
```

Specify Enable-Based Constraints

Before you generate the enable-based constraints, you must:

- Preserve the multicycle paths in your design. Before you enable generation of multicycle path constraints, make sure that you disable optimizations such as clock rate pipelining and adaptive pipelining in those regions where you want to apply multicycle path constraints.

- Make sure that the region that operates at a slower clock rate is bounded by timing controller based clock enable signals operating at zero phase.
- Specify the **Synthesis tool**. The format of the multicycle path constraints file that gets generated depends on the **Synthesis tool** that you specify. If you do not specify the synthesis tool and the **Generate EDA Scripts** check box is selected, HDL Coder does not generate multicycle path constraints.
- Use the single clock mode. In the **HDL Code Generation > Global Settings** pane, set **Clock Inputs** to **Single**.

You can specify generation of multicycle constraints in the Configuration Parameters dialog box, or in the HDL Workflow Advisor UI, or at the command line.

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Target and Optimizations** pane, select the **Enable based constraints** check box.
- In the HDL Workflow Advisor, on the **HDL Code Generation > Set Code Generation Options > Set Optimization Options** task, select the **Enable based constraints** check box.
- At the command line, use the `MulticyclePathConstraints` property with `hdlset_param` or `makehdl`.

Benefits of Using Enable-Based Constraints

If the synthesis tools identify the multicycle path constraints, you can:

- Realize higher clock rates and improve the timing of your design.
- Reduce the area footprint on the target FPGA device because multicycle path constraints do not introduce any pipeline registers.
- Reduce HDL code generation time because the code generator does not have to run many optimization settings.
- Reduce synthesis time since multicycle path constraints relax the timing requirements on the synthesis tool.
- Skip verification of your design after generating HDL code as the generated model with the constraints is identical to the original model.

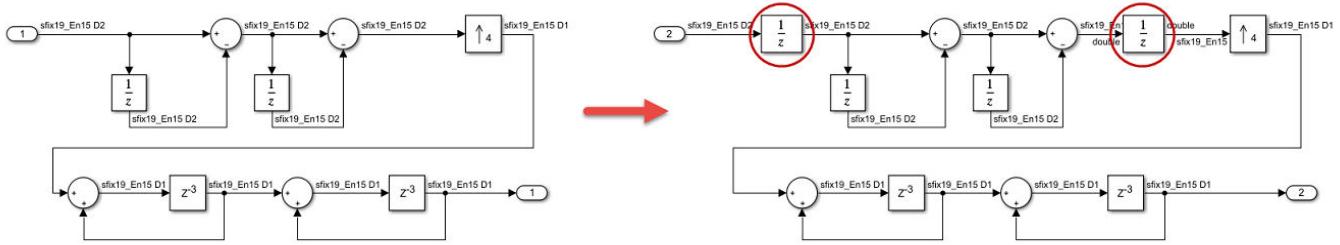
When you specify the multicycle path information to the synthesis tool, it is not recommended to use the **Register-to-register path info** setting in the **Target and Optimizations** pane. If you use this setting, the code generator outputs a text file that describes the multicycle path information in a format that is not native to a particular synthesis tool. You must convert this information to the multicycle path constraints format required by your synthesis tool.

When you use the enable-based constraints setting:

- The generated constraints are more robust to name changes in synthesis tools.
- HDL code generation is faster than when you use the **Register-to-register path info** setting.
- The **Target workflow** can be Generic ASIC/FPGA, FPGA Turnkey, IP Core Generation, and Simulink Real-Time FPGA I/O.
- The constraint file format is supported with Xilinx ISE, Xilinx Vivado, and Altera QUARTUS II.

Modeling Guidelines

When you specify generation of enable-based constraints, use these modeling patterns in your design. If your model contains slow-rate regions that are not bounded by registers, then add delays at the same slow rate to the input and output of the slow-rate regions. For example, if you enter `hdlcoder_clockdemo` at the command line in MATLAB, you see a multirate CIC Interpolation filter implemented in single clock mode. This figure shows how to bound the input and output of the slow-rate region annotated by the slow sample time D2 in the model with Unit Delay blocks so that the enable-based constraints can identify the slow-rate path.



Note You can use Rate Transition blocks to introduce the input and output registers but make sure that the registers are slow rate and have zero phase.

Multicycle Path Constraints for Various Synthesis Tools

Enable-based multicycle path constraints have various file formats that depend on the **Synthesis tool** that you specify.

Altera Quartus II

HDL Coder generates the constraints in the form of an SDC file. This code snippet shows the SDC file generated for Altera Quartus® II.

```
# Multicycle constraints for clock enable: DUT_tc.ul_d4_o0
set enbreg [get_registers *u_DUT_tc|phase_0]
set_multicycle_path 4 -to [get_fanouts $enbreg -through [get_pins -hier *|ena]] -end -setup
set_multicycle_path 3 -to [get_fanouts $enbreg -through [get_pins -hier *|ena]] -end -hold
```

Xilinx Vivado

HDL Coder generates the constraints in the form of an XDC file. This code snippet shows the XDC file generated for Xilinx Vivado.

```
# Multicycle constraints for clock enable: DUT_tc.ul_d4_o0
set enbregcell [get_cells -hier -filter {mcp_info=="DUT_tc.ul_d4_o0"}]
set enbregnet [get_nets -of_objects [get_pins -of_objects $enbregcell -filter {DIRECTION == OUT}]]
set reglist [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENABLE]]
set_multicycle_path 4 -setup -from $reglist -to $reglist -quiet
set_multicycle_path 3 -hold -from $reglist -to $reglist -quiet
```

The multicycle path constraints form enable-based register groups by querying the synthesis netlist for the ATTRIBUTE keyword. This code snippet shows this keyword in the synthesis netlist when you run any of the supported target workflows.

```
...
ATTRIBUTE mcp_info: string

ATTRIBUTE mcp_info OF phase_0 : SIGNAL IS "DUT_tc.ul_d4_o0";
...
```

The constraints file that is generated for Xilinx Vivado is more robust than pattern matching on module or signal names.

Xilinx ISE

HDL Coder generates the constraints in the form of a UCF file. This code snippet shows the UCF file generated for a model that has one slow-rate region controlled by a clock enable signal and has a target frequency of 300MHz. The snippet shows that the multicycle path constraints depend on the **Target Frequency** that you specify.

```
# Multicycle constraints for clock enable: DUT_tc.ul_d4_o0
NET "*u_DUT_tc/phase_0" TNM_NET = FFS "TN_u_DUT_tc_phase_0";
TIMESPEC "TS_u_DUT_tc_phase_0" = FROM "TN_u_DUT_tc_phase_0" TO "TN_u_DUT_tc_phase_0" TS_FPGA_CLK;
```

This code snippet shows the clock constraints that get generated when you run the **Generic ASIC/FPGA, FPGA Turnkey**, or the **Simulink Real-Time FPGA I/O** workflow with Xilinx ISE.

```
# Timing Specification Constraints

NET "clk" TNM_NET = "TN_clk";
TIMESPEC "TS_FPGA_CLK" = PERIOD "TN_clk" 300 MHz;
```

To use the multicycle path constraints when you generate HDL code by using the `makehdl` function, make sure that you add a `TS_FPGA_CLK` constraint to the UCF file.

Caveats and Limitations

- The multicycle path constraints file is not supported with the **FPGA-in-the-Loop** workflow.
- The **IP Core Generation** workflow does not generate a clock constraint and therefore does not support multicycle path constraints generation with Xilinx ISE.
- If the slow-rate region is not bounded by registers, multicycle path constraints requires you to add two Delay blocks at the slow rate, which increases the latency of your design.
- The code generator does not add constraints on paths between registers that have a nonzero phase value for the timing controller based enable signals. For the code generator to add constraints, use registers that derive from phase 0 clock enable signals, such as Delay blocks.
- The generated multicycle constraints can be less effective if you apply the constraints in regions that have optimizations such as clock-rate pipelining and adaptive pipelining enabled. With clock-rate pipelining, the registers operate at the faster clock rate and therefore may not retain the slow-rate registers in your design.
- HDL Coder does not generate multicycle path constraints for single-rate models.
- The code generator does not output the multicycle path constraints file if you use the multiple clock mode.

See Also

Related Examples

- “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 23-32

More About

- “Multicycle Path Constraints Parameters” on page 15-27
- “Clock-Rate Pipelining” on page 24-114
- “Adaptive Pipelining” on page 24-130
- “Generate Multicycle Path Information Files” on page 23-19

Use Multicycle Path Constraints to Meet Timing for Slow Paths

This example shows how to apply multicycle path constraints in your design to meet timing requirements. Using multicycle path constraints can save area and reduce synthesis run times. For more information, see the enable-based multicycle path constraints documentation.

Introduction

Algorithms modeled in Simulink for HDL code generation can have multiple sample rates. These multiple rates can be part of the Simulink model, or can be introduced with HDL Coder options such as oversampling. With oversampling specified, the generated HDL code will run on the FPGA at a faster clock rate. This faster rate allows additional optimizations to take effect.

When HDL Coder is configured to use a single clock, it generates a timing controller to control clocked elements, such as delays, at different sample rates with clock enables. These clock enables are synchronous to the single clock and toggle at rates that are multiple times slower than the base clock. The data paths between slow clocked element pairs are called multicycle paths, because they allow data to take multiple clock cycles to travel. However, synthesis tools cannot infer this acceptable delay directly from the HDL code. The tools assume that data changes every cycle and must travel from one register to the next within one clock cycle. Synthesis tools have to take more effort to meet the excessive timing requirement and therefore can fail timing. By declaring a set of data paths as multicycle paths and providing the actual timing of these paths to the downstream synthesis tool, HDL Coder can simplify and accelerate the process of meeting the desired timing constraints for a design.

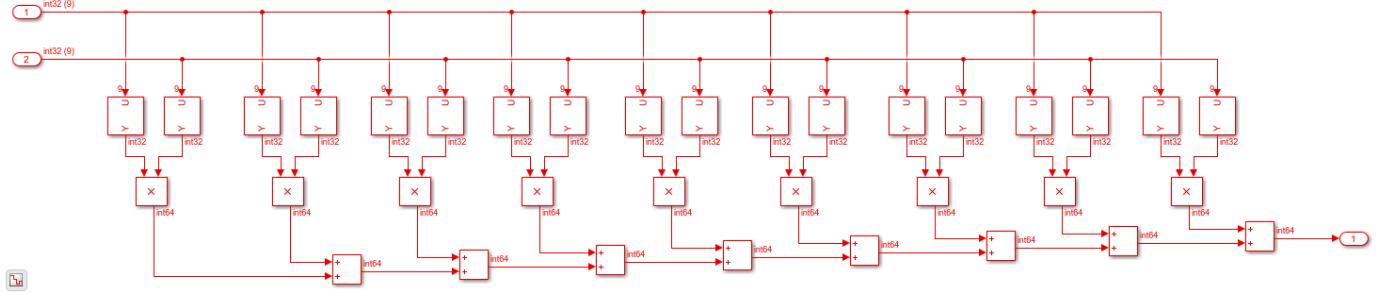
This example demonstrates how to generate multicycle path constraints in HDL Coder so that timing requirements may be specified, allowing efficient timing analysis in the synthesis tool.

Applying multicycle constraints using HDL Coder

In this example, we use Xilinx Vivado 2016.4 post place and routing static timing analysis results for a Virtex7 device (xc7v2000t, fgh1761, -1) to show the impact of enabled based multicycle path constraints. Other synthesis tools and devices have similar behavior. HDL Coder generates constraint files of XDC format for Xilinx Vivado, UCF format for Xilinx ISE, and SDC format for Altera Quartus II.

Consider the example, `hdlcoder_multi_cycle_path_constraints.slx`. The model contains a direct form FIR filter with an adder chain in the critical path. While the input data rate of this filter is specified to be 2MHz, we want this design to run as fast as possible so that it can be integrated with other IPs requiring high frequency. We start by choosing a 130MHz clock frequency without any timing optimization by setting 'Oversampling factor' to 65 and 'TargetFrequency' to 130.

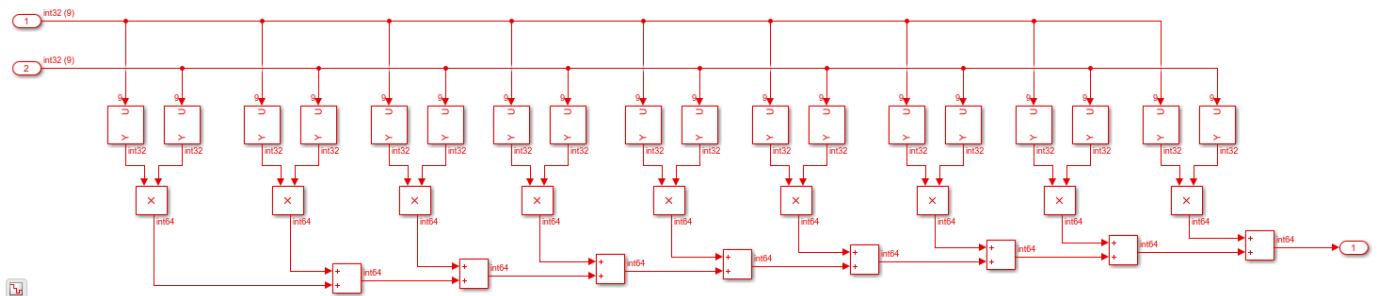
```
bdclose('all');
load_system('hdlcoder_multi_cycle_path_constraints');
open_system('hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'Oversampling', 65);
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'TargetFrequency', 130);
set_param('hdlcoder_multi_cycle_path_constraints', 'SimulationCommand', 'update');
```



Generate HDL with these settings and inspect the generated model. Notice that the generated model is actually identical to the original model.

```
makehdl('hdlcoder_multi_cycle_path_constraints/Subsystem');
load_system('gm_hdlcoder_multi_cycle_path_constraints');
set_param('gm_hdlcoder_multi_cycle_path_constraints', 'SimulationCommand', 'update');
open_system('gm_hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');

### Generating HDL for 'hdlcoder_multi_cycle_path_constraints/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multi_...
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_multi_cycle_path_constraints'.
### MESSAGE: The design requires 65 times faster clock with respect to the base rate = 2.
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product as hdl_prj\hdlsrc\hdlc...
### Working on Subsystem_tc as hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_tc...
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem as hdl_prj\hdlsrc\hdlcoder_multi...
### Generating package file hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_pkg.vh...
### Writing Vivado multicycle constraints XDC file <a href="matlab:edit('hdl_prj\hdlsrc\hdlcoder...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdlcoder_multi_cycle_path_constraints' complete with 0 errors, 0 warnings, and...
### HDL code generation complete.
```

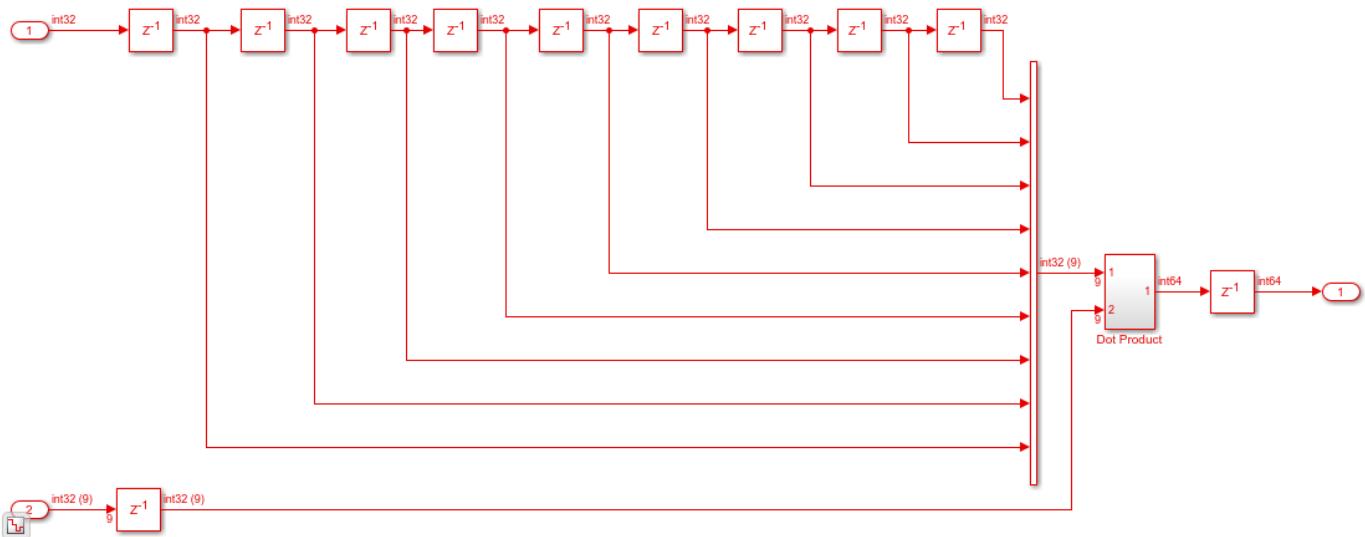


The generated HDL fails to meet the timing requirement for the clock at 130MHz. As shown in the timing report snippet below, the timing requirement is 7.692 ns (1/130MHz) and a negative slack indicates a timing violation of this requirement. Other synthesis tools' timing reports may look different, although the critical path from this example design will remain the same.

```
Max Delay Paths
-----
Slack (VIOLATED) : -1.979ns (required time - arrival time)
  Source: Delay6_out1_reg[8]/C
            (rising edge-triggered cell FDCE clocked by MWCLK {rise@0.000ns fall@3.846ns period=7.692ns})
  Destination: Delay1_out1_reg[61]/D
            (rising edge-triggered cell FDCE clocked by MWCLK {rise@0.000ns fall@3.846ns period=7.692ns})
Path Group: MWCLK
Path Type: Setup (Max at Slow Process Corner)
Requirement: 7.692ns (MWCLK rising@7.692ns - MWCLK rise@0.000ns)
Data Path Delay: 9.686ns (logic 7.297ns (75.333%) route 2.389ns (24.667%))
```

We are going to use multicycle path constraints to meet timing requirements. Check the original model.

```
open_system('hdlcoder_multi_cycle_path_constraints/Subsystem');
```



Notice that the Dot Product subsystem is surrounded by delays running at the desired 2MHz data rate. Due to the specified 65x oversampling, the design can tolerate multiple clock cycles for the data to propagate through it. HDL Coder requires multicycle regions to be surrounded by slow clocked elements, such as these delays, so that the constraints can define the paths among them as multicycle paths. Turn MulticyclePathConstraints on and HDL Coder will generate an additional file.

```
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'MulticyclePathConstraints', 'on');
```

We can even increase the target frequency to 300MHz.

```
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'Oversampling', 150);
hdlset_param('hdlcoder_multi_cycle_path_constraints', 'TargetFrequency', 300);
```

Generate HDL and multicycle path constraints.

```
makehdl('hdlcoder_multi_cycle_path_constraints/Subsystem');
```

```
### Generating HDL for 'hdlcoder_multi_cycle_path_constraints/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_multi_
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_multi_cycle_path_constraints'.
### MESSAGE: The design requires 150 times faster clock with respect to the base rate = 2.
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product as hdl_prj\hdlsrc\hdlc...
```

```
### Working on Subsystem_tc as hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_tc
### Working on hdlcoder_multi_cycle_path_constraints/Subsystem as hdl_prj\hdlsrc\hdlcoder_multi_
### Generating package file hdl_prj\hdlsrc\hdlcoder_multi_cycle_path_constraints\Subsystem_pkg.vl
### Writing Vivado multicycle constraints XDC file <a href="matlab:edit('hdl_prj\hdlsrc\hdlcoder_
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### HDL check for 'hdlcoder_multi_cycle_path_constraints' complete with 0 errors, 0 warnings, and
### HDL code generation complete.
```

Inspect the generated constraint XDC file.

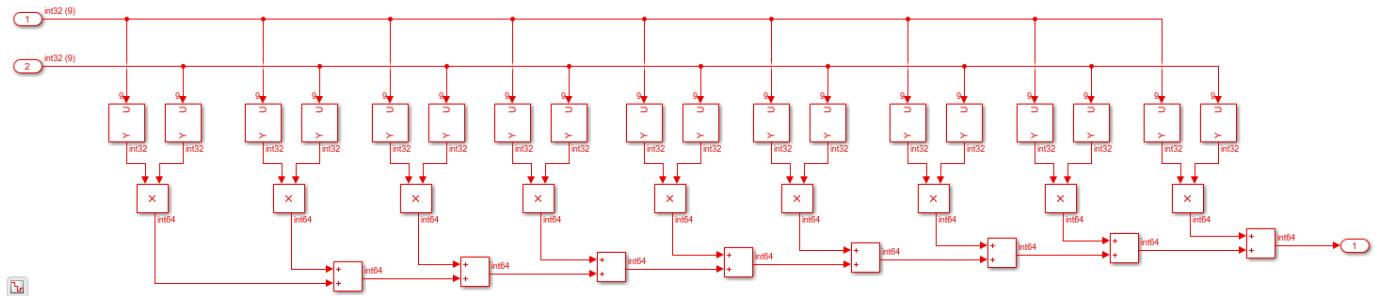
```
dbtype('hdl_prj/hdlsrc/hdlcoder_multi_cycle_path_constraints/Subsystem_constraints.xdc');
```

```
1 # Multicycle constraints for clock enable: Subsystem_tc.u1_d150_o0
2 set enbregcell [get_cells -hier -filter {mcp_info=="Subsystem_tc.u1_d150_o0"}]
3 set enbregnet [get_nets -of_objects [get_pins -of_objects $enbregcell -filter {DIRECTION == "OUT"}]]
4 set reglist [get_cells -of [filter [all_fanout -flat -endpoints_only $enbregnet] IS_ENABLE=1]]
5 set_multicycle_path 150 -setup -from $reglist -to $reglist -quiet
6 set_multicycle_path 149 -hold -from $reglist -to $reglist -quiet
7
```

These constraints first find flip flops that are driven by the 2MHz clock enable signal. Then, they define the paths among these flip flops multicycle paths to allow up to 150 cycles for data to propagate.

Inspect the generated model.

```
open_system('gm_hdlcoder_multi_cycle_path_constraints/Subsystem/Dot Product');
set_param('gm_hdlcoder_multi_cycle_path_constraints', 'SimulationCommand', 'update');
```



The generated model and HDL code are identical to the previous results, because generating enabled based multicycle path constraints does not alter the HDL architecture. The design was previously unable to meet the desired timing at 130MHz. However, when you specify multicycle path constraints, this design can run at 300MHz on the FPGA.

Max Delay Paths

Slack (MET) :	0.612ns (required time - arrival time)
Source:	u_Subsystem_tc/phase_0_reg/C (rising edge-triggered cell FDCE clocked by MWCLK {rise@0.000ns fall@1.666ns period=3.333ns})
Destination:	Delay6_out1_reg[4]/CE (rising edge-triggered cell FDCE clocked by MWCLK {rise@0.000ns fall@1.666ns period=3.333ns})
Path Group:	MWCLK
Path Type:	Setup (Max at Slow Process Corner)
Requirement:	3.333ns (MWCLK rise@3.333ns - MWCLK rise@0.000ns)
Data Path Delay:	2.408ns (logic 0.399ns (16.570%) route 2.009ns (83.430%))

Additional information about multicycle path constraints:

Multicycle path constraints are required for synthesis tools to understand timing requirements. This information is extracted from the Simulink model since it cannot be inferred from the generated HDL code. Multicycle path constraints identify paths between clocked elements driven by the same clock enable. It can fail to meet timing requirements in certain cases. For example, a data path is not recognized as a multicycle path, if it is not gated with both input and output delays or is between two delays of different rates. Therefore, if you want to use multicycle path constraints for certain parts in your design, it is important to retain the multicycle paths in that region from being altered by optimizations introducing pipelines, such as input and output pipelining, clock rate pipelining, adaptive pipelining resource sharing, streaming, pipelined math operations, e.g. Newton-Raphson method for sqrt or recip, Cordic algorithm for trigonometric functions, and floating-point IP mapping.

```
% LocalWords: Vivado xc ffg XDC UCF SDC slx IPs timingreport prj xdc Raphson  
% LocalWords: recip ug
```

Optimization

- “Speed and Area Optimizations in HDL Coder” on page 24-2
- “Automatic Iterative Optimization” on page 24-7
- “Generated Model and Validation Model” on page 24-10
- “Locate Numeric Differences After Speed Optimization” on page 24-13
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-17
- “Optimization with Constrained Overclocking” on page 24-22
- “Resolve Numerical Mismatch with Delay Balancing” on page 24-24
- “Streaming” on page 24-29
- “Resource Sharing” on page 24-32
- “Streaming: Area Optimization” on page 24-36
- “Resource Sharing For Area Optimization” on page 24-40
- “Single-rate Resource Sharing Architecture” on page 24-50
- “Improve Resource Sharing with Design Modifications” on page 24-53
- “Improve Resource Sharing with Clone Detection and Replacement” on page 24-58
- “Delay Balancing” on page 24-63
- “Delay Balancing and Validation Model Workflow In HDL Coder™” on page 24-68
- “Control the Scope of Delay Balancing” on page 24-75
- “Delay Balancing on multi-rate designs” on page 24-82
- “Find Feedback Loops” on page 24-90
- “Hierarchy Flattening” on page 24-92
- “RAM Mapping for Simulink Models” on page 24-95
- “RAM Mapping With the MATLAB Function Block” on page 24-96
- “Distributed Pipelining” on page 24-101
- “Hierarchical Distributed Pipelining” on page 24-105
- “Distributed Pipelining: Speed Optimization” on page 24-108
- “Constrained Output Pipelining” on page 24-112
- “Clock-Rate Pipelining” on page 24-114
- “Clock Rate Pipelining” on page 24-118
- “Adaptive Pipelining” on page 24-130
- “Critical Path Estimation Without Running Synthesis” on page 24-137
- “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-146
- “Subsystem Optimizations for Filters” on page 24-156
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166
- “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188

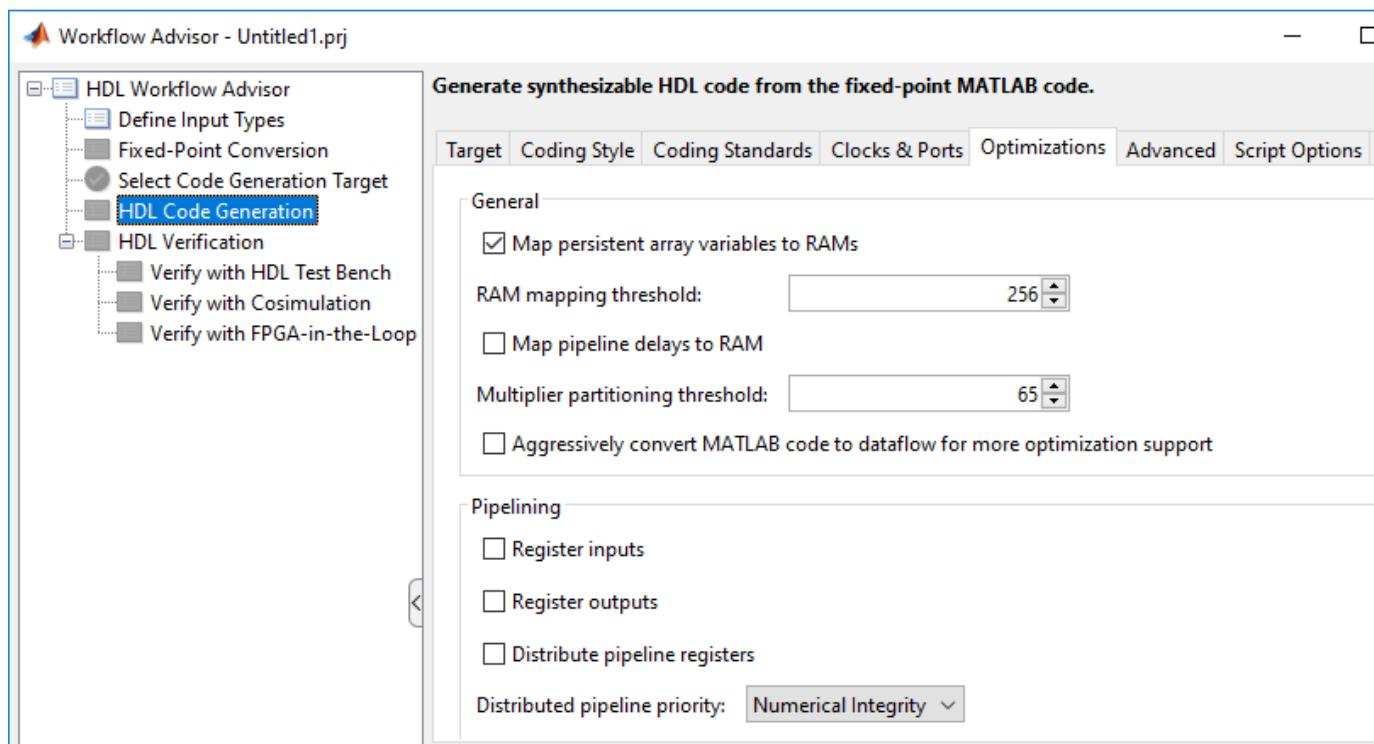
Speed and Area Optimizations in HDL Coder

Use area and speed optimizations in HDL Coder to save resources and improve the timing of your design on the target FPGA device. The optimizations do not change the functional behavior of your algorithm but can optimize certain resources in your design, introduce latency, or cause difference in sample rates.

You can initially generate HDL code and synthesize your design on your FPGA platform without enabling optimizations. If the design does not meet the timing requirements, you can enable the optimizations and rerun the workflow until your design meets the area and speed requirements. See “Basic HDL Code Generation Workflow”.

Optimizations in MATLAB HDL Code Generation

To enable optimizations on your MATLAB code, open the Workflow Advisor from MATLAB. In the Advisor, on the **HDL Code Generation** task, enable the settings in the **Optimization** tab.

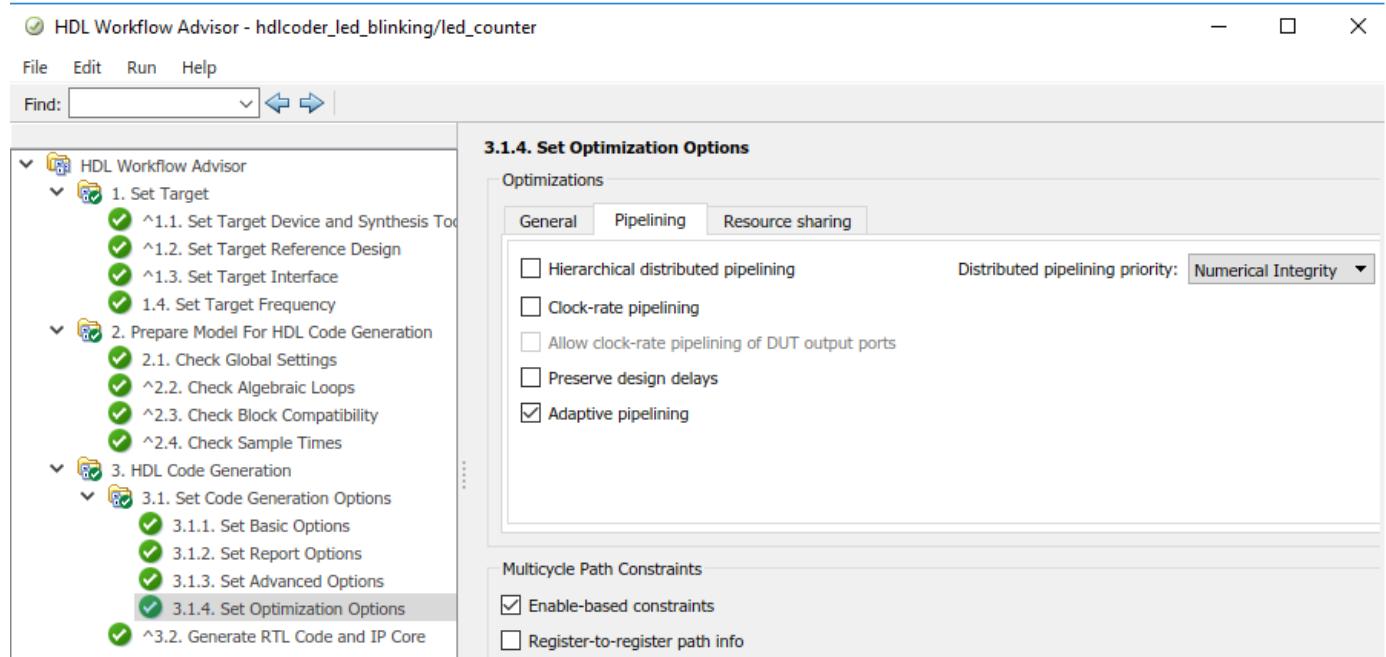


Optimizations in Simulink HDL Code Generation

You can enable optimizations at the model level and at the block level. Specify model-level optimizations:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Optimization** pane. See “HDL Code Generation Pane: Optimization” on page 12-7.

- At the command line by using the `makehdl` or `hdlset_param` function to set the property value.
- In the Simulink HDL Workflow Advisor, on the **Set Code Generation Options > Set Optimization Options** task.



Subsystems in your model inherit the model-level optimization settings. You can change the subsystem level settings in the HDL Block Properties dialog box for the subsystems or by using the `hdlset_param` function. You can also specify certain additional settings for certain blocks in your model such as adding pipelines at the input and output. This table illustrates various optimizations that are available at the block level and model level.

Optimization	Model Level?	Subsystem Level?	Comments
Delay balancing	Yes	Yes	-
RAM mapping	Yes	No	-
Adaptive pipelining	Yes	Yes	-
Clock rate pipelining	Yes	Yes	-
Distributed pipelining	Yes	Yes	At the model level, you use hierarchical distributed pipelining. To apply the optimization across subsystem hierarchies, enable distributed pipelining at each subsystem level.

Optimization	Model Level?	Subsystem Level?	Comments
Resource sharing	Yes	Yes	At the model level, you specify the type of resources you want to share such as adders and multipliers. At the block level, you specify the SharingFactor .
Streaming	No	Yes	-

To see the effect of the optimizations:

- You can generate an optimization report with the HDL code. To learn how to enable this report, see “Create and Use Code Generation Reports” on page 25-2.
- Open the generated model or generate the validation model. The generated model is a behavioral model of the HDL code that shows the effect of block implementations and optimizations that you enabled. To verify the numerics of the generated model with the original model, you can generate a validation model. See “Generated Model and Validation Model” on page 24-10.

Tip To effectively use optimizations, change the sample time setting for Constant blocks from Inf to -1.

General Optimizations

Your model can have design delays and pipeline delays. Design delays are delays that you manually add to your model. Pipeline delays are delays that are introduced by pipelining settings specified on the blocks, block implementations such as Newton-Raphson method, native floating-point operators, or speed optimizations. You see these delays in the generated HDL code, generated model, and validation model.

General optimization parameters includes:

- RAM mapping: use RAM mapping parameters to map large delays, persistent variables in MATLAB code, and pipeline delays to RAM based on a threshold bit width. See also “RAM Mapping for MATLAB Code” on page 8-2 and “RAM Mapping Parameters” on page 15-7.
- Delay balancing: Enabled by default, this optimization balances pipeline delays by inserting matching delays in parallel paths. The optimization matches numerics of the generated model with the original model. You see the effect of this optimization in the **Delay Balancing** section of the optimization report. See “Delay Balancing” on page 24-63.

Speed Optimizations

Speed optimizations improve the timing of your design on the target FPGA by optimizing the critical path. To identify the critical path, you can run the Generic ASIC/FPGA workflow for your FPGA device and then annotate the critical path or use the timing reports.

To identify the critical path more quickly and speed up the iterative process of finding and optimizing the critical path, use critical path estimation. You do not have to run synthesis or generate HDL code. Critical path estimation uses static timing analysis with timing data from target-specific timing databases. You see the effect of this optimization in the **Critical Path Estimation** section of the optimization report. See “Critical Path Estimation Without Running Synthesis” on page 24-137.

Speed optimizations include:

- Clock rate pipelining: A Simulink optimization that is enabled by default, and runs pipeline registers at a faster clock rate when you specify an **Oversampling factor** greater than one. Use clock-rate pipelining with hierarchy flattening to remove hierarchical boundaries in a subsystem, thereby improving retiming. See “Clock-Rate Pipelining” on page 24-114.
- Distributed pipelining: An optimization that retimes registers that are existing delays, or specified by using **InputPipeline** and **OutputPipeline** block settings. To preserve existing delays, enable the “Preserve design delays” on page 15-14 setting. Enable hierarchical distributed pipelining on the model and distributed pipelining on the subsystems for retiming registers across hierarchies. You see the effect of this optimization in the **Distributed Pipelining** section of the optimization report. See “Distributed Pipelining” on page 24-101 and “Hierarchical Distributed Pipelining” on page 24-105.
- Adaptive pipelining: A Simulink optimization that inserts pipeline registers at input or output or both ports of certain blocks to create patterns that efficiently map blocks to DSP units on the target FPGA device. The optimization considers the target device, target frequency, multiplier word lengths, and the HDL Block Property settings. You see the effect of this optimization in the **Adaptive Pipelining** section of the optimization report. See “Adaptive Pipelining” on page 24-130.
- Loop Unrolling: A MATLAB optimization that unrolls a loop by instantiating multiple instances of the loop body in the generated code. You can also partially unroll a loop. See “Optimize MATLAB Loops” on page 8-20

Area Optimizations

Area optimizations reduce resource usage of your design. Optimizing your design for area can reduce the speed at which your design runs on the FPGA.

Area optimizations include:

- Resource Sharing: An optimization that identifies multiple functionally equivalent resources and replaces them with a single resource. At the model level, you specify resources you want to share such as adders and multipliers. At the subsystem level, you specify a **SharingFactor** depending on the number of shareable resources in your design. By using the optimization with clock-rate pipelining, you can specify how to overclock the shared resources. See “Resource Sharing” on page 24-32
- Streaming: A Simulink optimization that splits a vector data path into multiple smaller vector data paths based on the **StreamingFactor** that you specify on the subsystems, thereby reducing hardware resource consumption. See “Streaming” on page 24-29.
- Loop Streaming: A MATLAB optimization that streams a loop by instantiating the loop body once and using that instance for each loop iteration. The code generator oversamples the loop body instance to keep the generated loop functionally equivalent to the original loop. See “Optimize MATLAB Loops” on page 8-20

See Also

`makehdl`

More About

- “Clock Rate Pipelining” on page 24-118

- “Distributed Pipelining: Speed Optimization” on page 24-108
- “Resource Sharing For Area Optimization” on page 24-40
- “Streaming: Area Optimization” on page 24-36

Automatic Iterative Optimization

In this section...

- ["How Automatic Iterative Optimization Works" on page 24-7](#)
- ["Automatic Iterative Optimization Output" on page 24-7](#)
- ["Automatic Iterative Optimization Report" on page 24-8](#)
- ["Requirements for Automatic Iterative Optimization" on page 24-8](#)
- ["Limitations of Automatic Iterative Optimization" on page 24-8](#)

Automatic iterative optimization enables you to optimize your clock frequency without specifying individual optimization options, such as input or output pipelining, distributed pipelining, or loop unrolling.

There are two ways to use `hdlcoder.optimizeDesign` to optimize your clock frequency:

- Best clock frequency: You specify the maximum number of iterations you want HDL Coder to perform, and the coder iterates to minimize the critical path in your design.
- Target clock frequency: You specify a clock frequency target for your design and the maximum number of iterations you want HDL Coder to perform. The coder iterates until it meets your target clock frequency or reaches the maximum number of iterations.

HDL Coder can also determine that your target clock frequency is not achievable because your target clock period is less than the latency of the largest atomic combinational group of logic in your design.

How Automatic Iterative Optimization Works

You specify your clock frequency goal and the maximum number of iterations. HDL Coder performs the following steps for each iteration:

- 1 Analyzes the logic in your design.
- 2 Generates code.
- 3 Uses the synthesis tool to analyze the generated code, and obtains post-map timing analysis data.
- 4 Back annotates the design with the timing analysis data.
- 5 Inserts pipeline registers to break the critical path.
- 6 Balances delays.
- 7 Saves iteration data in a new folder.

When HDL Coder has met your clock frequency goal or it has reached the maximum number of iterations, it saves the generated code and iteration data in a new folder and generates a report that describes the final critical path.

Automatic Iterative Optimization Output

When HDL Coder exits the optimization loop, it saves the results of the final iteration in a folder, `hdlsrc/your_model_name/hdlexpl/Final-timestamp`.

The final iteration folder contains:

- The generated HDL code, in `hdlsrc/your_model_name`
- A data file, `cpGuidance.mat`, that you can use with your original model to regenerate code without rerunning the iterative optimization.
- The optimization report, `summary.html`.

HDL Coder also saves

Automatic Iterative Optimization Report

HDL Coder generates a report for the final optimization iteration and saves it in the final iteration folder, `hdlsrc/your_model_name/hdlexpl/Final-timestamp`.

The final optimization report, `summary.html`, contains the following:

- Summary Section, with:
 - Final critical path latency.
 - Critical path latency and elapsed time for each iteration.
- Diagnostic Section, with:
 - Reason for stopping at the final iteration.
 - Model or block settings that can reduce the accuracy of the critical path analysis.

If your model has these settings, remove them where possible, and rerun `hdlcoder.optimizeDesign`. Some optimizations, such as distributed pipelining and constrained output pipeline, change the placement of pipeline registers after the coder analyzes the critical path.

- Critical path description, which shows signals and components in both the original model and generated model that are part of the critical path.

You may see a message that says a signal or component on the critical path cannot be traced back to the original model. HDL Coder may not be able to map its internal representation of your design back to the original design. Each optimization iteration changes the internal representation, so the final representation can have a structure that is different from your original design.

Requirements for Automatic Iterative Optimization

Your synthesis tool must be Xilinx ISE or Xilinx Vivado, and your target device must be a Xilinx FPGA.

Limitations of Automatic Iterative Optimization

- In the current release, automatic iterative optimization does not support Altera hardware.
- Running automatic iterative optimization can take a long time, depending on the complexity of your design. To help mitigate the time cost, `hdlcoder.optimizeDesign` can regenerate code from a previous run, or resume from an interrupted run.
- Automatic iterative optimization is available from the command line only.
- HDL Coder uses post-map timing information, which the synthesis tool generates before performing place and route. Post-map timing information is less accurate than timing information the synthesis tool generates after place and route, but is faster to obtain.

See Also

`hdlcoder.optimizeDesign`

Generated Model and Validation Model

In this section...

- “Generated Model” on page 24-10
- “Validation Model” on page 24-11

HDL Coder enables you to view the effect of HDL optimizations and block settings in the generated model.

Generated Model

Before generating code, HDL Coder creates a behavioral model of the HDL code called the generated model. The generated model is an intermediate model that captures cycle-accurate and bit-true behavior of the generated code in area and timing optimizations during code generation. It shows latency and numeric differences between your Simulink DUT and the generated HDL code. Delays that the code generator inserts are highlighted in the generated model in various colors. See the table below for different delays in code generation, the corresponding highlight color, and how the delays are named in the generated model.

Delays	Highlight Color	Naming Convention
<ul style="list-style-type: none"> • Block implementation • RAM mapping 	Cyan	The block is highlighted in cyan. Delay blocks inside this block use the default name <code>Delay</code> and are not highlighted.
<p>“Constrained Output Pipelining” on page 24-112</p> <ul style="list-style-type: none"> • “Distributed Pipelining” on page 24-101 • “InputPipeline” on page 22-13 and “OutputPipeline” on page 22-17 • “Delay Balancing” on page 24-63 • “Clock-Rate Pipelining” on page 24-114 • “Adaptive Pipelining” on page 24-130 	Green	Constrained output pipelining: <code>rd_n</code>

With timing and area optimizations, the generated model is substantially different from the original model. For example, you can see additional integer delays next to blocks if you request optimizations and additional balanced delays wherever necessary to maintain the accuracy of the algorithm. You see additional rates in the model if you request resource sharing or streaming optimization where the same operator is time multiplexed across multiple operations.

The generated model is used in RTL testbench generation. The input stimulus and output response are captured from the generated model instead of the original model because the generated model reflects the algorithms timing changes required for optimizations. If you disable model generation, you cannot generate a test bench in HDL Coder.

After code generation, the generated model is saved in the target folder. By default, the generated model prefix is `gm_`. For example, if your model name is `myModel`, your generated model name is `gm_myModel`.

Customize the Generated Model

To customize the prefix of the generated model name, use the `Generated modelNamePrefix` property with `makehdl` or `hdlset_param`. See “Prefix for generated model name” on page 17-85.

You can also specify various options for the naming and layout of the generated model. See “Naming and Layout Options for Model Generation” on page 17-85.

Validation Model

Because the generated model is often substantially different from the original model, the coder can also create a validation model to compare the original model to the generated model. The validation model inserts delays at the outputs of the original model to compensate for latency differences and compares the outputs of the two models. When you simulate the validation model, numeric differences in the output data trigger an assertion.

Using the validation model, you can verify that the output of the optimized DUT is bit-true to the results produced by the original DUT.

A validation model contains:

- A generated model.
- An original model that has compensating delays inserted.
- Original inputs, routed to the original model and the generated model.
- Scopes for comparing and viewing the outputs of the original model and generated model.

Latency Differences

Some block architectures and optimizations introduce latency. For example, for the Reciprocal block, you can specify HDL block architectures that implement the Newton-Raphson method. The Newton-Raphson method is iterative, so block architectures that use it are multicycle and introduce latency at the block rate.

Similarly, the resource sharing area optimization time-multiplexes data over a shared hardware resource, which introduces local multirate and latency at the upsampled rate.

Numeric Differences

HDL block architectures can introduce numeric differences. For example:

- HDL block property such as “InputPipeline” on page 22-13 specified on the block, or certain HDL architectures or optimizations such as distributed pipelining that moves delays to the input of the block.
- The Newton-Raphson method is an approximation. If you select a Newton-Raphson block implementation, the generated model shows a change in numerics.
- HDL implementations for signal processing blocks, such as filters, can change numerics.

See also “Locate Numeric Differences After Speed Optimization” on page 24-13.

Generate A Validation Model

- In the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > Model Generation** pane, select **Validation Model**.
- In the HDL Workflow Advisor, in the **HDL Code Generation > Generate RTL Code and Testbench** pane, enable **Generate validation model**.
- Use the `GenerateValidationModel` property with `makehdl` or `hdlset_param`.

Customize the Validation Model

To customize the suffix of the generated validation name, use the `Validation modelNameSuffix` property with `makehdl` or `hdlset_param`. See “[Suffix for validation model name](#)” on page 17-85.

Restrictions

- To generate a validation model, you must generate HDL code for the DUT Subsystem. Model generation is not supported for generating code for the entire model instead of the DUT Subsystem.
- Make sure the DUT subsystem has no unconnected output ports. See “[Terminate Unconnected Block Outputs and Usage of Commenting Blocks](#)” on page 21-25.

See Also

More About

- “[Delay Balancing](#)” on page 24-63

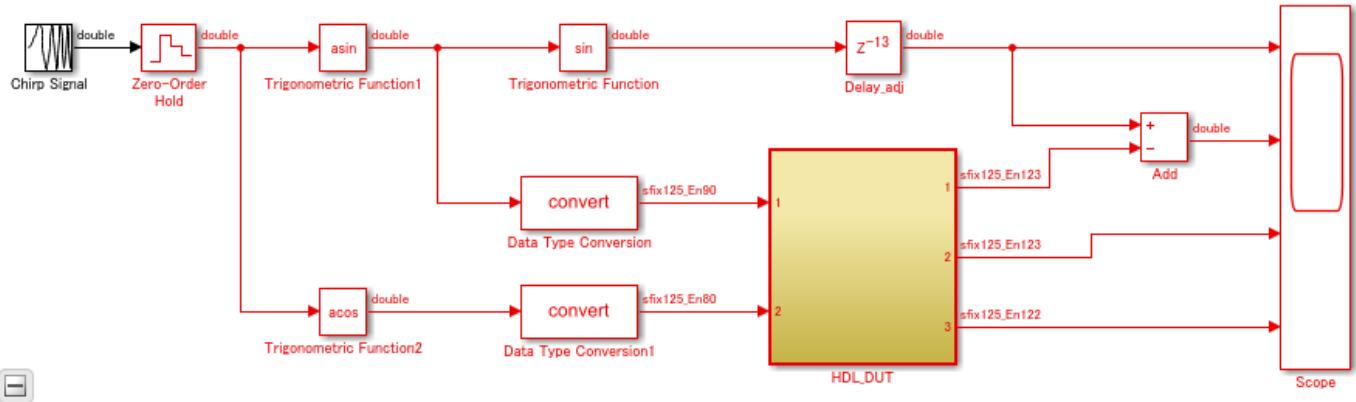
Locate Numeric Differences After Speed Optimization

This example shows how a model that contains Trigonometric Function blocks might have differences in numeric results after HDL code generation. You observe these numeric differences in the generated validation model. The validation model compares the original model with the generated model that shows the effect of block implementations and speed and area optimizations.

Trigonometric Function Model

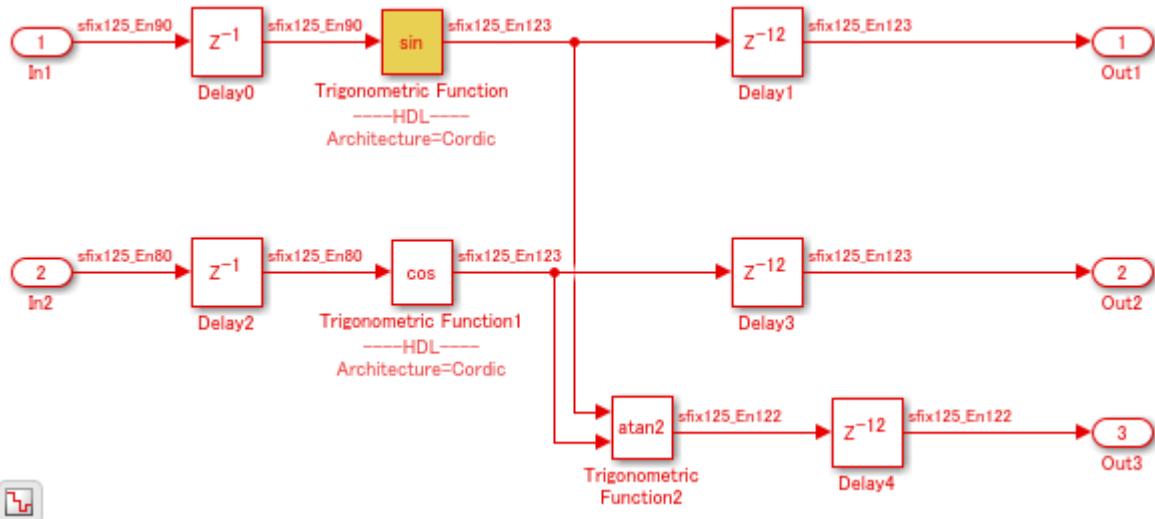
Open the model `hdlcoder_sincos_cordic_optimization`.

```
open_system('hdlcoder_sincos_cordic_optimization')
set_param('hdlcoder_sincos_cordic_optimization', 'SimulationCommand', 'Update')
```



Inside the `HDL_DUT` subsystem, this model uses Trigonometric Function blocks that have HDL architecture set to CORDIC and **LatencyStrategy** set to MAX. The block settings introduce pipelines at the input of the Trigonometric Function blocks.

```
open_system('hdlcoder_sincos_cordic_optimization/HDL_DUT')
```



The model has optimizations such as hierarchical distributed pipelining enabled on the model. To see the HDL parameters saved on the model, use the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_sincos_cordic_optimization')

% Set Model 'hdlcoder_sincos_cordic_optimization' HDL parameters
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ClockRatePipelining', 'off');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'EDAScriptGeneration', 'off');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'EnableTestpoints', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'GenerateCoSimModel', 'ModelSim');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLGenerateWebview', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSubsystem', 'hdlcoder_sincos_cordic_optimization');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthCmd', 'set_global_assignment -name V');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthFilePostfix', '_quartus.tcl');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthInit', 'load_package flow\nset top _');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthTerm', 'execute_flow -compile\nproject');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HDLSynthTool', 'Quartus');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'HierarchicalDistPipelining', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MaskParameterAsGeneric', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MinimizeClockEnables', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'MinimizeIntermediateSignals', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ResourceReport', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'ShareAdders', 'on');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'TargetLanguage', 'Verilog');
hdlset_param('hdlcoder_sincos_cordic_optimization', 'Traceability', 'on');

hdlset_param('hdlcoder_sincos_cordic_optimization/HDL_DUT/Trigonometric Function', 'Architecture');
hdlset_param('hdlcoder_sincos_cordic_optimization/HDL_DUT/Trigonometric Function1', 'Architecture');
```

Generate HDL Code and Validation Model

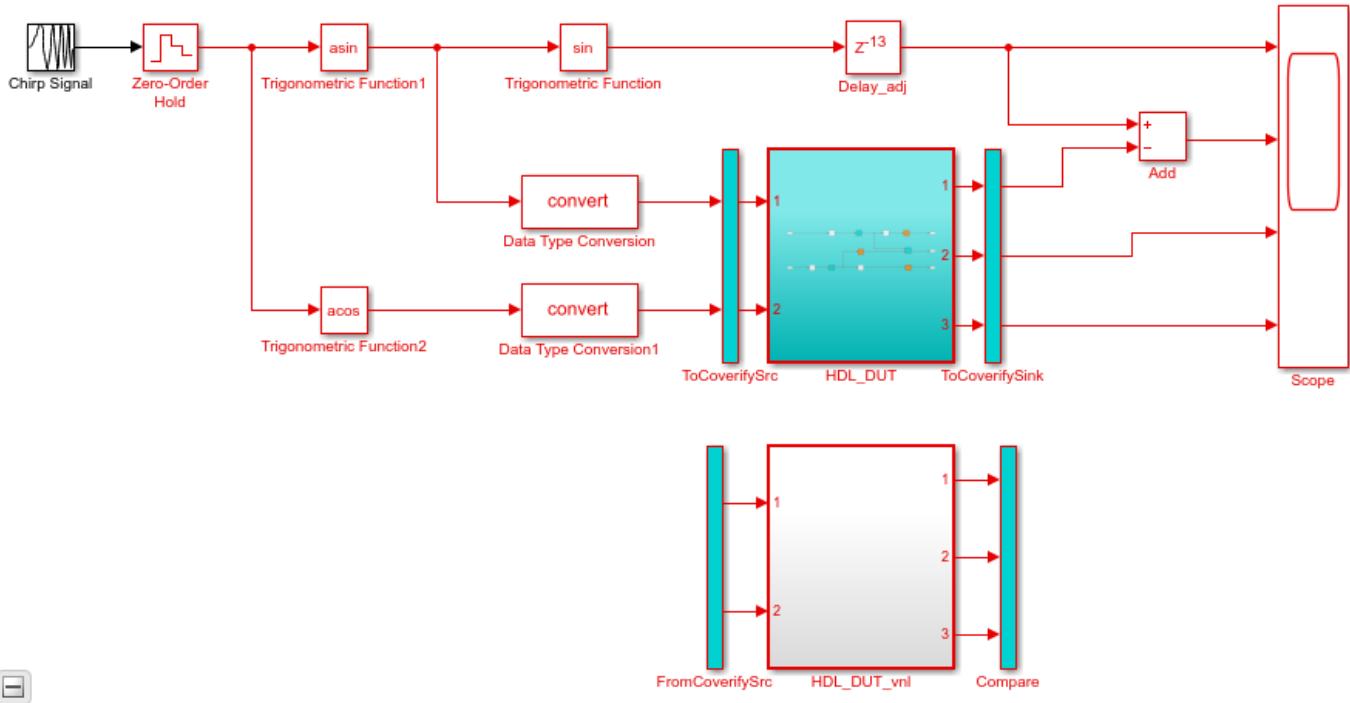
To see the effect of the optimization, generate HDL code and validation model for the `HDL_DUT` subsystem by using the `makehdl` function.

```
makehdl('hdlcoder_sincos_cordic_optimization')
```

When you open the HDL Check Report, you see a warning message displayed that indicates delays introduced at the inputs of the blocks, which might cause a numeric mismatch in the initial cycles when simulating the validation model.

After code generation, you see the model `gm_hdlcoder_sincos_cordic_optimization_vnl`. In this example, the model has been saved with the name `hdlcoder_sincos_cordic_optimization_validation`.

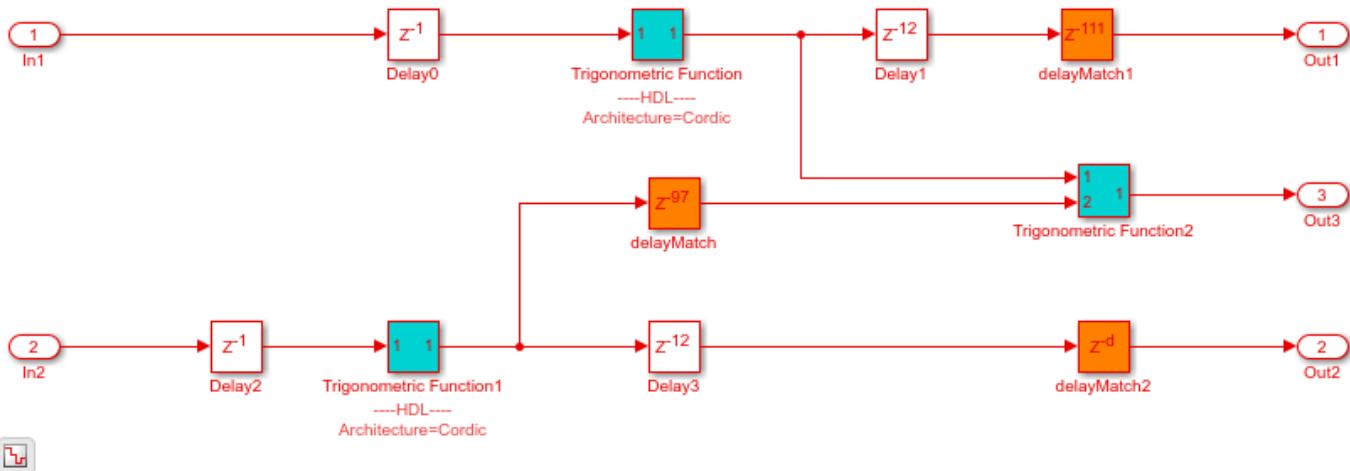
```
open_system('hdlcoder_sincos_cordic_optimization_validation')
set_param('hdlcoder_sincos_cordic_optimization_validation', 'SimulationCommand', 'Update')
```



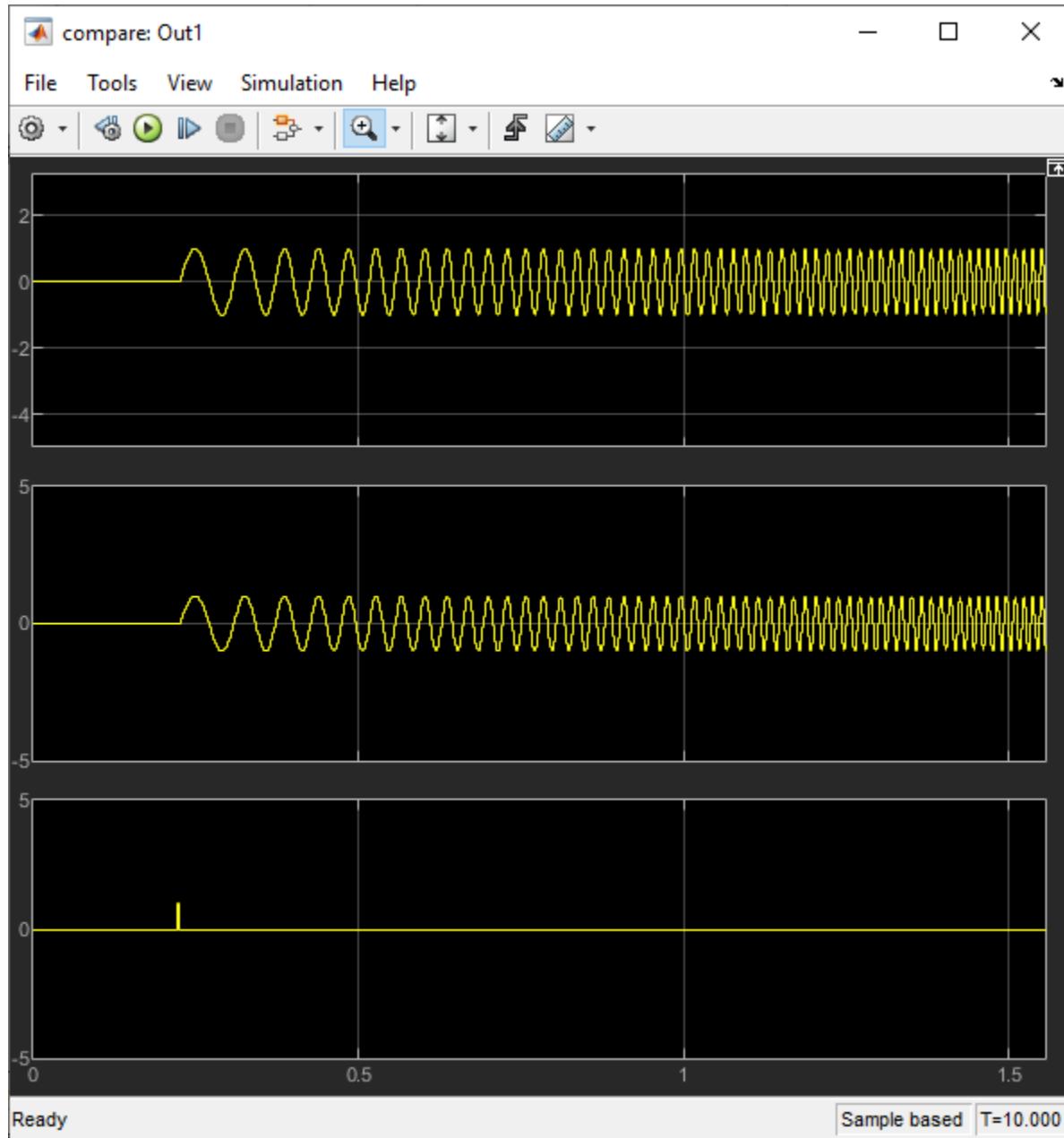
Observe Numeric Differences

The HDL_DUT subsystem highlighted in cyan indicates that this subsystem is different from the subsystem in the original model HDL_DUT_vnl. The HDL_DUT subsystem is part of the generated model after HDL code generation, and shows the effect of optimizations. You also see the pipelines introduced by the Trigonometric Function blocks.

```
open_system('hdlcoder_sincos_cordic_optimization_validation/HDL_DUT')
```



When you simulate the model, you see assertions detected in the initial cycles of simulation, which indicates a numeric mismatch. The mismatch is caused by pipelines introduced at the input of the block. To fix the mismatch, avoid using block implementations, or HDL block properties such as **InputPipeline**, or optimizations that introduce pipelines at the input of the blocks.



See Also

More About

- “Generated Model and Validation Model” on page 24-10
- “Resolve Numerical Mismatch with Delay Balancing” on page 24-24

Simplify Constant Operations and Reduce Design Complexity in HDL Coder™

HDL Coder performs certain optimization techniques that improve the quality of the generated HDL code. When you use floating-point data types in Native Floating Point mode and generate code from your model, at compile-time, HDL Coder searches for a subset of blocks that fit a certain pattern. When the code generator recognizes the pattern, it automatically performs certain optimization techniques to replace the blocks in the subset with other, simpler blocks.

The optimization techniques:

- Remove redundant run-time computations
- Simplify your design
- Improve quality and efficiency of generated HDL code
- Reduce latency and area footprint
- Improve timing of your design on target hardware

Simplification of Constant operations

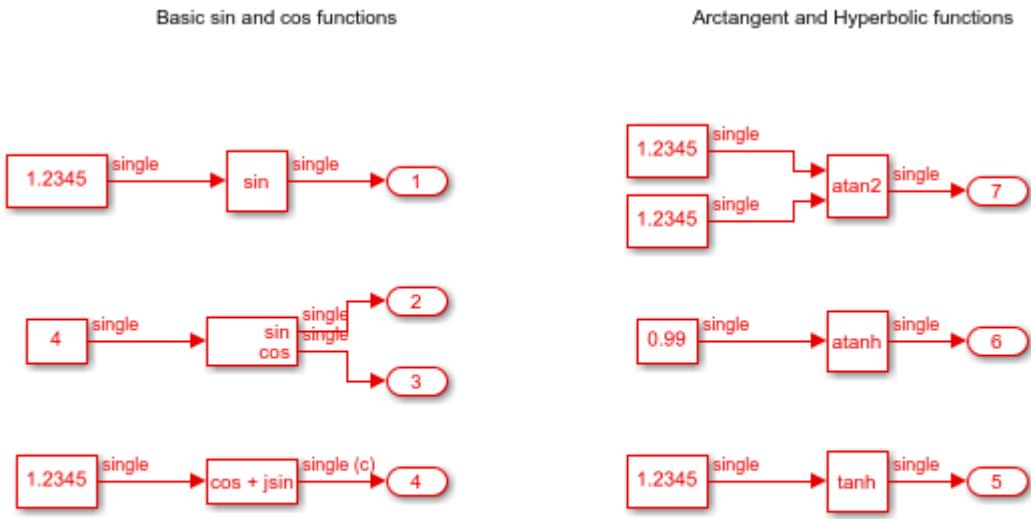
HDL Coder removes redundant operations in your design by evaluating constant subexpressions in advance. This optimization technique identifies Simulink® blocks in your model that have constant values at all input ports, and then replaces the blocks with Constant blocks. The code generator propagates the input constants inside the blocks to compute the resulting Constant value.

For example, $c = 3 * 5$ becomes $c = 15$.

This optimization works with any Simulink block that supports HDL code generation. For example:

- Open the model `hdlcoder_constant_simplification`. Double-click the Trigonometric Functions Subsystem.

```
open_system('hdlcoder_constant_simplification')
open_system('hdlcoder_constant_simplification/Trigonometric Functions')
set_param('hdlcoder_constant_simplification', 'SimulationCommand', 'update');
```

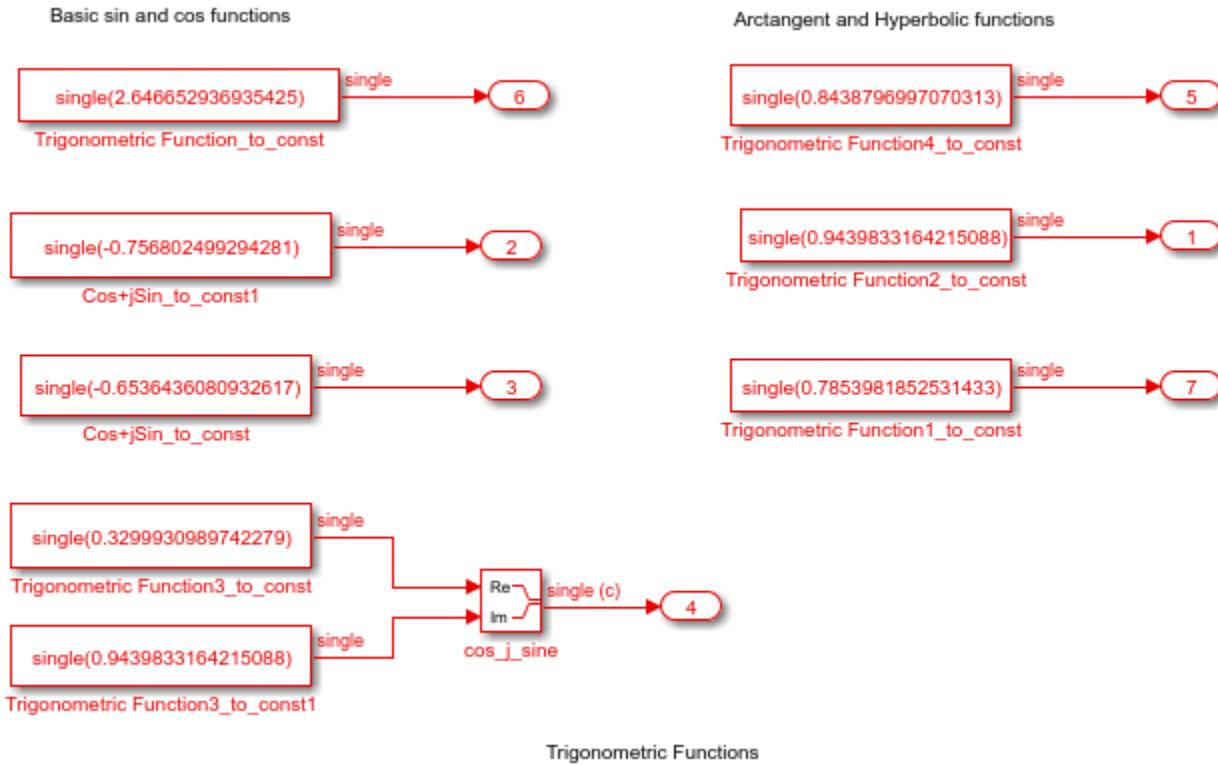


Trigonometric Functions

- To generate HDL code for the `hdlcoder_constant_simplification` model, enter this command.

```
makehdl('hdlcoder_constant_simplification')
```

- Open the generated model, and double-click the Trigonometric Functions subsystem.



HDL Coder™ recognized the modeling pattern and replaced the constant single-precision trigonometric operations with constants. This optimization results in significant area reduction and improvements to timing when you deploy the code onto the target platform.

Replacement of Slower Operations with Faster Equivalents

This optimization automatically replaces slower operations with faster equivalents.

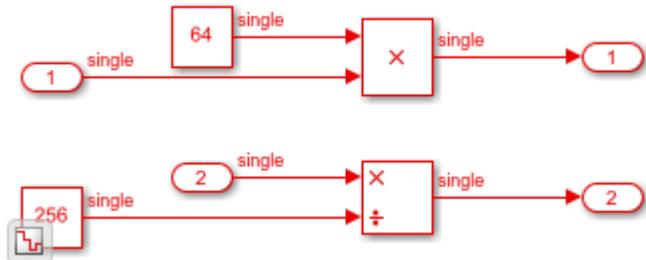
For example, $r1 = r2 / 2$ becomes $r1 = r2 \gg 1$.

Examples of this optimization technique include replacement of a Product block or a Divide block by a Gain block. If one of the inputs to a Product block or a Divide block is a constant and a power of two, the code generator replaces that block by a Gain block. The code generator propagates the **Constant value** inside the Product block or the Divide block to compute the **Gain** parameter. This optimization works with single data types in the Native Floating Point mode.

For example:

- Open the model `hdlcoder_slow_operation_replacement`. Double-click the DUT Subsystem.

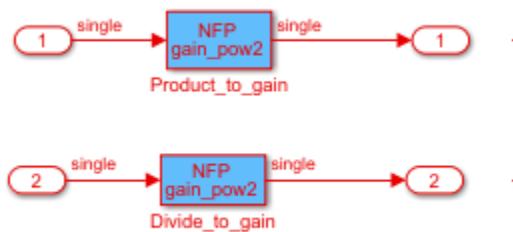
```
open_system('hdlcoder_slow_operation_replacement')
open_system('hdlcoder_slow_operation_replacement/DUT')
set_param('hdlcoder_slow_operation_replacement', 'SimulationCommand', 'update');
```



- To generate HDL code for the DUT Subsystem, enter this command:

```
makehdl('hdlcoder_slow_operation_replacement/DUT')
```

- Open the generated model, and double-click the DUT subsystem.



HDL Coder™ recognized the modeling pattern and replaced the Product block and the Divide block by a Gain block. This optimization significantly reduces the latency of your design, and improves area and timing on the target FPGA.

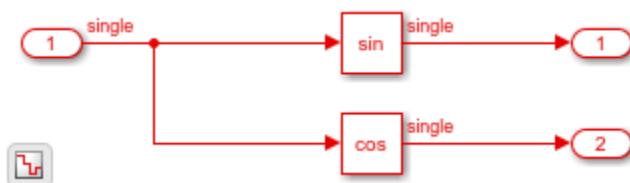
Combination of Multiple Operations

This optimization replaces several operations with one equivalent operation. Examples of this optimization technique include replacement of a Sin block and a Cos block by a Sincos block. If you provide the same input signal to a Sin block and a Cos block in your model, and when the HDL Block Properties of the Sin and Cos blocks match, the code generator replaces the blocks with a Sincos block. This optimization works with single data types in the Native Floating Point mode.

For example:

- Open the model `hdlcoder_combine_operations`. Double-click the DUT Subsystem.

```
open_system('hdlcoder_combine_operations')
open_system('hdlcoder_combine_operations/DUT')
set_param('hdlcoder_combine_operations', 'SimulationCommand', 'update');
```



- To generate HDL code for the DUT Subsystem, enter this command:

```
makehdl('hdlcoder_combine_operations/DUT')
```

- Open the generated model, and double-click the DUT Subsystem. The generated model appears as below.



HDL Coder™ recognized the modeling pattern and replaced the Sin block and the Cos block by a Sincos block. This optimization technique significantly improves the performance of your design on the target platform.

Considerations

- The optimizations work with floating-point data types. Fixed-point designs are not affected by this optimization. When you use single data types, enable the Native Floating Point mode. In the Configuration Parameters dialog box, on the **HDL Code Generation > Floating Point** pane, set **Floating-Point IP Library** to Native Floating Point. To learn about native floating-point support in HDL Coder, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103.
- The optimizations preserve all comments from the blocks in the generated code. To learn about specifying comments to blocks, see “Generate Code with Annotations or Comments” on page 25-13.
- The optimizations do not optimize blocks that use tunable parameters or generic inputs because the tunable parameters are not treated as constant values. To make these blocks participate in the optimization, in the Mask Editor for the blocks, clear the Tunable check box. To learn about tunable parameters, see “Generate DUT Ports for Tunable Parameters” on page 10-17.
- The optimizations treat enumeration values and constants from the Workspace browser as constant values. The code generator therefore propagates these values inside various components and simplifies the constant operations.

See Also

More About

- “Generated Model and Validation Model” on page 24-10
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166
- “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188

Optimization with Constrained Overclocking

In this section...

- “Why Constrain Overclocking?” on page 24-22
- “Optimizations that Overclock Resources” on page 24-22
- “How to Use Constrained Overclocking” on page 24-22
- “Constrained Overclocking Limitations” on page 24-23

Why Constrain Overclocking?

Area and timing optimizations that you specify can result in upsampled rates in your design. For example, when you use the resource sharing optimization, the code generator overclocks the shared resources by an overclocking factor (OCF). The OCF depends on the number of shareable resources, N , and the **SharingFactor**, SF , that you specify. If your clock rate is high, overclocking can cause your design clock rate to exceed the maximum clock rate of your target hardware. To constrain overclocking, use the **Oversampling factor** in conjunction with clock-rate pipelining to constrain the overclocking of your design.

Optimizations that Overclock Resources

Area and speed optimizations, and certain block implementations that you specify result in overclocking the resources in your design. For example, the following optimizations and implementations can result in upsampled rates in your design:

- RAM mapping
- Streaming
- Resource sharing
- Loop streaming
- Specific block implementations, such as cascade architectures, Newton-Raphson architectures, and some filter implementations

How to Use Constrained Overclocking

When using area and speed optimizations, you can specify constraints on overclocking using the **oversampling** parameter. If you want a single-rate design, you can use these parameters to prevent overclocking, or limit overclocking within a range.

Suppose that you have a design that does not currently fit in the target hardware, but is already running at the target device maximum clock frequency, and you know that the inputs to your design can change at most every N cycles. You can enable area optimizations, such as resource sharing, and specify a single-rate implementation using the **Oversampling factor**. You can specify the **Oversampling factor** in the **HDL Code Generation > Global Settings** pane of the Configuration Parameters dialog box.

By default, the clock-rate pipelining optimization is enabled, and it works in conjunction with the **Oversampling factor** to make the DUT sample time slower than the actual clock rate. You can design your model at the base sample time and then set the **Oversampling factor** to N . This setting gives HDL Coder a latency budget of N cycles to perform the computation. In this situation, HDL

Coder can reuse the shared resource at the original clock rate over N cycles, instead of implementing the sharing optimization by overclocking the shared resource.

Constrained Overclocking Limitations

When you constrain overclocking by specifying an **Oversampling factor** greater than 1, **ClockInputs** must be set to **Single**.

See Also

More About

- “Oversampling factor” on page 17-15
- “Clock-Rate Pipelining” on page 24-114

Resolve Numerical Mismatch with Delay Balancing

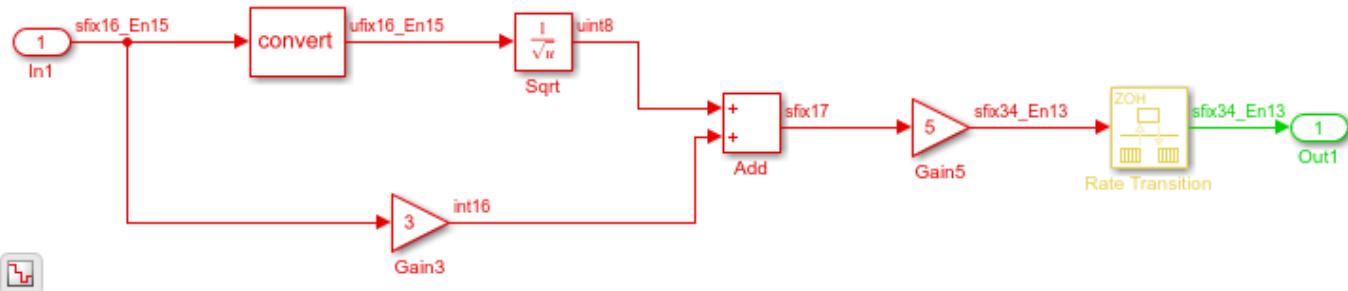
This example shows how to use delay balancing to resolve a numerical mismatch between the generated model and original model after HDL code generation.

Problem

The issue is that simulating the validation model results in a numerical mismatch between the original model and the generated model after HDL code generation. To illustrate this issue:

1. Open the `hdlcoder_resolve_delaybalancing` model. The DUT is a simple multirate design.

```
modelname = 'hdlcoder_resolve_delaybalancing';
dutname = 'hdlcoder_resolve_delaybalancing/Subsystem';
load_system(modelname)
open_system(dutname)
set_param(modelname, 'SimulationCommand', 'update');
```



2. Generate HDL code and validation model for the DUT.

```
makehdl(dutname, 'GenerateValidationModel', 'on', ...
        'TargetDirectory', 'C:/Temp/hdlsrc')

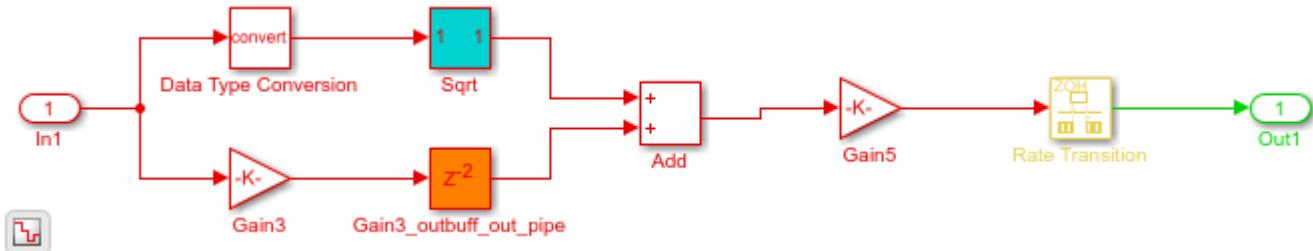
### Generating HDL for 'hdlcoder_resolve_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resolve...
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_resolve_delaybalanc...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_resolve_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc\hdlcoder...
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdlsrc\hdlcode...
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcoder_resolve...
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_tc.vhd.
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_resolve_delay...
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\Temp\hdlsrc\hdlcoder_resolve_delaybalanc...
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 2 warnings, and 4 mes...
### HDL code generation complete.
```

3. View the validation model. The validation model compares the generated model with the original model. The generated model displays the effect of optimizations and block-specific architectures that you specify. Use the validation model to verify that the DUT in the generated model is bit-true to the numerical results produced by the original DUT.

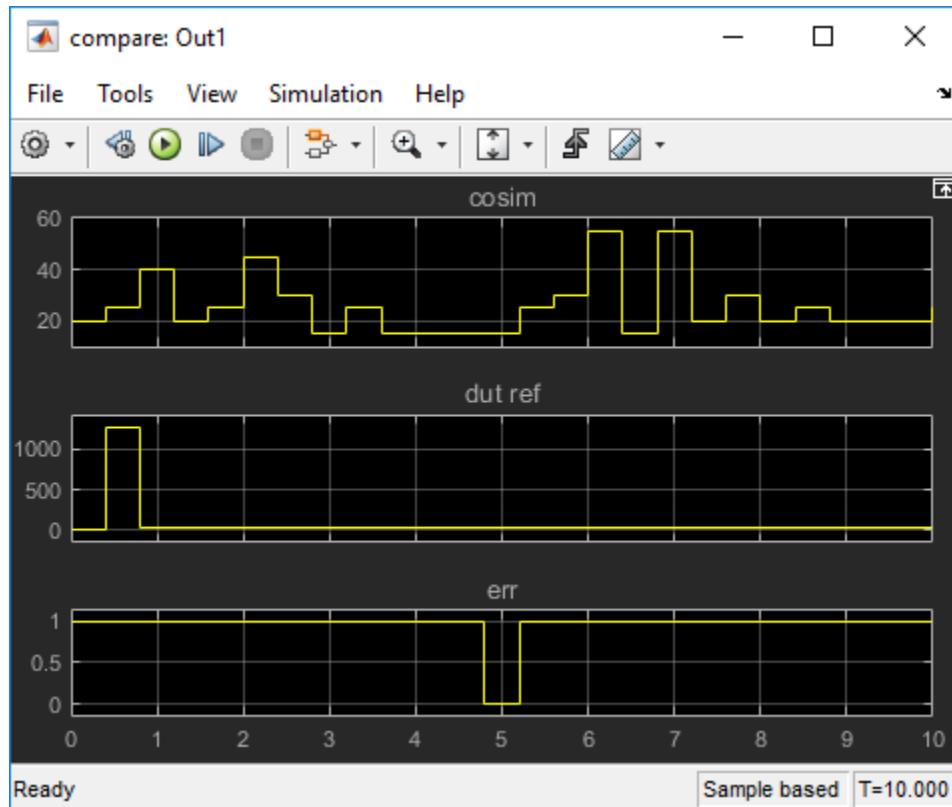
```

valmodelname = 'gm_hdlcoder_resolve_delaybalancing_vnl';
valmodelsubsys = 'gm_hdlcoder_resolve_delaybalancing_vnl/Subsystem';
load_system(valmodelname)
open_system(valmodelsubsys)
set_param(valmodelname, 'SimulationCommand', 'update');

```



4. Simulate the validation model. HDL Coder™ generates warnings that indicate an assertion detected at various time stamps. If you navigate through the validation model by double-clicking the Compare Subsystem and then the Assert_Out1 Subsystem, you see a compare: Out1 Scope block. This Scope block compares the output of the original model DUT with the generated model DUT and displays numerical differences as an error signal. When you double-click the Scope block, you see a nonzero error, which indicates a numerical mismatch.



Cause

To diagnose this issue:

1. Observe the parameters saved on the original model. You see that **BalanceDelays** is set to **off** on the model.

```
hdlsaveparams(modelname)

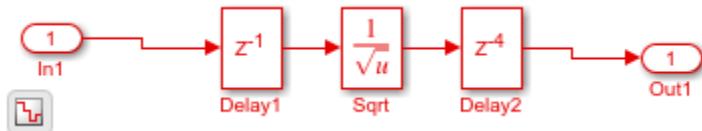
%% Set Model 'hdlcoder_resolve_delaybalancing' HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateHDLTestBench', 'off');
hdlset_param('hdlcoder_resolve_delaybalancing', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_resolve_delaybalancing', 'HDLSubsystem', 'hdlcoder_resolve_delaybalancing');

% Set Gain HDL parameters
hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Gain3', 'OutputPipeline', 2);

hdlset_param('hdlcoder_resolve_delaybalancing/Subsystem/Sqrt', 'Architecture', 'RecipSqrtNewton');
```

2. Inspect the validation model. Inside the DUT Subsystem, you see that the code generator implemented the reciprocal square root operation as a Subsystem. If you double-click the Sqrt Subsystem, you see that the implementation has a latency. This latency arises due to the Newton-Raphson implementation of reciprocal square root.

```
open_system('gm_hdlcoder_resolve_delaybalancing_vnl/Subsystem/Sqrt')
```



The simulation mismatch occurred because the Newton-Raphson choice for implementing the Reciprocal Sqrt block results in a latency difference between the original model and the generated model. In addition, the downsampling introduced by the Rate Transition block drops samples. As delay balancing is disabled on the model, the code generator did not add matching delays to account for this latency.

Solution

To fix this issue, enable delay balancing on the model. In the original model, set **BalanceDelays** to **on**. When you enable delay balancing, the code generator detects introduction of delays along one path and adds matching delays on other, parallel signal paths.

1. Enable **BalanceDelays** on the model and generate HDL code and validation model.

```
load_system(modelname)
makehdl(dutname, 'BalanceDelays', 'on', ...
        'GenerateValidationModel', 'on', ...
        'TargetDirectory', 'C:/Temp/hdlsrc')

### Generating HDL for 'hdlcoder_resolve_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_resolve...
### Starting HDL check.
### 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 addit...
### Output port 0: 2 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_resolve_delaybalanc...
```

```

### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_resolve_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_iv as C:\Temp\hdlsrc\hdlcoder_
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt/Sqrt_core as C:\Temp\hdlsrc\hdlcode
### Working on hdlcoder_resolve_delaybalancing/Subsystem/Sqrt as C:\Temp\hdlsrc\hdlcoder_resolve_
### Working on Subsystem_tc as C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_tc.vhd.
### Working on hdlcoder_resolve_delaybalancing/Subsystem as C:\Temp\hdlsrc\hdlcoder_resolve_delay
### Generating package file C:\Temp\hdlsrc\hdlcoder_resolve_delaybalancing\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\Temp\hdlsrc\hdlcoder_resolve_delaybalanc
### HDL check for 'hdlcoder_resolve_delaybalancing' complete with 0 errors, 0 warnings, and 4 mes
### HDL code generation complete.

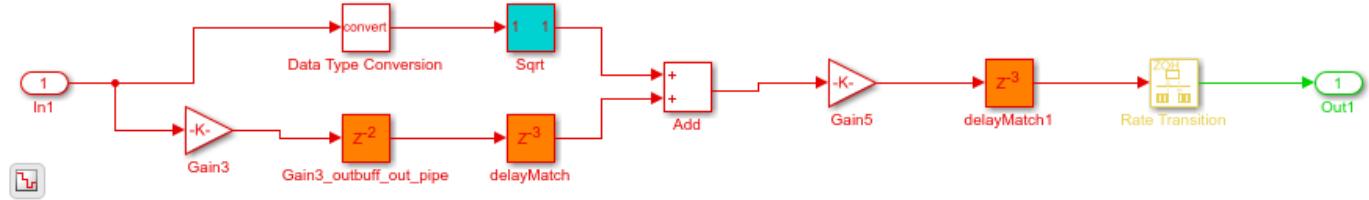
```

2. Open the validation model. You see that the code generator introduced matching delays to balance the latency introduced by the Sqrt block and to offset the effect of downsampling. The additional delays account for the latency difference.

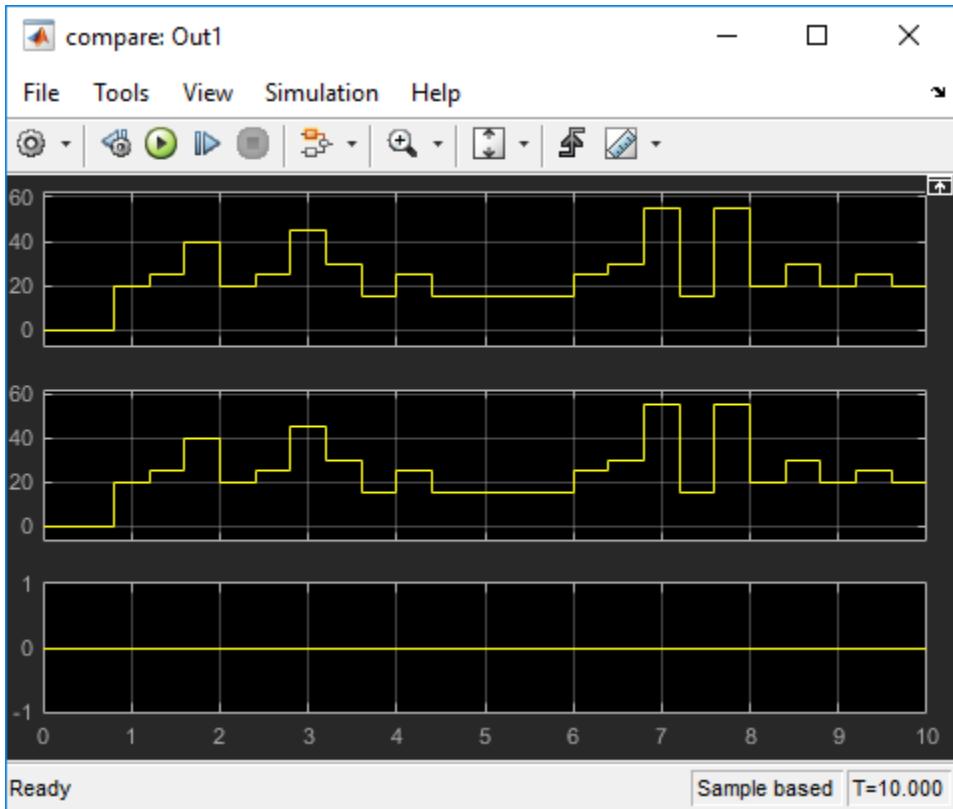
```

load_system(valmodelname)
open_system(valmodelsys)
set_param(valmodelname, 'SimulationCommand', 'update');

```



3. Simulate the validation model and open the compare: **Out1** Scope block. You see that the numerical mismatch has been resolved.



See Also

Related Examples

- “Delay Balancing and Validation Model Workflow In HDL Coder™” on page 24-68

More About

- “Delay Balancing” on page 24-63
- “Generated Model and Validation Model” on page 24-10
- “Check delay balancing setting” on page 38-11

Streaming

In this section...

- “What Is Streaming?” on page 24-29
- “Specify Streaming” on page 24-29
- “How to Determine Streaming Factor and Sample Time” on page 24-30
- “Determine Blocks That Support Streaming” on page 24-30
- “Requirements for Streaming Subsystems” on page 24-30
- “Streaming Report” on page 24-31

What Is Streaming?

Streaming is an area optimization in which HDL Coder transforms a vector data path to a scalar data path (or to several smaller-sized vector data paths). By default, HDL Coder generates fully parallel implementations for vector computations. For example, the code generator realizes a vector sum as several adders, executing in parallel during a single clock cycle. This technique can consume many hardware resources. With streaming, the generated code saves chip area by multiplexing the data over a smaller number of shared hardware resources.

By specifying a streaming factor for a subsystem, you can control the degree to which such resources are shared within that subsystem. When the ratio of streaming factor (N_{st}) to subsystem data path width (V_{dim}) is 1:1, HDL Coder implements an entirely scalar data path. A streaming factor of 0 (the default) produces a fully parallel implementation (that is, without sharing) for vector computations.

If you know the maximal vector dimensions and the sample rate for a subsystem, you can compute the possible streaming factors and resulting sample rates for the subsystem. However, even if the requested streaming factor is mathematically possible, the subsystem must meet other criteria for streaming.

By default, when you apply the streaming optimization, HDL Coder oversamples the shared hardware resource to generate an area-optimized implementation with the original latency. If the streamed data path is operating at a rate slower than the base rate, the code generator implements the data path at the base rate. You can also limit the oversampling ratio to meet target hardware clock constraints. To learn more, see “Clock-Rate Pipelining” on page 24-114.

You can generate and use the validation model to verify that the output of the optimized DUT is bit-true to the results produced by the original DUT. To learn more about the validation model, see “Generated Model and Validation Model” on page 24-10.

Specify Streaming

To specify streaming from the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the subsystem, model reference, or MATLAB Function block and then click **HDL Block Properties**. In the **StreamingFactor** field, enter the number of resources that you want to stream.

Note For MATLAB Function blocks, to specify the **StreamingFactor**, in the HDL Block Properties dialog box, you must set the HDL architecture of the block to **MATLAB Datapath**.

- Right-click the subsystem, model reference, or MATLAB Function block and select **HDL Code > HDL Block Properties**. In the **StreamingFactor** field, enter the number of resources that you want to stream.

At the command-line, you can set **StreamingFactor** using the `hdlset_param` function, as in the following example.

```
modelname = 'sfir_fixed';
dut = 'sfir_fixed/symmetric_fir';
open_system(modelname);
hdlset_param(dut, 'StreamingFactor', 4);
```

How to Determine Streaming Factor and Sample Time

In a given subsystem, if N_{st} is the streaming factor, and V_{dim} is the maximum vector dimension, then the data path of the resultant streamed subsystem is one of the following:

- Of width $V_{stream} = (V_{dim}/N_{st})$, if $V_{dim} > N_{st}$.
- Of width $V_{stream} = (N_{st}/V_{dim})$, if $N_{st} > V_{dim}$.
- Scalar.

If the original data path operated with a sample time, S , that is equal to the base sample time, then the streamed subsystem operates with a sample time of:

- S / N_{st} , if $V_{dim} > N_{st}$.
- S / V_{dim} , if $N_{st} > V_{dim}$.

If the original data path operated with a sample time, S , that is greater than the base sample time, S_{base} , then the streamed subsystem operates with a sample time of $S_{base} / \text{Oversampling}$. Notice that the streamed sample time is independent of the original sample time, S .

Determine Blocks That Support Streaming

HDL Coder supports many blocks for streaming. As a best practice, run the `checkhdl` function before generating streaming code for a subsystem. `checkhdl` reports blocks in your subsystem that are incompatible with streaming. If you initiate streaming code generation for a subsystem that contains incompatible blocks, the coder works around those blocks and generates non-streaming code for them.

HDL Coder cannot apply the streaming optimization to a model reference.

Requirements for Streaming Subsystems

Before applying streaming, HDL Coder performs a series of checks on the subsystems to be streamed. You can stream a subsystem if:

- The streaming factor N_{st} is a perfect divisor of the vector width V_{dim} , or the vector width must be a perfect divisor of the streaming factor.
- All inputs to the subsystem have the same vector size. If the inputs have different vector sizes, you can stream the subsystem by flattening the subsystem hierarchy. When you flatten the hierarchy, the streaming optimization identifies regions with different vector sizes and creates streaming

groups for these regions. These groups have different streaming factors that are inferred from the vector sizes.

Streaming Report

To see the streaming information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate an optimization report, in the **Streaming and Sharing** section, you see the effect of the streaming optimization. If streaming is unsuccessful, the report shows diagnostic messages and offending blocks that caused streaming to fail. When the requested streaming factor cannot be implemented, HDL Coder generates non-streaming code.

If streaming is successful, the report displays the **StreamingFactor** that was inferred, and a table that specifies:

- **Group:** A unique group ID for a group of Simulink blocks that belong to a streaming group.
- **Inferred Streaming Factor:** Streaming factor inferred by HDL Coder with the **Streaming Factor** that you specify in the HDL Block Properties.

To see groups of blocks that belong to a streaming group in your Simulink model and in the generated model, click the **Highlight streaming groups and diagnostics** link in the report.

See Also

More About

- “Resource Sharing” on page 24-32
- “Clock-Rate Pipelining” on page 24-114

Resource Sharing

In this section...

- “How Resource Sharing Works” on page 24-32
- “Benefits and Costs of Resource Sharing” on page 24-32
- “Shareable Resources in Different Blocks” on page 24-33
- “Specify Resource Sharing” on page 24-33
- “Limitations for Resource Sharing” on page 24-33
- “Block Requirements for Resource Sharing” on page 24-34
- “Resource Sharing Report” on page 24-34

Resource sharing is an area optimization in which HDL Coder identifies multiple functionally equivalent resources and replaces them with a single resource. The data is time-multiplexed over the shared resource to perform the same operations.

How Resource Sharing Works

You can specify a sharing factor SF for a subsystem or a MATLAB Function block. HDL Coder tries to identify a certain number of identical, shareable resources N up to SF. How the code generator shares these resources depends on N, SF, and the **Oversampling factor**.

By default, the **Oversampling factor** is 1, and resource sharing overclocks the shared resources by an overclocking factor (OCF) that depends on the remainder of SF and N.

```
if rem(SF,N) == 0
    OCF = N;
else
    OCF = SF;
end
```

If you specify an **Oversampling factor** greater than 1, your design operates a faster clock rate on the target hardware because clock-rate pipelining is enabled by default. When you specify a **SharingFactor**, the resource sharing optimization tries to share up to N resources, and overclocks the shared resources by a factor given by:

$$\text{Overclocking factor} = (\text{block_rate} \div \text{DUT_base_rate}) \times \text{Oversampling}$$

You can use the validation model to verify that the output of the optimized DUT is bit-true to the results produced by the original DUT. To learn more about the validation model, see “Generated Model and Validation Model” on page 24-10.

Benefits and Costs of Resource Sharing

Resource sharing can substantially reduce your chip area. For example, the generated code can use one multiplier to perform the operations of several identically configured multipliers from the original model. However, resource sharing has the following costs:

- Uses more multiplexers and can use more registers.

- Reduces opportunities for distributed pipelining or retiming, because HDL Coder does not pipeline across clock rate boundaries.
- Multiplies the clock rate of the target hardware by the sharing factor.

Shareable Resources in Different Blocks

If you specify a nonzero sharing factor for a MATLAB Function block, HDL Coder identifies and shares functionally equivalent multipliers.

If you specify a nonzero sharing factor for a Subsystem, HDL Coder identifies and shares functionally equivalent instances of the following types of blocks:

- Gain
- Product
- Multiply-Add
- Add or Sum with two inputs
- Atomic Subsystem
- MATLAB Function

The code generator shares functionally equivalent MATLAB Function blocks with fixed-point types. When you use floating-point types or use the MATLAB Datapath architecture for MATLAB Function blocks with fixed-point types, HDL Coder treats the MATLAB Function block as a regular Subsystem. You can then share functionally equivalent resources inside the MATLAB Function block. To learn more, see “Use MATLAB Datapath Architecture for Sharing with MATLAB Function Blocks” on page 21-110.

Specify Resource Sharing

To specify resource sharing from the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the subsystem, model reference, or MATLAB Function block and then click **HDL Block Properties**. In the **SharingFactor** field, enter the number of shareable resources.
- Right-click the subsystem, model reference, or MATLAB Function block and select **HDL Code > HDL Block Properties**. In the **SharingFactor** field, enter the number of shareable resources.

At the command-line, set the **SharingFactor** using `hdlset_param`, as in the following example.

```
modelname = 'sfir_fixed';
dut = 'sfir_fixed/symmetric_fir';
open_system(modelname)
hdlset_param(dut,'SharingFactor', 4);
```

Limitations for Resource Sharing

- Multirate sharing cannot share resources that have different number of pipelines inserted from adaptive pipelining.
- Model references are not supported for resource sharing.

Block Requirements for Resource Sharing

Blocks to be shared must have the following requirements:

- Single-rate.
- No infinite sample rate. The DUT must not contain blocks with **Sample time** set to Inf. For example, Constant blocks must have **Sample time** set to -1. To set the sample time to -1 for all Constant blocks in your DUT, use the following MATLAB code:

```
blks = find_system(dut, 'BlockType', 'Constant');
for i = 1:length(blks)
    set_param(blks{i}, 'SampleTime', '-1');
end
```

- No bus inputs or outputs.
- No tunable mask parameters. To share these blocks, in the Mask Editor, clear the **Tunable** check box.
- If the block is within a feedback loop, at least one Unit Delay or Delay block must be connected to each output port.

To learn about block-specific settings and requirements for resource sharing, see:

- “Resource Sharing Settings for Various Blocks” on page 21-105
- “Resource Sharing of Subsystems and Floating-Point IPs” on page 21-109

Resource Sharing Report

To see the resource sharing information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate the optimization report, in the **Streaming and Sharing** section, you see the effect of the resource sharing optimization. If resource sharing is unsuccessful, the report shows diagnostic messages and offending blocks that cause resource sharing to fail.

If resource sharing is successful, the report displays the **SharingFactor**, and a table that contains groups of blocks that shared resources. The table contains:

- **Group Id**: A unique ID for a group of similar Simulink blocks, such as add or product blocks, that share resources.
- **Resource Type**: The type of Simulink block in a sharing group.
- **I/O Wordlengths**: Word lengths of inputs to and output from the block in a sharing group.
- **Group size**: Number of blocks of the same type in a sharing group.
- **Block name**: Name of a block that belongs to a sharing group.
- **Color Legend**: Color that highlights all the blocks in a sharing group.

To see the shared resources in your Simulink model and in the generated model, click the **Highlight shared resources and diagnostics** link.

See Also

Related Examples

- Resource Sharing For Area Optimization
- “Single-rate Resource Sharing Architecture” on page 24-50

More About

- “Clock-Rate Pipelining” on page 24-114
- “Streaming” on page 24-29

Streaming: Area Optimization

This example shows how to use the subsystem level streaming optimization in HDL Coder.

Introduction

Streaming is a subsystem-wide optimization supported by HDL Coder for implementing area-efficient hardware. By default, the coder implements hardware that is bit-accurate and cycle-accurate to the Simulink model. This implies that vector datapaths in Simulink map inefficiently to hardware.

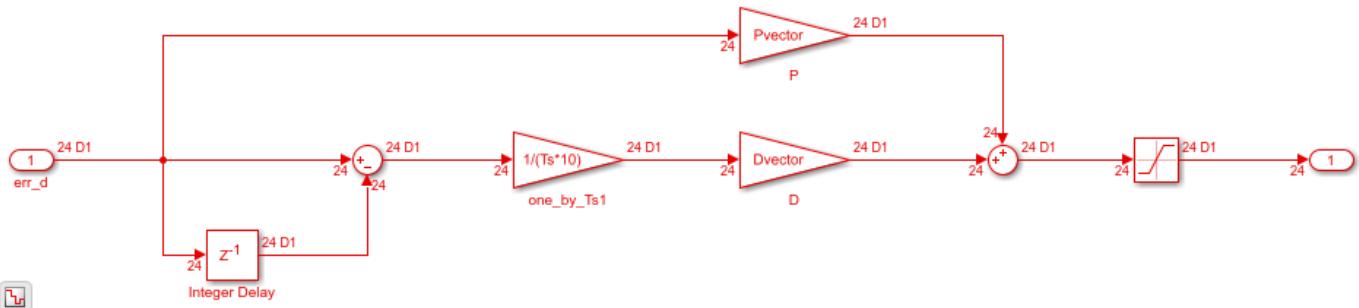
Consider a product block in Simulink that operates on two 64-element vector inputs and generates a 64-element vector output. This block executes 64 multiplications in a single Simulink time step. To remain cycle-accurate, HDL Coder maps this block to 64 parallel multipliers in the generated HDL code. Given that multipliers are expensive on FPGAs, this is an inefficient hardware implementation.

Streaming is an optimization that flattens a vector datapath to either a scalar or a smaller sized vector datapath. The idea is to serialize the execution of parallel hardware, so that resources can be shared and the vector data can be time-multiplexed over the shared resources.

Consider the following example model that operates on a 24-element vector datapath. This model contains 3 vector gains and 2 vector adds, resulting in a hardware implementation containing 72 multipliers and 24 adders. This can be confirmed by generating the resource utilization report when generating HDL code.

```
bdclose all;
load_system('hdl_areaopt1');
open_system('hdl_areaopt1/Controller');
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 0);
hdlset_param('hdl_areaopt1', 'ResourceReport', 'on');
makehdl('hdl_areaopt1/Controller');

### Generating HDL for 'hdl_areaopt1/Controller'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', ->
### Starting HDL check.
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Streaming to Scalarize the Datapath

An efficient area implementation of the same model can be realized by setting a positive integer value to the 'StreamingFactor' implementation parameter on the subsystem. This parameter specifies the

extent to which the datapath is scalarized - the higher the value, the greater the area savings. In this example, we have a 24-element vector datapath; to fully scalarize it, specify a 'StreamingFactor' value of 24. This can be done either through the HDL block properties dialog (opened by right-clicking on the 'Controller' subsystem) or through the command 'hdlset_param'.

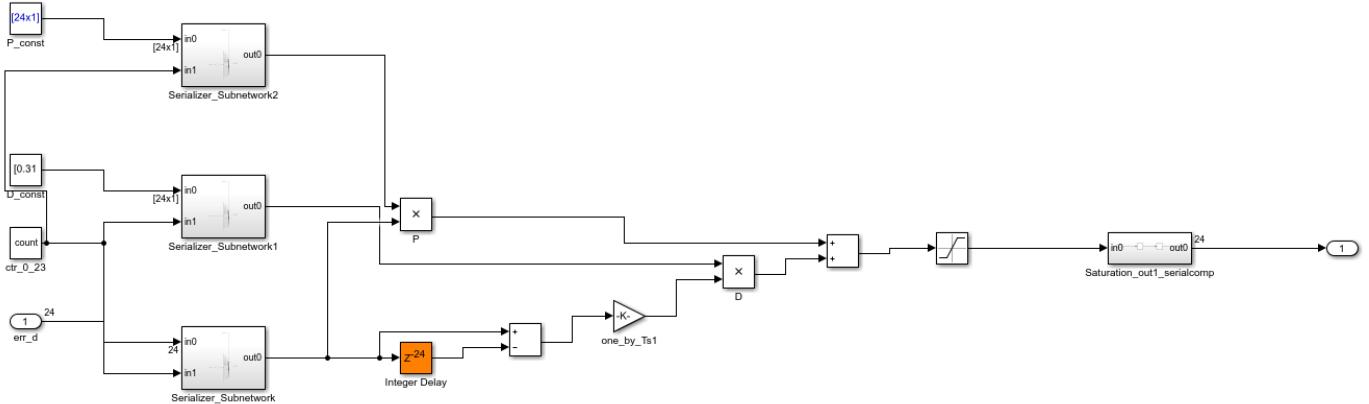
Generating HDL code with 'StreamingFactor' set to 24, generates HDL that uses only 3 multipliers and 2 adders (see the resource report after HDL code generation). The streamed architecture is implemented as local multi-rate or in single-rate mode depending on the context of the subsystem being streamed. If the subsystem logic is operating at a slower sample rate or if the "Oversampling factor" on page 17-15 is set to a value greater than one, then clock-rate pipelining kicks in and a streamed subsystem is implemented as a multi-cycle, single-rate architecture. See "Single-rate Resource Sharing Architecture" on page 24-50 for more details. In all other cases, a local multi-rate implementation is created, as described in this example. The elements of the vector datapath is streamed at a faster rate (in this case 24 times faster and denoted in red) and all computations operate on a scalar datapath. At the outputs, the vector is reconstructed using a tapped delay and the output is sampled back at the slower rate (in green).

```

hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 24);
hdlset_param('hdl_areaopt1', 'GenerateValidationModel', 'on');
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
%set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');

### Generating HDL for 'hdl_areaopt1/Controller'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', ->
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional
### Output port 0: 1 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdl_areaopt1_vnl')">gm_hdl_areaopt1_vnl
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### MESSAGE: The design requires 24 times faster clock with respect to the base rate = 2.
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\temp\hdl_areaopt1\report.html')">report.html
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\temp\hdl_areaopt1\check_report.html
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.

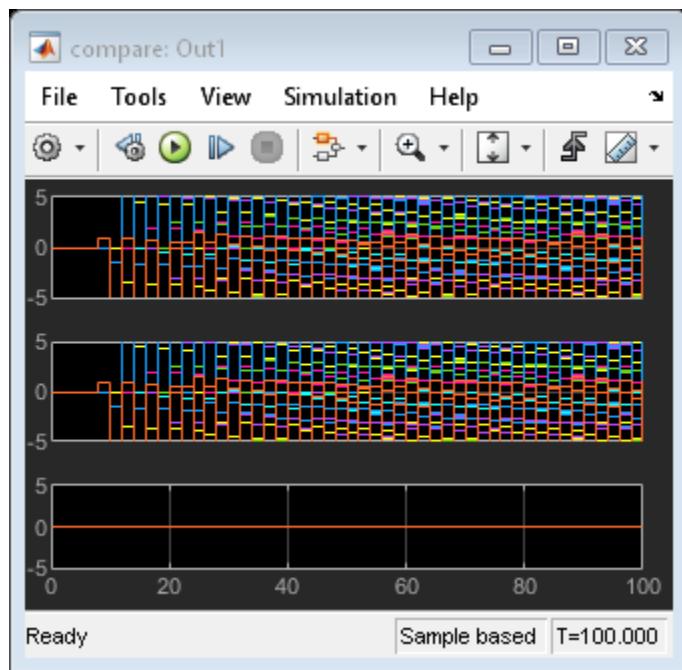
```



Delay Balancing and Functional Equivalence

The rate transitions that implement time-multiplexing in the streaming architecture introduce a cycle of additional latency. To maintain functional fidelity, this delay must be balanced across all cut-sets that this path is a member of. When the streaming option is turned on, the coder automatically also turns on the delay balancing option ('BalanceDelays') to automatically balance this additional delay. The coder also automatically turns on the validation model generation option so the user can verify that functional equivalence is maintained with respect to the original model.

```
sim('gm_hdl_areaopt1_vnl');
open_system('gm_hdl_areaopt1_vnl/Compare/Assert_Out1/compare: Out1')
```



Parameterizability for More Flexibility

By tuning the 'StreamingFactor' parameter, one can explore the design space along the datapath size dimension. A value of 1 implies no streaming (or fully parallel implementation), and a value of 24 (or the full vector length) implies maximal streaming (or fully serial implementation). By picking values between these two extremes, one can explore the design space from fully parallel to fully serial implementations.

If we set 'StreamingFactor' to 6 in this example model, we get a 4-element vector datapath in the generated HDL. This results in the use of 12 multipliers and 8 adders as shown in the resource report.

```
hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 6);
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
%set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');

### Generating HDL for 'hdl_areaopt1/Controller'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', ...
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit...
```

```
### Output port 0: 1 cycles.  
### Generating new validation model: <a href="matlab:open_system('gm_hdl_areaopt1_vnl')">gm_hdl_a  
### Validation model generation complete.  
### Begin VHDL Code Generation for 'hdl_areaopt1'.  
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 2.  
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.  
### Working on hdl_areaopt1/Controller as hdlsrc\hdl_areaopt1\Controller.vhd.  
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.  
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl.html')">C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl.html  
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl.html  
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.  
### HDL code generation complete.
```

Resource Sharing For Area Optimization

This example shows how to use the subsystem level sharing optimization in HDL Coder.

Introduction

Sharing is a subsystem-level optimization supported by HDL Coder for implementing area-efficient hardware.

By default, the coder implements hardware that is a 1-to-1 mapping of Simulink blocks to hardware module implementations. The resource sharing optimization enables users to share hardware resources by enabling an N-to-1 mapping of 'N' functionally-equivalent Simulink blocks to a single hardware module. The user specifies 'N' using the 'SharingFactor' implementation parameter.

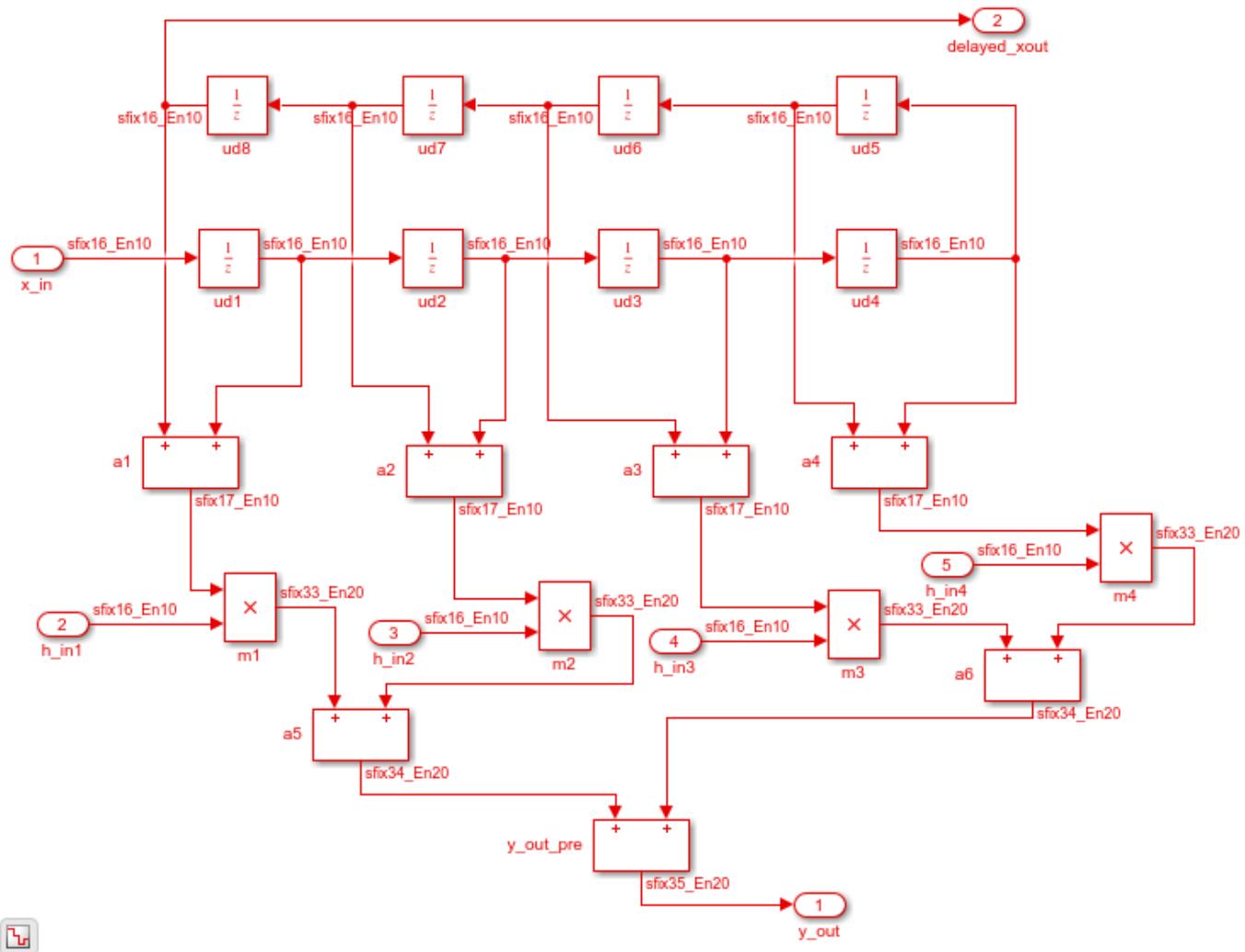
Since a time-multiplexed architecture incurs a longer latency to complete the operation, HDL Coder™ automatically manages the timing discrepancy depending on what resources are being shared.

Suppose that the shared resources are operating at the base sample rate, then resource sharing is implemented as a local multi-rate architecture, which is described in this example. If the shared resources are operating at a slower sample rate than the base sample rate, then HDL Coder™ invokes clock-rate pipelining to synthesize an implementation that utilizes the latency budget defined in the rate differential. In this case, the resource shared architecture is a single rate implementation and takes multiple time steps to complete all the shared operations. The "Single-rate Resource Sharing Architecture" on page 24-50 describes the details of this implementation.

The rest of this example illustrates the local multi-rate architecture of resource sharing. Consider the following symmetric FIR filter model. It contains 4 product blocks that are functionally equivalent and which map to 4 multipliers in hardware. The Resource Utilization Report lists the hardware resources used.

```
bdclose all;
load_system('sfir_fixed');
open_system('sfir_fixed/symmetric_fir');
hdlset_param('sfir_fixed', 'ResourceReport', 'on');
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 hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Sharing to Realize an N-to-1 Mapping

To reduce area resources, you can invoke the sharing optimization by setting the 'SharingFactor' parameter on the subsystem to a positive integer value. This parameter specifies 'N' in the N-to-1 hardware mapping. In this example, there are 4 product blocks, so generating HDL with 'SharingFactor' set to 4 generates HDL with 1 multiplier.

The code generation model reflects the sharing architecture. The inputs to the shared blocks are time-multiplexed over the shared resource at a faster rate (in this case 4x faster, shown in red). The outputs are then routed to the respective consumers at a slower rate (shown in green).

```

hdlset_param('sfir_fixed/symmetric_fir', 'SharingFactor', 4);
hdlset_param('sfir_fixed', 'GenerateValidationModel', 'on');
makehdl('sfir_fixed/symmetric_fir');
open_system('gm_sfir_fixed/symmetric_fir');
set_param('gm_sfir_fixed', 'SimulationCommand', 'update');

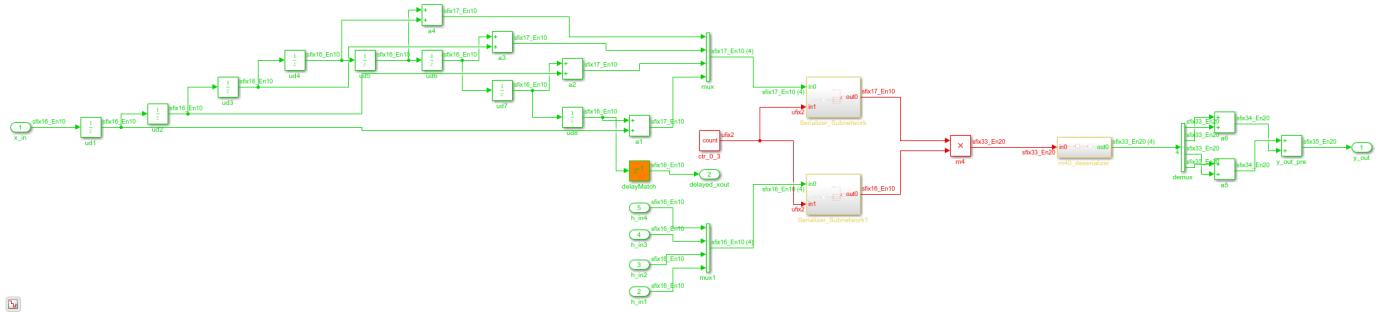
### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Starting HDL check.

```

```

### The DUT requires an initial pipeline setup latency. Each output port experiences these additional latencies:
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_sfir_fixed_vnl')">gm_sfir_fixed_vnl</a>
### Validation model generation complete.
### Begin VHDL Code Generation for 'sfir_fixed'.
### MESSAGE: The design requires 4 times faster clock with respect to the base rate = 1.
### Working on symmetric_fir_tc as hdlsrc\sfir_fixed\symmetric_fir_tc.vhd.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tphdlsrc\sfir_fixed\symmetric_fir.html')">here</a>
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.

```



The sharing optimization is implemented using time-division multiplexing. Simulink requires the outputs of the shared resource to be sampled at the predefined sample rate, so HDL Coder overclocks the shared resource at a faster rate than the data rate. In the above example, the shared architecture, which includes the shared resource, multiplexer-serializer at the inputs and demultiplexer-deserializer at the outputs, operates at 4 times the rate of the input data, because 'Sharingfactor' = 4.

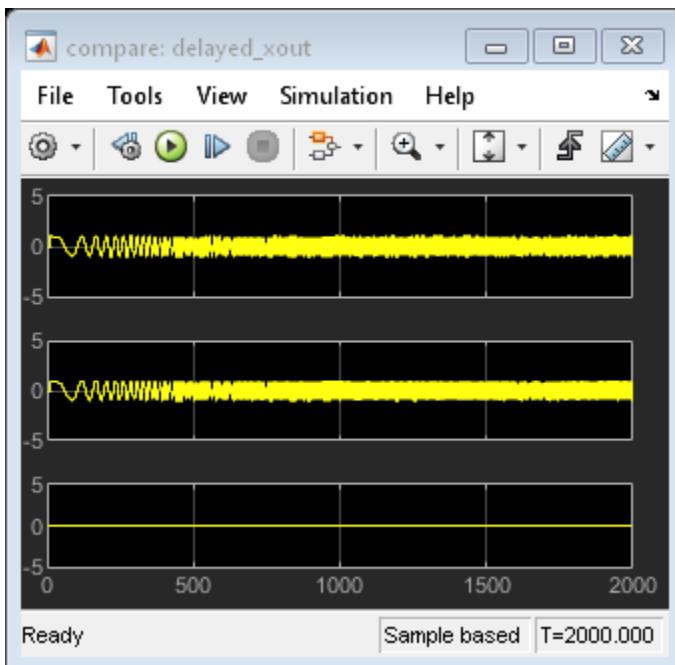
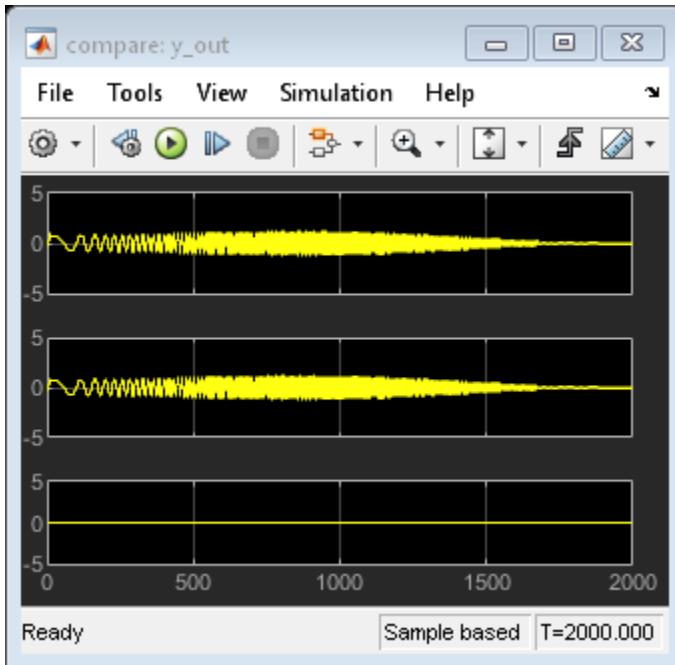
Delay Balancing and Functional Equivalence

The rate transitions that implement time-multiplexing in the resource sharing architecture introduce additional latency. To maintain functional equivalence, delay balancing automatically inserts matching delays in parallel merging paths. The generated validation model allows the user to verify functional equivalence by comparing the operation of the shared hardware architecture with the original model.

```

sim('gm_sfir_fixed_vnl');
open_system('gm_sfir_fixed_vnl/Compare/Assert_y_out/compare: y_out')
open_system('gm_sfir_fixed_vnl/Compare/Assert_delayed_xout/compare: delayed_xout')

```



Control Multiplicative Oversampling through SharingFactor

The net oversampling for the whole design is equivalent to the LCM of all 'SharingFactor' values set on the model. Consider the example, `hdlcoder_uniform_oversampling.slx`. It has two subsystems: subsystem 'Share3' has 3 gain blocks that can be shared and 'Share4' has 4 gain blocks that can be shared.

```
saved_warning_state = warning('off', 'hdlcoder:makehdl:DeprecateMaxOverSampling');
warning('off', 'hdlcoder:makehdl:DeprecateMaxComputationLatency');
```

```

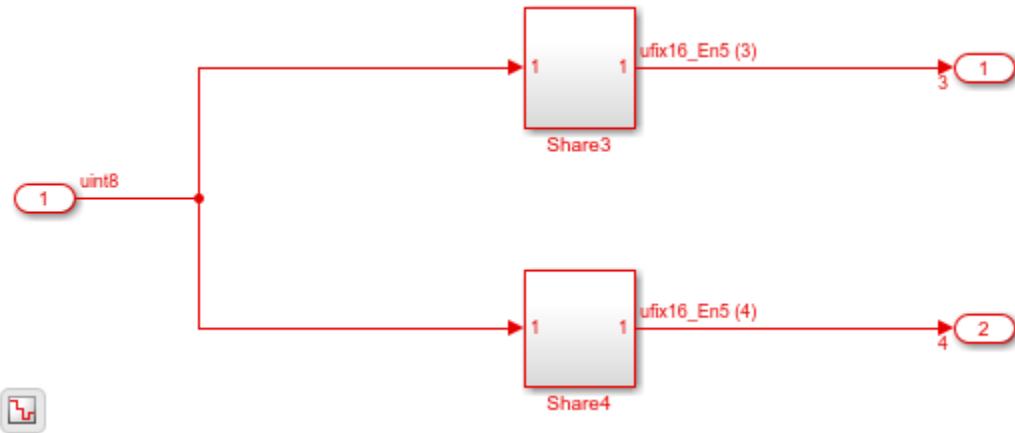
bdclose('all');
load_system('hdlcoder_uniform_oversampling');
open_system('hdlcoder_uniform_oversampling/Subsystem');
set_param('hdlcoder_uniform_oversampling', 'SimulationCommand', 'update');
hdlsaveparams('hdlcoder_uniform_oversampling/Subsystem');

% Set Model 'hdlcoder_uniform_oversampling' HDL parameters
hdlset_param('hdlcoder_uniform_oversampling', 'GenerateValidationModel', 'on');
hdlset_param('hdlcoder_uniform_oversampling', 'HDLSubsystem', 'hdlcoder_uniform_oversampling');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share3', 'SharingFactor', 3);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share4', 'SharingFactor', 4);

```



Notice that 'Share3' sets its 'SharingFactor' to 3 and 'Share4' sets its 'SharingFactor' to 4. HDL Coder™ applies local resource sharing to each subsystem and as a result, the HDL implementation requires $\text{LCM}(3, 4) = 12x$ oversampling. This is reported in the message during HDL code generation.

```

makehdl('hdlcoder_uniform_oversampling/Subsystem');

### Generating HDL for 'hdlcoder_uniform_oversampling/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_uniform...
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit...
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_uniform_oversampli...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_uniform_oversampling'.
### MESSAGE: The design requires 12 times faster clock with respect to the base rate = 0.1.
### Working on hdlcoder_uniform_oversampling/Subsystem/Share3 as hdlsrc\hdlcoder_uniform_oversamp...
### Working on hdlcoder_uniform_oversampling/Subsystem/Share4 as hdlsrc\hdlcoder_uniform_oversamp...
### Working on Subsystem_tc as hdlsrc\hdlcoder_uniform_oversampling\Subsystem_tc.vhd.
### Working on hdlcoder_uniform_oversampling/Subsystem as hdlsrc\hdlcoder_uniform_oversampling\S...
### Generating package file hdlsrc\hdlcoder_uniform_oversampling\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file:/C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpe...
### HDL check for 'hdlcoder_uniform_oversampling' complete with 0 errors, 0 warnings, and 1 messa...
### HDL code generation complete.

```

One way to circumvent this multiplicative effect of oversampling is to set the 'SharingFactor' of all subsystems to the available oversampling budget. In the above example, if the oversampling budget is only 4x, then set 'SharingFactor' = 4 for both 'Share3' and 'Share4'. In this case, HDL Coder can share fewer resources than the SharingFactor and stay idle for the remaining cycles.

```
hdlset_param('hdlcoder_uniform_oversampling/Subsystem/Share3', 'SharingFactor', 4);
makehdl('hdlcoder_uniform_oversampling/Subsystem');

### Generating HDL for 'hdlcoder_uniform_oversampling/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_uniform...
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit...
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_uniform_oversampli...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_uniform_oversampling'.
### MESSAGE: The design requires 4 times faster clock with respect to the base rate = 0.1.
### Working on hdlcoder_uniform_oversampling/Subsystem/Share3 as hlsrc\hdlcoder_uniform_oversamp...
### Working on hdlcoder_uniform_oversampling/Subsystem/Share4 as hlsrc\hdlcoder_uniform_oversamp...
### Working on Subsystem_tc as hlsrc\hdlcoder_uniform_oversampling\Subsystem_tc.vhd.
### Working on hdlcoder_uniform_oversampling/Subsystem as hlsrc\hdlcoder_uniform_oversampling\S...
### Generating package file hlsrc\hdlcoder_uniform_oversampling\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_uniform_oversampling' complete with 0 errors, 0 warnings, and 1 messa...
### HDL code generation complete.
```

Notice that the oversampling factor is now $\text{LCM}(4,4) = 4$, and that this is the value reported during code generation. In general, it is a good idea to set the 'SharingFactor' values to the available oversampling budget. If your design contains fewer shareable resources than the 'SharingFactor' value you specify, HDL Coder shares the shareable resources available, and overclocks them by the 'SharingFactor' value. However, if you want to apply both resource sharing and other optimizations that uses overclocking, such as streaming, or apply resource sharing in multiple nested subsystems, this general guideline may result in a higher oversampling factor.

Block Support, Atomic Subsystems, and Extensions

HDL Coder supports resource sharing of 4 block types: Product, Gain, Atomic Subsystem, and MATLAB Function. For MATLAB Function blocks, use the **MATLAB Datapath** architecture with fixed-point types. This architecture is the default setting for floating-point types. You can specify the **HDL Architecture** in the HDL Block Properties dialog box of the MATLAB Function block.

Sharing functionally equivalent Product and Gain blocks means that the multipliers in the HDL implementation will be shared. Two Product blocks are functionally equivalent if: a) the data types of their inputs and outputs are identical, b) their block parameter settings are identical, and c) their HDL block properties are identical. For a Gain block, functional equivalence additionally requires that the constant value and data types are also identical. However, if the gain constant data types are identical for two Gain blocks with different gain constant values, HDL Coder can share them. Similarly, if a Gain block can be implemented as a Product block with constant input, and it has the same data types as another Product block in the design, the coder can share them.

The third block type, Atomic Subsystem, is useful for sharing functionally equivalent islands of logic encapsulated inside atomic subsystems. Two atomic subsystems are functionally equivalent and can be shared if:

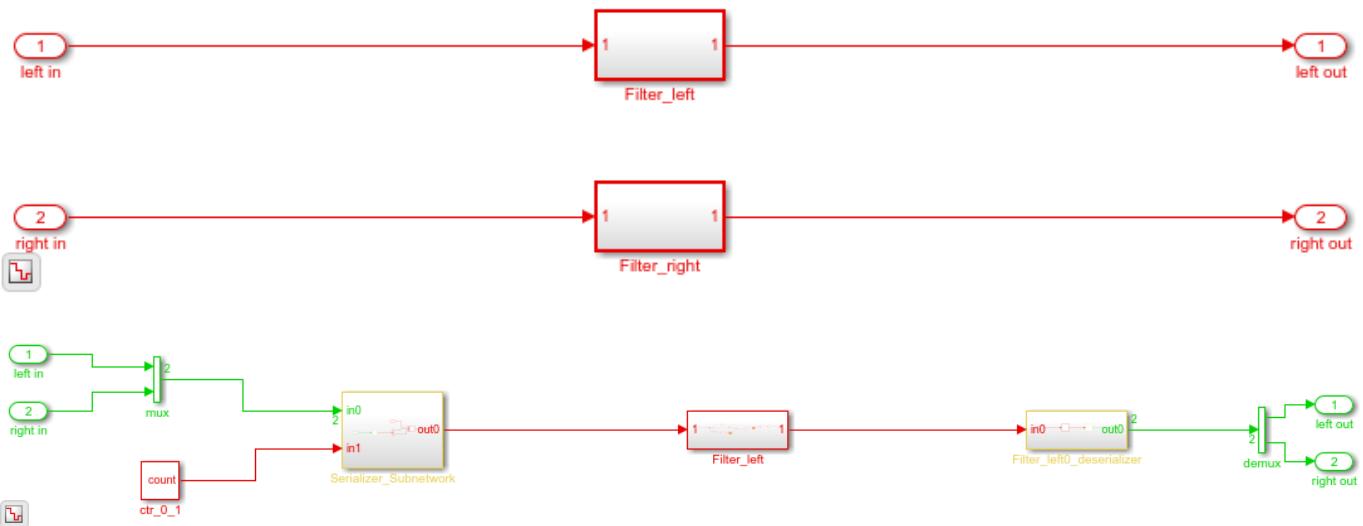
- Their Simulink checksums are identical
- Their HDL block properties are identical.

Sharing Atomic Subsystems

```
% The following example demonstrates an audio filtering model that applies
% the same filter on the left and right channels. By default, HDL Coder
% generates two filter modules in hardware.
bdclose all;
load_system('hdlcoder_audiofiltering');
open_system('hdlcoder_audiofiltering/Audio filter');

% The filters on the two audio channels can be shared by specifying a
% 'SharingFactor' value of 2 on the encompassing subsystem. This generates
% an architecture that uses only one filter, as shown below.
hdlset_param('hdlcoder_audiofiltering/Audio filter', 'SharingFactor', 2);
makehdl('hdlcoder_audiofiltering/Audio filter');
open_system('gm_hdlcoder_audiofiltering/Audio filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');

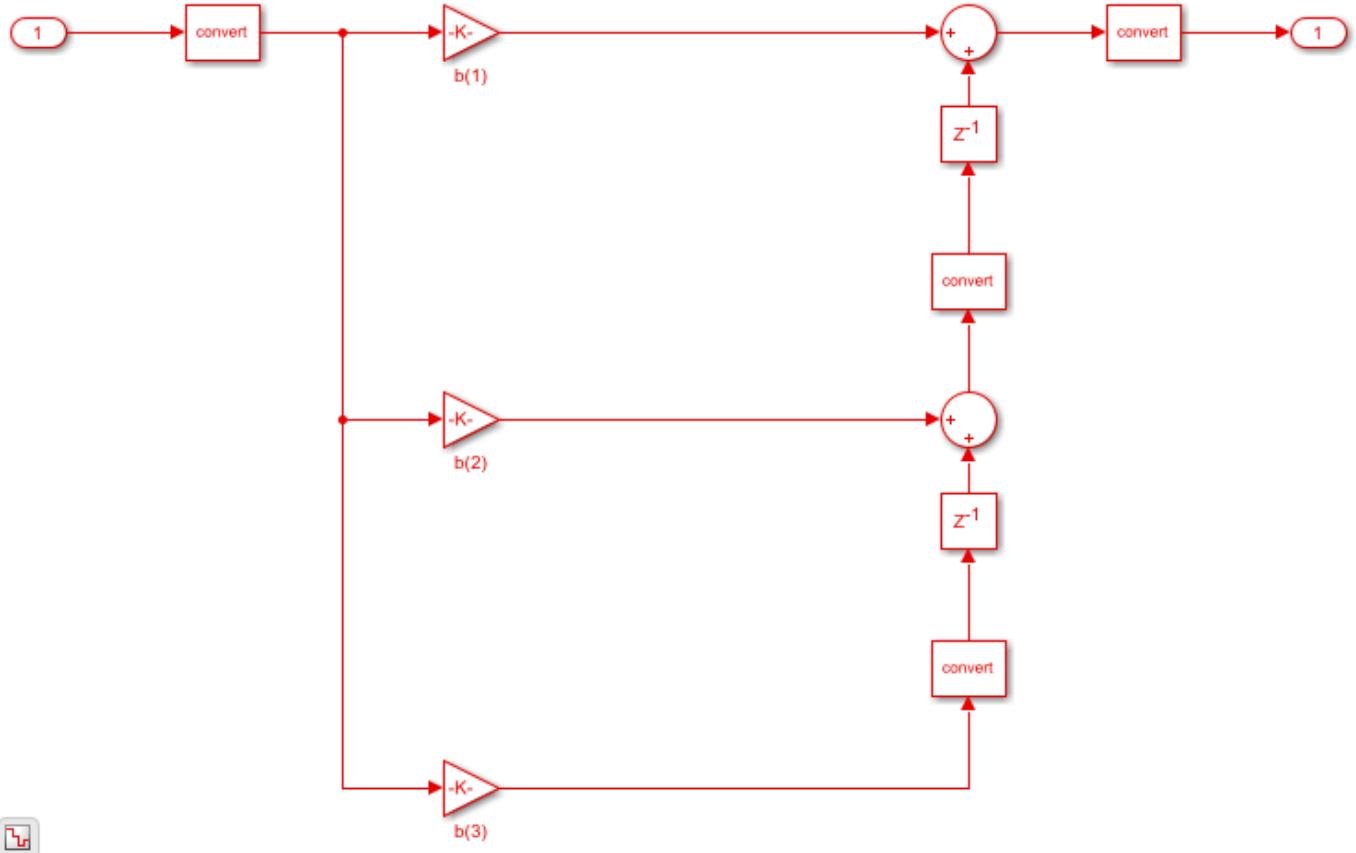
### Generating HDL for 'hdlcoder_audiofiltering/Audio filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiof
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0.00012207
### Working on hdlcoder_audiofiltering/Filter_left as hdsrc\hdlcoder_audiofiltering\
### Working on Audio_filter_tc as hdsrc\hdlcoder_audiofiltering\Audio_filter_tc.vhd.
### Working on hdlcoder_audiofiltering/Filter_right as hdsrc\hdlcoder_audiofiltering\Filter_right.vhd.
### Generating package file hdsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpe
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```



Opportunities Across Hierarchies

Since 'SharingFactor' is a subsystem-level parameter, different subsystems at different levels of the hierarchy can specify different sharing values. In the audio filter example, the filters on each channel use 3 gain blocks respectively.

```
open_system('hdlcoder_audiofiltering/Audio_filter/Filter_left');
```



Sharing at Higher Level of Hierarchy

We can specify a 'SharingFactor' value of 2 at the top-level of the DUT to share the filters on the two channels. Additionally, we can specify a 'SharingFactor' value of 3 on each of the filter subsystems to enable sharing of the 3 gain blocks in each channel. When HDL code is now generated, notice first that the left and right filters have been shared and we have only one filter at the top-level of the hierarchy.

```
hdlset_param('hdlcoder_audiofiltering/Audio_filter', 'SharingFactor', 2);
hdlset_param('hdlcoder_audiofiltering/Audio_filter/Filter_left', 'SharingFactor', 3);
hdlset_param('hdlcoder_audiofiltering/Audio_filter/Filter_right', 'SharingFactor', 3);
makehdl('hdlcoder_audiofiltering/Audio_filter');
open_system('gm_hdlcoder_audiofiltering/Audio_filter');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');

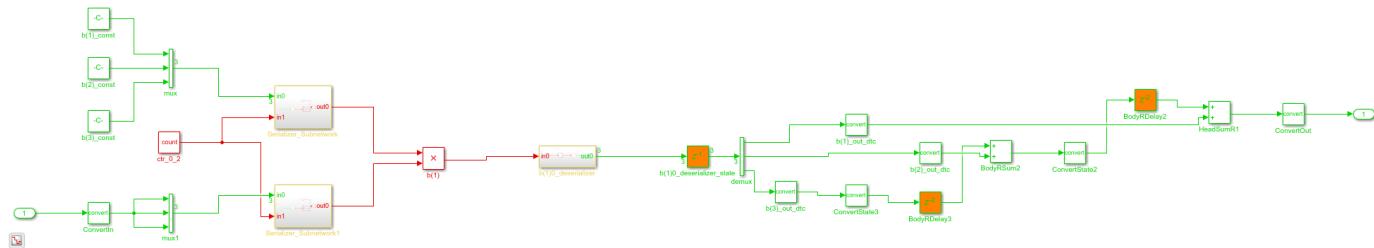
### Generating HDL for 'hdlcoder_audiofiltering/Audio_filter'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_audiofiltering')">.
### Starting HDL check.
```

```
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional latencies:
### Output port 0: 2 cycles.
### Output port 1: 2 cycles.
### Begin VHDL Code Generation for 'hdlcoder_audiofiltering'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 0.00012207.
### Working on hdlcoder_audiofiltering/Filter_left as hdsrsrc\hdlcoder_audiofiltering\Filter_left.vhd.
### Working on Audio filter_tc as hdsrsrc\hdlcoder_audiofiltering\Audio_filter_tc.vhd.
### Working on hdlcoder_audiofiltering/Filter as hdsrsrc\hdlcoder_audiofiltering\Filter.vhd.
### Generating package file hdsrsrc\hdlcoder_audiofiltering\Audio_filter_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpt...
### HDL check for 'hdlcoder_audiofiltering' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.
```

Sharing at Lower Level of Hierarchy

Additionally, the 3 gain blocks in the lower of the hierarchy (within the filter subsystem) have also been shared. The net result is that we have reduced the resource usage from 6 multipliers to just one.

```
open_system('gm_hdlcoder_audiofiltering/Filter_left');
set_param('gm_hdlcoder_audiofiltering', 'SimulationCommand', 'update');
```

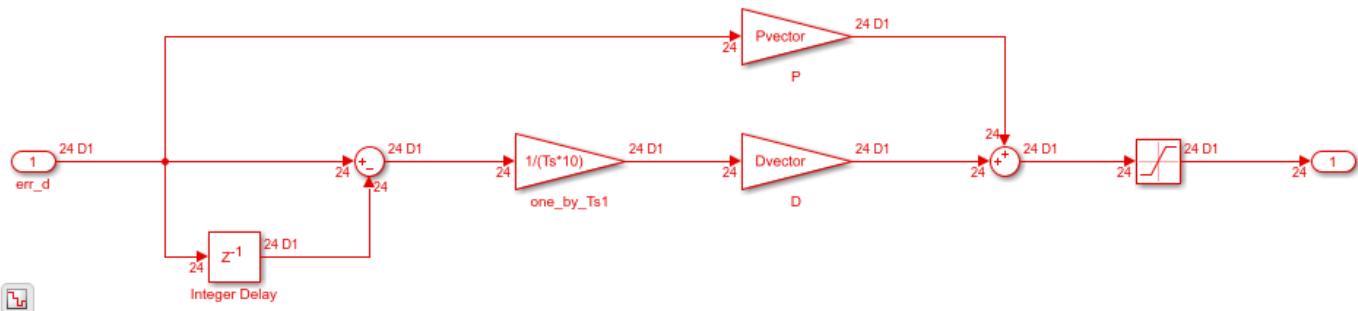


Combining Optimizations and Sharing

Resource sharing can also be combined with other optimizations such as the streaming optimization.

Consider the model below: it contains a 24-element vector datapath and 3 vector gain blocks, which will map to 72 multipliers, by default. Streaming can scalarize the vector datapath while sharing can share the 3 Gain blocks.

```
bdclose all;
load_system('hdl_areaopt1');
open_system('hdl_areaopt1/Controller');
set_param('hdl_areaopt1', 'SimulationCommand', 'update');
```



Streaming and Sharing Reduce the Design to Use One Multiplier

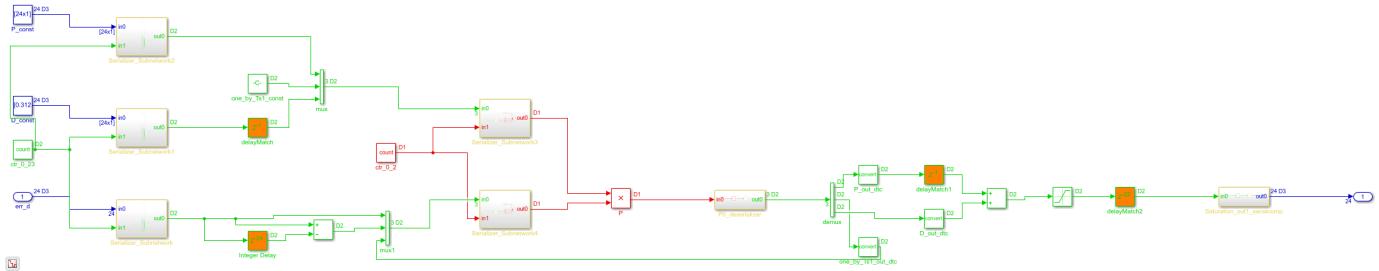
To invoke both optimizations, we set 'StreamingFactor' to 24 and 'SharingFactor' to 3. The former will reduce the number of multipliers from 72 to 3, and the latter reduces the 3 scalar multipliers to 1.

```

hdlset_param('hdl_areaopt1/Controller', 'StreamingFactor', 24);
hdlset_param('hdl_areaopt1/Controller', 'SharingFactor', 3);
makehdl('hdl_areaopt1/Controller');
open_system('gm_hdl_areaopt1/Controller');
set_param('gm_hdl_areaopt1', 'SimulationCommand', 'update');

### Generating HDL for 'hdl_areaopt1/Controller'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_areaopt1', ...
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional ...
### Output port 0: 2 cycles.
### Begin VHDL Code Generation for 'hdl_areaopt1'.
### MESSAGE: The design requires 72 times faster clock with respect to the base rate = 2.
### Working on Controller_tc as hdlsrc\hdl_areaopt1\Controller_tc.vhd.
### Working on hdl_areaopt1\Controller as hdlsrc\hdl_areaopt1\Controller.vhd.
### Generating package file hdlsrc\hdl_areaopt1\Controller_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tp...
### HDL check for 'hdl_areaopt1' complete with 0 errors, 0 warnings, and 1 messages.
### HDL code generation complete.

```

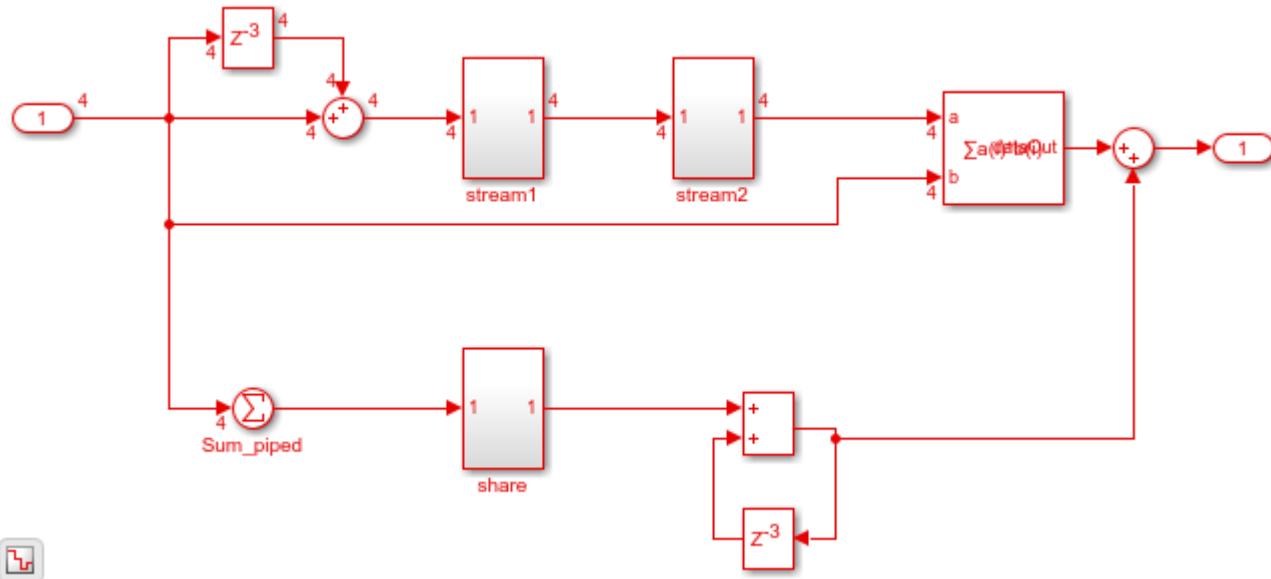


Single-rate Resource Sharing Architecture

This example shows how HDL Coder™ manages the execution of operations in the context of clock rate pipelining. By default, if resource sharing is applied in a region of the design operating at the fastest base sample rate, then a local multi-rate architecture is synthesized, as described in “Resource Sharing For Area Optimization” on page 24-40. If the shared resources are operating at a slower sample rate and clock rate pipelining is enabled, then the code generator synthesizes a single-rate architecture, which is described in this example.

Clock rate pipelining is an optimization that finds islands of logic in the Simulink design that operates on data at a slower sample rate and inserts pipelining and resource sharing logic at the (faster) clock rate. In these cases, resource sharing is implemented as a time multiplexed architecture that operates at a single rate and incurs a latency. In order to orchestrate execution of dependent operations and manage the additional latency introduced, HDL Coder (TM) synthesizes appropriate scheduling logic. Consider the model, `hdlcoder_singlerate_sharing.slx`.

```
bdclose('all');
load_system('hdlcoder_singlerate_sharing');
open_system('hdlcoder_singlerate_sharing/Subsystem');
set_param('hdlcoder_singlerate_sharing', 'SimulationCommand', 'update');
```



This model has an oversampling constraint set through the “Oversampling factor” on page 17-15, which specifies how much faster the FPGA clock rate runs with respect to the Simulink base sample time. This model sets Oversampling = 30, which essentially means that the clock-rate pipelined region can consume 30 clock cycles to complete execution.

The optimization options for individual blocks and subsystems are listed below. Let's generate code and inspect the validation model to understand the single-rate sharing architecture.

```
hdlsaveparams('hdlcoder_singlerate_sharing/Subsystem');
makehdl('hdlcoder_singlerate_sharing/Subsystem');

%% Set Model 'hdlcoder_singlerate_sharing' HDL parameters
hdlset_param('hdlcoder_singlerate_sharing', 'GenerateValidationModel', 'on');
```

```

hdlset_param('hdlcoder_singlerate_sharing', 'HDLSubsystem', 'hdlcoder_singlerate_sharing');
hdlset_param('hdlcoder_singlerate_sharing', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_singlerate_sharing', 'Oversampling', 30);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem', 'DistributedPipelining', 'on');

hdlset_param('hdlcoder_singlerate_sharing/Subsystem/Sum_piped', 'Architecture', 'Tree');
% Set Sum HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/Sum_piped', 'OutputPipeline', 2);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/share', 'SharingFactor', 2);

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream1', 'StreamingFactor', 2);

% Set Delay HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream1/Delay', 'UseRAM', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_singlerate_sharing/Subsystem/stream2', 'StreamingFactor', 4);

### Generating HDL for 'hdlcoder_singlerate_sharing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_singl...
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additi...
### Output port 0: 1 cycles.
### Clock-rate pipelining results can be diagnosed by running this script: <a href="matlab:run('...
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: <a ...
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcod...
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_singlerate_sharing...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_singlerate_sharing'.
### MESSAGE: The design requires 30 times faster clock with respect to the base rate = 0.1.
### Working on crp_temp_shared as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_shared.vhd.
### Working on hdlcoder_singlerate_sharing/Subsystem/share as hdlsrc\hdlcoder_singlerate_sharin...
### Working on crp_temp_streamed as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_streamed.vhd.
### Working on crp_temp_streamed_block as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_streamed_b...
### Working on hdlcoder_singlerate_sharing/Subsystem/stream1 as hdlsrc\hdlcoder_singlerate_sharin...
### Working on crp_temp_streamed_block1 as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_streamed_b...
### Working on crp_temp_streamed_block2 as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_streamed_b...
### Working on hdlcoder_singlerate_sharing/Subsystem/stream2 as hdlsrc\hdlcoder_singlerate_sharin...
### Working on crp_temp_MAC as hdlsrc\hdlcoder_singlerate_sharing\crp_temp_MAC.vhd.
### Working on Subsystem_tc as hdlsrc\hdlcoder_singlerate_sharing\Subsystem_tc.vhd.
### Working on hdlcoder_singlerate_sharing/Subsystem as hdlsrc\hdlcoder_singlerate_sharing\Subsy...
### Generating package file hdlsrc\hdlcoder_singlerate_sharing\Subsystem_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_152...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_singlerate_sharing' complete with 0 errors, 3 warnings, and 5 message...
### HDL code generation complete.

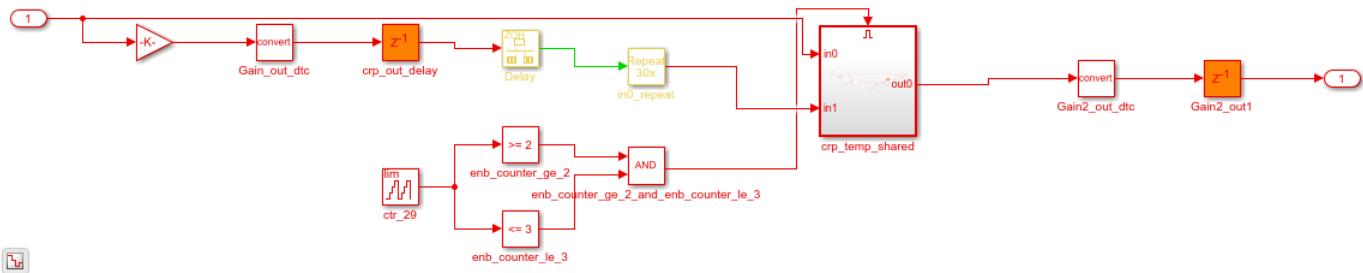
```

At the global level, the coder schedules each of these locally shared and streamed subsystems according to their latency. The unit of scheduling is a clock-rate pipelined region that has been automatically identified by the coder. For each such region, a simple counter block is used as a sequencer for the scheduling logic. The counter counts from zero to (clock-rate budget - 1), where the budget is defined as the ratio of the shared resource sample rate to the FPGA clock rate. In this

example, the budget is 30 because we set Oversampling = 30. The code generator assigns a time interval within which each streamed and shared subsystem executes: specifically, the subsystem itself is encapsulated within an enabled subsystem so that it is only active during that time interval. The counter or sequencer value specifies the current time step, and logic that computes the time interval drives the enable inputs to these subsystems.

For each subsystem that specifies resource sharing or streaming, a single-rate resource-shared architecture implements the time division multiplexing. For example, see 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share'. If SharingFactor = N, it takes (N-1) cycles to execute the shared architecture per cycle of the original computation.

```
open_system('gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share');
set_param('gm_hdlcoder_singlerate_sharing_vnl', 'SimulationCommand', 'update');
```



Notice that 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share' is assigned the time interval [2, 3]. This is because the sum-of-elements block, 'hdlcoder_singlerate_sharing/Subsystem/Sum_piped', with OutputPipeline = 2, is on the path between the DUT inputs and the inputs to this subsystem. The shared subsystem starts execution in time step 2, and, since SharingFactor=3, takes (3-1 = 2) cycles to complete. The enable input to 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/share/crp_temp_shared' is asserted only when the global counter is greater than or equal to 2 or lesser than or equal to 3.

In addition to streamed and shared subsystems, the code generator also schedules any blocks or subsystems containing state or implement multi-cycle operations. For example, the design uses a multiply-accumulate block, which computes the dot-product on two 4-element vectors (see 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/crp_temp_MAC'). This takes 4 cycles to execute and is scheduled in the time interval [4, 7]. This is because there are two streaming regions on the path from the inputs to this multiply-accumulate block. The first streaming region, 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/stream1' is scheduled in time interval [0, 1] due to a streaming factor of 2 and the second streaming region, 'gm_hdlcoder_singlerate_sharing_vnl/Subsystem/stream2', is scheduled in time interval [1, 4] due to a streaming factor of 4.

The generated validation model has non-trivial changes but precisely captures the essence of the single-rate sharing architecture that has been synthesized. This model also compares the numerics of this synthesized architecture with the original model modulo added latency. For more details, see "Delay Balancing and Validation Model Workflow In HDL Coder™" on page 24-68. Running the validation model, by pressing the play button, will compare the numerics between the synthesized and the original model in each time steps and will throw an assertion on mismatches.

Improve Resource Sharing with Design Modifications

This example shows how to improve opportunities for resource sharing to optimize your model design by making certain modifications to your design. Resource sharing is an HDL Coder optimization that improves area utilization in the design on the target FPGA device. The optimization identifies multiple functionally equivalent resources and replaces them with a single resource. For more information, see “Resource Sharing” on page 24-32.

Analyze the Current Model

Execute the following lines of code to copy the necessary example files into a temporary folder.

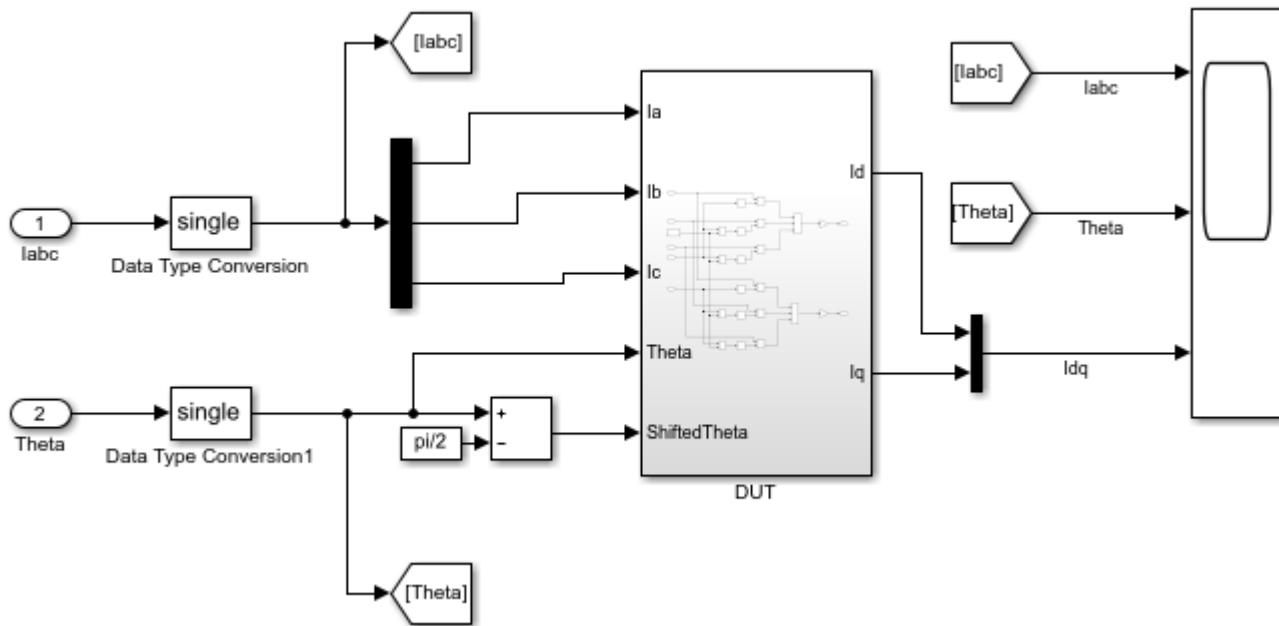
```
design_name = 'hdlcoderParkTransform';
design_new_name = 'hdlcoderParkTransformCopy';

hdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos');
hdlc_temp_dir = [tempdir 'hdlcoderParkTransformDir'];

% Create a temporary folder and copy the Simulink model.
cd(tempdir);
[~, ~, ~] = rmdir(hdlc_temp_dir, 's');
mkdir(hdlc_temp_dir);
cd(hdlc_temp_dir);

copyfile(fullfile(hdlc_demo_dir, [design_name, '.slx']), fullfile(hdlc_temp_dir, [design_new_name, '.slx']));

% Open the model.
open_system(design_new_name);
```



```
% Set a SharingFactor of 6 on the subsystem of interest and generate
% HDL along with the corresponding reports from the model.
subsystem = [design_new_name '/DUT'];
```

```

hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);

### Generating HDL for 'hdlcoderParkTransformCopy/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addition
### Output port 0: 43 cycles.
### Output port 1: 43 cycles.
### Begin VHDL Code Generation for 'hdlcoderParkTransformCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 1e-05.
### Working on hdlcoderParkTransformCopy/DUT/nfp_sin_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_add2_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on hdlcoderParkTransformCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTransfo
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DU_tc.vhd.
### Working on hdlcoderParkTransformCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DU.vhd.
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformCopy\DU_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd
### HDL check for 'hdlcoderParkTransformCopy' complete with 0 errors, 1 warnings, and 1 messages
### HDL code generation complete.

```

In the generated Streaming and Sharing report, notice that there are specific groups that were identified to be eligible for sharing:

Resource Type	Group Size	Block Name
Trigonometry	6	sine
Sum	2	AddSub2
Sum	2	AddSub1
Product	6	Product
Product	2	Gain

The Native Floating-Point Resource Report shows the operators needed:

Resource	Usage
Adders	4
Multipliers	2
Sin	1
Subtractors	2

The High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	9
Adders/Subtractors	152
Registers	1121
Total 1-Bit Registers	15896
RAMs	0
Multiplexers	857

I/O Bits	228
Static Shift operators	5
Dynamic Shift operators	13

You can also run synthesis workflow on the model. The synthesis results for this model are:

Resource	Usage
Slice LUTs	29381
Slice Registers	24140
DSPs	50
Block Ram Tile	0
URAM	0

For more details regarding synthesis workflow, see "HDL Code Generation and FPGA Synthesis from Simulink Model".

Detect Patterns in Model

Upon further analysis of the model, you can see that there are a few patterns that can be modified to improve resource sharing.

The first such pattern is with the Add and Subtract blocks whose names start with "AddSub" and whose output goes to Sin blocks.

The second pattern corresponds to a Sum (as an addition or subtraction) block followed by a Gain block.

In both of these patterns, there are some dissimilarities in the highlighted blocks that prevent resource sharing from happening optimally. The dissimilarities are as follows:

First, the signs of the Sum blocks in the region near the Gain blocks are different. Going from top to bottom, the signs are +++ and ---.

You can fix this inconsistency with the following changes:

- 1 Switch the signs of the bottom Sum block from --- to +++
- 2 Make the three bottom Sin blocks to Cos blocks.
- 3 Change the input to the Cos block and the two bottom Sum blocks from Import 5 "ShiftedTheta" to Import 4 "Theta".

Second, the signs of the Sum blocks in the different regions are different. Going from the top to bottom, the signs are +-, ++, +- and ++.

Right now, the blocks with both positive signs are being shared separately from the blocks with alternating signs, requiring more resources. You can simplify your model further to reduce the number of Sum blocks in your model while improving resource sharing. The input to the Sum blocks are the same, and you can reduce the number of Sum blocks you need to two. Then, the inconsistency between the signs is easily fixed by introducing a Unary Minus block.

The discrepancy is then resolved, and this model now shares more resources than the original.

This edited model is saved as `hdlcoderParkTransformShare.slx`. Copy this version of the model into the temporary directory using the following commands:

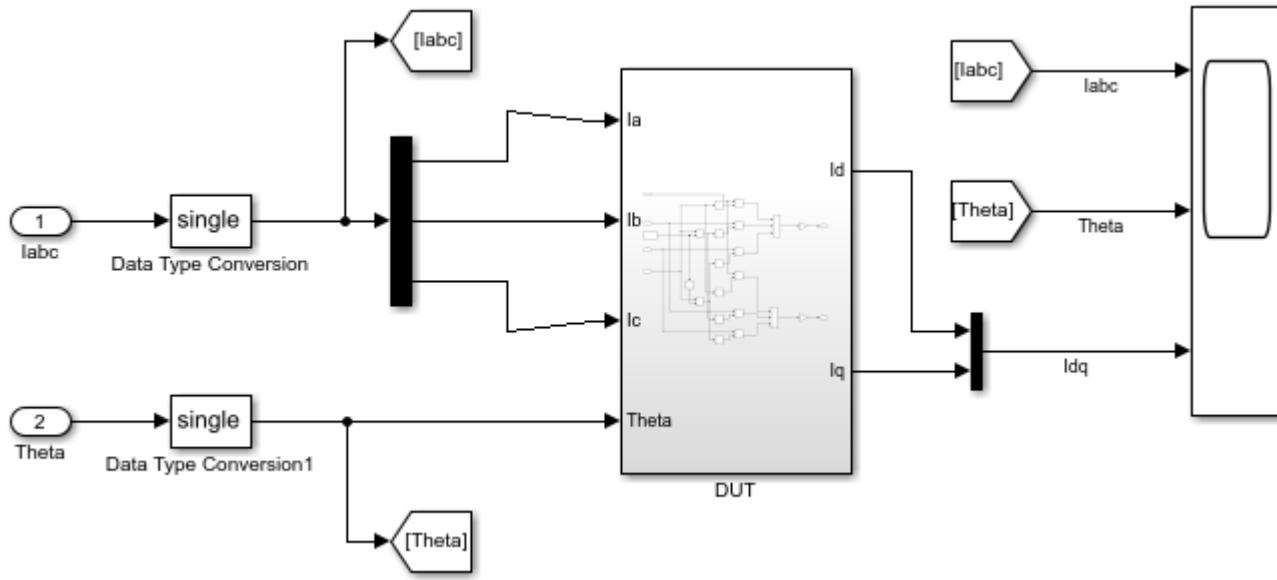
```
design_name = 'hdlcoderParkTransformShare';
design_new_name = 'hdlcoderParkTransformShareCopy';
```

```

copyfile(fullfile(hdlc_demo_dir, [design_name, '.slx']), fullfile(hdlc_temp_dir, [design_new_name

% Open the model.
open_system(design_new_name);

```



Compare Results from Optimized Model

Since this model has been improved for resource sharing, more blocks are shared and results in an improvement in resource utilization. Set a **SharingFactor** of 3 on the subsystem of interest and generate HDL along with the corresponding reports from the optimized model

```

subsystem = [design_new_name '/DUT'];

hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);

### Generating HDL for 'hdlcoderParkTransformShareCopy/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 0: 39 cycles.
### Output port 1: 39 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderParkTransformShareCo
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 6.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderPar
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_tc.vhd.
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCop
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_pkg.vhd.

```

```
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd'>
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd'
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 1 warnings, and 1 message
### HDL code generation complete.
```

After making those design modifications, the Native Floating-Point Resource Report shows the updated operators needed:

Resource	Usage
Adders	2
Multipliers	2
SinCos	1
Subtractors	1
Unary Minus	1

Notice that the **Adders**, **Multipliers**, and **Subtractors** count have reduced after sharing.

After sharing, the High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	13
Adders/Subtractors	178
Registers	1012
Total 1-Bit Registers	16888
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

After running synthesis on the model, you can see the following results:

Resource	Usage
Slice LUTs	8792
Slice Registers	8419
DSPs	13
Block Ram Tile	0
URAM	0

Notice that the **Slice LUTs**, **Slice Registers**, and **DSPs** counts have reduced after sharing.

To learn how you can further improve resource sharing with automatically replacement of recurring patterns using the Clone Detection application, see “[Improve Resource Sharing with Clone Detection and Replacement](#)” on page 24-58.

Improve Resource Sharing with Clone Detection and Replacement

This example shows how you can automatically identify and replace recurring patterns by using the Clone Detector App. Consequently, it also improves opportunities for optimizing the model by using resource sharing. The Clone Detector App refactors models by identifying clones in a design and replacing clones with links to subsystem blocks in a library. This example uses a model that has already been modified for optimal resource sharing by using the Clone Detector App. To learn more about these modifications, see "Improve Resource Sharing with Design Modifications."

Clone Detection

Clone Detection is a tool that can be used to identify modeling patterns in a design that are similar to a few sample patterns provided as inputs through a custom library file. Once the Clone Detector App identifies these patterns, it replaces the patterns with atomic subsystems. This enables the generation of optimal code through code reuse and results in better resource utilization through sharing of resources among the atomic subsystems. For more information regarding Clone Detection, please refer to "Enable Component Reuse by Using Clone Detection" (Simulink Check).

Set Up Model for Resource Sharing

This edited model is saved as `hdlcoderParkTransformShare.slx`. Copy this version of the model into the temporary directory using the following commands:

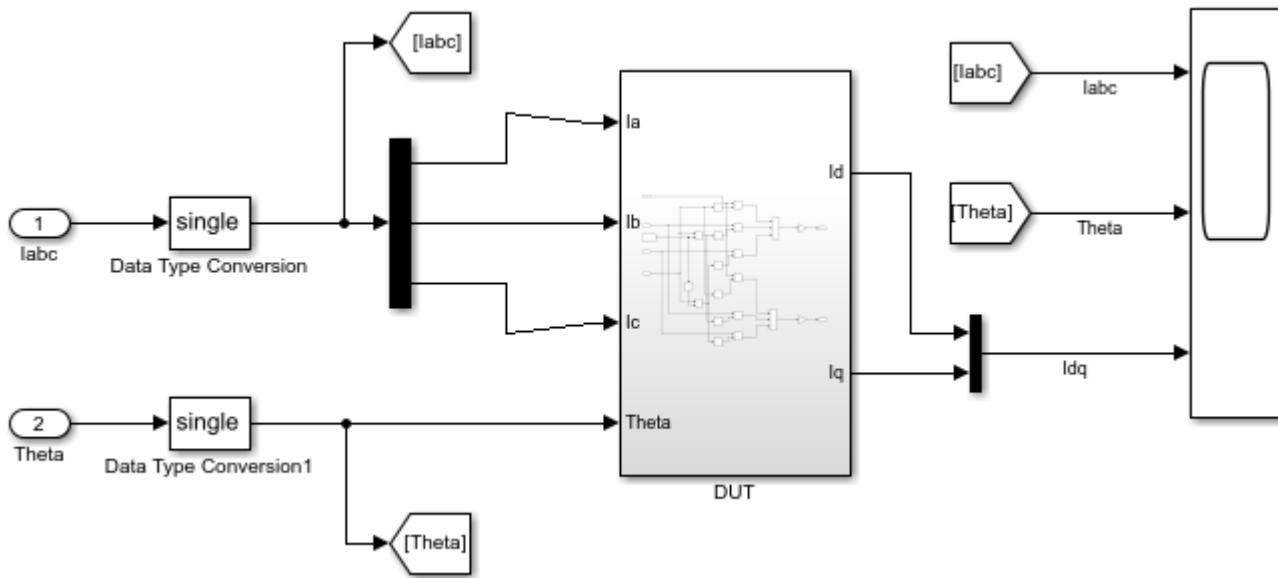
```
design_name = 'hdlcoderParkTransformShare';
design_new_name = 'hdlcoderParkTransformShareCopy';

hdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos');
hdlc_temp_dir = [tempdir 'hdlcoderParkTransformDir'];

% Create a temporary folder and copy the Simulink model.
cd(tempdir);
[~, ~, ~] = rmdir(hdclc_temp_dir, 's');
mkdir(hdclc_temp_dir);
cd(hdclc_temp_dir);

copyfile(fullfile(hdclc_demo_dir, [design_name, '.slx']), fullfile(hdclc_temp_dir, [design_new_name

% Open the model.
open_system(design_new_name);
```



Set a **SharingFactor** of 6 on the subsystem of interest and generate HDL along with the corresponding reports from the model.

```

subsystem = [design_new_name '/DUT'];
hdlset_param(subsystem, 'SharingFactor', 6);
makehdl(subsystem);

### Generating HDL for 'hdlcoderParkTransformShareCopy/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTransformShareCopy')">.
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these additional latencies:
### Output port 0: 39 cycles.
### Output port 1: 39 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderParkTransformShareCopy')">.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.
### MESSAGE: The design requires 6 times faster clock with respect to the base rate = 6.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hdldoc.html')">.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hdldoc.html.
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 1 warnings, and 1 message.
### HDL code generation complete.

```

The High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage

Multipliers	13
Adders/Subtractors	178
Registers	1012
Total 1-Bit Registers	16888
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

You can also run synthesis workflow on the model. The synthesis results for this model are:

Resource	Usage
Slice LUTs	8792
Slice Registers	8419
DSPs	13
Block Ram Tile	0
URAM	0

For more details regarding synthesis workflow, see “HDL Code Generation and FPGA Synthesis from Simulink Model”.

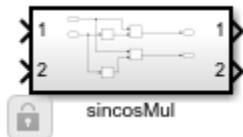
Use Clone Detection to Optimize the Model for Resource Sharing

Before you use the Clone Detector App, create a customer library that contains each repeating pattern in the model as an Atomic Subsystem. For this example, you are given a custom library called hdlcoderParkTransformShareLib.slx.

```
% Copy the library into the temporary directory
lib_name = 'hdlcoderParkTransformShareLib';
lib_new_name = 'hdlcoderParkTransformShareLibCopy';

copyfile(fullfile(hdlc_demo_dir, [lib_name, '.slx']), fullfile(hdlc_temp_dir, [lib_new_name, '.slx'])

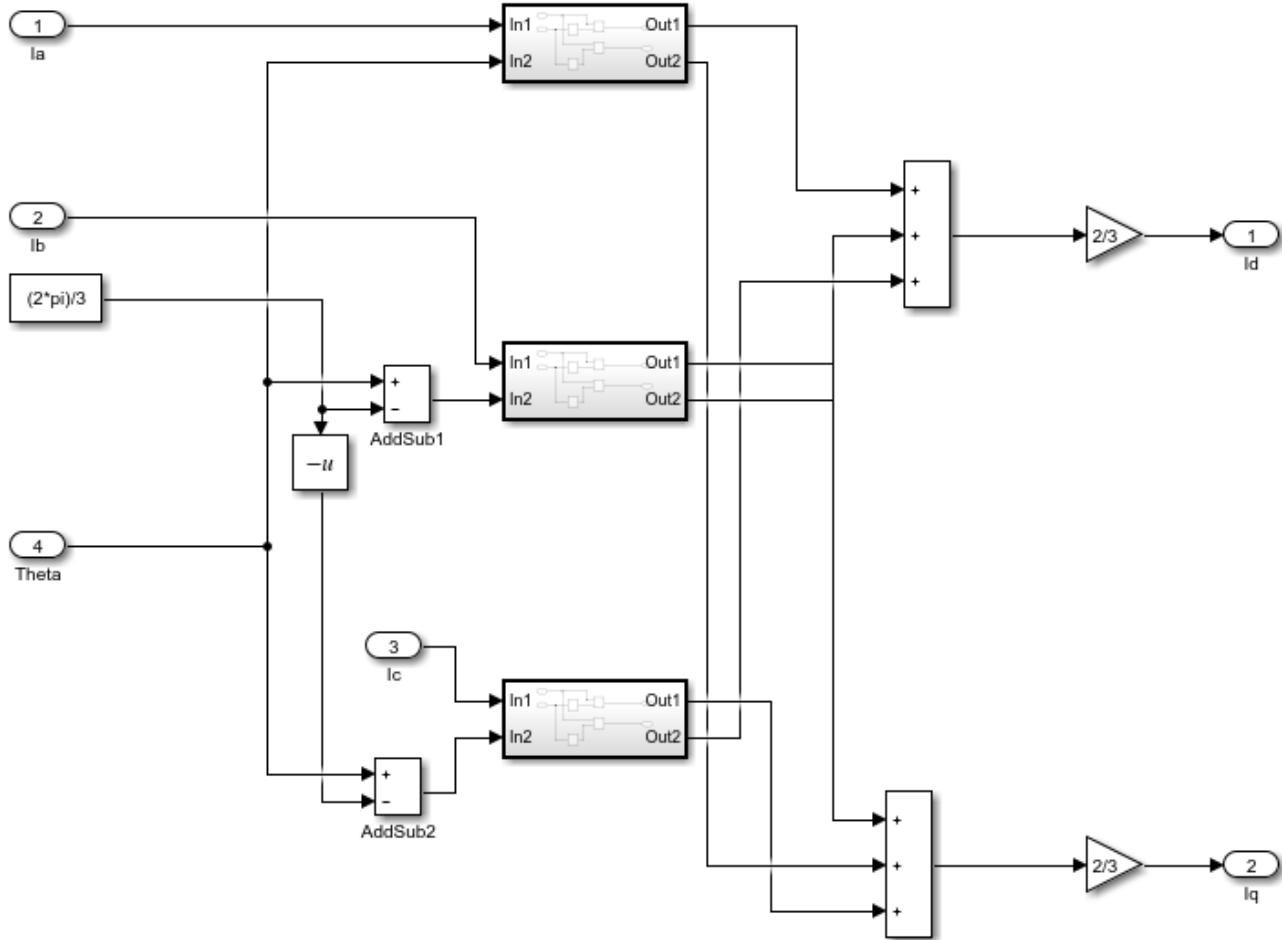
% Open the custom library
open_system('hdlcoderParkTransformShareLibCopy');
```



Replace patterns of the model with the library containing the Atomic Subsystem. To configure the Clone Detector App and replace the clones, refer to “Enable Component Reuse by Using Clone Detection” (Simulink Check). Specifically, follow the instructions in the *Set the Parameters for Clone Detection* section to link the library for clone detection, and set the **Maximum number of different parameters** value to 0.

Then, follow the *Replace Clones* section to apply this library to the model for exact clone replacement.

Notice that all the patterns in the original model has been replaced with atomic subsystems from the library as shown below:



Save the changes made to this system. In the Simulink Editor, on the **Simulation** tab, click **Save**.

Compare Results from Optimized Model

Since this model contains the Sin, Cos, and Product blocks inside atomic subsystems, they can now be shared and results in an improvement in resource utilization.

```

subsystem = [design_new_name '/DUT'];
hdlset_param(subsystem, 'SharingFactor', 3);
% Generate HDL code and the corresponding reports for the optimized model
makehdl(subsystem);

### Generating HDL for 'hdlcoderParkTransformShareCopy/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderParkTran
### Starting HDL check.
### The DUT requires an initial pipeline setup latency. Each output port experiences these addit
### Output port 0: 33 cycles.
### Output port 1: 33 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderParkTransformShareCo
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderParkTransformShareCopy'.

```

```
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 6.
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sincos_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_sub_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_mul_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_add_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on hdlcoderParkTransformShareCopy/DUT/nfp_uminus_single as hdl_prj\hdlsrc\hdlcoderParkT
### Working on DUT_tc as hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_tc.vhd.
### Working on hdlcoderParkTransformShareCopy/DUT as hdl_prj\hdlsrc\hdlcoderParkTransformShareCop
### Generating package file hdl_prj\hdlsrc\hdlcoderParkTransformShareCopy\DUT_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\hd
### HDL check for 'hdlcoderParkTransformShareCopy' complete with 0 errors, 1 warnings, and 1 mess
### HDL code generation complete.
```

After sharing, the High-level Resource Report shows that the resource utilization for the design was estimated to be:

Resource	Usage
Multipliers	13
Adders/Subtractors	177
Registers	1004
Total 1-Bit Registers	16657
RAMs	0
Multiplexers	610
I/O Bits	196
Static Shift operators	8
Dynamic Shift operators	8

Notice that the **Adders/Subtractors**, **Registers**, and **Total 1-Bit Registers** counts have reduced after the atomic subsystems are shared.

You can also run synthesis workflow on the model. The synthesis results for this model are:

Resource	Usage
Slice LUTs	8848
Slice Registers	8308
DSPs	13
Block Ram Tile	0
URAM	0

Notice that the **Slice Registers** count has reduced after sharing.

Delay Balancing

In this section...

- “Why Use Delay Balancing?” on page 24-63
- “Specify Delay Balancing” on page 24-63
- “Delay Balancing Limitations” on page 24-64
- “Delay Balancing Report” on page 24-66

Why Use Delay Balancing?

HDL Coder supports several optimizations, block implementations, and options that introduce discrete delays into the model, with the goal of more efficient hardware usage or achieving higher clock rates. Examples include:

- *Optimizations*: Optimizations such as output pipelining, streaming, or resource sharing can introduce delays.
- *Cascading*: Some blocks support cascade implementations, which introduce a cycle of delay in the generated code.
- *Block implementations*: Some block implementations such as the Newton-Raphson architecture inherently introduce delays in the generated code.

When optimizations or block implementation options introduce delays along the critical path in a model, the numerics of the original model and generated model or HDL code can differ because equivalent delays are not introduced on other, parallel signal paths. Manual insertion of compensating delays along the other paths is possible, but is error prone and does not scale well to large models with many signal paths or multiple sample rates.

To help you solve this problem, HDL Coder supports delay balancing. By default, delay balancing is enabled on the model. The code generator detects introduction of new delays along one path, and then inserts matching delays on the other paths. When delay balancing is enabled, the generated model is functionally equivalent to the original model. It is not recommended that you disable delay balancing on the model. If you disable this setting, HDL Coder generates a warning that numerical differences can occur in the validation model. To fix this warning, enable **Balance delays** on the model or run the model check “Check delay balancing setting” on page 38-11.

Specify Delay Balancing

You can set delay balancing for an entire model. For finer control, you can also set delay balancing for subsystems within the top-level DUT subsystem.

Set Delay Balancing for a Model

Use the following `makehdl` properties to set delay balancing for a model:

- **BalanceDelays**: By default, model-level delay balancing is enabled, and subsystems within the model inherit the model-level setting. To learn how to set delay balancing for a model, see “Balance delays” on page 15-3.
- **GenerateValidationModel**: By default, validation model generation is disabled. When you enable delay balancing, generate a validation model to view delays and other differences between

your original model and the generated model. To learn how to enable validation model generation, see the **Generate validation model** section in “Model Generation Parameters for HDL Code” on page 17-82.

For example, the following commands generate HDL code with delay balancing and generate a validation model.

```
dut = 'ex_rsqrt_delaybalancing/Subsystem';
makehdl(dut,'BalanceDelays','on','GenerateValidationModel','on');
```

Disable Delay Balancing for a Subsystem

You can disable delay balancing for an entire model or disable a subsystem within the top-level DUT subsystem. For example, if you do not want to balance delays for a control path, you can put the control path in a subsystem, and disable delay balancing for that subsystem.

To disable delay balancing for a subsystem within the top-level DUT subsystem, disable delay balancing at the model level. When you disable delay balancing for the model, the validation model does not compensate for latency inserted in the generated model due to optimizations or block implementations. The validation model can therefore show mismatches between the original model and generated model.

To disable delay balancing for a subsystem within the top-level DUT subsystem:

- 1 Disable delay balancing for the model.
- 2 Enable delay balancing for the top-level DUT subsystem.
- 3 Disable delay balancing for a subsystem within the DUT subsystem.

When delay balancing is enabled on the model, the delay balancing setting on individual subsystems is ignored. To learn how to set delay balancing for a subsystem, see “Set Delay Balancing For a Subsystem” on page 22-5.

Delay Balancing Limitations

If delay balancing is unsuccessful, `hdlcoder.optimizeDesign` cannot optimize the generated HDL code.

Unsupported Blocks

The following blocks and subsystems do not support delay balancing:

- Triggered Subsystem
- Atomic Subsystem
- HDLCosimulation
- Data Type Duplicate
- Decrement To Zero
- Frame Conversion
- Ground
- FFT HDL Optimized
- LMS Filter
- Model Reference

- To VCD File
- Magnitude-Angle to Complex

Block Mode Restrictions

The following block implementations do not support delay balancing:

- `hdldefaults.ConstantSpecialHDL Emission`
- `hdldefaults.NoHDL`

Subsystem-Level Restrictions

HDL Coder does not support delay balancing, if:

- There are multiple instances of an Atomic Subsystem in different conditional subsystems.
In the Block Parameters dialog box of the Atomic Subsystem, you can set **Function packaging** to **Nonreusable function**.
- The “BalanceDelays” on page 22-5 block property for all instances of an Atomic Subsystem or Model Reference resolves to a different value.

To fix this error, disable BalanceDelays for all instances of the Atomic Subsystem or Model Reference.

- The block is inside a conditional subsystem and has pipeline delays.
- A subsystem with **BlackBox Architecture** has the **ImplementationLatency** block property set to a negative value.

To fix this error, for **ImplementationLatency**, enter a nonnegative integer.

Other Restrictions

HDL Coder does not support delay balancing, if:

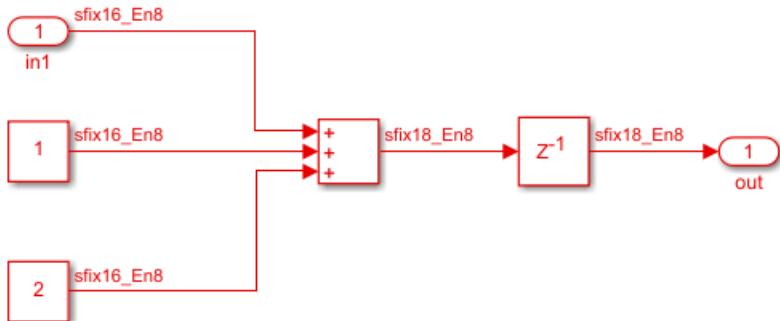
- Delays are introduced in a feedback loop and HDL Coder cannot balance the path delays. For example, if you apply clock-rate pipelining inside a feedback loop, HDL Coder introduces a delay at the clock-rate, and can cause delay balancing to fail.

To reduce the number of clock-rate delays, increase the “Oversampling factor” on page 17-15.

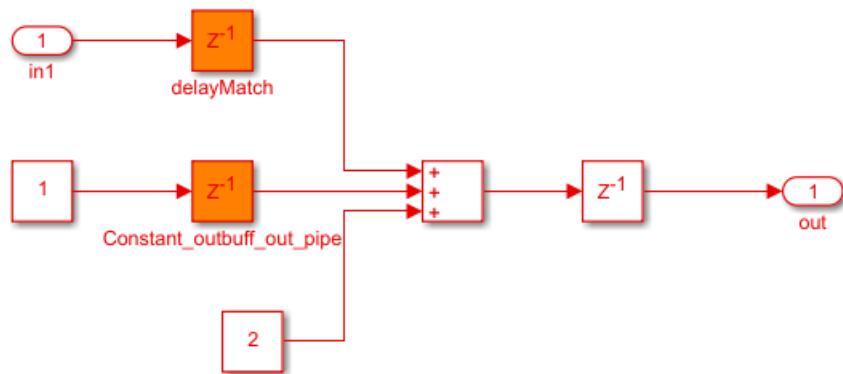
- The sample time is not discrete or the ratio of sample times of the fastest to the slowest rate is too large.

Delay Balancing with Constants

When you have Constant blocks as inputs inside the DUT Subsystem for which delay balancing is enabled, you see an initial simulation mismatch in the validation model. Consider this model inside a DUT Subsystem. The Constant block that outputs a value of 1 has the HDL block property **OutputPipeline** set to 1 .



This figure displays the generated validation model. You see that delay balancing added a matching delay to the input port to balance the pipeline register inserted for the Delay block. The code generator does not insert a matching delay on the parallel path containing the Constant block with the value 2 because the output value of the block is a constant. This delay not inserted results in an initial simulation mismatch



To resolve the simulation mismatch, in the validation model, manually add a matching delay at the output of the Constant block with the value 2.

Delay Balancing Report

To see the delay balancing information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate code for each subsystem, model reference, or MATLAB Function block, HDL Coder produces the optimization report. In the report, select the **Delay Balancing** section of the report.

The Delay Balancing Report shows latency changes, pipeline delay and phase delay at the output ports, and the number of pipelines added at the output ports to match the delays. If delay balancing fails, the report mentions the criteria that was violated and displays the link to any block or subsystem that caused delay balancing to fail.

See Also

Related Examples

- “Delay Balancing and Validation Model Workflow In HDL Coder™” on page 24-68

More About

- “Resolve Numerical Mismatch with Delay Balancing” on page 24-24
- “Create and Use Code Generation Reports” on page 25-2
- “Validation Model” on page 24-11

Delay Balancing and Validation Model Workflow In HDL Coder™

This example shows how HDL Coder can automatically balance delays within a model. HDL Coder may introduce additional delays in the HDL implementation for a given model. These delays may be introduced by either certain block implementations or by optimizations for the purpose of improving the efficiency of the hardware implementations. However, introducing delays on only certain paths can result functional behavior that is different from the original intent of the user model, thereby violating functional equivalence between the original user model and the HDL implementation.

Delay Balancing is a feature supported by HDL Coder for automatically balancing such newly introduced delays across all cut-sets, ensuring that functional integrity is preserved with reference to the original model. This equivalence relationship can be confirmed by invoking the validation model workflow that enables the user to visualize the HDL Code-generation model, the delays introduced by implementations and those introduced by delay balancing and verify the equivalence relationship with the original model.

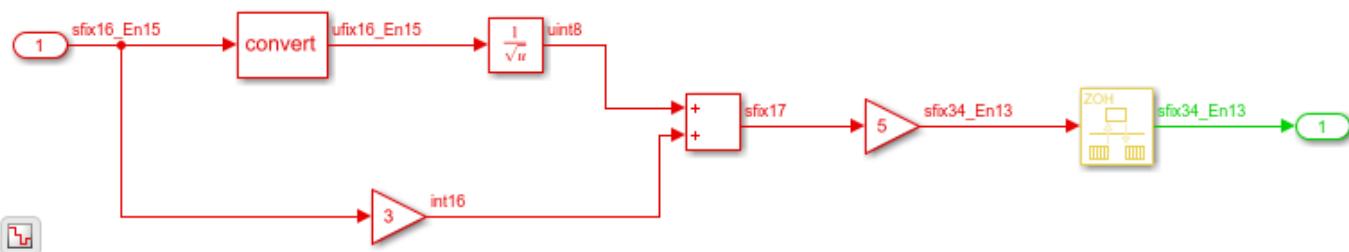
Implementations and Optimizations introduce Latency

Some of the arithmetic blocks in Simulink require complex hardware algorithms. Consider, for example, the reciprocal square root block. This block computes its answer in a single time step in Simulink. If the corresponding hardware implementation should stay cycle-accurate with Simulink, the hardware algorithm for this block must compute in a single clock cycle. However, this results in a long critical path that degrades the clock frequency and efficiency of hardware. Thus, HDL Coder implements this block with a 5-cycle latency, which means that every path containing this block will introduce a 5-cycle delay.

Certain optimizations supported by HDL Coder may also introduce additional delays. For example, specifying 'InputPipeline' or 'OutputPipeline' as an implementation parameter on a block introduces additional pipeline delays in the generated HDL. This is again unmatched across cut-sets and will result in functional differences with the original model.

Consider an example model that contains a square root block implementing the 'rSqrt' function and the 'Gain3' block along the parallel path has the 'OutputPipeline' implementation parameter set to 2.

```
bdclose all;
load_system('hdl_delaybalancing');
open_system('hdl_delaybalancing/Subsystem');
set_param('hdl_delaybalancing', 'SimulationCommand', 'update');
```



Validation Model Generation

Due to changes in latency, the HDL Coder always generates a Code Generation model that captures the added delays during implementation. The RTL verification and automatic co-simulation model generation features validate that the RTL simulation of the generated HDL code is bit-accurate and

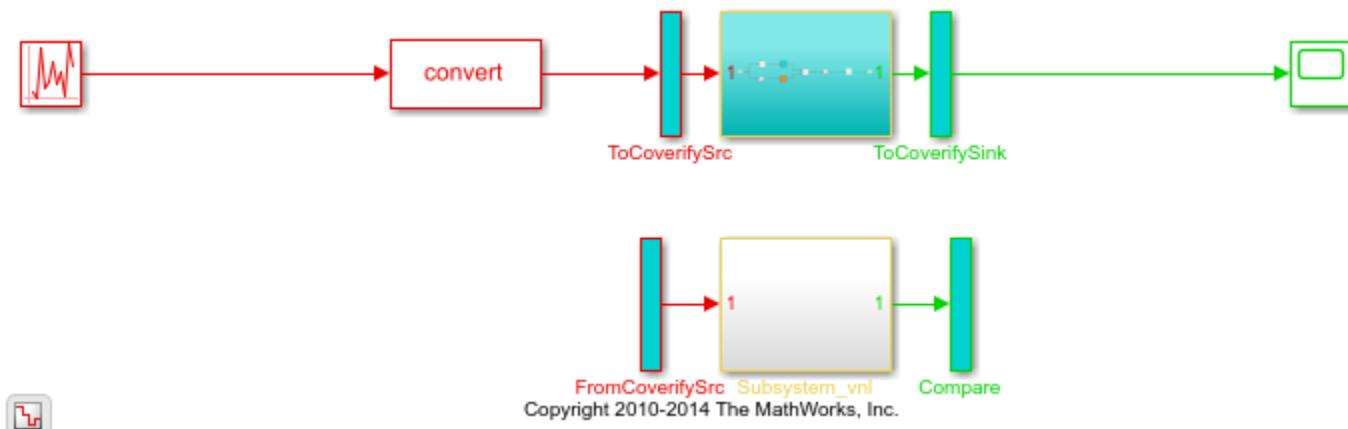
cycle-accurate with the Simulink simulation of the Code-generation model. However, this does not say anything about the functional relationship with the original, user model.

The Validation model enables the user to verify that the functional equivalence of the original, user model with the Code-generation model. This feature is turned on by the model-level parameter, 'GenerateValidationModel'. This parameter can be set by either the `hdlset_param` command or can be supplied as a `makehdl` argument. Then, during code generation, notice a message that says that validation model has been generated.

The Validation Model consists of two parts: the DUT from the original model (called '`gm_hdl_delaybalancing_vnl/Subsystem_vnl`') and the DUT from the Code-generation model ('`gm_hdl_delaybalancing_vnl/Subsystem`').

```
hdlset_param('hdl_delaybalancing', 'GenerateValidationModel', 'on');
hdlset_param('hdl_delaybalancing', 'BalanceDelays', 'off');
makehdl('hdl_delaybalancing/Subsystem');
open_system('gm_hdl_delaybalancing_vnl');
set_param('gm_hdl_delaybalancing_vnl', 'SimulationCommand', 'update');

### Generating HDL for 'hdl_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_delaybalanc...
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdl_delaybalancing_vnl')">gr...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_iv as hdlsrc\hdl_delaybalancing\Sqrt_iv.vhd
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_core as hdlsrc\hdl_delaybalancing\Sqrt_core.vhd
### Working on hdl_delaybalancing/Subsystem/Sqrt as hdlsrc\hdl_delaybalancing\Sqrt.vhd.
### Working on Subsystem_tc as hdlsrc\hdl_delaybalancing\Subsystem_tc.vhd.
### Working on hdl_delaybalancing/Subsystem as hdlsrc\hdl_delaybalancing\Subsystem.vhd.
### Generating package file hdlsrc\hdl_delaybalancing\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpe...
### HDL check for 'hdl_delaybalancing' complete with 0 errors, 2 warnings, and 4 messages.
### HDL code generation complete.
```

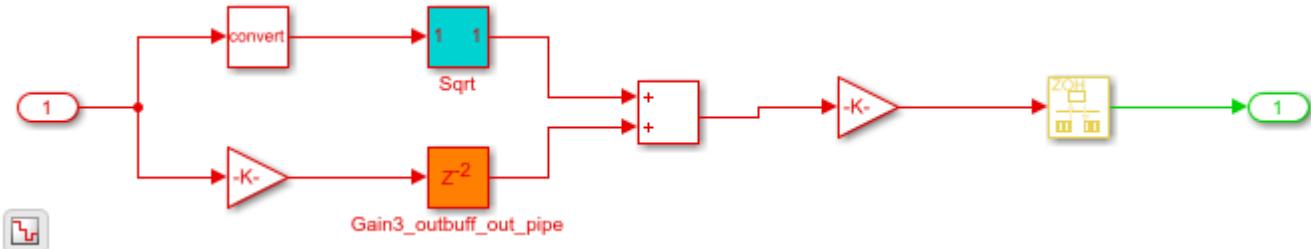


Code Generation DUT: Output Pipeline Insertion

The top subsystem ('`gm_hdl_delaybalancing_vnl/Subsystem`') in the Validation model is the DUT as implemented for HDL code generation and this is reference DUT when performing RTL testbench

verification and Cosimulation block based verification. Notice that 'OutputPipeline' parameter on the 'Gain3' block is implemented by an integer delay of length 2.

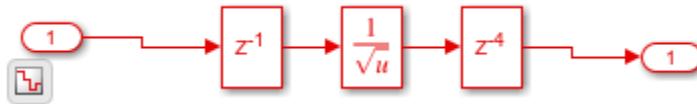
```
open_system('gm_hdl_delaybalancing_vnl/Subsystem');
```



Code Generation DUT: Sqrt Block Implementation

Implementing the square root function in one clock cycle is not efficient for hardware. The coder implements a pipelined architecture and this is reflected in the Code-Generation DUT model (under the square root subsystem) by the 5 additional delays.

```
open_system('gm_hdl_delaybalancing_vnl/Subsystem/Sqrt');
```



Equivalence Checking with the Validation Model

Validation model performs equivalence checking by routing the same inputs to both (the original and code-generation) DUTs using 'From' and 'Goto' blocks. This is encapsulated in the 'ToCoverifySrc' and 'FromCoverifySrc' subsystems. Both DUTs now respond to the same stimuli in each time step. The outputs from both DUTs are then sampled in each time step and their equivalence is checked. This is done by comparing the outputs from each output port, computing their difference, which should always be zero for functional equivalence.

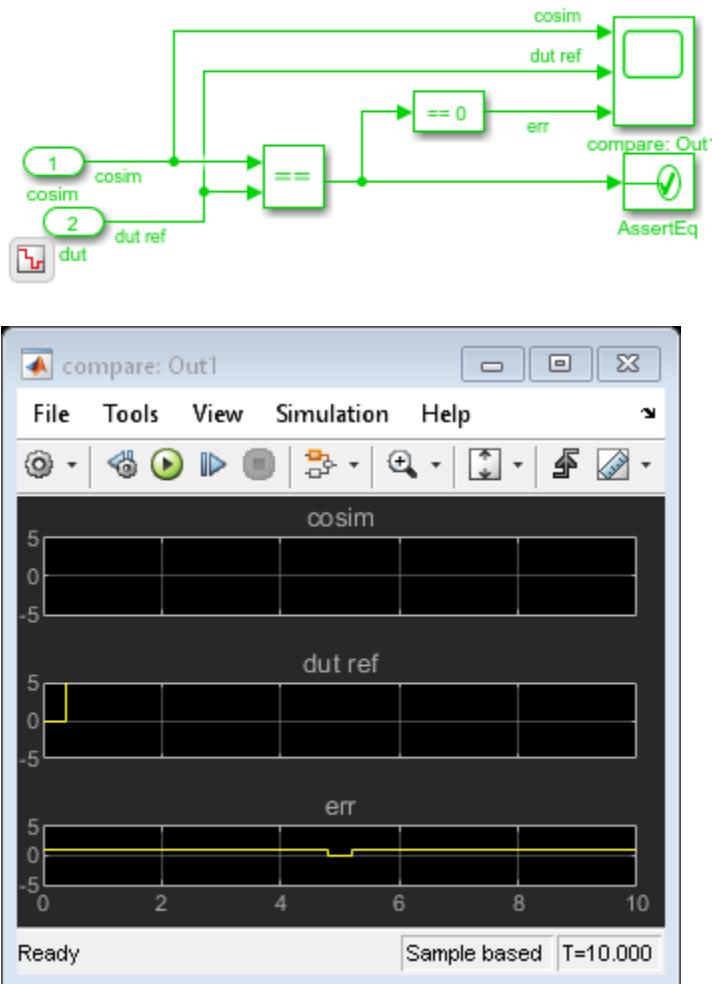
In the current example, however, notice that functional equivalence is violated. The difference between the two outputs is non-zero in several time steps. This results in mismatch assertions and is also reflected in the last panel of the comparison scope.

```
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1');
sim('gm_hdl_delaybalancing_vnl');
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/compare: Out1')

Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0
Warning: Division by zero occurred. Quotient was saturated. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
• - Suppress

Warning: Saturate on overflow detected. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
• - Suppress
```

```
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 0.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 1.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 1.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 2.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 3.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 3.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 4.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 5.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 5.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 6.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 7.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 7.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8.4
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 8.8
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 9.2
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 9.6
Warning: Assertion detected in
'gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/AssertEq' at time 10
```



Automatic Delay Balancing

To solve the functional equivalence problem, the user can turn on the delay balancing feature by setting the model-level 'BalanceDelays' option to 'on'. This can be done through either the `hdlset_param` command or as a `makehdl` argument.

With this option turned on, HDL Coder will automatically identify the locations where matching delays need to be added to guarantee functional equivalence. This will cover regular cut-sets, multi-rate boundaries and subsystem boundaries, after taking in to account, all the implementation- and optimization-induced delays.

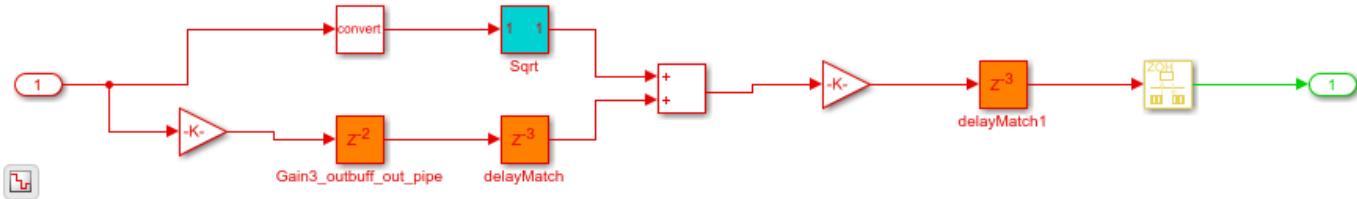
Now when we observe the Code-generation DUT from the validation model notice that several additional delays have been added for matching delays introduced by the `Sqrt` block and `OutputPipeline` option. The names of these delays are typically prefixed with 'delayMatch'. Notice that the coder also automatically computes the appropriate delays needed when crossing rate boundaries.

```
hdlset_param('hdl_delaybalancing', 'BalanceDelays', 'on');
makehdl('hdl_delaybalancing/Subsystem');
open_system('gm_hdl_delaybalancing_vnl/Subsystem')
set_param('gm_hdl_delaybalancing_vnl', 'SimulationCommand', 'update');
```

```

### Generating HDL for 'hdl_delaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_delaybalanc...
### Starting HDL check.
### 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 0: 2 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdl_delaybalancing_vnl')">gr...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdl_delaybalancing'.
### MESSAGE: The design requires 3 times faster clock with respect to the base rate = 0.1.
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_iv as hdlsrc\hdl_delaybalancing\Sqrt_iv.vhd
### Working on hdl_delaybalancing/Subsystem/Sqrt/Sqrt_core as hdlsrc\hdl_delaybalancing\Sqrt_core...
### Working on hdl_delaybalancing/Subsystem/Sqrt as hdlsrc\hdl_delaybalancing\Sqrt.vhd.
### Working on Subsystem_tc as hdlsrc\hdl_delaybalancing\Subsystem_tc.vhd.
### Working on hdl_delaybalancing/Subsystem as hdlsrc\hdl_delaybalancing\Subsystem.vhd.
### Generating package file hdlsrc\hdl_delaybalancing\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdl_delaybalancing' complete with 0 errors, 0 warnings, and 4 messages.
### HDL code generation complete.

```



Initial Latency and Functional Validation

The delays introduced by implementations essentially construct a pipelined hardware architecture to improve clock frequency and hardware efficiency. The pipeline however introduces an initial latency and the first output sample is generated after this initial latency. While these pipeline delays are automatically balanced inside the DUT, it is the user's responsibility to balance delays outside the DUT in the rest of the model. The amount of delay (or initial latency) is communicated to the user during code generation as follows:

```

### Some latency changes occurred in the DuT. Each output port experiences these additional de...
### Output port 0: 2 cycles

```

The equivalence checking in the Validation model uses this initial latency information for delaying the output from the original DUT. This is an example of balancing the delay outside DUT, since the balancing occurs at the inputs of the equivalence checking subsystem. Now, when we simulate the Validation model, note that there are no assertions and thus functional equivalence is preserved. While the pipeline delays are automatically balanced inside the DUT, it is the user's responsibility to balance delays outside the DUT in the rest of the model.

```

close_system('gm_hdl_delaybalancing_vnl/Subsystem')
sim('gm_hdl_delaybalancing_vnl');
open_system('gm_hdl_delaybalancing_vnl/Compare/Assert_Out1/compare: Out1')

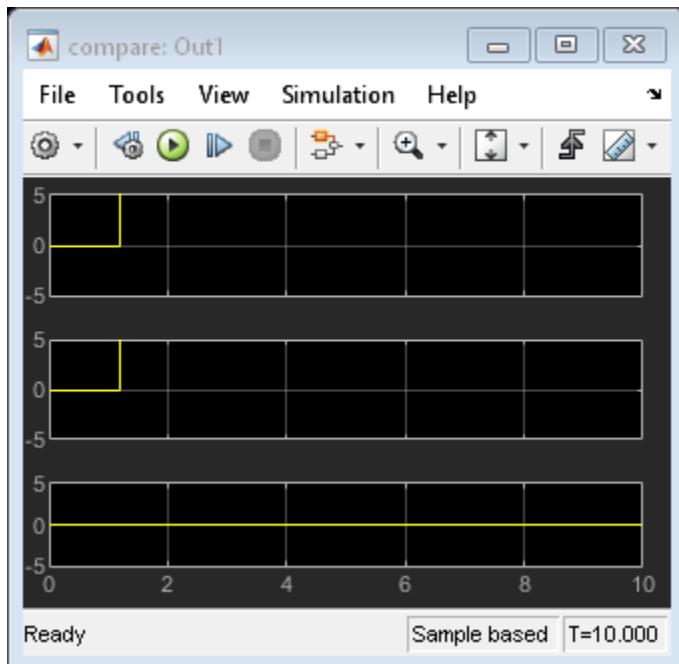
```

Warning: Division by zero occurred. Quotient was saturated. This originated from
 'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'

Suggested Actions:

- - Suppress

Warning: Saturate on overflow detected. This originated from
'gm_hdl_delaybalancing_vnl/Subsystem/Sqrt/Sqrt'
Suggested Actions:
• - Suppress



Control the scope of delay balancing

The examples above describe the delay balancing feature as applied to the whole DuT. Sometimes, the design may explicitly model control and data paths, and you may not want to insert matching delays on the control path during delay balancing. The examples in “Control the Scope of Delay Balancing” on page 24-75 show how this option can be applied locally to individual subsystems instead of the entire DuT.

Control the Scope of Delay Balancing

This example shows how to balance delays in specific parts of a design, without balancing delays on the entire design.

Introduction

You can use the 'BalanceDelays' option to balance the additional delays introduced by HDL Coder™ for certain block implementations and optimizations. This model-level option controls delay balancing for the entire model. However, for certain designs, you may want to balance delays in only some parts of the design. For example, in a design containing a data path and a control path, delay balancing should be applied only on the data path of the design, i.e., the paths requiring data synchronization. This example shows how to use a subsystem-level 'BalanceDelays' option provides fine-grained control on how HDL Coder balances delays in individual subsystems.

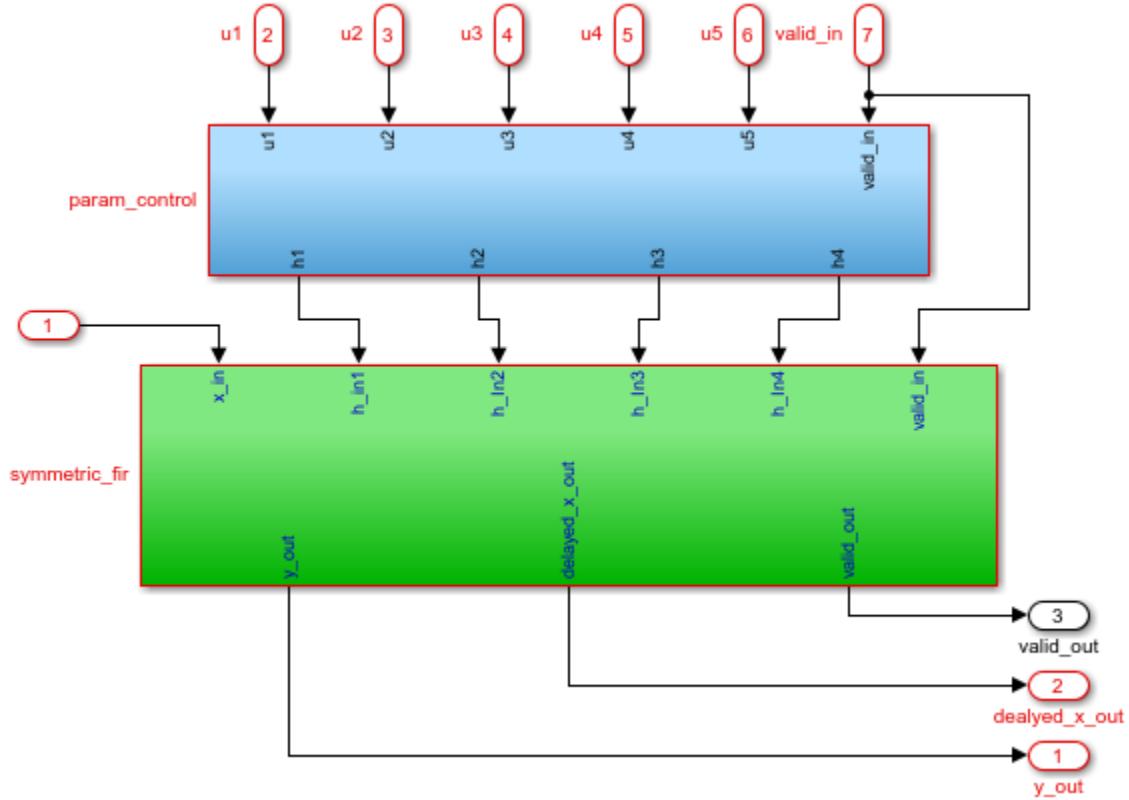
We use two examples to demonstrate the use of this subsystem-level feature:

- 1 hdlcoder_localdelaybalancing.slx shows how to disable delay balancing on user-defined control paths.
- 2 hdlcoder_localdelaybalancing_sharing.slx shows how the user can apply HDL optimizations like resource sharing in the presence of complicated control paths that require carefully constrained delay balancing.

Example 1: Constraining Delay Balancing to the data path

The example model, "hdlcoder_localdelaybalancing.slx", has two subsystems under hdlcoder_localdelaybalancing/Subsystem: param_control and symmetric_fir, containing the control logic and the data path, respectively.

```
bdclose('all');
open_system('hdlcoder_localdelaybalancing');
open_system('hdlcoder_localdelaybalancing/Subsystem');
```



Each subsystem has one block that has one output pipeline register to achieve good timing results.

```
hdldispblkparams('hdlcoder_localdelaybalancing/Subsystem/param_control/And');
hdldispblkparams('hdlcoder_localdelaybalancing/Subsystem/symmetric_fir/Add');
```

```
%%%%%%%%%%%%%
HDL Block Parameters ('hdlcoder_localdelaybalancing/Subsystem/param_control/And')
%%%%%%%%%%%%%
```

Implementation

Architecture : default

Implementation Parameters

OutputPipeline : 1

```
%%%%%%%%%%%%%
HDL Block Parameters ('hdlcoder_localdelaybalancing/Subsystem/symmetric_fir/Add')
%%%%%%%%%%%%%
```

Implementation

Architecture : Linear

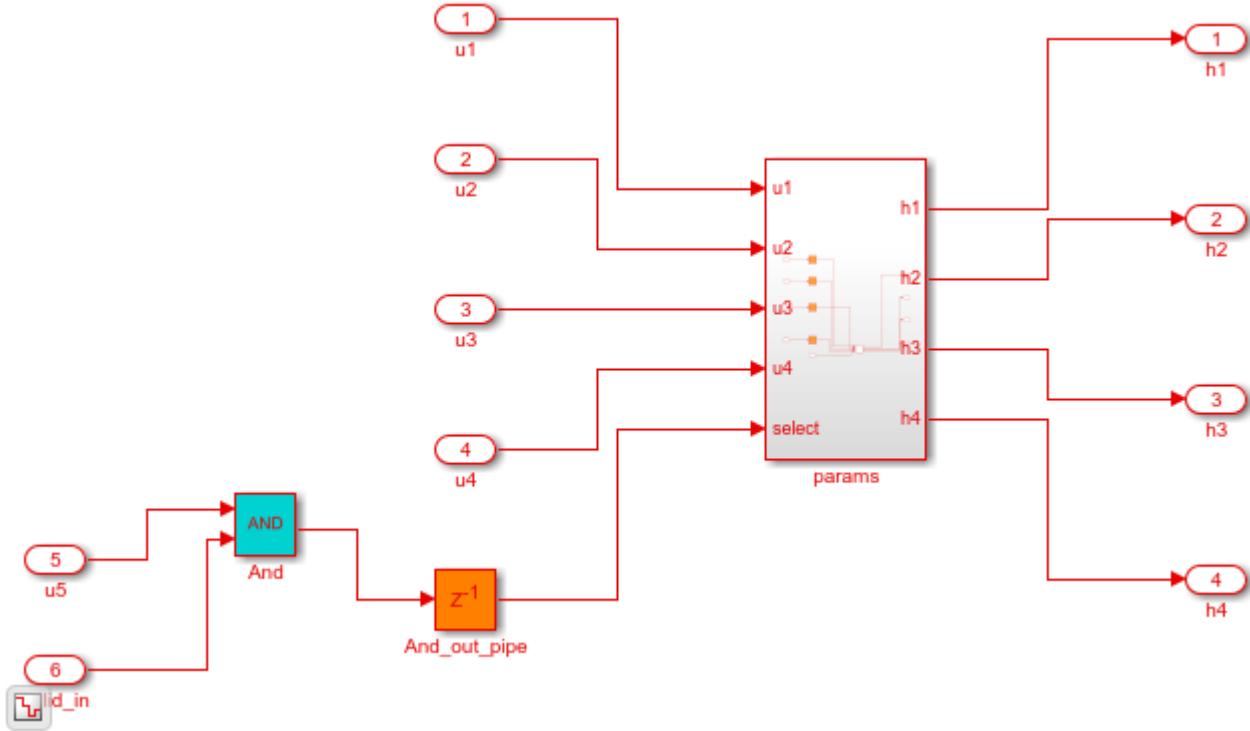
Implementation Parameters

```
OutputPipeline : 1
```

When the global, model-level 'BalanceDelays' option is set to 'on', then delay balancing inserts matching delays on both the control path as well as the data path, as shown in the validation model.

```
hdlset_param('hdlcoder_localdelaybalancing', 'BalanceDelays', 'on');
hdlset_param('hdlcoder_localdelaybalancing', 'GenerateValidationModel', 'on');
makehdl('hdlcoder_localdelaybalancing/Subsystem');
load_system('gm_hdlcoder_localdelaybalancing_vnl');
set_param('gm_hdlcoder_localdelaybalancing_vnl', 'SimulationCommand', 'update');
open_system('gm_hdlcoder_localdelaybalancing_vnl/Subsystem/param_control');

### Generating HDL for 'hdlcoder_localdelaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_localde...
### Starting HDL check.
### 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 addit...
### Output port 0: 1 cycles.
### Output port 1: 1 cycles.
### Output port 2: 1 cycles.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_localdelaybalancing...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_localdelaybalancing'.
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control/params as hdlsrc\hdlcoder_lo...
### Working on hdlcoder_localdelaybalancing/Subsystem/param_control as hdlsrc\hdlcoder_localdelay...
### Working on hdlcoder_localdelaybalancing/Subsystem/symmetric_fir as hdlsrc\hdlcoder_localdelay...
### Working on hdlcoder_localdelaybalancing/Subsystem as hdlsrc\hdlcoder_localdelaybalancing\Sub...
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_localdelaybalancing' complete with 0 errors, 0 warnings, and 0 messag...
### HDL code generation complete.
```



In this example design, only the data path, `symmetric_fir`, requires data synchronization. The outputs from `param_control` are coefficients to the FIR filter and do not have to synchronize with each other or with the processed data. Turning off delay balancing on the control logic therefore saves resources. In order to achieve this, the model-level '`BalanceDelays`' option must be '`off`', and the subsystem-level '`BalanceDelays`' options must be set appropriately on the data path and control path.

```
hdlset_param('hdlcoder_localdelaybalancing', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/param_control', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing/Subsystem/symmetric_fir', 'BalanceDelays', 'on');
```

```

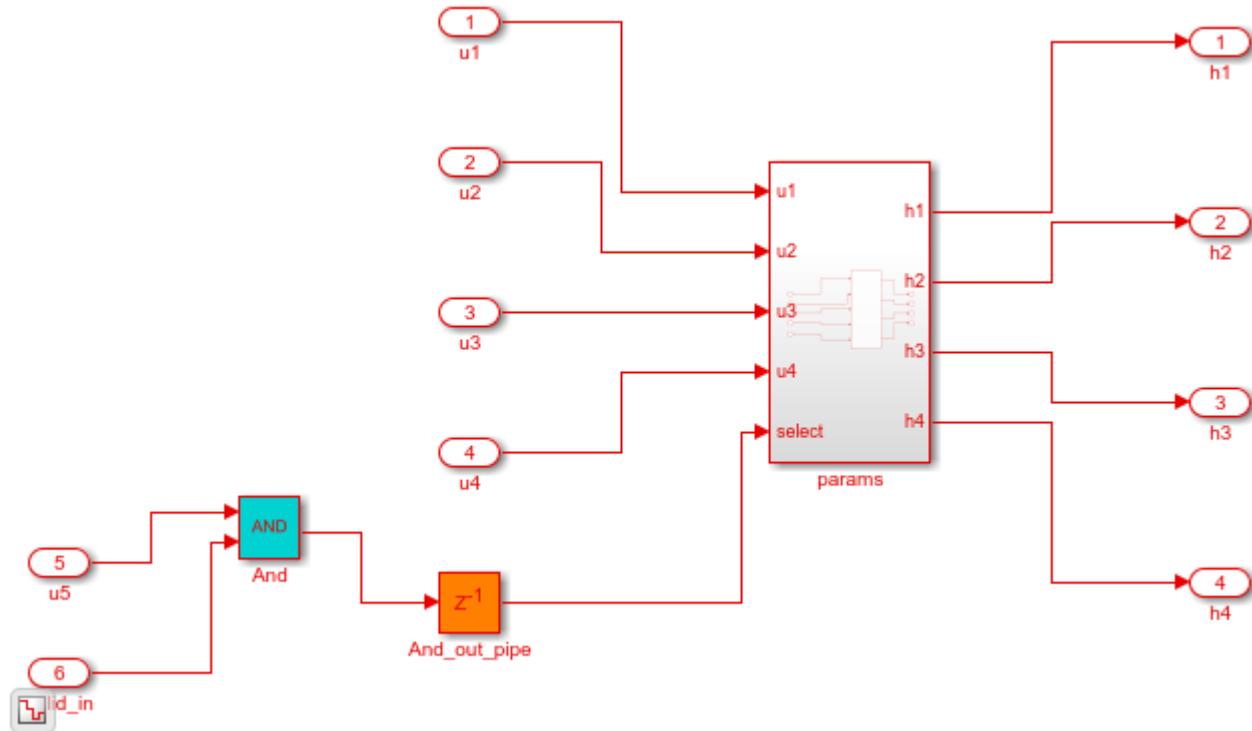
Now when HDL code is generated, delay balancing is only active in the data path subsystem and does
not insert any delays in the control path subsystem.

bdclose('gm_hdlcoder_localedelaybalancing_vnl');
makehdl('hdlcoder_localedelaybalancing/Subsystem');
load_system('gm_hdlcoder_localedelaybalancing_vnl');
set_param('gm_hdlcoder_localedelaybalancing_vnl', 'SimulationCommand', 'update');
open_system('gm_hdlcoder_localedelaybalancing_vnl/Subsystem/param_control');

### Generating HDL for 'hdlcoder_localedelaybalancing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_localedelaybalancing_vnl')">hdlcoder_localedelaybalancing_vnl</a>.
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_localedelaybalancing_vnl/Subsystem/param_control')">gm_hdlcoder_localedelaybalancing_vnl/Subsystem/param_control</a>.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_localedelaybalancing'.
### Working on hdlcoder_localedelaybalancing/Subsystem/param_control/params as hdlsrc\hdlcoder_localedelaybalancing\Subsystem\param_control\params
### Working on hdlcoder_localedelaybalancing/Subsystem/param_control as hdlsrc\hdlcoder_localedelaybalancing\Subsystem\param_control
### Working on hdlcoder_localedelaybalancing/Subsystem/symmetric_fir as hdlsrc\hdlcoder_localedelaybalancing\Subsystem\symmetric_fir
### Working on hdlcoder_localedelaybalancing/Subsystem as hdlsrc\hdlcoder_localedelaybalancing\Subsystem
### Creating HDL Code Generation Check Report file:///C:/TEMP/Bdoc20b_1527579_10488\ib61345F\0\tpl

```

```
### HDL check for 'hdlcoder_localedelaybalancing' complete with 0 errors, 2 warnings, and 0 messages
### HDL code generation complete.
```

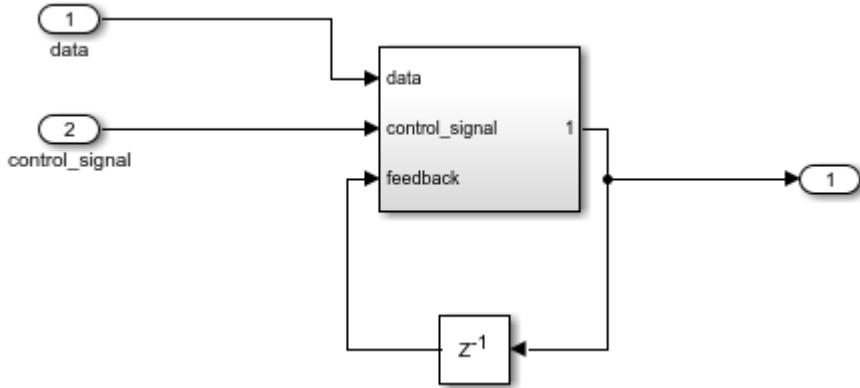


Notice that simulating the validation model now shows mismatches, because the validation model does not compensate for latency inserted by optimizations or block implementations.

Example 2: Localized Delay Balancing and Resource Sharing

The resource sharing optimization saves area usage in the final HDL implementation, at the cost of introducing a cycle of latency for each sharing group. This additional latency is usually balanced during delay balancing so that the numerics and functionality of the algorithm are preserved. One of the restrictions of resource sharing is that it cannot be applied on a subsystem within a feedback loop. Thus, if resource sharing is specified for a subsystem within a loop, then the optimization will fail. You can observe this in `hdlcoder_localedelaybalancing_sharing.slx`, where `hdlcoder_localedelaybalancing_sharing/Subsystem/Subsystem` is within a feedback loop.

```
bdclose('all');
load_system('hdlcoder_localedelaybalancing_sharing');
open_system('hdlcoder_localedelaybalancing_sharing/Subsystem');
```



However, in this design, you may know that the feedback loop is rarely used since the control signal causes the switch block, 'hdlcoder_localdelaybalancing_sharing/Subsystem/Subsystem/Switch', to choose the top input, the feed-forward path, most of the time. This user insight implies that it is fine to go ahead with resource sharing in this subsystem and disregard the feedback loop in the parent subsystem. In such cases, if you wish to ignore feedback loops during delay balancing, you must turn off delay balancing in the subsystem containing the feedback loop. This enables HDL Coder (TM) to ignore the feedback loop and proceed with resource sharing.

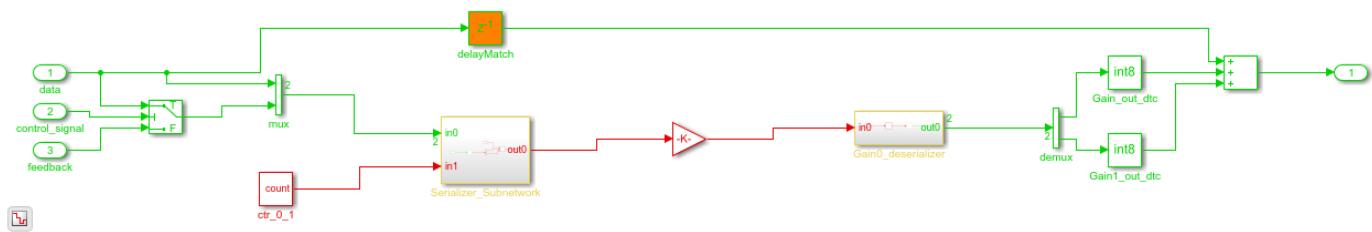
```

hdlset_param('hdlcoder_localdelaybalancing_sharing', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing_sharing/Subsystem', 'BalanceDelays', 'off');
hdlset_param('hdlcoder_localdelaybalancing_sharing/Subsystem/Subsystem', 'BalanceDelays', 'on');
makehdl('hdlcoder_localdelaybalancing_sharing/Subsystem');
load_system('gm_hdlcoder_localdelaybalancing_sharing');
set_param('gm_hdlcoder_localdelaybalancing_vnl', 'SimulationCommand', 'update');

### Generating HDL for 'hdlcoder_localdelaybalancing_sharing/Subsystem'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_localde...
### Starting HDL check.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoder_localdelaybalancing...
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_localdelaybalancing_sharing'.
### MESSAGE: The design requires 2 times faster clock with respect to the base rate = 0.1.
### Working on hdlcoder_localdelaybalancing_sharing/Subsystem/Subsystem as hdsrc\hdlcoder_local...
### Working on Subsystem_tc as hdsrc\hdlcoder_localdelaybalancing_sharing\Subsystem_tc.vhd.
### Working on hdlcoder_localdelaybalancing_sharing/Subsystem as hdsrc\hdlcoder_localdelaybalanc...
### Generating package file hdsrc\hdlcoder_localdelaybalancing_sharing\Subsystem_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\t...
### HDL check for 'hdlcoder_localdelaybalancing_sharing' complete with 0 errors, 2 warnings, and ...
### HDL code generation complete.
  
```

Notice that not only does sharing succeed in the inner subsystem, but local delay balancing also succeeds within this subsystem by inserting matching delays on the inputs to the adder.

```
open_system('gm_hdlcoder_localdelaybalancing_vnl/Subsystem/Subsystem');
```



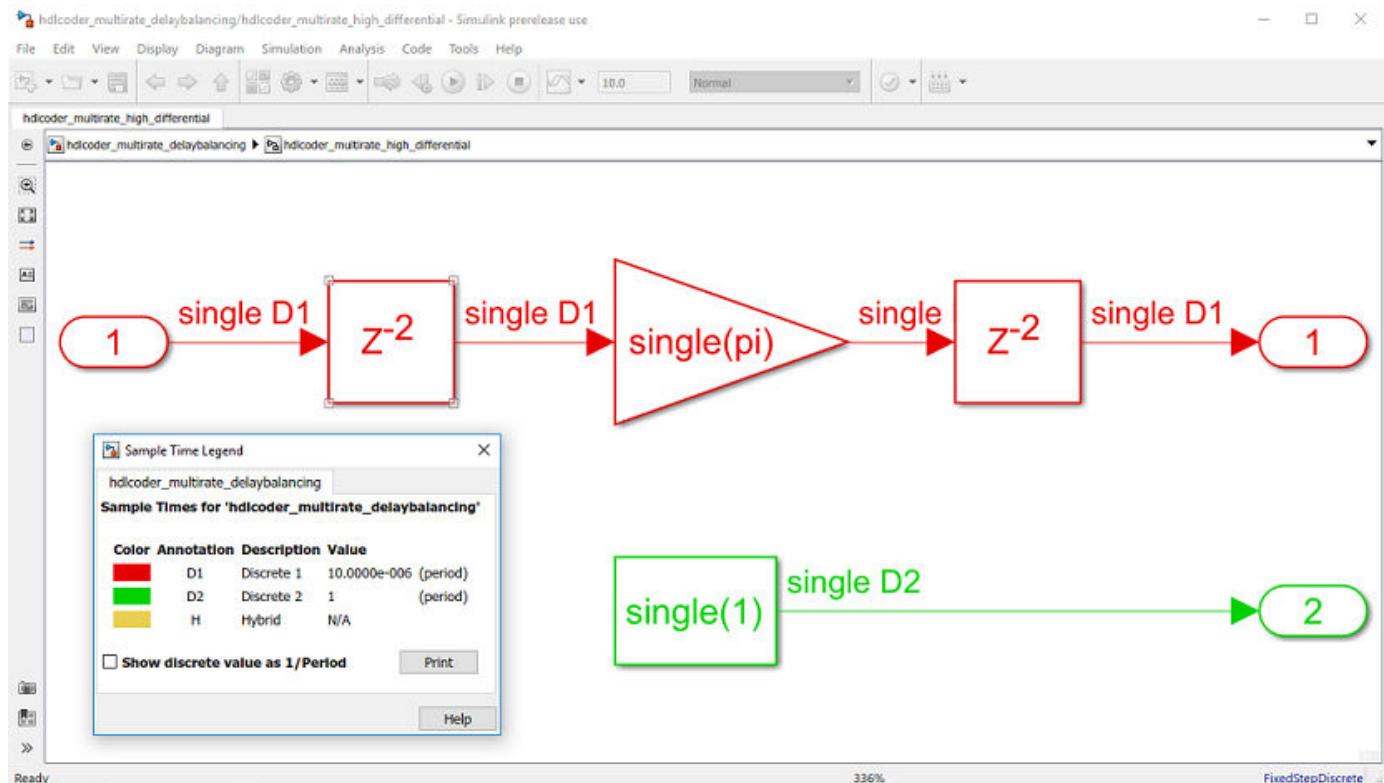
Delay Balancing on multi-rate designs

This example shows how an indiscrete usage of Simulate rates on a multi-rate design can generate an undesirable HDL code, and provides few recommendations for optimal code generation.

Introduction

This example model contains 3 subsystems, the first one demonstrates the issue and the others provide practical ways of resolving the issue.

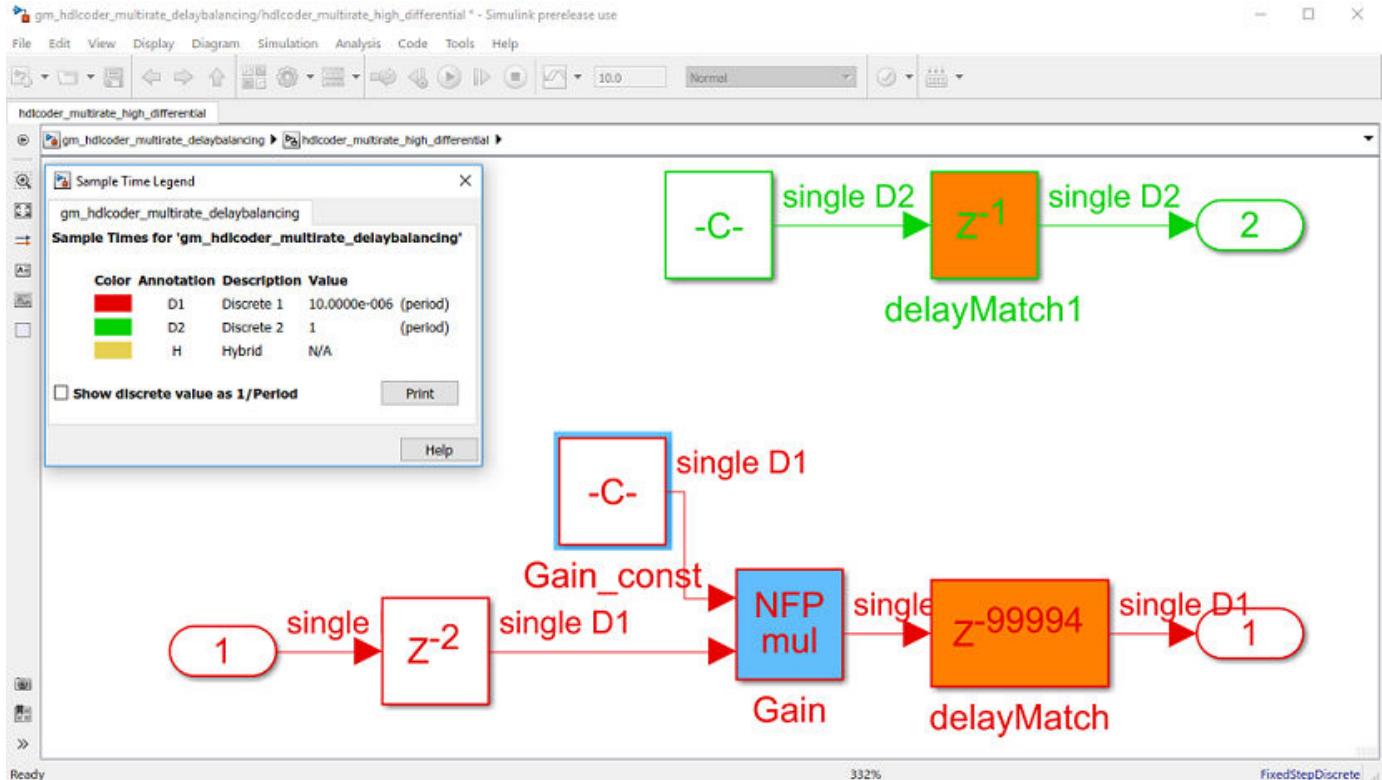
Please note in the below design there are two islands of logics, both running at different rates. The rate differential between the two rates is 10E-06, which is a very high number and possibly unrealistic for practical FPGA design. This model has a floating-point Gain block, a multi-cycle operator, in the fast-clock region.



Running code generation on this model, we get:

```
>> makehdl(gcb)
## Generating HDL for 'hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential'.
## Using the config set for model hdlcoder\_multirate\_delaybalancing for HDL code generation parameters.
## Starting HDL check.
## The code generation and optimization options you have chosen have introduced additional pipeline delays.
## The delay balancing feature has automatically inserted matching delays for compensation.
## The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
## Output port 0: 100000 cycles.
## Output port 1: 1 cycles.
## Begin VHDL Generation for 'hdlcoder_multirate_delaybalancing'.
## Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential/nfp_mul_comp as hdlsrc\hdlcoder\_multirate\_delaybalancing\nfp\_mul\_comp.vhd.
## Working on hdlcoder_multirate_high_differential_tc as hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_high\_differential\_tc.vhd.
## Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_high_differential as hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_high\_differential.vhd.
## Generating package file hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_high\_differential\_pkg.vhd.
## Creating HDL Code Generation Check Report hdlcoder\_multirate\_high\_differential\_report.html
## HDL check for 'hdlcoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
## HDL code generation complete.
```

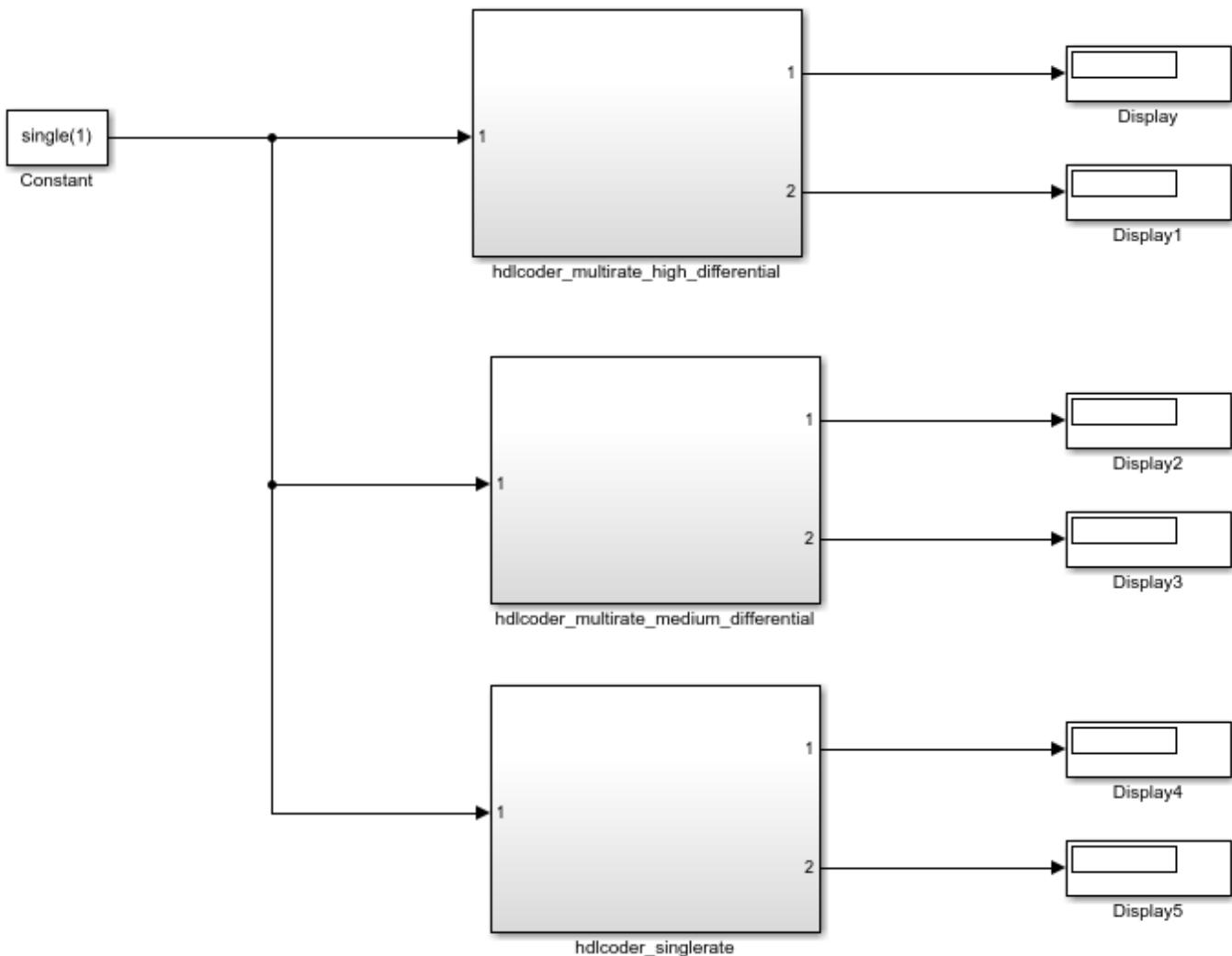
The compiled generated model looks as below. Please note that the high output latency on the fast clock rate region of the design are added to balance delays across multiple output paths of the system.



The high number of registers in the fast clock rate region has an undesired effect post HDL-code generation: # Generated HDL files are by itself very large. # The large number of pipeline registers will make fitting the design into an FPGA improbable.

The following sections of this document create a general awareness of the resource constraint that multi-rate models can create when used in the presence of multi-cycle operations, and provides few recommendations for optimum resource usage.

```
open_system('hdlcoder_multirate_delaybalancing');
```



Guidelines for the users

User Simulink models may have different clock-rate paths due to different modeling reasons. In the presence of optimizations, like I/O pipelining, distributed pipelining, streaming and/or sharing, or multi-cycle operations, like floating-point IPs, fixed-point math functions like sqrt or divide, pipelines are introduced which are applied at the same rate at which the signal path operates.

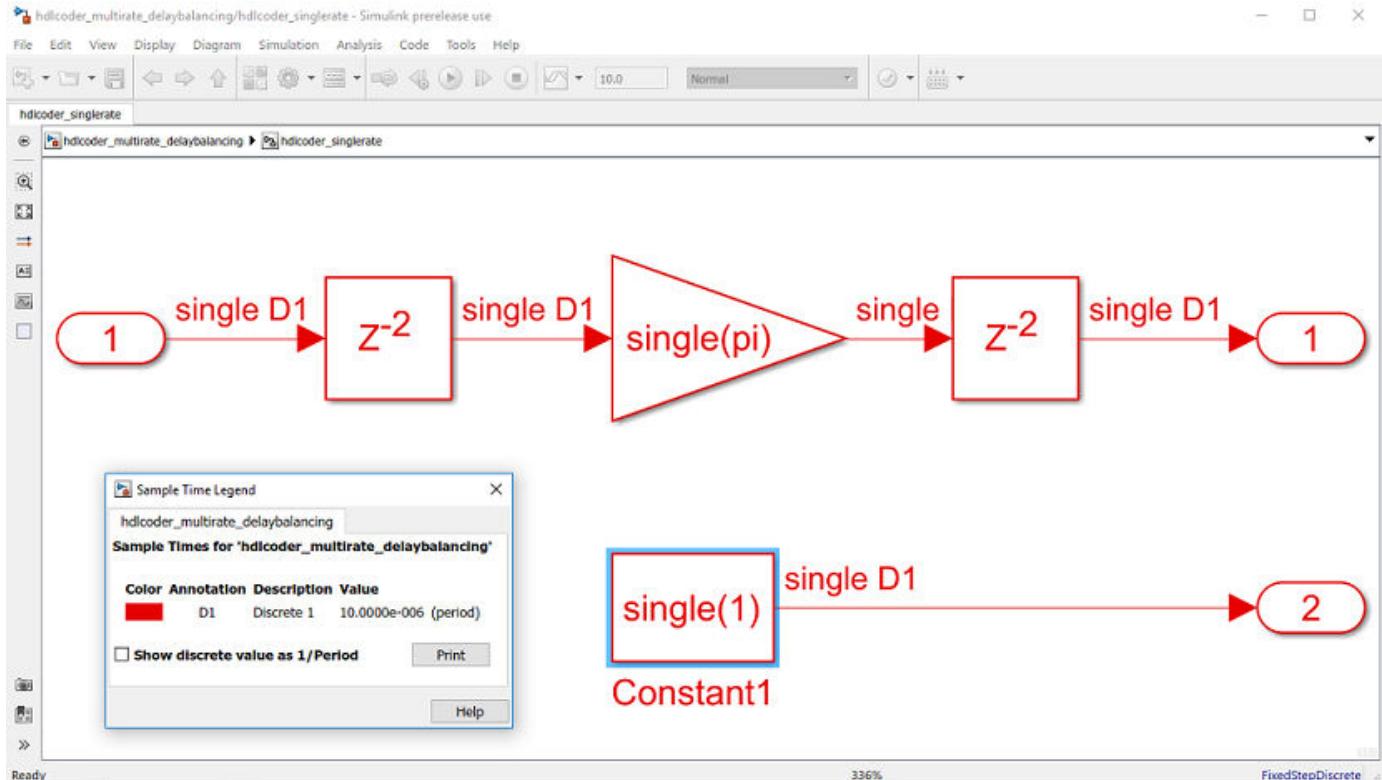
Introducing any additional pipelining introduces undesirable latency overhead that needs to be balanced across multiple output paths, operating at different rates. If the ratio difference between the fastest and slowest clock rate in the Simulink model is very large, it causes a large number of registers to be generated in the final HDL code. The HDL files become large and the design may not even fit into an FPGA.

Recommendation #1: Remove unintentional multi-rates

The user may be unaware of the undesirable effect that the rate differential of his model has on HDL code. For instance, in the above model, the sample rate specified on the constant block was not given

due consideration and set it to value that caused a rate differential of 10E06 with the base model rate. Such a high 'rate differential' seemed **unintentional**.

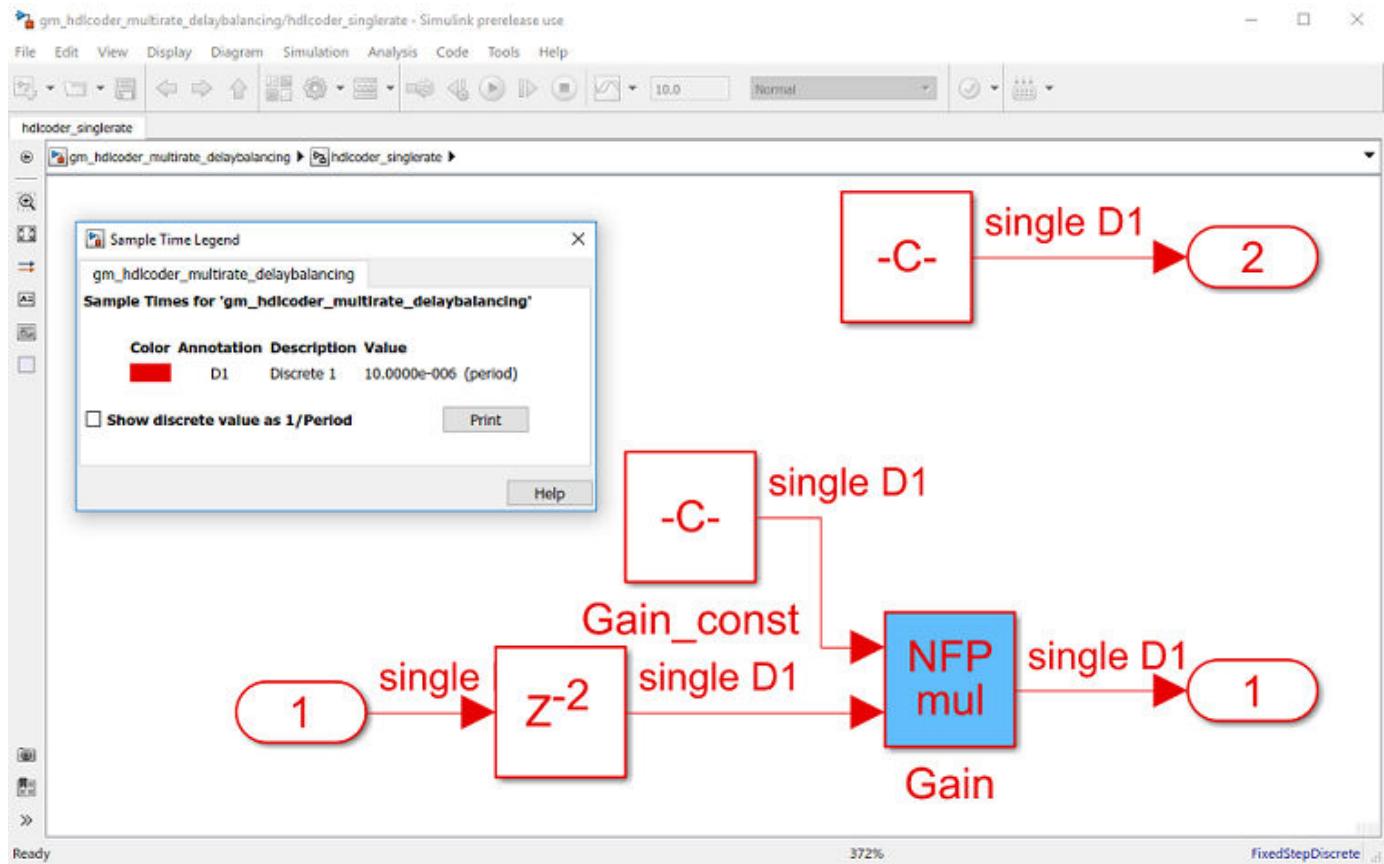
Our **suggestion** would be to change the sample rate of the constant block to run at the **same rate as the base model**, for such a situation.



Running code generation on this model, we get:

```
>> makehdl(gcb)
## Generating HDL for 'hdlicoder_multirate_delaybalancing/hdlicoder_singlerate'.
## Using the config set for model hdlicoder\_multirate\_delaybalancing for HDL code generation parameters.
## Starting HDL check.
## The code generation and optimization options you have chosen have introduced additional pipeline delays.
## The delay balancing feature has automatically inserted matching delays for compensation.
## The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
## Output port 0: 6 cycles.
## Output port 1: 6 cycles.
## Begin VHDL Code Generation for 'hdlicoder_multirate_delaybalancing'.
## Working on hdlicoder_multirate_delaybalancing/hdlicoder_singlerate/nfp_mul_comp as hdlsrc\hdlicoder\_multirate\_delaybalancing\nfp\_mul\_comp.vhd.
## Working on hdlicoder_multirate_delaybalancing/hdlicoder_singlerate as hdlsrc\hdlicoder\_multirate\_delaybalancing\hdlicoder\_singlerate.vhd.
## Generating package file hdlsrc\hdlicoder\_multirate\_delaybalancing\hdlicoder\_singlerate\_pkg.vhd.
## Creating HDL Code Generation Check Report hdlicoder\_singlerate\_report.html
## HDL check for 'hdlicoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
## HDL code generation complete.
```

Please note that the output latency numbers have decreased significantly. The compiled generated model looks as below.

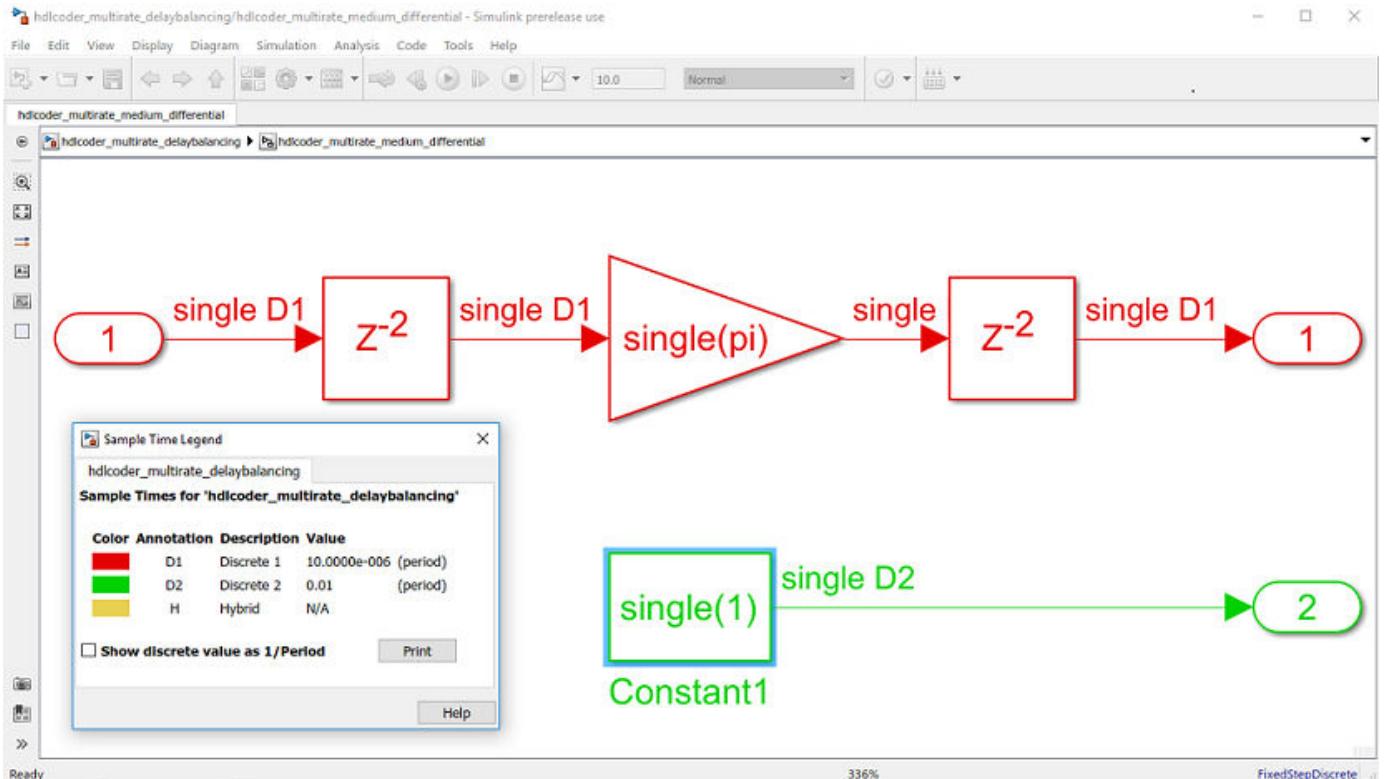


There is no undesirable high number of registers.

Recommendation #2: Keep rate differential practical

If **multi-rate is a desirable property** that user needs, the user needs to consider making the rate differential as **practical** as possible.

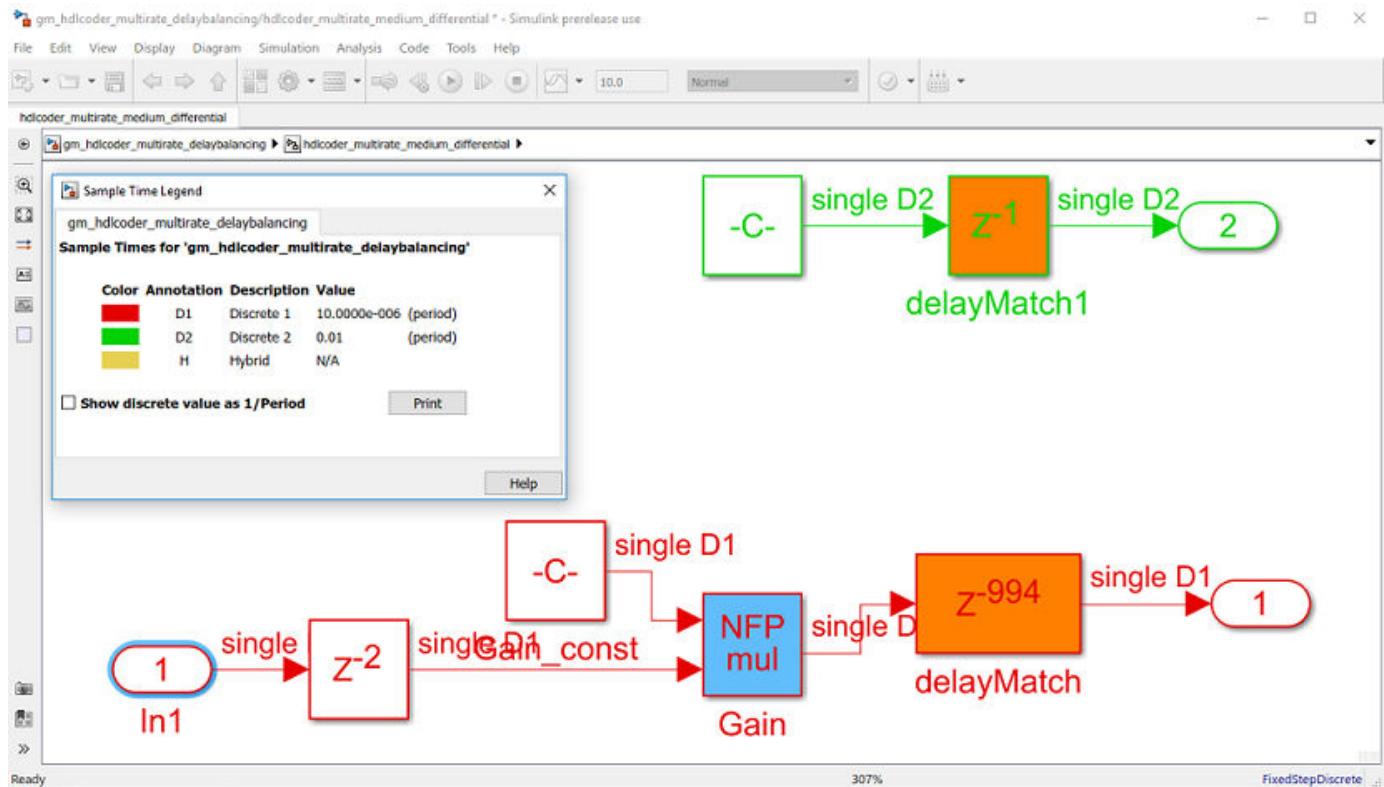
For instance, if one path of the design running at 'ns' and other path of his design is at 'us' is a desirable feature of the design, the user can still choose to have multi-rate paths in his model with the awareness that delay balancing may cause high number of registers.



Running code generation on this model, we get:

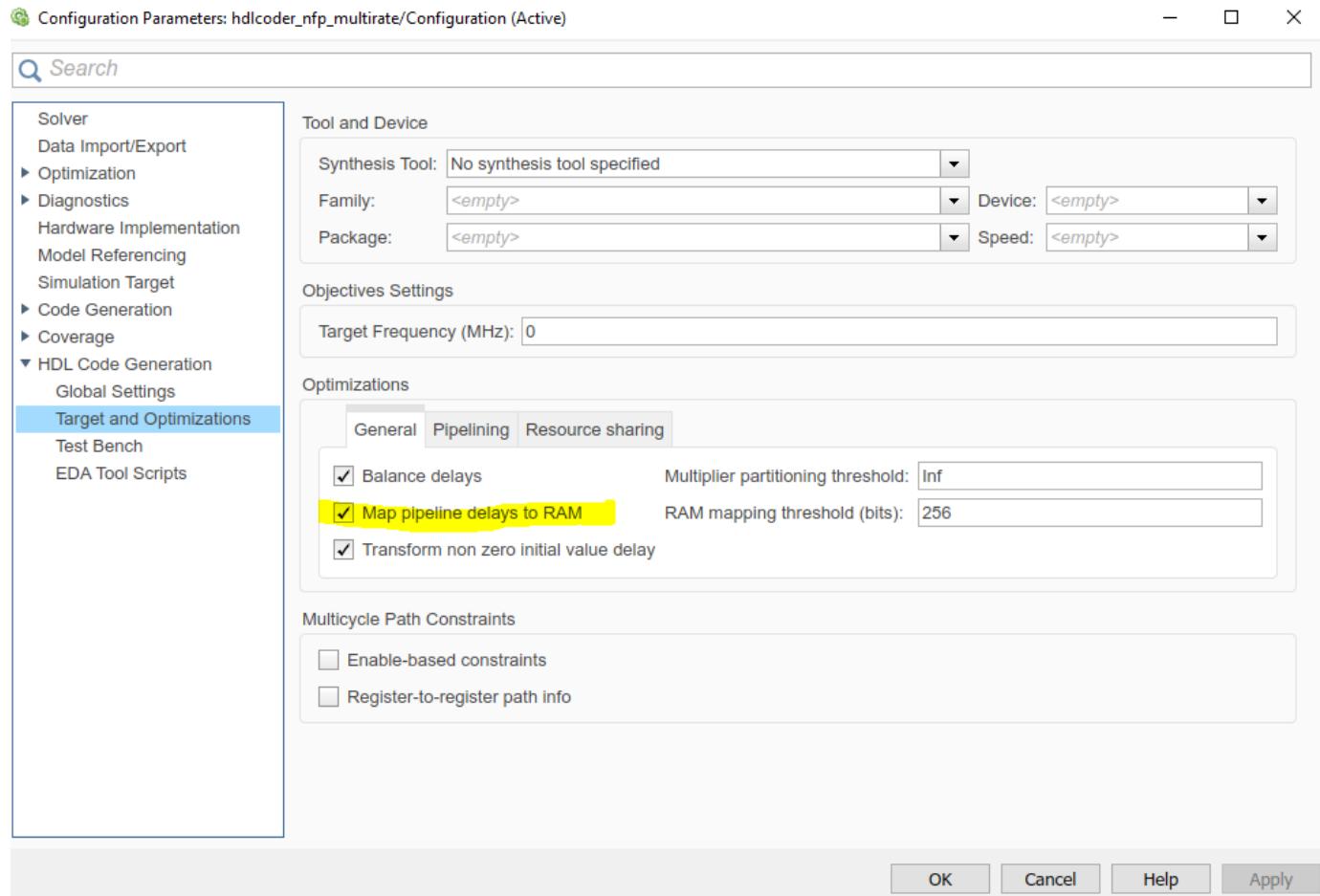
```
>> makehdl(gcb)
## Generating HDL for 'hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential'.
## Using the config set for model hdlcoder\_multirate\_delaybalancing for HDL code generation parameters.
## Starting HDL check.
## The code generation and optimization options you have chosen have introduced additional pipeline delays.
## The delay balancing feature has automatically inserted matching delays for compensation.
## The DUT requires an initial pipeline setup latency. Each output port experiences these additional delays.
## Output port 0: 1000 cycles.
## Output port 1: 1 cycles.
## Begin VHDL Code Generation for 'hdlcoder_multirate_delaybalancing'.
## Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential/nfp_mul_comp as hdlsrc\hdlcoder\_multirate\_delaybalancing\nfp\_mul\_comp.vhd.
## Working on hdlcoder_multirate_medium_tc as hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_medium\_differential\_tc.vhd.
## Working on hdlcoder_multirate_delaybalancing/hdlcoder_multirate_medium_differential as hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_medium\_differential.vhd.
## Generating package file hdlsrc\hdlcoder\_multirate\_delaybalancing\hdlcoder\_multirate\_medium\_differential\_pkg.vhd.
## Creating HDL Code Generation Check Report hdlcoder\_multirate\_medium\_differential\_report.html
## HDL check for 'hdlcoder_multirate_delaybalancing' complete with 0 errors, 0 warnings, and 0 messages.
## HDL code generation complete.
```

The compiled generated model looks like the figure below. In the generated model and HDL code, we will have close to 1000 registers in the fast clock rate output path. The additional cost of registers is **not unusual** for control logics that are running 1000x faster than the system. The user just needs to be aware of the hardware resource constraints for such a model.



To optimize on the total number of registers in FPGA, the user can also use the HDLCoder "Mapping pipeline delays to RAM" feature. Doing this will tradeoff RAM resource to save on logic area.

```
>> hdlset_param(gcs, 'MapPipelineDelaysToRAM', 'on');
```



Find Feedback Loops

In this section...

["Specify Highlighting of Feedback Loops" on page 24-90](#)

["Remove Highlighting" on page 24-90](#)

["Limitations" on page 24-91](#)

Feedback loops in your Simulink design can inhibit delay balancing and optimizations such as resource sharing and streaming.

To find feedback loops in your design that are inhibiting optimizations, you can generate and run a MATLAB script that highlights one or more feedback loops in your original model and the generated model. When you run the script, different feedback loops are highlighted in different colors. The feedback loop highlighting script is saved in the same target folder as the HDL code.

After you generate code, if feedback loops are inhibiting optimizations, the command window shows a link that you can click to highlight feedback loops. If you generate an Optimization Report, the report also contains a link you can click to highlight feedback loops.

The script can highlight feedback loops that are inhibiting the following optimizations:

- Resource sharing
- Streaming
- MATLAB variable pipelining
- Delay balancing

Specify Highlighting of Feedback Loops

By default, highlighting of feedback loops is enabled. This setting is available:

- In the Configuration Parameters dialog box, on the **HDL Code Generation > Global Settings > Advanced** tab, select **Highlight feedback loops inhibiting delay balancing and optimizations**.
- To generate a feedback loop highlighting script programmatically, use the **HighlightFeedbackLoops** property with **makehdl** or **hdlset_param**. For example, to generate a feedback loop highlight script for a model, *myModel*, enter:

```
hdlset_param ('myModel', 'HighlightFeedbackLoops', 'on');
```

Remove Highlighting

By default, HDL Coder generates a script to highlight feedback loops and a script to clear the highlighting of feedback loops in your model. You can turn off highlighting using either of these ways:

- Click the **clearhighlighting** script in the MATLAB Command Window
- In the Simulink Toolstrip, select **Debug > Trace Signal**.

Limitations

- Feedback loop highlighting cannot highlight blocks that have names that contain a single quote (').
- In some cases, feedback loop highlighting might highlight a subsystem or one block instead of the lowest-level block.

See Also

More About

- “Diagnostic Parameters for Optimizations” on page 17-89
- “Create and Use Code Generation Reports” on page 25-2
- “Generated Model and Validation Model” on page 24-10

Hierarchy Flattening

In this section...

- “What Is Hierarchy Flattening?” on page 24-92
- “When to Flatten Hierarchy” on page 24-92
- “Considerations” on page 24-92
- “How to Flatten Hierarchy” on page 24-93
- “Limitations for Hierarchy Flattening” on page 24-93
- “Hierarchy Flattening Report” on page 24-94

What Is Hierarchy Flattening?

Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design.

HDL Coder considers blocks within a flattened subsystem to be at the same level of hierarchy, and no longer grouped into separate subsystems. This consideration allows the coder to reorganize blocks for optimization across the original hierarchical boundaries, while preserving functionality.

When to Flatten Hierarchy

To preserve the modularity of the design and have a one-to-one mapping from subsystem name to corresponding HDL module or entity name, do not flatten the hierarchy. The generated HDL code is more readable when you don't flatten the hierarchy.

Flatten hierarchy to:

- Enable more extensive area and speed optimization.
- Reduce the number of HDL output files. For every subsystem that you flatten, HDL Coder generates one less HDL output file.

Considerations

- Before you flatten the hierarchy, you must have the `MaskParameterAsGeneric` property set to `off`. For more information, see “Generate parameterized HDL code from masked subsystem” on page 17-57.
- When you use optimizations such as resource sharing or streaming with hierarchy flattening, in certain cases, HDL Coder might retain the subsystem hierarchy in the generated model. However, the HDL code generated for the flattened subsystems is inlined, which reduces the number of HDL files.
- When you use floating-point data types in `Native Floating Point` mode, HDL Coder might not flatten the hierarchy. This is because floating-point designs generate hundreds of lines of code and inlining the HDL files makes the generated code less readable.

How to Flatten Hierarchy

By default, a subsystem inherits its hierarchy flattening setting from the parent subsystem. However, you can enable or disable flattening for individual subsystems. This table lists options you can specify for hierarchy flattening options for a subsystem are listed in the following table.

Hierarchy Flattening Setting	Description
inherit (default)	Use the hierarchy flattening setting of the parent subsystem. If this subsystem is the highest-level subsystem, do not flatten.
on	Flatten this subsystem.
off	Do not flatten this subsystem, even if the parent subsystem is flattened.

To set hierarchy flattening using the HDL Block Properties dialog box:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. For **FlattenHierarchy**, select **on**, **off**, or **inherit**.

To set hierarchy flattening from the command line, use `hdlset_param`. For example, to turn on hierarchy flattening for a subsystem, `my_dut`:

```
hdlset_param('my_dut', 'FlattenHierarchy', 'on')
```

See also `hdlset_param`.

Limitations for Hierarchy Flattening

A subsystem cannot be flattened if the subsystem is:

- A Synchronous Subsystem or uses the State Control block in Synchronous mode.
- A model reference implementation.
- A Triggered Subsystem when “Use trigger signal as clock” on page 17-41 is enabled.
- A masked subsystem that contains any of the following:
 - Bus.
 - Enumerated data type.
 - Lookup table blocks: 1-D Lookup Table, 2-D Lookup Table, Cosine HDL Optimized, Direct LookupTable (n-D), Prelookup, Sine HDL Optimized, n-D Lookup Table.
 - MATLAB System block.
 - Stateflow blocks: Chart, State Transition Table, Sequence Viewer.
 - Blocks with a pass-through or no-op implementation. See “Pass through, No HDL, and Cascade Implementations” on page 22-56.

Note This option removes subsystem boundaries before code generation. It does not necessarily generate HDL code with a completely flat hierarchy.

Hierarchy Flattening Report

To see the hierarchy flattening information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

The report displays subsystems in your model that have **FlattenHierarchy** set to on and off, hierarchy flattening status, and the HDL files that are inlined. You can use the report to more effectively flatten the subsystem hierarchy and improve opportunities for optimizations such as clock-rate pipelining on the model.

If hierarchy flattening is unsuccessful, the report shows a table that contains subsystems that are not flattened, and reasons for not flattening the subsystem. Subsystems that have a * highlighted beside it indicates whether the HDL files are inlined though hierarchy flattening failed.

See Also

`makehdl`

More About

- “Clock-Rate Pipelining” on page 24-114
- “Resource Sharing” on page 24-32
- “Streaming” on page 24-29

RAM Mapping for Simulink Models

RAM mapping is an area optimization. You can map to RAMs in HDL code by using:

- `UseRAM` to map delays to RAM. For details, see “[UseRAM](#)” on page 22-25.
- `MapPersistentVarsToRAM` to map persistent arrays in a MATLAB Function block to RAM. For details, see “[MapPersistentVarsToRAM](#)” on page 22-16.
- RAM blocks from the HDL Operations library:
 - Single Port RAM
 - Single Port RAM System
 - Dual Port RAM
 - Dual Port RAM System
 - Simple Dual Port RAM
 - Simple Dual Port RAM System
 - Dual Rate Dual Port RAM
- Blocks with a RAM implementation.

See Also

Simulink Configuration Parameters

“[RAM Mapping Parameters](#)” on page 15-7

More About

- “[UseRAM](#)” on page 22-25
- “[MATLAB Function Block Design Patterns for HDL](#)” on page 29-19
- “[MapPersistentVarsToRAM](#)” on page 22-16
- “[RAM Mapping With the MATLAB Function Block](#)” on page 24-96

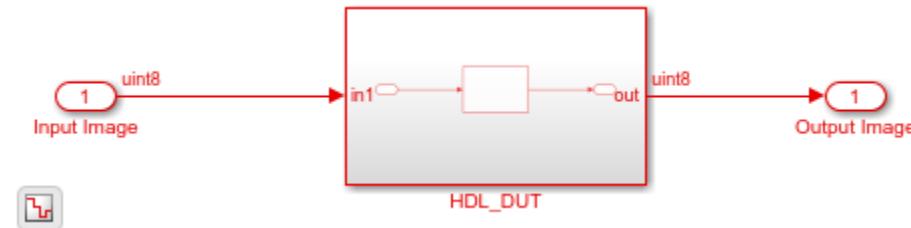
RAM Mapping With the MATLAB Function Block

This example shows how to map persistent arrays to RAM by using the MapPersistentVarsToRAM block-level parameter. The RAM size must be greater than or equal to the RAMMappingThreshold. The resource report shows the difference in area improvements resulting from RAM mapping.

Line Buffer Model

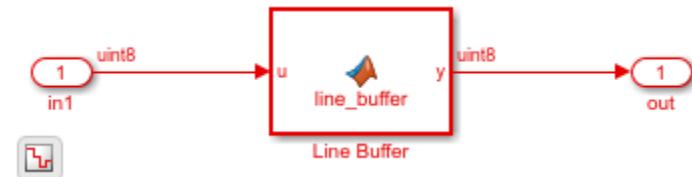
Open the model `hdlcoder_ram_mapping_matlab_function`.

```
open_system('hdlcoder_ram_mapping_matlab_function')
set_param('hdlcoder_ram_mapping_matlab_function', 'SimulationCommand', 'Update')
```



The DUT Subsystem in the model drives a Line Buffer MATLAB Function block.

```
open_system('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```



To see the MATLAB® code implementation of the line buffer, open the MATLAB Function block.

```
open_system('hdlcoder_ram_mapping_matlab_function/HDL_DUT/Line Buffer')
```

```
%%%%%
% Line buffer: Uses a persistent array to store the image
%%%%%

function y = line_buffer(u)
%#codegen

persistent u_d ctr;

if isempty(u_d)
    u_d = uint8(zeros(1,80)); % You can map this to RAM
    ctr = uint8(1);
end

y = u_d(ctr);
u_d(ctr) = u;

if ctr == uint8(80)
    ctr = uint8(1);
else
    ctr = ctr + 1;
end

end
```

Generate HDL Code

1. Enable generation of the resource utilization report. The report displays the number of adders, subtractors, multipliers, registers, and RAMs that the design consumes.

```
hdlset_param('hdlcoder_ram_mapping_matlab_function', 'resourcereport', 'on')
```

2. Generate HDL code for the HDL_DUT Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

HDL Coder™ displays the Code Generation Report. In the report, select the **High-Level Resource Report** section. The design consumes 81 registers and 648 1-bit registers. By default, the MapPersistentVarsToRAM property is disabled and the code generator does not infer or consume RAM resources.

Multipliers	0
Adders/Subtractors	3
Registers	81
Total 1-Bit Registers	648
RAMs	0
Multiplexers	1
I/O Bits	20
Static Shift operators	0
Dynamic Shift operators	0

Enable RAM Mapping and Generate HDL Code

1. Enable the MapPersistentVarsToRAM HDL parameter on the MATLAB Function block.

```
ml_subsys = 'hdlcoder_ram_mapping_matlab_function/HDL_DUT/Line Buffer';
hdlset_param(ml_subsys, 'MapPersistentVarsToRAM', 'on')
```

2. Generate HDL code for the HDL_DUT Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

In the Code Generation Report, select the **High-Level Resource Report** section. The design consumes one register, eight 1-bit registers, and one RAM. The number of RAMs inferred depends on the RAMMappingThreshold that you specify. See RAM mapping threshold(bits).

Multipliers	0
Adders/Subtractors	3
Registers	1
Total 1-Bit Registers	8
RAMs	1
Multiplexers	3
I/O Bits	20
Static Shift operators	0
Dynamic Shift operators	0

RAM Mapping with MATLAB Datapath Architecture

The MATLAB Datapath architecture treats the MATLAB Function block like a regular Subsystem. The architecture converts the MATLAB code that you wrote to a dataflow representation in

Simulink® HDL Coder can then more widely use optimizations across the MATLAB Function block with other Simulink blocks in your model.

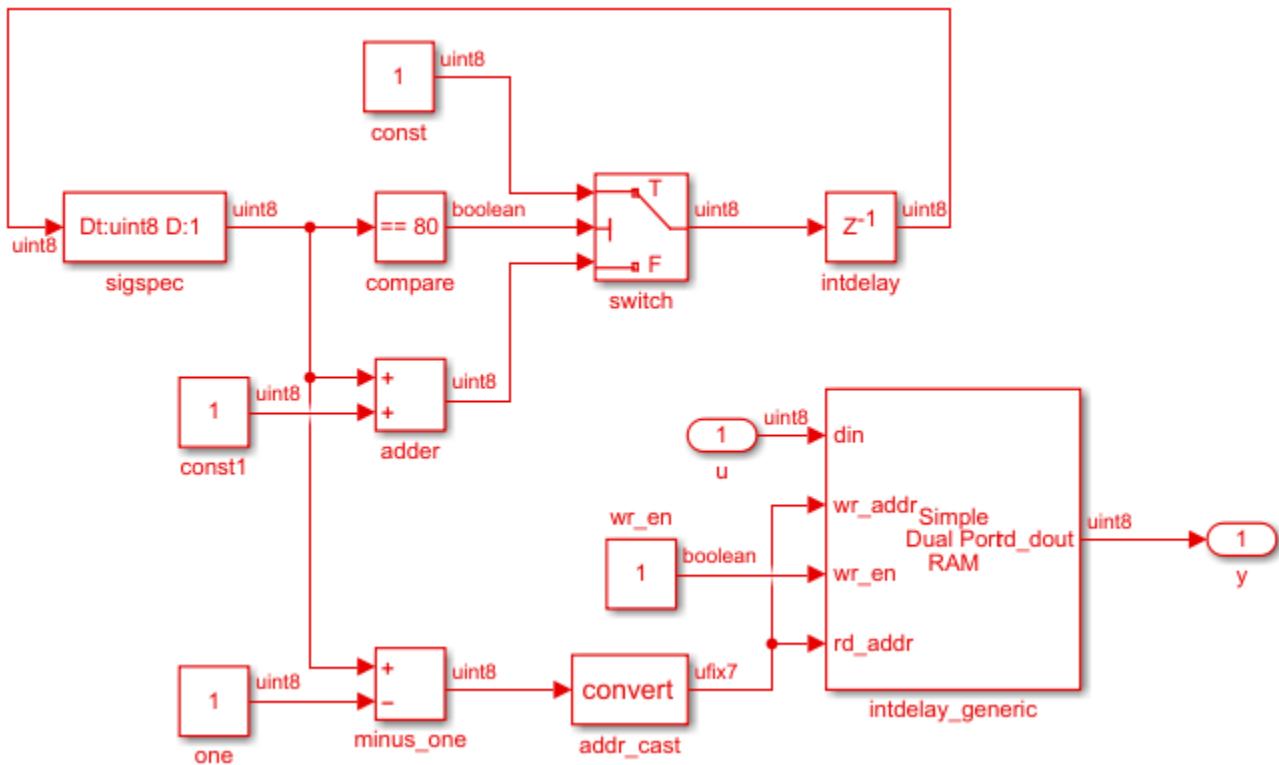
1. Enable the MATLAB Datapath HDL architecture and then set the MapPersistentVarsToRAM parameter on the MATLAB Function block.

```
hdlset_param(ml_subsys, 'Architecture', 'MATLAB Datapath')
hdlset_param(ml_subsys, 'MapPersistentVarsToRAM', 'on')
```

2. Generate HDL code for the HDL_DUT Subsystem.

```
makehdl('hdlcoder_ram_mapping_matlab_function/HDL_DUT')
```

The **High-Level Resource Report** indicates that the design consumes the same number of resources as the design that used the default architecture of the MATLAB Function block. To see how MATLAB Datapath architecture modifies the MATLAB code to a Simulink dataflow representation, open the generated model `gm_hdlcoder_ram_mapping_matlab_function` and navigate to the `HDL_DUT` Subsystem. There is a Line Buffer Subsystem in place of the MATLAB Function block. Inside the Subsystem block is the dataflow representation that displays a RAM block inferred.



To learn about design patterns that enable efficient RAM mapping of persistent arrays in MATLAB Function blocks, see the `eml_hdl_design_patterns/RAMs` library.

See Also

More About

- “RAM Mapping Parameters” on page 15-7
- “MATLAB Function Block Design Patterns for HDL” on page 29-19
- “MapPersistentVarsToRAM” on page 22-16

Distributed Pipelining

In this section...

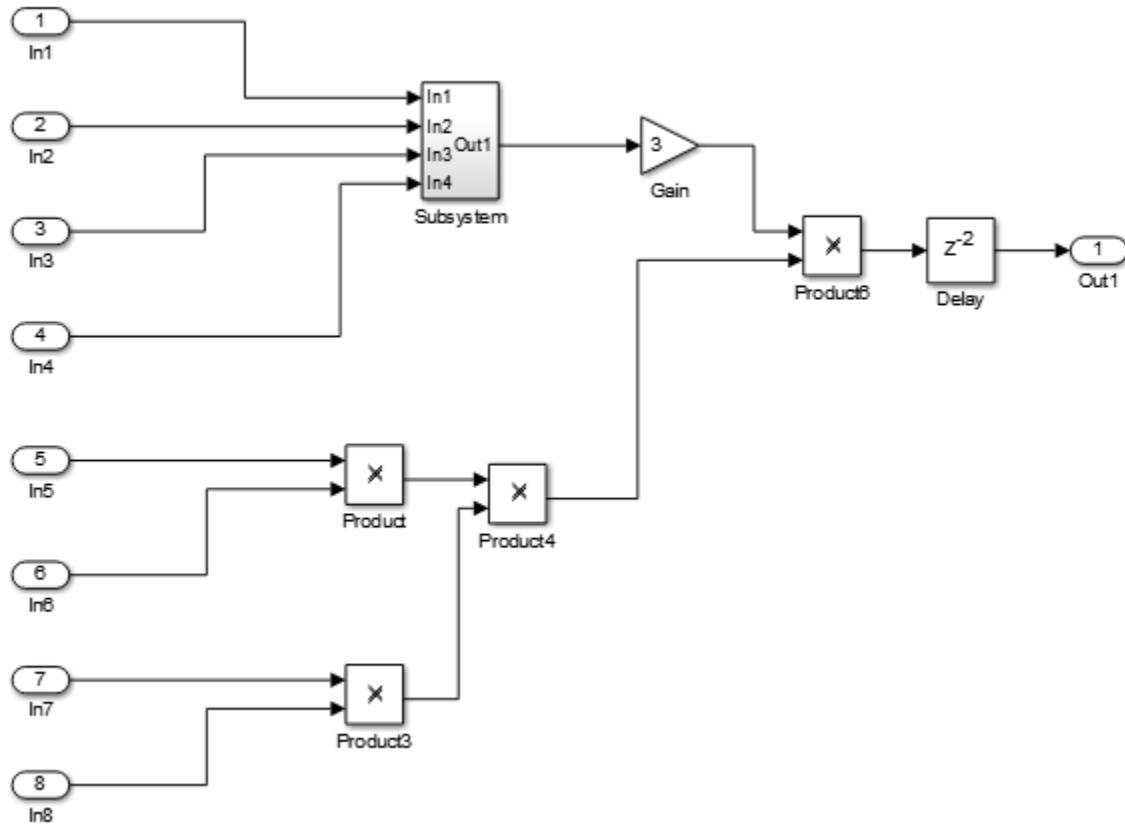
- ["What Is Distributed Pipelining?" on page 24-101](#)
- ["Benefits and Costs of Distributed Pipelining" on page 24-102](#)
- ["Requirements for Distributed Pipelining" on page 24-102](#)
- ["Specify Distributed Pipelining" on page 24-103](#)
- ["Limitations of Distributed Pipelining" on page 24-103](#)
- ["Distributed Pipelining Report" on page 24-104](#)
- ["Selected Bibliography" on page 24-104](#)

What Is Distributed Pipelining?

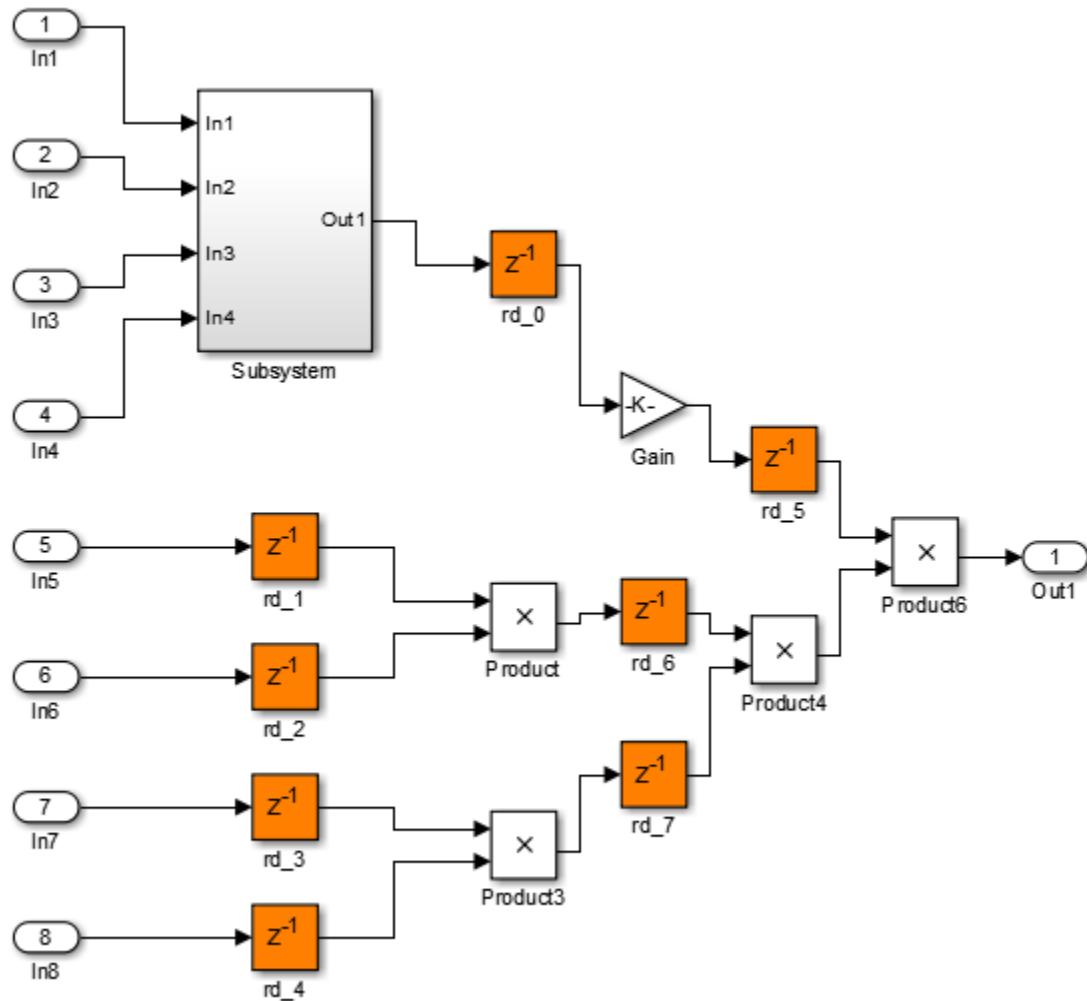
Distributed pipelining, or register retiming, is a speed optimization that moves existing delays in a design to reduce the critical path while preserving functional behavior.

The HDL Coder software uses an adaptation of the Leiserson-Saxe retiming algorithm.

For example, in the following model, there is a delay of 2 at the output.



The following diagram shows the generated model after distributed pipelining redistributes the delay to reduce the critical path.



Benefits and Costs of Distributed Pipelining

Distributed pipelining can reduce your design's critical path, enabling you to use a higher clock rate and increase throughput.

However, distributed pipelining requires your design to contain a number of delays. If you need to insert additional delays in your design to enable distributed pipelining, this increases the area and the initial latency of your design.

Requirements for Distributed Pipelining

Distributed pipelining requires your design to contain delays or registers that can be redistributed. You can use input pipelining or output pipelining to insert more registers.

If your design does not meet your timing requirements at first, try adding more delays or registers to improve your results.

Specify Distributed Pipelining

You can specify distributed pipelining for a subsystem, and Stateflow charts and MATLAB Function blocks within a subsystem. See “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37.

To specify distributed pipelining using the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. Set **DistributedPipelining** to **on** and click **OK**.
- Right-click the Subsystem and select **HDL Code > HDL Block Properties**. Set **DistributedPipelining** to **on** and click **OK**.

To enable distributed pipelining, on the command line, enter:

```
hdlset_param('path/to/block', 'DistributedPipelining', 'on')
```

Tip Output data could be in an invalid state initially if you insert pipeline registers. To avoid test bench errors resulting from initial invalid samples, disable output checking for those samples. For more information, see “Ignore output data checking (number of samples)” on page 19-19.

Limitations of Distributed Pipelining

The distributed pipelining optimization has the following limitations:

- Your pipelining results might not be optimal in hardware because the operator latencies in your target hardware may differ from the estimated operator latencies used by the distributed pipelining algorithm.
- The HDL Coder software generates pipeline registers at the outputs in the following situations instead of distributing the registers to reduce critical path:
 - Stateflow chart containing a state, local variable, or a matrix with statically unresolvable index.
- HDL Coder distributes pipeline registers around the following blocks instead of within them:
 - Model
 - Sum (Cascade implementation)
 - Product (Cascade implementation)
 - MinMax
 - Upsample
 - Downsample
 - Rate Transition
 - Zero-Order Hold
 - Reciprocal Sqrt (RecipSqrtNewton implementation)
 - Trigonometric Function (CORDIC Approximation)
 - Single Port RAM
 - Dual Port RAM
 - Simple Dual Port RAM

- If you enable distributed pipelining for a subsystem that contains these blocks, HDL Coder generates a message during code generation. To fix this message, place these blocks inside one or more subsystems within the original subsystem, and disable hierarchical distributed pipelining. HDL Coder distributes pipeline registers around nested subsystems.
 - M-PSK Demodulator Baseband
 - M-PSK Modulator Baseband
 - QPSK Demodulator Baseband
 - QPSK Modulator Baseband
 - BPSK Demodulator Baseband
 - BPSK Modulator Baseband
 - PN Sequence Generator
 - Repeat
 - HDL Counter
 - LMS Filter
 - Sine Wave
 - Viterbi Decoder
 - Triggered Subsystem
 - Counter Limited
 - Counter Free-Running
 - Frame Conversion

Distributed Pipelining Report

To see the distributed pipelining information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate the optimization report, in the **Distributed Pipelining** section, you see the effect of the distributed pipelining optimization. If distributed pipelining is unsuccessful, the report shows diagnostic messages and offending blocks that caused distributed pipelining to fail.

If distributed pipelining is successful, the report displays comparative listings of registers before and after you apply the distributed pipelining transform.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

See Also

More About

- "Pipelining Parameters" on page 15-9
- "Diagnostic Parameters for Optimizations" on page 17-89

Hierarchical Distributed Pipelining

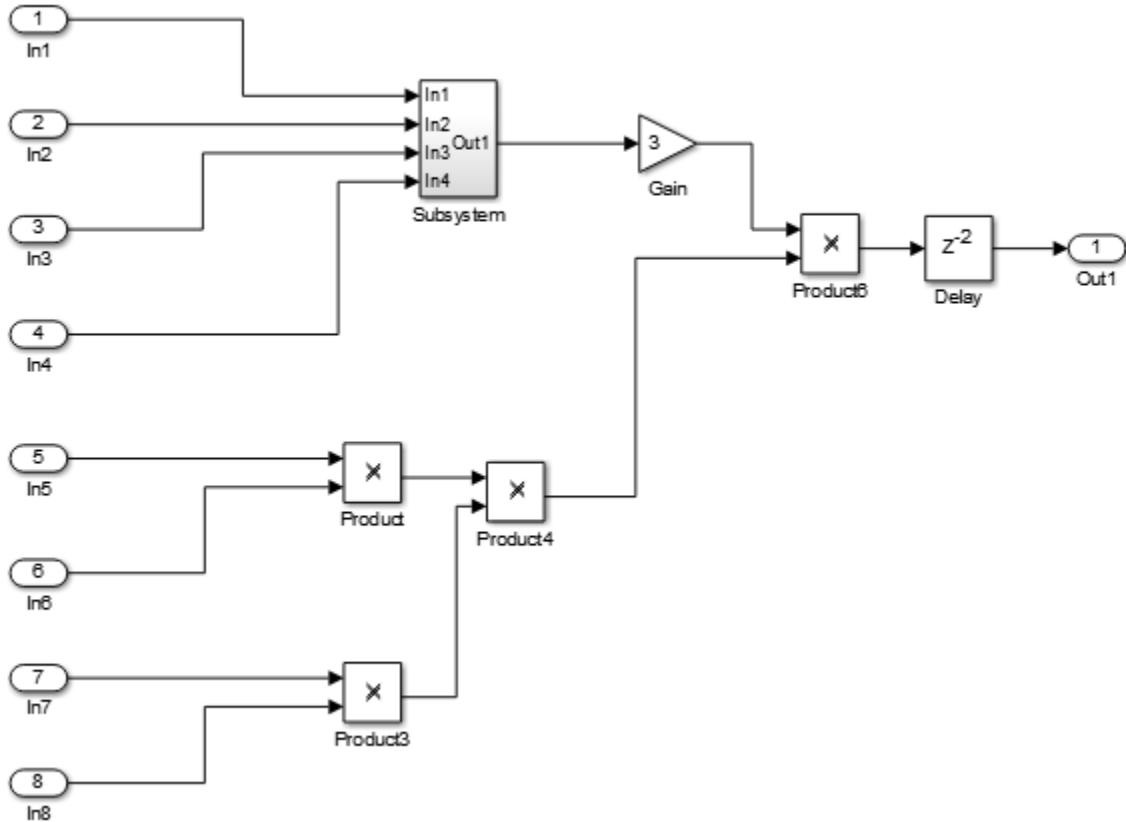
What Is Hierarchical Distributed Pipelining?

Hierarchical distributed pipelining extends the scope of distributed pipelining by moving delays across hierarchical boundaries within a subsystem while preserving subsystem hierarchy.

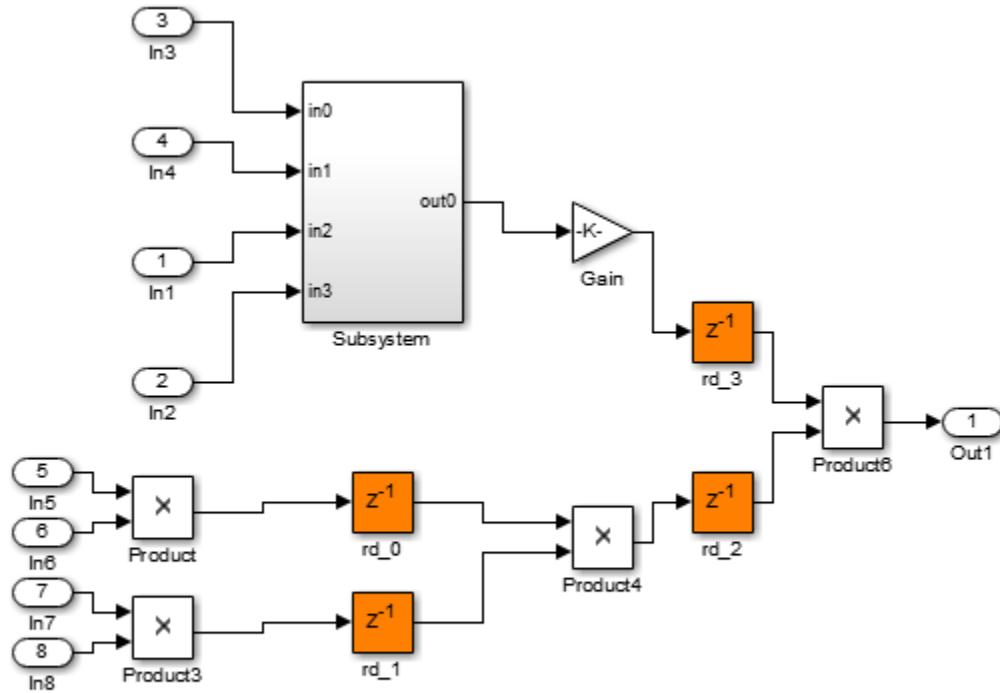
If a subsystem in the hierarchy does not have distributed pipelining enabled, HDL Coder does not move delays across that subsystem.

How Hierarchical Distributed Pipelining Works

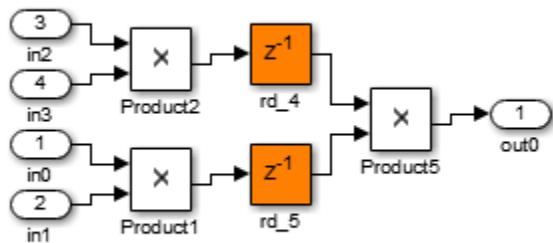
For example, the following model has one level of subsystem hierarchy:



The following diagram shows the model after applying hierarchical distributed pipelining:



The subsystem now contains pipeline registers:



Benefits of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining enables distributed pipelining to operate on a larger part of your design, which increases the chance that distributed pipelining can further reduce your critical path.

Hierarchical distributed pipelining preserves the original subsystem hierarchy, which enables you to trace the changes that occur during pipelining for nested Subsystem blocks.

Specify Hierarchical Distributed Pipelining

You can specify hierarchical distributed pipelining for your model. To specify hierarchical distributed pipelining using the UI, in the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click the **Subsystem** and then click **HDL Block Properties**. Set **DistributedPipelining** to **on**

- 1** In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2** Click **Settings**. In the **HDL Code Generation > Optimization > Pipelining** tab, select **Hierarchical distributed pipelining** and click **OK**.

To enable hierarchical distributed pipelining, on the command line, enter:

```
hdlset_param('modelname', 'HierarchicalDistPipelining', 'on')
```

Limitations of Hierarchical Distributed Pipelining

Hierarchical distributed pipelining must be disabled if your DUT subsystem contains a model reference.

Hierarchical Distributed Pipelining Report

To see the hierarchical distributed pipelining information in the report, before you generate code for each subsystem or model reference, enable the optimization report. To enable this report, in the **HDL Code** tab, select **Report Options**, and then select **Generate optimization report**.

When you generate the optimization report, in the **Distributed Pipelining** section, you see the effect of the hierarchical distributed pipelining optimization. If hierarchical distributed pipelining is unsuccessful, the report shows diagnostic messages and offending blocks that caused hierarchical distributed pipelining to fail.

If hierarchical distributed pipelining is successful, the report displays colored sections to distinguish between different regions where HDL Coder applied hierarchical distributed pipelining.

Selected Bibliography

Leiserson, C.E, and James B. Saxe. "Retiming Synchronous Circuitry." *Algorithmica*. Vol. 6, Number 1, 1991, pp. 5-35.

Distributed Pipelining: Speed Optimization

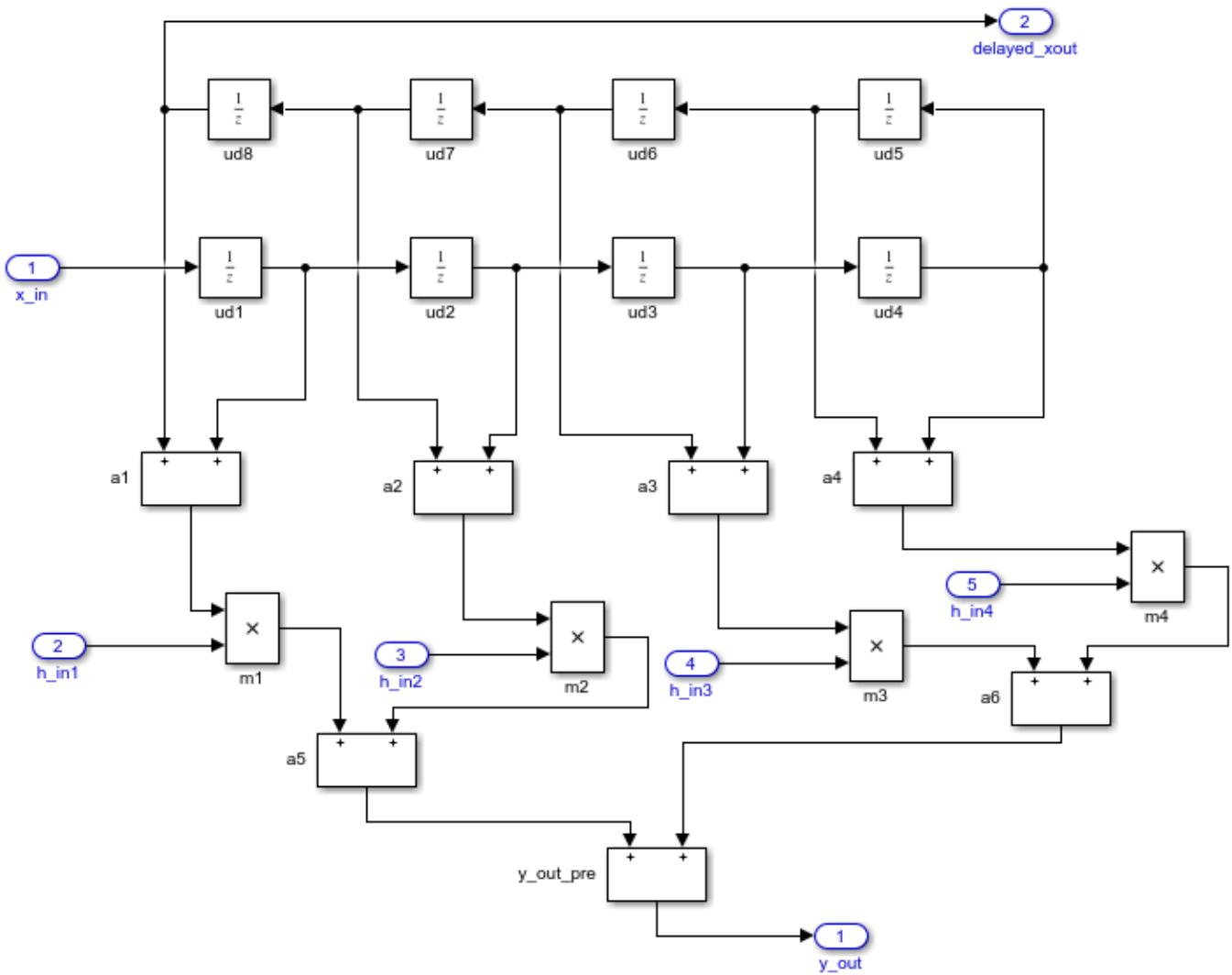
This example shows how to use distributed pipelining to optimize a design for speed in HDL Coder.

Introduction

Distributed pipelining is a subsystem-wide optimization supported by HDL Coder for achieving high clock speed hardware. By turning on 'Distributed Pipelining', the coder redistributes the input pipeline registers, output pipeline register of the subsystem and the registers in the subsystem to appropriate positions to minimize the combinatorial logic between registers and maximize the clock speed of the chip synthesized from the generated HDL code.

Consider the following example model of a symmetric FIR filter. The combinatorial logic from an input or a register to an output or another register contains a product block and an adder tree. Distributed pipelining will move the output registers set at the subsystem level to reduce the levels of the combinatorial logic.

```
bdclose all;
load_system('sfir_fixed');
open_system('sfir_fixed/symmetric_fir');
```



Setting Output Pipeline Stage

To increase the clock speed, the user can set a number of pipeline stages for any subsystem. Without turning on distributed pipelining, the specified number of registers will be added to each of the output ports of the subsystem. Some synthesis tools support features like retiming that optimize the position of the registers during synthesis.

In this example, the subsystem output pipeline register is set to 5.

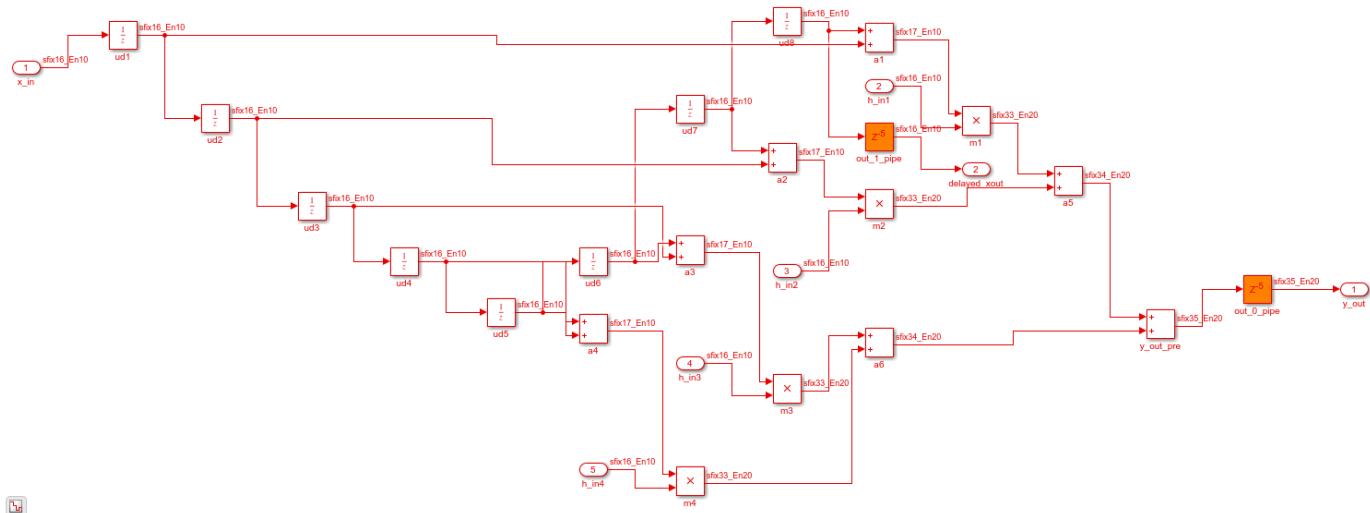
The code-generation model explicitly reflects the inserted register at output ports of the subsystem (highlighted in orange).

```

hdlset_param('sfir_fixed/symmetric_fir', 'OutputPipeline', 5);
makehdl('sfir_fixed/symmetric_fir');
open_system('gm_sfir_fixed/symmetric_fir');
set_param('gm_sfir_fixed', 'SimulationCommand', 'update');

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

```
### Starting HDL check.
### 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 0: 5 cycles.
### Output port 1: 5 cycles.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'sfir_fixed' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Setting Distributed Pipelining

Distributed pipelining is one of the subsystem block options. Once turned on, the registers in the subsystem, including output pipeline registers and input pipeline registers, will be repositioned to achieve best clock speed. It is equivalent to retiming at subsystem level.

The code-generation model explicitly reflects the distributed registers in the subsystem (highlighted in orange).

```
hdlset_param('sfir_fixed/symmetric_fir', 'DistributedPipelining', 'on');
makehdl('sfir_fixed/symmetric_fir');
open_system('gm_sfir_fixed/symmetric_fir');
set_param('gm_sfir_fixed', 'SimulationCommand', 'update');

### Generating HDL for 'sfir_fixed/symmetric_fir'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('sfir_fixed', {
### Starting HDL check.
### 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 0: 5 cycles.
### Output port 1: 5 cycles.
### Begin VHDL Code Generation for 'sfir_fixed'.
### Working on sfir_fixed/symmetric_fir as hdlsrc\sfir_fixed\symmetric_fir.vhd.
### Generating package file hdlsrc\sfir_fixed\symmetric_fir_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
```

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

Opportunities for Distributed Pipelining Across Subsystem Hierarchies

Since 'Distributed Pipelining' is a subsystem-level parameter, different subsystems at different levels of the hierarchy can specify different pipeline stage values and different 'Distributed Pipelining' settings. By default, the coder distributes only registers of the specified subsystem in this subsystem, not through the lower level subsystems. If cross hierarchy distribution is desired, users can set the 'Distributed Pipelining' for lower subsystems to 'On', then turn on the global option 'Hierarchical Distributed Pipelining'. When the local and global options are on, the entire subsystem, including the lower level subsystems, will be considered as a single subsystem when registers are distributed.

```
bdclose all;
```

Constrained Output Pipelining

In this section...

- “What Is Constrained Output Pipelining?” on page 24-112
- “When to Use Constrained Output Pipelining” on page 24-112
- “Requirements for Constrained Output Pipelining” on page 24-112
- “Specify Constrained Output Pipelining” on page 24-112
- “Limitations of Constrained Output Pipelining” on page 24-113

What Is Constrained Output Pipelining?

With constrained output pipelining, you can specify a nonnegative number of registers at the outputs of a block.

Constrained output pipelining does not add registers, but instead redistributes existing delays within your design to try to meet the constraint. If HDL Coder cannot meet the constraint with existing delays, it reports the difference between the number of desired and actual output registers in the timing report.

Distributed pipelining does not move registers you specify with constrained output pipelining.

When to Use Constrained Output Pipelining

Use constrained output pipelining when you want to place registers at specific locations in your design. This can enable you to optimize the speed of your design.

For example, if you know where the critical path is in your design and want to reduce it, you can use constrained output pipelining to place registers at specific locations along the critical path.

Requirements for Constrained Output Pipelining

Your design must contain existing delays or registers. When there are fewer registers than HDL Coder needs to satisfy your constraint, the coder reports the difference between the number of desired and actual output registers.

You can add registers to your design using input or output pipelining.

Specify Constrained Output Pipelining

To specify constrained output pipelining for a block using the UI:

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select the Subsystem and then click **HDL Block Properties**. For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.
- Right-click the block and select **HDL Code > HDL Block Properties**. For **ConstrainedOutputPipeline**, enter the number of registers you want at the output ports.

To specify constrained output pipelining, on the command line, enter:

```
hdlset_param(path_to_block,...  
    'ConstrainedOutputPipeline',number_of_output_registers)
```

For example, to constrain six registers at the output ports of a subsystem, *subsys*, in your model, *mymodel*, enter:

```
hdlset_param('mymodel/subsys','ConstrainedOutputPipeline', 6)
```

Limitations of Constrained Output Pipelining

HDL Coder does not constrain output pipeline register placement:

- Within a DUT subsystem, if the DUT contains a subsystem, model reference, or model reference with black box implementation.
- At the outputs of any type of delay block or the top-level DUT subsystem.

Clock-Rate Pipelining

In this section...

- “Rationale for Clock-Rate Pipelining” on page 24-114
- “How Clock-Rate Pipelining Works” on page 24-114
- “Clock-Rate Pipelining and Hierarchy Flattening” on page 24-115
- “Clock-Rate Pipelining for DUT Output Ports” on page 24-115
- “Best Practices for Clock-Rate Pipelining” on page 24-116
- “Specify Clock-Rate Pipelining” on page 24-116
- “Limitations for Clock-Rate Pipelining” on page 24-116

When you enable speed and area optimizations that insert pipeline registers, use the clock-rate pipelining optimization to identify multicycle paths in your design. Clock-rate pipelining inserts pipeline registers at the faster clock rate, which improves clock frequency without introducing additional latency or by adding minimal latency.

Rationale for Clock-Rate Pipelining

The code generator introduces pipelines when you specify certain block implementations or enable some optimizations on the model, such as:

- Multi-cycle block implementations
- Input and output pipelining
- Distributed pipelining
- Floating-point library mapping
- Native floating-point HDL code generation
- Resource sharing
- Streaming

By default, in slow paths, these pipeline registers operate at the slow data rate. When you enable clock-rate pipelining, the pipeline registers operate at the faster clock rate. Clock-rate pipelining does not affect existing design delays in your model. It is an alternative to using multicycle path constraints with your synthesis tool.

How Clock-Rate Pipelining Works

The clock-rate pipelining optimization identifies slow paths or regions in the model by analyzing the block sample times. Blocks that have a sample time greater than the DUT base sample time are part of the slow path, and are potential candidates for clock-rate pipelining. In these slow paths, the code generator enables optimizations to introduce pipeline delays at the clock rate.

If you specify an “Oversampling factor” on page 17-15 greater than one, the DUT sample time becomes slower than the actual clock rate. The code generator determines the maximum number of clock-rate pipelines that it can insert based on the DUT-to-block sample time ratio and the oversampling factor.

$$\text{Maximum number of clock-rate delays} = (\text{block_rate} \div \text{DUT_base_rate}) \times \text{Oversampling}$$

Clock-rate pipelining identifies regions in the model that have the same slow data rate, and are delimited by either Delay blocks or blocks that introduce a rate transition. The code generator converts these regions to the faster clock rate by introducing Repeat blocks at the input of the region and Rate Transition blocks at the output of the region. If the output of a clock-rate region is a Delay block at the data rate, HDL Coder absorbs that Delay block. To accommodate the delay, the code generator introduces several clock-rate pipelines corresponding to the ratio of the data rate to the clock rate.

HDL Coder generates a script that highlights blocks in your model that are obstacles to clock-rate pipelining and a script to clear the highlighting. Sometimes, if the code generator is unable to implement resource sharing or streaming at the clock rate, it displays a code generation error with a recommendation for changing the **Oversampling** value. To clear highlighting, click the `clearhighlighting` script in the MATLAB Command Window.

Clock-Rate Pipelining and Hierarchy Flattening

You can use the clock-rate pipelining optimization with or without flattening the subsystem hierarchy. Flatten the subsystem hierarchy when you want to maximize opportunities for sharing resources in your design. To flatten the subsystem hierarchy, enable **FlattenHierarchy** on the top-level Subsystem. By default, all Subsystem blocks inside the top-level subsystem inherit this **FlattenHierarchy** setting. Hierarchy flattening brings several clock-rate regions to the same level in the hierarchy and combines them, which increases opportunities for clock-rate pipelining. However, it breaks the modularity of your design and affects the readability of the generated HDL code. See also “Hierarchy Flattening” on page 24-92.

To apply clock-rate pipelining without flattening the hierarchy, on the top-level subsystem in your model, disable **FlattenHierarchy**. If your design uses fixed-point data types, enable some optimizations on the underlying subsystems. In this case, the code generator introduces clock-rate pipelines in your design while preserving the subsystem hierarchy, which:

- Improves the modularity of your design and makes navigation through the generated model easier especially in large designs with complex hierarchies.
- Improves readability of the generated HDL code by creating multiple Verilog or VHDL files for the various Subsystem blocks in your design.

Clock-Rate Pipelining for DUT Output Ports

To insert DUT output port pipeline registers at the clock rate instead of the data rate, select the **Allow clock-rate pipelining of DUT output ports** option or use the `ClockRatePipelineOutputPorts` property. This option changes the timing of your DUT interface because it changes the sample time of your DUT output ports from a slow rate to the clock rate. To adjust for the difference in timing, HDL Coder generates messages that provides the phase offset of each output port. For example, this message means that the output data from `portname` is valid after 31 clock cycles: `Phase of output port portname: 31 clock cycles`.

The validation model adjusts for the timing difference by inserting a Rate Transition block at the DUT output and comparing the output of the Rate Transition with the original output. The RTL test bench logs the output data at the input of the Rate Transition and compares it with the DUT output in the RTL simulation.

Best Practices for Clock-Rate Pipelining

- If your design uses a Rate Transition block, switch the Rate Transition block with a Downsample block that has nonzero **Sample offset**. Clock-rate pipelining optimizes the Downsample block by avoiding the additional latency that the Rate Transition block can introduce, which saves area and timing.
- Design your DUT at one rate and specify the **Oversampling factor**. Avoid using Rate Transition, Upsample, Downsample, or other rate-changing blocks.

Specify Clock-Rate Pipelining

You can set clock-rate pipelining on a model or, for finer control, on subsystems within the top-level DUT subsystem. By default, clock-rate pipelining is enabled on the model. To disable clock-rate pipelining from the UI:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Click **Settings**. In the **HDL Code Generation > Optimization > Pipelining** tab, clear **Clock-rate pipelining** and click **OK**.

At the command line, use the `makehdl` or `hdlset_param` function to set the `ClockRatePipelining` property to `off`.

You can use clock-rate pipelining for a subsystem within the top-level DUT subsystem. To model a control path in your design at the data rate instead of the clock rate, put the control path in a subsystem, and disable clock-rate pipelining for that subsystem. To disable clock-rate pipelining for a subsystem within the top-level DUT subsystem, set **ClockRatePipelining** to `off` for that subsystem. See also “Set Clock-Rate Pipelining For a Subsystem” on page 22-6.

Limitations for Clock-Rate Pipelining

These blocks inhibit clock-rate pipelining, and therefore delimit clock-rate pipelining regions:

- Counter Free-Running
- Counter Limited
- Deserializer1D
- Discrete PID Controller
- Dual Port RAM
- Dual Rate Dual Port RAM
- FFT HDL Optimized
- HDL Cosimulation
- HDL FIFO
- HDL Counter
- Hit Crossing
- HDL Minimum Resource FFT
- HDL Streaming FFT
- MATLAB System, if it uses persistent variables

- Rate Transition
- Serializer1D
- Simple Dual Port RAM
- Single Port RAM
- Subsystem, if `FlattenHierarchy` is not enabled

The code generator does not support clock-rate pipelining for:

- Black box subsystem or black box model reference blocks.
- Subsystems that contain blocks not supported for clock-rate pipelining.
- Altera DSP Builder subsystems.
- Xilinx System Generator subsystems
- Communications Toolbox blocks.
- DSP System Toolbox blocks, except for Delay and Discrete FIR Filter.
- Stateflow blocks.

The code generator does not support applying both the streaming and sharing optimizations on the same resource when you use the clock-rate pipelining optimization. Either disable clock-rate pipelining or use either the streaming optimization or the sharing optimization on the same resource when clock-rate pipelining is enabled.

See Also

Simulink Configuration Parameters

[“Pipelining Parameters” on page 15-9](#) | [“Diagnostic Parameters for Optimizations” on page 17-89](#)

Related Examples

- [“Clock Rate Pipelining” on page 24-118](#)

More About

- [“ClockRatePipelining” on page 22-5](#)
- [“Delay Balancing” on page 24-63](#)
- [“Hierarchy Flattening” on page 24-92](#)

Clock Rate Pipelining

This example shows how to apply clock rate pipelining to optimize slow paths in your design and thereby reduce latency, increase clock frequency and decrease area usage. For more information on how to use clock-rate pipelining, see “Clock-Rate Pipelining” on page 24-114.

Introduction

Algorithmic design with Simulink may introduce many slow-rate datapaths in the generated HDL design. These slow paths correspond to slower Simulink sample time operations or even due to the algorithmic data-rate operating at a slower rate than the HDL clock rate.

Clock-rate pipelining identifies the maximal subregions in the model operating at the same data rate, and are delimited either by rate-change blocks or delay blocks. These subregions are called clock-rate regions because they make good candidates for clock-rate pipelining. If the output of a clock-rate region is a Delay block at the data rate, then HDL Coder absorbs that Delay block. This allows a budget of several clock-rate pipelines corresponding to the ratio of data rate to clock rate.

Consider the Field-Oriented control example “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 10-55. It describes a motor-control design to be mapped to an FPGA. The input samples in this design are arriving every $20 \mu s$ or 50 KHz. In a closed control loop, it is essential that the controller's latency is within the desired response time. In this model, there is a delay on the output port resulting in a latency of $20 \mu s$.

To meet design constraints like timing and area, we may want to apply several optimizations like input/output pipelining, distributed pipelining, streaming and/or sharing. Further, non-trivial math functions like sqrt or divide may have to be implemented as multi-cycle pipelined operations. Pipelines introduced by any of the above features and optimizations are applied at the same rate at which the signal path operates, which is $20 \mu s$. Thus, introducing any additional pipelining introduces undesirable latency overhead and may violate the closed loop latency budget.

However, the FPGA can implement this controller in the order of MHz, which means that the introduced pipelines can then operate at the MHz rate thereby minimizing the impact on latency. Clock-rate pipelining is a technique to leverage this rate differential, pipeline the controller and thereby improve its area and timing characteristics on the FPGA. This example walks through the steps for taking this design and incrementally applying timing and area optimizations using clock-rate pipelining.

Preparing the model

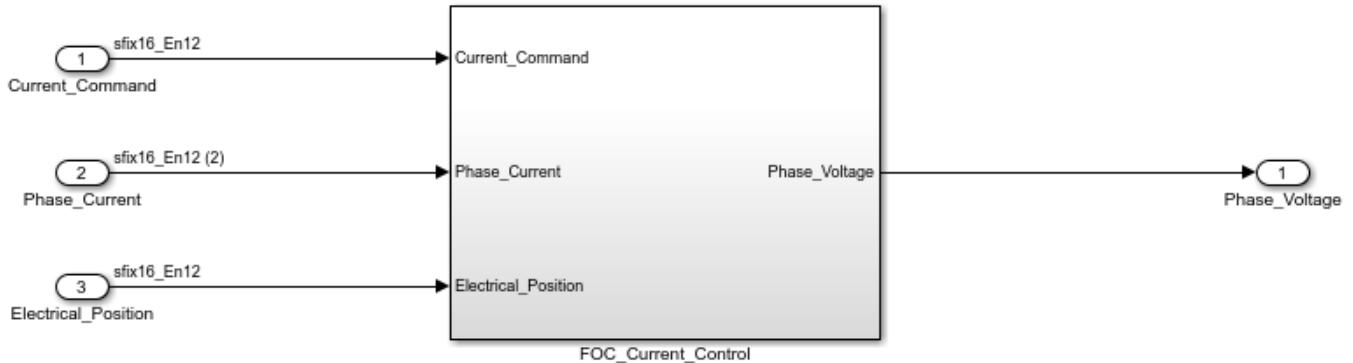
An important first step in applying clock-rate pipelining is to prepare the model so that it is amenable to clock-rate pipelining. Below are some of the main steps:

- Defining the rate differential: Signal paths in Simulink end up on slow paths in HDL because of two primary reasons. First, the signal path is operating at a sample time that is slower than the base sample time of the model. Second, the Simulink base sample time may correspond to the data-rate instead of the clock-rate. For example, the base sample time in the `hdlcoderFocCurrentFixptHdl` model is 20μ secs. The final FPGA implementation of the controller may target 40 MHz (or 25 ns).

```
open_system('hdlcoderFocCurrentFixptHdl')
```

FOC Current Control

Copyright 2014-2017 The MathWorks, Inc.



The trouble with setting the model's sample time to 25 ns is that it drastically slows down Simulink simulation performance. To get around this, HDL Coder provides a setting, called "Oversampling factor" on page 17-15 which specifies how much faster the FPGA clock rate runs with respect to the Simulink base sample time. Thus, in this case, we require a 800x oversampling.

- Set optimizations on subsystems: For fixed-point designs, clock-rate pipelining is applied on a Subsystem only when the coder needs to insert pipelining. HDL Coder options that result in introduction of pipelining are distributed pipelining, sharing, streaming, input/output pipelining, constrained output pipeline, adaptive pipelining and any block implementations that introduce multi-cycle implementations including floating point implementations (for more details, refer to the documentation on individual blocks to understand the impact of their HDL properties on latency. Optimizations can be applied either locally, by maintaining the subsystem hierarchies or globally if the underlying subsystems are all flattened. In the former case, apply pipelining and optimization settings on individual subsystems and in the global case, these settings should be on the top-level subsystem. See "Hierarchy Flattening" on page 24-92 for more information on prerequisites for Hierarchy Flattening.

Applying Clock-rate Pipelining

Now, we are ready to apply clock-rate pipelining. The feature option is on by default and will automatically find clock-rate regions. See "Clock-Rate Pipelining" on page 24-114 to understand how the pipeline budget is determined and how clock-rate regions are formed.

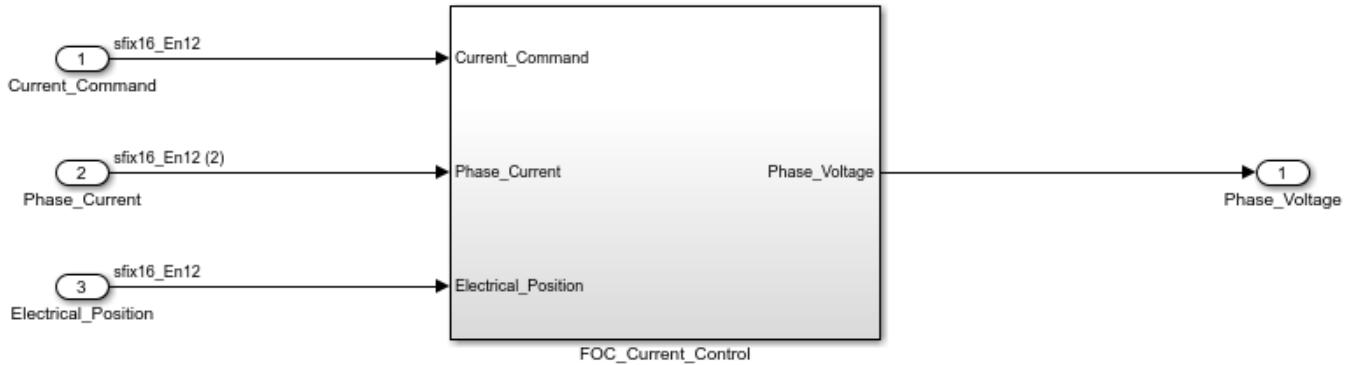
```

srcHdlModel = 'hdlcoderFocCurrentFixptHdl';
dstHdlModel = 'hdlcoderFocClockRatePipelining';
dstHdlDut  = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel = ['gm_' dstHdlModel];
gmHdlDut  = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);
  
```

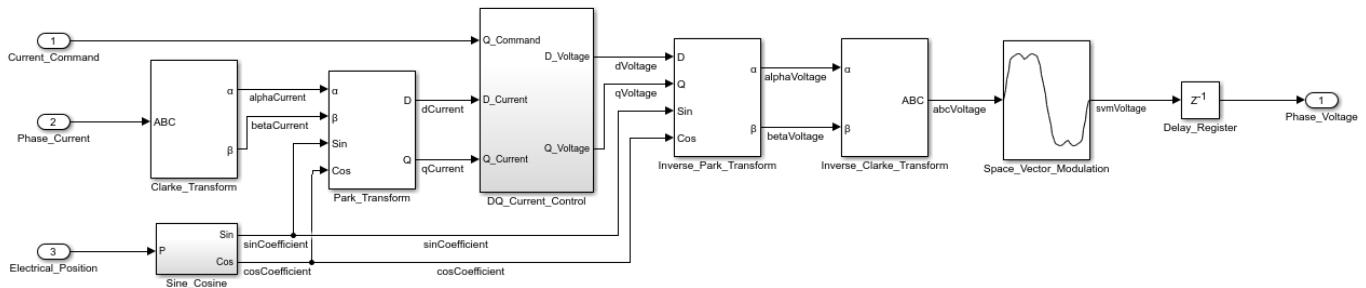
FOC Current Control

Copyright 2014-2017 The MathWorks, Inc.



The subsystem FOC_Current_Control contains the algorithm from which we will generate HDL code

```
open_system(dstHdldut);
```



We can now configure the model to use clock-rate pipelining.

```

hdlset_param(dstHdldut, 'ClockRatePipelining', 'on');
hdlset_param(dstHdldut, 'Oversampling', 800);
hdlset_param(dstHdldut, 'DistributedPipelining', 'on');
set_param([dstHdldut '/DQ_Current_Control/D_Current_Control'], 'TreatAsAtomicUnit', 'off');
set_param([dstHdldut '/DQ_Current_Control/Q_Current_Control'], 'TreatAsAtomicUnit', 'off');

hdlset_param([dstHdldut '/DQ_Current_Control/D_Current_Control'], 'DistributedPipelining', 'on')
hdlset_param([dstHdldut '/DQ_Current_Control/Q_Current_Control'], 'DistributedPipelining', 'on')

hdlset_param([dstHdldut '/Clarke_Transform'], 'DistributedPipelining', 'on');
hdlset_param([dstHdldut '/Park_Transform'], 'DistributedPipelining', 'on');
hdlset_param([dstHdldut '/Sine_Cosine'], 'DistributedPipelining', 'on');
hdlset_param([dstHdldut '/Inverse_Park_Transform'], 'DistributedPipelining', 'on');
hdlset_param([dstHdldut '/Inverse_Clark_Transform'], 'DistributedPipelining', 'on');
hdlset_param([dstHdldut '/Space_Vector_Modulation'], 'DistributedPipelining', 'on');

save_system(dstHdldut);

```

To see the impact of clock-rate pipelining, generate HDL code and look inside the top-level subsystem of the generated model.

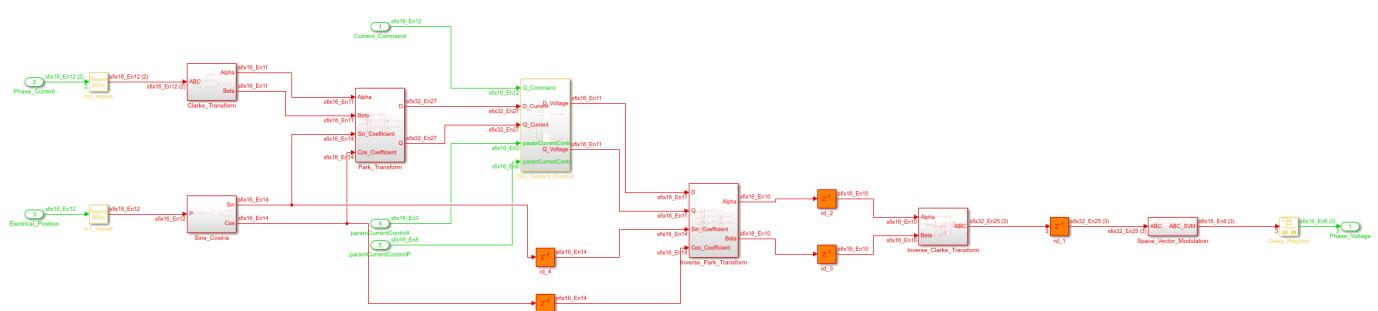
We can review the generated model and observe that the entire design has been Clock-rate pipelined and is running at the fast rate. If there are subsystems in the generated model which are not clock rate pipelined then check (as mentioned above) if there were optimizations set on the subsystem in the original model.

```

open_system(gmHdLut);
set_param(gmHdLModel, 'SimulationCommand', 'update');
set_param(gmHdLut, 'ZoomFactor', 'FitSystem');

% Further, rate-transitions are introduced on the design inputs to bring them
% to the clock-rate, which is determined as the original base sample time divided by the Oversamp
% which is  $2e-5/800 = 2.5e-8$  or 25 ns. All pipelines are introduced at this rate and are thus ope
% Finally, observe that the output-side delay has been replaced by a down-sampling
% rate transition bringing the signal back to the data-rate. The clock frequency of the design w

```



As with all optimizations, it is recommended that the validation model and co-simulation model are generated and the user verifies that the functional behavior of the design is unchanged. The “Verification” describe these concepts in more depth.

Local optimization with subsystem options

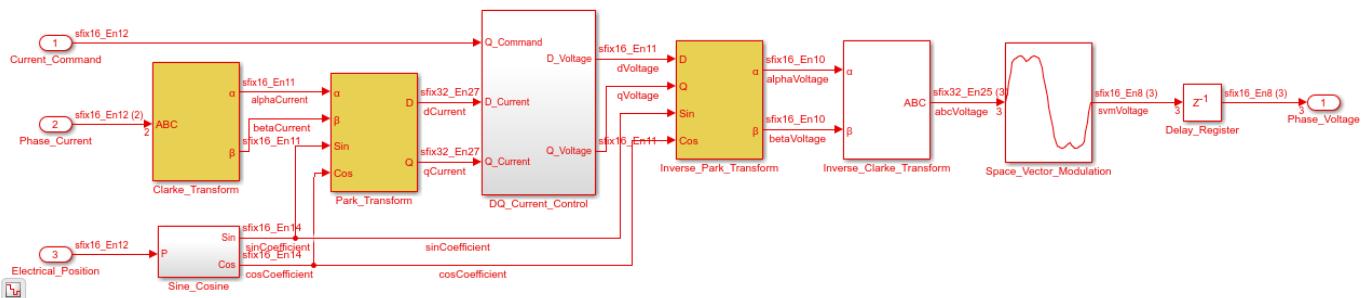
The rate differential on slow path implies that computation along this path can take several clock cycles. Specifically, the allowed latency is defined by the clock-rate budget (see “Clock-Rate Pipelining” on page 24-114). Apart from adding pipelines to improve clock frequency, we could reuse hardware resources by leveraging the latency budget. Setting resource sharing options like StreamingFactor and SharingFactor in a slow-path region does exactly that. This section demonstrates how resource sharing is applied within clock-rate regions.

When resource sharing is applied to a clock-rate path, HDL Coder oversamples the shared resource architecture for time-multiplexing as illustrated in the “Resource Sharing For Area Optimization” on page 24-40. However, if sharing or streaming is requested in a slow datapath, then HDL Coder implements resource sharing without oversampling. To trigger such sharing, set either sharing or streaming on the subsystem on which you want to apply resource sharing or streaming.

```
srcHdlModel = 'hdlcoderFocClockRatePipelining';
dstHdlModel = 'hdlcoderFocSharing';
dstHdlDut   = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel  = ['gm_' dstHdlModel];
gmHdlDut   = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

open_system(dstHdlDut);
hilite_system([dstHdlDut '/Park_Transform']);
hilite_system([dstHdlDut '/Inverse_Park_Transform']);
hilite_system([dstHdlDut '/Clarke_Transform']);
```



The Park_Transform subsystem and the Inverse_Park_Transform subsystem each use 4 multipliers within them that can be potentially shared. Additionally, the Clarke Transform subsystem and the Inverse_Clarke_Transform subsystem each use 2 gains, which may be potentially shared, unless they are simply power-of-2 gains, which results in shifts instead of multiplications. Hence, the gain in Inverse_Clarke_Transform cannot be shared. Now, we can set the appropriate sharing factors on each of the subsystem on which we want to apply resource sharing.

```
hdlset_param([dstHdlDut '/Park_Transform'], 'SharingFactor', 4);
hdlset_param([dstHdlDut '/Inverse_Park_Transform'], 'SharingFactor', 4);
hdlset_param([dstHdlDut '/Clarke_Transform'], 'SharingFactor', 2);

save_system(dstHdlModel);
```

```

makehdl(dstHdlDut);

### Generating HDL for 'hdlcoderFocSharing/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocSharing')">.
### Starting HDL check.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocSharing\hdlcoderFocSharing_HDLCheck.m')">.
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocSharing\hdlcoderFocSharing_HDLHighlightClear.m')">.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocSharing_vnl')">.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocSharing'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Saturation
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/D_Current_Control as hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/Q_Current_Control/Saturation
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control/Q_Current_Control as hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/DQ_Current_Control as hdlsrc\hdlcoderFocSharing
### Working on Clarke_Transform_shared as hdlsrc\hdlcoderFocSharing\Clarke_Transform_shared.vhd.
### Working on hdlcoderFocSharing/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdlsrc\hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocSharing\Sine_Cosine
### Working on Park_Transform_shared as hdlsrc\hdlcoderFocSharing\Park_Transform_shared.vhd.
### Working on hdlcoderFocSharing/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderFocSharing
### Working on Inverse_Park_Transform_shared as hdlsrc\hdlcoderFocSharing\Inverse_Park_Transform
### Working on hdlcoderFocSharing/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc\hdlcoderFocSharing
### Working on hdlcoderFocSharing/FOC_Current_Control/Space_Vector_Modulation as hdlsrc\hdlcoderFocSharing
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocSharing\FOC_Current_Control_tc.vhd.
### Working on hdlcoderFocSharing/FOC_Current_Control as hdlsrc\hdlcoderFocSharing\FOC_Current_Control
### Generating package file hdlsrc\hdlcoderFocSharing\FOC_Current_Control_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl')">.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'hdlcoderFocSharing' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.

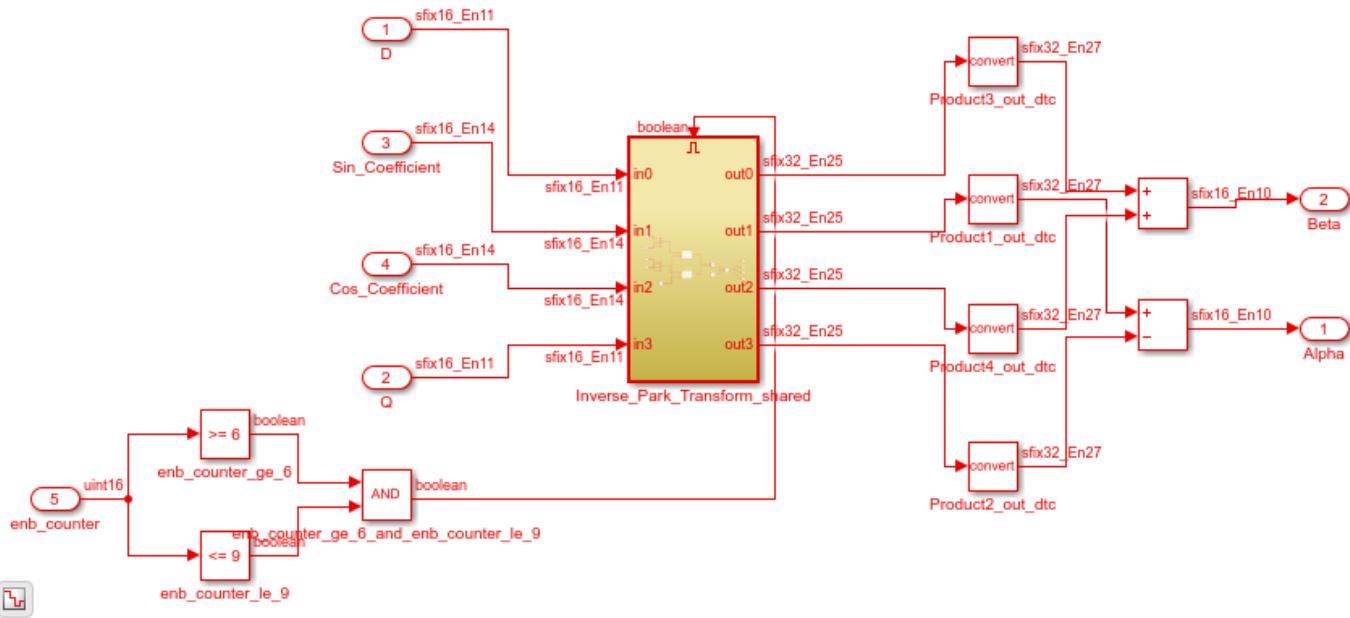
```

We can review the generated model and observe that HDL Coder implements time-multiplexing in the clock-rate using knowledge of the available latency budget due to the slow datapath.

```

open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
hilite_system([gmHdlDut '/ctr_799']);
hilite_system([gmHdlDut '/ctr_7991']);
hilite_system([gmHdlDut '/Clarke_Transform/Clarke_Transform_shared']);
hilite_system([gmHdlDut '/Park_Transform/Park_Transform_shared']);
hilite_system([gmHdlDut '/Inverse_Park_Transform/Inverse_Park_Transform_shared']);

```



The time-multiplexing architecture, also known as the single-rate sharing architecture is described in “Single-rate Resource Sharing Architecture” on page 24-50. A global scheduler is created to enable and disable different regions of the design using enabled subsystems. The enable/disable control is implemented using a limited counter `ctr_799` and `ctr_7991` that counts to the latency budget (0 to 799). The shared regions are implemented as enabled subsystems that are enabled according to an automatically determined schedule order. In this design, we found 2 groups of multipliers that was shared by 4-ways and 1 group of multipliers that was shared by 2-ways. The multiplier count for the design has reduced from 20 to 13 without any latency penalties.

Global optimization with flattening

Global cross-subsystem optimizations can be applied by leveraging the subsystem-flattening feature. With flattening, there are more number of resources that can be shared at the same level of hierarchy. To trigger such sharing, set either sharing or streaming on the top-level subsystem. The sharing factor value chosen must be an upper bound. To determine a good value, the resource usage of the design must be analyzed.

```

srcHdlModel = 'hdlcoderFocCurrentFixptHdl';
dstHdlModel = 'hdlcoderFocSharingWithFlattening';
dstHdlDut  = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel = ['gm_' dstHdlModel];
gmHdlDut  = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

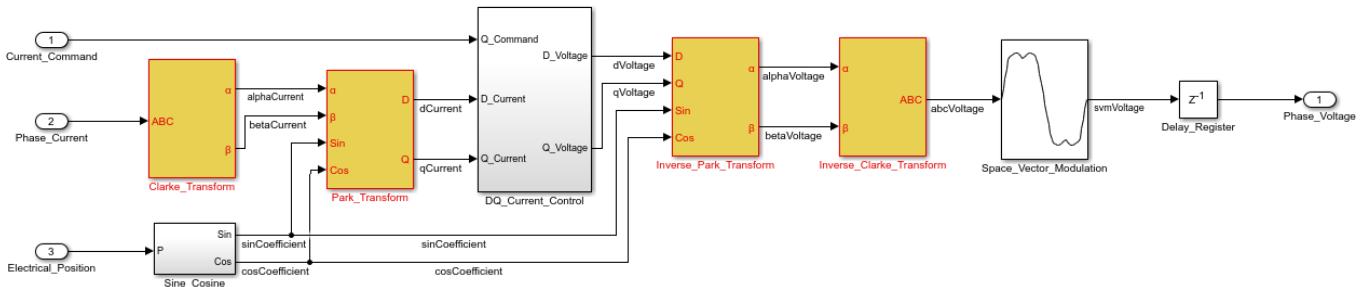
open_system(dstHdlDut);

hdlset_param(dstHdlModel, 'ClockRatePipelining', 'on');
hdlset_param(dstHdlModel, 'Oversampling', 800);
hdlset_param(dstHdlDut, 'FlattenHierarchy', 'on');
hdlset_param(dstHdlDut, 'DistributedPipelining', 'on');

hilite_system([dstHdlDut '/Park_Transform']);

```

```
hilite_system([dstHdlDut '/Inverse_Park_Transform']);
hilite_system([dstHdlDut '/Clarke_Transform']);
hilite_system([dstHdlDut '/Inverse_Clarke_Transform']);
```



The Park_Transform subsystem and the Inverse_Park_Transform subsystem each use 4 multipliers within them that can be potentially shared. Additionally, the Clarke_Transform subsystem and the Inverse_Clarke_Transform subsystem each use 2 gains, which may be potentially shared, unless they are simply power-of-2 gains, which results in shifts instead of multiplications. Therefore, we can choose the upper-bound value of 4 for SharingFactor and generate code.

```
hdlset_param(dstHdlDut, 'SharingFactor', 4);
save_system(dstHdlModel);

makehdl(dstHdlDut);

### Generating HDL for 'hdlcoderFocSharingWithFlattening/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocSharingWithFlattening/FOC_Current_Control')">.
### Starting HDL check.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocSharingWithFlattening\hdlcoderFocSharingWithFlattening\hdpipelineHighlighting.m')">.
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocSharingWithFlattening\hdlcoderFocSharingWithFlattening\hdpipelineClearHighlighting.m')">.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocSharingWithFlattening/FOC_Current_Control')">.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocSharingWithFlattening'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Working on crp_temp_shared as hdlsrc\hdlcoderFocSharingWithFlattening\crp_temp_shared.vhd.
### Working on crp_temp_shared_block as hdlsrc\hdlcoderFocSharingWithFlattening\crp_temp_shared_block.vhd.
### Working on crp_temp_shared_block1 as hdlsrc\hdlcoderFocSharingWithFlattening\crp_temp_shared_block1.vhd.
### Working on crp_temp_shared_block2 as hdlsrc\hdlcoderFocSharingWithFlattening\crp_temp_shared_block2.vhd.
### Working on crp_temp_shared_block3 as hdlsrc\hdlcoderFocSharingWithFlattening\crp_temp_shared_block3.vhd.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocSharingWithFlattening\FOC_Current_Control_tc.vhd.
### Working on hdlcoderFocSharingWithFlattening/FOC_Current_Control as hdlsrc\hdlcoderFocSharingWithFlattening\FOC_Current_Control.vhd.
### Generating package file hdlsrc\hdlcoderFocSharingWithFlattening\FOC_Current_Control_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl')">.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl.
### HDL check for 'hdlcoderFocSharingWithFlattening' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.
```

We can review the generated model and observe that HDL Coder implements time-multiplexing in the clock-rate using knowledge of the available latency budget due to the slow datapath.

```
open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
hilite_system([gmHdlDut '/ctr_799']);
hilite_system([gmHdlDut '/crp_temp_shared']);
hilite_system([gmHdlDut '/crp_temp_shared1']);
```

```
hilite_system([gmHdlDut '/crp_temp_shared2']);
hilite_system([gmHdlDut '/crp_temp_shared3']);
hilite_system([gmHdlDut '/crp_temp_shared4']);
```



In this design, we found five groups of multipliers that were shared by 4-ways or less. These 5 subsystems have crp_temp_shared as part of their names.

In summary, the multiplier count for the design has reduced from 20 to 7 without any latency penalties as opposed to 13 when the design was not flattened.

Minimizing latency

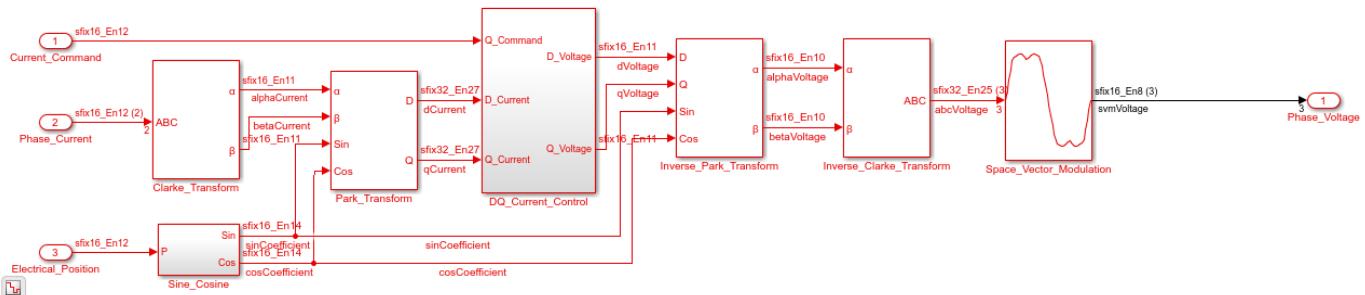
As an advanced maneuver, it is possible to reduce the output latency by removing the output Delay_Register and instead using the option to allow clock-rate pipelining of DUT output ports.

```
srcHdlModel = 'hdlcoderFocSharing';
dstHdlModel = 'hdlcoderFocMinLatency';
dstHdlDut = [dstHdlModel '/FOC_Current_Control'];
gmHdlModel = ['gm_' dstHdlModel];
gmHdlDut = ['gm_' dstHdlDut];

open_system(srcHdlModel);
save_system(srcHdlModel,dstHdlModel);

delete_line(dstHdlDut,'Space_Vector_Modulation/1','Delay_Register/1');
delete_line(dstHdlDut,'Delay_Register/1','Phase_Voltage/I');
delete_block([dstHdlDut,'/Delay_Register'])
add_line(dstHdlDut,'Space_Vector_Modulation/1','Phase_Voltage/I');

open_system(dstHdlDut);
```



The clock-rate pipelining for output ports option is available in the configuration parameters dialog under the 'HDL Code Generation' -> 'Optimization' -> 'Pipelining' tab: check the 'Allow clock-rate pipelining of DUT output ports' option. This command-line property name for this option is 'ClockRatePipelineOutputPorts'. When the 'ClockRatePipelineOutputPorts' option is turned on and the output register removed, the generated HDL code does not wait for the full sample step to generate

the output. Rather, it will generate the output within a few clock cycles as soon as the data is ready. The generated HDL code will generate the output at the clock-rate without waiting for the next sample step.

```
hdlset_param(dstHdlModel, 'ClockRatePipelineOutputPorts', 'on');
save_system(dstHdlModel);

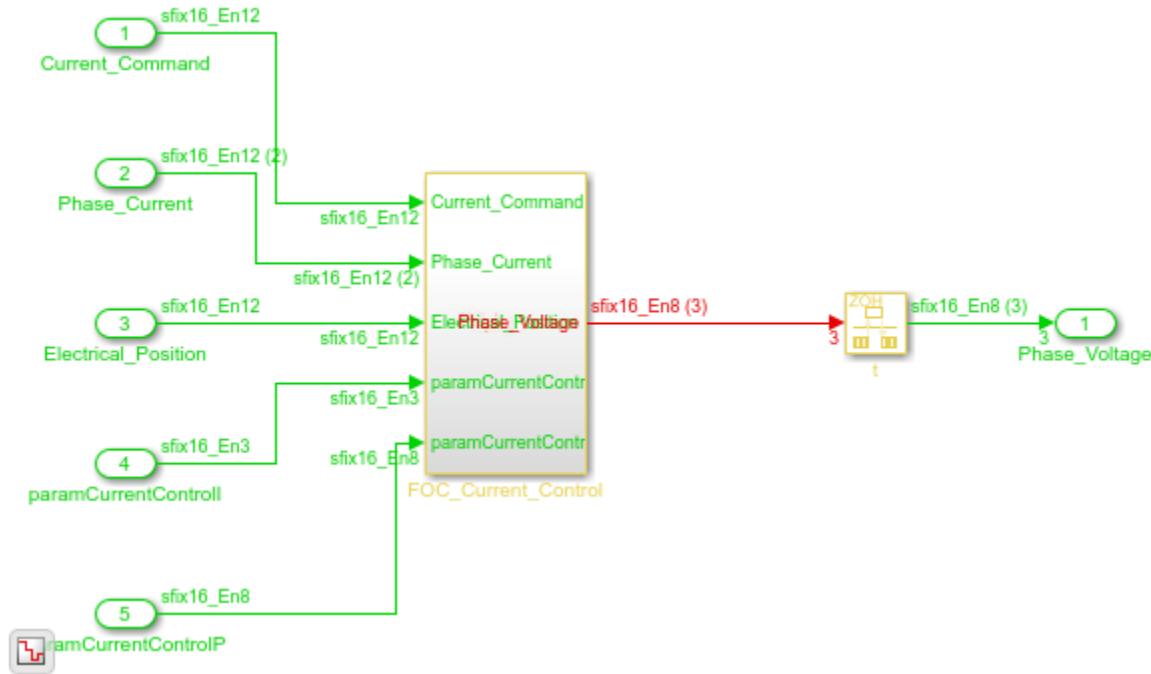
makehdl(dstHdlDut);

### Generating HDL for 'hdlcoderFocMinLatency/FOC_Current_Control'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocMinLatency')">.
### Starting HDL check.
### Clock-rate pipelining was applied on signals connected to the DUT's output ports. The DUT output port 0 has a latency of 11 clock cycles.
### Phase of output port 0: 11 clock cycles.
### To highlight blocks that obstruct distributed pipelining, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocMinLatency\highlightBlocks.m')">.
### To clear highlighting, click the following MATLAB script: <a href="matlab:run('hdlsrc\hdlcoderFocMinLatency\clearHighlighting.m')">.
### Generating new validation model: <a href="matlab:open_system('gm_hdlcoderFocMinLatency_vnl')">.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoderFocMinLatency'.
### MESSAGE: The design requires 800 times faster clock with respect to the base rate = 2e-05.
### Working on Clarke_Transform_shared as hdlsrc\hdlcoderFocMinLatency\Clarke_Transform_shared.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Clarke_Transform as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Clarke_Transform.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/D_Current_Control/Signed as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Space_Vector_Modulation\Space_Vector_Modulation.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/D_Current_Control as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Space_Vector_Modulation\Space_Vector_Modulation.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/Q_Current_Control/Signed as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Space_Vector_Modulation\Space_Vector_Modulation.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/DQ_Current_Control/Q_Current_Control as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Space_Vector_Modulation\Space_Vector_Modulation.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Inverse_Clarke_Transform as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Inverse_Clarke_Transform.vhd.
### Working on Inverse_Park_Transform_shared as hdlsrc\hdlcoderFocMinLatency\Inverse_Park_Transform_shared.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Inverse_Park_Transform as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Inverse_Park_Transform.vhd.
### Working on Park_Transform_shared as hdlsrc\hdlcoderFocMinLatency\Park_Transform_shared.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Park_Transform as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Park_Transform.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Sine_Cosine/Sine_Cosine_LUT as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Sine_Cosine_LUT.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control/Sine_Cosine as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control\Sine_Cosine.vhd.
### Working on hdlcoderFocMinLatency/FOC_Current_Control_Space_Vector_Modulation as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control_Space_Vector_Modulation.vhd.
### Working on FOC_Current_Control_tc as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control_tc.vhd.
### Working on FOC_Current_Control as hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control.vhd.
### Generating package file hdlsrc\hdlcoderFocMinLatency\FOC_Current_Control_pkg.vhd.
### Generating HTML files for code generation report at <a href="matlab:web('C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tphome\hdlcoderFocMinLatency\report.html')">.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tphome\hdlcoderFocMinLatency\report.html.
### HDL check for 'hdlcoderFocMinLatency' complete with 0 errors, 0 warnings, and 2 messages.
### HDL code generation complete.
```

Notice that the 'makehdl' command has generated a message, '### Phase of output port 0:'. This message instructs the user on how to sample the DUT's outputs. The number of clock cycles specified here corresponds to how quickly the DUT's outputs can be sampled and, in essence, this is the latency of the design. Thus, the total latency of the design is down from a data-rate sample step of 20 μ s to a few nanoseconds.

We can review the generated model to observe that a new DUT subsystem is created whose output operates at the clock-rate, which is 25 ns.

```
open_system(gmHdlDut);
set_param(gmHdlModel, 'SimulationCommand', 'update');
set_param(gmHdlDut, 'ZoomFactor', 'FitSystem');
```



We must be careful when using this option since additional latency is introduced into the generated HDL code that was not in the original simulation model. In doing this, the sample-time of the output port has changed to the clock-rate. This introduces a possible discrepancy in results during the validation and verification flow since the test-harness expects the design to generate outputs at the data-rate. The validation model addresses this problem by inserting a down-sampling rate-transition to bring the output back to the data-rate. Thus, the validation model still compares outputs at the data-rate. The HDL testbench will, however, compare the new DUT's outputs at the clock-rate since the generated HDL outputs are emitted at the clock-rate.

Fine-tuning for performance

While this example illustrates the basic workflow to use clock-rate pipelining to minimize design latency, there are many other options available for fine-tuning HDL performance. The following are tips to leverage the feature's full potential. Note that these guidelines may not correspond to good modeling practices, but rather they are good practices for preparing your implementation model for HDL code generation and optimization.

- **Multi-rate designs:** In this example, the source model is operating at a single rate, which is the data-rate. The **Oversampling factor** option specifies its relationship to the clock-rate. This setup works best for minimizing design latency. Clock-rate pipelining also works well in multi-rate designs by optimizing the slow-paths, but may introduce sample delays at the rate-transition boundaries. Thus, for minimizing latency, use a single-rate (the data-rate) for the whole design.
- **Clock frequency:** You will notice in this design that distributed pipelining did not pipeline the whole datapath. This is because the optimization is cognizant of the consequences of retiming across certain blocks that may cause a numerical mismatch; see the distributed pipelining documentation for more details. Often, these numerical integrity issues occur at boundary conditions. If your design does not hit these boundary conditions, you can enable the **Distributed pipelining priority** parameter. In this case, the you must go through validation to confirm that design is working properly and is robust to all operating conditions.

- Flatten hierarchy : You can turn on hierarchy flattening to either maximize global cross-subsystem optimizations and/or to improve the effectiveness of clock rate pipelining in the presence of feedback loops. In particular, when there are feedback loops that cross subsystem boundaries, it is recommended to turn on hierarchy flattening in the highest-level subsystem that contains the feedback loop. However, to be effective, please check that all the requirements for "Hierarchy Flattening" on page 24-92 are satisfied for the lower level subsystems.
- Provide sufficient budget: When the total number of clock-rate pipelines applied is equal to or more than the available oversampling budget, then understanding the timing impact can be hard. Therefore, provide sufficient budget, or value of **Oversampling factor**, for clock-rate pipelining. The only drawback of too big of an oversampling value is that the counters used by the timing controller and scheduler may be larger. The area overhead is, therefore, quite small.

Summary

Clock-rate pipelining is a technique to optimize and pipeline slow paths in your design. Clock-rate pipelining ensures that pipelines are introduced at the clock-rate for the following HDL Coder constructs and features:

- Pipelined math operations: Several math blocks implement a multi-cycle, pipelined HDL implementation, e.g., Newton-Rhapson method for sqrt or recip, Cordic algorithm for trigonometric functions. These pipelines are introduced at clock-rate if the block operates on a slow path.
- Floating point mapping: As described above, floating point library mapping utilizes clock-rate pipelines when implementing floating point math.
- Pipelining optimizations: All pipelining optimizations including input/output pipelining, adaptive pipelining and distributed pipelining use clock-rate registers on slow paths.
- Resource sharing and streaming: Time-multiplexing of resource-shared architectures are implemented at the clock-rate.

Slow paths are identified as paths using a slower Simulink sample time or when **Oversampling factor** is set in the HDL Coder settings. Using clock-rate pipelining, the design's speed and area properties can be improved without compromising the design's total latency.

Adaptive Pipelining

In this section...

- “Requirements” on page 24-130
- “Specify Adaptive Pipelining” on page 24-130
- “Supported Blocks” on page 24-131
- “Pipeline Insertion for Lookup Tables” on page 24-131
- “Pipeline Insertion for Rate Transition and Downsample Blocks” on page 24-132
- “Pipeline Insertion for Product and Gain Blocks” on page 24-132
- “Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks” on page 24-133
- “Pipeline Insertion for MATLAB Function Blocks” on page 24-135
- “Adaptive Pipelining Report” on page 24-136

Certain patterns or combination of blocks with registers can improve the achievable clock frequency and reduce the area usage on the FPGA boards. The adaptive pipelining optimization creates these patterns by inserting pipeline registers to the blocks in your design. To determine the optimal number of pipeline registers to insert in your design, the optimization considers the target device, target frequency, multiplier word lengths, and the settings in the HDL Block Properties. Use adaptive pipelining with:

- Clock-rate pipelining to insert pipeline registers at a faster clock-rate instead of the slower data-rate. With clock-rate pipelining, you can design your DUT at one rate, and then specify the **Oversampling factor**.
- Resource sharing which saves area and timing because the code generator shares resources and inserts adaptive pipeline registers.

Requirements

- For HDL Coder to insert adaptive pipelines, specify the target device. When your design has multipliers, specify the target device and the target frequency.

Note If you use a target device that is not characterized for adaptive pipelining, the optimization uses Xilinx Virtex-7 when Xilinx Vivado is specified as the **Synthesis Tool**, and uses Intel Stratix® V when the **Synthesis Tool** is Altera Quartus II or Intel Quartus Pro.

- Make sure that delay balancing is enabled for the subsystem that you want HDL Coder to insert adaptive pipelines for. If you disable delay balancing, the code generator does not insert adaptive pipelines.
- Make sure that your design does not have floating-point data types or operations.

Note In some cases, when you have blocks inside a feedback loop, adaptive pipelining is unable to insert the required number of pipeline registers at the output. Delay balancing can then fail.

Specify Adaptive Pipelining

You can set adaptive pipelining for an entire model or, for finer control, you can set adaptive pipelining for subsystems within the top-level DUT subsystem.

Disable Adaptive Pipelining for a Model

By default, adaptive pipelining is enabled at the model level. You can disable adaptive pipelining in one of the following ways:

- In the HDL Workflow Advisor, on the **HDL Code Generation > Set Code Generation Options > Set Optimization Options > Pipelining** tab, select **Adaptive pipelining**.
- In the Configuration Parameters dialog box, on the **HDL Code Generation > Optimization > Pipelining** tab, select **Adaptive pipelining** and click **OK**.
- At the command line, use the `makehdl` or `hdlset_param` function to set “Adaptive pipelining” on page 15-13 to off.

```
hdlset_param(gcs, 'AdaptivePipelining', 'off')
```

Enable Adaptive Pipelining for a Subsystem

By default, subsystems in your model inherit the model-level adaptive pipelining setting. If you want HDL Coder to selectively disable adaptive pipelines for a subsystem in your model, set **AdaptivePipelining** to off for that subsystem.

To learn how to set adaptive pipelining for a subsystem, see “Set Adaptive Pipelining For a Subsystem” on page 22-4.

Supported Blocks

Adaptive pipelining supports these lookup tables, multipliers, multiply accumulate, and rate transition blocks for automatic pipeline insertion.

- n-D Lookup Table
- Direct Lookup Table (n-D)
- Sine HDL Optimized
- Cosine HDL Optimized
- Downsample
- Rate Transition
- Product
- Gain
- Multiply-Add
- Multiply-Accumulate
- MATLAB Function

Pipeline Insertion for Lookup Tables

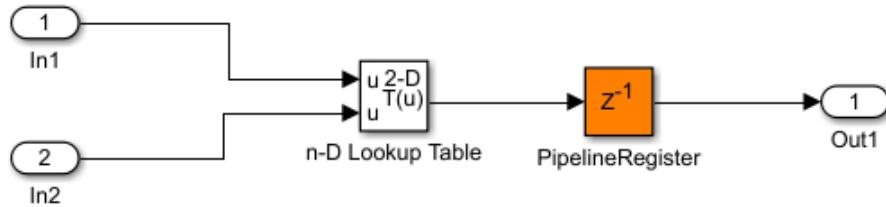
Lookup tables are blocks in the **HDL Coder > Lookup Tables** library including the Sine HDL Optimized and Cosine HDL Optimized blocks.

To insert adaptive pipelines for lookup table blocks:

- 1 Specify the target device.
- 2 Make sure that **Interpolation method** is set to Flat or Linear.

When generating code, HDL Coder inserts a register without reset at the output of the lookup table. The combination of lookup table block and register without reset can potentially map to RAM blocks on the FPGA.

This figure is the generated model for an n-D Lookup Table block with Xilinx Virtex7 as the target FPGA device.



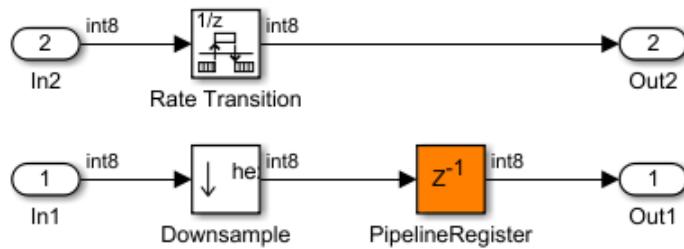
Pipeline Insertion for Rate Transition and Downsample Blocks

To insert adaptive pipelines for the Rate Transition and Downsample blocks:

- 1 Specify the target device.
- 2 Make sure that Downsample blocks have a **Downsample factor** greater than two.

When generating code, HDL Coder inserts a pipeline register at the output port of the Downsample block. Addition of the pipeline register can avoid the bypass register logic, which saves area on the target FPGA.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device.



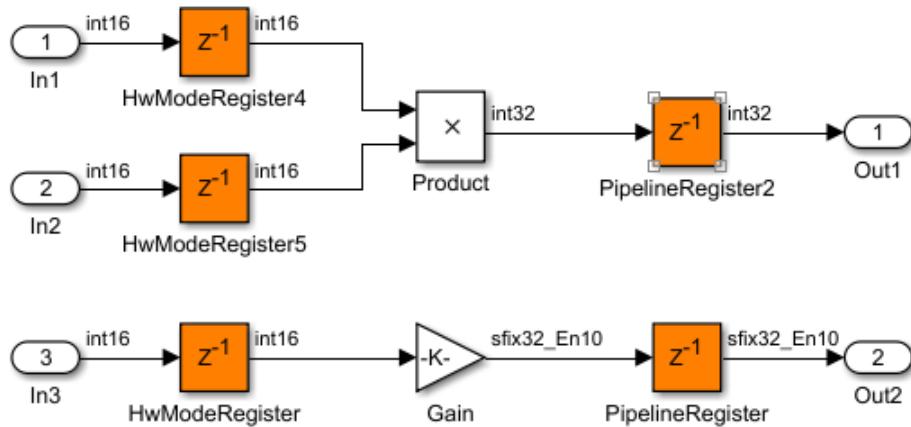
Pipeline Insertion for Product and Gain Blocks

To insert adaptive pipelines for these blocks:

- 1 Specify the target device.
- 2 Specify a target frequency greater than zero.

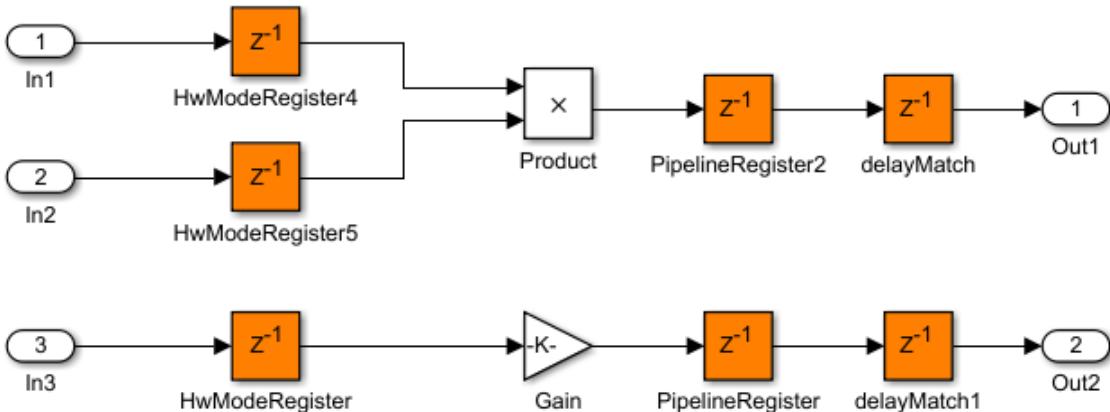
When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

This figure is the generated model for the Product, Gain, and Multiply-Add blocks with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type int16.



The pattern and number of pipeline registers that HDL Coder inserts can vary depending on the target device, target frequency, and the multiplier word lengths.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device and a target frequency of 1500 MHz. The inputs are of type `int8`.



The blocks have a different number of pipeline registers at the output ports. To match the delays, HDL Coder adds a delay at the output of the Product and Gain blocks.

Pipeline Insertion for Multiply-Add and Multiply-Accumulate Blocks

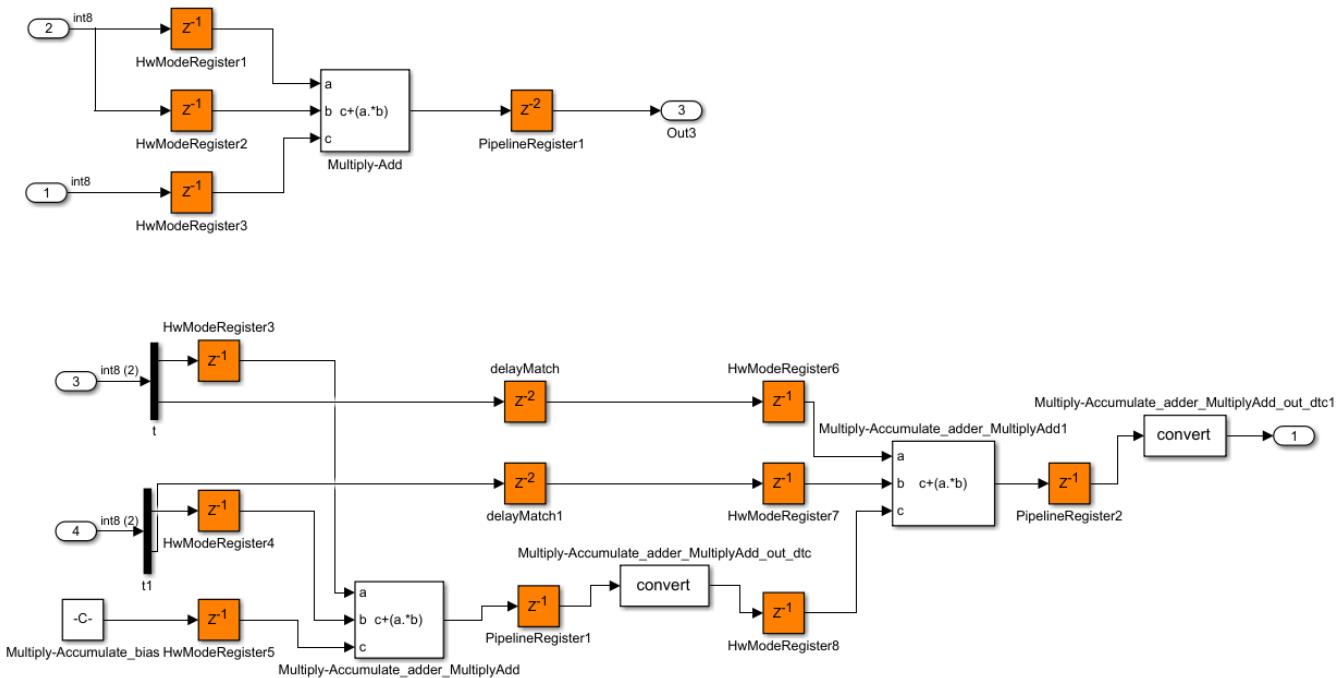
To insert adaptive pipelines for these blocks:

- 1 Specify the target device.
- 2 Specify a target frequency greater than zero.
- 3 Use the Parallel HDL architecture for the Multiply-Accumulate block. For an input vector of size N, this architecture uses N Multiply-Add blocks in series to compute the result.

Caution The Multiply-Add block with **PipelineDepth** set to **auto** or a value greater than zero and the Multiply-Accumulate block with HDL architecture specified as **Parallel** ignore the adaptive pipelining setting. If you specify the target FPGA device and a target frequency greater than zero, the code generator inserts pipeline registers at the inputs and outputs of the block even when adaptive pipelining is disabled.

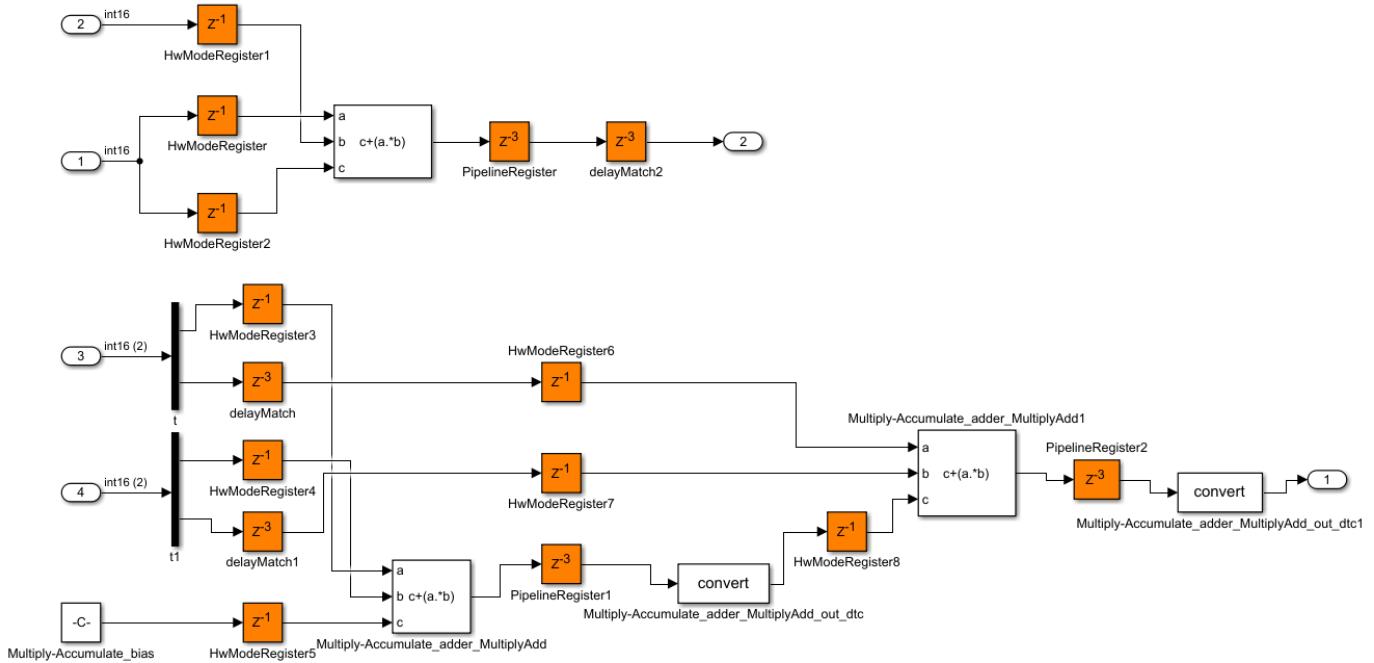
When generating code, HDL Coder inserts registers at the input and output ports of the blocks. The combination of the blocks with the registers can potentially map to DSP units on the target device.

This figure is the generated model for the Multiply-Add and Multiply-Accumulate with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type **int8**.



The pattern and number of pipeline registers that HDL Coder inserts can vary depending on the target device, target frequency, and the multiplier word lengths.

This figure is the generated model for the blocks with Xilinx Virtex7 as the target FPGA device and a target frequency of 1500 MHz. The inputs are of type **int16**.



Pipeline Insertion for MATLAB Function Blocks

To insert adaptive pipelines for MATLAB Function blocks:

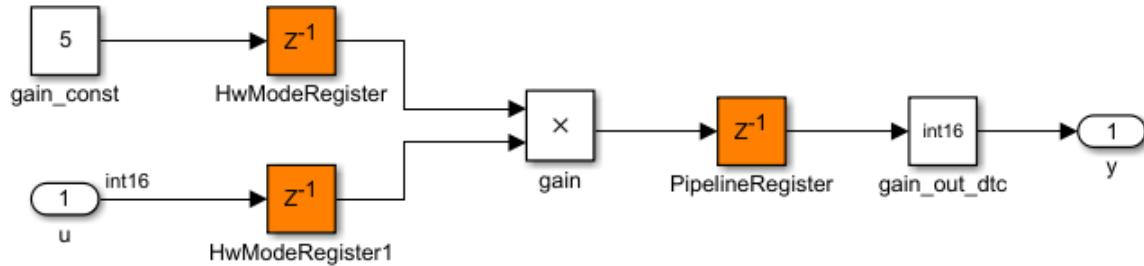
- 1 Specify the target device.
- 2 Specify a target frequency greater than zero.
- 3 Set the HDL architecture for the MATLAB Function blocks to **MATLAB Datapath**.

HDL Coder treats MATLAB Function blocks with architecture set to **MATLAB Datapath** like regular Subsystems. The code generator converts the MATLAB algorithm to a Simulink block diagram. If the Simulink diagram uses blocks that are supported by adaptive pipelining such as Product or Add, the code generator inserts pipeline registers at the input and output ports of the blocks. The combination of multipliers with the registers can potentially map to DSP units on the target device.

Consider a MATLAB Function block that uses the **MATLAB Datapath** architecture. This code is the algorithm inside the MATLAB Function block.

```
function y = fcn(u)
y = u^5;
```

This figure is the generated model for the MATLAB Function block with Intel Arria10 as the target FPGA device and a target frequency of 500 MHz. The inputs to the blocks are of type **int16**. The code generator inferred the algorithm as a multiplication by a constant and inserted adaptive pipelines at the input and output.



See “HDL Applications for the MATLAB Function Block” on page 29-2.

Adaptive Pipelining Report

To see the adaptive pipelining information in the report, before you generate code for each Subsystem or model reference, enable the Code Generation report. To enable the Code Generation report, in the Configuration Parameters dialog box, on the **HDL Code Generation** pane, select **Generate optimization report**.

When you generate code, HDL Coder produces the Code Generation report. Select the **Adaptive Pipelining** section of the Optimization report.

The Adaptive Pipelining report displays the status of the adaptive pipelining optimization and whether HDL Coder inserted adaptive pipelines in your design.

If adaptive pipelining is successful, the report displays the blocks for which HDL Coder inserted pipeline registers, the number of pipeline registers inserted, and any additional notes. Click the link to the block to see the pipeline registers inserted to the blocks in your design.

If adaptive pipelining fails, the report displays the criteria that caused adaptive pipelining to fail.

See Also

More About

- “Balance delays” on page 15-3
- “AdaptivePipelining” on page 22-4
- “Clock-Rate Pipelining” on page 24-114

Critical Path Estimation Without Running Synthesis

In this section...

["How Critical Path Estimation Works" on page 24-137](#)

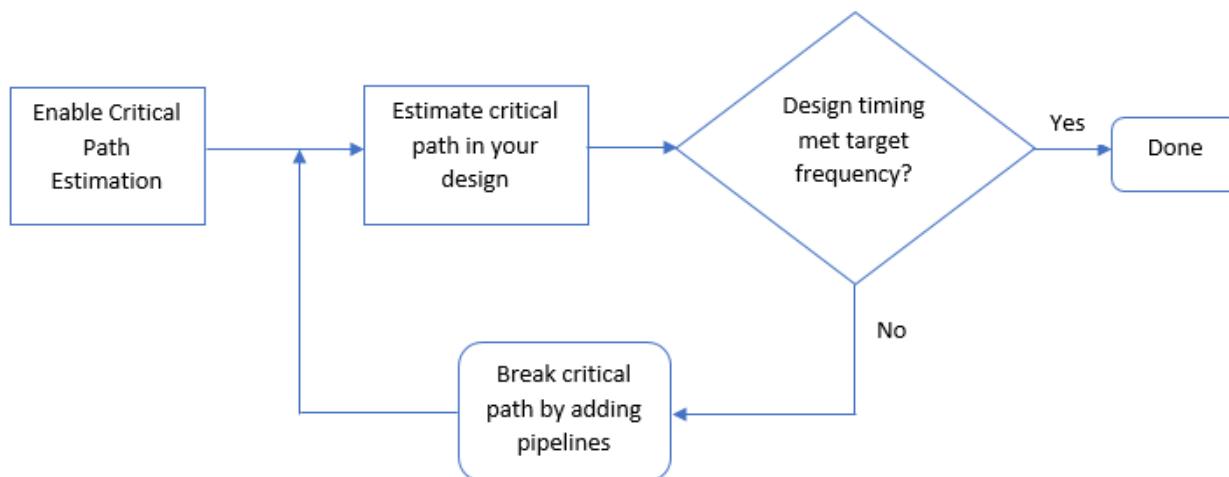
["How to Use Critical Path Estimation" on page 24-138](#)

["Characterized Blocks" on page 24-140](#)

["Caveats" on page 24-143](#)

Critical path is a combinational path between an input and output that has the maximum delay. Use the HDL Coder software to find the critical path in your design. To make the critical path timing meet the target frequency that you want your design to achieve, break the critical path by adding delays. The additional delays do increase the latency and register usage on the target FPGA.

To quickly identify the most likely critical path in your design, use critical path estimation. With critical path estimation, you do not have to run synthesis or generate HDL code. Critical path estimation speeds up this iterative process of finding the critical path, and then optimizing the critical path until your design timing meets the target frequency that you want.



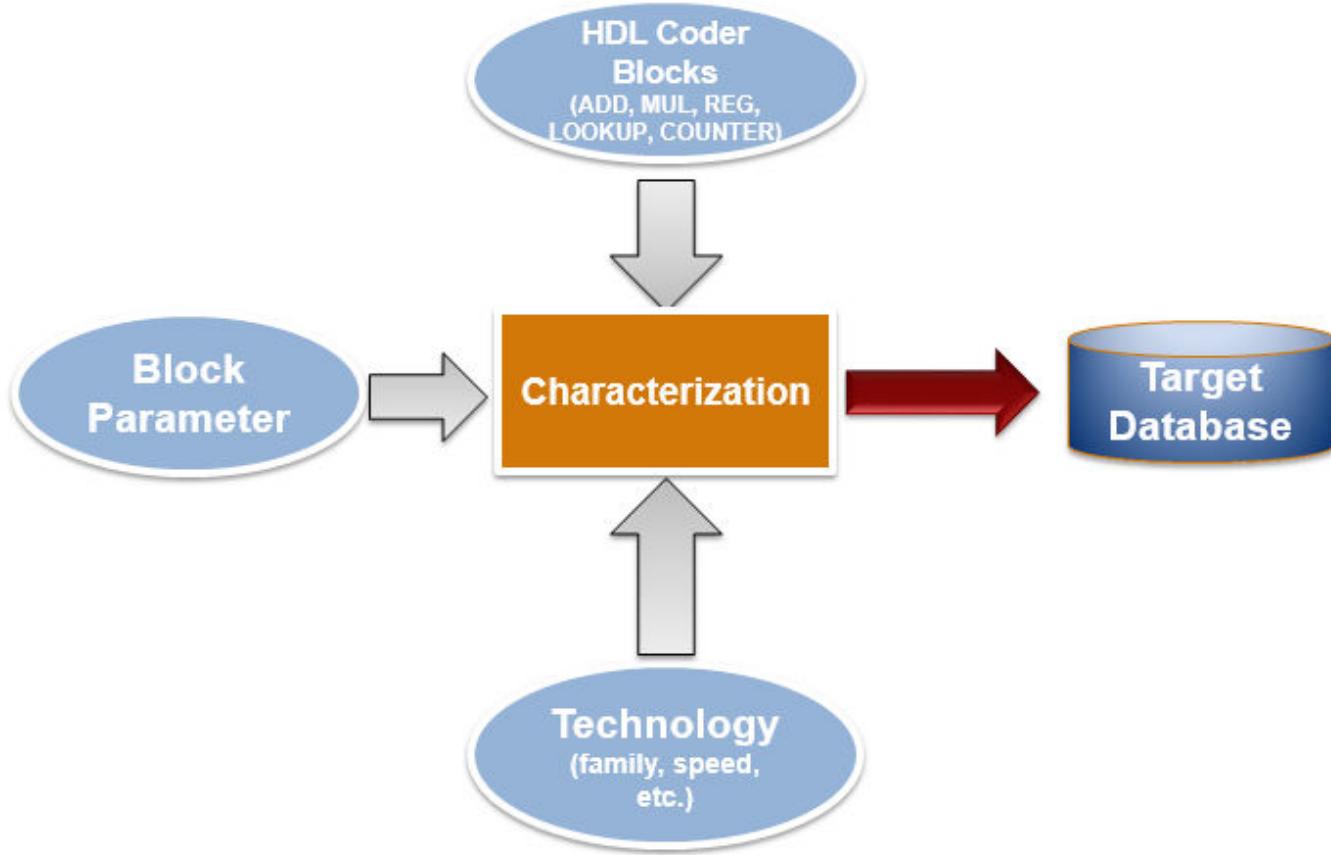
Estimating the critical path without using synthesis tools can lead to inaccurate timing results. Critical path estimation is intended to speed up the design iteration process. Critical path estimation is an alternative to annotating the critical path by performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor.

How Critical Path Estimation Works

HDL Coder finds the estimated critical path by performing static timing analysis with timing data from target-specific timing databases. HDL Coder has timing databases for these target devices:

- Altera Cyclone V
- Intel Stratix V
- Xilinx Virtex-7, speed grade -1
- Xilinx Zynq, speed grade -1

To create timing databases, HDL Coder characterizes basic design components, such as Simulink blocks, block architectures, and subcomponents of those blocks, for specific target devices.



The code generator analyzes your model design to decompose it into the blocks and subcomponents in the timing databases. If your design consists of blocks or subcomponents in the timing databases, the code generator can estimate the timing critical path more accurately. If your design uses components that are not in the timing databases, a separate highlighting script is generated to show the uncharacterized blocks. If the timing data is incomplete for parts of your design, it is possible that the estimated critical path does not match your actual critical path.

If your target hardware is one of the target devices supported for critical path estimation, the timing numbers and estimated critical path are more accurate. If your target hardware is not a supported device, or is not in the same device family, you can estimate the critical path, but it is possible that the timing numbers are not accurate.

How to Use Critical Path Estimation

You can estimate the critical path for your design either in the Configuration Parameters dialog box or at the command line. To estimate the critical path in the Configuration Parameters dialog box:

- 1 Enable generation of critical path estimation report.

- a In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
 - b Select **Settings > Report Options**, and then select **Generate high-level timing critical path report**.
- 2 Disable HDL code generation for your model. In the **HDL Code Generation > Global Settings > Advanced** tab, clear the **Generate HDL Code** check box.
- To estimate the critical path in your design, you do not have to run the complete code generation process. When you disable HDL code generation, you run the process until HDL Coder creates the generated model and displays the critical path estimation script. You avoid running a larger portion of the code generation process, which saves time in estimating the critical path, especially for large models.
- 3 If your design contains floating-point data types, enable the **Native Floating Point** mode. In the Configuration Parameters dialog box, on the **HDL Code Generation > Floating Point** pane, set **Floating Point IP Library** to **Native Floating Point**.
 - 4 Generate a critical path estimation report. In the **HDL Code Generation** pane, click **Apply**, and then click **Generate**.

HDL Coder generates a critical path estimation report and displays messages in the MATLAB Command Window that include a link to a highlighting script and a script that clears the highlighting.

To script this workflow or generate the report at the command line, enter these commands. Specify the `modelname` and `dutname` based on the design that you want to estimate the critical path for. This example uses the `sfir_single` model.

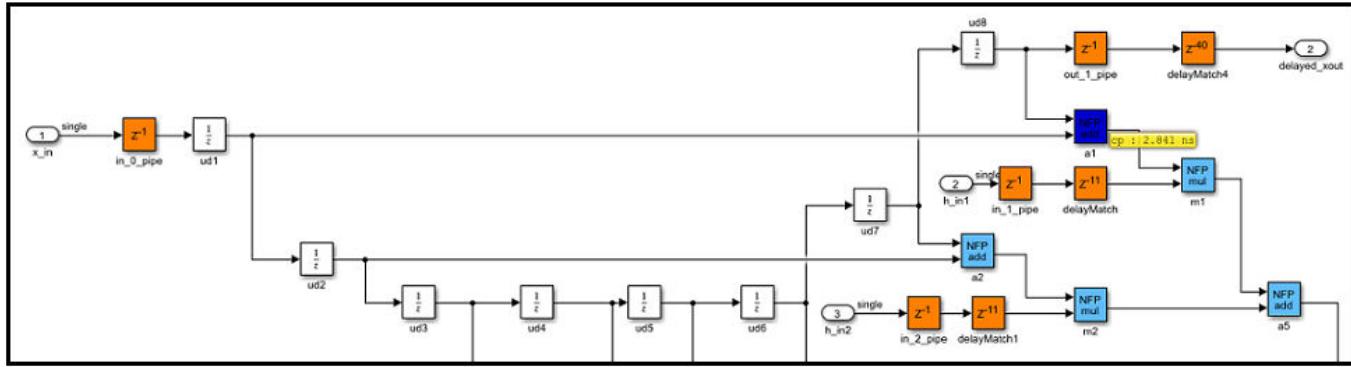
```
% Specify the model and Subsystem names
modelname = 'sfir_single';
dutname = 'sfir_single/symmetric_fir';
open_system(modelname)

% Disable HDL code generation for faster generation
% of critical path estimation report
hdlset_param(modelname, 'CriticalPathEstimation', 'on');
hdlset_param(modelname, 'GenerateHDLCode', 'off');

% If your design contains single data types,
% enable native floating-point support
fpconfig = hdlcoder.createFloatingPointTargetConfig('NativeFloatingPoint');
hdlset_param(modelname, 'FloatingPointTargetConfig', fpconfig);

% Generate the report
makehdl(dutname)
```

When you click the link to the `criticalpathestimated` script, the code generator highlights the critical path in the generated model. In the generated model, you see the critical path timing information and blocks that are on this path. This figure shows a section of a Simulink model that has the critical path annotated. The native floating-point operators are highlighted in light blue and the delays are highlighted in orange. The block that is part of the critical path is highlighted in dark blue with the critical path value annotated beside the block. To learn more about the different colors, see “Generated Model and Validation Model” on page 24-10.



You can clear the highlighting by clicking the link to the [clearhighlighting](#) script.

To optimize the critical path, break the critical path by adding pipeline registers. If you are using Native Floating Point, set the **LatencyStrategy** to Max to improve timing. Regenerate the critical path estimation report and the script that highlights the critical path in your design. You can repeat this process until your design timing meets the target frequency that you want.

Characterized Blocks

This table shows blocks that are characterized with fixed-point and single-precision native floating-point types. These blocks are part of the timing database for each supported target device.

Math Operations

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Abs	✓	✓
Add	✓ (The block cannot have more than two inputs)	✓
Subtract	✓	✓
Product	✓	✓
Gain	✓	✓
Divide	✓	✓
HDL Reciprocal	✓	✓
Rounding Function	✓	✓
Unary Minus	✓	✓
Sign	✓	✓
Reshape	✓	✓
Complex to Real-Imag	✓	✓

Math Functions

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Reciprocal	✓	✓
Hypot	-	✓
Rem	-	✓
Mod	-	✓
Sqrt	✓	✓
Reciprocal Sqrt	✓	✓

Trigonometric Functions

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Sin	-	✓
Cos	-	✓
Tan	-	✓
Sincos	-	✓
Asin	-	✓
Acos	-	✓
Atan	-	✓
Atan2	-	✓
Sinh	-	✓
Cosh	-	✓
Tanh	-	✓
Atanh	-	✓

Exponent/Logarithm/Power

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Exp	-	✓
Gain to power of two	-	✓
Pow10	-	✓
Log	-	✓
Log10	-	✓

Conversions and Comparisons

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Data Type Conversion	✓	✓
Float Typecast	-	✓
Relational Operator	✓	✓
Compare To Constant	✓	✓
MinMax	✓	✓

Logic and Bit Operations

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Bit Concat	✓	-
Extract Bits	✓	-
Bit Shift	✓	-
Bit Slice	✓	-
Bitwise Operator	✓	-
Logical Operator	✓	✓

Delays and Signal Routing

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Unit Delay	✓	✓
Delay	✓	✓
Bus Creator	✓	✓
Bus Selector	✓	✓
Demux	✓	✓
Multiport Switch	✓	✓
Selector	✓	✓
Switch	✓	✓

HDL Operations and HDL RAMs

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Counter Free-Running	✓	✓
Counter Limited	✓	✓
HDL Counter	✓	✓
Dual Port RAM	✓	✓
Dual Rate Dual Port RAM	✓	✓
Simple Dual Port RAM	✓	✓
Single Port RAM	✓	✓
Deserializer1D	✓	✓
Serializer1D	✓	✓

Signal Attributes and Lookup Tables

Simulink Blocks	Fixed Point	Single (Native Floating Point)
Constant	✓	✓
1-D Lookup Table	✓	✓
2-D Lookup Table	✓	✓
n-D Lookup Table	✓	✓
Rate Transition	✓	✓
Signal Conversion	✓	✓
Signal Specification	✓	✓

User-Defined Functions

Simulink Blocks	Fixed Point	Single (Native Floating Point)
MATLAB Function	-	✓

When you use MATLAB Function blocks and generate code by using the MATLAB Datapath architecture, HDL Coder converts the MATLAB algorithm to a Simulink block diagram. In the generated model, critical path estimation can annotate the critical path inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks. See also “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-146.

Caveats

Critical Path Estimation for Multirate Models

Critical path estimation does not consider the clock gating information to different sequential elements in your design.

If your model contains multiple sample rates or uses speed and area optimizations that insert pipeline registers, your design becomes multirate and can have multicycle paths. For multirate models,

critical path estimation treats the slow and fast data paths to be running at the same rate. A data path that has a faster clock rate might be highlighted as the critical path when the design has another data path at a slower rate. This might cause critical path estimation to report inaccurate timing results.

To verify the estimated critical path information, open the HDL Workflow Advisor and run the Generic ASIC/FPGA workflow for your target device to the **Annotate model with synthesis result** task.

Critical Path Estimation with Native Floating Point

If you have **single** data types in your design and you use the **Native Floating Point** mode, the critical path estimation script sometimes highlights a single floating-point operator in the generated model. The code generator highlights a single block because floating-point algorithms are computation-intensive, and the critical path can be an internal register-to-register path within the floating-point operator.

In this case, to optimize the critical path timing, set the **LatencyStrategy** to **Max** for the Simulink block corresponding to that operator.

In addition, critical path estimation with native floating-point does not support for blocks with **LatencyStrategy** set to **Custom**.

HDL Code Generation Behavior

When you enable critical path estimation, it is possible that the generated HDL code is different from the report for a Delay block that has an external reset or an enable port. In addition, for blocks such as MinMax, the number of generated HDL files might differ when you enable critical path estimation. This change occurs due to certain optimizations performed by the code generator when you enable this optimization. The optimization only changes how the code appears and does not affect the functionality.

Following are the Simulink blocks for which the generated HDL code can potentially be different.

- Delay block that has external reset or enable port
- MinMax
- Unit Delay Enabled Synchronous
- Unit Delay Resettable Synchronous
- Unit Delay Enabled Resettable Synchronous
- Enabled Delay
- Resettable Delay
- Tapped Delay
- Discrete FIR Filter
- Biquad Filter
- MATLAB Function

Inaccuracy in Critical Path Estimation

- Critical path estimation tries to account for routing delay by using an estimation factor. Without running place and route, it is difficult to accurately account for routing delay.

- HDL Coder infers uncharacterized blocks that are combinational in nature as zero-delay combinational blocks. The code generator treats other blocks as registers.
- If your target device does not have timing characteristics that are similar to one of the supported target devices, critical path estimation cannot accurately compute your critical path.

See Also

`hdlcoder.FloatingPointTargetConfig` | `makehdl`

More About

- “Create and Use Code Generation Reports” on page 25-2
- “Generated Model and Validation Model” on page 24-10
- “Distributed Pipelining” on page 8-15
- “Getting Started with HDL Coder Native Floating-Point Support” on page 10-80

HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture

This example shows how to use various optimizations inside the MATLAB Function block and across the MATLAB Function block boundary with other blocks in your Simulink® model. The example also illustrates the difference in area and timing when you use different HDL architecture settings of the MATLAB Function block.

Why Use MATLAB Datapath Architecture?

HDL code generation for a MATLAB Function block supports two HDL architectures: **MATLAB Function** and **MATLAB Datapath**. Specify the **HDL Architecture** in the HDL Block Properties dialog box of the MATLAB Function block.

Use the **MATLAB Datapath** architecture to:

- Model complex fixed-point and floating-point MATLAB algorithms inside MATLAB Function blocks and interface this algorithm with other Simulink blocks in your model.
- Improve area and timing of your design significantly by optimizing the algorithm inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks in your model.

This architecture is the default setting for MATLAB Function blocks with floating-point types. By enabling the **MATLAB Datapath** architecture for fixed-point operations, you can use various optimizations that include:

- Hierarchy flattening
- Resource sharing and streaming
- Clock-rate pipelining
- Adaptive pipelining
- Distributed pipelining and hierarchical distributed pipelining
- Critical path estimation

How MATLAB Datapath Architecture Works

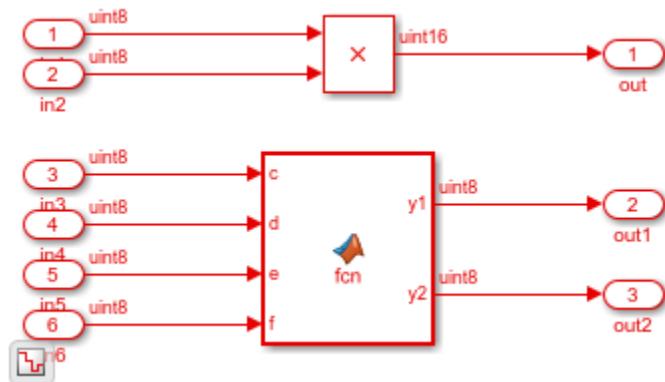
Fixed-point Simulink® models use the **MATLAB Function** architecture by default. Certain HDL optimizations such as resource sharing and distributed pipelining that you enable with this architecture optimize the blocks surrounding the MATLAB Function block and the algorithm inside the MATLAB Function block. To see the effect of the optimizations inside the MATLAB Function block, examine the generated HDL code for the block. This architecture does not apply the optimizations across the MATLAB Function block boundary with other Simulink blocks.

Floating-point Simulink models use the **MATLAB Datapath** architecture even if you specify **MATLAB Function** as the architecture setting for the block. When you use floating-point types, specify the native floating-point mode. With this architecture, the code generator treats the block like a regular Subsystem block. HDL Coder transforms the control flow algorithm of the MATLAB code inside the MATLAB Function block to a dataflow representation that uses Simulink blocks. The **MATLAB Datapath** architecture unrolls loops in your code due to this transformation. If you want to stream loops, either use the loop streaming optimization with the **MATLAB Function** architecture or use the streaming optimization with **MATLAB Datapath** as the HDL architecture.

By using the MATLAB Datapath architecture, You can more effectively perform various HDL Coder™ optimizations with the MATLAB Function block that you would otherwise perform with a Subsystem block. The MATLAB Datapath architecture applies the optimization settings that you specify on the algorithm inside the MATLAB Function block and across the MATLAB Function block boundary with other blocks in your Simulink model.

For example, consider this model with a DUT Subsystem that consists of a Product block and a MATLAB Function block.

```
open_system('hdlcoder_MLFB_simple_datapath')
set_param('hdlcoder_MLFB_simple_datapath', 'SimulationCommand', 'Update')
open_system('hdlcoder_MLFB_simple_datapath/HDL_DUT')
```



The MATLAB Function block implements two multiplications.

```
open_system('hdlcoder_MLFB_simple_datapath/HDL_DUT/MATLAB_Function')

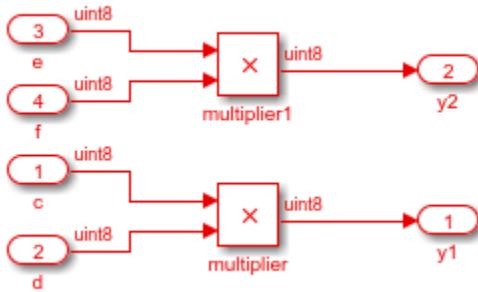
function [y1, y2] = fcn(c, d, e, f)
 %#codegen

y1 = c * d;
y2 = e * f;
```

The HDL architecture of the MATLAB Function block is set to MATLAB Datapath. To generate HDL code for the HDL_DUT Subsystem, run this command:

```
makehdl('hdlcoder_MLFB_simple_datapath/HDL_DUT')
```

When you generate HDL code, the code generator replaces the MATLAB Function block with a subsystem that performs the multiplications $c * d$ and $e * f$.



With the MATLAB Datapath architecture, you can perform optimizations inside the MATLAB Function block and across the MATLAB Function block with other Simulink blocks. In this example, you can share the two multipliers inside the MATLAB Function block. To optimize the blocks, set the **SharingFactor** to 2 on the MATLAB Function block.

```
mlsubsys = 'hdlcoder_MLFB_simple_datapath/HDL_DUT/MATLAB Function';
hdlset_param(mlsubsys, 'SharingFactor', 2)
```

When you generate HDL code, the code generator shares the multiplications inside the MATLAB Function block. The sharing group is displayed in the Optimization Report. When you click the links in the sharing group, HDL Coder displays the shared multipliers inside the MATLAB Function block in the generated model and the original model.

Sharing Report

Subsystem: [MATLAB Function](#)

SharingFactor: 2

[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	8x8 -> 16	2	multiplier	

You can apply the optimization across the MATLAB Function block with other Simulink blocks. In this example, you can share the Product block outside the MATLAB Function block with the multipliers inside the MATLAB Function block. To share these resources, remove the **SharingFactor** on the MATLAB Function block, and on the parent subsystem, **HDL_DUT**, enable **FlattenHierarchy** and set **SharingFactor** to 3.

```
hdlset_param(mlsubsys, 'SharingFactor', 0)
hdlset_param('hdlcoder_MLFB_simple_datapath/HDL_DUT',...
    'FlattenHierarchy', 'on', 'SharingFactor', 3)
```

Note: Do not use the **InlineMATLABCode** property with the MATLAB Datapath architecture of the block. Use **FlattenHierarchy** instead.

When you generate HDL code, the code generator shares the multiplications inside the MATLAB Function block with the Product block outside. You see the sharing group of three multipliers in the

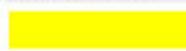
Optimization Report. When you click the links in the sharing group, the shared multipliers are highlighted in the generated model and the original model.

Sharing Report

Subsystem: [HDL_DUT](#)

SharingFactor: 3

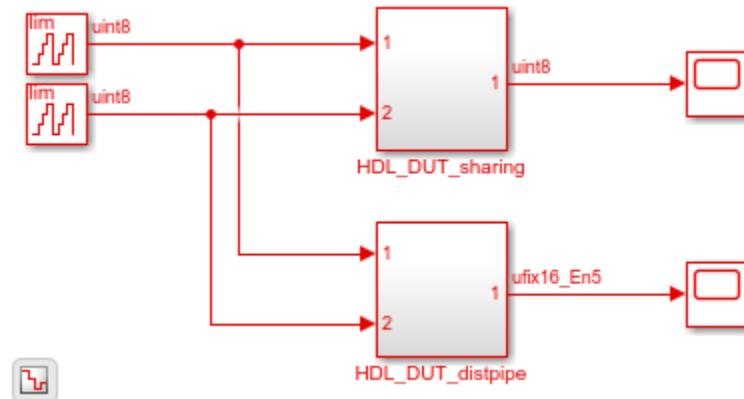
[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	8x8 -> 16	3	Product	

MATLAB Function Block Model With Default MATLAB Function Architecture

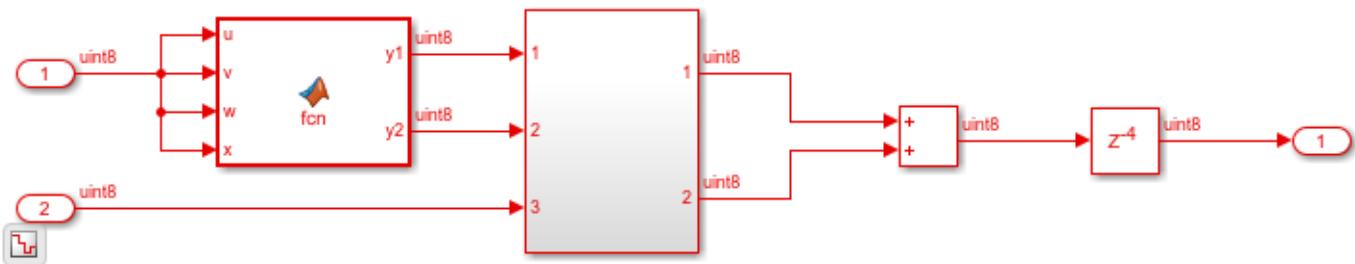
For an example model that illustrates the MATLAB Datapath architecture and how it differs from the MATLAB Function architecture, open the model `hdlcoder_MLFB_share_pipeline`. The model uses integer types. For an example that illustrates how you use the MATLAB Datapath architecture with floating-point types, see “Generate Target-Independent HDL Code with Native Floating-Point” on page 10-103.

```
open_system('hdlcoder_MLFB_share_pipeline')
set_param('hdlcoder_MLFB_share_pipeline','SimulationCommand','Update')
```

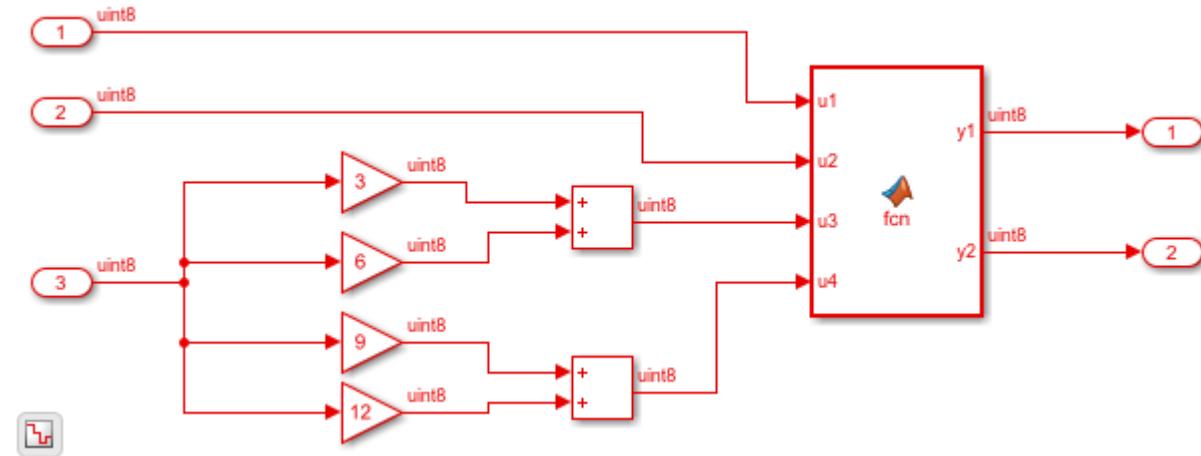


The model contains two DUT subsystems at the top level `HDL_DUT_sharing` and `HDL_DUT_distpipe`. The subsystems illustrate how you can use resource sharing and distributed pipelining optimizations across the MATLAB Function block boundary with other blocks. Both subsystems perform basic additions and multiplications inside and outside the MATLAB Function block.

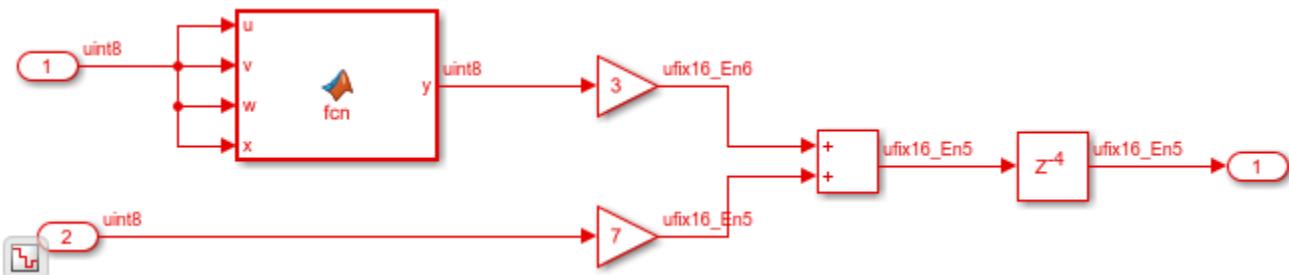
```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```



```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/Subsystem')
```



```
open_system('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe')
```



To see the HDL parameters that are saved on the model, run the `hdlsaveparams` function.

```
hdlsaveparams('hdlcoder_MLFB_share_pipeline')
```

```
% Set Model 'hdlcoder_MLFB_share_pipeline' HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline', 'CriticalPathEstimation', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'HDLSubsystem', 'hdlcoder_MLFB_share_pipeline/HDL_DU
hdlset_param('hdlcoder_MLFB_share_pipeline', 'OptimizationReport', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'Oversampling', 40);
hdlset_param('hdlcoder_MLFB_share_pipeline', 'ResourceReport', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'ShareAdders', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_MLFB_share_pipeline', 'Traceability', 'on');
```

```
% Set SubSystem HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe', 'DistributedPipelining', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe', 'FlattenHierarchy', 'on');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing', 'FlattenHierarchy', 'on');
hdlset_param('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing', 'SharingFactor', 8);
```

You see that the default MATLAB Function HDL architecture is saved on the model.

Generate HDL Code using MATLAB Function Architecture

To generate HDL code for the sharing DUT, run this command:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing')
```

When you open the Streaming and Sharing Report, the report displays four multipliers and three adders as shared resources.

Sharing Report

Subsystem: [HDL_DUT_sharing](#)

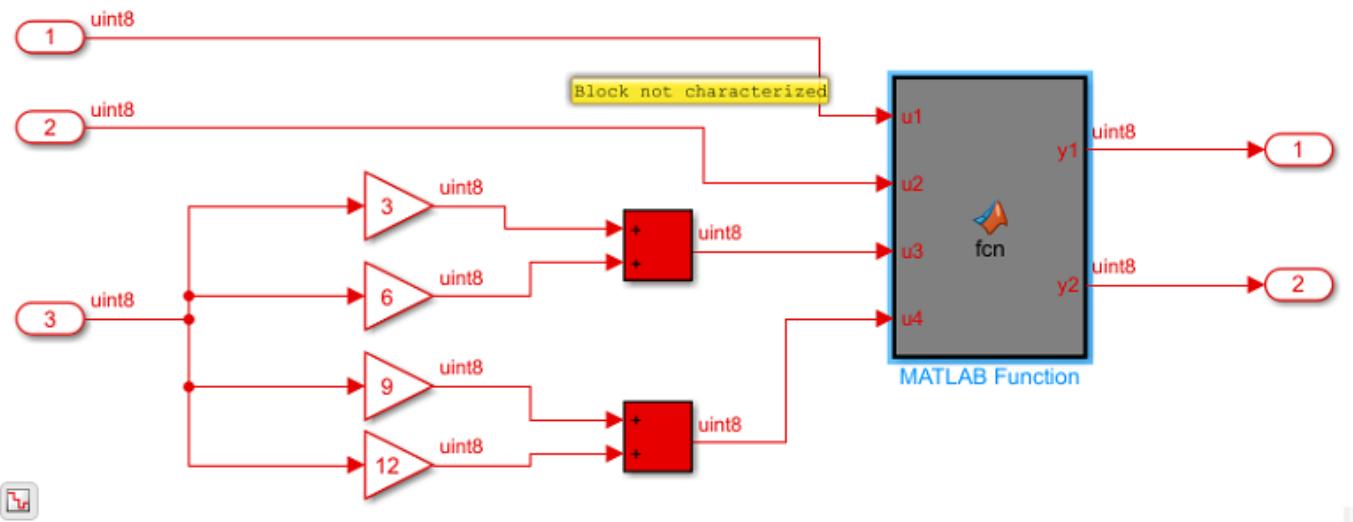
SharingFactor: 8

[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	9x9 -> 17	4	Gain2	
2	Sum	8+8 -> 8	3	Add1	

When you click the second sharing group, the code generator highlights three adders surrounding the MATLAB Function block. The sharing group includes the two adders inside the Subsystem and the Add block outside. The code generator did not share the multipliers and adders that are inside the MATLAB Function block.

Critical path estimation is enabled on the model. When you annotate the critical path, the MATLAB Function block acts as a barrier to this optimization. If the critical path is inside the MATLAB Function block and if you want to highlight the critical path, use the **MATLAB Datapath** architecture.



To generate HDL code for `HDL_DUT_distpipe`, run this command:

```
makehdl('hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe')
```

When you open the Distributed Pipelining Report, you see that the code generator moved pipelines inside the `HDL_DUT_distpipe` subsystem but did not distribute pipelines inside the MATLAB Function block.

Detailed Report

Subsystem: [MATLAB Function](#)

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Status: Distributed Pipelining unsuccessful.

Subsystem: [HDL_DUT_distpipe](#)

Implementation Parameters: DistributedPipelining: 'on'; InputPipeline: 0; OutputPipeline: 0

Status: Distributed Pipelining successful.

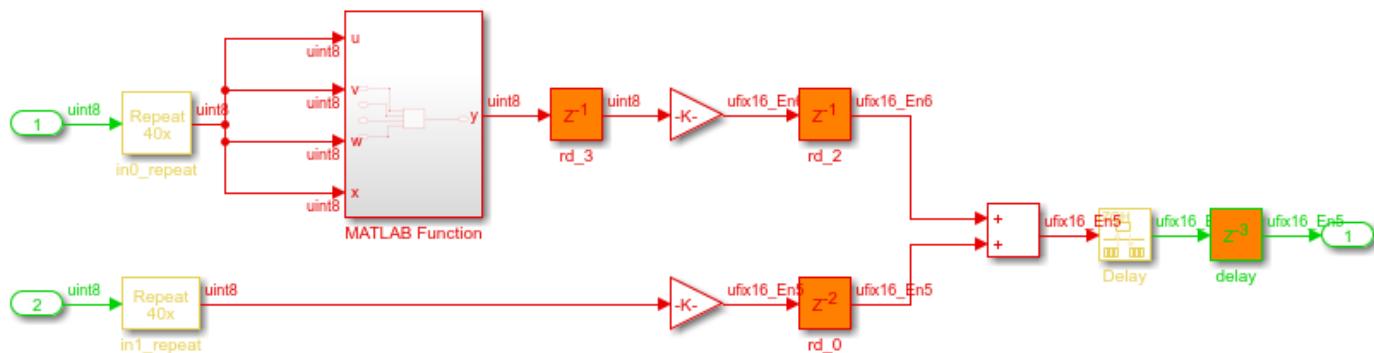
Before Distributed Pipelining : 7 registers (104 flip-flops)

Registers	Count
16-bit	6
8-bit	1

After Distributed Pipelining : 7 registers (104 flip-flops)

Registers	Count
8-bit	1
16-bit	6

This figure displays how distributed pipelining moved pipeline registers inside the subsystem.



Apply Optimizations Across MATLAB Function Block and Other Simulink Blocks

To improve area and timing of your design, use the MATLAB Datapath architecture. For the HDL_DUT_sharing subsystem, you can combine resource sharing with clock-rate pipelining and share the resources inside the MATLAB Function block and across the MATLAB Function block with other blocks.

To share resources:

1. Enable **FlattenHierarchy** and specify a **SharingFactor** on the parent subsystem HDL_DUT_sharing. Set the **SharingFactor** to 8.

```
share_subsys = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing';
hdlset_param(share_subsys, 'FlattenHierarchy', 'on', 'SharingFactor', 8);
```

2. Specify the MATLAB Datapath architecture for the MATLAB Function blocks inside the HDL_DUT_sharing subsystem.

```
share_mlfcn1 = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/MATLAB Function';
share_mlfcn2 = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_sharing/Subsystem/MATLAB Function';
hdlset_param(share_mlfcn1, 'architecture', 'MATLAB Datapath');
hdlset_param(share_mlfcn2, 'architecture', 'MATLAB Datapath');
```

When you open the Streaming and Sharing report, the report displays a sharing group of eight multipliers and two sharing groups of adders.

Sharing Report

Subsystem: [HDL_DUT_sharing](#)

SharingFactor: 8

[Highlight shared resources and diagnostics](#)

Group Id	Resource Type	I/O Wordlengths	Group Size	Block Name	Color Legend
1	Product	9x9 -> 17	8	multiplier	
2	Sum	8+8 -> 8	4	adder	
3	Sum	8+8 -> 8	2	Add1	

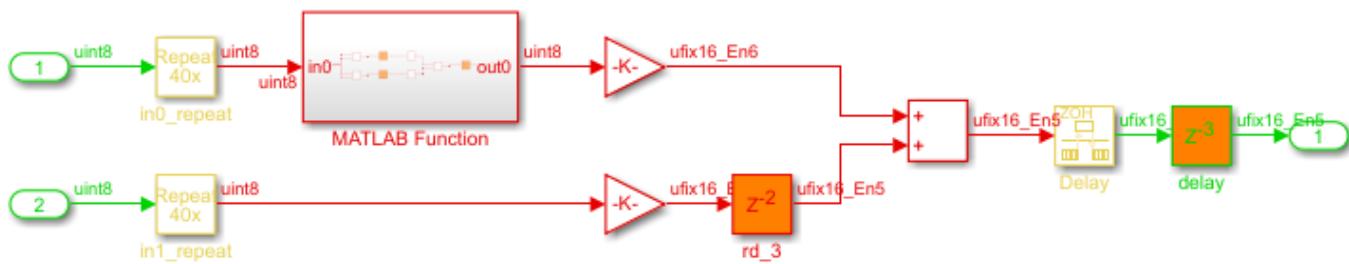
When you select the first sharing group, you see that the optimization shared the multipliers inside the MATLAB Function with the four Gain blocks outside. The second sharing group consists of adders inside each of the two MATLAB Function blocks.

You can use the distributed pipelining optimization with the **Distributedpipe_MLFB** subsystem. Enable hierarchical distributed pipelining at the top level and set the HDL architecture of the MATLAB Function to MATLAB Datapath with **DistributedPipelining** set to on.

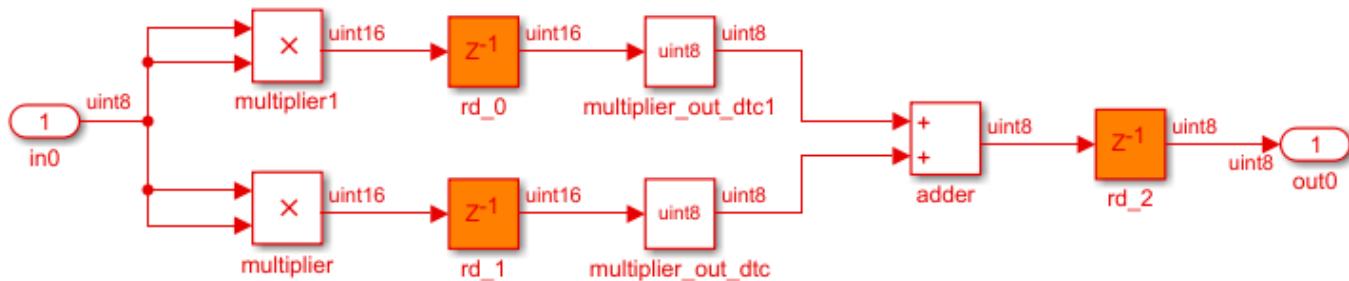
```
hdlset_param('hdlcoder_MLFB_share_pipeline', 'HierarchicalDistPipelining', 'on');
dist_subsys = 'hdlcoder_MLFB_share_pipeline/HDL_DUT_distpipe/MATLAB Function';
```

```
hdlset_param(dist_subsys, 'architecture', 'MATLAB Datapath');
hdlset_param(dist_subsys, 'DistributedPipelining', 'on');
```

After you generate HDL code, open the generated model. The code generator uses hierarchical distributed pipelining and distributed pipelining to move the pipeline registers across the blocks and inside the MATLAB Function Subsystem.



This figure displays how distributed pipelining moved pipeline registers inside the MATLAB Function Subsystem.



See Also

Blocks

MATLAB Function

Functions

`hdlsaveparams` | `makehdl`

More About

- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37
- “RAM Mapping With the MATLAB Function Block” on page 24-96

Subsystem Optimizations for Filters

The Discrete FIR Filter (when used with scalar or multichannel input data) and Biquad Filter blocks participate in subsystem-level optimizations. To set optimization properties, right-click on the subsystem and open the **HDL Properties** dialog box.

For these blocks to participate in subsystem-level optimizations, you must leave the block-level **Architecture** set to the default, **Fully parallel**.

You cannot use these subsystem optimizations when using the Discrete FIR Filter in frame-based input mode.

Sharing

These filter blocks support sharing resources within the filter and across multiple blocks in the subsystem. When you specify a **SharingFactor**, the optimization tools generate a filter implementation in HDL that shares resources using time-multiplexing. To generate an HDL implementation that uses the minimum number of multipliers, set the **SharingFactor** to a number greater than or equal to the total number of multipliers. The sharing algorithm shares multipliers that have the same input and output data types. To enable sharing between blocks, you may need to customize the internal data types of the filters. Alternatively, you can target a particular system clock rate with your choice of **SharingFactor**.

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

For more information, see “Resource Sharing” on page 24-32 and the “Area Reduction of Multichannel Filter Subsystem” on page 24-157 example.

You can also use a **SharingFactor** with multichannel filters. See “Area Reduction of Filter Subsystem” on page 24-162.

Streaming

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

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

See “Streaming” on page 24-29 and the “Area Reduction of Filter Subsystem” on page 24-162 example.

Pipelining

You can enable **DistributedPipelining** at the subsystem level to allow the filter to participate in pipeline optimizations. The optimization tools operate on the **InputPipeline** and **OutputPipeline**

pipeline stages specified at subsystem level. The optimization tools also operate on these block-level pipeline stages:

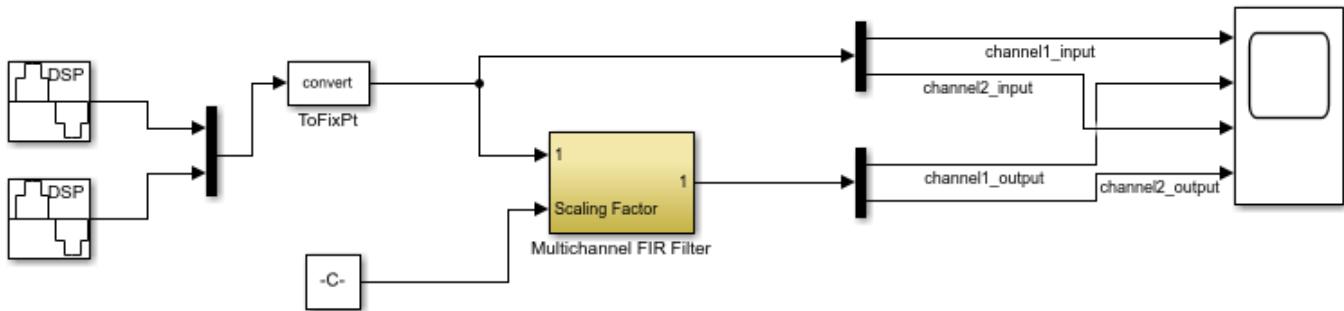
- **InputPipeline** and **OutputPipeline**
- **MultiplierInputPipeline** and **MultiplierOutputPipeline**
- **AddPipelineRegisters**

The optimization tools do not move design delays within the filter architecture. See “Distributed Pipelining” on page 8-15.

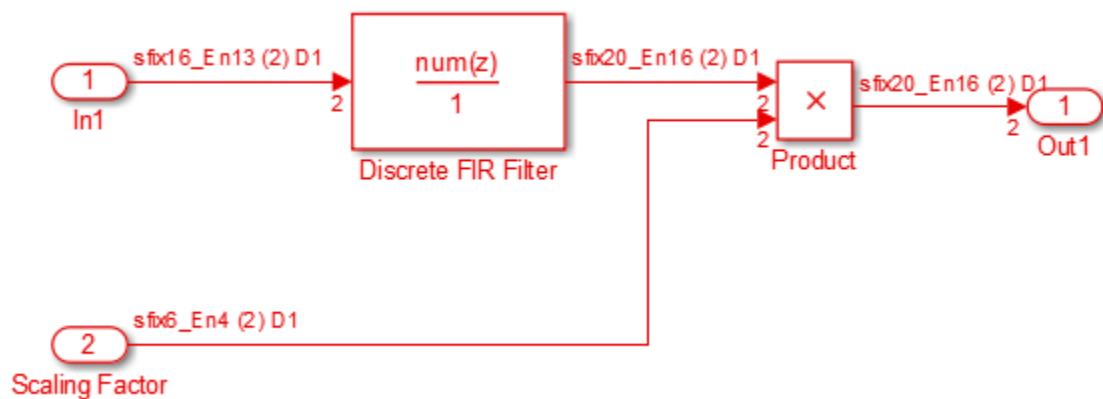
The filter block also participates in clock-rate pipelining, if enabled in **Configuration Parameters**. This feature is enabled by default. See “Clock-Rate Pipelining” on page 24-114.

Area Reduction of Multichannel Filter Subsystem

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

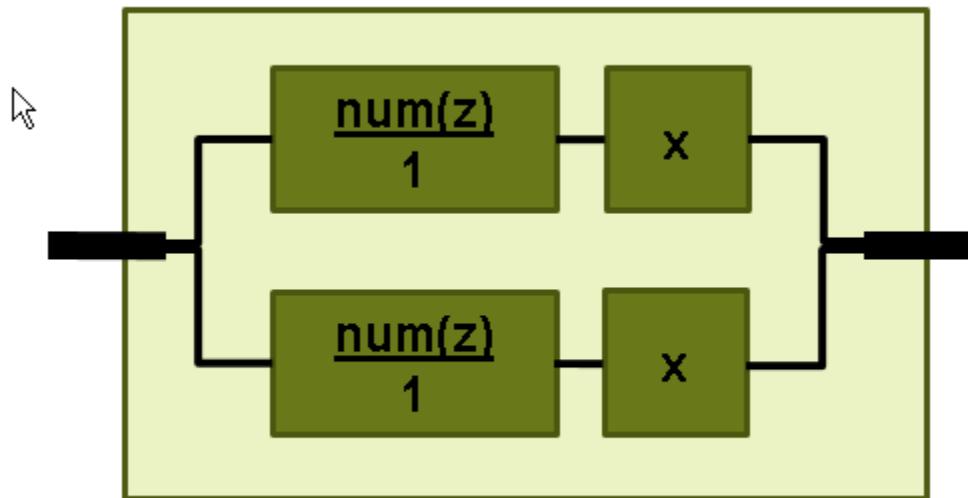


The model includes a two-channel sinusoidal signal source feeding a filter subsystem targeted for HDL code generation.

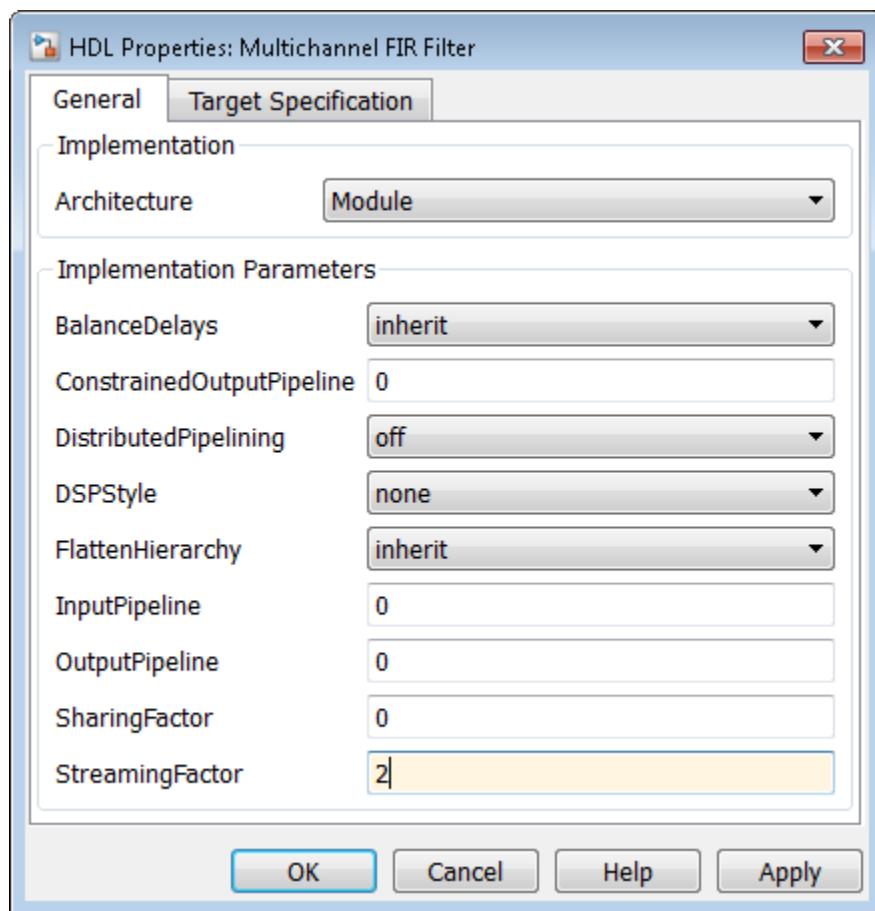


The subsystem contains a Discrete FIR Filter block and a constant multiplier. The multiplier is included to show the optimizations operating over all eligible logic in a subsystem.

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



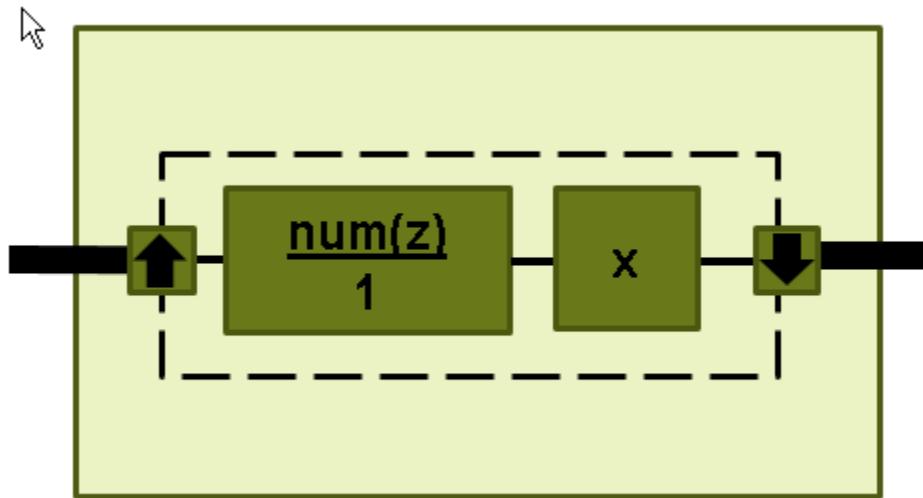
To enable streaming optimization for the Multichannel FIR Filter Subsystem, right-click the subsystem and select **HDL Code > HDL Block Properties**.



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

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

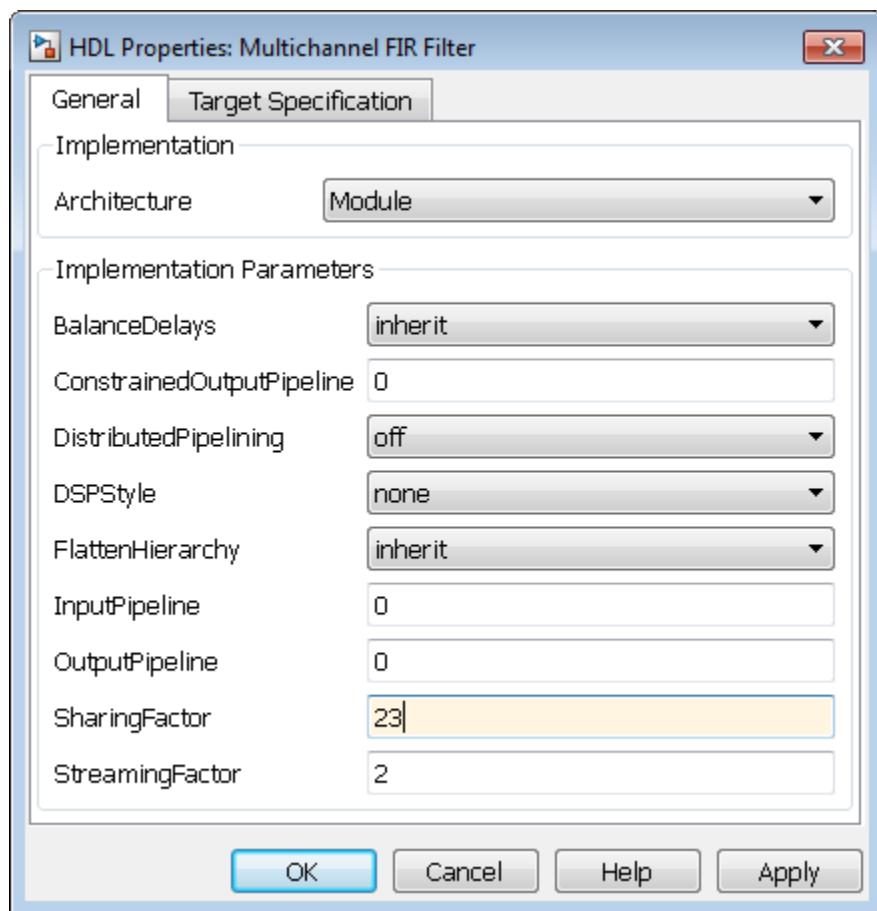
With the streaming factor applied, the logic for one channel is instantiated once and run at twice the rate of the original model.



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

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

To apply **SharingFactor** to multichannel filters, set the **SharingFactor** to 23.



The optimized HDL now uses only 2 multipliers. The optimization tools do not share multipliers of different sizes.

Summary

Multipliers	2
Adders/Subtractors	47
Registers	166
Total 1-Bit Registers	3566
RAMs	0
Multiplexers	37
I/O Bits	80

Detailed Report

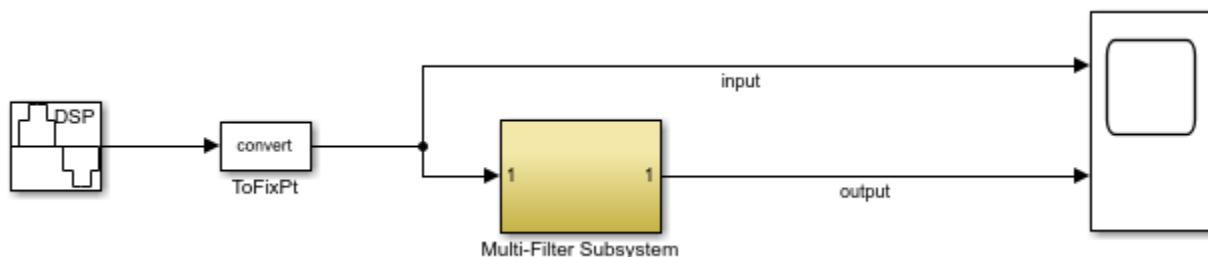
Report for Subsystem: [Multichannel FIR Filter](#)

Multipliers (2)

```
16x16-bit Multiply : 1
[+] 16x6-bit Multiply : 1
```

Area Reduction of Filter Subsystem

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



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

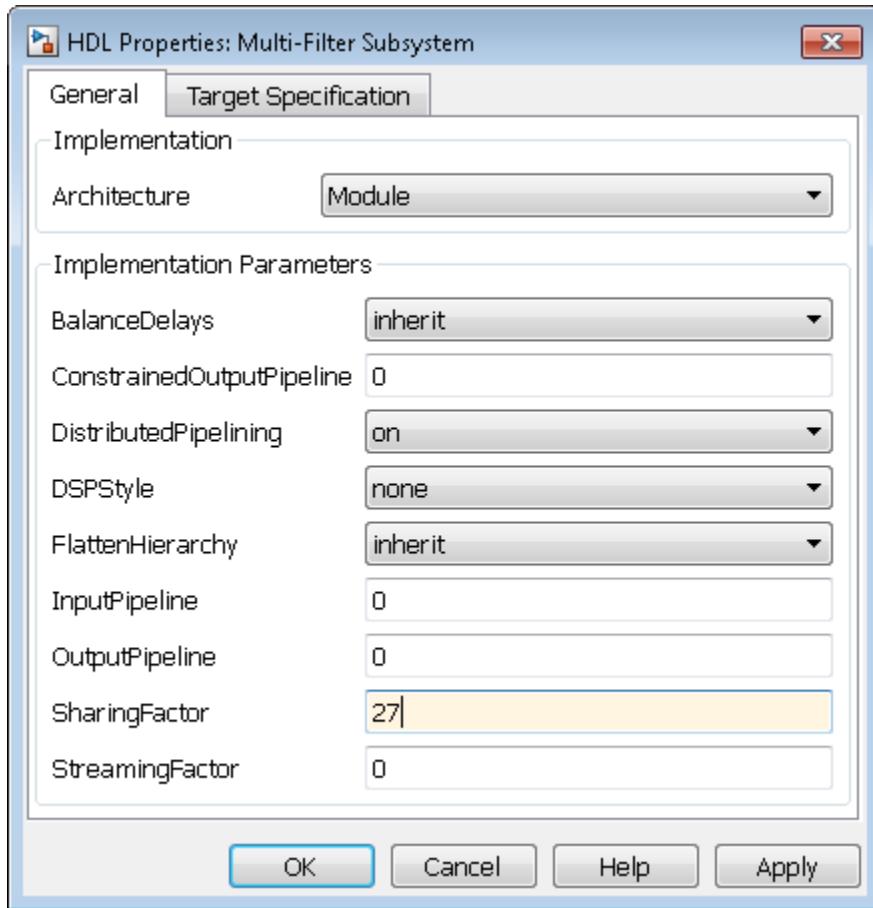


The subsystem contains a Discrete FIR Filter block and a Biquad Filter block. This design demonstrates how the optimization tools share resources between multiple filter blocks.

The Discrete FIR Filter block has 43 symmetric coefficients. The Biquad Filter block has 6 coefficients, two of which are unity. With no optimizations enabled, the generated HDL code takes advantage of symmetry and unity coefficients. The nonoptimized HDL implementation of the subsystem uses 27 multipliers.

Multipliers	27
Adders/Subtractors	46
Registers	49
Total 1-Bit Registers	800
RAMs	0
Multiplexers	0
I/O Bits	52

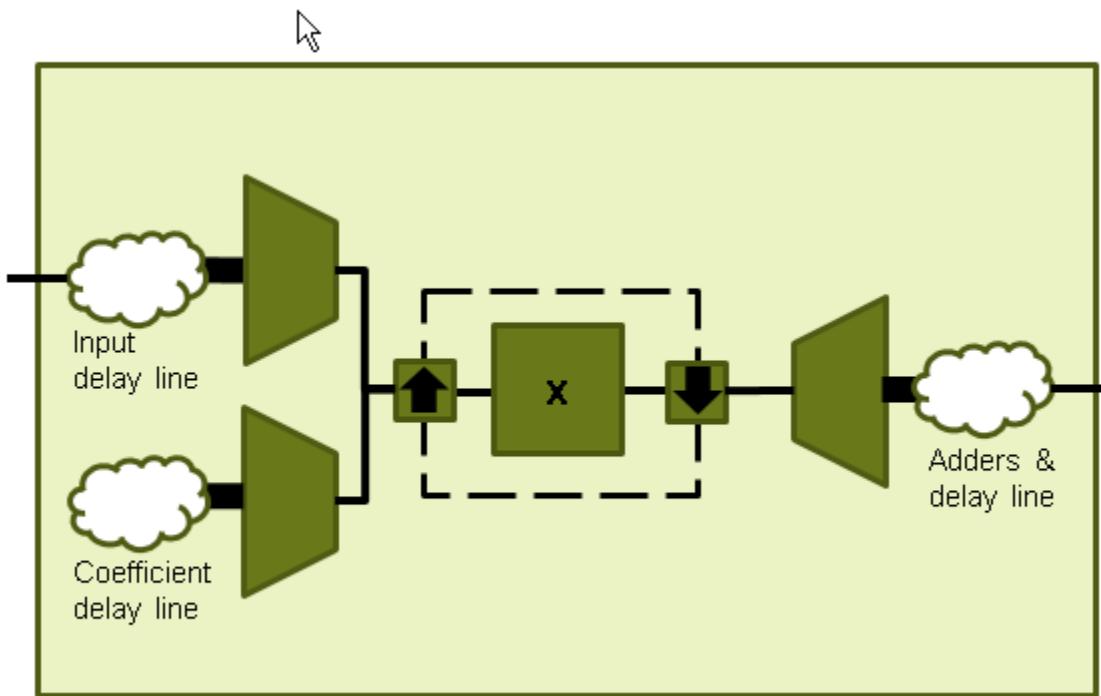
To enable streaming optimization for the **Multi-Filter Subsystem**, right-click the subsystem and select **HDL Code > HDL Block Properties**.



Set the **SharingFactor** to 27 to reduce the design to a single multiplier. The optimization tools attempt to share multipliers with matching data types. To reduce to a single multiplier, you must set the internal data types of the filter blocks to match each other.

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

With the **SharingFactor** applied, the subsystem upsamples the rate by 27 to share a single multiplier for all the coefficients.



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

Multipliers	1
Adders/Subtractors	48
Registers	102
Total 1-Bit Registers	2457
RAMs	0
Multiplexers	7
I/O Bits	52

See Also

More About

- “Resource Sharing” on page 24-32
- “Streaming” on page 24-29
- “Clock-Rate Pipelining” on page 24-114

Remove Redundant Logic and Unused Blocks in Generated HDL Code

If your design contains redundant logic or unused blocks, HDL Coder™ removes the blocks, components, or part of the HDL code that does not contribute to the output.

Redundant Logic Considerations

Components that do not contribute to the output in the design are removed during HDL code generation. Removing redundant logic reduces code size and avoids potential synthesis failures with downstream tools when deploying your code onto a target platform. If a component or logic is preserved during HDL code generation, that component or logic is considered *active*. This optimization improves the performance of your design on the target hardware. The optimization does not affect the traceability support.

Redundant logic in your design is removed in conjunction with unused port deletion optimization. To learn about this optimization, see “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188.

HDL Coder does not treat a component or logic as redundant in these cases:

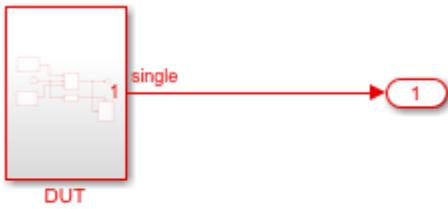
- The component has at least one output port that contributes to the evaluation of the DUT output.
- The component has at least one output port that contributes to the evaluation of the control port of a Subsystem block that is active.
- The component has at least one output port that contributes to the evaluation of input of a component that is preserved during HDL code generation.
- The component is an active black box subsystem, or contains an active black box subsystem, or is connected to an active black box subsystem, as described below.
- The component is an FPGA Data Capture block.

In other cases, the code generator treats the logic as redundant and removes the associated blocks or components during code generation. In addition, blocks that have HDL architecture set to No HDL, such as Scope, Assertion, Terminator, and To Workspace blocks are considered redundant, and are removed during code generation.

How Redundant Logic Removal Works

During code generation, HDL Coder removes redundant logic or blocks that do not contribute to the DUT output. Open the model `hdlcoder_remove_redundant_logic`.

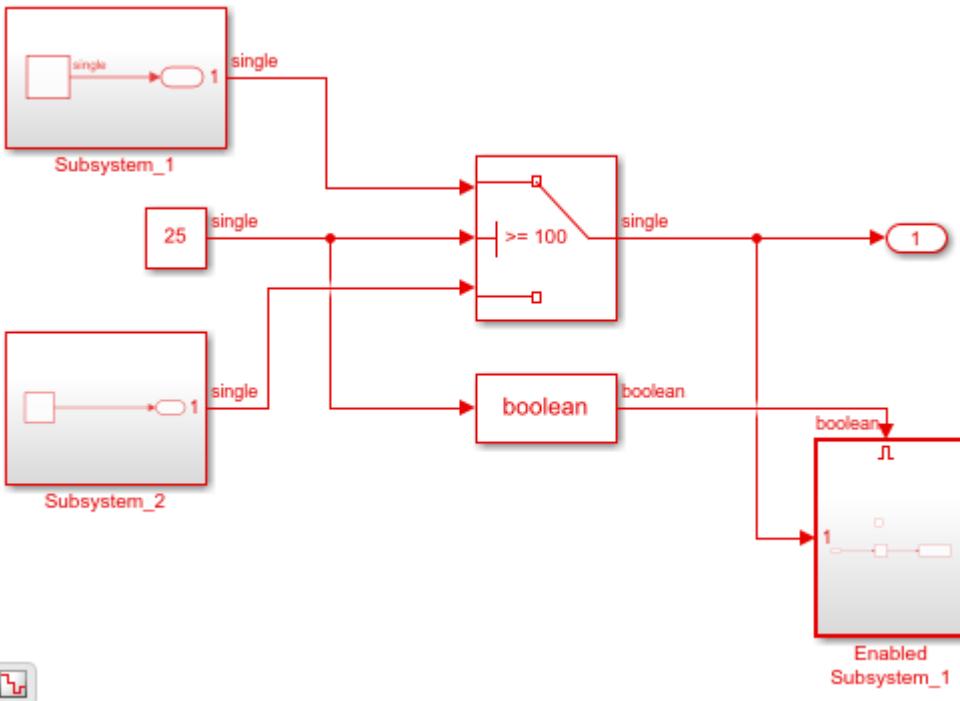
```
open_system('hdlcoder_remove_redundant_logic')
set_param('hdlcoder_remove_redundant_logic', 'SimulationCommand', 'update');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem block contains a Switch block and an Enabled Subsystem block. Based on the control input to the Switch block, the false path from Subsystem_2 is passed to the output. The EnabledSubsystem_1 block output is terminated by a Display block and does not actively contribute to the output.

```
open_system('hdlcoder_remove_redundant_logic/DUT')
```



To generate HDL code for the design, at the MATLAB® command prompt, enter:

```
makehdl('hdlcoder_remove_redundant_logic/DUT')
```

The generated VHDL code shows that HDL Coder evaluated the Switch block condition at compile time to pass the input from Subsystem_2 to the output, and eliminated Subsystem_1 input branch. The EnabledSubsystem_1 block is removed during HDL code generation since it does not have an active output.

```
ARCHITECTURE rtl OF DUT IS
```

```
-- Component Declarations
COMPONENT Subsystem_2
    PORT( Out1      : OUT    std_logic_vector(31 DOWNTO 0)  -- single
          );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : Subsystem_2
    USE ENTITY work.Subsystem_2(rtl);

-- Signals
SIGNAL Subsystem_2_out1      : std_logic_vector(31 DOWNTO 0);  -- ufix32

BEGIN
    u_Subsystem_2 : Subsystem_2
        PORT MAP( Out1 => Subsystem_2_out1  -- single
                  );
Out1 <= Subsystem_2_out1;
END rtl;
```

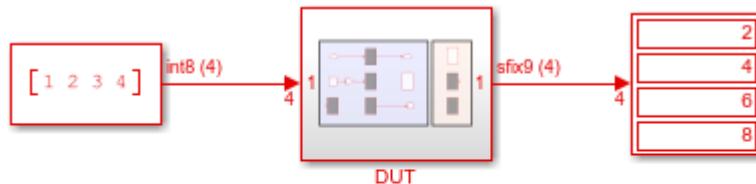
Redundant Logic in Black Box Subsystems

Black box subsystems are subsystem blocks that have HDL architecture set to **BlackBox**. A black box interface for a subsystem is the generated VHDL component or Verilog module that includes only the HDL input and output port definitions for the subsystem. Use the generated interface to integrate existing manually written HDL code, third-party IP, or other code generated by HDL Coder. See “Generate Black Box Interface for Subsystem” on page 27-4.

Black box subsystems that have at least one input port are considered valid and preserved during HDL code generation. The input port can be unconnected. In this case, Simulink® inserts a virtual signal that has a constant zero value as input to the block. In the case of output ports, the black box subsystems must have at least one output port that is connected to a downstream block. When the output port is connected, the code generator identifies the black box subsystem as a source that contributes to the output value computation, and preserves it during code generation.

Open the model `hdlcoder_blackbox_redundant_logic`.

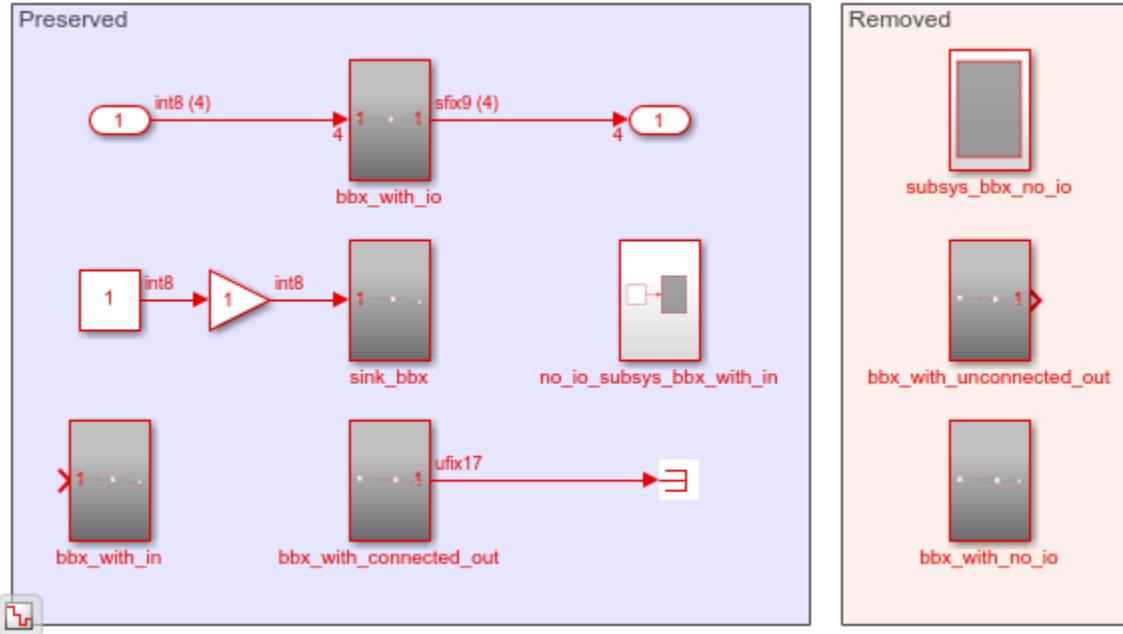
```
open_system('hdlcoder_blackbox_redundant_logic')
sim('hdlcoder_blackbox_redundant_logic');
```



The DUT subsystem contains black box subsystems inside boxes labelled **Preserved** and **Removed**. Black box subsystems inside **Preserved** are not removed during code generation because they have

at least one input port. The other black box subsystems that do not have an input port or have an unconnected output port are removed during code generation.

```
open_system('hdlcoder_blackbox_redundant_logic/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_remove_redundant_logic/DUT')
```

The generated HDL code shows the subsystems in the Preserved section. The unconnected input port is automatically connected to a constant zero.

```
ARCHITECTURE rtl OF DUT IS
  -- Component Declarations
  COMPONENT bbx_with_io
    PORT( clk           : IN  std_logic;
          clk_enable   : IN  std_logic;
          reset        : IN  std_logic;
          In1          : IN  vector_of_std_logic_vector8(0 TO 3);  -- int8 [4]
          Out1         : OUT vector_of_std_logic_vector9(0 TO 3)  -- sfix9 [4]
        );
  END COMPONENT;

  COMPONENT no_io_subsys_bbx_with_in
    PORT( clk           : IN  std_logic;
          reset        : IN  std_logic;
          enb          : IN  std_logic
        );
  END COMPONENT;

  COMPONENT bbx_with_in
    PORT( clk           : IN  std_logic;
          clk_enable   : IN  std_logic;
          enb          : IN  std_logic;
          In1          : IN  vector_of_std_logic_vector8(0 TO 3);  -- int8 [4]
          Out1         : OUT vector_of_std_logic_vector9(0 TO 3)  -- sfix9 [4]
        );
  END COMPONENT;
```

```

        reset      : IN    std_logic;
        In1       : IN    std_logic_vector(15 DOWNT0 0) -- int16
    );
END COMPONENT;

COMPONENT sink_bbx
    PORT( clk          : IN    std_logic;
          clk_enable   : IN    std_logic;
          reset        : IN    std_logic;
          In1         : IN    std_logic_vector(7 DOWNT0 0) -- int8
    );
END COMPONENT;

COMPONENT bbx_with_connected_out
    PORT( clk          : IN    std_logic;
          clk_enable   : IN    std_logic;
          reset        : IN    std_logic;
          Output       : OUT   std_logic_vector(16 DOWNT0 0) -- ufix17
    );
END COMPONENT;

...
END rtl;

```

Redundant Logic in Subsystem Blocks

Subsystem blocks are preserved in the generated HDL code if the blocks have at least one output port that contributes to the evaluation of a DUT output. The Subsystem blocks are also preserved if they contain a black box subsystem that is valid. A black box subsystem is valid if it contains an input port, or an output port that is connected, as described above.

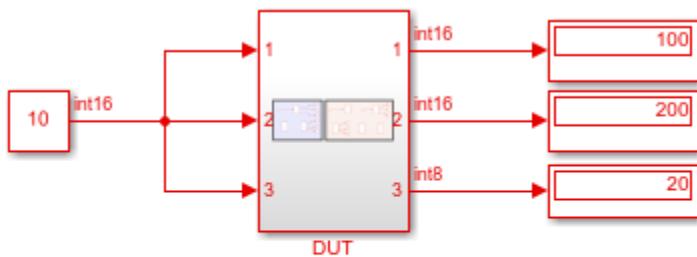
The different kinds of subsystem blocks follow this convention.

- Subsystem
- Atomic Subsystem
- Model References
- Variant Subsystem
- Foreach Subsystem
- Triggered Subsystem
- Enabled Subsystem
- Synchronous subsystems
- Masked subsystems

For individual subsystem ports, the removal of redundant logic also varies depending on whether you specify the **Remove Unused Ports** setting in the Configuration Parameters dialog box. See “Remove Unused Ports” on page 15-6.

Open the model `hdlcoder_subsys_redundant_logic`.

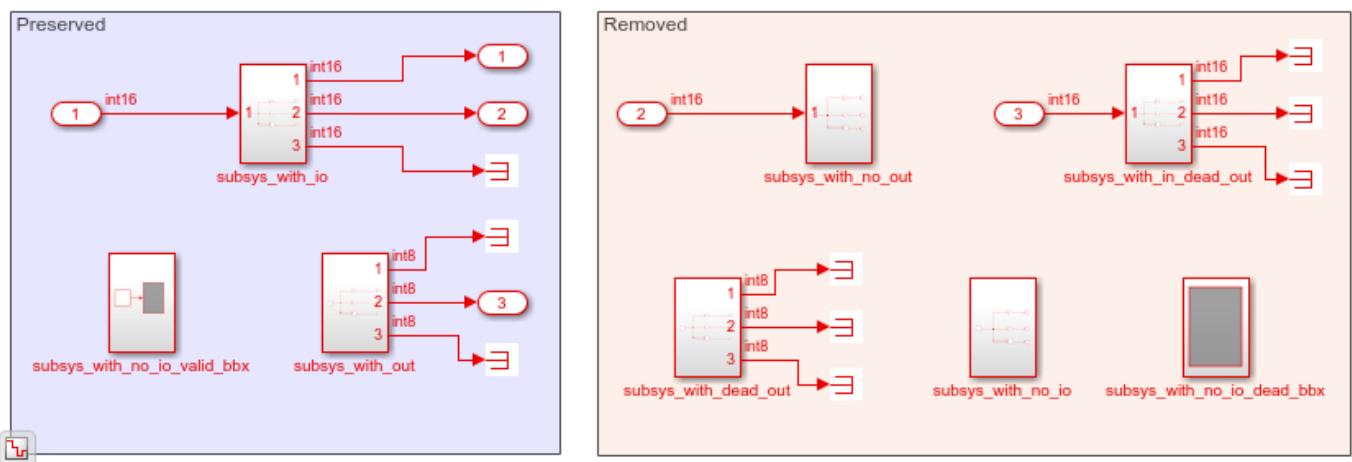
```
open_system('hdlcoder_subsys_redundant_logic')
sim('hdlcoder_subsys_redundant_logic');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains subsystem blocks inside boxes labelled **Preserved** and **Removed**. Subsystem blocks inside **Preserved** are not removed during code generation because they have at least one output port that contribute to the evaluation of the DUT output, or have a valid black box subsystem. The other black box subsystems that do not have an active output port are removed during code generation.

```
open_system('hdlcoder_subsys_redundant_logic/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_subsys_redundant_logic/DUT')
```

The generated HDL code shows the subsystems in the **Preserved** section.

ARCHITECTURE rtl OF DUT IS

```
-- Component Declarations
COMPONENT subsys_with_io
  PORT( In1      : IN  std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     : OUT std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     : OUT std_logic_vector(15 DOWNTO 0)   -- int16
      );
END COMPONENT;
```

```

COMPONENT subsys_with_out
  PORT( Out2      : OUT    std_logic_vector(7 DOWNTO 0)  -- int8
        );
END COMPONENT;

COMPONENT subsys_with_no_io_valid_bbx
  PORT( clk       : IN     std_logic;
        reset     : IN     std_logic;
        enb      : IN     std_logic
        );
END COMPONENT;

...
END rtl;

```

Redundant Logic in Subsystem Ports

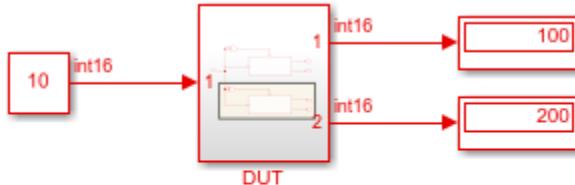
For subsystem data ports, the removal of redundant logic also depends on whether you specify the **Remove Unused Ports** setting. See “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188.

Control ports are not affected by the **Remove Unused Ports** setting. The control ports and components that contribute to evaluation of the control ports are preserved in the generated HDL code only if the entire subsystem instance is considered active.

```

open_system('hdlcoder_control_redundant_logic')
sim('hdlcoder_control_redundant_logic');

```



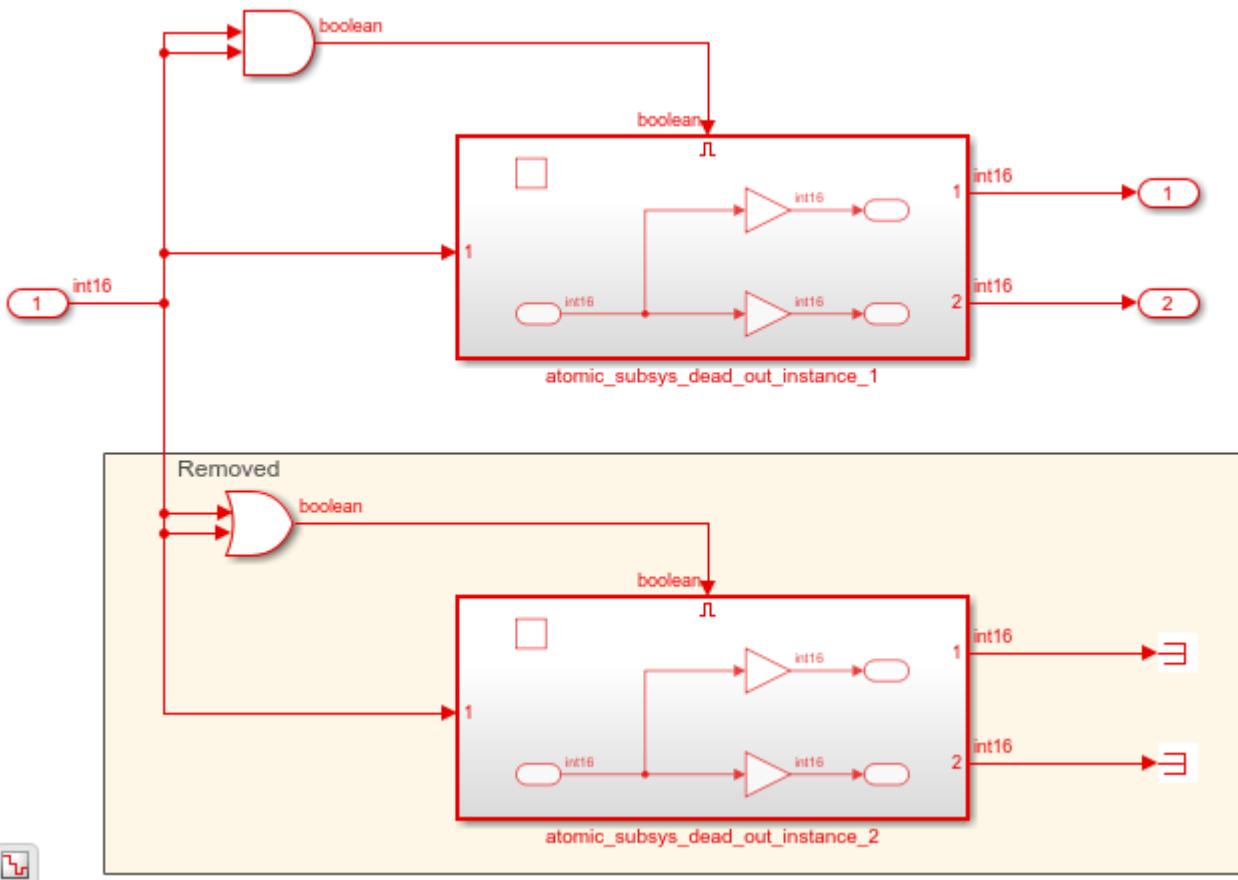
Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems with an input that drives the Enable port. The subsystem `atomic_subsys_dead_out_instance_2` is not active as the outputs are terminated.

```

open_system('hdlcoder_control_redundant_logic/DUT')

```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_control_redundant_logic/DUT')
```

The `atomic_subsys_dead_out_instance_2` including the control port and input signal is removed in the generated HDL code.

```
ENTITY DUT IS
  PORT( clk : IN std_logic;
        reset : IN std_logic;
        clk_enable : IN std_logic;
        In1 : IN std_logic_vector(15 DOWNTO 0); -- int16
        ce_out : OUT std_logic;
        Out1 : OUT std_logic_vector(15 DOWNTO 0); -- int16
        Out2 : OUT std_logic_vector(15 DOWNTO 0) -- int16
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
  PORT( clk : IN std_logic;
        reset : IN std_logic;
        enb : IN std_logic;
        In1 : IN std_logic_vector(15 DOWNTO 0); -- int16
      );
END COMPONENT;

BEGIN
  ce_out <= In1;
  Out1 <= atomic_subsys_dead_out_instance_1(clk, reset, enb, In1);
  Out2 <= atomic_subsys_dead_out_instance_1(clk, reset, enb, In1);
END;
```

```

    Enable      : IN    std_logic;
    Out1       : OUT   std_logic_vector(15 DOWNTO 0); -- int16
    Out2       : OUT   std_logic_vector(15 DOWNTO 0); -- int16
  );
END COMPONENT;

...
END rtl;

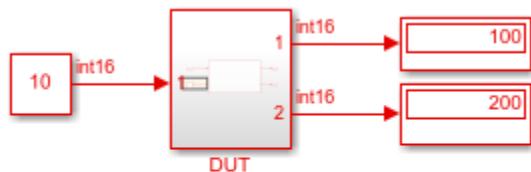
```

Redundant Logic in Atomic Subsystems and Model References

Redundant logic in atomic subsystems, model references, and Foreach Subsystem blocks are treated in the same manner during HDL code generation. Redundant logic at the boundary of atomic subsystems are removed during HDL code generation.

Open the model `hdlcoder_atomic_subsys2_redundant`.

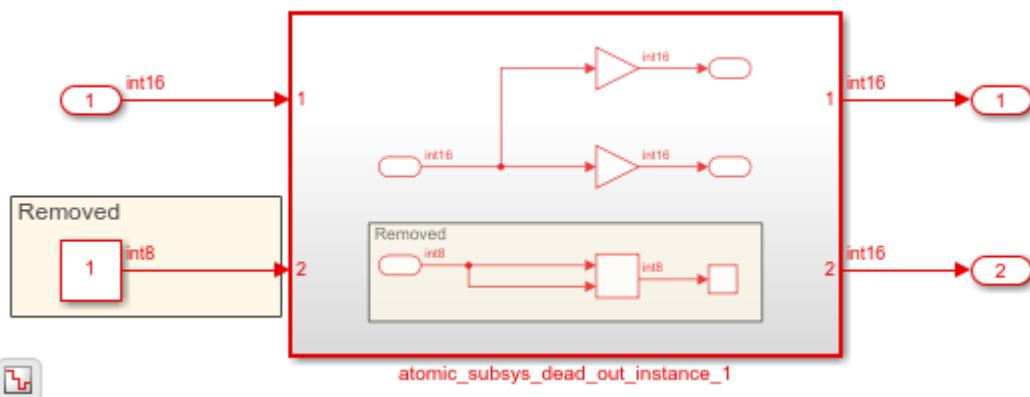
```
open_system('hdlcoder_atomic_subsys2_redundant')
sim('hdlcoder_atomic_subsys2_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains a single Atomic Subsystem block. The Constant block is input to the subsystem that has an Add block connected to a Terminator block.

```
open_system('hdlcoder_atomic_subsys2_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys2_redundant/DUT')
```

When you generate HDL code, the Add block and the input port are removed because the blocks do not contribute to the evaluation of a DUT outport.

```
ENTITY atomic_subsys_dead_out_instance_1 IS
  PORT( In1      : IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     : OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     : OUT   std_logic_vector(15 DOWNTO 0)  -- int16
      );
END atomic_subsys_dead_out_instance_1;

ARCHITECTURE rtl OF atomic_subsys_dead_out_instance_1 IS
  ...
  In1_signed <= signed(In1);

  Gain_mul_temp <= to_signed(16#000A#, 16) * In1_signed;
  Gain_out1 <= Gain_mul_temp(15 DOWNTO 0);

  Out1 <= std_logic_vector(Gain_out1);

  Gain1_mul_temp <= to_signed(16#0014#, 16) * In1_signed;
  Gain1_out1 <= Gain1_mul_temp(15 DOWNTO 0);

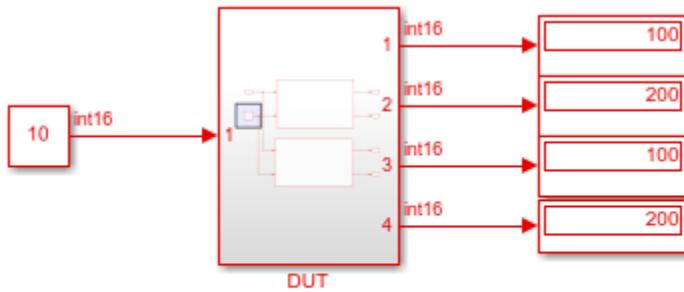
  Out2 <= std_logic_vector(Gain1_out1);

END rtl;
```

If there are more than one active instances of the Atomic Subsystem blocks, the redundant logic computation does not cross the subsystem boundary, and the blocks are preserved in the generated HDL code.

Open the model `hdlcoder_atomic_subsys1_redundant`.

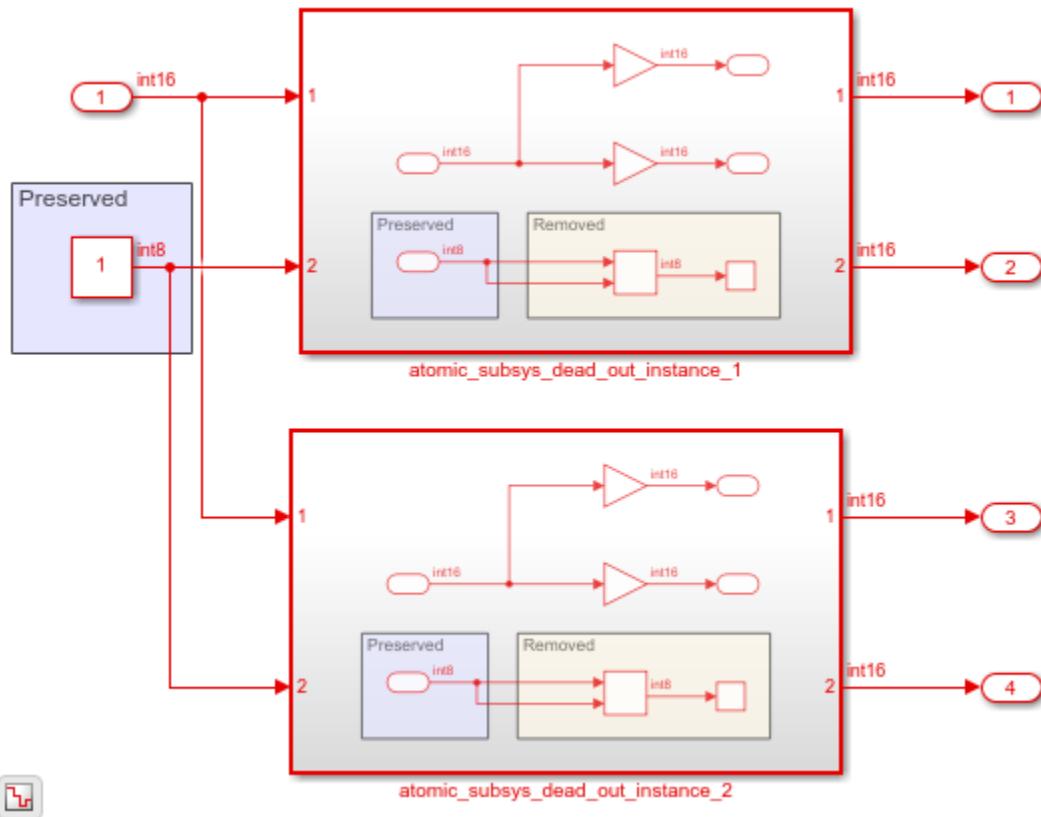
```
open_system('hdlcoder_atomic_subsys1_redundant')
sim('hdlcoder_atomic_subsys1_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems. Inside these subsystems, an Add block is connected to a Terminator block.

```
open_system('hdlcoder_atomic_subsys1_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys1_redundant/DUT')
```

When you generate HDL code, the Add block is removed because it does not contribute to the evaluation of a DUT output. As there are two atomic subsystem instances that are active, the input port to the Add block is preserved during HDL code generation.

A single HDL file `atomic_subsys_dead_out_instance_1` is generated for the atomic subsystems. This file contains the In2 port declaration but is unused in the HDL code. At the DUT level, `DUT.vhd`, the Constant block is preserved though it is feeding into an input port that does not drive any component inside the Atomic Subsystem.

```
ENTITY atomic_subsys_dead_out_instance_1 IS
  PORT( In1      : IN  std_logic_vector(15 DOWNTO 0);  -- int16
        In2      : IN  std_logic_vector(7 DOWNTO 0);  -- int8
        Out1     : OUT std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     : OUT std_logic_vector(15 DOWNTO 0)  -- int16
      );
END atomic_subsys_dead_out_instance_1;

ARCHITECTURE rtl OF atomic_subsys_dead_out_instance_1 IS
  -- Signals
  SIGNAL In1_signed      : signed(15 DOWNTO 0);  -- int16
  SIGNAL Gain_mul_temp   : signed(31 DOWNTO 0);  -- sfix32
  SIGNAL Gain_out1       : signed(15 DOWNTO 0);  -- int16
```

```

SIGNAL Gain1_mul_temp    : signed(31 DOWNTO 0); -- sfix32
SIGNAL Gain1_outl       : signed(15 DOWNTO 0); -- int16

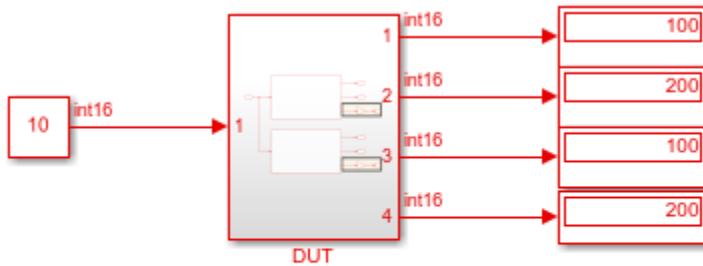
...
END rtl;

```

When you have multiple instances of Atomic Subsystem blocks that are active, these instances are preserved in the generated code.

Open the model `hdlcoder_atomic_subsys1_ports_redundant`.

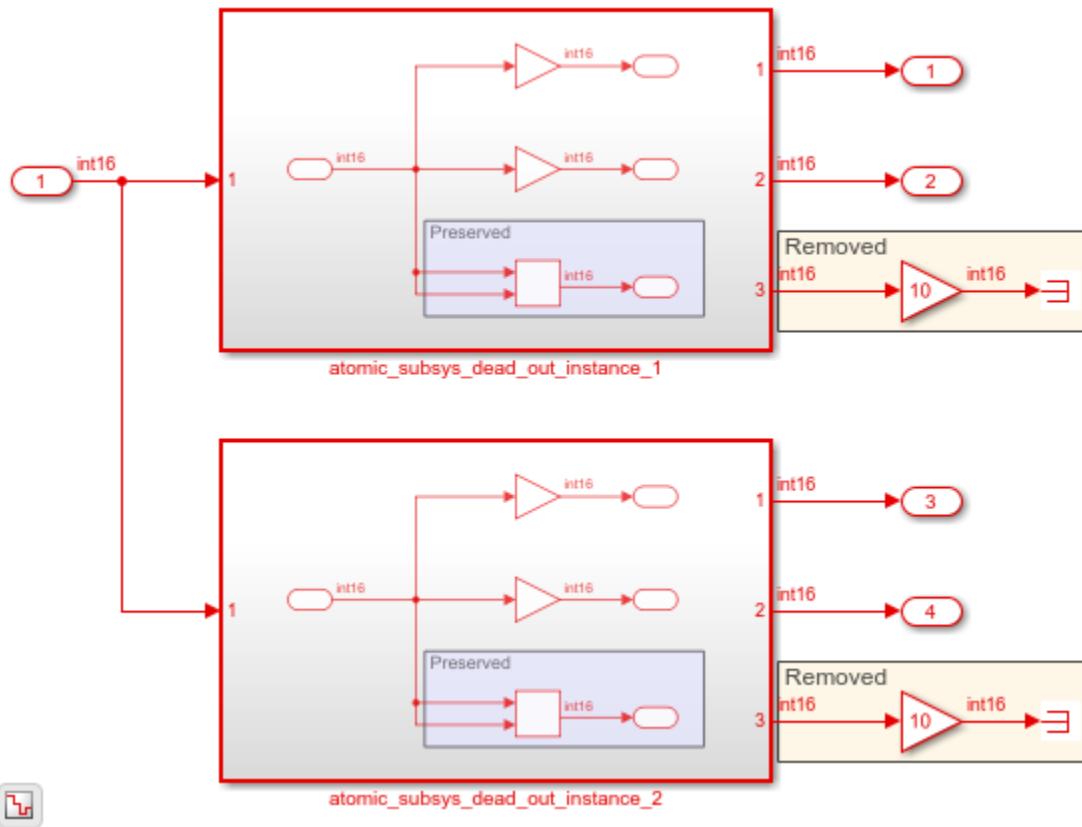
```
open_system('hdlcoder_atomic_subsys1_ports_redundant')
sim('hdlcoder_atomic_subsys1_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems that are active.

```
open_system('hdlcoder_atomic_subsys1_ports_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys1_ports_redundant/DUT')
```

The atomic subsystems are preserved in the generated HDL code but the Gain block calculation is removed from the code. In this case, the multiple atomic subsystem instances are active and thus the redundant logic is not removed across the port boundary.

```

ENTITY DUT IS
  PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out3     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out4     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

  -- Component Declarations
  COMPONENT atomic_subsys_dead_out_instance_1
    PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out3     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
  END COMPONENT;

```

```

...
Out1 <= atomic_subsys_dead_out_instance_1_out1;
Out2 <= atomic_subsys_dead_out_instance_1_out2;
Out3 <= atomic_subsys_dead_out_instance_2_out1;
Out4 <= atomic_subsys_dead_out_instance_2_out2;
END rtl;

```

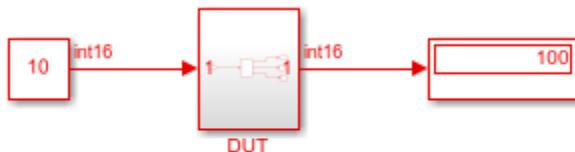
When at least one instance of Atomic Subsystem block is not active and when there are unused output ports outside the Atomic Subsystem blocks, the generated HDL code varies depending on the **Remove Unused Porta** setting. See “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188.

Redundant Logic in Masked Subsystems

Redundant logic in masked subsystems are removed in the same manner as for regular Subsystem blocks. When generating mask parameters as generics in the HDL code by setting **MaskParameterAsGeneric** to on, the generic ports are preserved in the code.

Open the model `hdlcoder_masked_subsys_redundant`.

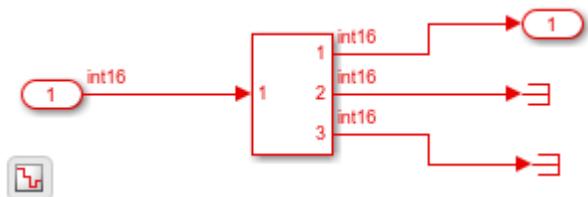
```
open_system('hdlcoder_masked_subsys_redundant')
sim('hdlcoder_masked_subsys_redundant');
```



Copyright 2020 The MathWorks, Inc.

The model contains a masked subsystem that has two mask parameters `Gain` and `Gain1`. `Gain` has the value 10 and `Gain1` has the value 20.

```
open_system('hdlcoder_masked_subsys_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_masked_subsys_redundant/DUT')
```

The generated code shows that generic ports `Gain` and `Gain1` are preserved but the output port `Out2` is removed.

```

ENTITY Subsystem IS
  GENERIC( Gain      : integer := 10;
            Gain1     : integer := 20
          );
  PORT( In1       : IN    std_logic_vector(15 DOWNTO 0); -- int16
        Out1      : OUT   std_logic_vector(15 DOWNTO 0) -- int16
      );
END Subsystem;

ARCHITECTURE rtl OF Subsystem IS
  ...
  kconst <= to_signed(Gain, 16);
  In1_signed <= signed(In1);
  Gain_mul_temp <= kconst * In1_signed;
  Gain_out1 <= Gain_mul_temp(15 DOWNTO 0);
  Out1 <= std_logic_vector(Gain_out1);
END rtl;

```

Redundant Logic in DocBlock and Annotations

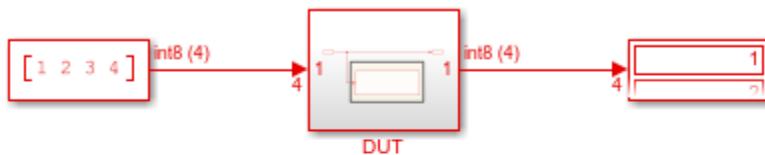
Annotations or DocBlock blocks that are inside active subsystems are preserved in the generated HDL code. A subsystem is active if it contains at least one output port that leads to the evaluation of the DUT output, or has an active black box subsystem, as described above. If HDL Coder determines that a subsystem containing an annotation or DocBlock is redundant, then that annotation or DocBlock is also removed from the generated code.

Open the model `hdlcoder_annotation_redundant_logic`.

```

open_system('hdlcoder_annotation_redundant_logic')
sim('hdlcoder_annotation_redundant_logic');

```



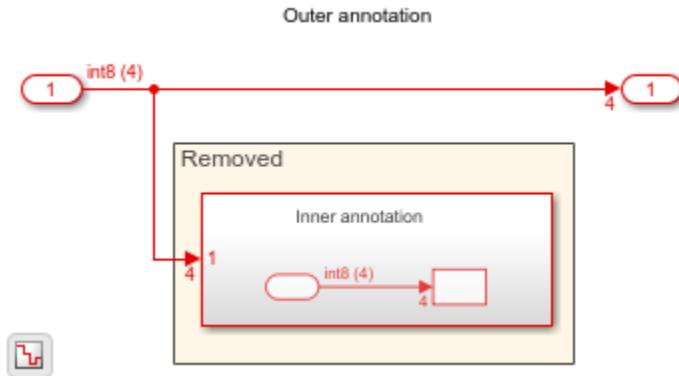
Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an Inner annotation subsystem. This subsystem is redundant because it does not have an active output port.

```

open_system('hdlcoder_annotation_redundant_logic/DUT')

```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_annotation_redundant_logic/DUT')
```

The generated HDL code shows the `Inner annotation` subsystem including the annotation `Inner annotation` removed from the generated HDL code.

```
ENTITY DUT IS
  PORT( In1    :  IN   vector_of_std_logic_vector8(0 TO 3);  -- int8 [4]
        Out2   :  OUT  vector_of_std_logic_vector8(0 TO 3)  -- int8 [4]
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

BEGIN
  -- Removed
  --
  -- Outer annotation

  Out2 <= In1;

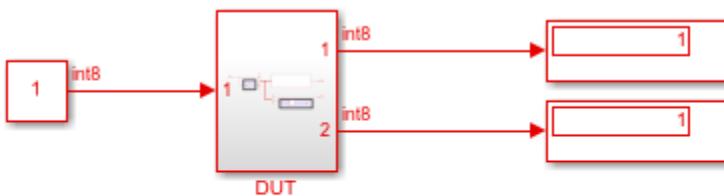
END rtl;
```

Redundant Logic in Bus Signals and Bus Element Ports

Redundant logic and unused blocks are removed from the model when it contains buses. The redundant logic optimization applies in the same manner to virtual buses, nonvirtual buses, and bus element ports.

Open the model `hdlcoder_virtual_bus_redundant`.

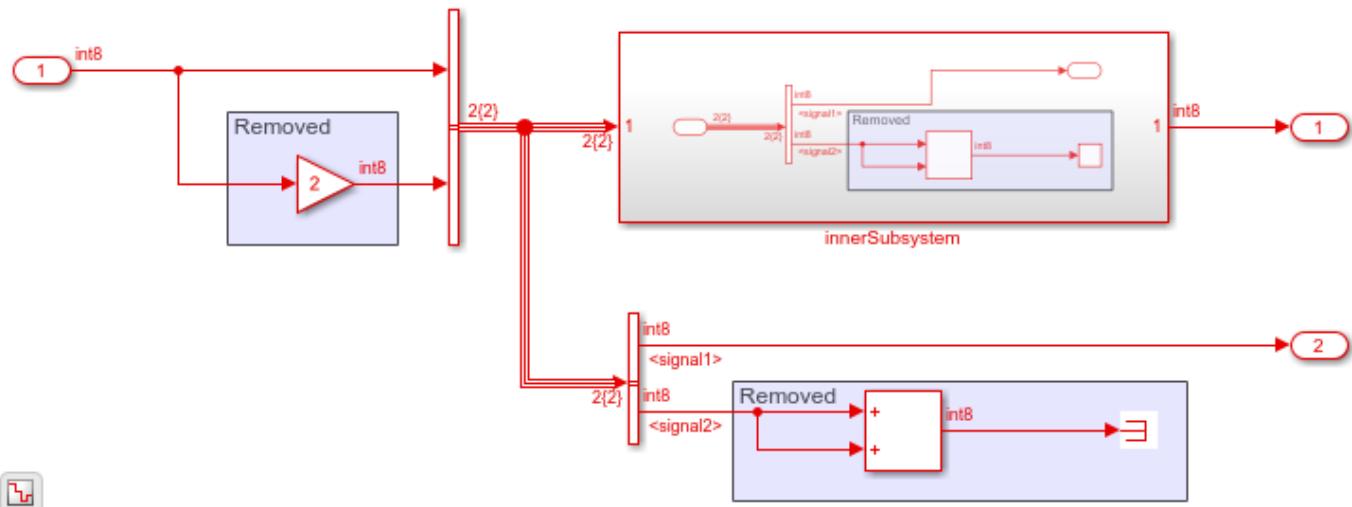
```
open_system('hdlcoder_virtual_bus_redundant')
sim('hdlcoder_virtual_bus_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains a virtual bus that drives an `innerSubsystem` block and an Add block. One of the bus signals is connected to Terminator blocks at the output.

```
open_system('hdlcoder_virtual_bus_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_virtual_bus_redundant/DUT')
```

The generated HDL code shows that the redundant logic is removed in the design and the input port and output ports that are active are preserved.

```

ENTITY DUT IS
  PORT(
    In1 : IN std_logic_vector(7 DOWNTO 0); -- int8
    Out1 : OUT std_logic_vector(7 DOWNTO 0); -- int8
    Out2 : OUT std_logic_vector(7 DOWNTO 0) -- int8
  );
END DUT;

ARCHITECTURE rtl OF DUT IS
  -- Component Declarations
  COMPONENT innerS
    PORT(
      In1_signal1 : IN std_logic_vector(7 DOWNTO 0); -- int8
      Out1 : OUT std_logic_vector(7 DOWNTO 0) -- int8
    );
  END COMPONENT;
  SIGNAL signal1 : std_logic_vector(7 DOWNTO 0);
  SIGNAL signal2 : std_logic_vector(7 DOWNTO 0);
BEGIN
  In1 <=> signal1;
  signal1 <=> innerS.In1_signal1;
  signal1 <=> signal2;
  signal2 <=> innerS.Out1;
  signal2 <=> adder.Add1;
  signal2 <=> adder.Add2;
  process(signal1, signal2)
    begin
      if signal1 = "11111111" then
        signal2 <=> adder.Add1;
      else
        signal2 <=> adder.Add2;
      end if;
    end process;
  adder : ADDER
    generic map(MSB => 7, LSB => 0)
    port map(
      Add1 <=> signal2,
      Add2 <=> signal2
    );
  end;

```

```

);
END COMPONENT;

...
END rtl;

```

Inside the `innerSubsystem` block, the Add block connected to the Terminator block is removed.

```

ENTITY innerSubsystem IS
  PORT( In1_signal1           : IN    std_logic_vector(7 DOWNTO 0);  -- int8
        Out1                : OUT   std_logic_vector(7 DOWNTO 0)  -- int8
        );
END innerSubsystem;

ARCHITECTURE rtl OF innerSubsystem IS
  -- Signals
  SIGNAL signal1             : signed(7 DOWNTO 0);  -- int8

BEGIN
  -- Removed
  signal1 <= signed(In1_signal1);
  Out1 <= std_logic_vector(signal1);
END rtl;

```

Limitations

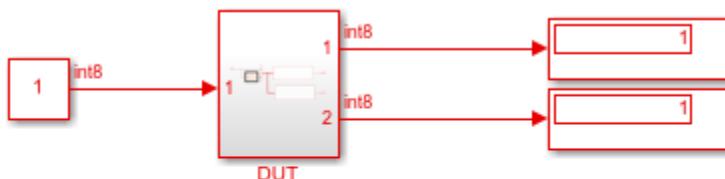
- When your model contains multiple instances of atomic subsystems, model references, or Foreach Subsystem blocks, if these blocks are determined to be active during HDL code generation, then all ports are preserved in the generated code. Components connected upstream to these ports are also considered active. The ports are preserved independent of whether you enable or disable the `DeleteUnusedPorts` setting. This limitation also applies when you use bus signals. In the case of bus signals, when there are multiple instances of atomic subsystems or model references that are considered active, the entire bus is preserved.

Open the model `hdlcoder_atomic_bus_virtual_redundant`.

```

open_system('hdlcoder_atomic_virtual_bus_redundant')
sim('hdlcoder_atomic_virtual_bus_redundant');

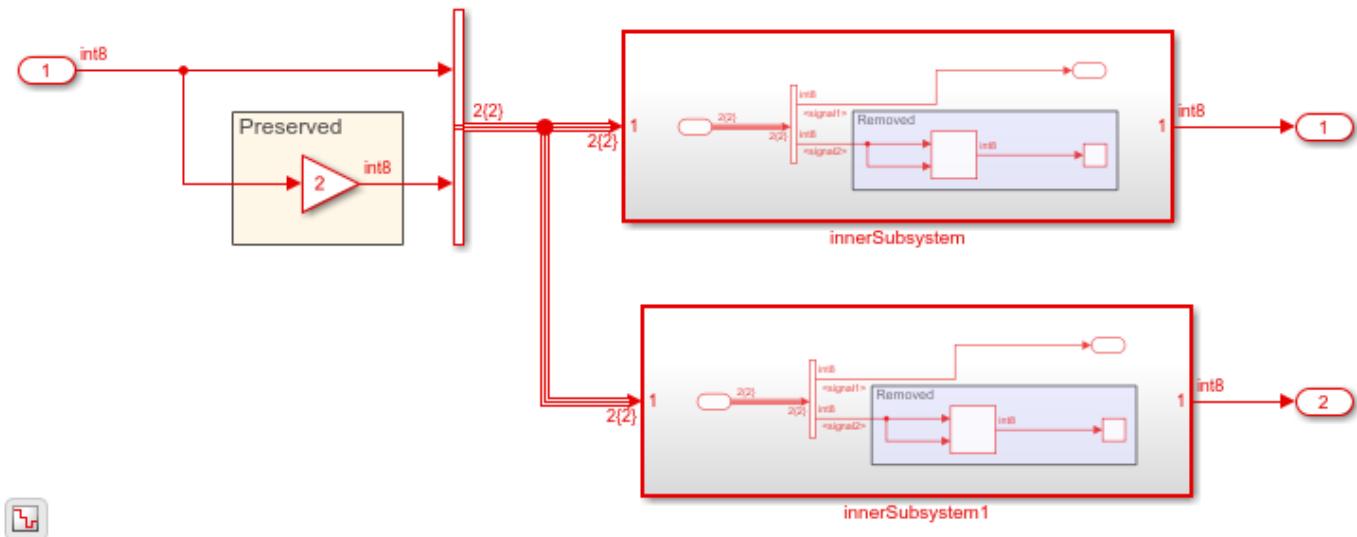
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains two atomic subsystems, `innerSubsystem` and `innerSubsystem` block. Both atomic subsystem instances are active.

```
open_system('hdlcoder_atomic_virtual_bus_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_virtual_bus_redundant/DUT')
```

The generated HDL code shows that the Gain block that drives the atomic subsystem blocks is preserved.

```
ENTITY DUT IS
  PORT( In1                      : IN   std_logic_vector(7 DOWNTO 0);  -- int8
        Out1                     : OUT  std_logic_vector(7 DOWNTO 0);  -- int8
        Out2                     : OUT  std_logic_vector(7 DOWNTO 0)  -- int8
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

  -- Component Declarations
  COMPONENT innerSubsystem
    PORT( In1_signal1           : IN   std_logic_vector(7 DOWNTO 0);  -- int8
          In1_signal2           : IN   std_logic_vector(7 DOWNTO 0);  -- int8
          Out1                   : OUT  std_logic_vector(7 DOWNTO 0)  -- int8
        );
  END COMPONENT;

  ...
  u_innerSubsystem : innerSubsystem
  PORT MAP( In1_signal1 => In1,  -- int8
            In1_signal2 => std_logic_vector(Gain_out1),  -- int8
            Out1 => innerSubsystem_out1  -- int8
          );

  u_innerSubsystem1 : innerSubsystem
  PORT MAP( In1_signal1 => In1,  -- int8
            In1_signal2 => std_logic_vector(Gain_out1),  -- int8
            Out1 => innerSubsystem1_out1  -- int8
          );
```

```

In1_signed <= signed(In1);
Gain_cast <= resize(In1_signed & '0', 16);
Gain_out1 <= Gain_cast(7 DOWNTO 0);

Out1 <= innerSubsystem_out1;
Out2 <= innerSubsystem1_out1;

END rtl;

```

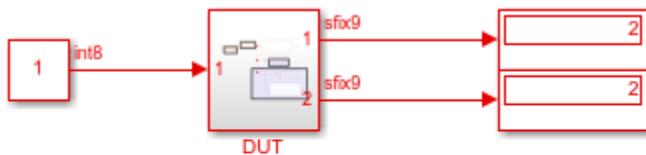
2. For models that have vector signals, Demux blocks act as boundaries for the redundant logic optimization. Redundant logic and components that are downstream of the Demux block are removed during HDL code generation. The optimization does not cross the Demux block boundary and therefore preserves components that are upstream of the Demux block. This limitation when using vectors also applies when you convert buses to vectors.

Open the model `hdlcoder_vector_redundant_logic`.

```

open_system('hdlcoder_vector_redundant_logic')
sim('hdlcoder_vector_redundant_logic');

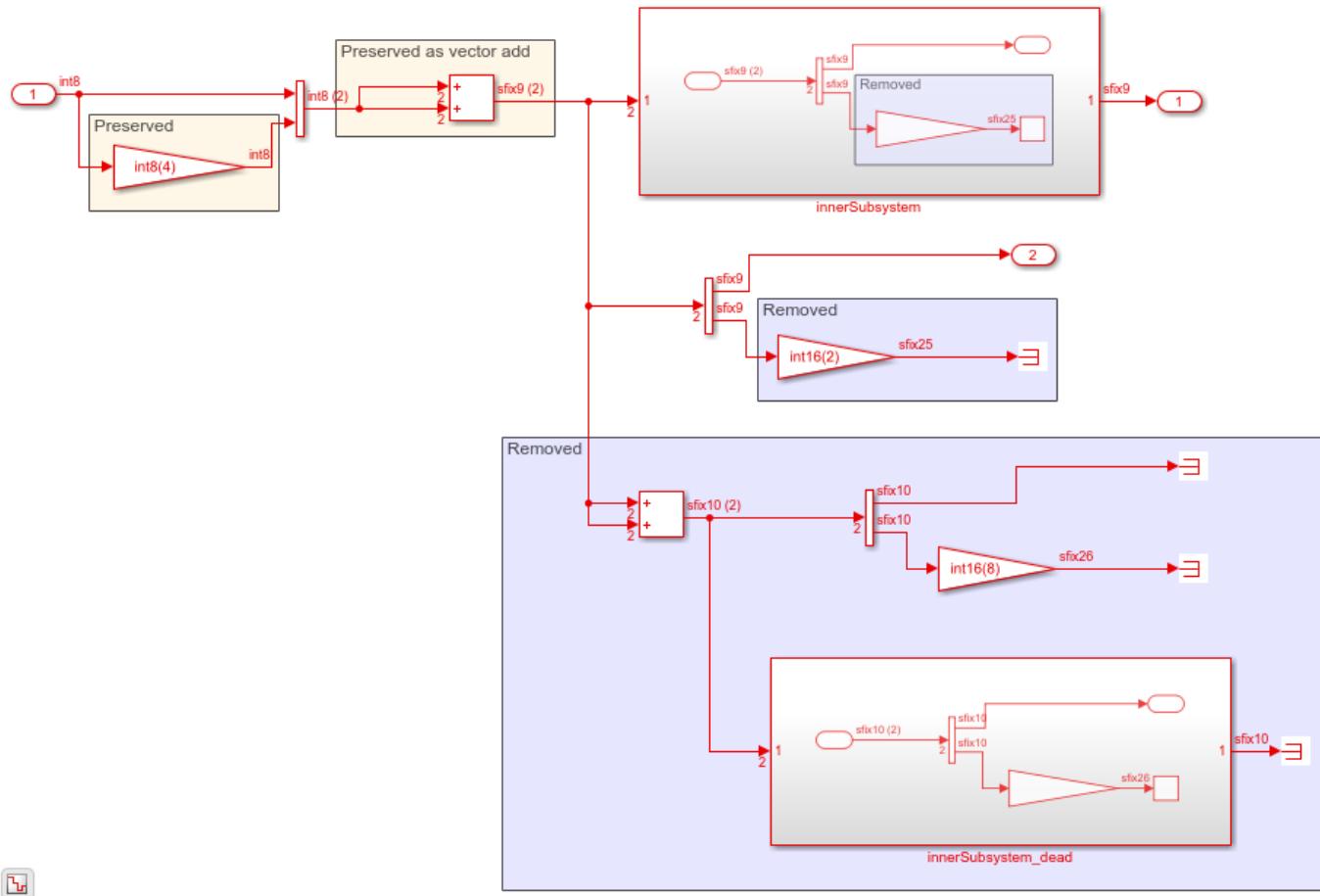
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an `innerSubsystem` block. This subsystem has one active output port and the other port is terminated.

```
open_system('hdlcoder_vector_redundant_logic/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_vector_redundant_logic/DUT')
```

The generated HDL code shows that the Add block performs a vector Add calculation. Both the Gain block and Add block are preserved in the generated HDL code as the redundant logic optimization does not cross the Demux block boundary.

```

ENTITY DUT IS
  PORT( In1      : IN    std_logic_vector(7 DOWNTO 0);  -- int8
        Out1     : OUT   std_logic_vector(8 DOWNTO 0);  -- sfix9
        Out2     : OUT   std_logic_vector(8 DOWNTO 0)  -- sfix9
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT innerSubsystem
  PORT( In1      : IN    vector_of_std_logic_vector9(0 TO 1);  -- sfix9 [2]
        Out1     : OUT   std_logic_vector(8 DOWNTO 0)  -- sfix9
      );
END COMPONENT;

...

```

```
In1_signed <= signed(In1);

Gain1_cast <= resize(In1_signed & '0' & '0', 16);
Gain1_out1 <= Gain1_cast(7 DOWNTO 0);

Mux_out1(0) <= signed(In1);
Mux_out1(1) <= Gain1_out1;

Add_out1_gen: FOR t_0 IN 0 TO 1 GENERATE
    Add_out1(t_0) <= resize(Mux_out1(t_0), 9) + resize(Mux_out1(t_0), 9);
END GENERATE Add_out1_gen;

outputgen: FOR k IN 0 TO 1 GENERATE
    Add_out1_1(k) <= std_logic_vector(Add_out1(k));
END GENERATE;

Out2 <= std_logic_vector(Add_out1(0));

...
END rtl;
```

See Also

More About

- “Generated Model and Validation Model” on page 24-10
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-17
- “Optimize Unconnected Ports in Generated HDL Code for Simulink Models” on page 24-188

Optimize Unconnected Ports in Generated HDL Code for Simulink Models

During HDL code generation, the unconnected ports from the generated code are removed without removing ports from top-level DUT models or subsystems. This optimization includes removing unconnected vector and scalar ports, bus element ports, and bus ports. Removing unconnected ports improves the readability of the generated VHDL or Verilog code and reduces code size and area usage. The reduction avoids synthesis failure caused by unused ports in the generated code.

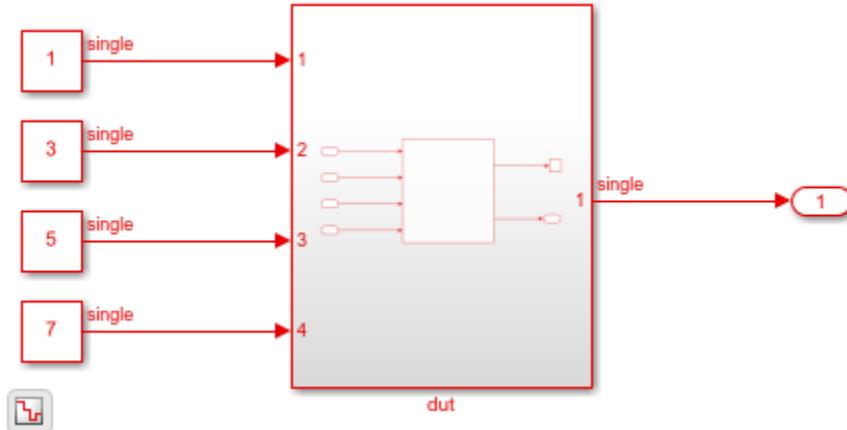
How Unused Port Deletion Works

Unused port deletion works in conjunction with removal of unused blocks in your design. To learn how unused blocks are removed, see “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166.

See the generated HDL code to observe the effect of this optimization. The generated model and the validation model do not show the effect of this optimization. These models contain the unused ports that are removed from the HDL code.

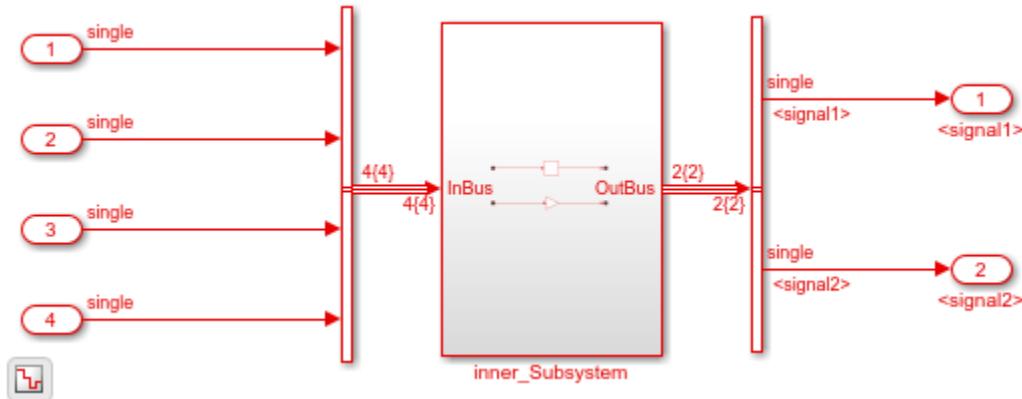
Open the model `hdlcoder_RemoveUnconnectedPorts` containing Bus Element ports and a port connected to an inactive output.

```
open_system('hdlcoder_RemoveUnconnectedPorts')
set_param('hdlcoder_RemoveUnconnectedPorts', 'SimulationCommand', 'update');
```



Open the `dut` Subsystem block, and then open the `mid_Subsystem` block. The `mid_Subsystem` contains the Bus Element ports. One of the output signals is connected to a Terminator block.

```
open_system('hdlcoder_RemoveUnconnectedPorts/dut/mid_Subsystem')
```



To generate HDL code for the design, at the MATLAB® command prompt, enter:

```
makehdl('hdlcoder_RemoveUnconnectedPorts/dut')
```

The generated code `mid_Subsystem.vhd` shows that unconnected ports are removed during HDL code generation. The input `InBus_signal3` at the DUT input port is multiplied by a Gain block and then connected to the output port `OutBus_signal2`, which is then passed to the DUT output port. Since the other input and output ports are unused at the DUT level, these ports are removed from the generated HDL code.

```
ARCHITECTURE rtl OF mid_Subsystem IS

-- Component Declarations
COMPONENT inner_Subsystem
  PORT( clk           :  IN    std_logic;
        reset         :  IN    std_logic;
        enb          :  IN    std_logic;
        InBus_signal3 :  IN    std_logic_vector(31 DOWNTO 0);  -- single
        OutBus_signal2 : OUT   std_logic_vector(31 DOWNTO 0)  -- single
      );
END COMPONENT;

...
END rtl;
```

Disable Unused Port Deletion Optimization

By default, the optimization is enabled and unused ports are removed in the generated HDL code.

If you do not want unconnected ports to be removed from the design:

- In the Configuration Parameters dialog box, clear the “Remove Unused Ports” on page 15-6 check box.
- When you run the HDL Workflow Advisor, in the **Set Code Generation Options > Set Optimization Options** task, clear the **Remove Unused Ports** check box.
- At the command line, set `DeleteUnusedPorts` to off with `hdlset_param` or `makehdl`. For example, to specify that you want to preserve the unused ports in the `hdlcoder_RemoveUnconnectedPorts` model, run this command:

```
makehdl('hdlcoder_RemoveUnconnectedPorts/dut', 'DeleteUnusedPorts', 'off')
```

The generated HDL code preserves the unused Bus Elements ports.

```

ARCHITECTURE rtl OF mid_Subsystem IS

-- Component Declarations
COMPONENT inner_Subsystem
  PORT( clk           : IN    std_logic;
        reset         : IN    std_logic;
        enb          : IN    std_logic;
        InBus_signal1 : IN    std_logic_vector(31 DOWNTO 0); -- single
        InBus_signal2 : IN    std_logic_vector(31 DOWNTO 0); -- single
        InBus_signal3 : IN    std_logic_vector(31 DOWNTO 0); -- single
        InBus_signal4 : IN    std_logic_vector(31 DOWNTO 0); -- single
        OutBus_signal1: OUT   std_logic_vector(31 DOWNTO 0); -- single
        OutBus_signal2: OUT   std_logic_vector(31 DOWNTO 0)  -- single
      );
END COMPONENT;

...
END rtl;

```

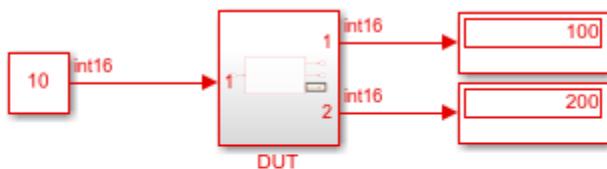
Unused Port Deletion for Subsystem Ports

This optimization can remove unused subsystem data ports. Control ports and ports of a referenced model are not removed.

A subsystem data port is considered to be active when it contributes to a DUT output port downstream, or is connected to an active black box subsystem, or a component that is preserved during HDL code generation.

Open the model `hdlcoder_subsys_ports_unused`.

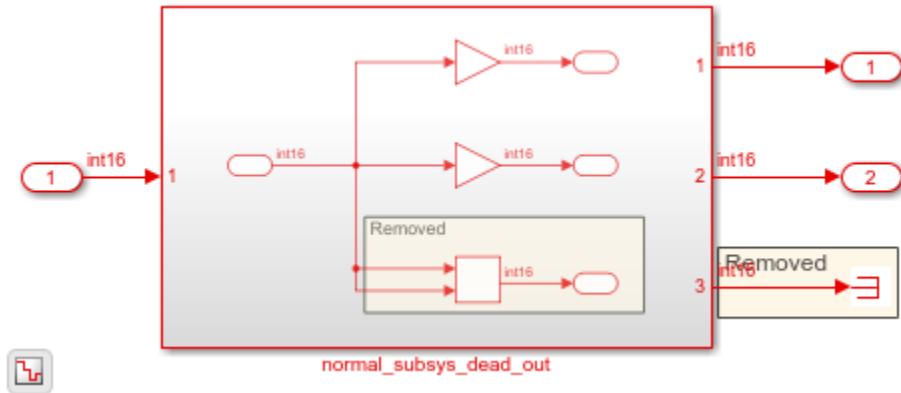
```
open_system('hdlcoder_subsys_ports_unused')
sim('hdlcoder_subsys_ports_unused');
```



Copyright 2020 The MathWorks, Inc.

The model contains a `normal_subsys_dead_out` subsystem that contains an output port `out3` that is terminated and does not contribute to the DUT output.

```
open_system('hdlcoder_subsys_ports_unused/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_subsys_ports_unused/DUT')
```

By default, when `DeleteUnusedPorts` is on, the Add block calculation and output port, `Out3`, are removed in the generated HDL code.

```
ENTITY normal_subsys_dead_out IS
  PORT( In1      : IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     : OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     : OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END normal_subsys_dead_out;

ARCHITECTURE rtl OF normal_subsys_dead_out IS
...
  ...
  Out1 <= std_logic_vector(Gain_out1);
  ...
  Out2 <= std_logic_vector(Gain1_out1);
END rtl;
```

To disable `DeleteUnusedPorts` optimization, run this command:

```
makehdl('hdlcoder_subsys_ports_unused/DUT', 'DeleteUnusedPorts', 'off')
```

When you set `DeleteUnusedPorts` to off, this port and the Add block calculation is preserved in the generated HDL code.

```
ENTITY normal_subsys_dead_out IS
  PORT( In1      : IN    std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     : OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     : OUT   std_logic_vector(15 DOWNTO 0);  -- int16
        Out3     : OUT   std_logic_vector(15 DOWNTO 0)  -- int16
        );
END normal_subsys_dead_out;

ARCHITECTURE rtl OF normal_subsys_dead_out IS
...

```

```

Out1 <= std_logic_vector(Gain_out1);
...
Out2 <= std_logic_vector(Gain1_out1
Add_out1 <= to_signed(16#0000#, 16);
Out3 <= std_logic_vector(Add_out1);
END rtl;

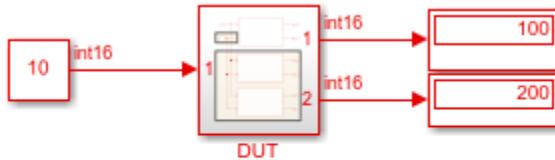
```

Unused Port Deletion for Atomic Subsystems

When you have unused ports outside atomic subsystems, the atomic subsystem instances are removed from the generated HDL code.

Open the model |hdlcoder_atomic_subsys3_redundant|

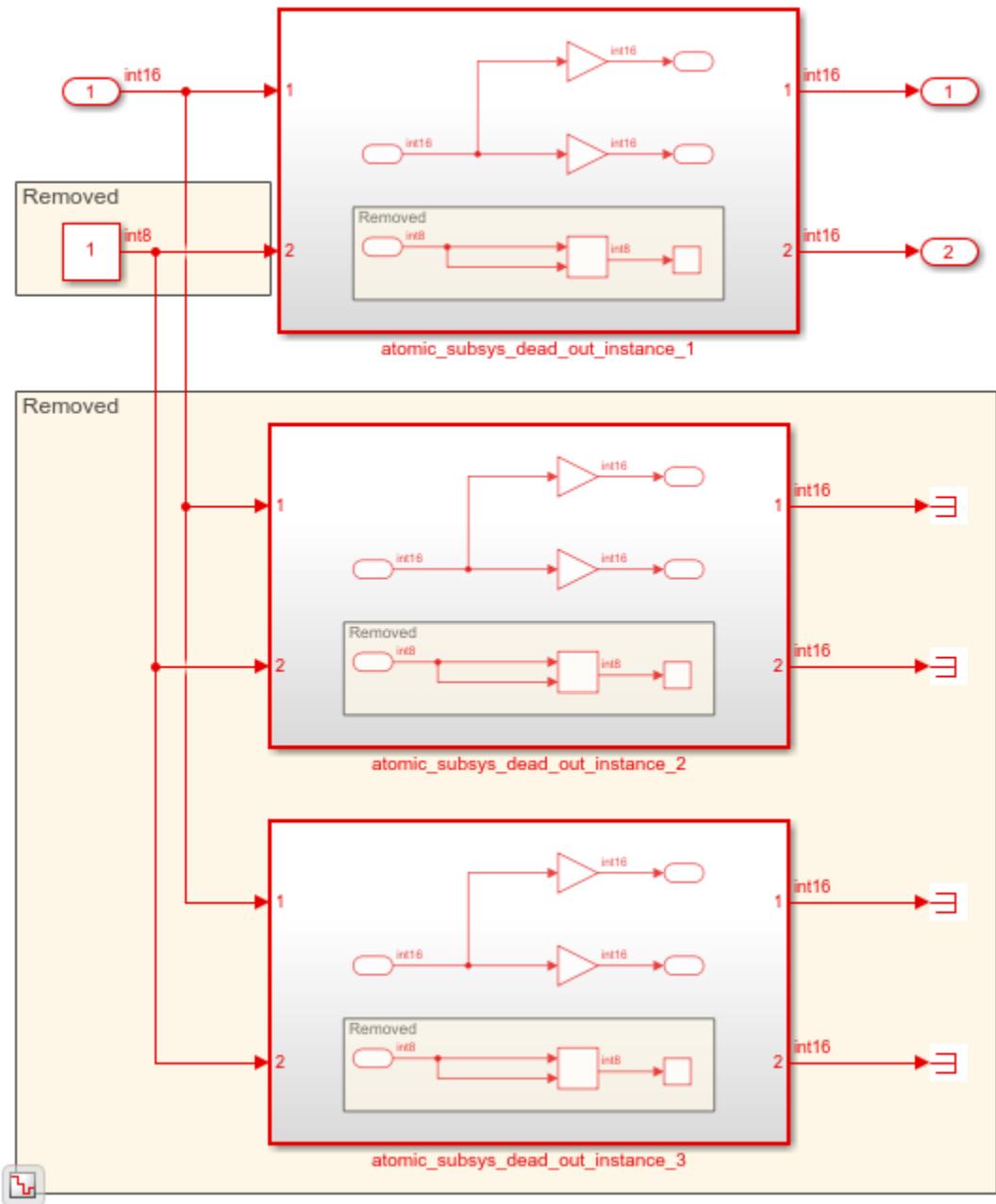
```
open_system('hdlcoder_atomic_subsys3_redundant')
sim('hdlcoder_atomic_subsys3_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains three atomic subsystem instances. The outputs of two of the atomic subsystem instances are connected to Terminator blocks.

```
open_system('hdlcoder_atomic_subsys3_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys3_redundant/DUT')
```

```
ENTITY DUT IS
  PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
      );
END DUT;
```

```
ARCHITECTURE rtl OF DUT IS
```

```
-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
    PORT( In1      : IN      std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     : OUT     std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     : OUT     std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;
...
END rtl;
```

To set `DeleteUnusedPorts` to off, run this command:

```
makehdl('hdlcoder_atomic_subsys3_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

If you set `DeleteUnusedPorta` to off, the input port `In2` is preserved in the generated HDL code but it is feeding into an input port that is driving an unused component.

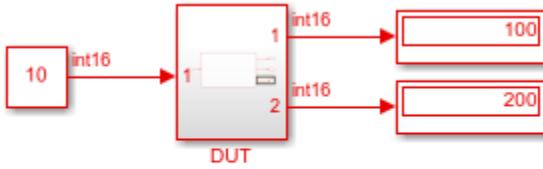
```
ENTITY DUT IS
    PORT( In1      : IN      std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     : OUT     std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     : OUT     std_logic_vector(15 DOWNTO 0)  -- int16
        );
END DUT;
ARCHITECTURE rtl OF DUT IS
-- Component Declarations
COMPONENT atomic_subsys_dead_out_instance_1
    PORT( In1      : IN      std_logic_vector(15 DOWNTO 0);  -- int16
          In2      : IN      std_logic_vector(7 DOWNTO 0);  -- int8
          Out1     : OUT     std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     : OUT     std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;
...
END rtl;
```

Unused Port Deletion for Output Ports of Atomic Subsystems

When you have multiple active atomic subsystem instances, the redundant logic across the boundary including any ports are preserved in the HDL code independent of the `DeleteUnusedPorts` setting. When you have only one Atomic Subsystem instance, by default, when `DeleteUnusedPorts` is on, any redundant logic and ports across the subsystem boundary are removed. However, if you set `DeleteUnusedPorts` to off, any unused port is preserved though the logic is redundant.

Open the model `hdlcoder_atomic_subsys2_ports_redundant`.

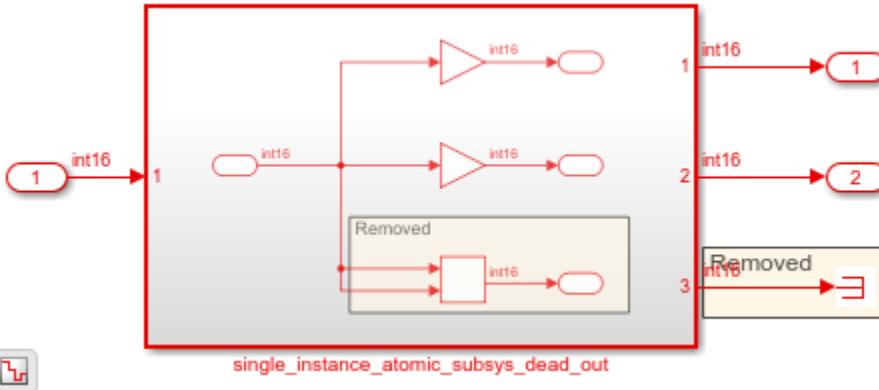
```
open_system('hdlcoder_atomic_subsys2_ports_redundant')
sim('hdlcoder_atomic_subsys2_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an Atomic Subsystem block that contains an Add block connected to an output port terminated outside the subsystem.

```
open_system('hdlcoder_atomic_subsys2_ports_redundant/DUT')
```



To generate HDL code for the DUT subsystem, run this command:

```
makehdl('hdlcoder_atomic_subsys2_ports_redundant/DUT')
```

In the generated HDL code, the Add block and corresponding output port Out3 is removed because it does not contribute to an active output.

```

ENTITY DUT IS
  PORT( In1    :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
      );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT single_instance_atomic_subsys_dead_out
  PORT( In1    :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
      );
END COMPONENT;
...
```

```
END rtl;
```

To set DeleteUnusedPorts to off, run this command:

```
makehdl('hdlcoder_atomic_subsys2_ports_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

If you set DeleteUnusedPorts to off, the output port Out3 is preserved in the generated HDL code.

```
ENTITY DUT IS
  PORT( In1    :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT single_instance_atomic_subsys_dead_out
  PORT( In1    :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        Out3   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

...
END rtl;

ENTITY single_instance_atomic_subsys_dead_out IS
  PORT( In1    :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
        Out1   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out2   :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
        Out3   :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END single_instance_atomic_subsys_dead_out;

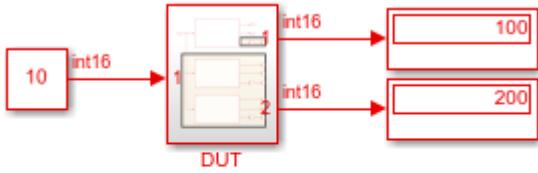
ARCHITECTURE rtl OF single_instance_atomic_subsys_dead_out IS

...
Add_out1 <= to_signed(16#0000#, 16);
Out3 <= std_logic_vector(Add_out1);
END rtl;
```

If there are multiple atomic subsystem instances that are not active, the unused port and logic are removed or preserved depending on the DeleteUnusedPorts setting.

Open the model hdlcoder_atomic_subsys3_ports_redundant.

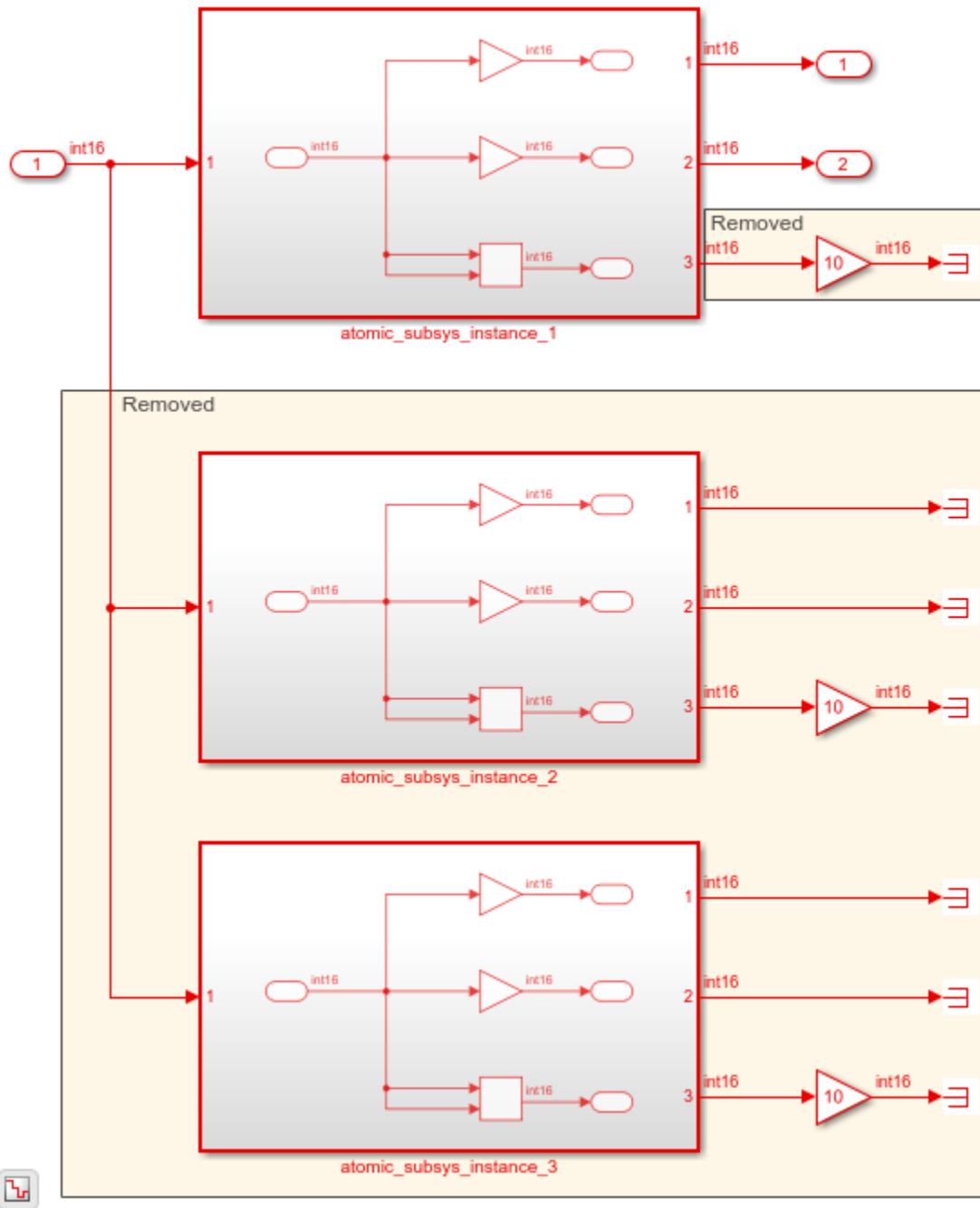
```
open_system('hdlcoder_atomic_subsys3_ports_redundant')
sim('hdlcoder_atomic_subsys3_ports_redundant');
```



Copyright 2020 The MathWorks, Inc.

The DUT subsystem contains an Atomic Subsystem block that contains an Add block connected to an output port terminated outside the subsystem.

```
open_system('hdlcoder_atomic_subsys3_ports_redundant/DUT')
```



The two atomic subsystem instances that have the output ports terminated are removed in the generated HDL code. In the other Atomic Subsystem block, the Add block calculation and its output is removed.

```
ENTITY DUT IS
PORT( In1      : IN  std_logic_vector(15 DOWNTO 0); -- int16
      Out1     : OUT std_logic_vector(15 DOWNTO 0); -- int16
      Out2     : OUT std_logic_vector(15 DOWNTO 0)  -- int16
```

```

        );
END DUT;

ARCHITECTURE rtl OF DUT IS
    -- Component Declarations
COMPONENT atomic_subsys_instance_1
    PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;
    ...
-- Removed
-- Removed
u_atomic_subsys_instance_1 : atomic_subsys_instance_1
PORT MAP( In1 => In1,  -- int16
          Out1 => atomic_subsys_instance_1_out1,  -- int16
          Out2 => atomic_subsys_instance_1_out2  -- int16
        );
Out1 <= atomic_subsys_instance_1_out1;
Out2 <= atomic_subsys_instance_1_out2;
END rtl;

```

To set `DeleteUnusedPorts` to off, run this command:

```
makehdl('hdlcoder_atomic_subsys3_ports_redundant/DUT', 'DeleteUnusedPorts', 'off')
```

If you set `DeleteUnusedPorts` to off, the output port `Out3` is preserved in the generated HDL code.

```

ENTITY DUT IS
    PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END DUT;

ARCHITECTURE rtl OF DUT IS
    -- Component Declarations
COMPONENT atomic_subsys_instance_1
    PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
          Out1     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out2     :  OUT  std_logic_vector(15 DOWNTO 0);  -- int16
          Out3     :  OUT  std_logic_vector(15 DOWNTO 0)  -- int16
        );
END COMPONENT;

```

If you see the generated HDL code for the Atomic Subsystem instance, it shows that the Add block computation and `Out3` is preserved in the generated HDL code.

```
ENTITY atomic_subsys_instance_1 IS
    PORT( In1      :  IN   std_logic_vector(15 DOWNTO 0);  -- int16
```

```
Out1      : OUT  std_logic_vector(15 DOWNTO 0);  -- int16
Out2      : OUT  std_logic_vector(15 DOWNTO 0);  -- int16
Out3      : OUT  std_logic_vector(15 DOWNTO 0)  -- int16
);
END atomic_subsys_instance_1;

ARCHITECTURE rtl OF atomic_subsys_instance_1 IS
...
Add_out1 <= In1_signed + In1_signed;
Out3 <= std_logic_vector(Add_out1);
END rtl;
```

Limitations

When your model contains multiple instances of atomic subsystems, model references, or Foreach Subsystem blocks, if these blocks are determined to be active during HDL code generation, then all ports are preserved in the generated code. Components connected upstream to these ports are also considered active. The ports are preserved independent of whether you enable or disable the `DeleteUnusedPorts` setting.

This limitation also applies to bus signals. In this case, the entire bus is preserved in the generated HDL code. For an example, see “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166.

See Also

More About

- “Generated Model and Validation Model” on page 24-10
- “Remove Redundant Logic and Unused Blocks in Generated HDL Code” on page 24-166
- “Simplify Constant Operations and Reduce Design Complexity in HDL Coder™” on page 24-17

Code Generation Reports, HDL Compatibility Checker, Block Support Library, and Code Annotation

- “Create and Use Code Generation Reports” on page 25-2
- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-4
- “Web View of Model in Code Generation Report” on page 25-10
- “Generate Code with Annotations or Comments” on page 25-13
- “Check Your Model for HDL Compatibility” on page 25-16
- “Show Blocks Supported for HDL Code Generation” on page 25-18
- “Trace Code Using the Mapping File” on page 25-21
- “Add or Remove the HDL Configuration Component” on page 25-24

Create and Use Code Generation Reports

Report Generation

The HDL Coder software creates and displays an HTML code generation report when you select one or more of the following options. You can specify the UI options in the **HDL Code Generation > Report** pane of the Configuration Parameters dialog box.

GUI option	<code>makehdl</code> Property	Dependency
Generate traceability report	Traceability	“Generate HDL code” on page 17-94 must be enabled.
Generate resource utilization report	ResourceReport	“Generate HDL code” on page 17-94 and “Generated model” on page 17-82 must be enabled.
Generate high-level timing critical path report	CriticalPathEstimation	“Generate HDL code” on page 17-94 and “Generated model” on page 17-82 must be enabled.
Generate optimization report	OptimizationReport	“Generate HDL code” on page 17-94 and “Generated model” on page 17-82 must be enabled.
Generate model Web view	HDLGenerateWebview	“Generate HDL code” on page 17-94 must be enabled.

When you generate code, the Code Generation Report appears in a separate window.

Code Generation Report

The Code Generation Report is an HTML file that includes a **Summary**, a **Clock Summary**, a **Code Interface Report**, and one or more of the following optional sections:

- Traceability report
- Resource utilization report
- High-level timing critical path report
- Optimization report
- Model web view

Summary

The **Summary** lists information about the model, the DUT, the date of code generation, and top-level coder settings. The **Summary** also lists model properties that have nondefault values.

Code Interface Report

The **Code Interface Report** shows the DUT input and output port names, data types, and bit widths. The report displays links corresponding to each input port and output port in your Simulink model.

Timing and Area Report

When you select **Generate resource utilization report**, HDL Coder adds a **Timing and Area Report** section to the Code Generation Report. This section of the report contains the following subsections:

- **High-level Resource Report:** This section The **Summary** section summarizes multipliers, adders/subtractors, and registers consumed by the device under test (DUT).
The **Detailed Report** section contains more information on the resources that each subsystem uses. Wherever possible, the detailed report links back to corresponding blocks in your model. The **Detailed Report** section also contains a **Registers** section. This section displays the total 1-bit registers that is calculated as the sum of products over the bit widths of the registers and their frequency of occurrence.
- **Target-Specific Report:** When you request target-specific code generation on the model, this subsection shows the resource utilization report.

Optimization Report

When you select **Generate optimization report**, HDL Coder adds an Optimization Report section, with three subsections:

- **Distributed Pipelining:** If a subsystem has the **DistributedPipelining** option enabled, this subsection displays comparative listings of registers before and after you apply the distributed pipelining transform.
- **Streaming and Sharing:** Summary and detailed information about the subsystems for which you specify sharing or streaming optimizations and the delay balancing summary.
- **Target Code Generation:** Summary, status, and path delay information about the subsystems after target code generation.
- **Delay Balancing:** Lists the number of pipeline delays and phase delays added at the output ports to match the delays.

See Also

More About

- “Navigate Between Simulink Model and HDL Code by Using Traceability” on page 25-4
- “Critical Path Estimation Without Running Synthesis” on page 24-137
- “Web View of Model in Code Generation Report” on page 25-10

Navigate Between Simulink Model and HDL Code by Using Traceability

In this section...

- “How Traceability Works” on page 25-4
- “Generate Traceability Report” on page 25-5
- “Report Location” on page 25-5
- “View the Traceability Report” on page 25-6
- “Code-to-Model Navigation” on page 25-6
- “Model-to-Code Navigation” on page 25-8
- “Traceability Report Limitations” on page 25-9

Even a relatively small model can generate hundreds of lines of HDL code. To identify the mapping between your source model and the generated HDL code more easily, use the traceability support in HDL Coder.



How Traceability Works

When you enable traceability support and generate HDL code for your model, the code generator creates and displays an HTML code generation report.

By default, the code generator uses the line-level style to generate a traceability report. The report generated by using this style contains hyperlinks for each line of HDL code to navigate between code and model. You can customize the traceability style to generate a comment-based report. This style contains hyperlinked comments above a block of code that correspond to a searchable tag for a certain block in your model. To learn more about the two traceability styles, see “Traceability style” on page 18-4.

You can generate reports for the root-level model or for subsystems, blocks, Stateflow charts, or MATLAB Function blocks. By default, HDL Coder generates a report for the top-level model.

After you generate the report, you can navigate from:

- Model to code: Select a certain block in your model and navigate to corresponding lines of HDL code in the report.
- Code to model: Select a line of code in the report and navigate to Simulink blocks corresponding to that line of code.

HDL Coder provides this two-way navigation or bidirectional traceability. With traceability support, you can:

- Verify that the generated code is as you expect. You can identify which model elements correspond to a line of code, and track code from different model elements that you have or have not reviewed.
- Verify whether the generated code meets the design requirements. You can assign the requirements to model elements and include the requirements as hyperlinks in the traceability report.

Generate Traceability Report

You can generate the report in the Configuration Parameters dialog box or at the command-line.

- 1 Enable generation of the traceability report.
 - In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **Settings > Report Options**, and then select **Generate traceability report**.
 - At the command line, use `hdlset_param` to set the **Traceability** property on the model. To learn more about this parameter, see “Generate traceability report” on page 18-3.
- 2 Specify the traceability style. To generate a line-level traceability report, leave this setting as the default. To generate a comment-based traceability report:
 - On the **HDL Code Generation > Report** pane, specify the **TraceabilityStyle**.
 - At the command line, use `hdlset_param` to specify the **TraceabilityStyle** property on the model. To learn more about this parameter, see “Traceability style” on page 18-4.
- 3 Generate HDL code and the traceability report. Either select the DUT Subsystem and click **Generate HDL Code** on the Simulink Toolbar, or run `makehdl` on the DUT Subsystem at the command line.

When HDL code generation is complete, the HTML code generation report appears in a new window.

Report Location

By default, HDL Coder writes the code generation report files to a folder in the `hdlsrc\html\` folder of the build folder. If you close the report, you can navigate to this folder to reopen the report.

Before generating code, you can customize the target folder that stores the HDL code and the report files.

- In the Configuration Parameters dialog box, specify the target folder by using the **Target** setting.
- At the command line, use the **TargetDirectory** property.

To learn how to specify this parameter, see “Folder” on page 13-4.

To keep your traceability report up to date, regenerate the HDL code and report after modifying the source model.

View the Traceability Report

In the HTML code generation report window, select the **Traceability Report** section. In the left pane of the report, click the names of **Generated Source Files** to view their contents in a MATLAB web browser window.

This figure shows a typical traceability report.

Eliminated / Virtual Blocks

Block Name	Comment
<code><S2>/x_in</code>	Import
<code><S2>/h_in1</code>	Import
<code><S2>/h_in2</code>	Import
<code><S2>/h_in3</code>	Import
<code><S2>/h_in4</code>	Import
<code><S2>/y_out</code>	Output
<code><S2>/delayed_xout</code>	Output

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Subsystem: [`sfir_fixed/symmetric_fir`](#)

Object Name	Code Location
<code><S2>/a1</code>	<code>symmetric_fir.vhd:208, 209, 210</code>
<code><S2>/a2</code>	<code>symmetric_fir.vhd:212, 213, 214</code>

The traceability report has several subsections that indicate the blocks or subsystems from which the code was generated:

- The **Eliminated / Virtual Blocks** section accounts for blocks that are untraceable because they are not included in the generated HDL code.
- The **Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions** section provides a complete mapping between model elements and code.

If you assigned block requirements, you can see the requirements as hyperlinked comments in the traceability report. For more information, see “Include requirements in block comments” on page 17-47.

Code-to-Model Navigation

To navigate from the HDL code to the model:

- 1 In the traceability report, on the **Code Location** column, click any hyperlink.

The code generator highlights that line of HDL code in the generated source file.

- 2 Select the link corresponding to that line of code in the source file.

The code generator opens a separate window that displays the highlighted Simulink block corresponding to that line of code.

This figure shows how to navigate from the HDL code to the model by using the traceability report when you specify **Line Level** as the **Traceability style**.

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

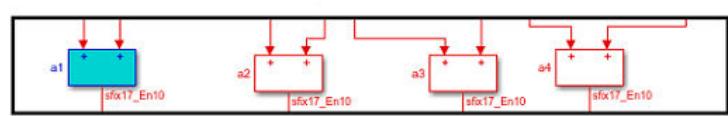
Subsystem: [sfir_fixed/symmetric_fir](#)

Object Name	Code Location
<code><S2>/a1</code>	symmetric_fir.vhd:208, 209, 210
<code><S2>/a2</code>	symmetric_fir.vhd:212, 213, 214
<code><S2>/a3</code>	symmetric_fir.vhd:216, 217, 218

```

206
207
208 a1_add_cast <= resize(ud8_out1, 17);
209 a1_add_cast_1 <= resize(ud1_out1, 17);
210 al_out1 <= a1_add_cast + a1_add_cast_1;
211
212 a2_add_cast <= resize(ud7_out1, 17);
213 a2_add_cast_1 <= resize(ud2_out1, 17);
214 a2_out1 <= a2_add_cast + a2_add_cast_1;
215

```



In the traceability report, you see that HDL Coder generates line-level hyperlinks to the HDL code in the **Code Location** column. Click the link to highlight that line of code in the HDL source file, and then click the hyperlink for that line of code in the source file to highlight the corresponding block in your model.

This figure shows how to navigate from the HDL code to the model using the traceability report when you specify **Comment Based** as the **Traceability style**.

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

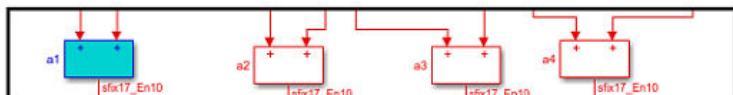
Subsystem: [sfir_fixed/symmetric_fir](#)

Object Name	Code Location
<code><S2>/a1</code>	symmetric_fir.vhd:216
<code><S2>/a2</code>	symmetric_fir.vhd:221
<code><S2>/a3</code>	symmetric_fir.vhd:226

```

214
215
216 -- <S2>/a1
217 a1_add_cast <= resize(ud8_out1, 17);
218 a1_add_cast_1 <= resize(ud1_out1, 17);
219 al_out1 <= a1_add_cast + a1_add_cast_1;
220
221 -- <S2>/a2
222 a2_add_cast <= resize(ud7_out1, 17);
223 a2_add_cast_1 <= resize(ud2_out1, 17);
224 a2_out1 <= a2_add_cast + a2_add_cast_1;
225

```



In the traceability report, when you select a hyperlink in the **Code Location** column, you see that HDL Coder highlights a hyperlinked comment `<S2>/a1` in the HDL code. When you click the hyperlinked comment in the HDL source file, the code generator highlights the corresponding block `a1` in your model.

Model-to-Code Navigation

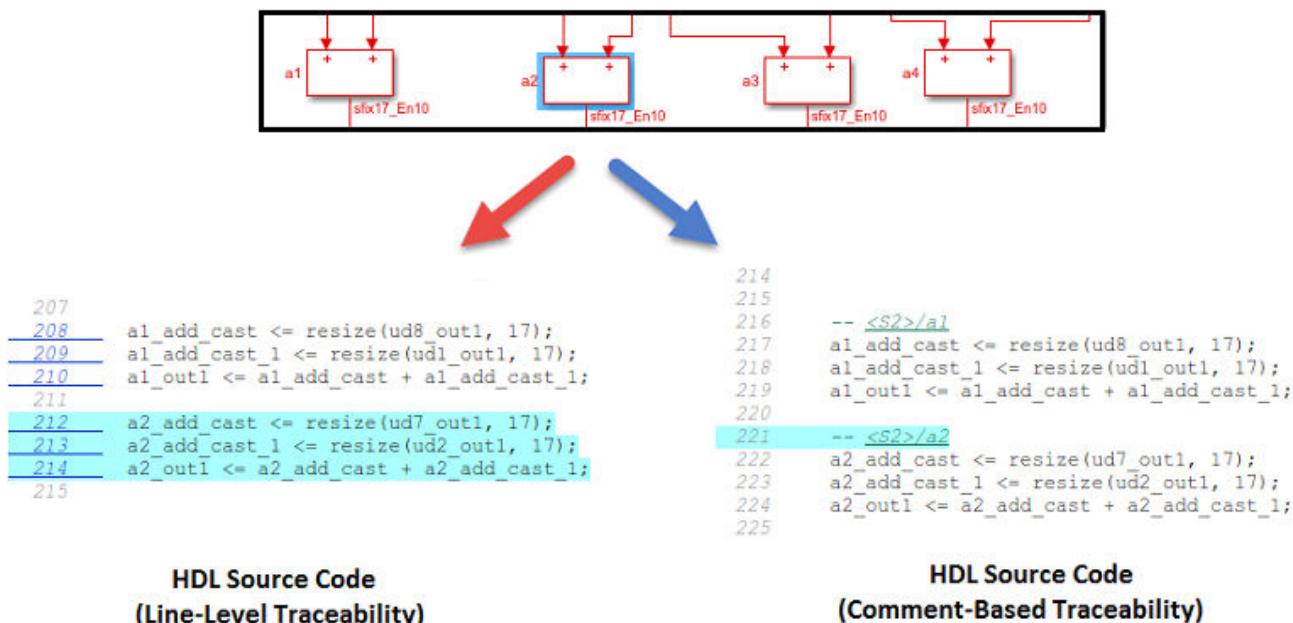
Use model-to-code traceability to select a component at any level of the model and view the code references to that component in the traceability report. For tracing, you can select these objects:

- Subsystem
- Simulink block
- MATLAB Function block
- Stateflow chart, or these elements of a Stateflow chart:
 - State
 - Transition
 - Truth table
 - MATLAB function inside a chart

You can navigate from a certain block in the model to the HDL code generated for that block by using either of these approaches.

- Select that block and click **Navigate to Code** on the **HDL Code** tab.
- Right-click that block in your Simulink model and select **HDL Code > Navigate to Code**.

This figure shows the model-to-code navigation for both line-level and comment-based traceability style.



If you use **Line Level** as the **Traceability style** and navigate from the model to the HDL code, the traceability report highlights all lines of HDL code corresponding to that block.

If you use **Comment Based** as the **Traceability style** and navigate from the model to the HDL code, the traceability report highlights the traceable block comment in the HDL code.

Traceability Report Limitations

- If a block name in your model contains a single quote ('), code-to-model and model-to-code traceability are disabled for that block.
- If an asterisk (*) in a block name in your model causes a name-mangling ambiguity relative to other names in the model, code-to-model highlighting and model-to-code highlighting are disabled for that block. This is most likely to occur if an asterisk precedes or follows a slash (/) in a block name or appears at the end of a block name.
- If a block name in your model contains the character ý (char(255)), code-to-model highlighting and model-to-code highlighting are disabled for that block.
- If you use certain subsystem types, the Subsystem block is not traceable from the model to the HDL code at the subsystem level. It is possible that you can trace individual blocks within the Subsystem block. You cannot trace from the model to the code for these subsystem types:
 - Virtual
 - Masked
 - Nonvirtual for which code has been optimized away
- Traceability does not support Model Reference as the top-level Subsystem block.

See Also

More About

- “Create and Use Code Generation Reports” on page 25-2

Web View of Model in Code Generation Report

In this section...

"About Model Web View" on page 25-10

"Generate HTML Code Generation Report with Model Web View" on page 25-10

"Model Web View Limitations" on page 25-11

About Model Web View

To review and analyze the generated code, it is helpful to navigate between the code and model. You can include a Web view of the model within the HTML code generation report. You can then share your model and generated code outside of the MATLAB environment. When you generate the report, the Web view includes the block diagram attributes displayed in the Simulink Editor, such as, block sorted execution order, signal properties, and port data types.

A Simulink Report Generator license is required to include a Web view (Simulink Report Generator) of the model in the code generation report.

Browser Requirements for Web View

Web view requires a Web browser that supports Scalable Vector Graphics (SVG). Web view uses SVG to render and navigate models.

You can use the following Web browsers:

- Mozilla Firefox Version 1.5 or later, which has native support for SVG. To download the Firefox browser, go to www.mozilla.com/.
- The Microsoft® Internet Explorer® Web browser with the Adobe® SVG Viewer plug-in. To download the Adobe SVG Viewer plug-in, go to www.adobe.com/svg/.
- Apple Safari Web browser

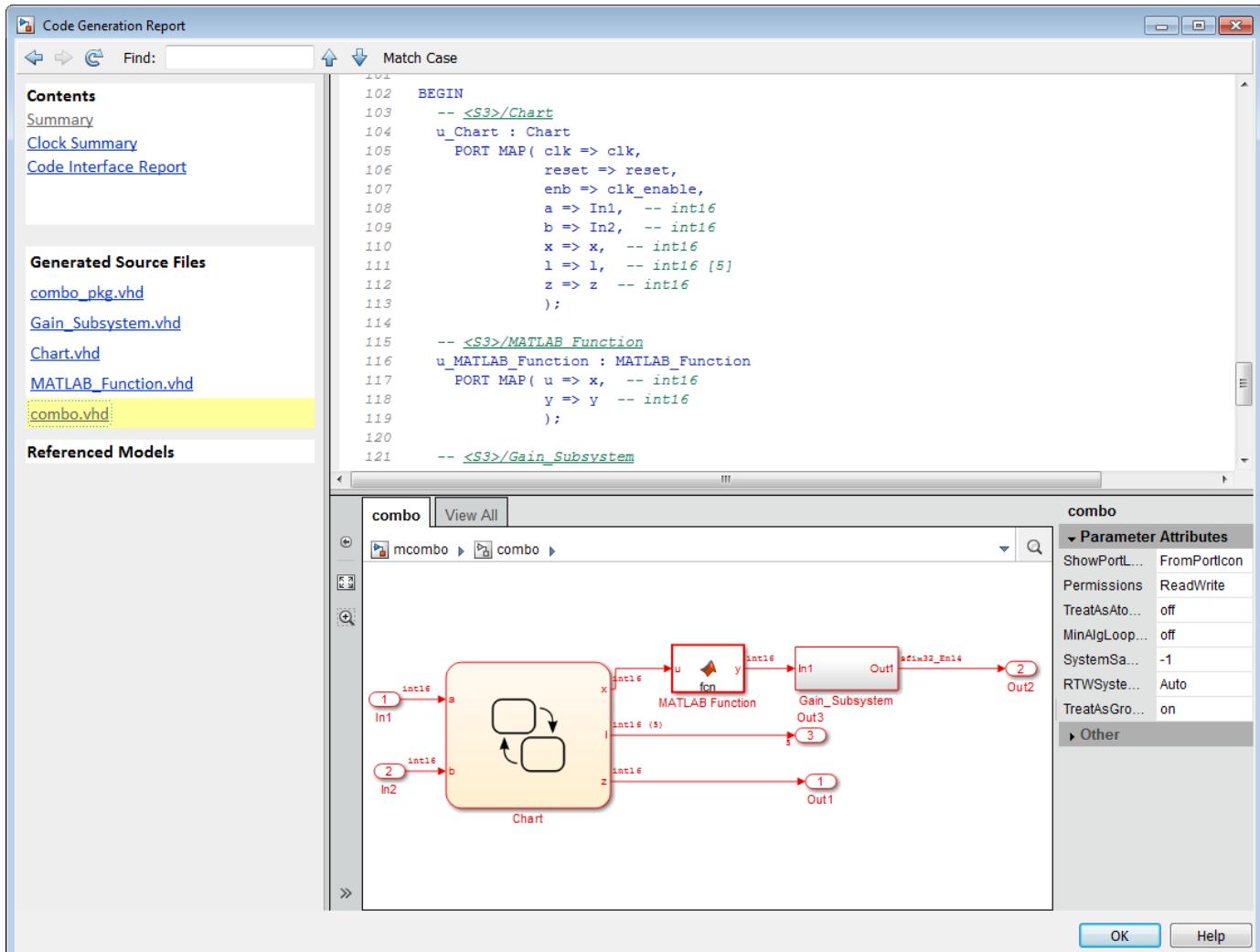
Generate HTML Code Generation Report with Model Web View

This example shows how to create an HTML code generation report which includes a Web view of the model diagram.

- 1 Open the `mcombo` model.
- 2 Open the **Configuration Parameters** dialog box or **Model Explorer** and navigate to the **HDL Code Generation** pane.
- 3 Under **Code generation report**, select **Generate model Web view**.
- 4 Click the **Generate** button.

After building the model and generating code, the code generation report opens in a MATLAB Web browser.

- 5 In the left navigation pane, select a source code file. The corresponding source code is displayed in the right pane and includes hyperlinks.



- 6 Click a link in the code. The model Web view displays and highlights the corresponding block in the model.
- 7 To highlight the generated code for a block in your model, click the block. The corresponding code is highlighted in the source code pane.
- 8 To go back to the code generation report for the top model, at the top of the left navigation pane, click the **Back** button until the top model's report is displayed.

For more information about exploring a model in a Web view, see “Navigate the Web View” (Simulink Report Generator).

Model Web View Limitations

The HTML code generation report includes the following limitations when using the model Web view:

- Code is not generated for virtual blocks. In the model Web view of the code generation report, when tracing between the model and the code, when you click a virtual block, it is highlighted yellow.

- In the model Web view, you cannot open a referenced model diagram by double-clicking the referenced model block in the top model. Instead, open the code generation report for the referenced model by clicking a link under **Referenced Models** in the left navigation pane.
- Stateflow truth tables, events, and links to library charts are not supported in the model Web view.
- Searching in the code generation report does not find or highlight text in the model Web view.
- If you navigate from the actual model diagram (not the model Web view in the report), to the source code in the HTML code generation report, the model Web view is disabled and not visible. To enable the model Web view, open the report again, see “Open Code Generation Report”.
- For a subsystem build, the traceability hyperlinks of the root level import and outport blocks are disabled.
- “Traceability Limitations” (Embedded Coder) that apply to tracing between the code and the actual model diagram.

Generate Code with Annotations or Comments

In this section...

- “Simulink Annotations” on page 25-13
- “Signal Descriptions” on page 25-13
- “Text Comments” on page 25-13
- “Requirements Comments and Hyperlinks” on page 25-14

The following sections describe how to use the HDL Coder software to add text annotations to generated code, in the form of model annotations, text comments or requirements comments.

Simulink Annotations

You can enter text directly on the block diagram as Simulink annotations. HDL Coder renders text from Simulink annotations as plain text comments in generated code. The comments are generated at the same level in the model hierarchy as the subsystem(s) that contain the annotations, as if they were Simulink blocks.

For Constant blocks, to reflect the annotations as comments in the HDL code, clear the “Minimize intermediate signals” on page 17-56 check box and set “Traceability style” on page 18-4 to **Comment Based**.

See “Describe Models Using Notes and Annotations” for general information on annotations.

Signal Descriptions

You can provide a description for the signals in your Simulink model. The generated HDL code displays these descriptions as comments above the signal declaration statements. To specify a description for the signal, right-click the signal, and select **Properties** to open the Signal Properties dialog box. Then, select the **Documentation** tab, and in the **Description** section, enter a description for the signal. For the signal description, use ASCII characters because non-ASCII characters in the generated code can potentially interfere with downstream synthesis and lint tools. In some cases, due to certain optimizations that act on the signals, the generated code may not translate all signal descriptions to HDL comments or may create replicas of HDL comments for certain signal descriptions.

Text Comments

You can enter text comments at any level of the model by placing a DocBlock at the desired level and entering text comments. HDL Coder renders text from the DocBlock in generated code as plain text comments. The comments are generated at the same level in the model hierarchy as the subsystem that contains the DocBlock.

Set the **Document type** parameter of the DocBlock to **Text**. HDL Coder does not support the **HTML** or **RTF** options.

See DocBlock for general information on the DocBlock.

Requirements Comments and Hyperlinks

You can assign requirement comments to blocks.

If your model includes requirements comments, you can choose to render the comments in one of the following formats:

- *Text comments in generated code:* To include requirements as text comments in code, use the defaults for **Include requirements in block comments** (on) and **Generate traceability report** (off) in the Configuration Parameters dialog box.

If you generate code from the command line, set the Traceability and RequirementComments properties:

```
makehdl(gcb, 'Traceability', 'off', 'RequirementComments', 'on');
```

The following figure highlights text requirements comments generated for a Gain block from the mcombo model.

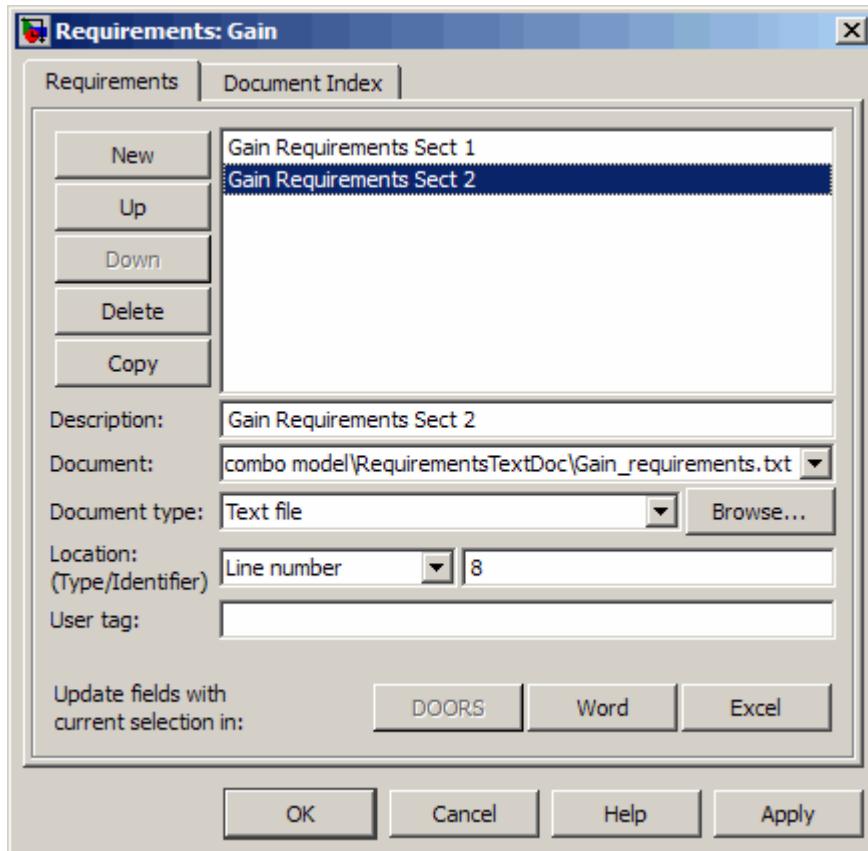
```
36 BEGIN
37     In1_signed <= signed(In1);
38
39     --
40     -- Block requirements for <S10>/Gain
41     -- 1. Gain Requirements Sect 1
42     -- 2. Gain Requirements Sect 2
43     Gain_gainparam <= to_signed(16384, 16);
44
45     Gain_out1 <= resize(In1_signed(15 DOWNTO 0) & '0'
46
47
48     Out1 <= std_logic_vector(Gain_out1);
49
50 END rtl;
```

- *Hyperlinked comments:* To include requirements comments as hyperlinked comments in an HTML code generation report, select both **Generate traceability report** and **Include requirements in block comments** in the Configuration Parameters dialog box.

If you generate code from the command line, set the Traceability and RequirementComments properties:

```
makehdl(gcb, 'Traceability', 'on', 'RequirementComments', 'on');
```

The comments include links back to a requirements document associated with the block and to the block within the original model. For example, the following figure shows two requirements links assigned to a Gain block. The links point to sections of a text requirements file.



The following figure shows hyperlinked requirements comments generated for the Gain block.

```

36      BEGIN
37          In1_signed <= signed(In1);
38
39          -- <S10>/Gain
40          --
41          --
42          -- Block requirements for <S10>/Gain
43          -- 1. Gain Requirements Sect 1
44          -- 2. Gain Requirements Sect 2
45          Gain_gainparam <= to_signed(16384, 16);
46
47          Gain_out1 <= resize(In1_signed(15 DOWNTO 0) &
48
49
50          Out1 <= std_logic_vector(Gain_out1);
51
52      END rtl;

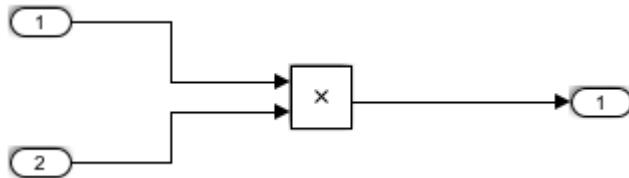
```

Check Your Model for HDL Compatibility

This example shows how to check whether a subsystem or model is compatible for HDL code generation by using the HDL compatibility checker. The HDL compatibility checker examines the specified system for compatibility problems, such as use of unsupported blocks, illegal data type usage, and so on. The HDL compatibility checker generates an HDL Code Generation Check Report. The report is stored in the target `hdlsrc` folder. The report file naming convention is `system_report.html`, where `system` is the name of the subsystem or model that is passed to the HDL compatibility checker. The HDL Code Generation Check Report is displayed in a MATLAB™ web browser window. Each entry in the HDL Code Generation Check Report has hyperlinks to the block or subsystem that is not compatible for HDL code generation.

Open this Simulink™ model that has a Product block inside a DUT Subsystem. The inputs to the block are a mix of double and integer data types.

```
load_system('hdlcoder_product_mixed_types')
open_system('hdlcoder_product_mixed_types/DUT')
```



To check whether the DUT Subsystem is compatible for HDL code generation, run the compatibility checker. To run the checker from the command line, use the `checkhdl` function. To learn more about the `checkhdl` function, see `checkhdl`.

```
checkhdl('hdlcoder_product_mixed_types/DUT', ...
    'TargetDirectory','C:/HDL_Checks/hdlsrc')

### Starting HDL check.
### Creating HDL Code Generation Check Report file://C:\HDL_Checks\hdlsrc\hdlcoder_product_mixed...
### HDL check for 'hdlcoder_product_mixed_types' complete with 1 errors, 1 warnings, and 0 messages
```

HDL check for 'hdlcoder_product_mixed_types' complete with 1 errors, 1 warnings, and 0 messages.

The following table describes blocks for which errors, warnings or messages were reported.

Simulink Blocks and resources	Level	Description
<code>hdlcoder_product_mixed_types/DUT/Product</code>	Error	Unhandled mixed double, single, and non-real datatypes at ports of block.
<code>hdlcoder_product_mixed_types/DUT</code>	Warning	Signals of type 'Double' will not generate synthesizable HDL. Consider enabling native floating-point mode and retyping all 'Double' typed signals to 'Single' to generate synthesizable code. More information

Click the `hdlcoder_product_mixed_types/DUT/Product` link to highlight the Product block inside the DUT Subsystem.

To run the compatibility checker from the UI:

- 1 Open the Configuration Parameters dialog box or the Model Explorer. Select the **HDL Code Generation** pane.
- 2 From the **Generate HDL for** dropdown, select the DUT Subsystem you want to check.
- 3 Click the **Run Compatibility Checker** button.

For a Subsystem that passes the HDL compatibility check, the HDL Code Generation Check Report contains a hyperlink to that subsystem.

Show Blocks Supported for HDL Code Generation

The `hdllib` function displays the blocks that are compatible with for HDL code generation in the Library Browser. It only displays those blocks for which you have a license. If you construct models using blocks from this Library Browser view, your models are compatible with HDL code generation.

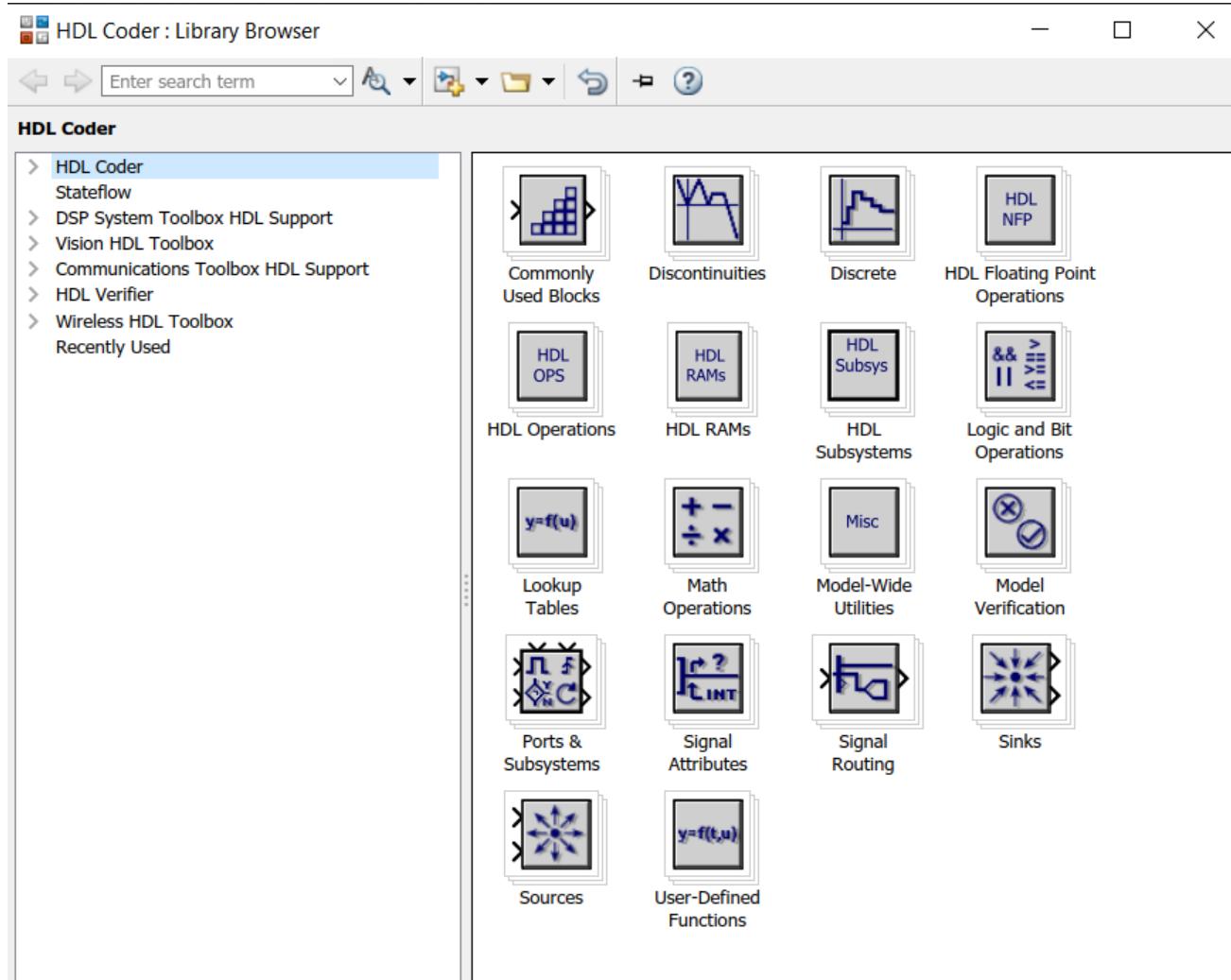
Parameter settings for blocks in this Library Browser view are compatible with HDL code generation, and therefore can differ from the default settings.

Show Supported Blocks in Library Browser

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

- In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Select **HDL Block Properties** > **Open HDL Block Library**.
- Alternatively, at the command prompt, enter:

```
hdllib
```



The Library Browser opens. You can drag and drop the blocks from the Library Browser into your model.

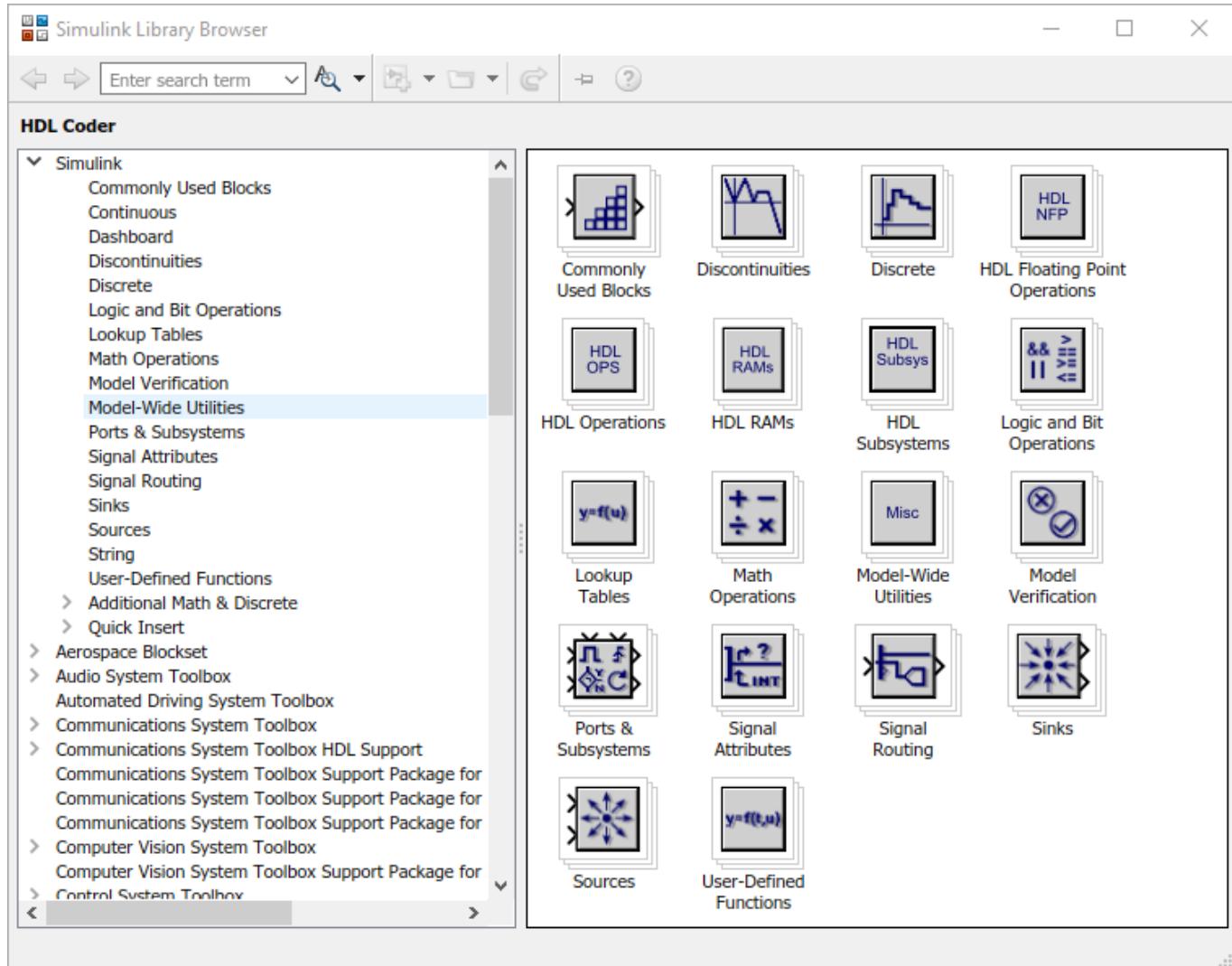
If you close and reopen the Library Browser in the same MATLAB session, you continue to see only the blocks that are compatible with HDL code generation. To reset the Library Browser view to show

all blocks, click the  button.

Reset Library Browser to Show All Blocks

To reset the Library Browser view so that it shows all blocks, regardless of HDL code generation compatibility, at the command prompt, enter:

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



To change the Library Browser view to show only those blocks that are compatible with HDL code generation, click the  button.

Generate a Supported Blocks Report

To generate an HTML table that lists the blocks that are compatible with HDL Code generation:

- 1 At the command prompt, enter:

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

hdllib creates the hdsupported library and the following HTML reports:

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

- 2 To see the generated list of blocks, click the `hdlblklist.html` link.

See Also

`hdllib`

More About

- “Create HDL-Compatible Simulink Model”
- “View HDL-Supported Blocks and HDL-Specific Block Documentation” on page 22-2

Trace Code Using the Mapping File

Note This section refers to generated VHDL entities or Verilog modules generically as “entities.”

A mapping file is a text report file generated by `makehdl`. Mapping files are generated as an aid in tracing generated HDL entities back to the corresponding systems in the model.

A mapping file shows the relationship between systems in the model and the VHDL entities or Verilog modules that were generated from them. A mapping file entry has the form

path --> *HDL_name*

where *path* is the full path to a system in the model and *HDL_name* is the name of the VHDL entity or Verilog module that was generated from that system. The mapping file contains one entry per line.

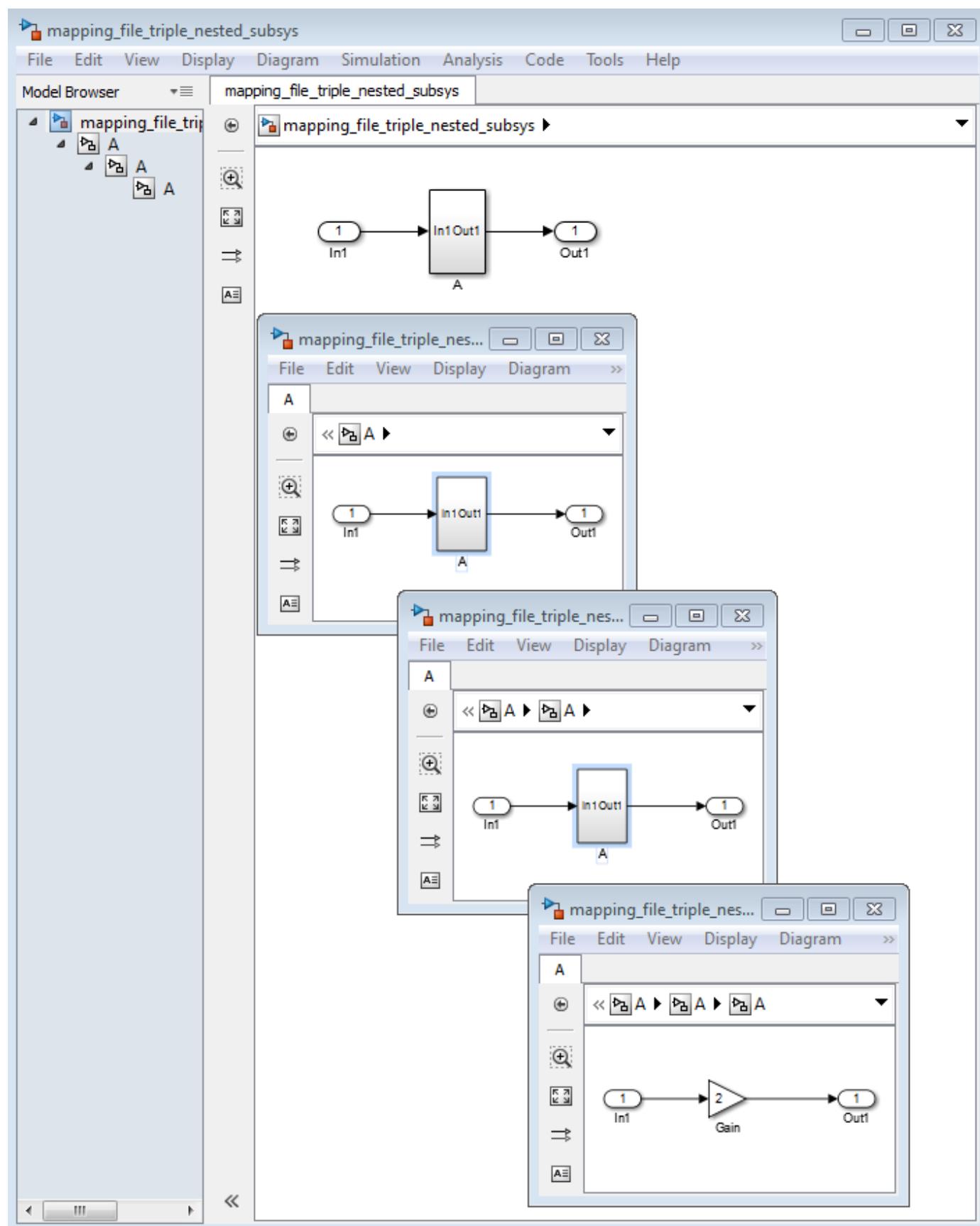
In simple cases, the mapping file may contain only one entry. For example, the `symmetric_fir` subsystem of the `sfir_fixed` model generates the following mapping file:

```
sfir_fixed/symmetric_fir --> symmetric_fir
```

Mapping files are more useful when HDL code is generated from complex models where multiple subsystems generate many entities, and in cases where conflicts between identically named subsystems are resolved by HDL Coder.

If a subsystem name is unique within the model, HDL Coder simply uses the subsystem name as the generated entity name. Where identically named subsystems are encountered, the coder attempts to resolve the conflict by appending a postfix string (by default, `'_entity'`) to the conflicting subsystem. If subsequently generated entity names conflict in turn with this name, incremental numerals ($1, 2, 3, \dots, n$) are appended.

As an example, consider the model shown in the following figure. The top-level model contains subsystems named A nested to three levels.



When code is generated for the top-level subsystem A, `makehdl` works its way up from the deepest level of the model hierarchy, generating unique entity names for each subsystem.

```
makehdl('mapping_file_triple_nested_subsys/A')
### Working on mapping_file_triple_nested_subsys/A/A/A as A_entity1.vhd
### Working on mapping_file_triple_nested_subsys/A/A as A_entity2.vhd
### Working on mapping_file_triple_nested_subsys/A as A.vhd

### HDL Code Generation Complete.
```

The following example lists the contents of the resultant mapping file.

```
mapping_file_triple_nested_subsys/A/A/A --> A_entity1
mapping_file_triple_nested_subsys/A/A --> A_entity2
mapping_file_triple_nested_subsys/A --> A
```

Given this information, you can trace a generated entity back to its corresponding subsystem by using the `open_system` command, for example:

```
open_system('mapping_file_triple_nested_subsys/A/A')
```

Each generated entity file also contains the path for its corresponding subsystem in the header comments at the top of the file, as in the following code excerpt.

```
-- Module: A_entity2
-- Simulink Path: mapping_file_triple_nested_subsys/A
-- Hierarchy Level: 0
```

Add or Remove the HDL Configuration Component

In this section...

["What Is the HDL Configuration Component?" on page 25-24](#)

["Adding the HDL Coder Configuration Component To a Model" on page 25-24](#)

["Removing the HDL Coder Configuration Component From a Model" on page 25-24](#)

What Is the HDL Configuration Component?

The HDL configuration component is an internal data structure that HDL Coder creates and attaches to a model. This component lets you view the **HDL Code Generation** pane in the Configurations Parameters dialog box and set HDL code generation options. Normally, you do not need to interact with the HDL configuration component. However, there are situations where you might want to add or remove the HDL configuration component:

- A model that was created on a system that did not have HDL Coder installed does not have the HDL configuration component attached. In this case, you might want to add the HDL configuration component to the model.
- If a previous user removed the HDL configuration component, you might want to add the component back to the model.
- If a model will be running on some systems that have HDL Coder installed, and on other systems that do not, you might want to keep the model consistent between both environments. If so, you might want to remove the HDL configuration component from the model.

Adding the HDL Coder Configuration Component To a Model

To add the HDL Coder configuration component to a model, In the Simulink Toolbar, on the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. On the **HDL Code** tab, select **Settings > Add HDL Coder Configuration to Model**.

Removing the HDL Coder Configuration Component From a Model

To remove the HDL Coder configuration component from a model, on the **HDL Code** tab, select **Settings > Remove HDL Configuration from Model**.

HDL Coding Standards

- “HDL Coding Standard Report” on page 26-2
- “HDL Coding Standards” on page 26-4
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5
- “Basic Coding Practices” on page 26-9
- “RTL Description Techniques” on page 26-18
- “RTL Design Methodology Guidelines” on page 26-41
- “Generate an HDL Lint Tool Script” on page 26-45

HDL Coding Standard Report

The HDL coding standard report shows how your generated HDL code conforms to an industry coding standard you select when generating code.

The report can contain errors, warnings, and messages. Errors and warnings in the report link to elements in your original design so you can fix problems, then regenerate code. Messages show where HDL Coder automatically corrected the code to conform to the coding standard.

The report also lists the rules in the coding standard with which the generated code complies. You can inspect the report to see which coding standard rules the coder checks.

HDL Coder Industry Compliance Report for the Simulink System demo_double/Subsystem1

Industry Rule Summary

Industry Compliance report with 1 errors, 5 warnings, 5 messages.

Generated by HDL Coder v.3.3, on 10-Jul-2013 10:36:47.

Rule	Level	Description
CGSL-1.A.A.2-3	⚠	Identifiers and names should follow recommended naming convention (1.A.A.2), and keywords in Verilog-HDL(IEEE1364), and keywords in VHDL(IEEE1076.X) must not be used (1.A.A.3). Violations
CGSL-1.A.A.4	🚫	Do not use names starting with VDD, VSS, VCC, GND or VREF. Violations
CGSL-1.A.A.9	🚫	Top-level module/entity and port names should be less than or equal to 16 characters in length and not be mixed-case. Violations
CGSL-1.A.D.1	⚠	Package file name should be followed by ".pac.vhd". Violations
CGSL-1.A.E.2	⚠	Clock, Reset, and Enable signals should follow recommended naming convention. Violations
CGSL-3.B.D.1	🚫	Non-integer type used in the declaration of a generic may be unsynthesizable. Violations

Industry Rule Hierarchy

- CGSL-1 Basic Coding Practices
 - CGSL-1.A General Naming Conventions
 - CGSL-1.A.A Design, Top-level Naming Conventions
 - CGSL-1.A.A.1 (recommended) ⓘ File names containing entities should have the extension .vhd or .vhdl.
 - CGSL-1.A.A.2-3 (default) ⚠ Identifiers and names should follow recommended naming convention (1.A.A.2), and keywords in Verilog-HDL(IEEE1364), and keywords in VHDL(IEEE1076.X) must not be used (1.A.A.3).
 - ⚠ Invalid industry identifier or keyword, found in Product block name 'Product'; consider renaming it. [demo_double/Subsystem1/Product](#)
 - CGSL-1.A.A.3vb. (default) ⓘ Do not use standard VHDL names.
 - CGSL-1.A.A.4 (mandatory) 🚫 Do not use names starting with VDD, VSS, VCC, GND or VREF.

To learn more about HDL coding standards, see “HDL Coding Standards” on page 26-4.

Rule Summary

The rule summary section shows the total numbers of errors, warnings, and messages, and lists the corresponding rules. Each rule shown in the summary links to the rule in the detailed rule hierarchy section.

Rule Hierarchy

The rule hierarchy section lists every rule HDL Coder checks, within three categories:

- Basic coding practices, including rules for names, clocks, and reset.
- RTL description techniques, including rules for combinatorial and synchronous logic, operators, and finite state machines.
- RTL design methodology guidelines, including rules for ports, function libraries, files, and comments.

If your HDL code does not conform to a specific rule, the rule shows either the automated correction, or a link to the original design element causing the error or warning. When you click a link, the

design opens with the design element highlighted. You can fix the problem in your design, then regenerate code.

Rule and Report Customization

You can configure the report so that it does not display passing rules by using the `ShowPassingRules` property of the HDL coding standard customization object. You can also disable or customize coding standard rules. See [HDL Coding Standard Customization](#).

How to Fix Warnings and Errors

To learn more about warnings and errors you can fix by modifying your design, see:

- “Basic Coding Practices” on page 26-9
- “RTL Description Techniques” on page 26-18
- “RTL Design Methodology Guidelines” on page 26-41

See Also

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

HDL Coding Standards

Industry coding standards recommend using certain HDL coding guidelines. HDL Coder generates code that follows industry standard rules and generates a report that shows how well your generated HDL code conforms to industry coding standards. See “HDL Coding Standard Report” on page 26-2.

HDL Coder checks for conformance of your Simulink model or MATLAB algorithm to the HDL coding standard rules.

The coder can also generate third-party lint tool scripts to use to check your generated HDL code. The industry standard rules fall under the following three sections:

- Section 1: “Basic Coding Practices” on page 26-9.
- Section 2: “RTL Description Techniques” on page 26-18.
- Section 3: “RTL Design Methodology Guidelines” on page 26-41.

When generating a coding standard report, HDL Coder adds a prefix to the rules. The rule prefix depends on whether you generate the report from MATLAB or Simulink. The rule prefix for MATLAB is CGML and for Simulink is CGSL.

To fix errors or warnings related to these rules, update your model design. You can customize some of the coding standard rules. See HDL Coding Standard Customization.

HDL coding standards provide language-specific code usage rules to help you generate more efficient, portable, and synthesizable HDL code, such as coding guidelines for:

- Names
- Ports, reset, and clocks
- Combinatorial and synchronous logic
- Finite state machines
- Conditional statements and operators

See Also

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

Generate an HDL Coding Standard Report from Simulink

In this section...

"Using the HDL Workflow Advisor" on page 26-5

"Using the Command Line" on page 26-7

You can generate an HDL coding standard report that shows how well your generated code follows industry standards. You can optionally customize the coding standard report and the coding standard rules.

Using the HDL Workflow Advisor

To generate an HDL coding standard report with the HDL Workflow Advisor:

- 1 In the **HDL Code Generation** task, in **Set Code Generation Options > Set Advanced Options**, select the **Coding standards** tab.
- 2 For **HDL coding standard**, select **Industry** and click **Apply**.

Additional settings

General Ports Optimization Coding style Coding standards Diagnostics Floating Point Target

Choose coding standard

HDL coding standard: Industry ▾

Report options

Do not show passing rules in coding standard report

Basic coding rules

Check for duplicate names

Check for HDL keywords in design names

Check module, instance, entity name length

Minimum 2

Maximum 32

Check signal, port, parameter name length

Minimum 2

Maximum 40

RTL description rules

Check for clock enable signals

Detect usage of reset signals

Detect usage of asynchronous reset signals

Minimize use of variables

Check for initial statements that set RAM initial values

Check for conditional statements in processes

Length 1

Check if-else statement chain length

Length 7

Check if-else statement nesting depth

Depth 3

Check multiplier width

Maximum 16

RTL design rules

Check for non-integer constants

Check line wrap length

- 3 Optionally, using the other options in the **Coding standards** tab, customize the coding standard rules and click **Apply**.

After you generate code, the message window shows a link to the HTML compliance report. To open the report, click the report link.

Using the Command Line

To generate an HDL coding standard report using the command-line interface, set the `HDLCodingStandard` property to `Industry` by using `makehdl` or `hdlset_param`.

For example, to generate HDL code and an HDL coding standard report for a subsystem, `sfir_fixed/symmetric_sfir`, enter the following command:

```
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry')

### 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 hdsrc\sfir_fixed\symmetric_fir.vhd
### Industry Compliance report with 4 errors, 18 warnings, 5 messages.
### Generating Industry Compliance Report symmetric_fir_Industry_report.html
### Generating SpyGlass script file sfir_fixed_symmetric_fir_spyglass.prj
### HDL code generation complete.
```

To open the report, click the report link.

You can customize the coding standard report and coding standard rule checks by specifying an HDL coding standard customization object. For example, for a subsystem, `sfir_fixed/symmetric_sfir`, you can create an HDL coding standard customization object, `cso`, set the maximum if-else statement chain length to 5 by using the `IfElseChain` property, and generate code:

```
cso = hdlcoder.CodingStandard('Industry');
cso.IfElseChain.length = 5;
makehdl('sfir_fixed/symmetric_fir','HDLCodingStandard','Industry', ...
'HDLCodingStandardCustomizations',cso)
```

See Also

Properties

HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-9
- “RTL Description Techniques” on page 26-18

- “RTL Design Methodology Guidelines” on page 26-41

Basic Coding Practices

In this section...

- “1.A General Naming Conventions” on page 26-10
- “1.B General Guidelines for Clocks and Resets” on page 26-15
- “1.C Guidelines for Initial Reset” on page 26-15
- “1.D Guidelines for Clocks” on page 26-16
- “1.F Guidelines for Hierarchical Design” on page 26-17

HDL Coder conforms to the following naming conventions and basic coding guidelines and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

1.A General Naming Conventions

1.A.A Design and Top-Level Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.A.1 Warning	Verilog: Source file name should be same as the name of the module in the file.	By default, HDL Coder generates code that has the same module and file name. If you use BlackBox architecture for your subsystem and generate code, the source names and file names can be different.	If you use BlackBox architecture for your subsystem, make sure that the source file name and module name are the same.
	VHDL: File names containing entities should have the extension .vhd or .vhdl.	Source file name has to use certain recommended naming conventions and file extensions.	Use the VHDL file extension option in the HDL Workflow Advisor, or the VHDLFileExtension property from the command line.
1.A.A.2 Message	Verilog/VHDL: Identifiers and names should follow recommended naming convention.	A name in the design does not start with a letter or contains a character other than a number, letter, or underscore.	Update the names in your design so that they start with a letter of the alphabet (a - z, A - Z), and contain only alphanumeric characters (a - z, A - Z, 0 - 9) and underscores (_).
1.A.A.3 Message	Verilog/VHDL: Keywords in Verilog-HDL(IEEE1364), SystemVerilog(v3.1a), and keywords in VHDL(IEEE1076.X) must not be used.	There are Verilog, SystemVerilog, or VHDL keywords within the names in your design.	Update the names in your design so that they do not contain Verilog, SystemVerilog, or VHDL keywords. You can disable this rule checking by using the HDLKeywords property of the HDL coding standard customization object.
1.A.A.3vb Message	VHDL: Do not use standard VHDL names.	HDL Coder does not use standard VHDL names.	No action required.
1.A.A.4 Error	Verilog/VHDL: Do not use names starting with VDD, VSS, VCC, GND or VREF.	A name or names in the design are not using the standard naming convention.	Update the names in your design so that they start with a letter of the alphabet (a - z, A - Z), and contain only alphanumeric characters (a - z, A - Z, 0 - 9) and underscores (_).

Rule / Severity	Message	Problem	Recommendations
1.A.A.5 Error	Verilog/VHDL: Do not use case variants of name in the same scope.	Two or more names in your design, within the same scope, are identical except for case. For example, the names foo and Foo cannot be in the same scope.	Update the names in your design so that no two names within the same scope differ only in case. You can disable this rule checking by using the DetectDuplicateNamesCheck property of the HDL coding standard customization object.
1.A.A.6 Warning	Verilog: Primary port names or module names must follow recommended naming convention.	HDL Coder generates code that complies with this rule for Verilog and VHDL.	No action required.
	VHDL: Component name should be same as its corresponding entity name.		
1.A.A.9 Warning	Verilog/VHDL: Top-level module/entity and port names should be less than or equal to 16 characters in length and not be mixed-case.	A top-level module, entity, or port name in the generated code is longer than 16 characters, or uses letters with mixed case.	Update the indicated name in your design so that it is less than or equal to 16 characters long, and all letters are lowercase. all letters must be either all uppercase or all lowercase. You can customize this rule by using the ModuleInstanceEntityNameLength property of the HDL coding standard customization object.

1.A.B Module Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.B.1-1b Error	<p>Verilog: Module and Instance names should be between 2 and 32 characters in length. The instance names including hierarchy should be less than or equal to 128 characters in length.</p> <p>VHDL: Entity names and instance names should be between 2 and 32 characters in length. The instance names including hierarchy should be less than or equal to 128 characters in length.</p>	A module, instance, or entity name in the generated code is fewer than 2 characters or more than 32 characters in length.	<p>Update the indicated name in your design so that it is from 2 through 32 characters in length.</p> <p>You can customize this rule by using the <code>ModuleInstanceEntityNameLength</code> property of the HDL coding standard customization object.</p>

1.A.C Signal Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.C.3 Error	<p>Verilog: Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length.</p> <p>VHDL: Signal names, variable names, type names, label names, and function names should be between 2 and 40 characters in length.</p>	A signal, port, parameter, define, or function name in the generated code is fewer than 2 characters, or more than 40 characters in length.	<p>Update function names or subsystem names in your design to be from 2 through 40 characters in length.</p> <p>You can customize this rule by using the <code>SignalPortParamNameLength</code> property of the HDL coding standard customization object.</p>

1.A.D File, Package, and Parameter Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.D.1 Warning	Verilog: Include files must have extensions that match ".h", ".vh", ".inc", and ".h", ".inc", "ht", ".tsk" for testbench.	The generated include files match these extensions for the testbench.	No action required.
	VHDL: Package file name should be followed by "pac.vhd".	By default, the generated package file postfix is _pkg.	In the Configuration Parameters dialog box, on the HDL Code Generation > Global Settings > General pane, specify the Package postfix to _pac.
1.A.D.4 Warning	Verilog: Macros defined outside a module must not be used in the module.	HDL Coder does not generate macros in the Verilog code, or redefine constants in the VHDL code.	No action required.
	VHDL: Constants should not be redefined.		
1.A.D.9 Warning	Verilog: Bit-width must be specified for parameters with more than 32 bits.	HDL Coder does not specify a bit-width greater than 32 bits in the generated code.	No action required.
	VHDL: Generic must not be used at top-level module.	If you use generics at top-level module or if you have mask parameters in your design and set the MaskParameterAsGeneric property, HDL Coder reports this violation.	If you have mask parameters in your design, set the MaskParameterAsGeneric to off.

1.A.E Register and Clock Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.E.2 Warning	Verilog/VHDL: Clock, Reset, and Enable signals should follow recommended naming convention.	The clock, reset, and enable signals are not using the recommended naming convention.	In the Configuration Parameters dialog box, on the HDL Code Generation > Global Settings pane, using the clock input port , reset input port , and clock enable input port options, update the names for the clock, reset, and enable signals respectively. Clock signal names must contain <code>clk</code> or <code>ck</code> , reset signal names must contain <code>rstx</code> , <code>resetx</code> , <code>rst_x</code> , or <code>reset_x</code> , and clock enable signal names must contain <code>en</code> .

1.A.F Architecture Naming Conventions

Rule / Severity	Message	Problem	Recommendations
1.A.F.1 Warning	VHDL: Architecture name must contain RTL.	In the generated VHDL code, the architecture name does not contain RTL.	In HDL Code Generation > Global Settings > General tab, update the VHDL architecture name to use an architecture name that contains RTL.
1.A.F.4 Warning	VHDL: An entity and its architecture must be described in the same file.	By default, HDL Coder describes the entity and architecture of the VHDL code in the same file. If you set the SplitEntityArch property to on , the generated VHDL code describes the entity and architecture in separate files, so HDL Coder reports a warning.	Set SplitEntityArch to off so that HDL Coder describes the entity and architecture of the VHDL code in the same file.

1.B General Guidelines for Clocks and Resets

1.B.A Clock Constraints

Rule / Severity	Message	Problem	Recommendations
1.B.A.1 Message	VHDL: Design should have only a single clock and use only one edge of the clock.	Your design uses multiple edges of the clock or contains more than one clock signals. If you set the ClockInputs property to multiple or use TriggerAsClock to use the trigger signal for a triggered subsystem as clock, HDL Coder generates this message.	Update your design to use a single clock signal. In the HDL Code Generation > Global Settings panel, set Clock inputs to Single , and Clock edge to Rising or Falling .
1.B.A.2 Error	Verilog/VHDL: Do not create an RS latch or flip-flop using primitive cells such as AND, OR.	HDL Coder does not create latches, and complies with this rule.	No action required.
1.B.A.3 Error	Verilog/VHDL: Remove combinational loops.	HDL Coder does not create combinational loops.	No action required.

1.C Guidelines for Initial Reset

1.C.A Flip-Flop Clock Constraints

Rule / Severity	Message	Problem	Recommendations
1.C.A.3 Warning	Verilog/VHDL: Do not use asynchronous set/reset signals other than initial reset.	HDL Coder does not use asynchronous reset signals as non-reset or synchronous reset signals.	No action required.
1.C.A.6 Error	Verilog/VHDL: Signals must not be used as both asynchronous reset and synchronous reset.	HDL Coder adds the reset control logic outside the DUT and does not generate both asynchronous reset and synchronous reset signals.	No action required.
1.C.A.7 Warning	Verilog/VHDL: A flip-flop must not have both asynchronous set and asynchronous reset.	HDL Coder does not generate code with both asynchronous set and reset signals.	No action required.

1.C.B Reset Conventions

Rule / Severity	Message	Problem	Recommendations
1.C.B.1a Message	Verilog/VHDL: Asynchronous resets or sets must not be gated.	HDL Coder does not gate asynchronous set or reset signals.	No action required.
1.C.B.1b Message	Verilog/VHDL: Reset must be generated in separate module instantiated at top-level.	The generated code complies with this rule, because the DUT does not contain reset instantiation.	No action required.
1.C.B.2 Warning	Verilog/VHDL: Do not use signals other than initial reset for asynchronous reset input of flip-flop.	HDL Coder uses only initial reset signals for asynchronous reset input of flip-flop.	No action required.

1.D Guidelines for Clocks

1.D.A Clock Packaging Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.A.1 Warning	Verilog/VHDL: Clock should be generated in separate module or entity instantiated at top-level.	HDL Coder generates code that complies with this rule, because the DUT does not contain clock instantiation.	No action required.

1.D.C Clock Gating Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.C.2-4 Message	Verilog/VHDL: Do not use flip-flop outputs as clocks of other flip-flops and flip-flop clock signals as non-clock signals.	HDL Coder does not use the output of flip-flops as clocks of other flip-flops, or flip-flop clock signals as nonclock signals.	No action required.
1.D.C.6 Message	Verilog/VHDL: Do not use flip-flops with inverted edges.	If your Simulink model uses a Triggered Subsystem block with rising and falling triggers and has TriggerAsClock enabled, HDL Coder violates this rule.	Disable TriggerAsClock or do not use Triggered Subsystem blocks with both rising and falling triggers in your Simulink model.

1.D.D Clock Hierarchy Constraints

Rule / Severity	Message	Problem	Recommendations
1.D.D.2 Message	Verilog: One hierarchical level should have a single clock only.	Your Simulink model uses multiple clock signals.	Update your design to use a single clock signal. In the HDL Code Generation > Global Settings panel, set Clock inputs to Single.

1.F Guidelines for Hierarchical Design

1.F.A Basic Block Size Guidelines

Rule / Severity	Message	Problem	Recommendations
1.F.A.4 Error	Verilog/VHDL: Clock generation, reset generation, RAM, Setup/Hold ensure buffers, and I/O cells must be a module at top-level.	HDL Coder generates separate modules for the DUT, RAM, timing controller, so that it complies with this rule.	No action required.

See Also

Properties

HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

More About

- “HDL Coding Standard Report” on page 26-2
- “RTL Description Techniques” on page 26-18
- “RTL Design Methodology Guidelines” on page 26-41

RTL Description Techniques

In this section...

- “2.A Guidelines for Combinational Logic” on page 26-18
- “2.B Guidelines for “Always” Constructs of Combinational Logic” on page 26-23
- “2.C Guidelines for Flip-Flop Inference” on page 26-25
- “2.D Guidelines for Latch Description” on page 26-28
- “2.E Guidelines for Tristate Buffer” on page 26-29
- “2.F Guidelines for Always/Process Construct with Circuit Structure into Account” on page 26-30
- “2.G Guidelines for “IF” Statement Description” on page 26-30
- “2.H Guidelines for “CASE” Statement Description” on page 26-32
- “2.I Guidelines for “FOR” Statement Description” on page 26-35
- “2.J Guidelines for Operator Description” on page 26-36
- “2.K Guidelines for Finite State Machine Description” on page 26-39

HDL Coder conforms to the following RTL description rules and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

2.A Guidelines for Combinational Logic

2.A.A Combinatorial Logic Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.A.1 Reference	VHDL: Package IEEE.std_logic_1164 must be included in each entity.	HDL Coder includes the package in each entity in the generated VHDL code.	No action required.
2.A.A.2 Warning	Verilog: A function description must assign return values to all possible states of the function.	HDL Coder does not generate functions for DUT.	No action required.
2.A.A.3 Warning	Verilog: Check using RTL parsing tool for error prevention.	HDL Coder generates VHDL and Verilog code with the correct syntax and complies with this rule.	No action required.

2.A.B Function Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.B.1 Error	Verilog: Function statement should not be used for asynchronous reset line logic in an always construct for FF inference.	HDL Coder does not generate functions for DUT.	No action required.
	VHDL: Use std_logic or std_logic_vector data types to describe ports of an entity.	At the inputs and outputs, HDL Coder uses std_logic or std_logic_vector to describe the ports.	No action required.
2.A.B.2-3 Error	Verilog: Do not use nonblocking assignment, or input argument as input in function description.	The generated HDL code complies with this rule for Verilog.	No action required.
	VHDL: Use range specification for integer types.	By default, HDL Coder specifies the range for integer types in the generated code.	No action required.
2.A.B.4 Error	Verilog: Task constructs should not be used in the design.	HDL Coder does not use tasks or fork-join constructs in the Verilog code.	No action required.
	VHDL: Do not use bit and bit vector data types in the design.	HDL Coder does not use bit or bit vector data types in the generated code.	No action required.
2.A.B.5 Error	Verilog: Clock edges should not be used in a task description.	When generating Verilog code, HDL Coder does not use clock edges in a task description.	No action required.
2.A.B.6 Error	VHDL: Specify range for std_logic_vector.	HDL Coder complies with this rule, because the generated VHDL code specifies the range that std_logic_vector uses.	No action required.

2.A.C Bit Width Matching Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.C.1-2 Error	Verilog: Ensure that bitwidth of function arguments matches that of corresponding function inputs, and bitwidth of function return value matches that of assignment destination signal.	At module instantiation, HDL Coder enforces type matching, so that it complies with this rule.	No action required.
	VHDL: Use only 'in', 'out', and 'inout' ports. Do not use buffer and linkage.	When generating VHDL code, HDL Coder specifies 'IN', 'OUT', or 'INOUT' ports, and does not use buffer or linkage.	No action required.
2.A.C.3 Error	Verilog: Use concatenation when assigning to multiple signals.	HDL Coder complies with this rule.	No action required.
	VHDL: Port mode must be explicitly specified.	When generating VHDL code, HDL Coder specifies 'IN', 'OUT', or 'INOUT' ports and does not use buffer or linkage.	No action required.
2.A.C.4-5 Error	Verilog: In function description, do not assign global signals, and return value assignment must be the last statement.	HDL Coder generates Verilog code that complies with this rule.	No action required.
	VHDL: Input port must not be described with initial value.	In the generated VHDL code, HDL Coder does not specify an initial value to the input port.	No action required.

2.A.D Operators Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.D.5 Message	Verilog: Bit-wise operators must be used instead of logical operators in multi-bit operations.	In the generated Verilog code, HDL Coder complies with this rule for multibit operators.	No action required.
2.A.D.6 Message	Verilog: Reduction of a single-bit or large expression should not be performed.	By default, HDL Coder does not reduce a single-bit or a large expression. If your design performs bit-reduction operations, the resulting HDL code can perform reduction of a large expression.	Update your design so that there are no calls to bit reduction operations.

2.A.E Conditional Statement Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.E.3 Message	Verilog: Ensure that conditional expressions evaluate to a scalar.	HDL Coder complies with this rule.	No action required.

2.A.F Array, Vector, Matrix Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.F.2 Warning	Verilog/VHDL: LSB of vectors/memory should be zero.	Your design contains vectors whose LSB has a nonzero value.	Update your design so that the generated code contains vectors or memory whose LSB value is zero.
2.A.F.4 Warning	Verilog/VHDL: Index variable width should not be too short.	HDL Coder enforces type matching and ensures that the index variable width is not too short.	No action required.
2.A.F.5 Error	Verilog/VHDL: Do not use x and z for an array index.	In the generated code, HDL Coder does not use x or z for an array index.	No action required.

2.A.G Assignment Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.G.1 Error	VHDL: Direct assignment must be used for aggregates.	HDL Coder directly assigns aggregates in the generated code without performing any intervening operations.	No action required.

2.A.H Function Return Value Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.H.1 Reference	VHDL: Constrained arrays should not be used as sub-program description.	In the generated code, HDL Coder does not use constrained arrays in subprogram description.	No action required.
2.A.H.2 Reference	VHDL: Specify range for return values in function description when return type is array.	In function description, when the return type is array, HDL Coder specifies the range for return values in function in the generated code.	No action required.
2.A.H.4-6 Error	VHDL: In a sub-program description, use only OTHERS clause when specifying aggregates, not use or call a nested subprogram description, and not read Global signals.	HDL Coder complies with this rule.	No action required.
2.A.H.9-10 Warning	VHDL: A function must have a return statement, return a valid value in all possible states, and not have any other statement following the return statement.	HDL Coder complies with this rule.	No action required.

2.A.I Built-in Attribute Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.I.4-5 Error	VHDL: Do not use user-defined attributes, or built-in attributes except range, length, left, right, high, low, reverse_range, and event.	By default, HDL Coder does not use user-defined attributes in the generated code. If you set HDL block properties, such as DSPStyle in your design, the generated code uses synthesis directives.	To fix this error, in your design, clear the HDL block property that you have set for using synthesis directives in the generated code.

2.A.J VHDL Specific Conventions

Rule / Severity	Message	Problem	Recommendations
2.A.J.1-6 Warning	VHDL: In a design, do not use block statements, objects of type record, shared variables, while-loop statements, procedures, or selected signal assignments.	If your design uses loop statements, HDL Coder generates this warning.	To avoid this warning, update your design so that there are no looping statements.
2.A.J.8-13 Error	VHDL: In a design, do not use access types, alias declarations, bus and register signals, disconnect specifications, waveforms, and attributes that are defined in Synopsys library.	HDL Coder complies with this rule.	No action required.

2.B Guidelines for “Always” Constructs of Combinational Logic

2.B.A Latch Constraints

Rule / Severity	Message	Problem	Recommendations
2.B.A.2 Reference	Verilog/VHDL: Check latch creation from RTL lint checker and synthesis tools; Design should not have latches.	HDL Coder does not create latches.	No action required.

2.B.B Signal Constraints - I

Rule / Severity	Message	Problem	Recommendations
2.B.B.2-3 Message	Verilog/VHDL: In the sensitivity list of a process or always block, do not define constants, use wait statements, or include a signal that is not read inside that block.	HDL Coder generates code that complies with the use of these constructs inside a process block (VHDL) or an always block (Verilog).	No action required.
	Verilog: Do not describe multiple event expressions with always constructs.	HDL Coder does not describe more than one event expression in an always construct.	No action required.

2.B.C Signal Constraints - II

Rule / Severity	Message	Problem	Recommendations
2.B.C.1-2 Error	Verilog: Do not use nonblocking assignments in combinational always blocks, or when assigning initial values in always constructs of sequential blocks.	Your design uses constructs that generate Verilog code with nonblocking assignments in combinational always blocks or assigns initial values in always constructs of sequential blocks.	Update your MATLAB algorithm or Stateflow design so that the generated Verilog code does not use these constructs.
2.B.C.3 Message	Verilog/VHDL: Do not assign a signal more than once in an always construct for sequential circuits.	In an always construct for sequential circuits, HDL Coder does not perform multiple assignments to a signal.	No action required.

2.C Guidelines for Flip-Flop Inference

2.C.A Assignment Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.A.1-2c Error	Verilog/VHDL: In flip-flop description, do not use quasi-continuous assignments, deassign statements, blocking assignments, variable assignment statements, or stable attribute.	HDL Coder does not introduce any additional data or add these constructs when generating flip-flops in process blocks (VHDL) or always blocks. (Verilog)	No action required.
2.C.A.4-5b Warning	Verilog/VHDL: Only flip-flop data paths can have delays. The delay values must be integral and non-negative.	HDL Coder does not generate code that uses DELAY attributes for the DUT. The generated testbench can contain DELAY attributes.	No action required.
2.C.A.6 Error	Verilog/VHDL: Check the logic level of the reset signal as specified in the sensitivity list of the always block.	HDL Coder uses posedge or negedge to denote transitions at clock edges in the generated code.	No action required.
2.C.A.7 Message	Verilog/VHDL: A flip-flop should not have two asynchronous resets. Do not use functions in the asynchronous reset description.	HDL Coder does not generate multiple asynchronous resets. The generated code can contain multiple synchronous resets.	No action required.
2.C.A.8 Error	VHDL: Do not use wait constructs.	HDL Coder does not use wait constructs.	No action required.
2.C.A.9 Error	VHDL: Functions 'rising_edge' or 'falling_edge' should not be used in the design.	<p>By default, HDL Coder uses the event syntax for clock events.</p> <p>By using the <code>UseRisingEdge</code> property, you can specify whether to use the <code>rising_edge</code> or <code>falling_edge</code> to detect clock transitions.</p>	To fix this error, you can control the <code>UseRisingEdge</code> property such that the generated code uses the event syntax.

2.C.B Blocking Statement Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.B.1-2 Warning	Verilog/VHDL: Use blocking assignment in flip-flop description. Do not use blocking and nonblocking assignments together in the same always block.	HDL Coder complies with this rule.	No action required.
2.C.B.4 Error	VHDL: Variables, if used, must be assigned to a signal before the end of the process.	The generated HDL code does not contain dead code, so HDL Coder complies with this rule.	No action required.

2.C.C Clock Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.C.1-2b Error	Verilog/VHDL: Do not use edges of multiple clocks or both edges of the same clock in an always block. Do not describe multiple clock edges in a single process/always block for same edge of a single clock.	HDL Coder uses the rising edge or falling edge of the clock, but does not use both edges of the clock.	No action required.
2.C.C.4-5 Error	Verilog/VHDL: Minimize, and if possible, remove clock enable signals and reset signal on networks.	If your design generates code that uses clock enables and reset signals on networks, HDL Coder generates an error.	<p>To minimize clock enables in the generated HDL code, in the HDL coding standard customization properties, enable the MinimizeClockEnableCheck property.</p> <p>To remove reset signals on the networks, in the HDL coding standard customization properties, enable the RemoveResetCheck setting.</p>
2.C.C.6 Warning	Verilog/VHDL: Do not use asynchronous reset signals.	Your Simulink model design or MATLAB code uses asynchronous reset signals.	To avoid this violation, use synchronous reset signals for your design. In the Configuration Parameters dialog box, set Reset type to Synchronous.

2.C.D Initial Value Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.D.1 Error	Verilog/VHDL: Do not specify flip-flop or RAM initial value using <code>initial</code> construct.	The generated HDL code for your design contains an unsynthesizable <code>initial</code> statement.	<p>Disable the Initialize block RAM or Initialize all RAM blocks option in the HDL Workflow Advisor.</p> <p>You can disable this rule checking by using the <code>InitialStatements</code> property of the HDL coding standard customization object.</p>

2.C.F Mixed Timing Constraints

Rule / Severity	Message	Problem	Recommendations
2.C.F.1-2a Warning	Verilog/VHDL: Do not use multiple resets or mix descriptions of flip-flops with and without asynchronous reset in the same process/always block.	HDL Coder complies with this rule.	No action required.

2.D Guidelines for Latch Description

2.D.A Module Constraints

Rule / Severity	Message	Problem	Recommendations
2.D.A.2-3 Warning	Verilog/VHDL: Latch descriptions should not have asynchronous set or asynchronous reset, or be mixed with other descriptions in the same module.	HDL Coder does not create latches in the generated code.	No action required.
2.D.A.4-5 Error	Verilog/VHDL: Do not use combinational loops that contain latches or level two latches in the same phase clock.	By default, HDL Coder does not create combinational loops. If your MATLAB algorithm contains combinational loops, the generated HDL code can use combinational loops.	Update your MATLAB code so that the generated HDL code does not contain any combinational loops.

2.E Guidelines for Tristate Buffer

2.E.A Module Constraints

Rule / Severity	Message	Problem	Recommendations
2.E.A.1-2 Warning	Verilog/VHDL: Tristate descriptions must not be mixed with other descriptions in the same module and should not contain logic in tristate enable conditions.	HDL Coder does not create latches or tristate buffers in the generated code.	No action required.
2.E.A.4-5b Reference	Verilog/VHDL: Tristate bus must not be driven by more than specified number of drivers. A net that is not tristated or a signal without a resolution function must not have multiple drivers.	HDL Coder does not create latches or tristate buffers in the generated code.	No action required.
2.E.A.6-9 Error	Verilog/VHDL: Inout port should not be directly connected to input/output. Do not use tristate output in an if conditional expression or in the selection expression of a case statement that assigns a fixed value in others choice.	<p>By default, HDL Coder does not connect input or output ports directly to bidirectional ports.</p> <p>In your Simulink model, on the HDL block properties for the input or output port, if you set BidirectionalPort to on, the generated HDL code can directly connect inout to input or output ports.</p>	<p>In your Simulink model, on the HDL block properties for the input or output port, set BidirectionalPort to off.</p>

2.E.B Connectivity Constraints

Rule / Severity	Message	Problem	Recommendations
2.E.B.1 Warning	Verilog/VHDL: Logic directly driven by tristate nets should be in a separate module.	HDL Coder does not have tristate nets in the generated HDL code.	No action required.

2.F Guidelines for Always/Process Construct with Circuit Structure into Account

2.F.B Constraints on Number of Conditional Statements

Rule / Severity	Message	Problem	Recommendations
2.F.B.1 Error	Verilog/VHDL: Do not describe more than one statement (if/case/while/for/loop) separately within a single always or process block.	The generated HDL code for your design contains more than one conditional statement (if-else, case, and loops) that is described separately within a process block (for VHDL code) or an always block (for Verilog code).	Update your design so that there is not more than one conditional statement that is described separately in a process block. You can customize this rule by using the <code>ConditionalRegionCheck</code> property of the HDL coding standard customization object.
2.F.B.2 Error	Verilog/VHDL: A variable in the sensitivity list is modified inside the same process or always block.	HDL Coder does not modify the variables in the sensitivity list, including clock, reset, and enable signals.	No action required.

2.G Guidelines for “IF” Statement Description

2.G.B Common Sub-Expression Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.B.2 Warning	Verilog/VHDL: Avoid describing conditions that will not be executed.	The generated HDL code does not contain dead code, or result in conditions that are not executed.	No action required.

2.G.C Nesting Depth Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.C.1a-b Message	Verilog/VHDL: Nesting in if-else constructs should not be deeper than N levels. Where feasible case statements should be used, rather than if-else statements, if performance is important.	The MATLAB code contains an if-elseif statement with more than N levels of nesting. By default, N is 3.	<p>Modify if-elseif statements in your MATLAB code so there are N or fewer levels of nesting.</p> <p>For example, the following if-elseif pseudocode contains three levels of nesting:</p> <pre>if ... if ... if ... else else else</pre> <p>You can customize this rule by using the <code>IfElseNesting</code> property of the HDL coding standard customization object.</p>
2.G.C.1c Message	Verilog/VHDL: Chain of if...else if constructs must not be exceed default number of levels.	The generated HDL code contains an if-elseif statement with more than seven branches.	<p>Modify if-elseif statements in your MATLAB code so that the number of branches is seven or fewer.</p> <p>For example, the following if-elseif pseudocode contains three branches:</p> <pre>if ... elseif ... elseif ... else</pre> <p>You can customize this rule by using the <code>IfElseChain</code> property of the HDL coding standard customization object.</p>

2.G.D Begin-End Decorator Constraints

Rule / Severity	Message	Problem	Recommendations
2.G.D.2-3 Message	Verilog/VHDL: Attach begin-end to "if" statements.	The generated HDL code complies with these code constructs.	No action required.
	Verilog: Do not use fork-join constructs.		

2.H Guidelines for “CASE” Statement Description

2.H.A CASE Structure Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.A.3-5 Reference	Verilog/VHDL: case constructs should not have overlapping clause conditions. Do not use full_case directive.	The generated HDL code complies with these constructs for case statements and does not use the full_case directive.	No action required.

2.H.C Default Value Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.C.3 Warning	Verilog: Do not use //synposys full_case pragma when all conditions are not described as case clause or the default clause is missing.	HDL Coder describes all possible cases in a case statement so that the synthesis tool does not infer a latch.	No action required.
2.H.C.4 Message	Verilog/VHDL: A signal that is assigned don't care value in a case default clause should not be used in if conditions, ternary and case constructs.	HDL Coder does not use a signal that is assigned a <i>don't care</i> value in the default clause.	No action required.
2.H.C.5 Warning	Verilog/VHDL: Default clause in case construct must be the last clause.	To avoid latch inference, HDL Coder describes all possible cases, including the default clause.	No action required.
2.H.C.6-7 Message	Verilog/VHDL: Do not use a signal to which don't care is assigned for selection expression of casex statements or case statements that do not assign 'X' in default clause.	HDL Coder does not use <i>don't care</i> values, and explores the entire space of an n-bit select signal.	No action required.

2.H.D Don't Care Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.D.1-4 Message	Verilog: Design should not use casex or casez constructs. casex or casez constructs must contain a dont-care condition, and not have complex clause conditions. The don't care condition in casex or casez branches must follow proper coding style.	HDL Coder does not generate casex or casez constructs, so that it complies with this rule.	No action required.

2.H.E Additional CASE Constraints

Rule / Severity	Message	Problem	Recommendations
2.H.E.1-4 Message	Verilog: Do not use parallel_case directive. In a case clause condition, do not use fixed values, variables, expressions, and logical, arithmetic, bitwise, or reduction operations.	HDL Coder does not use the parallel_case directive and generates code that complies with these constructs.	No action required.

2.I Guidelines for “FOR” Statement Description

2.I.A Loop Body Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.A.2a-b Message	Verilog: Loop variable and terminating condition of "for" construct must have constant initial value.	HDL Coder does not generate casex or casez constructs so that it complies with this rule.	No action required.
2.I.A.2c-e Message	Verilog: Loop variable of "for" construct must have a constant value inside the construct and must not be used outside the construct.	HDL Coder generates the right loop constructs and complies with this rule.	No action required.
	Verilog: The loop termination condition must not be a constant.		

2.I.B Non-Constant Operation Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.B.4 Error	Verilog/VHDL: Separate for loops must be used in reset and logic parts of flip-flop descriptions.	HDL Coder uses separate for loops in the reset and logic parts of flip-flop descriptions.	No action required.

2.I.C Exit Constraints

Rule / Severity	Message	Problem	Recommendations
2.I.C.1 Error	VHDL: Exit or next statement must not be used in a for loop.	The generated code contains for loops only when HDL Coder knows the number of iterations. When the loop is executing, HDL Coder does not exit from the for loop,	No action required.

2.J Guidelines for Operator Description

2.J.A Comparison and Precedence Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.A.4a-c Message	Verilog: Signals must not be compared with X or Z, or values containing X or Z.	By default, HDL Coder does not generate code that contains these constructs. If your Simulink model design uses Constant blocks with Architecture set to Logic Value and uses these constructs, the coder displays this message.	Update your Simulink model design so that the Constant blocks do not use these constructs when Architecture is set to Logic Value . Alternatively, change the Architecture to Constant .
2.J.A.4v Error	Verilog/VHDL: Do not assign X except for the others clause of case statements.	By default, HDL Coder does not use X in the others clause of case statements. In certain cases, if the generated code does not comply with 2.J.A.4a-c , HDL Coder can assign X in the others clause.	Update your Simulink model design so that the generated HDL code does not use constructs that rule 2.J.A.4a-c specifies.
2.J.A.5-6 Warning	Verilog: Do not use values containing 'X' or 'Z'.	If your design uses unknown or high-impedance constants, HDL Coder displays a warning.	Update your Simulink model or MATLAB algorithm so that there are no high-impedance constants.
	VHDL: Do not use values including 'X', 'Z', 'U'-'', 'W', 'H', 'L', or constants that contain the values 'X', 'Z', 'U'-'', 'W', 'H', 'L'.		
2.J.A.7-8 Message	Verilog: Do not use RAM output signals for a conditional expression of if statements, or selection expression of case statements that assign 'x' in the default clause.	By default, HDL Coder complies with this rule. If your Simulink model uses RAM output signals with a Switch or Multiport switch block, the generated HDL code can use these constructs.	Update your Simulink model so that there are no RAM output signals to Switch or Multiport switch blocks.

2.J.B Vector Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.B.3 Message	Verilog/VHDL: Do not perform logical negation on vectors.	HDL Coder does not perform logical negation on vectors.	No action required.

2.J.C Relational Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.C.1-6 Error	Verilog/VHDL: Bitwidths of operands of a relational or logical operator must match.	HDL Coder ensures that the data types of the operands match in a relational or logical expression.	No action required.
	Verilog/VHDL: Bitwidths should be specified for conditional expression.		

2.J.D Signed Signal, Data Type Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.D.3-5 Warning	Verilog/VHDL: Take care when assigning integer to reg or wire, and when comparing negative value reg and integer variables. Integer objects must not be assigned negative values.	HDL Coder complies with this rule.	No action required.
2.J.D.6 Warning	VHDL: Signed data type must be used in signed operation and std_logic_vector calling std_logic_unsigned package must be used in unsigned operation.	HDL Coder complies with this rule.	No action required.
2.J.D.8 Warning	VHDL: Function To_stdlogicvector should not be used in the design.	HDL Coder does not use the function To_stdlogicvector in the code.	No action required.

2.J.E Number of Operator Repetition Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.E.5 Warning	Verilog: Do not describe arithmetic operators with conditional operators(?) in assign statement.	HDL Coder complies with this rule.	No action required.

2.J.F Precision Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.F.5 Warning	Verilog/VHDL: Large multipliers must not be described using the multiplication operator with RTL.	The generated HDL code contains a multiplication operator (*) where the output of the multiplication has a bitwidth of 16 or greater.	<p>In your design, implement multiplications by using a shift-and-add algorithm, or ensure that the data size of the output of a multiplication does not require a bitwidth of 16 or greater.</p> <p>You can customize this rule by using the <code>MultiplierBitWidth</code> property of the HDL coding standard customization object.</p>

2.J.G Common Sub-Expression Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.G.2 Warning	Verilog/VHDL: common operational expressions should be described separately.	HDL Coder identifies the common operational expressions and describes them separately.	No action required.

2.J.H Division Operator Constraints

Rule / Severity	Message	Problem	Recommendations
2.J.H.1 Message	Verilog/VHDL: Do not use arithmetic and logical expressions in the right and left sides of the division or modulus operator.	HDL Coder homogenizes the division operator into a separate statement and complies with this rule.	No action required.
2.J.H.2-3 Message	Verilog/VHDL: Keep the left side of the division or modulus operator within 12 bits. If right side of the division or modulus operator is not a power of two, keep it within 8 bits.	In your design, the left side of the modulus or division operation is greater than 12 bits, or the right side is not a power of two and greater than eight bits.	Update your design so that the number of bits in the operands of the division or modulus operation are within the bounds that the rule specifies.

2.K Guidelines for Finite State Machine Description

2.K.A State Transition Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.A.4 Warning	Verilog/VHDL: Number of states of an FSM should be within 40.	Your model design contains a Stateflow Chart or State Transition Table that uses more than 40 states.	Update your model design so that there are not more than 40 states.

2.K.C Logic Separation Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.C.1 Reference	Verilog/VHDL: Ensure that sequential and combinational parts of an FSM are in separate always block.	By default, HDL Coder puts the sequential and combinational parts of a Finite State Machine (FSM) in separate always blocks.	No action required.

2.K.E Encoding Constraints

Rule / Severity	Message	Problem	Recommendations
2.K.E.2 Warning	VHDL: Do not assign state encoding by attaching attributes to the state variable which is declared as a type.	HDL Coder does not attach attributes to state variables in the generated code.	No action required.

See Also

Properties

HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-9
- “RTL Design Methodology Guidelines” on page 26-41

RTL Design Methodology Guidelines

In this section...

- “3.A Guidelines for Creating Function Libraries” on page 26-41
- “3.B Guidelines for Using Function Libraries” on page 26-42
- “3.C Guidelines for Test Facilitation Design” on page 26-43

HDL Coder conforms to the following RTL design methodology guidelines, and checks for modeling constructs that violate these rules. HDL Coder reports potential rule violations in the HDL coding standard report. To avoid these violations, see the rule recommendations.

3.A Guidelines for Creating Function Libraries

3.A.C Signal, Port Constraints - I

Rule / Severity	Message	Problem	Recommendations
3.A.C.1 Warning	Verilog: The order of module port declarations and instance port connections lists should be same as the order in the module port map.	HDL Coder preserves the order of module port declarations and instance port connections as they appear in the original Simulink DUT.	No action required.
3.A.C.4a Message	Verilog/VHDL: Define only one port or signal per line in I/O, reg, and wire declaration.	HDL Coder complies with this rule.	No action required.

3.A.D Signal, Port Constraints - II

Rule / Severity	Message	Problem	Recommendations
3.A.D.4-5 Warning	Verilog/VHDL: Multiple assignments should not be made in one line. Verilog/VHDL: The maximum number of characters in one line should not be more than N.	The generated HDL code contains multiple assignments in one line or lines greater than N characters. You have a name or identifier in your original design that contains more than N characters.	Shorten names in your design that are longer than N characters. You can also customize N by using the <code>LineLength</code> property of the HDL coding standard customization object. HDL Coder folds the long lines in the design only so far as the HDL code syntax is not broken.

3.A.F Generic Usage Constraints

Rule / Severity	Message	Problem	Recommendations
3.A.F.1 Reference	Verilog: Generic should be used in conditional expression of if generate statement.	HDL Coder does not generate if-generate statements, but can generate for-generate statements in the generated HDL code.	No action required.

3.B Guidelines for Using Function Libraries

3.B.B Parameters, Constant Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.B.2b-4 Message	Verilog: Define macros should be read using include files. Include files must be specified with more than 1 level higher relative path.	HDL Coder does not generate macros in the HDL code.	No action required.
3.B.B.5-7 Message	Verilog: Text macros should not be nested, and constants should be defined using parameters only.	HDL Coder does not generate macros in the HDL code.	No action required.

3.B.C Port Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.C.1 Message	Verilog/VHDL: Port/ Generic connections in instantiations must be made by named association rather than position association.	HDL Coder preserves the association of ports, so that it complies with this rule.	No action required.
3.B.C.2 Message	Verilog: Bit-width of the component port and its connected net must match.	HDL Coder enforces type and bit-width matching, so that it complies with this rule.	No action required.
3.B.C.3 Message	VHDL: Do not use entity instantiation in the design.	HDL Coder does not use entity instantiation in the design. The generated HDL code is generic and reusable.	No action required.

3.B.D Generic Constraints

Rule / Severity	Message	Problem	Recommendations
3.B.D.1 Error	Verilog/VHDL: Non-integer type used in the declaration of a generic may be unsynthesizable.	The generated HDL code contains a noninteger data type.	If you have floating-point data types in your design, you can map them to HDL Coder native floating-point libraries so that the generated code does not use floating-point data types. Alternatively, modify your design so that it does not use floating-point data types. You can disable this rule checking by using the <code>NonIntegerTypes</code> property of the HDL coding standard customization object.
3.B.D.3 Error	Verilog: Do not use <code>defparam</code> statements.	HDL Coder complies with this rule.	No action required.

3.C Guidelines for Test Facilitation Design

3.C.A Clock Constraints - I

Rule / Severity	Message	Problem	Recommendations
3.C.A.1-4 Error	Verilog/VHDL: Internal clocks and asynchronous sets/resets must be controllable from external pins.	In the generated HDL code, you can control clocks from external pins. If you have a triggered subsystem and enable <code>TriggerAsClock</code> , then the trigger signal becomes a clock signal that you can control from external pins. For reset signals that you model in Simulink, the generated VHDL code can have a load port, which is a primary input in the generated code.	To avoid this rule violation, disable the <code>TriggerAsClock</code> .

3.C.B Black Box Constraints

Rule / Severity	Message	Problem	Recommendations
3.C.B.3 Error	Verilog/VHDL: Do not connect the outputs of a black box to clock, reset, or tristate enable pins.	HDL Coder connects the clock bundle to the entity or blackbox and does not modify it, so the generated code complies with this rule.	No action required.

3.C.C Clock Constraints - II

Rule / Severity	Message	Problem	Recommendations
3.C.C.1 Error	Verilog/VHDL: A clock must not be connected to the D input of a flip-flop.	HDL Coder does not use clock as data.	No action required.

3.C.F Clock Constraints - III

Rule / Severity	Message	Problem	Recommendations
3.C.F.2 Error	Verilog/VHDL: Do not mix clock and reset lines.	HDL Coder connects the clock bundle to the entity or blackbox and does not modify it, so the generated code complies with this rule.	No action required.

See Also

Properties

HDL Coding Standard Customization

Related Examples

- “Generate an HDL Coding Standard Report from MATLAB” on page 5-25
- “Generate an HDL Coding Standard Report from Simulink” on page 26-5

More About

- “HDL Coding Standard Report” on page 26-2
- “Basic Coding Practices” on page 26-9
- “RTL Description Techniques” on page 26-18

Generate an HDL Lint Tool Script

You can generate a lint tool script to use with a third-party lint tool to check your generated HDL code.

HDL Coder can generate Tcl scripts for the following lint tools:

- Ascent Lint
- HDL Designer
- Leda
- SpyGlass
- Custom

If you specify one of the supported third-party lint tools, you can either generate a default tool-specific script, or customize the script by specifying the initialization, command, and termination names as a character vector. If you want to generate a script for a custom lint tool, you must specify the initialization, command, and termination names.

HDL Coder writes the initialization, command, and termination names to a Tcl script that you can use to run the third-party tool.

How to Generate an HDL Lint Tool Script

Using the Configuration Parameters Dialog Box

- 1 In the Configuration Parameters dialog box, select **HDL Code Generation > EDA Tool Scripts**.
- 2 Select the **Lint script** option.
- 3 For **Choose lint tool**, select **Ascent Lint**, **HDL Designer**, **Leda**, **SpyGlass**, or **Custom**.
- 4 Optionally, enter text to customize the **Lint initialization**, **Lint command**, and **Lint termination** strings. For a custom tool, specify these fields.

After you generate code, the message window shows a link to the lint tool script.

Using the Command Line

To generate an HDL lint tool script from the command line, set the **HDLLintTool** parameter to **AscentLint**, **HDLDesigner**, **Leda**, **SpyGlass**, or **Custom** using **makehdl** or **hdlset_param**.

To disable HDL lint tool script generation, set the **HDLLintTool** parameter to **None**.

For example, to generate HDL code and a default SpyGlass lint script for a DUT subsystem, **sfir_fixed\symmetric_fir**, enter the following:

```
makehdl('sfir_fixed/symmetric_fir','HDLLintTool','SpyGlass')
```

After you generate code, the message window shows a link to the lint tool script.

To generate an HDL lint tool script with custom initialization, command, and termination names, use the **HDLLintTool**, **HDLLintInit**, **HDLLintTerm**, and **HDLLintCmd** parameters.

For example, you can use the following command to generate a custom Leda lint script for a DUT subsystem, `sfir_fixed\symmetric_fir`, with custom initialization, command, and termination names:

```
makehdl('sfir_fixed/symmetric_fir','HDLLintTool','Leda',...
    'HDLLintInit','myInitialization','HDLLintCmd','myCommand %s',...
    'HDLLintTerm','myTermination')
```

Custom Lint Tool Command Specification

If you want to generate a lint tool script for a custom lint tool, you must use `%s` as a placeholder for the HDL file name in the generated Tcl script.

Specify the **Lint command** or **HDLLintCmd** using the following format:

```
hdlset_param ('HDLLintCmd', 'custom_lint_tool_command -option1 -option2 %s')
```

For example, to set **HDLLintCmd**, where the lint command is *custom_lint_tool_command -option1 -option2*, at the command line, enter:

```
hdlset_param ('HDLLintCmd', 'custom_lint_tool_command -option1 -option2 %s')
```

Interfacing Subsystems and Models to HDL Code

- “Model Referencing for HDL Code Generation” on page 27-2
- “Generate Black Box Interface for Subsystem” on page 27-4
- “Generate Black Box Interface for Referenced Model” on page 27-8
- “Integrate Custom HDL Code Using DocBlock” on page 27-10
- “Customize Black Box or HDL Cosimulation Interface” on page 27-12
- “Specify Bidirectional Ports” on page 27-15
- “Generate Reusable Code for Atomic Subsystems” on page 27-17
- “Scalarization of Vector Ports in Generated VHDL Code” on page 27-24
- “Create a Xilinx System Generator Subsystem” on page 27-27
- “Create an Altera DSP Builder Subsystem” on page 27-29
- “Using Altera DSP Builder Advanced Blockset with HDL Coder” on page 27-31
- “Using Xilinx® System Generator for DSP with HDL Coder™” on page 27-36
- “Choose a Test Bench for Generated HDL Code” on page 27-39
- “Generate a Cosimulation Model” on page 27-41
- “HDL Verifier Cosimulation Model Generation in HDL Coder™” on page 27-57
- “Verify HDL Design Using SystemVerilog DPI Test Bench” on page 27-79
- “Pass-Through and No-Op Implementations” on page 27-84
- “Synchronous Subsystem Behavior with the State Control Block” on page 27-85
- “Using the State Control block to generate more efficient code with HDL Coder™” on page 27-91
- “Resettable Subsystem Support in HDL Coder™” on page 27-97

Model Referencing for HDL Code Generation

In this section...

- “Benefits of Model Referencing for Code Generation” on page 27-2
- “How To Generate Code for a Referenced Model” on page 27-2
- “Generate Code for Model Arguments” on page 27-3
- “Generate Comments” on page 27-3
- “Limitations” on page 27-3

Benefits of Model Referencing for Code Generation

Model referencing in your DUT subsystem enables you to:

- Partition a large design into a hierarchy of smaller designs for reuse, modular development, and accelerated simulation.
- Incrementally generate and test code.

HDL Coder incrementally generates code for referenced models according to the **Configuration Parameters dialog box > Model Referencing pane > Rebuild** options.

However, HDL Coder treats **If any changes detected** and **If any changes in known dependencies detected** as the same. For example, if you set **Rebuild** to either **If any changes detected** or **If any changes in known dependencies detected**, HDL Coder regenerates code for referenced models only when the referenced models have changed.

How To Generate Code for a Referenced Model

By default, “Generate VHDL code for model references into a single library” on page 17-31 is enabled. The VHDL code is generated in a single library instead of separate libraries. In this case, set the **ScalarizePorts** property to **off** before generating HDL code.

When generating code, if you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the **ScalarizePorts** property to generate non-conflicting port definitions. For more information, see “Scalarize ports” on page 17-42.

You can generate HDL code for the referenced model using the UI or the command line.

Using the UI

- 1 Right-click the Model block and select **HDL Code > HDL Block Properties**.
- 2 For **Architecture**, select **ModelReference**.
- 3 Generate HDL code from your DUT subsystem.

Using the Command Line

- 1 Set the **Architecture** property of the Model block to **ModelReference**. For example, for a DUT subsystem, `mydut`, that includes a model reference, `referenced_model`, enter this command:

```

hdlset_param ('mydut/referenced_model', ...
    'Architecture', 'ModelReference');

2 Generate HDL code for your DUT subsystem.

makehdl ('mydut');

```

Generate Code for Model Arguments

To generate a single Verilog module or VHDL entity for instances of a referenced model with different model argument values, see “Generate Parameterized Code for Referenced Models” on page 10-20.

Generate Comments

If you enter text in the Model Block Properties dialog box **Description** field, HDL Coder generates a comment in the HDL code.

Limitations

- Model block must have default values for the Block parameters.
- Model block cannot be a masked subsystem.
- Multiple model references that refer to the same model must have the same HDL block properties.
- Referenced models cannot be protected models.
- Hierarchical distributed pipelining must be disabled.

HDL Coder cannot move registers across a model reference. Therefore, referenced models can inhibit these optimizations:

- Distributed pipelining
- Constrained output pipelining
- Streaming

When you have model references and generate HDL code, the generated model, validation model, and cosimulation model can fail to compile or simulate. To fix compilation or simulation errors, make sure that the referenced models are loaded or are on the search path.

The coder can apply the resource sharing optimization to share referenced model instances. However, you can apply this optimization only when all model references that point to the same referenced model have the same rate after optimizations and rate propagation. The model reference final rate may differ from the original rate, but all model references that point to the same referenced model must have the same final rate.

Generate Black Box Interface for Subsystem

In this section...

- “What Is a Black Box Interface?” on page 27-4
- “Requirements” on page 27-4
- “Generate a Black Box Interface for a Subsystem” on page 27-4
- “Generate Code for a Black Box Subsystem Implementation” on page 27-6

What Is a Black Box Interface?

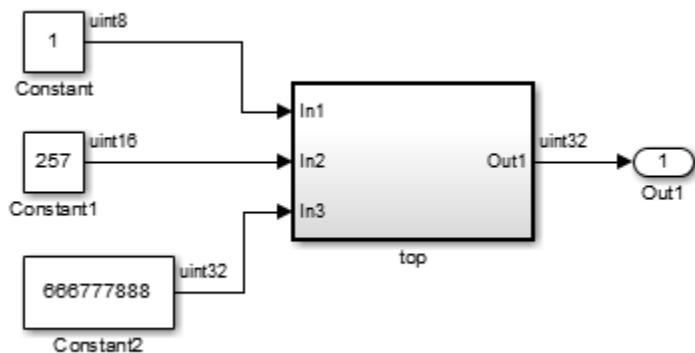
A *black box* interface for a subsystem is a generated VHDL component or Verilog module that includes only the HDL input and output port definitions for the subsystem. By generating such a component, you can use a subsystem in your model to generate an interface to existing manually written HDL code, third-party IP, or other code generated by HDL Coder.

Requirements

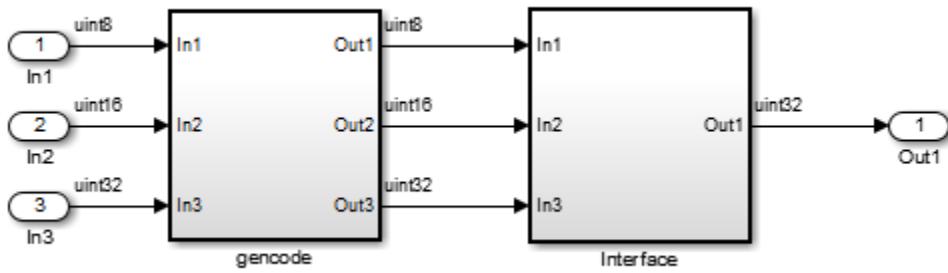
- The black box implementation is available only for subsystem blocks below the level of the DUT. Virtual and atomic subsystem blocks of custom libraries that are below the level of the DUT also work with black box implementations.
- You can generate at most one clock port and one clock enable port for a black box subsystem. Therefore, the black box subsystem must be single-rate even if it is in a multirate DUT.

Generate a Black Box Interface for a Subsystem

To generate the interface, select the **BlackBox** implementation for one or more Subsystem blocks. Consider the following model that contains a subsystem **top**, which is the device under test.



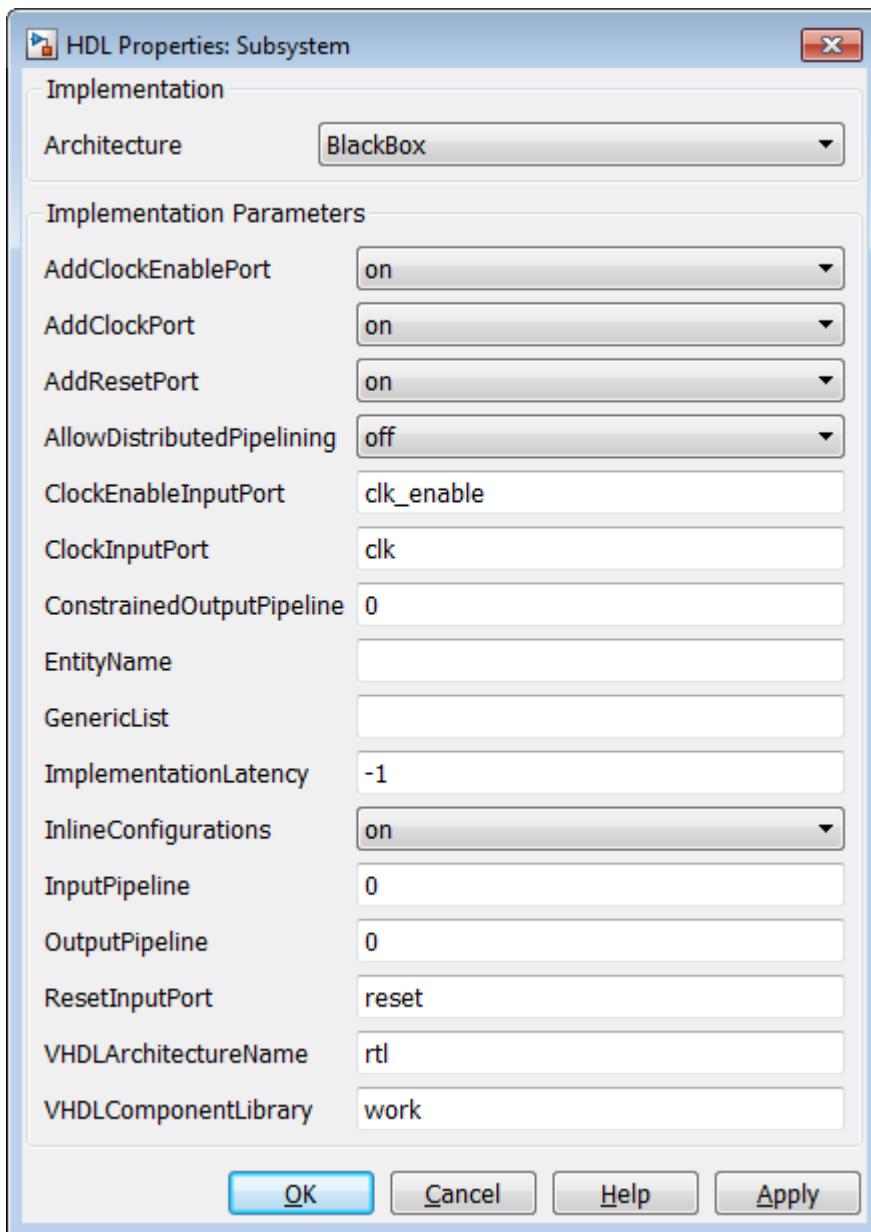
The subsystem **top** contains two lower-level subsystems:



Suppose that you want to generate HDL code from **top**, with a black box interface from the **Interface** subsystem. To specify a black box interface:

- 1 Right-click the **Interface** subsystem and select **HDL Code > HDL Block Properties**.
The HDL Properties dialog box appears.
- 2 Set **Architecture** to **BlackBox**.

The following parameters are available for the black box implementation:



The HDL block parameters available for the black box implementation enable you to customize the generated interface. See “Customize Black Box or HDL Cosimulation Interface” on page 27-12 for information about these parameters.

- 3 Change parameters as desired, and click **Apply**.
- 4 Click **OK** to close the HDL Properties dialog box.

Generate Code for a Black Box Subsystem Implementation

When you generate code for the DUT in the `ex_blackbox_subsys` model, the following messages appear:

```
>> makehdl('ex_blackbox_subsys/top')
### Generating HDL for 'ex_blackbox_subsys/top'
```

```

### Starting HDL Check.
### HDL Check Complete with 0 errors, 0 warnings and 0 messages.

### Begin VHDL Code Generation
### Working on ex_blackbox_subsys/top/gencode as hdlsrc\gencode.vhd
### Working on ex_blackbox_subsys/top as hdlsrc\top.vhd
### HDL Code Generation Complete.

```

In the progress messages, observe that the `gencode` subsystem generates a separate file, `gencode.vhd`, for its VHDL entity definition. The `Interface` subsystem does not generate such a file. The interface code for this subsystem is in `top.vhd`, generated from `ex_blackbox_subsys/top`. The following code listing shows the component definition and instantiation generated for the `Interface` subsystem.

```

COMPONENT Interface
  PORT( clk : IN std_logic;
        clk_enable : IN std_logic;
        reset : IN std_logic;
        In1 : IN std_logic_vector(7 DOWNTO 0); -- uint8
        In2 : IN std_logic_vector(15 DOWNTO 0); -- uint16
        In3 : IN std_logic_vector(31 DOWNTO 0); -- uint32
        Out1 : OUT std_logic_vector(31 DOWNTO 0) -- uint32
      );
END COMPONENT;
...
u_Interface : Interface
  PORT MAP( clk => clk,
            clk_enable => enb,
            reset => reset,
            In1 => gencode_out1, -- uint8
            In2 => gencode_out2, -- uint16
            In3 => gencode_out3, -- uint32
            Out1 => Interface_out1 -- uint32
          );
enb <= clk_enable;
ce_out <= enb;
Out1 <= Interface_out1;

```

By default, the black box interface generated for subsystems includes clock, clock enable, and reset ports. “Customize Black Box or HDL Cosimulation Interface” on page 27-12 describes how you can rename or suppress generation of these signals, and customize other aspects of the generated interface.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-12
- “Generate Black Box Interface for Referenced Model” on page 27-8
- “Integrate Custom HDL Code Using DocBlock” on page 27-10

Generate Black Box Interface for Referenced Model

In this section...

["When to Generate a Black Box Interface" on page 27-8](#)

["How to Generate a Black Box Interface" on page 27-8](#)

["Caveats and Limitations" on page 27-8](#)

When to Generate a Black Box Interface

Specify a black box implementation for the Model block when you already have legacy or manually-written HDL code. HDL Coder generates the HDL code that is required to interface to the referenced HDL code.

Code is generated with the following assumptions:

- Every HDL entity or module requires clock, clock enable, and reset ports. Therefore, these ports are defined for each generated entity or module.
- Use of Simulink data types is assumed. For VHDL code, port data types are assumed to be `STD_LOGIC` or `STD_LOGIC_VECTOR`.

If you want to generate code for a multirate, multiclock DUT that includes a referenced model, see ["Model Referencing for HDL Code Generation" on page 27-2](#).

How to Generate a Black Box Interface

To instantiate an HDL wrapper, or black box interface, for a referenced model:

- 1 Right-click the Model block and select **HDL Code > HDL Block Properties**.

In the HDL Block Properties dialog box:

- For **Architecture**, select **BlackBox**.
- Customize the ports and other implementation parameters. To learn more about customizing the ports, see ["Customize Black Box or HDL Cosimulation Interface" on page 27-12](#).

- 2 Generate HDL code for your DUT subsystem.

Caveats and Limitations

- If you run the `checkhdl` function to check the compatibility of your model for HDL code generation, the function does not check the port data types within the referenced model.
- If you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the `ScalarizePorts` property to generate nonconflicting port definitions. For more information, see ["Scalarize ports" on page 17-42](#).

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-12
- “Generate Black Box Interface for Subsystem” on page 27-4
- “Integrate Custom HDL Code Using DocBlock” on page 27-10

Integrate Custom HDL Code Using DocBlock

In this section...

- “When To Use DocBlock to Integrate Custom Code” on page 27-10
- “How To Use DocBlock to Integrate Custom Code” on page 27-10
- “Restrictions” on page 27-10
- “Example” on page 27-11

You can use one or more DocBlock blocks to integrate custom HDL code into your design.

When To Use DocBlock to Integrate Custom Code

If you want to keep the HDL code with your model, instead of as a separate file, use a DocBlock to integrate custom HDL code. The text in the DocBlock is your custom VHDL or Verilog code.

You include each DocBlock that contains custom HDL code by placing it in a black box subsystem, and including the black box subsystem in your DUT. One HDL file is generated per black box subsystem.

Alternatives for Custom Code Integration

If you want to keep your custom HDL code separate from your model, such as when the custom code is IP or a library from a third party, use a black box subsystem on page 27-4 or black box model reference on page 27-8.

How To Use DocBlock to Integrate Custom Code

- 1 In your DUT, at any level of hierarchy, add a Subsystem block.
- 2 For the Subsystem block, in the HDL Block Properties dialog box:
 - Set **Architecture** to BlackBox.
 - Customize the black box subsystem interface so that it matches your custom HDL code interface. To learn more about customizing the black box interface, see “Customize Black Box or HDL Cosimulation Interface” on page 27-12.
- 3 In the subsystem, add a DocBlock block.
- 4 For the DocBlock, in the HDL Block Properties dialog box:
 - Set **Architecture** to HDLText.
 - Set **TargetLanguage** to your target language, either Verilog or VHDL¹.
- 5 In the DocBlock, enter the HDL code for your custom Verilog module or VHDL entity.

The language must match the DocBlock **TargetLanguage** setting.

Restrictions

- The black box subsystem that contains the DocBlock cannot be the top-level DUT.

- You can have a maximum of two DocBlock blocks in the black box subsystem. If you have two DocBlock blocks, one must have **TargetLanguage** set to VHDL, and the other must have **TargetLanguage** set to Verilog.

When generating code, HDL Coder only integrates the code from the DocBlock that matches the target language for code generation.

Example

The `hdlcoderIncludeCustomHdlUsingDocBlockExample` model shows how to integrate custom VHDL and Verilog code into your design with the DocBlock block.

See Also

More About

- “Customize Black Box or HDL Cosimulation Interface” on page 27-12
- “Generate Black Box Interface for Subsystem” on page 27-4
- “Generate Black Box Interface for Referenced Model” on page 27-8

Customize Black Box or HDL Cosimulation Interface

You can customize port names and set attributes of the external component when you generate an interface from the following blocks:

- Model with black box implementation
- Subsystem with black box implementation
- HDL Cosimulation

Interface Parameters

Open the HDL Block Properties dialog box to see the interface generation parameters.

The following table summarizes the names, value settings, and purpose of the interface generation parameters.

Note You cannot specify clock, reset, and clock enable signals explicitly in your Simulink model by using the **AddClockEnablePort**, **AddClockPort**, and **AddResetPort** parameters. Instead, use these parameters to add a clock, reset, or clock enable port in the generated HDL code.

Parameter Name	Values	Description
AddClockEnablePort	on off Default: on	If on, add a clock enable input port to the interface generated for the block. The name of the port is specified by ClockEnableInputPort .
AddClockPort	on off Default: on	If on, add a clock input port to the interface generated for the block. The name of the port is specified by ClockInputPort .
AddResetPort	on off Default: on	If on, add a reset input port to the interface generated for the block. The name of the port is specified by ResetInputPort .
AllowDistributedPipelining	on off Default: off	If on, allow HDL Coder to move registers across the block, from input to output or output to input.
ClockEnableInputPort	Default: clk_enable	Specifies HDL name for block's clock enable input port.
ClockInputPort	Default: clk	Specifies HDL name for block's clock input signal.
ConstrainedOutputPipeline	Default: 0	Specifies the number of delays that you want the code generator to insert at the output of the interface by redistributing existing delays in your design.

Parameter Name	Values	Description
EntityName	Default: Entity name string is derived from the block name, and modified when necessary to generate a legal VHDL entity name.	Specifies VHDL <code>entity</code> or Verilog <code>module</code> name generated for the block.
GenericList	<p>Pass a cell array variable that contains cell arrays each with two or three strings, or enter a cell array of cell arrays that each contain two or three strings. The strings represent the name, value, and optional data type of a VHDL generic or Verilog parameter. The default data type is <code>integer</code>.</p> <p>Default: none</p>	<p>Specifies a list of VHDL <code>generic</code> or Verilog <code>parameter</code> name-value pairs, each with an optional data type specification, to pass to a subsystem with a <code>BlackBox</code> implementation.</p> <p>For example, in the HDL Block Properties dialog box, enter <code>{'name', 'value', 'type'}</code>, or, if the data type is <code>integer</code>, enter <code>{'name', 'value'}</code>.</p> <p>To set <code>GenericList</code> using <code>hdlset_param</code>, at the command line, enter:</p> <pre>hdlset_param (blockname, 'GenericList', {'name', 'value', 'type'});</pre> <p>If the data type is <code>integer</code>, at the command line, enter:</p> <pre>hdlset_param (blockname, 'GenericList', {'name', 'value'});</pre>
ImplementationLatency	<p>-1 0 positive integer</p> <p>Default: -1</p>	<p>Specifies the additional latency of the external component in time steps, relative to the Simulink block.</p> <p>If 0 or greater, this value is used for delay balancing. Your inputs and outputs must operate at the same rate.</p> <p>If -1, latency is unknown. This disables delay balancing.</p>
InlineConfigurations (VHDL only)	<p>on off</p> <p>Default: If this parameter is unspecified, defaults to the value of the global <code>InlineConfigurations</code> property.</p>	If <code>off</code> , suppress generation of a configuration for the block, and require a user-supplied external configuration.

Parameter Name	Values	Description
InputPipeline	Default: 0	Specifies the number of input pipeline stages (pipeline depth) in the generated code.
OutputPipeline	Default: 0	Specifies the number of output pipeline stages (pipeline depth) in the generated code.
ResetInputPort	Default: <code>reset</code>	Specifies HDL name for block's reset input.
VHDLArchitectureName (VHDL only)	Default: <code>rtl</code>	Specifies RTL architecture name generated for the block. The architecture name is generated only if InlineConfigurations is on.
VHDLComponentLibrary (VHDL only)	Default: <code>work</code>	Specifies the library from which to load the VHDL component.

See Also

More About

- “Generate Black Box Interface for Subsystem” on page 27-4
- “Generate Black Box Interface for Referenced Model” on page 27-8
- “Integrate Custom HDL Code Using DocBlock” on page 27-10
- “Specify Bidirectional Ports” on page 27-15

Specify Bidirectional Ports

You can specify bidirectional ports for Subsystem blocks with black box implementation. In the generated code, the bidirectional ports have the Verilog or VHDL `inout` keyword.

In the FPGA Turnkey workflow, you can use the bidirectional ports to connect to external RAM.

In this section...

["Requirements" on page 27-15](#)

["How To Specify a Bidirectional Port" on page 27-15](#)

["Limitations" on page 27-15](#)

Requirements

- The bidirectional port must be a black box subsystem port.
- There must be no logic between the bidirectional port and the corresponding top-level DUT subsystem port. Otherwise, the generated code does not compile.

How To Specify a Bidirectional Port

To specify a bidirectional port using the UI:

- 1 In the black box Subsystem, right-click the Import or Outport block that represents the bidirectional port. Select **HDL Code** > **HDL Block Properties**.
- 2 For **BidirectionalPort**, select on.

To specify a bidirectional port at the command line, set the `BidirectionalPort` property to 'on' using `hdlset_param` or `makehdl`.

For example, suppose you have a model, `my_model`, that contains a DUT subsystem, `dut_subsys`, and the DUT subsystem contains a black box subsystem, `blackbox_subsys`. If `blackbox_subsys` has an Import, `input_A`, specify `input_A` as bidirectional by entering:

```
hdlset_param('mymodel/dut_subsys/blackbox_subsys/input_A','BidirectionalPort','on');
```

Limitations

- In the FPGA Turnkey workflow, in the **Target platform interfaces table**, you must map a bidirectional port to either `Specify FPGA Pin {'LSB', ..., 'MSB'}` or one of the other interfaces where the interface bitwidth exactly matches your bidirectional port bitwidth.

For example, you can map a 32-bit bidirectional port to the `Expansion Headers J6 Pin 2-64[0:31]` interface.

- You cannot generate a Verilog test bench if there is a bidirectional port within your DUT subsystem.
- HDL Coder does not support bidirectional ports for masked subsystems that use `BlackBox` as the **HDL Architecture**.
- Simulink does not support bidirectional ports, so you cannot simulate the bidirectional behavior in Simulink.

See Also

More About

- “Generate Black Box Interface for Subsystem” on page 27-4
- “Generate Black Box Interface for Referenced Model” on page 27-8
- “Integrate Custom HDL Code Using DocBlock” on page 27-10
- “Customize Black Box or HDL Cosimulation Interface” on page 27-12

Generate Reusable Code for Atomic Subsystems

In this section...

["Requirements for Generating Reusable Code for Atomic Subsystems" on page 27-17](#)

["Generate Reusable Code for Atomic Subsystems" on page 27-17](#)

["Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters" on page 27-19](#)

HDL Coder can detect atomic subsystems that are identical, or identical except for their mask parameter values, at any level of the model hierarchy, and generate a single reusable HDL module or entity. The reusable HDL code is generated as a single file and instantiated multiple times.

Requirements for Generating Reusable Code for Atomic Subsystems

To generate reusable HDL code for atomic subsystems:

- The **DefaultParameterBehavior** Simulink Configuration Parameter must be **Inlined**. You can set this parameter at the command line by using the `set_param` or `hdlsetup` function. To specify this setting in the Configuration Parameters dialog box, you must have Simulink Coder.

Note Using `hdlsetup` sets `InlineParams` property to `on`. Enabling this parameter is the same as setting `DefaultParameterBehavior` to `Inlined`. Setting `InlineParams` to `off` changes `DefaultParameterBehavior` value to `Tunable`.

- Logging functionality must not be used, such as signal logging or using blocks such as `To Workspace` or `To File`.
- The atomic subsystems must be identical, or identical except for their mask parameter values.
 - `MaskParameterAsGeneric` must be `on`. For more information, see "Generate parameterized HDL code from masked subsystem" on page 17-57.
 - Mask parameters must be nontunable. The code generator does not share atomic subsystems with mask parameters that are tunable.
 - Mask parameter data types cannot be `double` or `single`.
 - The tunable parameter must be used in only Constant or Gain blocks.
 - Port data types must match.

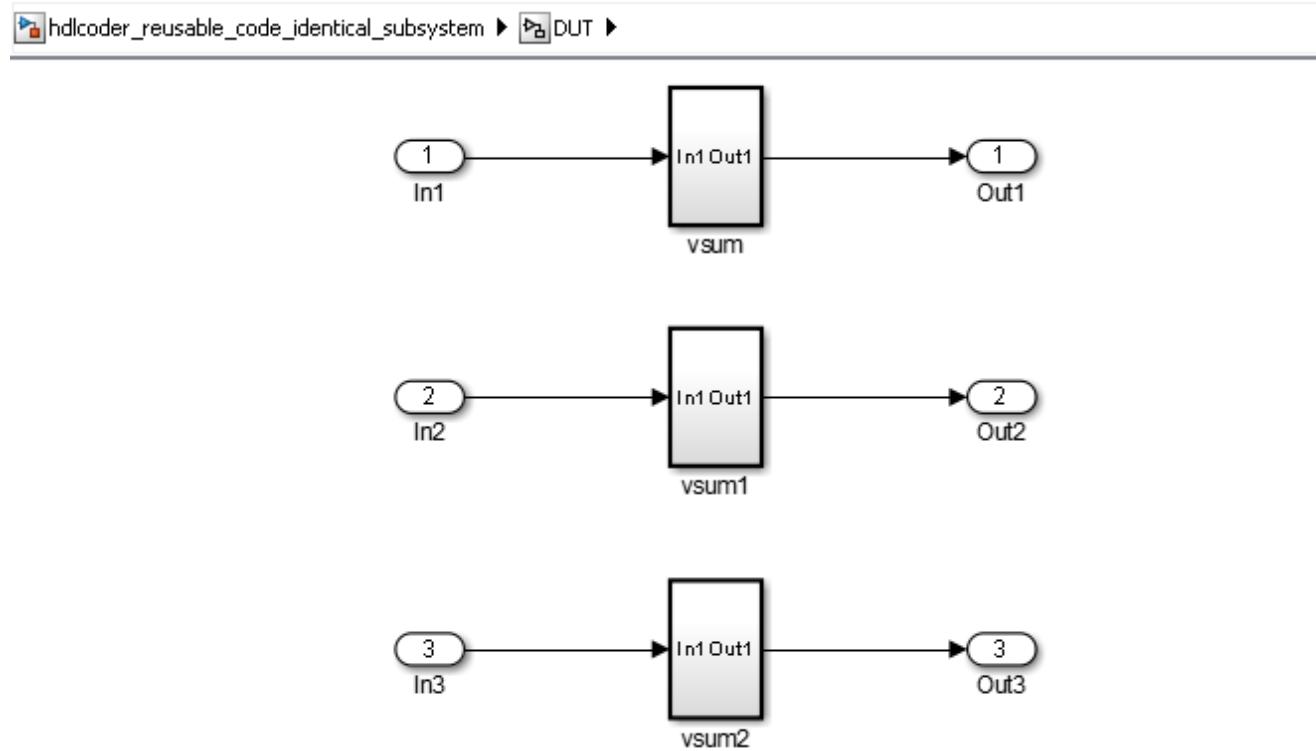
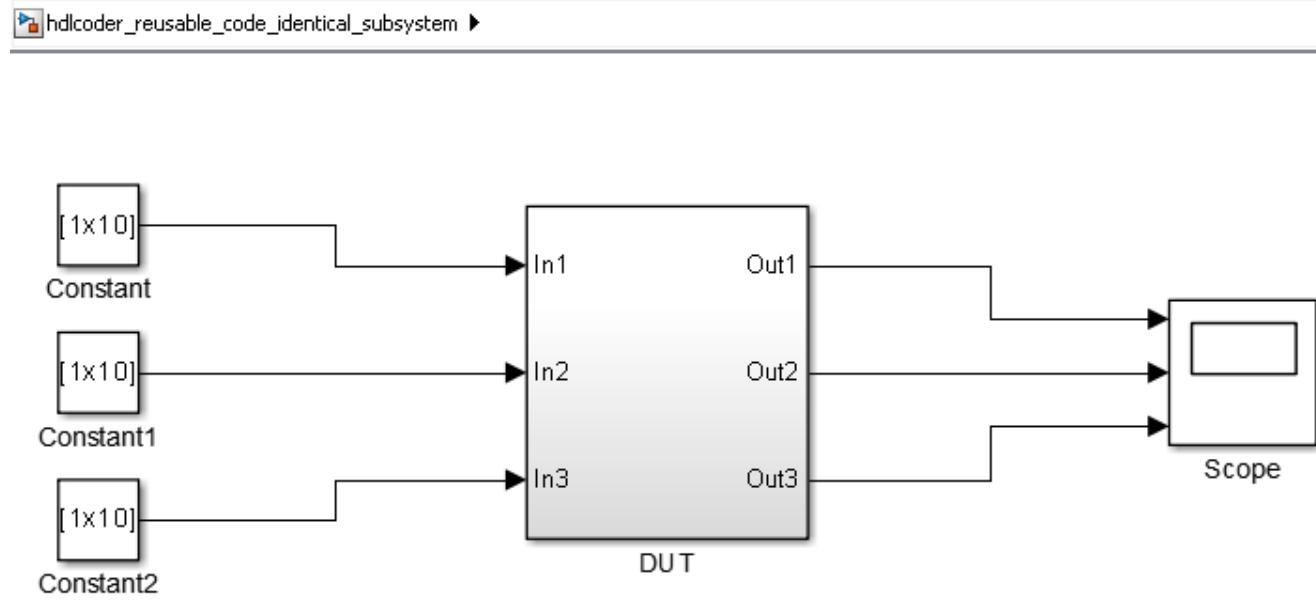
If you change the value of the tunable mask parameter, the output port data type can change. If one of the atomic subsystems has a different port data type, the code generated for that subsystem also differs.

Generate Reusable Code for Atomic Subsystems

If your design contains identical atomic subsystems, the coder generates one HDL module or entity for the subsystem and instantiates it multiple times.

Example

The `hdlcoder_reusable_code_identical_subsystem` model shows an example of a DUT subsystem containing three identical atomic subsystems.



HDL Coder generates a single VHDL file, `vsum.vhd`, for the three subsystems.

```
makehdl('hdlcoder_reusable_code_identical_subsystem/DUT')
```

```

### Generating HDL for 'hdlcoder_reusable_code_identical_subsystem/DUT'.
### Starting HDL check.
### Generating new validation model: gm_hdlcoder_reusable_code_identical_subsystem_vnl.
### Validation model generation complete.
### Begin VHDL Code Generation for 'hdlcoder_reusable_code_identical_subsystem'.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT/vsum/Sum of Elements as
### hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\Sum_of_Elements.vhd.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT/vsum as
### hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\Sum_of_Elements.vhd.
### Working on hdlcoder_reusable_code_identical_subsystem/DUT as
### hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\Sum_of_Elements.vhd.
### Generating package file hdl_prj\hdlsrc\hdlcoder_reusable_code_identical_subsystem\Sum_of_Elements.vhd.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_reusable_code_identical_subsystem' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

```

The generated code for the DUT subsystem, `DUT.vhd`, contains three instantiations of the `vsum` component.

```

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT vsum
  PORT( In1           : IN   vector_of_std_logic_vector16(0 TO 9); -- int16 [10]
        Out1          : OUT  std_logic_vector(19 DOWNTO 0) -- sfix20
      );
END COMPONENT;

-- Component Configuration Statements
FOR ALL : vsum
  USE ENTITY work.vsum(rtl);

-- Signals
SIGNAL vsum_out1      : std_logic_vector(19 DOWNTO 0); -- ufix20
SIGNAL vsum1_out1     : std_logic_vector(19 DOWNTO 0); -- ufix20
SIGNAL vsum2_out1     : std_logic_vector(19 DOWNTO 0); -- ufix20

BEGIN
  u_vsum : vsum
    PORT MAP( In1 => In1, -- int16 [10]
              Out1 => vsum_out1 -- sfix20
            );

  u_vsum1 : vsum
    PORT MAP( In1 => In2, -- int16 [10]
              Out1 => vsum1_out1 -- sfix20
            );

  u_vsum2 : vsum
    PORT MAP( In1 => In3, -- int16 [10]
              Out1 => vsum2_out1 -- sfix20
            );

  Out1 <= vsum_out1;
  Out2 <= vsum1_out1;
  Out3 <= vsum2_out1;
END rtl;

```

Generate Reusable Code for Atomic Subsystems with Tunable Mask Parameters

If your design contains atomic subsystems that are identical except for their tunable mask parameter values, you can generate one HDL module or entity for the subsystem. In the generated code, the module or entity is instantiated multiple times.

To generate reusable code for identical atomic subsystems, enable `MaskParameterAsGeneric` for the model. By default, `MaskParameterAsGeneric` is disabled.

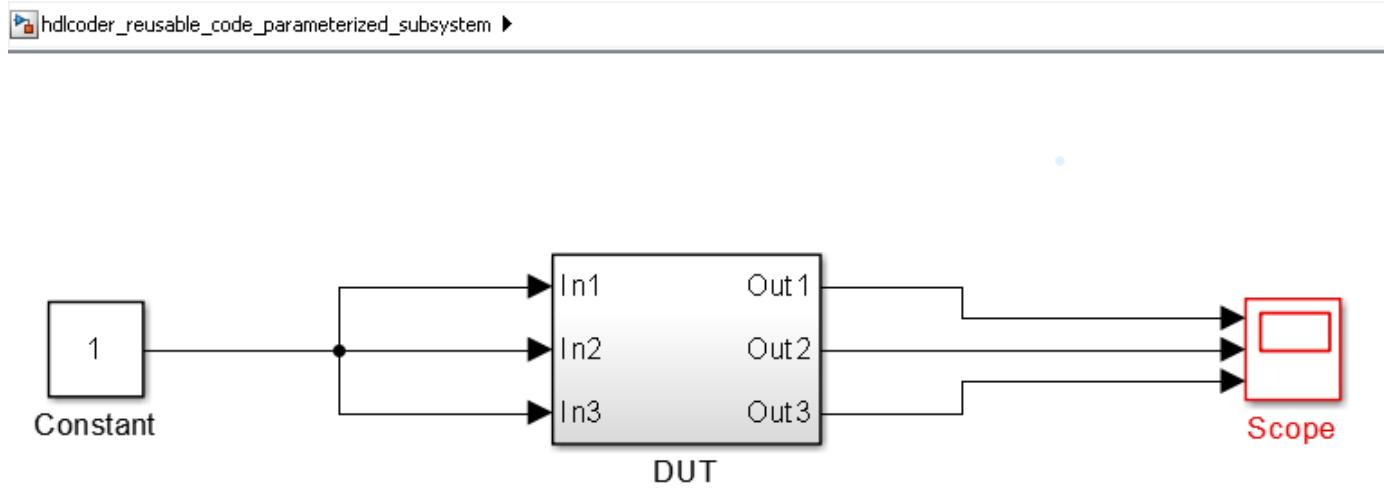
For example, to enable the generation of reusable code for the atomic subsystems with tunable parameters in the `hdlcoder_reusable_code_parameterized_subsystem` model, enter:

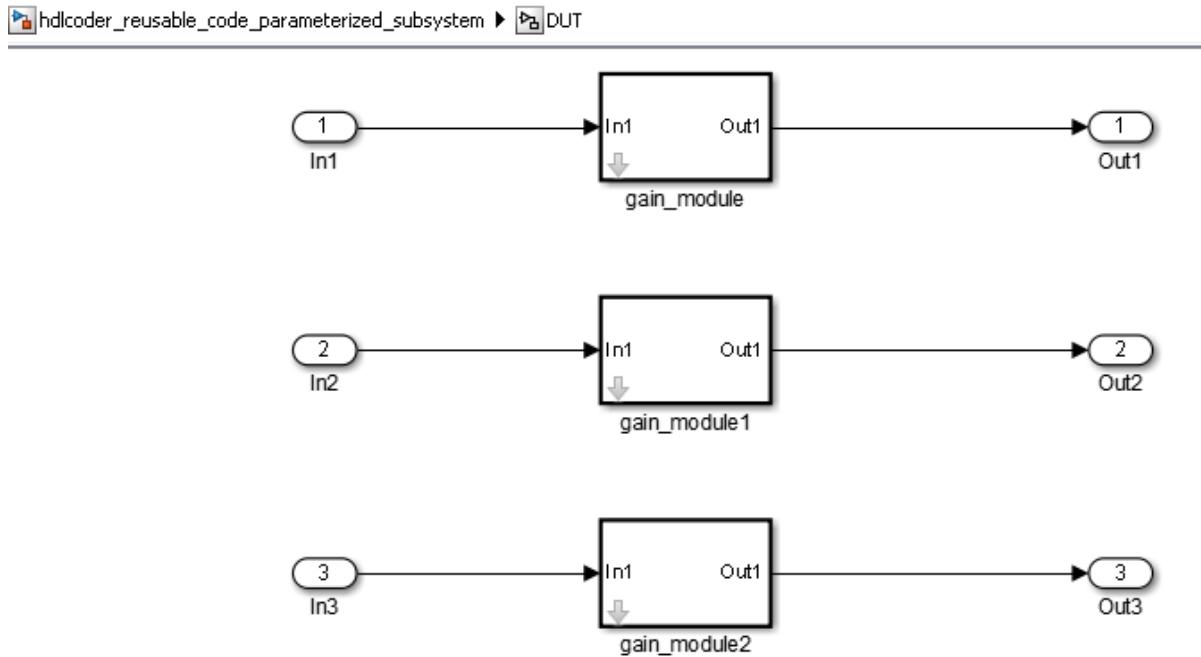
```
hdlset_param('hdlcoder_reusable_code_parameterized_subsystem','MaskParameterAsGeneric','on')
```

Alternatively, in the Configuration Parameters dialog box, in the **HDL Code Generation > Global Settings > Coding Style** tab, enable the **Generate parameterized HDL code from masked subsystem** option.

Example

The `hdlcoder_reusable_code_parameterized_subsystem` model shows an example of a DUT subsystem containing atomic subsystems that are identical except for their tunable mask parameter values.





In `hdlcoder_reusable_code_parameterized_subsystem/DUT`, the gain modules are subsystems with gain values represented by tunable mask parameters. Gain values are: 4 for `gain_module`, 5 for `gain_module1`, and 7 for `gain_module2`.

With `MaskParameterAsGeneric` enabled, HDL Coder generates a single source file, `gain_module.v`, for the three gain module subsystems.

```
makehdl('hdlcoder_reusable_code_parameterized_subsystem/DUT','MaskParameterAsGeneric','on',...
    'TargetLanguage','Verilog')
### Generating HDL for 'hdlcoder_reusable_code_parameterized_subsystem/DUT'.
### Starting HDL check.
### Begin Verilog Code Generation for 'hdlcoder_reusable_code_parameterized_subsystem'.
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT/gain_module as
hdlsrc\hdlcoder_reusable_code_parameterized_subsystem\gain_module.v.
### Working on hdlcoder_reusable_code_parameterized_subsystem/DUT as
hdlsrc\hdlcoder_reusable_code_parameterized_subsystem\DUT.v.
### Creating HDL Code Generation Check Report DUT_report.html
### HDL check for 'hdlcoder_reusable_code_parameterized_subsystem' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```

The generated code for the DUT subsystem, `DUT.v`, contains three instantiations of the `gain_module` component.

```
module DUT
(
    In1,
    In2,
    In3,
    Out1,
    Out2,
    Out3
);

input [7:0] In1; // uint8
input [7:0] In2; // uint8
input [7:0] In3; // uint8
output [31:0] Out1; // uint32
output [31:0] Out2; // uint32
```

```
output [31:0] Out3; // uint32

wire [31:0] gain_module_out1; // uint32
wire [31:0] gain_module1_out1; // uint32
wire [31:0] gain_module2_out1; // uint32

gain_module # (.myGain(4)
)
u_gain_module (.In1(In1), // uint8
               .Out1(gain_module_out1) // uint32
);

assign Out1 = gain_module_out1;

gain_module # (.myGain(5)
)
u_gain_module1 (.In1(In2), // uint8
                .Out1(gain_module1_out1) // uint32
);

assign Out2 = gain_module1_out1;

gain_module # (.myGain(7)
)
u_gain_module2 (.In1(In3), // uint8
                .Out1(gain_module2_out1) // uint32
);

assign Out3 = gain_module2_out1;

endmodule // DUT
```

In `gain_module.v`, the `myGain` Verilog parameter is generated for the tunable mask parameter.

```
module gain_module
(
  In1,
  Out1
);

input [7:0] In1; // uint8
output [31:0] Out1; // uint32

parameter [31:0] myGain = 4; // ufix32

wire [31:0] kconst; // ufix32
wire [39:0] Gain_mul_temp; // ufix40
wire [31:0] Gain_out1; // uint32

assign kconst = myGain;
assign Gain_mul_temp = kconst * In1;
assign Gain_out1 = Gain_mul_temp[31:0];

assign Out1 = Gain_out1;
endmodule // gain_module
```

See Also

More About

- “Generate parameterized HDL code from masked subsystem” on page 17-57

- “Generate parameterized HDL code from masked subsystem” on page 17-57
- “Generate Parameterized Code for Referenced Models” on page 10-20
- “Create and Add Tunable Parameter That Maps to DUT Ports” on page 10-17

Scalarization of Vector Ports in Generated VHDL Code

This example shows how to flatten the vector signals in a Simulink® model into a structure of scalar signals in the generated VHDL code.

Specify Scalarization of Vector Ports

When you generate Verilog code, by default, the vector signals are flattened into scalars by default. When you generate VHDL code, you can specify whether to flatten the vector signals on the entire model or at the DUT level. Flattening the vector ports only at the DUT level speeds up code generation especially for large models that have many vector inputs.

You can scalarize the vector ports into scalars when generating HDL code from MATLAB® and Simulink. For the MATLAB® to HDL workflow, in the **HDL Code Generation** task, on the **Clocks & Ports** tab, set **Scalarize ports** to **dutlevel** or **on**.

To scalarize the vector ports when generating HDL code for a Simulink model:

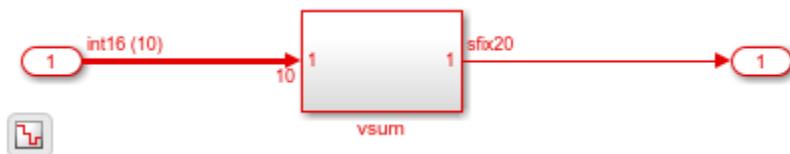
- In the Configuration Parameters dialog box, on the HDL Code Generation > Global Settings > Ports* tab, set **Scalarize Ports** parameter to **dutlevel** or **on**.
- In the HDL Workflow Advisor, on the **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Ports** tab, set **Scalarize Ports** parameter to **dutlevel** or **on**.
- At the MATLAB command prompt, set the **ScalarizePorts** property to **on** or **dutlevel** by using **hdlset_param** or **makehdl**.

See “Scalarize ports” on page 17-42.

Vector Sum Model

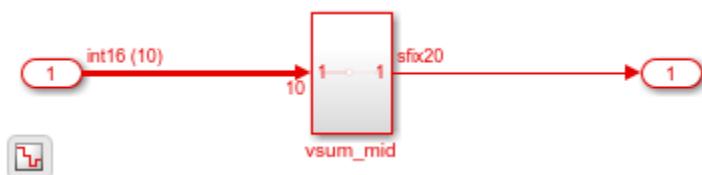
To see the flattening of vector ports, open the model `hdlcoder_hdlcoder_vector_sum_nested`. The model is driven by a vector input of width 10 and has a scalar output.

```
open_system('hdlcoder_vector_sum_nested')
set_param('hdlcoder_vector_sum_nested','SimulationCommand','update')
```



This model is the same as the `simplevectorsum` model and consists of a Subsystem, `vsum_mid`, inside the `vsum` subsystem.

```
open_system('hdlcoder_vector_sum_nested/vsum')
```



The `vsum_mid` subsystem contains a Sum of Elements block that is configured for vector summation. The model is configured to use the Tree implementation when generating HDL code for the Sum of Elements block within the `vsum` subsystem. This implementation is optimized for minimal latency, generates a tree-shaped structure of adders for the block.

```
open_system('hdlcoder_vector_sum_nested/vsum/vsum_mid')
```



Scalarize Vector Ports

By default, the `ScalarizePorts` property is off. HDL Coder™ generates a type definition and port declaration for the vector port `In1` as shown in this code.

```
PACKAGE simplevectorsum_pkg IS
  TYPE vector_of_std_logic_vector16 IS ARRAY (NATURAL RANGE <>)
    OF std_logic_vector(15 DOWNTO 0);
  TYPE vector_of_signed16 IS ARRAY (NATURAL RANGE <>) OF signed(15 DOWNTO 0);
END simplevectorsum_pkg;

...
ENTITY vsum IS
  PORT( In1      : IN   vector_of_std_logic_vector16(0 TO 9); -- int16 [10]
        Out1     : OUT  std_logic_vector(19 DOWNTO 0)  -- sfix20
      );
END vsum;
```

To scalarize the vector ports when generating HDL code, either set the **Scalarize ports** parameter in the Configuration Parameters dialog box or set the `ScalarizePorts` property to `on` or `dutlevel` with `hdlset_param` or `makehdl`. When you set `ScalarizePorts` to `dutlevel`, only the vector signals at the DUT are flattened into scalars. The scalars are input to the `vsim_mid` subsystem as vectors.

```
makehdl('hdlcoder_vector_sum_nested/vsum', 'ScalarizePorts', 'dutlevel')

ENTITY vsum IS
  PORT( In1_0      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_1      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_2      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_3      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_4      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_5      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_6      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_7      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_8      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        In1_9      : IN   std_logic_vector(15 DOWNTO 0); -- int16
        Out1       : OUT  std_logic_vector(19 DOWNTO 0)  -- sfix20
      );
END vsum;

ENTITY vsum_mid IS
  PORT( In1      : IN   vector_of_std_logic_vector16(0 TO 9); -- int16 [10]
        Out1     : OUT  std_logic_vector(19 DOWNTO 0)  -- sfix20
      );
END vsum_mid;
```

To flatten the vector ports on the entire model, set `ScalarizePorts` to `on`. The vector ports at `vsum_mid` and the inputs to the Sum of Elements block are also flattened into scalars.

```
makehdl('hdlcoder_vector_sum_nested/vsum', 'ScalarizePorts', 'on')

ENTITY vsum_mid IS
  PORT( In1_0           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_1           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_2           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_3           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_4           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_5           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_6           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_7           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_8           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        In1_9           : IN  std_logic_vector(15 DOWNTO 0); -- int16
        Out1            : OUT std_logic_vector(19 DOWNTO 0)  -- sfix20
      );
END vsum_mid;
```

Usage Notes and Restrictions

- When you use the `ScalarizePorts` property for a protected model, you must use the same value for this property as the value specified for this parameter in the top model from which it is referenced.
- When you use the IP Core Generation and FPGA-in-the-Loop workflows, vector ports are not supported. Set `ScalarizePorts` as `on` or `dutlevel`. For faster code generation, set `ScalarizePorts` to `dutlevel`.
- Vector or matrix ports as input to a model reference interface must be scalarized before generating HDL code. Set `ScalarizePorts` to `dutlevel`.
- By default, **Generate VHDL code for model references into a single library** is enabled. The VHDL code is generated in a single library instead of separate libraries. In this case, set the `ScalarizePorts` property to `off` before generating HDL code.
- When generating code, if you encounter typing or naming conflicts between vector ports when interfacing two or more generated VHDL code modules, use the `ScalarizePorts` property to generate non-conflicting port definitions.

See Also

`makehdl`

More About

- "Model Referencing for HDL Code Generation" on page 27-2
- "Generate Black Box Interface for Referenced Model" on page 27-8

Create a Xilinx System Generator Subsystem

In this section...

- “Why Use Xilinx System Generator Subsystems?” on page 27-27
- “Requirements for Xilinx System Generator Subsystems” on page 27-27
- “How to Create a Xilinx System Generator Subsystem” on page 27-28
- “Limitations for Code Generation from Xilinx System Generator Subsystems” on page 27-28

Why Use Xilinx System Generator Subsystems?

You can generate HDL code from a model with both Simulink and Xilinx blocks using Xilinx System Generator (XSG) subsystems.

Using both Simulink and Xilinx blocks in your model provides the following benefits:

- A single platform for combined Simulink and Xilinx System Generator simulation, code generation, and synthesis.
- Targeted code generation: Xilinx System Generator for DSP generates code from Xilinx blocks; HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Xilinx System Generator Subsystems

You must group your Xilinx blocks into one or more Xilinx System Generator (XSG) subsystems for code generation. An XSG subsystem can contain a hierarchy of subsystems.

To generate code from a Xilinx System Generator subsystem:

- Use Vivado or ISE Design Suite 13.4 or later.
- If your design uses boolean data types, select the **Use STD_LOGIC type for Boolean or 1 bit wide gateways** setting on the Xilinx System Generator window. By default, Xilinx System Generator uses `std_logic_vector` to represent boolean types whereas HDL Coder uses `std_logic`, which can result in a mismatch.

An XSG subsystem is a Subsystem block with:

- Architecture set to **Module**.
- One System Generator token, placed at the top level of the XSG subsystem hierarchy.
- Xilinx blocks.
- Simulink blocks not requiring code generation.
- Input and output ports connected directly to Gateway In or Gateway Out blocks.
- **Propagate data type to output** option enabled on Gateway Out blocks.
- Matching input and output data types on Gateway In blocks. See “Limitations for Code Generation from Xilinx System Generator Subsystems” on page 27-28.

How to Create a Xilinx System Generator Subsystem

- 1 Create a subsystem containing the Xilinx blocks and set its architecture to **Module**.
- 2 Add a System Generator token at the top level of the subsystem.

You can have subsystem hierarchy in a Xilinx System Generator subsystem, but there must be a System Generator token at the top level of the hierarchy.

- 3 Connect each subsystem input or output port directly to a Gateway In or Gateway Out block.
- 4 On each Gateway Out block, select the **Propagate data type to output** option.

Limitations for Code Generation from Xilinx System Generator Subsystems

Code generation from Xilinx System Generator (XSG) subsystems has the following limitations:

- `ConstrainedOutputPipeline`, `InputPipeline`, and `OutputPipeline` are the only valid block properties for an XSG subsystem.
- HDL Coder does not generate code for blocks within an XSG subsystem, including Simulink blocks.
- Gateway In blocks must not do nontrivial data type conversion. For example, a Gateway In block can convert between the `sfix8_en6` and `Fix_8_6` data types, but changing data sign, word length, or fraction length is not allowed.
- For Verilog code generation, Simulink block names in your design cannot be the same as Xilinx names. Similarly, Xilinx blocks in your design cannot have the same name as other Xilinx blocks. HDL Coder cannot resolve these name conflicts, and generates an error late in the code generation process.

See Also

`makehdl` | `makehdltb`

Related Examples

- “Using Xilinx® System Generator for DSP with HDL Coder™” on page 27-36

Create an Altera DSP Builder Subsystem

In this section...

- "Why Use Altera DSP Builder Subsystems?" on page 27-29
- "Requirements for Altera DSP Builder Subsystems" on page 27-29
- "How to Create an Altera DSP Builder Subsystem" on page 27-29
- "Determine Clocking Requirements for Altera DSP Builder Subsystems" on page 27-30
- "Limitations for Code Generation from Altera DSP Builder Subsystems" on page 27-30

Why Use Altera DSP Builder Subsystems?

You can generate HDL code from a model with both Simulink and Altera DSP Builder Advanced blocks using Altera DSP Builder (DSPB) subsystems.

Using both Simulink and Altera blocks in your model provides the following benefits:

- A single platform for combined Simulink and Altera DSP Builder simulation, code generation, and synthesis.
- Targeted code generation: Altera DSP Builder generates code from Altera blocks; HDL Coder generates code from Simulink blocks.
- HDL Coder area and speed optimizations for Simulink components.

Requirements for Altera DSP Builder Subsystems

You must group your Altera blocks into one or more Altera DSP Builder (DSPB) subsystems for code generation. A DSPB subsystem can contain a hierarchy of subsystems.

To generate code from a Altera DSP Builder subsystem, you must use Quartus II 13.0 or later.

A DSPB subsystem is a Subsystem block with:

- Architecture set to **Module**.
- A valid DSP Builder Advanced Blockset design, including a top-level Device block and DSP Builder Advanced blocks, as defined in the Altera DSP Builder documentation.

How to Create an Altera DSP Builder Subsystem

- 1** Create an Altera DSP Builder Advanced Blockset design as defined in the Altera DSP Builder documentation.
- 2** Create a subsystem containing the Altera DSP Builder Advanced Blockset design, and set its **Architecture** to **Module**.

To see an example that shows HDL code generation for an Altera DSP Builder subsystem, see Using Altera DSP Builder Advanced Blockset with HDL Coder.

Determine Clocking Requirements for Altera DSP Builder Subsystems

DSPB subsystems must either run at the DUT subsystem base rate, or you can provide a custom clock.

Determining the DUT subsystem base rate can be an iterative process. Area optimizations, such as RAM mapping or resource sharing, may cause HDL Coder to oversample area-optimized parts of the design. Therefore, the DUT subsystem initial base rate may differ from the final base rate, and you may not know the model base rate until you generate code.

To determine the model base rate, iteratively generate code until your model converges on a base rate:

- 1 Generate code for the DUT subsystem that contains your DSPB subsystem.
- 2 If HDL Coder displays an error message that says that your DSPB subsystem rate is slower than the base rate, modify the DSPB subsystem inputs so that the DSPB subsystem runs at the base rate in the message.
For example, you can insert an Upsample block.
- 3 Repeat these steps until your DSPB subsystem rate matches the base rate.

To provide a custom clock for your DSPB subsystem:

- 1 In the HDL Workflow Advisor, for **HDL Code Generation > Set Code Generation Options > Set Advanced Options > Clock inputs**, select **Multiple**.
- 2 In the generated HDL code, connect your custom clocks to the DUT clock input ports that corresponds to your DSPB subsystems clock.

Limitations for Code Generation from Altera DSP Builder Subsystems

Code generation for Altera DSP Builder (DSPB) subsystems has the following limitations:

- The DUT subsystem cannot be a DSPB subsystem.
- DSPB subsystems must run at the Simulink model base rate. You may need to iteratively generate code to determine the base rate, because area optimizations can cause local multirate. See “Determine Clocking Requirements for Altera DSP Builder Subsystems” on page 27-30 for a workflow.
- Altera blocks with bus interfaces are not supported.
- Altera DSP Builder does not generate Verilog code.
- Test bench simulation mismatches can occur because the Simulink data comparison does not take Altera valid signals into account. For an example and workaround, see Using Altera DSP Builder Advanced Blockset with HDL Coder.

Using Altera DSP Builder Advanced Blockset with HDL Coder

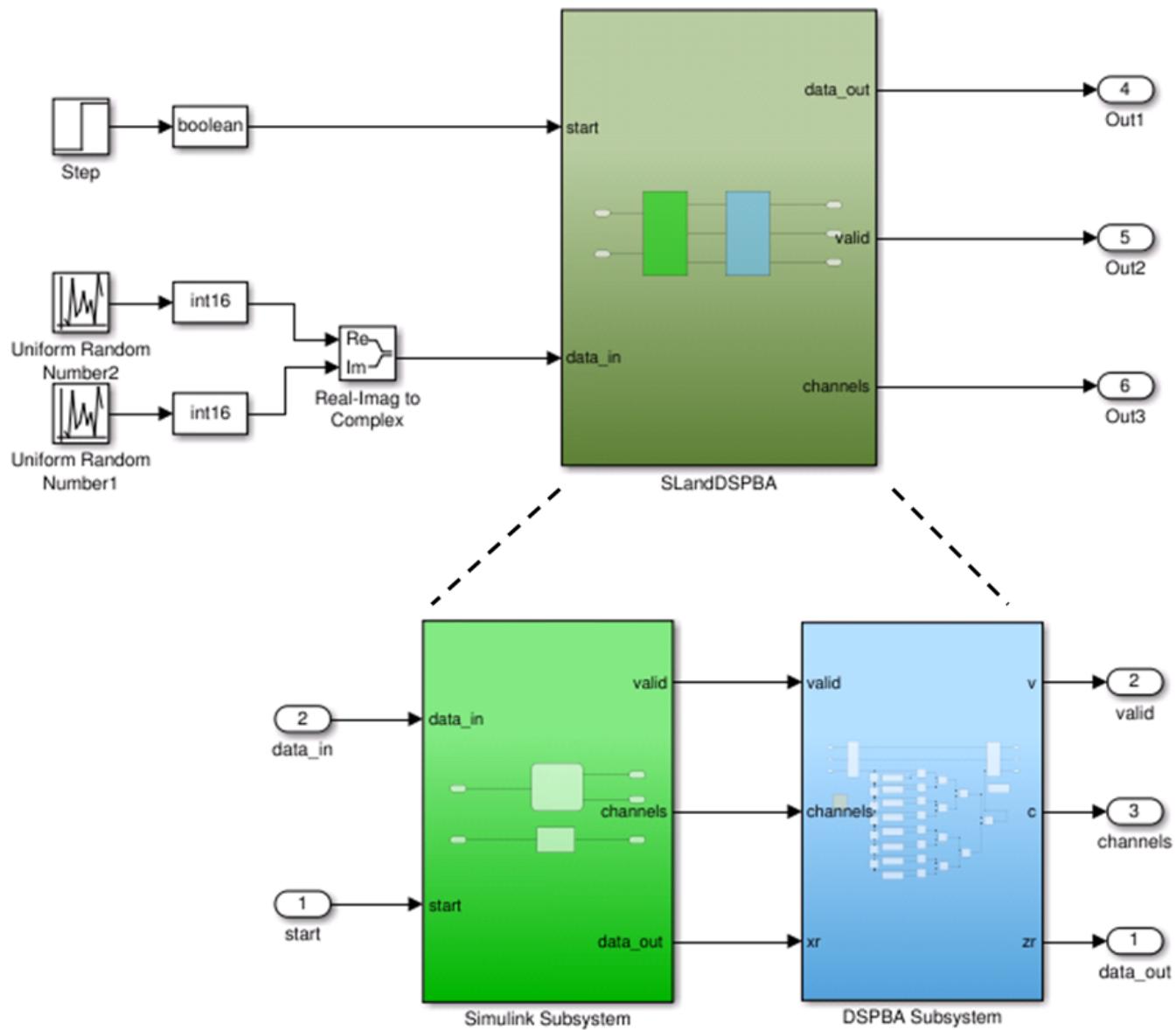
This example shows how to use the Altera® DSP Builder Advanced Blockset with HDL Coder™.

Introduction

Using the Altera® DSP Builder Advanced Blockset Subsystem block, or DSPBA Subsystem block, enables you to model designs using blocks from both Simulink® and Altera®, and to automatically generate integrated HDL code. HDL Coder™ generates HDL code from the Simulink® blocks, and uses Altera® DSP Builder to generate HDL code from the DSPBA Subsystem blocks.

In this example, the design, or code generation subsystem, contains two parts: one with Simulink® native blocks, and one with Altera® DSP Builder Advanced blocks. The Altera® blocks are grouped in a DSPBA Subsystem (hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem). Altera® DSP Builder optimizes these blocks for Altera® FPGAs. In the rest of the design, Simulink® blocks and HDL Coder™ offer many model-based design features, such as distributed pipelining and delay balancing, to perform model-level optimizations.

```
open_system('hdlcoder_sldspba');
open_system('hdlcoder_sldspba/SLandDSPBA');
```



Setup for Altera® DSP Builder Advanced Blockset

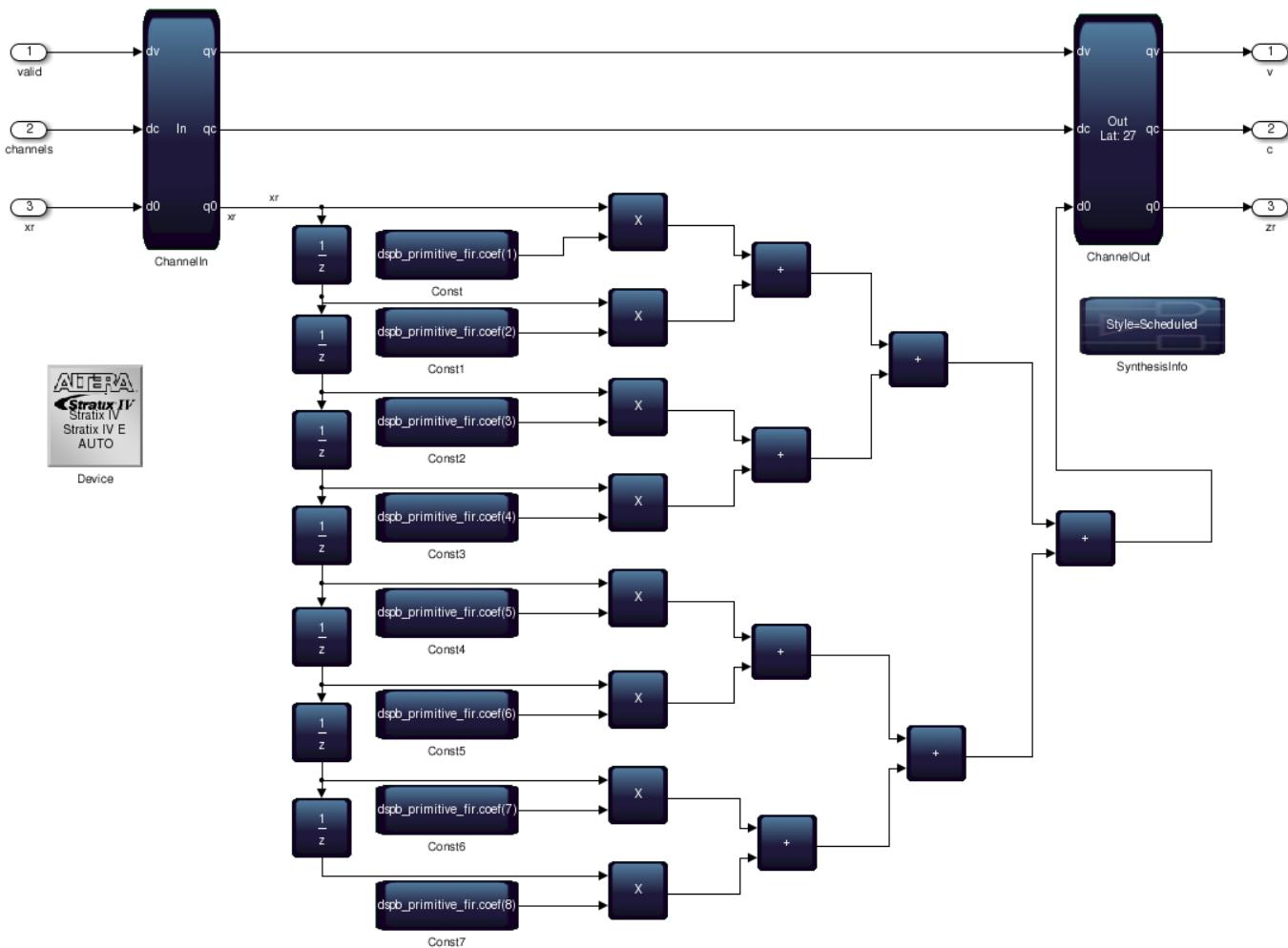
In order to use the Altera® DSP Builder Advanced Blockset Subsystem block, you must have Altera® Quartus II set up with Simulink®. For version compatibility, please refer to the HDL Coder documentation.

Create Altera® DSP Builder Advanced Blockset Subsystem

To create a DSPBA Subsystem:

- 1 Put the Altera® blocks in one subsystem and set its architecture to "Module" (the default value).
- 2 Place a Device block at the top level of the subsystem. You can have subsystem hierarchy in a DSPBA Subsystem, but there must be a Device block at the top level of the hierarchy.

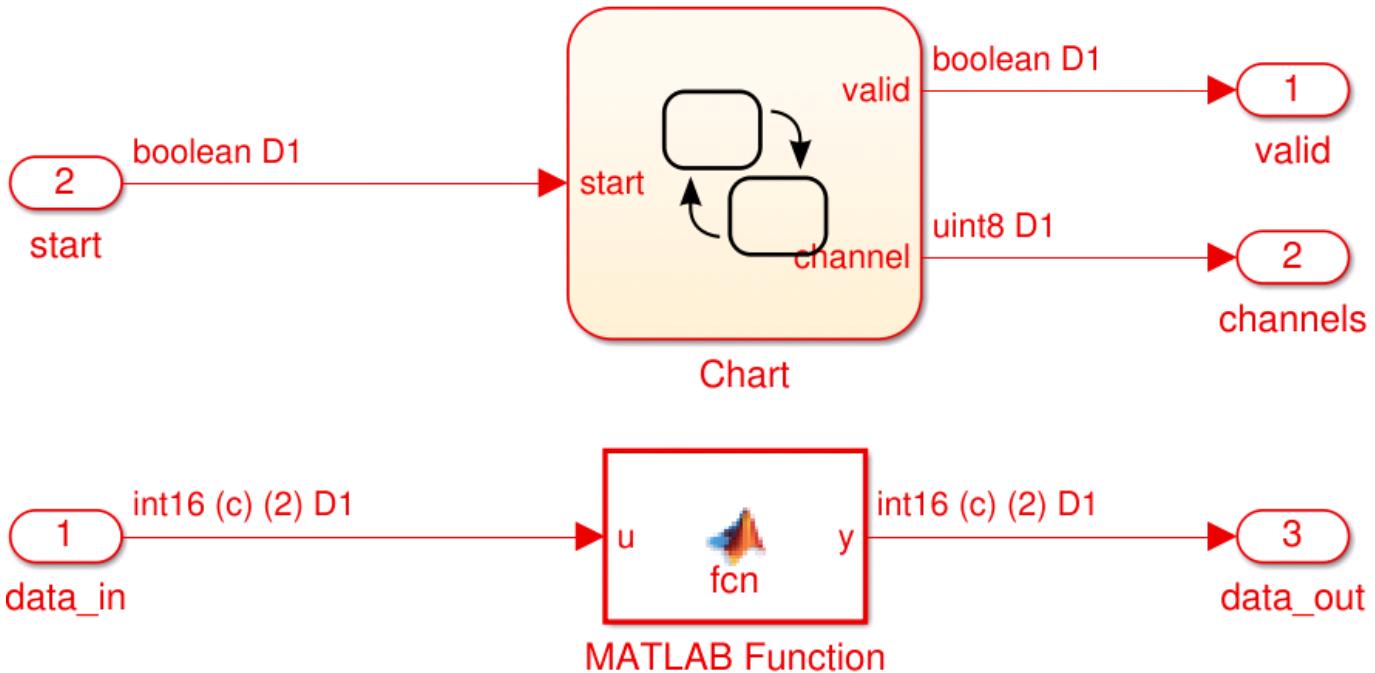
```
open_system('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem');
```



Simulink® Components

In this example, a Stateflow chart generates the channel and valid signals to drive the DSPBA subsystem.

```
open_system('hdlcoder_sldspba/SLandDSPBA/Simulink Subsystem');
```



Generate HDL Code

You can use either `makehdl` at the command line or HDL Workflow Advisor to generate HDL code. To use `makehdl`:

```
makehdl('hdlcoder_sldspba/SLandDSPBA');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Handle Simulation Mismatch When Valid Signal Not Asserted

The DSPBA Subsystem simulation may not match its generated code's behavior when the valid signal is not asserted under certain circumstances, such as when the folding option in both `hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelIn` and `hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelOut` are turned on. The mismatch affects the downstream Simulink design and causes a test bench simulation failure.

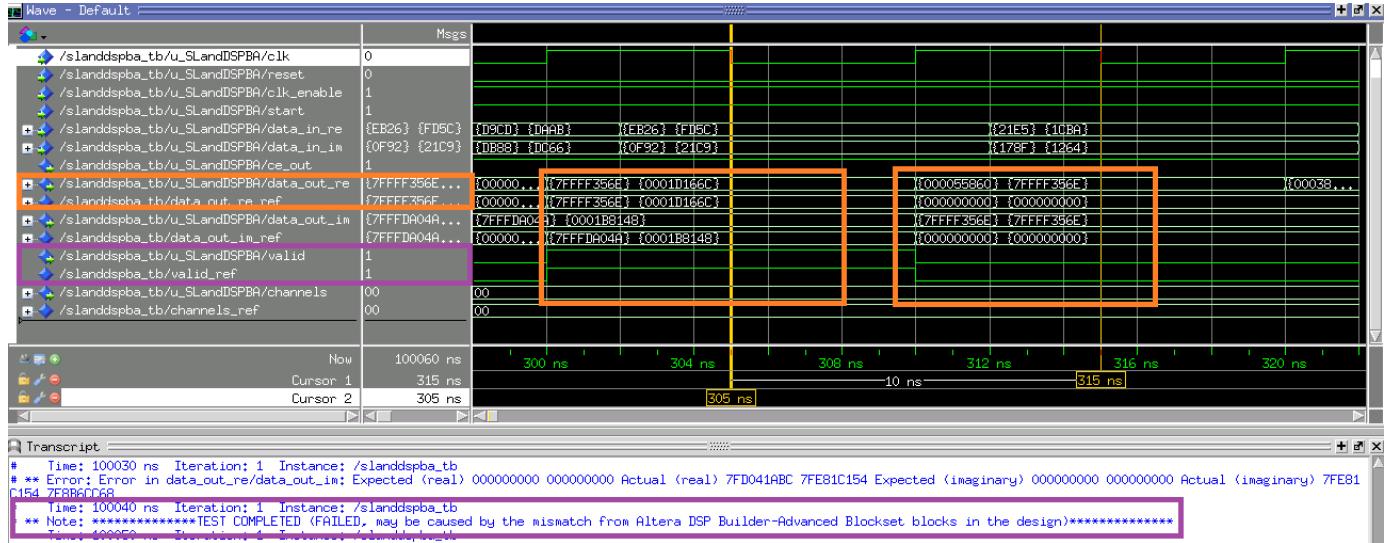
To see the mismatch, you can turn the folding setting on the ChannelIn and ChannelOut blocks:

```
set_param('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelIn', 'FoldingEnabled', 1);
set_param('hdlcoder_sldspba/SLandDSPBA/DSPBA Subsystem/ChannelOut', 'FoldingEnabled', 1);
```

Then, generate the HDL code and test bench again:

```
makehdl('hdlcoder_sldspba/SLandDSPBA');
makehdltb('hdlcoder_sldspba/SLandDSPBA');
```

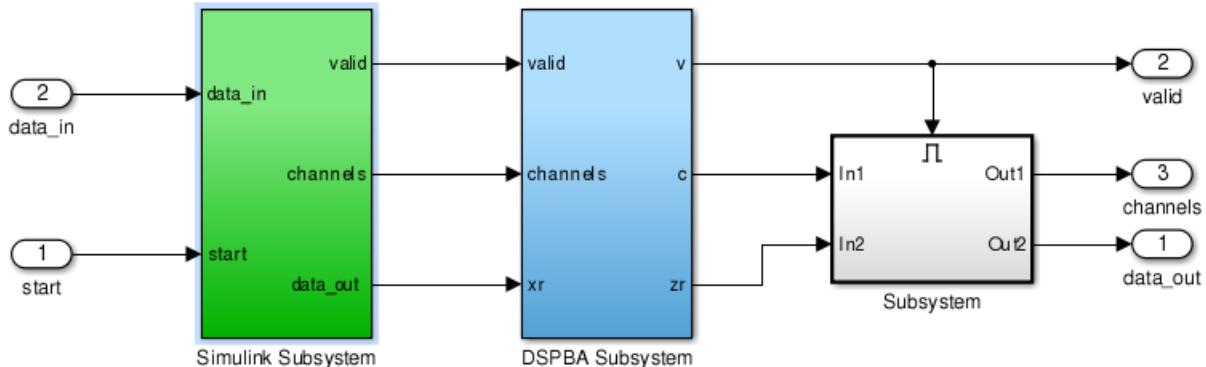
After simulating the generated code and test bench, you can see that the outputs from HDL coder match the reference data only when the valid signal is asserted.



As the message from the test bench indicates, the mismatch is expected.

To avoid this simulation mismatch, insert an enabled subsystem at the DSPBA Subsystem output signals, before they reach the Simulink part of your design or the output ports of the overall design. The following subsystem shows how to connect signals to the enabled subsystem.

```
open_system('hdlcoder_sldspba/SLandDSPBA2');
```



Using Xilinx® System Generator for DSP with HDL Coder™

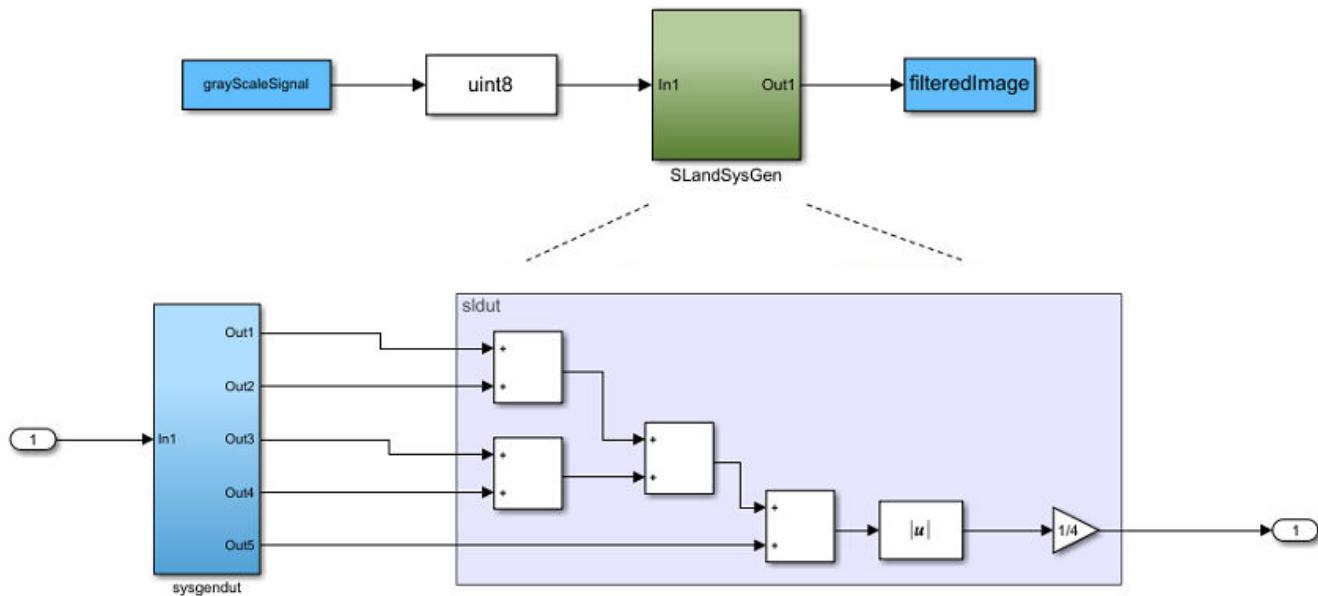
This example shows how to use Xilinx System Generator for DSP with HDL Coder.

Introduction

Using the Xilinx System Generator Subsystem block enables you to model designs using blocks from both Simulink® and Xilinx, and to automatically generate integrated HDL code. HDL Coder™ generates HDL code from the Simulink blocks, and uses Xilinx System Generator to generate HDL code from the Xilinx System Generator Subsystem blocks.

In this example, the design, or code generation subsystem, contains two parts: one with Simulink native blocks, and one with Xilinx blocks. The Xilinx blocks are grouped into a Xilinx System Generator Subsystem `sysgendut` that is inside a `SLandSysGen` Subsystem at the top level of the model `hdlcoder_slsysgen`. System Generator optimizes these blocks for Xilinx FPGAs. In the rest of the design, Simulink blocks and HDL Coder offer model-based design capabilities and HDL optimizations, such as distributed pipelining and delay balancing.

```
open_system('hdlcoder_slsysgen');
open_system('hdlcoder_slsysgen/SLandSysGen');
```

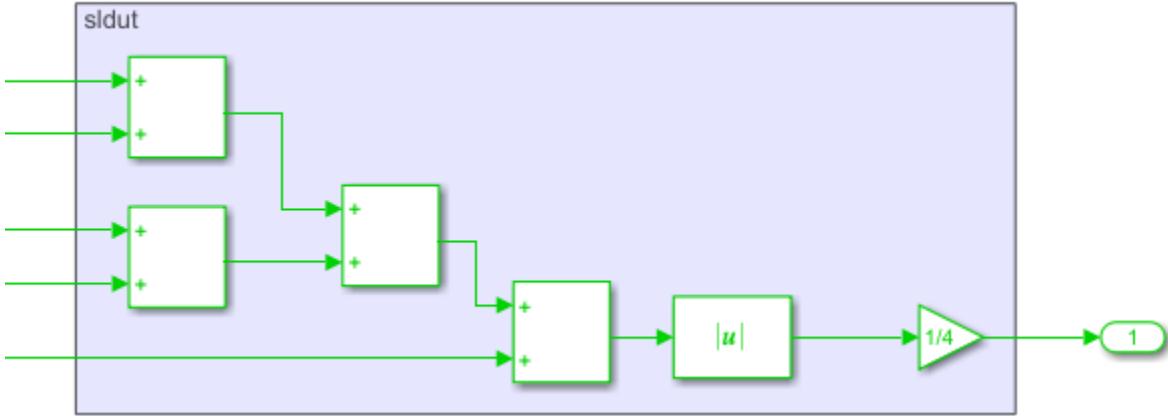


Perform Model-Level Optimizations for Simulink® Components

In this example, a sum tree indicated by the `sldut` section inside the `SLandSysGen` Subsystem is modeled with Simulink blocks. You can use distributed pipelining feature to take care of the speed optimization.

Distributed pipelining can move pipeline registers into the sum tree to reduce the critical path without changing the model function. Other optimizations, such as resource sharing, are also available, but not used in this example.

```
open_system('hdlcoder_slsysgen/SLandSysGen');
```

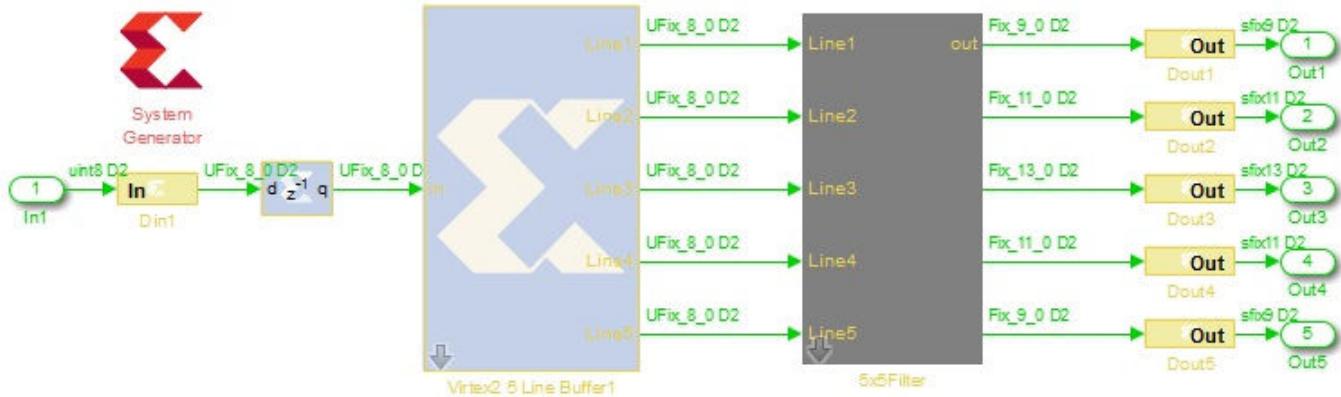


Create Xilinx System Generator Subsystem

To create a Xilinx System Generator subsystem:

- 1 Put the Xilinx blocks in one subsystem and leave the HDL architecture set to default value of **Module**.
- 2 Place a System Generator token at the top level of the subsystem. You can have subsystem hierarchy in a Xilinx System Generator Subsystem, but there must be a System Generator token at the top level of the hierarchy.

```
open_system('hdlcoder_slsysgen/SLandSysGen/sysgendut');
```



Configure Gateway In and Gateway Out Blocks

In each Xilinx System Generator subsystem, you must connect input and output ports directly to Gateway In and Gateway Out blocks.

Gateway In blocks must not do non-trivial data type conversion. For example, a Gateway In block can convert between `uint8` and `UFix_8_0`, but changing data sign, word length, or fraction length is not allowed.

Generate HDL Code

You can use either makehdl at the command line or HDL Workflow Advisor to generate HDL code. To use makehdl:

```
makehdl('hdlcoder_slsysgen/SLandSysGen');
```

You can also generate a testbench, simulate, and synthesize the design as you would for any other model.

Choose a Test Bench for Generated HDL Code

When you generate HDL code with HDL Coder, you can optionally generate a test bench as well. The coder also generates build-and-run scripts for the HDL simulator you specify. The test bench options are:

- HDL test bench — An HDL test bench that includes the generated HDL DUT and files containing input and output data vectors. This test bench verifies the generated HDL DUT against test vectors generated from your Simulink model. See “Test Bench Generation” on page 35-15.
- Cosimulation model — A Simulink model that includes an HDL Cosimulation block that runs your generated HDL code in an HDL simulator. The model also includes your original Simulink stimulus generation, your behavioral model, and any blocks for display or analysis of the output data. The model compares the output of the HDL Cosimulation block against the output of the source subsystem. See “Generate a Cosimulation Model” on page 27-41.
- SystemVerilog DPI test bench — An HDL test bench that includes the generated HDL DUT and a generated C-language component. The C component creates input stimuli and runs a behavioral model of the DUT subsystem. The test bench uses a direct programming interface (DPI) to run the C component inside an HDL simulation. This test bench verifies the generated HDL DUT against a C representation of the source Simulink model. See “Verify HDL Design Using SystemVerilog DPI Test Bench” on page 27-79.
- FPGA-in-the-loop — A Simulink model that includes an FPGA-in-the-Loop block that communicates with your HDL design while it runs on the FPGA board. The model also includes your original Simulink stimulus generation, your behavioral model, and any blocks for display or analysis of the output data. The model compares the output of the FPGA-in-the-Loop block against the output of the source subsystem. See “FIL Simulation with HDL Workflow Advisor for Simulink” (HDL Verifier).

Select test bench options in HDL Workflow Advisor under **HDL Code Generation > Set Testbench Options**, or in the Model Configuration Parameters dialog box, under **HDL Code Generation > Test Bench**. Alternatively, for command-line access, select your test bench using the properties of `makehdltb`.

For FPGA-in-the-loop, select the target workflow in HDL Workflow Advisor under **Set Target > Set Target Device and Synthesis Tool**. Then select your FPGA and synthesis tool. You can also generate an FPGA-in-the-loop model for existing HDL code by using **FPGA-in-the-Loop Wizard**.

Test Bench	License Requirements	Pros	Cons
HDL test bench		<ul style="list-style-type: none"> • Fast compile time in HDL simulator 	<ul style="list-style-type: none"> • Runs simulation to generate data files, which can take a long time for large data sets • File I/O can slow down simulation for large data sets • Run test in HDL simulator • Fixed input stimulus

Test Bench	License Requirements	Pros	Cons
Cosimulation model	• HDL Verifier	<ul style="list-style-type: none"> Fast compile time in HDL simulator Run tests from Simulink, including changing parameters to affect input stimulus Automatic test bench execution from HDL Workflow Advisor 	
SystemVerilog DPI test bench	<ul style="list-style-type: none"> HDL Verifier Simulink Coder 	<ul style="list-style-type: none"> Fast generation time because the coder does not run a simulation Fast simulation time for large data sets, because the stimulus comes from generated code rather than files 	<ul style="list-style-type: none"> Run test in HDL simulator No tunable parameters in stimulus generation
FPGA-in-the-loop	<ul style="list-style-type: none"> HDL Verifier HDL Verifier Support Package for Xilinx FPGA Boards or HDL Verifier Support Package for Intel FPGA Boards 	<ul style="list-style-type: none"> Run tests from Simulink, including changing parameters to affect input stimulus Prototype hardware implementation of your DUT 	<ul style="list-style-type: none"> Long generation time due to synthesis into FPGA Hardware setup

See Also

More About

- “Set HDL Code Generation Options” on page 12-2

Generate a Cosimulation Model

In this section...

- ["Requirements" on page 27-41](#)
- ["What Is A Cosimulation Model?" on page 27-41](#)
- ["Generating a Cosimulation Model from the GUI" on page 27-42](#)
- ["Structure of the Generated Model" on page 27-45](#)
- ["Launching a Cosimulation" on page 27-50](#)
- ["The Cosimulation Script File" on page 27-52](#)
- ["Complex and Vector Signals in the Generated Cosimulation Model" on page 27-54](#)
- ["Generating a Cosimulation Model from the Command Line" on page 27-55](#)
- ["Naming Conventions for Generated Cosimulation Models and Scripts" on page 27-55](#)
- ["Limitations for Cosimulation Model Generation" on page 27-56](#)

Requirements

- To use this feature, your installation must include an HDL Verifier license.
- Make sure the DUT subsystem has no unconnected output ports. See "Terminate Unconnected Block Outputs and Usage of Commenting Blocks" on page 21-25.

What Is A Cosimulation Model?

A cosimulation model is an automatically generated Simulink model configured for both Simulink simulation and cosimulation of your design with an HDL simulator. HDL Coder supports automatic generation of a cosimulation model as a part of the test bench generation process.

The cosimulation model includes:

- A behavioral model of your design, realized in a Simulink subsystem.
- A corresponding HDL Cosimulation block, configured to cosimulate the design using HDL Verifier. HDL Coder configures the HDL Cosimulation block for use with either Mentor Graphics ModelSim or Cadence Incisive.
- Test input data, calculated from the test bench stimulus that you specify.
- Scope blocks, which let you observe and compare the DUT and HDL cosimulation outputs, and any error between these signals.
- Goto and From blocks that capture the stimulus and response signals from the DUT and use these signals to drive the cosimulation.
- A comparison/assertion mechanism that reports discrepancies between the original DUT output and the cosimulation output .

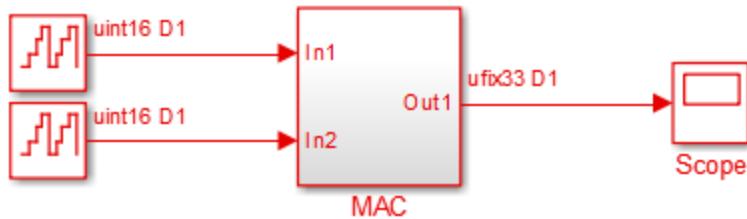
In addition to the generated model, HDL Coder generates a TCL script that launches and configures your cosimulation tool. Comments in the script file document clock, reset, and other timing signal information defined by the coder for the cosimulation tool.

Generating a Cosimulation Model from the GUI

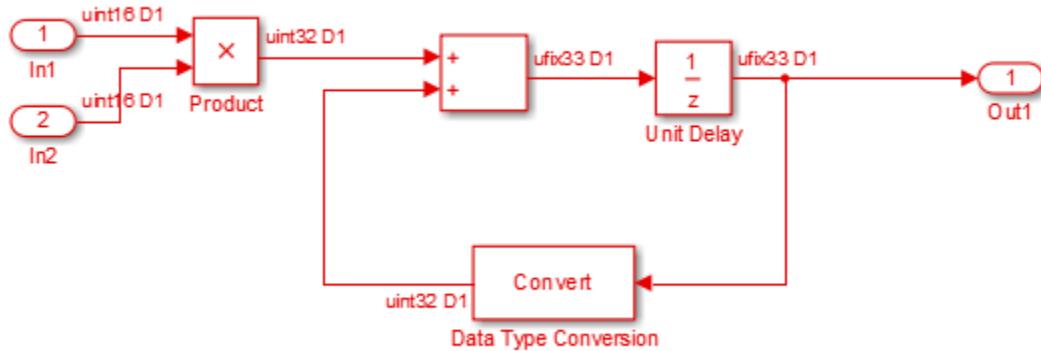
This example demonstrates the process for generating a cosimulation model. The example model, `hdl_cosim_demo1`, implements a simple multiply and accumulate (MAC) algorithm. Open the model by entering the name at the MATLAB command line:

```
hdl_cosim_demo1
```

The following figure shows the top-level model.

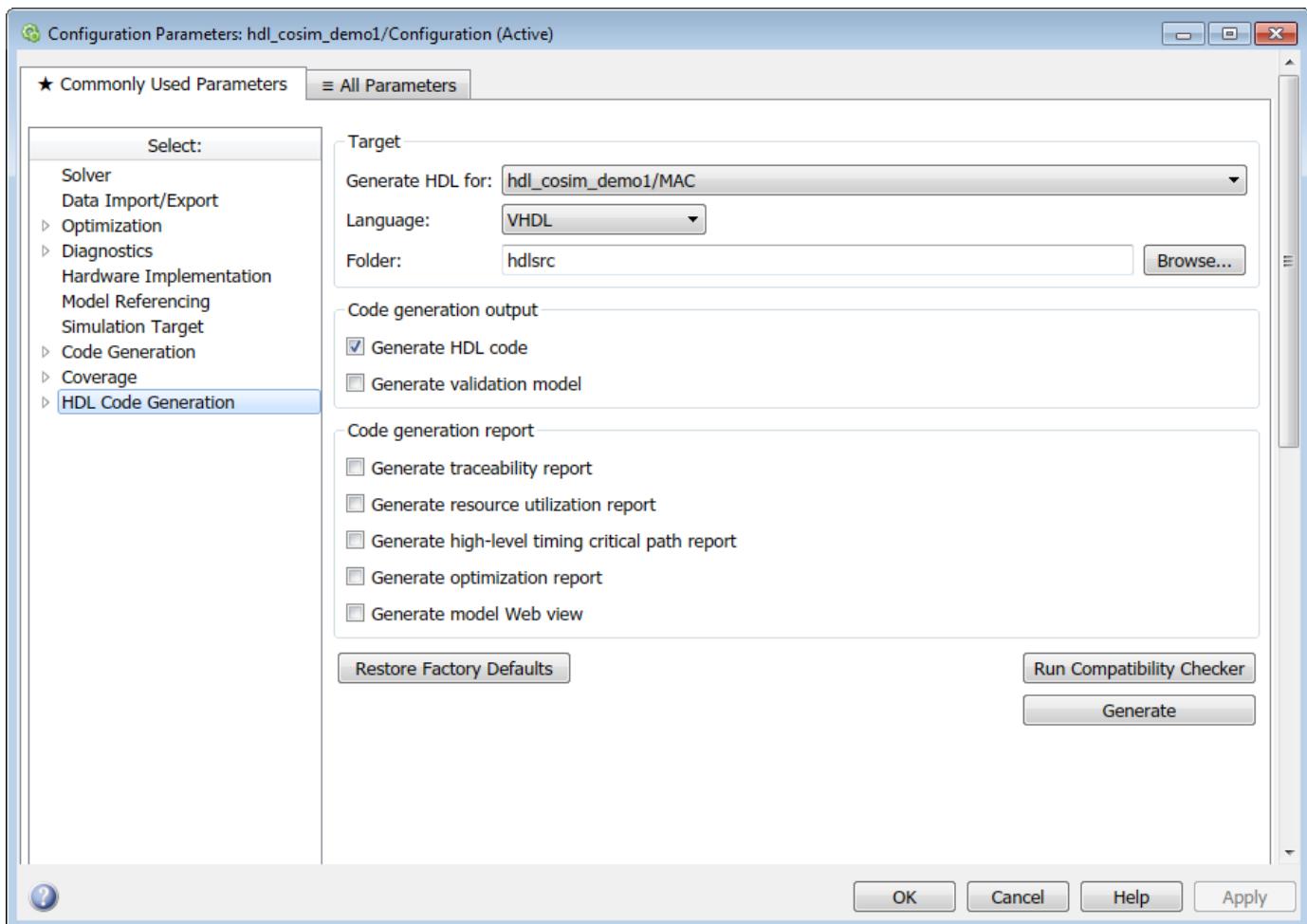


The DUT is the MAC subsystem.



Cosimulation model generation takes place during generation of the test bench. As a best practice, generate HDL code before generating a test bench, as follows:

- 1 In the **HDL Code Generation** pane of the Configuration Parameters dialog box, select the DUT for code generation. In this case, it is `hdl_cosim_demo1/MAC`.



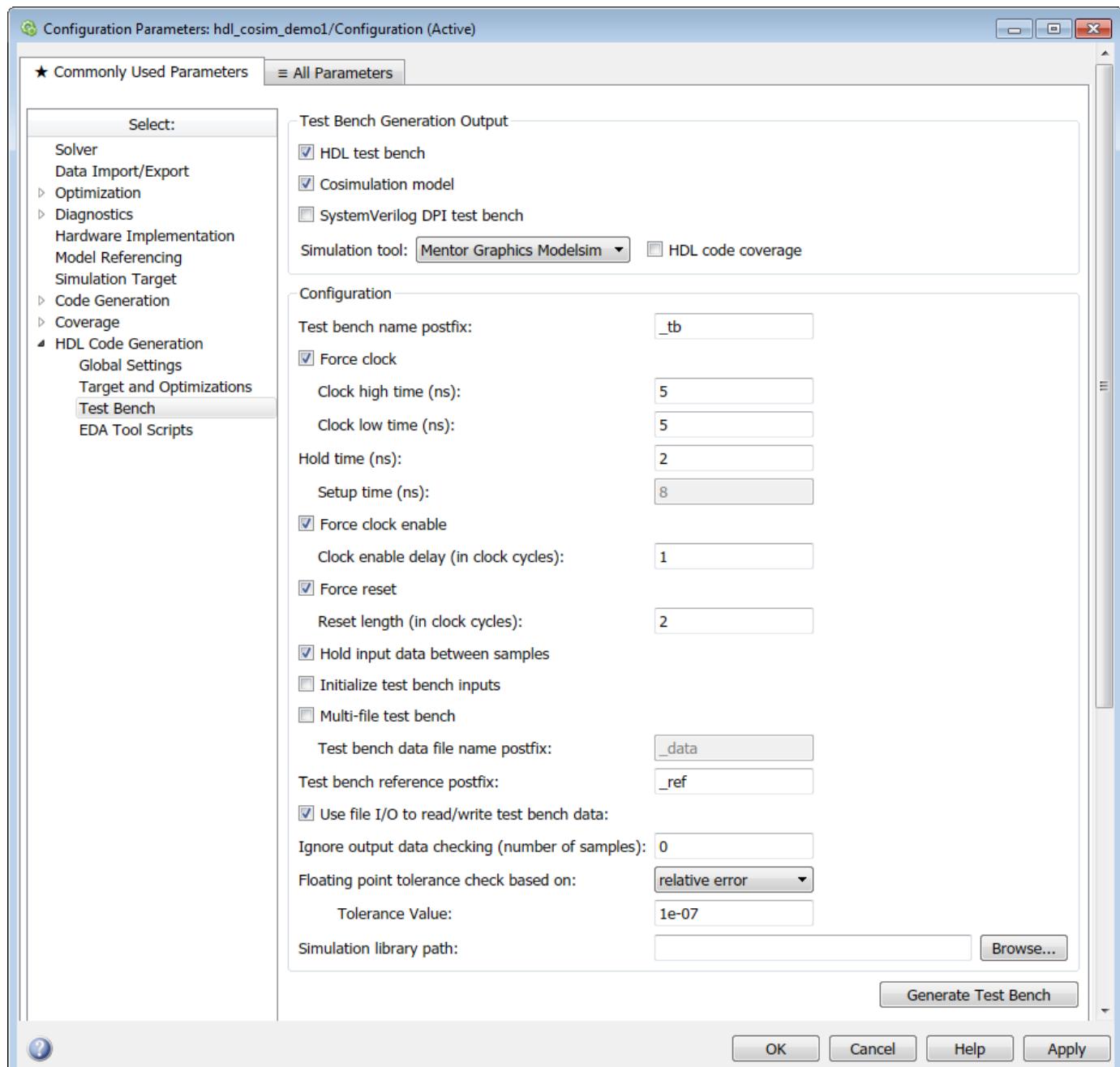
- 2** Click **Apply**.
- 3** Click **Generate**. HDL Coder displays progress messages, as shown in the following listing:

```
### Applying HDL Code Generation Control Statements
### Starting HDL Check.
### HDL Check Complete with 0 error, 0 warning and 0 message.

### Begin VHDL Code Generation
### Working on hdl_cosim_demo1/MAC as hdlsrc\MAC.vhd
### HDL Code Generation Complete.
```

Next, configure the test bench options to include generation of a cosimulation model:

- 1** Select the **HDL Code Generation > Test Bench** pane of the Configuration Parameters dialog box.
- 2** Select the **Cosimulation model** check box. Then select your **Simulation tool** in the pull-down menu.



- 3 Configure required test bench options. HDL Coder records option settings in a generated script file (see “The Cosimulation Script File” on page 27-52).
- 4 Click **Apply**.

Next, generate test bench code and the cosimulation model:

- 1 At the bottom of the **Test Bench** pane, click **Generate Test Bench**. HDL Coder displays progress messages as shown in the following listing:

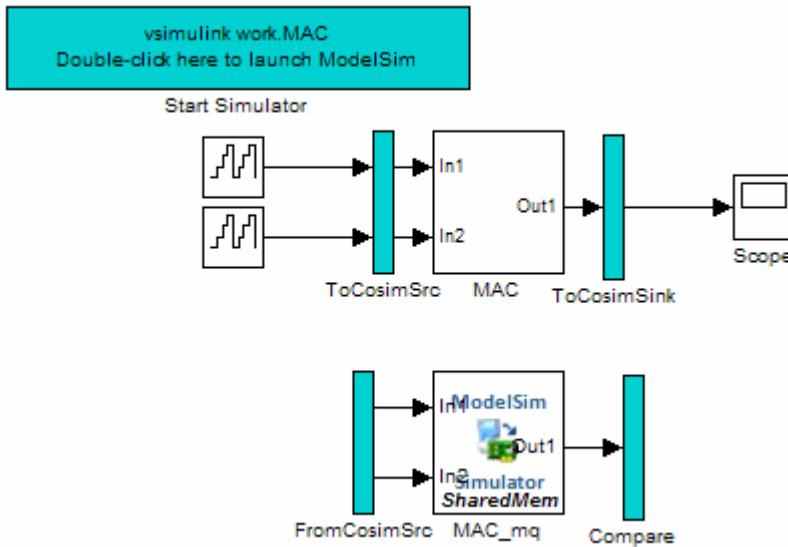
```
### Begin TestBench Generation
### Generating new cosimulation model: gm_hdl_cosim_demo1_mq0.mdl
### Generating new cosimulation tcl script: hdlsrc/gm_hdl_cosim_demo1_mq0_tcl.m
```

```

### Cosimulation Model Generation Complete.
### Generating Test bench: hlsrsrc\MAC_tb.vhd
### Please wait ...
### HDL TestBench Generation Complete.

```

When test bench generation completes, HDL Coder opens the generated cosimulated model. The following figure shows the generated model.



2 Save the generated model.

The generated model exists only in memory unless you save it.

As indicated by the code generation messages, HDL Coder generates the following files in addition to the usual HDL test bench file:

- A cosimulation model (`gm_hdl_cosim_demo1_mq`)
- A file that contains a TCL cosimulation script and information about settings of the cosimulation model (`gm_hdl_cosim_demo1_mq_tcl.m`)

Generated file names derive from the model name, as described in “Naming Conventions for Generated Cosimulation Models and Scripts” on page 27-55.

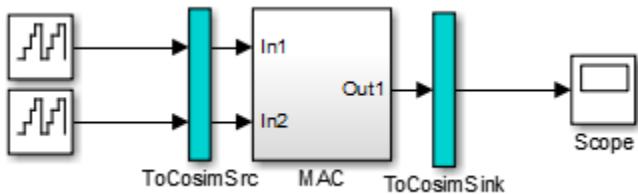
The next section, “Structure of the Generated Model” on page 27-45, describes the features of the model. Before running a cosimulation, become familiar with these features.

Structure of the Generated Model

You can set up and launch a cosimulation using controls located in the generated model. This section examines the model generated from the example MAC subsystem.

Simulation Path

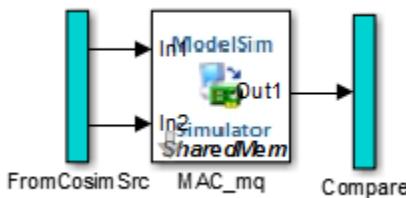
The model comprises two parallel signal paths. The simulation path, located in the upper half of the model window, is nearly identical to the original DUT. The purpose of the simulation path is to execute a normal Simulink simulation and provide a reference signal for comparison to the cosimulation results. The following figure shows the simulation path.



The two subsystems labelled `ToCosimSrc` and `ToCosimSink` do not change the performance of the simulation path. Their purpose is to capture stimulus and response signals of the DUT and route them to and from the HDL cosimulation block (see “Signal Routing Between Simulation and Cosimulation Paths” on page 27-48).

Cosimulation Path

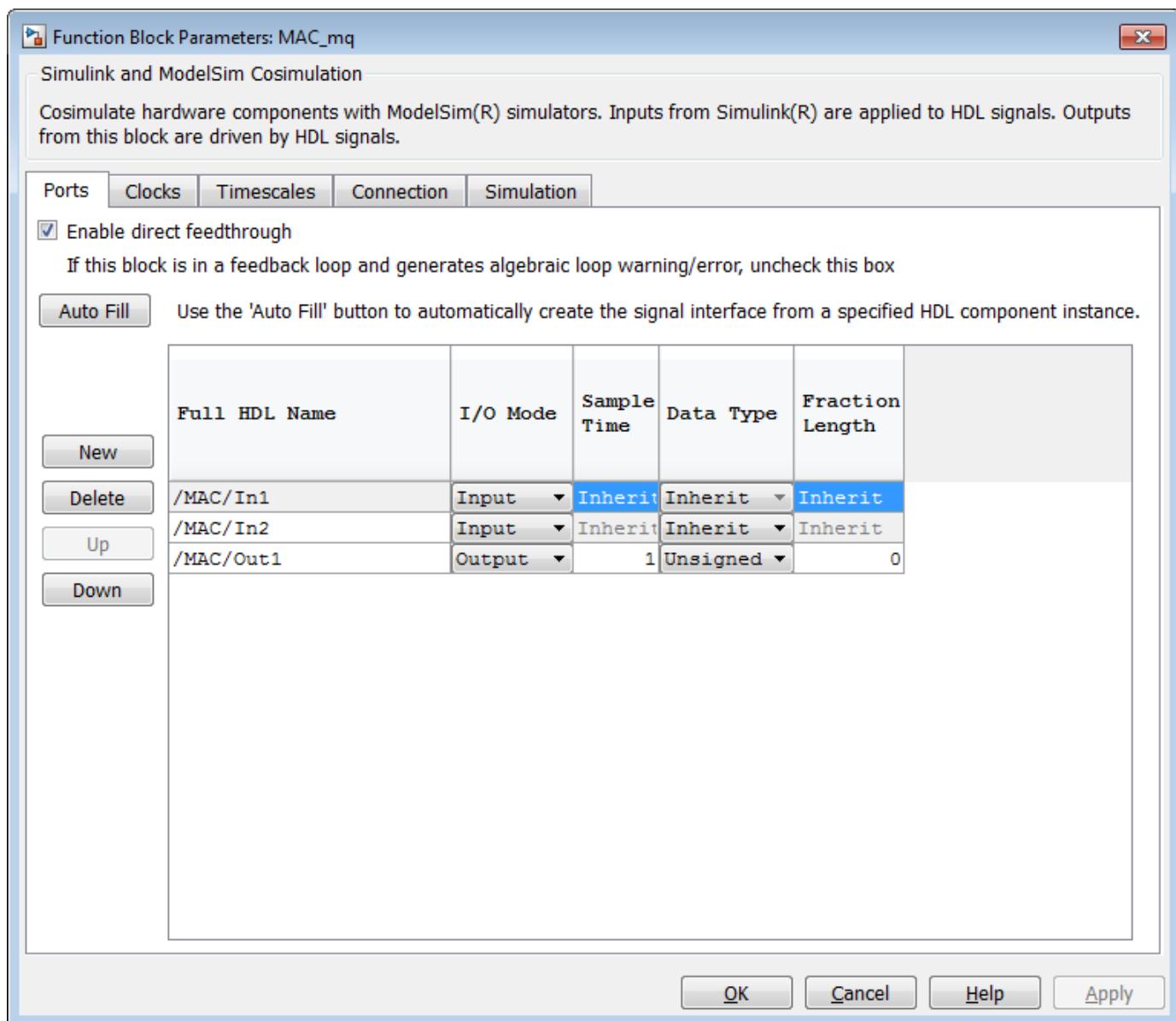
The cosimulation path, located in the lower half of the model window, contains the generated HDL Cosimulation block. The following figure shows the cosimulation path.



The `FromCosimSrc` subsystem receives the same input signals that drive the DUT. In the `gm_hdl_cosim_demo1_mq0` model, the subsystem simply passes the inputs on to the HDL Cosimulation block. Signals of some other data types require further processing at this stage (see “Signal Routing Between Simulation and Cosimulation Paths” on page 27-48).

The `Compare` subsystem at the end of the cosimulation path compares the cosimulation output to the reference output produced by the simulation path. If the comparison detects a discrepancy, an Assertion block in the `Compare` subsystem displays a warning message. If desired, you can disable assertions and control other operations of the `Compare` subsystem. See “Controlling Assertions and Scope Displays” on page 27-49 for details.

HDL Coder populates the HDL Cosimulation block with the compiled I/O interface of the DUT. The following figure shows the **Ports** pane of the `Mac_mq` HDL Cosimulation block.



HDL Coder sets the **Full HDL Name**, **Sample Time**, **Data Type**, and other fields as required by the model. HDL Coder also configures other HDL Cosimulation block parameters under the **Timescales** and **Tcl** panes.

Tip HDL Coder configures the generated HDL Cosimulation block for the Shared Memory connection method.

Start Simulator Control

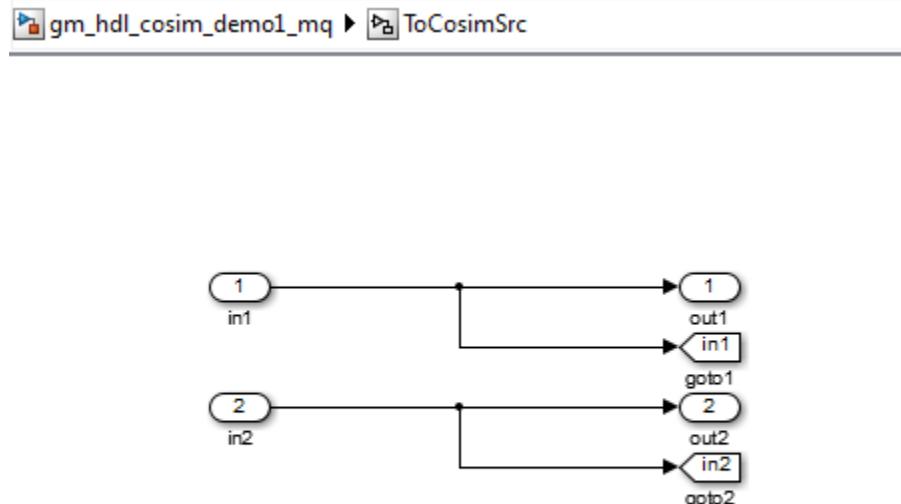
When you double-click the Start Simulator control, it launches the selected cosimulation tool and passes in a startup command to the tool. The Start Simulator icon displays the startup command, as shown in the following figure.

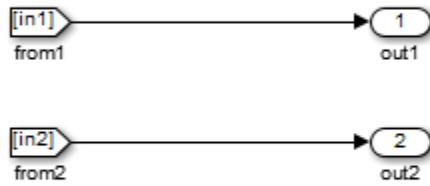


The commands executed when you double-click the Start Simulator icon launch and set up the cosimulation tool, but they do not start the actual cosimulation. “Launching a Cosimulation” on page 27-50 describes how to run a cosimulation with the generated model.

Signal Routing Between Simulation and Cosimulation Paths

The generated model routes signals between the simulation and cosimulation paths using Goto and From blocks. For example, the Goto blocks in the ToCosimSrc subsystem route each DUT input signal to a corresponding From block in the FromCosimSrc subsystem. The following figures show the Goto and From blocks in each subsystem.

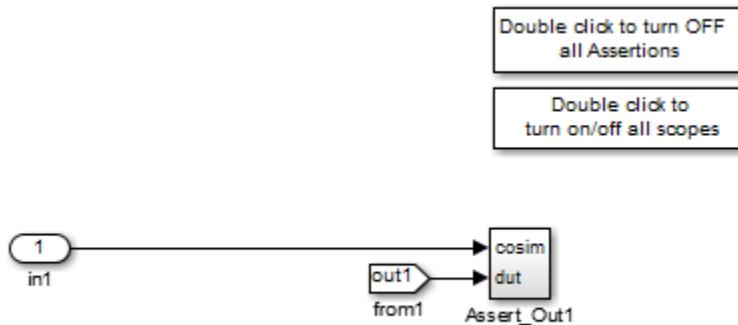




The preceding figures show simple scalar inputs. Signals of complex and vector data types require further processing. See “Complex and Vector Signals in the Generated Cosimulation Model” on page 27-54 for further information.

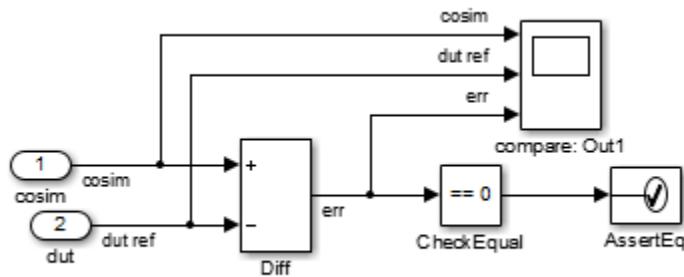
Controlling Assertions and Scope Displays

The Compare subsystem lets you control the display of signals on scopes, and warning messages from assertions. The following figure shows the Compare subsystem for the `gm_hdl_cosim_demo1_mq0` model.



For each output of the DUT, HDL Coder generates an assertion checking subsystem (`Assert_OutN`). The subsystem computes the difference (`err`) between the original DUT output (`dut ref`) and the corresponding cosimulation output (`cosim`). The subsystem routes the comparison result to an Assertion block. If the comparison result is not zero, the Assertion block reports the discrepancy.

The following figure shows the `Assert_Out1` subsystem for the `gm_hdl_cosim_demo1_mq0` model.



This subsystem also routes the `dut_ref`, `cosim`, and `err` signals to a Scope for display at the top level of the model.

By default, the generated cosimulation model enables all assertions and displays all Scopes. Use the buttons on the Compare subsystem to disable assertions or hide Scopes.

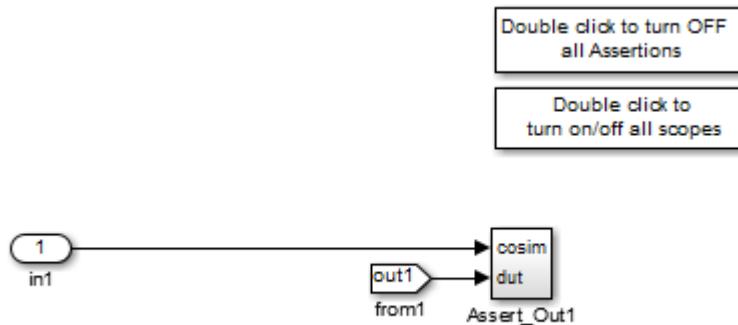
Tip Assertion messages are warnings and do not stop simulation.

Launching a Cosimulation

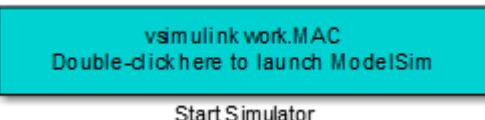
To run a cosimulation with the generated model:

- Double-click the Compare subsystem to configure Scopes and assertion settings.

If you want to disable Scope displays or assertion warnings before starting your cosimulation, use the buttons on the Compare subsystem (shown in the following figure).



- Double-click the Start Simulator control.



The Start Simulator control launches your HDL simulator (in this case, HDL Verifier for use with Mentor Graphics ModelSim).

The HDL simulator in turn executes a startup script. In this case the startup script consists of the TCL commands located in `gm_hdl_cosim_demo1_mq0_tcl.m`. When the HDL simulator finishes executing the startup script, it displays a message like the following.

```
# Ready for cosimulation...
```

- 3 In the Simulink Editor for the generated model, start simulation.

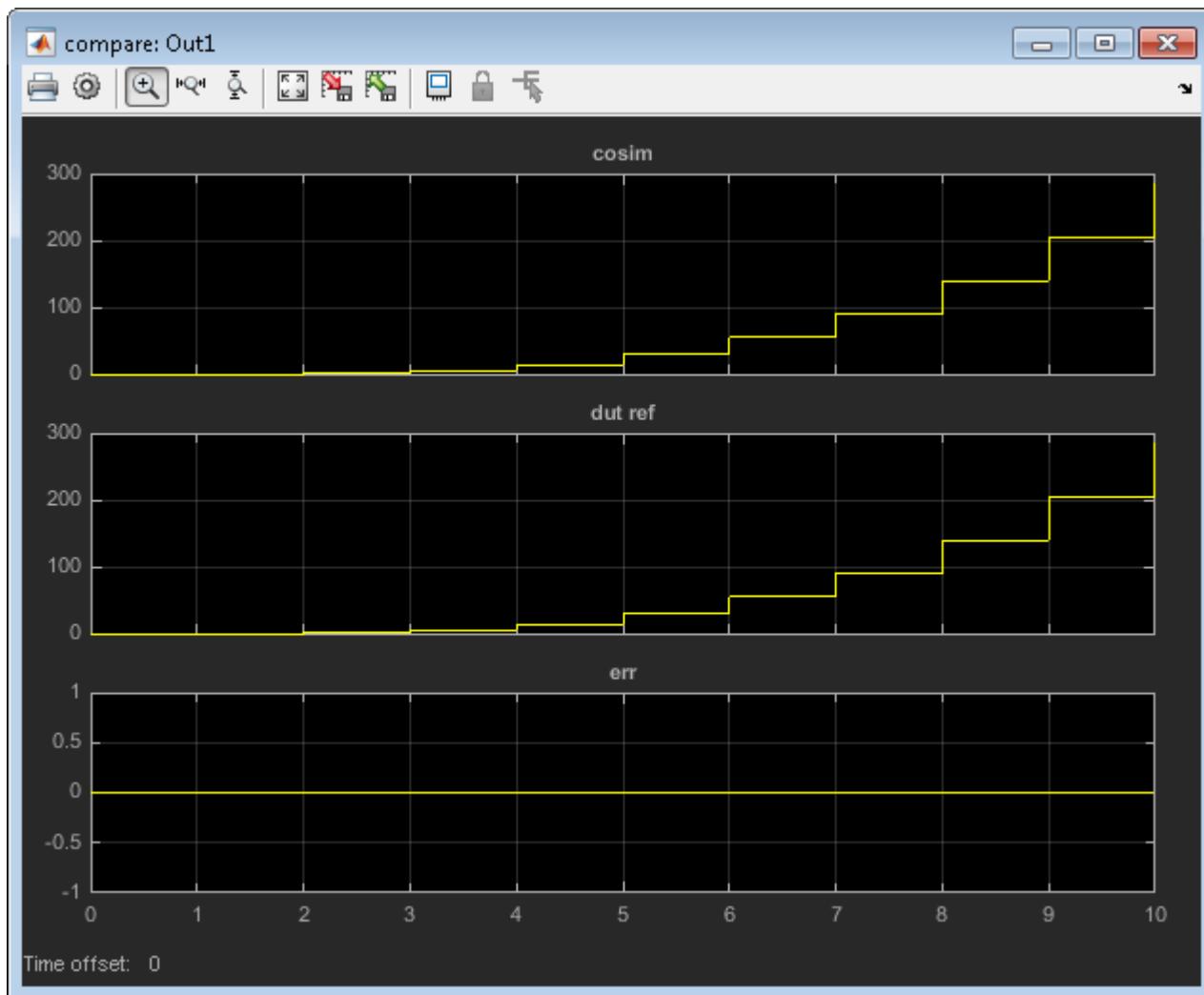
As the cosimulation runs, the HDL simulator displays messages like the following.

```
# Running Simulink Cosimulation block.
# Chip Name: --> hdl_cosim_demo1/MAC
# Target language: --> vhdl
# Target directory: --> hdlsrc
# Fri Jun 05 4:26:34 PM Eastern Daylight Time 2009
# Simulation halt requested by foreign interface.
# done
```

At the end of the cosimulation, if you have enabled Scope displays, the compare scope displays the following signals:

- `cosim`: The result signal output by the HDL Cosimulation block.
- `dut ref`: The reference output signal from the DUT.
- `err`: The difference (error) between these two outputs.

The following figure shows these signals.



The Cosimulation Script File

The generated script file has two sections:

- A comment section that documents model settings that are relevant to cosimulation.
- A function that stores several lines of TCL code into a variable, `tclCmds`. The cosimulation tools execute these commands when you launch a cosimulation.

Header Comments Section

The following listing shows the comment section of a script file generated for the `hdl_cosim_demo1` model:

```
%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%
% Source Model      : hdl_cosim_demo1.mdl
% Generated Model   : gm_hdl_cosim_demo1.mdl
% Cosimulation Model : gm_hdl_cosim_demo1_mq.mdl
%
% Source DUT        : gm_hdl_cosim_demo1_mq/MAC
% Cosimulation DUT   : gm_hdl_cosim_demo1_mq/MAC_mq
```

```

%
% File Location      : hdlsrc/gm_hdl_cosim_demo1_mq_tcl.m
% Created            : 2009-06-16 10:51:01
%
% Generated by MATLAB 7.9 and HDL Coder 1.6
%%%%%%%%%%%%%
%
% ClockName          : clk
% ResetName          : reset
% ClockEnableName   : clk_enable
%
% ClockLowTime       : 5ns
% ClockHighTime      : 5ns
% ClockPeriod        : 10ns
%
% ResetLength        : 20ns
% ClockEnableDelay   : 10ns
% HoldTime           : 2ns
%%%%%%%%%%%%%
%
% ModelBaseSampleTime : 1
% OverClockFactor    : 1
%%%%%%%%%%%%%
%
% Mapping of DutBaseSampleTime to ClockPeriod
%
% N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
% 1 sec in Simulink corresponds to 10ns in the HDL
% Simulator(N = 10)
%
%%%%%%%%%%%%%
%
% ResetHighAt         : (ClockLowTime + ResetLength + HoldTime)
% ResetRiseEdge        : 27ns
% ResetType            : async
% ResetAssertedLevel   : 1
%
% ClockEnableHighAt   : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
% ClockEnableRiseEdge  : 37ns
%%%%%%%%%%%%%

```

The comments section comprises the following subsections:

- *Header comments*: This section documents the files names for the source and generated models and the source and generated DUT.
- *Test bench settings*: This section documents the `makehdltb` property values that affect cosimulation model generation. The generated TCL script uses these values to initialize the cosimulation tool.
- *Sample time information*: The next two sections document the base sample time and oversampling factor of the model. HDL Coder uses `ModelBaseSampleTime` and `OverClockFactor` to map the clock period of the model to the HDL cosimulation clock period.
- *Clock, clock enable, and reset waveforms*: This section documents the computations of the duty cycle of the `clk`, `clk_enable`, and `reset` signals.

TCL Commands Section

The following listing shows the TCL commands section of a script file generated for the `hdl_cosim_demo1` model:

```

function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
  'do MAC_compile.do',...% Compile the generated code
  'vsimulink work.MAC',...% Initiate cosimulation
  'add wave /MAC/clk',...% Add wave commands for chip input signals
  'add wave /MAC/reset',...
  'add wave /MAC/clk_enable',...
  'add wave /MAC/In1',...

```

```

'add wave /MAC/In2',...
'add wave /MAC/ce_out',...% Add wave commands for chip output signals
'add wave /MAC/Out1',...
iset UserTimeUnit ns',...% Set simulation time unit
'puts "",...
'puts "Ready for cosimulation...",...
};

end

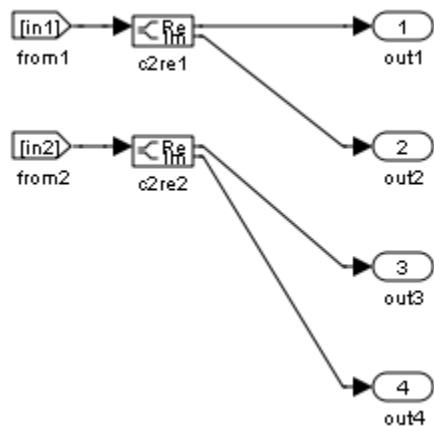
```

Complex and Vector Signals in the Generated Cosimulation Model

Input signals of complex or vector data types require insertion of additional elements into the cosimulation path. This section describes these elements.

Complex Signals

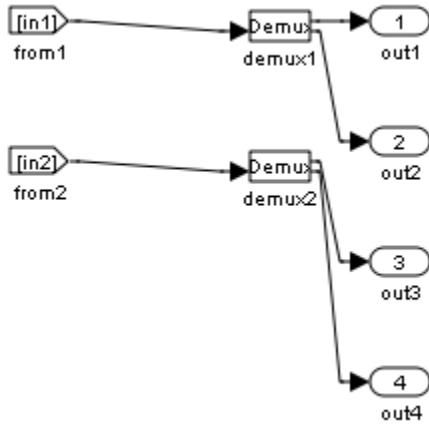
The generated cosimulation model automatically breaks complex inputs into real and imaginary parts. The following figure shows a `FromCosimSrc` subsystem that receives two complex input signals. The subsystem breaks the inputs into real and imaginary parts before passing them to the subsystem outputs.



The model maintains the separation of real and imaginary components throughout the cosimulation path. The `Compare` subsystem performs separate comparisons and separate scope displays for the real and imaginary signal components.

Vector Signals

The generated cosimulation model flattens vector inputs. The following figure shows a `FromCosimSrc` subsystem that receives two vector input signals of dimension 2. The subsystem flattens the inputs into scalars before passing them to the subsystem outputs.



Generating a Cosimulation Model from the Command Line

To generate a cosimulation model from the command line, pass the `GenerateCosimModel` property to the `makehdltb` function. `GenerateCosimModel` takes one of the following property values:

- '`'ModelSim'`' : generate a cosimulation model configured for HDL Verifier for use with Mentor Graphics ModelSim.
- '`'Incisive'`' : generate a cosimulation model configured for HDL Verifier for use with Cadence Incisive.

In the following command, `makehdltb` generates a cosimulation model configured for HDL Verifier for use with Mentor Graphics ModelSim.

```
makehdltb('hdl_cosim_demo1/MAC', 'GenerateCosimModel', 'ModelSim');
```

Naming Conventions for Generated Cosimulation Models and Scripts

The naming convention for generated cosimulation models is

prefix_modelname_toolid_suffix, where:

- *prefix* is the string `gm`.
- *modelname* is the name of the generating model.
- *toolid* is an identifier indicating the HDL simulator chosen by the **Cosimulation model for use with:** option. Valid *toolid* strings are '`mq`' and '`in`'.
- *suffix* is an integer that provides each generated model with a unique name. The suffix increments with each successive test bench generation for a given model. For example, if the original model name is `test`, then the sequence of generated cosimulation model names is `gm_test_toolid_0`, `gm_test_toolid_1`, and so on.

The naming convention for generated cosimulation scripts is the same as that for models, except that the file name extension is `.m`.

Limitations for Cosimulation Model Generation

When you configure a model for cosimulation model generation, observe the following limitations:

- Explicitly specify the sample times of source blocks to the DUT in the simulation path. Use of the default sample time (-1) in the source blocks may cause sample time propagation problems in the cosimulation path of the generated model.
- The HDL Coder software does not support continuous sample times for cosimulation model generation. Do not use sample times 0 or Inf in source blocks in the simulation path.
- If you set **Clock Inputs** to **Multiple**, HDL Coder does not support generation of a cosimulation model.
- Combinatorial output paths (caused by absence of registers in the generated code) have a latency of one extra cycle in cosimulation. To avoid discrepancy in the comparison between the simulation and cosimulation outputs, the **Allow direct feedthrough** option on the **Ports** pane of the HDL Cosimulation block is automatically selected.

Alternatively, you can avoid the latency by specifying output pipelining (see “OutputPipeline” on page 22-17). This will fully register outputs during code generation.

- Double data types are not supported for the HDL Cosimulation block. Avoid use of double data types in the simulation path when generating HDL code and a cosimulation model.

HDL Verifier Cosimulation Model Generation in HDL Coder™

This example shows how to generate a cosimulation model in of HDL Coder and integrate the generated HDL code into an HDL Verifier™ workflow. Automation of cosimulation model generation enables seamless verification of the generated hardware design.

Quick Introduction

Cosimulation is a challenging task, especially with automatically generated code; one has to keep in sync various aspects of the source model including sample rates, feedforward/feedthrough systems, and other various parameters and settings used during code generation while setting up the HDL Verifier block and the target EDA Simulator.

The automated cosimulation model generation takes the guess-work out of the HDL cosimulation block and simulator setup by deciphering all the compiled model and code generation information; in addition all the automated settings are documented in the generated scripts. The end result is a cosimulation model ready to verify the generated code.

```
% >> docsearch('Code Generation for HDL Cosimulation Model')
```

Generating the Cosimulation Model

Let us take a simple accumulator design in Simulink and automatically generate a cosimulation model for it as a part of test bench generation.

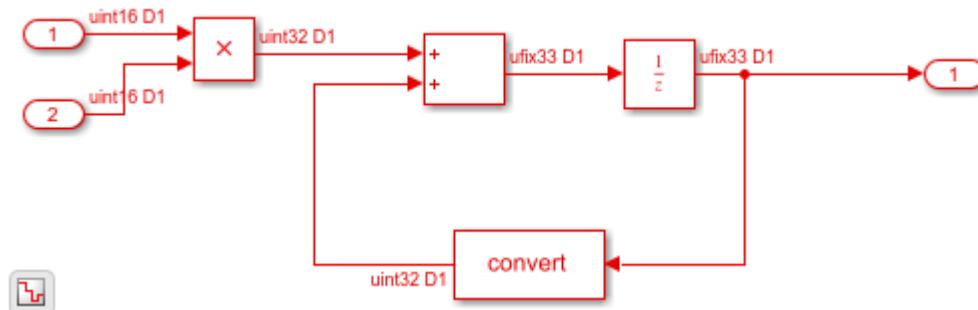
Multiply Accumulator Design in Simulink

Open the source design/model

```
bdclose all;
load_system('hdl_cosim_demo1')
open_system('hdl_cosim_demo1/MAC')

% Now generate vhdl code for the device under test 'MAC' in that
% model in the source Simulink model.
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vh')

### Generating HDL for 'hdl_cosim_demo1/MAC'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo1/MAC')>.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdl_cosim_demo1'.
### Working on hdl_cosim_demo1/MAC as hdlsrc/hdl_cosim_demo1/MAC.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdl_cosim_demo1' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



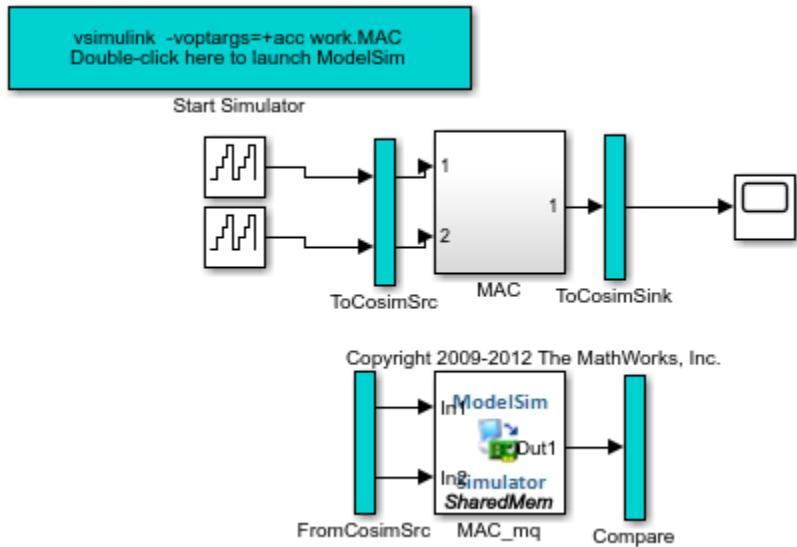
Generate HDL Test Bench with Cosimulation Model

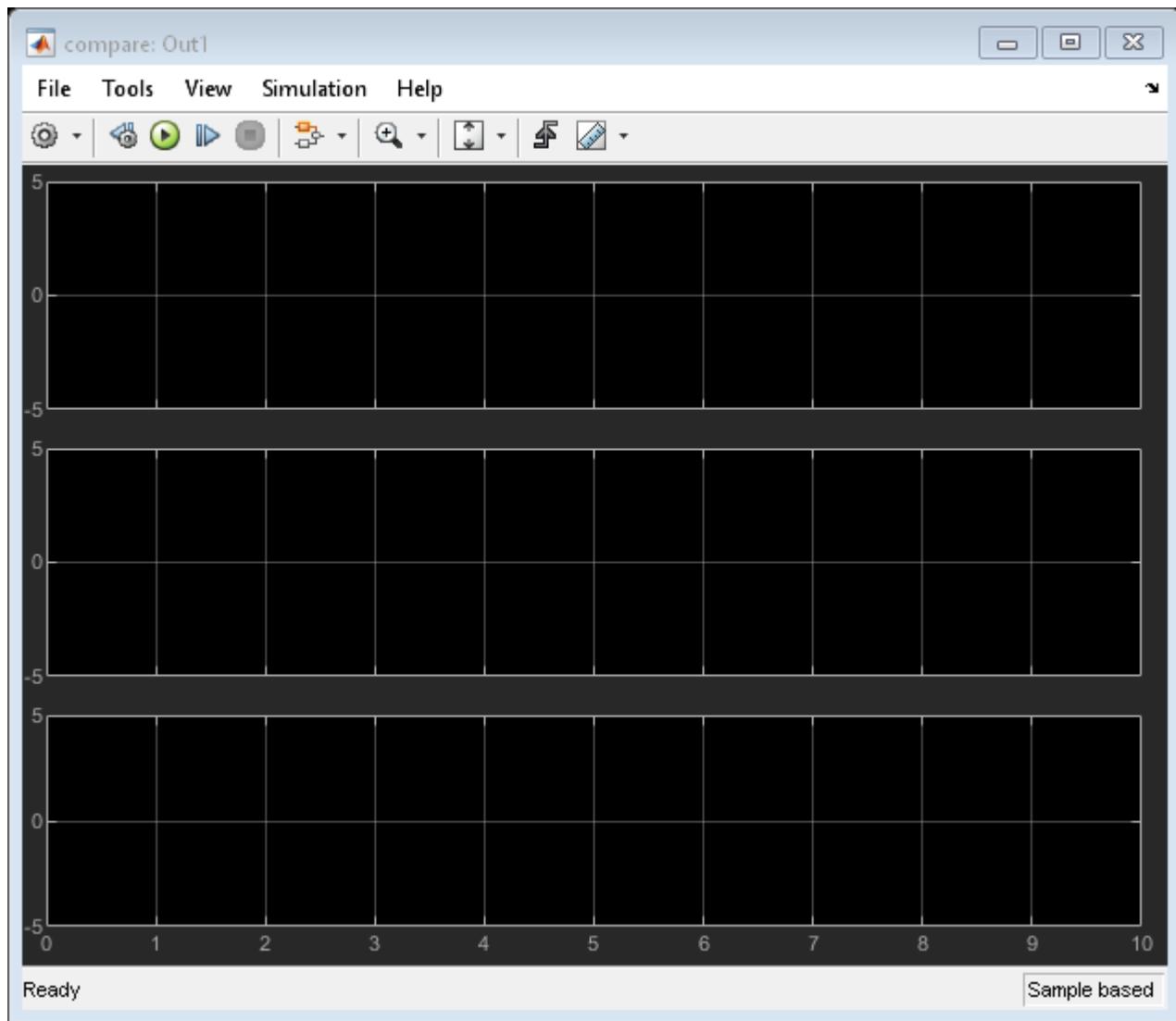
HDL Coder supports generation of cosimulation model with an HDL Verifier block for Mentor Graphics 'Modelsim' or Cadence 'Incisive'

```
% Now as a part of test bench generation specify that in addition to the
% textual based test bench a cosimulation model needs to be generated. Use
% the new makehdl parameter 'GenerateCosimModel' with value 'ModelSim' or
% 'Incisive' to choose between the two HDL Verifier blocks to generate the
% cosimulation model.
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim')

### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo1/MAC'.
### Generating new cosimulation model: <a href="matlab:open_system('gm_hdl_cosim_demo1_mq')">gm_
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_tcl.m
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_batch_t
### Note: Option 'Allow Direct Feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo1_mq/MAC_m
### Begin simulation of the model 'gm_hdl_cosim_demo1'...

### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In1.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In2.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\Out1_expected.dat.
### Working on MAC_tb as hdlsrc\hdl_cosim_demo1\MAC_tb.vhd.
### Generating package file hdlsrc\hdl_cosim_demo1\MAC_tb_pkg.vhd.
### HDL TestBench generation complete.
```





(Optional) Generate HDL Code Coverage Report and Database

To instrument the HDL Simulator to generate a code coverage database, either:

- On the 'HDL Code Generation > Test Bench' pane, select the check box labeled 'HDL code coverage'.
- When you call 'makehdltb', set 'HDLCodeCoverage' to 'on'. For example:

```
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim', 'HDLCodeCov')
```

The HDL code coverage artifacts are generated in the source directory after the test bench is simulated.

New Code Generation Messages

As you can see from the following additional code generation messages in the command window a cosimulation model 'gm_hdl_cosim_demo1_mq' is generated; In addition to the code generated in the

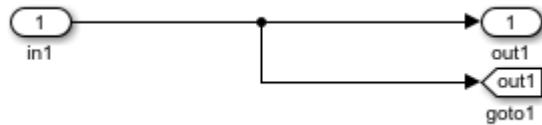
target directory 'hdlsrc' an additional cosimulation script 'gm_hdl_cosim_demo1_mq_tcl.m' is generated to prepare the target simulator for cosimulation with Simulink.

```
### Generating new cosimulation model: gm_hdl_cosim_demo1_mq
### Generating new cosimulation tcl script: hdlsrc/gm_hdl_cosim_demo1_mq_tcl.m
### Cosimulation Model Generation Complete.
```

Cosimulation Model Stimulus and Response Capture

As can be seen from the cosimulation model the original device under test (DUT) is intercepted by two subsystems 'ToCosimSrc' and 'ToCosimSink'; As shown below the purpose of these two subsystems is to capture the stimulus and the response of the DUT and use it for driving the cosimulation using 'Goto' blocks. The number of 'Goto' blocks in each of the following subsystem match the number of inputs and outputs of the DUT.

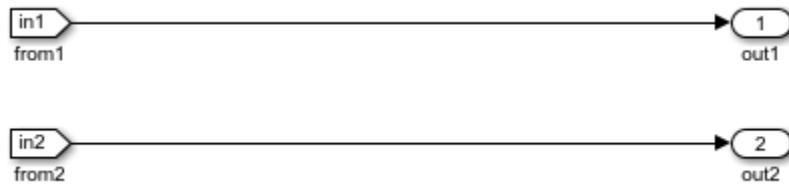
```
open_system('gm_hdl_cosim_demo1_mq/ToCosimSrc')
open_system('gm_hdl_cosim_demo1_mq/ToCosimSink')
```



Stimulus to the HDL Cosimulation Block

The stimulus that is originally driving the DUT is fed to the fully configured HDL cosimulation block using the 'From' block as shown below. In some cases input stimulus signals cannot be directly fed to the HDL Cosimulation block; for example the HDL Cosimulation block does not allow complex and vector signals and in such cases further massaging of the input stimulus signals is done automatically. In the current model the 'From' blocks directly feed the contents of corresponding 'Goto' blocks.

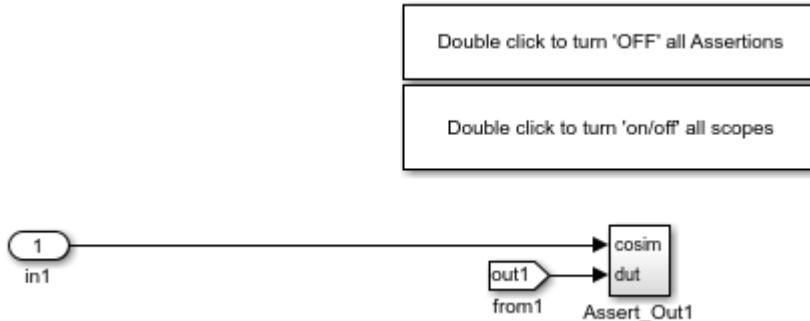
```
open_system('gm_hdl_cosim_demo1_mq/FromCosimSrc')
```



Comparison of the Results

The response from the original DUT is compared with the response from the HDL Cosimulation block in HDL Verifier using the Sink blocks provided by Simulink for visualization of the response data.

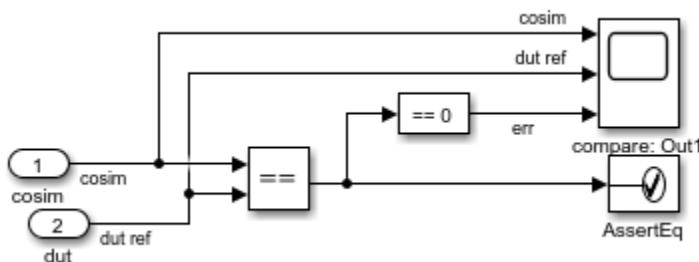
```
open_system('gm_hdl_cosim_demo1_mq/Compare')
```



Assertion Checking in the Generated Model

For each output of the device under test subsystem the following assertion-checking model is generated that checks the original output ('dut ref') with cosimulation output ('cosim') and generates assertion messages when the input to the assertion block detects a mismatch.

```
open_system('gm_hdl_cosim_demo1_mq/Compare/Assert_Out1')
```



Using Assertion Blocks

Assertions are enabled in the Assertion block but do not stop simulation. If as a part of cosimulation there are any assertions from the following block you should see a warning from the Assertion block:

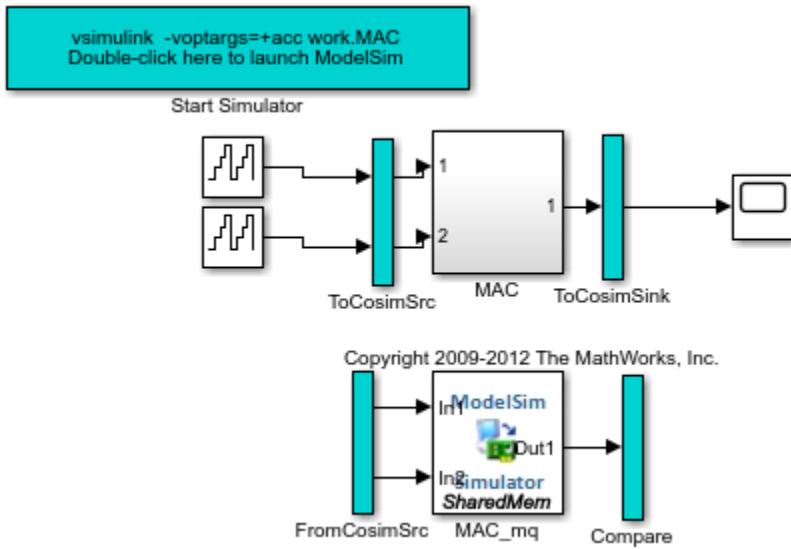
```
Warning: Assertion detected in 'gm_hdl_cosim_demo1_mq/Compare/Assert_Out1/AssertEq' at time 1.000
```

```
open_system('gm_hdl_cosim_demo1_mq/Compare/Assert_Out1/AssertEq')
```

HDL Cosimulation Block Setup

The HDL Cosimulation block is automatically populated with the compiled input output interface of the DUT. The 'Ports' panel is fully populated with 'Full HDL Name', 'Sample Time' and 'Data type' information. Similarly various HDL Cosimulation block setup parameters such as TimeScale and tcl port panes are automatically populated. Note that cosimulation model is always configured in the 'Shared Memory' connection method.

```
open_system('gm_hdl_cosim_demo1_mq/MAC_mq')
```



Target Simulator Launch and Setup

Now let's look at the automation associated with the launch and setup of the target simulator (ModelSim or Incisive). As can be seen in the top level of the generated model, a Subsystem with the name 'Start Simulator' is generated with the following callback function; this subsystem is used to launch the target simulator of choice.

```
get_param('gm_hdl_cosim_demo1_mq/Start Simulator', 'OpenFcn')
```

```
ans =
```

```
'try
cosimDirName = pwd;
cd 'hdlsrc\hdl_cosim_demo1';
vsim('tclstart',gm_hdl_cosim_demo1_mq_tcl);
cd (cosimDirName);
clear cosimDirName;
catch me
    disp('Failed to launch cosimulator with "vsim"');
    disp(me.message);
    cd (cosimDirName);
    clear cosimDirName;
end'
```

Simulation of the Cosimulation Model

The following script is executed on launch

```
vsim('tclstart',gm_hdl_cosim_demo1_mq_tcl)
```

The MATLAB command 'vsim' for ModelSim (or 'hdlsimulink' for Incisive) launches the target simulator from within MATLAB environment with the necessary setup for cosimulation. The 'vsim' command is invoked with the 'tclstart' option that accepts an tcl string that configures the simulator on its launch. The file 'gm_hdl_cosim_demo1_mq_tcl' is also automatically generated by HDL Coder along with the cosimulation model.

Contents of the Generated tclstart Command File

The generated tclstart file contains commands for configuring the launched simulator as well as comments about how various settings of Cosimulation model are generated.

```
type hdlsrc/hdl_cosim_demo1/gm_hdl_cosim_demo1_mq_tcl

%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%
% Source Model      : hdl_cosim_demo1
% Generated Model   : gm_hdl_cosim_demo1
% Cosimulation Model : gm_hdl_cosim_demo1_mq
%
% Source DUT        : gm_hdl_cosim_demo1_mq/MAC
% Cosimulation DUT   : gm_hdl_cosim_demo1_mq/MAC_mq
%
% File Location     : hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_mq_tcl.m
% Created           : 2020-11-10 13:50:55
%
% Generated by MATLAB 9.9 and HDL Coder 3.17
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% ClockName         : clk
% ResetName         : reset
% ClockEnableName   : clk_enable
%
% ClockLowTime      : 5ns
% ClockHighTime     : 5ns
% ClockPeriod       : 10ns
%
% ResetLength       : 20ns
% ClockEnableDelay  : 10ns
% HoldTime          : 2ns
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% ModelBaseSampleTime : 1
% DutBaseSampleTime  : 1
% OverClockFactor    : 1
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% Mapping of DutBaseSampleTime to ClockPeriod
%
% N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
% 1 sec in Simulink corresponds to 10ns in the HDL Simulator(N = 10)
%
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% ResetHighAt        : (ClockLowTime + ResetLength + HoldTime)
% ResetRiseEdge       : 27ns
% ResetType          : async
% ResetAssertedLevel : 1
%
```

```
% ClockEnableHighAt    : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
% ClockEnableRiseEdge  : 37ns
%%%%%%%%%%%%%%%
function tclCmds = gm_hdl_cosim_demo1_mq_tcl
tclCmds = {
    'do MAC_compile.do',...% Compile the generated code
    'vsimulink -voptargs=+acc work.MAC',...% Initiate cosimulation
    'add wave /MAC/clk',...% Add wave commands for chip input signals
    'add wave /MAC/reset',...
    'add wave /MAC/clk_enable',...
    'add wave /MAC/In1',...
    'add wave /MAC/In2',...
    'add wave /MAC/ce_out',...% Add wave commands for chip output signals
    'add wave /MAC/Out1',...
    'set UserTimeUnit ns',...% Set simulation time unit
    'puts ""',...
    'puts "Ready for cosimulation..."',...
};
end
```

Header Comments in the 'tclstart' File

At the top level, the comments specify the source and generated model names of the DUT portion of the model for which code is generated and being co-simulated. The cosimulation HDL Verifier DUT is placed in parallel with our generated model DUT, (which captures any modifications/changes to bit-true or cycle-accuracy of the original model DUT as a part of code generation)

Test Bench Options Affecting Cosimulation Model Generation

The next part of the tclstart script file shows all the makehdl tb test bench parameters supported by HDL Coder and their initial values used in cosimulation scripts.

```
ClockName, ResetName, ClockEnableName
ClockLowTime, ClockHighTime, ClockPeriod
ResetLength, ClockEnableDelay, HoldTime
```

Model Sample Times and Mapping of DutBaseSampleTime to ClockPeriod

The next part of the comment section covers sample times in the model and how they influenced clocking of the HDL Cosimulation block in HDL Verifier.

```
N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
1 sec in Simulink corresponds to 10ns in the HDL Simulator(N = 10)
```

Generated 'tclstart' Script Output

The function in 'gm_hdl_cosim_demo1_mq_tcl' generates the necessary tcl command string (tclCmds).

If the 'EDAScriptGeneration' option is turned 'on' and compilation do files are generated for ModelSim as part of 'makehdl', then a single 'do' command is generated. If the 'EDAScriptGeneration' option is turned 'off', then explicit compilation commands are added for compiling the generated HDL code for the DUT.

Wave commands are added for all top-level interface signals.

Pre-simulation Commands

In the HDL Cosimulation block, the "Pre-simulation Tcl commands" parameter contains force commands that drive the clock bundle (clock, clock-enable, reset). The "Time to run HDL simulator before cosimulation starts" parameter initiates simulation with a run time necessary to bring the chip out of reset.

```
get_param('gm_hdl_cosim_demo1_mq/MAC_mq','TclPreSimCommand')
```

```
ans =
'puts "Running Simulink Cosimulation block.";
 puts "Chip Name: --> hdl_cosim_demo1/MAC";
 puts "Target language: --> vhdl";
 puts "Target directory: --> hdlsrc\hdl_cosim_demo1";
 puts [clock format [clock seconds]];
# Clock force command;
force /MAC/clk 0 0ns, 1 5ns -r 10ns;
# Clock enable force command;
force /MAC/clk_enable 0 0ns, 1 37ns;
# Reset force command;
force /MAC/reset 1 0ns, 0 27ns;

'
```

Launching the Simulator

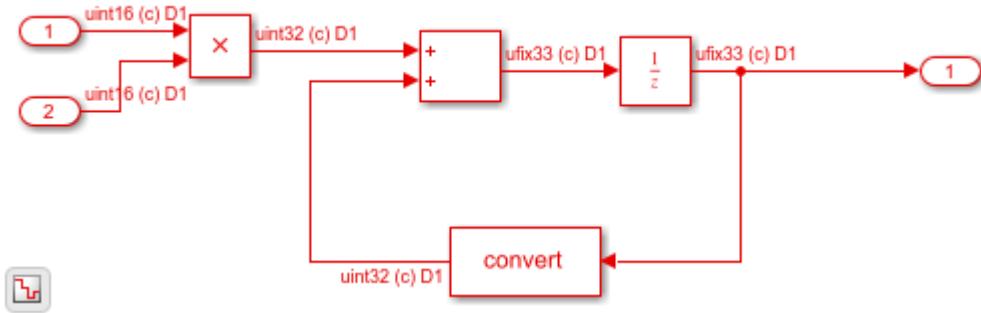
Double clicking the Start Simulator launches the simulator with the tcl commands in the generated 'tclstart' MATLAB script. Once the simulator is launched all the generated code is compiled and the HDL Cosimulation block is ready for simulation.

Support for Complex Signals

The model hdl_cosim_demo2 contains a complex MAC subsystem;

```
bdclose all;
load_system('hdl_cosim_demo2');
open_system('hdl_cosim_demo2/Complex MAC');
makehdl('hdl_cosim_demo2/Complex MAC', 'targetlang', 'vh');

### Generating HDL for 'hdl_cosim_demo2/Complex MAC'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo2')">.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdl_cosim_demo2'.
### Working on hdl_cosim_demo2/Complex MAC as hdlsrc\hdl_cosim_demo2\Complex_MAC.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdl_cosim_demo2' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



Cosimulation Model for Complex MAC

Let's generate a cosimulation model as a part of test bench generation and observe the stimulus part of cosimulation model:

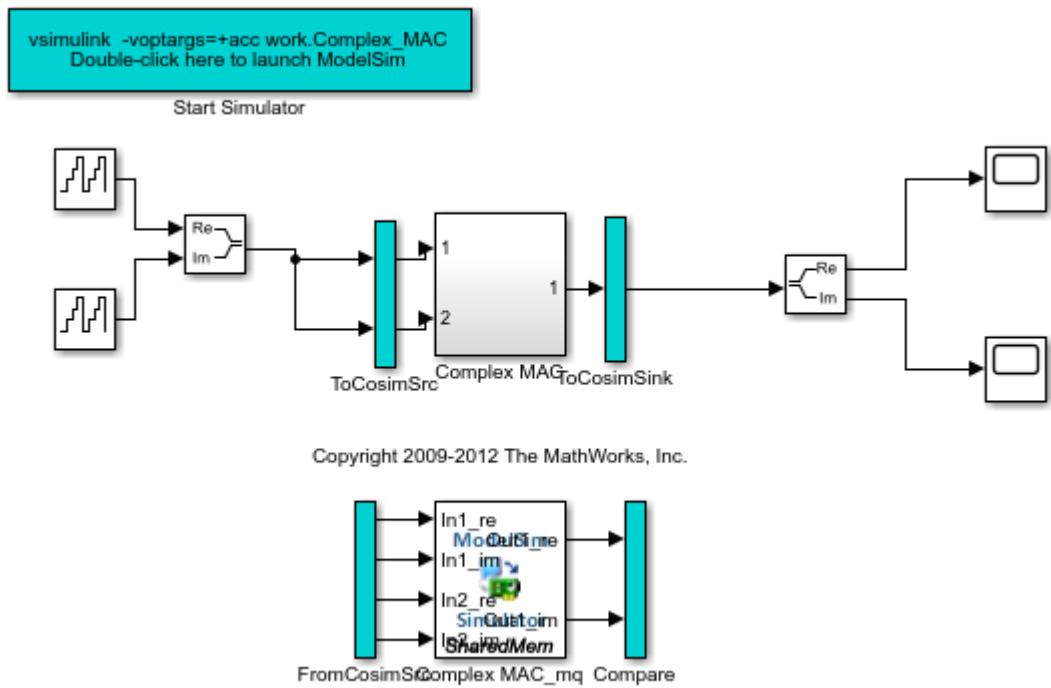
```

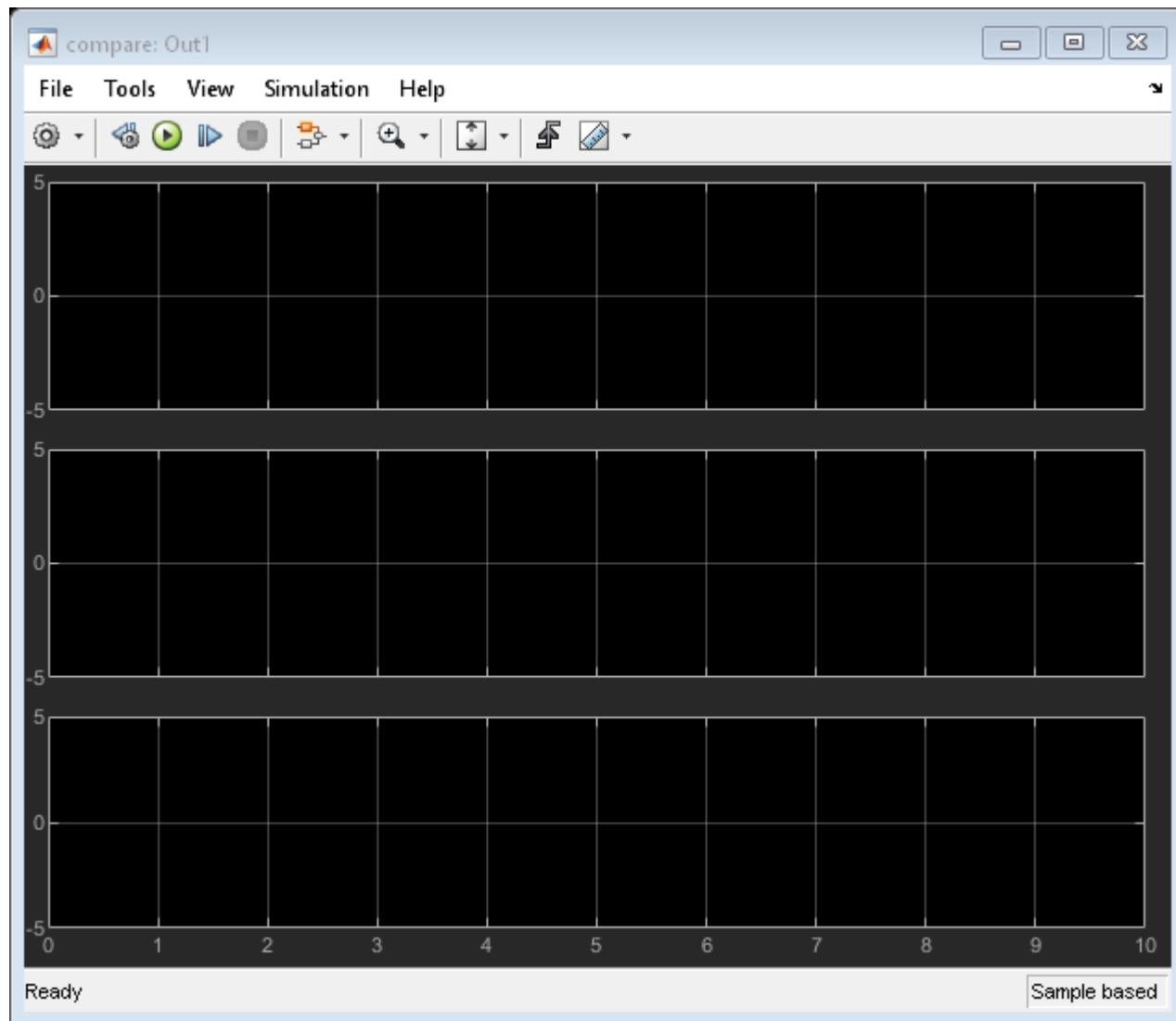
makehdltb('hdl_cosim_demo2/Complex MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim')

### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo2/Complex MAC'.
### Generating new cosimulation model: <a href="matlab:open_system('gm_hdl_cosim_demo2_mq')">gm...
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo2\gm_hdl_cosim_demo2_mq_tcl.m
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo2\gm_hdl_cosim_demo2_mq_batch_t...
### Note: Option 'Allow Direct Feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo2_mq/Comple...
### Begin simulation of the model 'gm_hdl_cosim_demo2'...

### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo2\In1_re.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo2\In1_im.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo2\Out1_im_expected.dat.
### Working on Complex_MAC_tb as hdlsrc\hdl_cosim_demo2\Complex_MAC_tb.vhd.
### Generating package file hdlsrc\hdl_cosim_demo2\Complex_MAC_tb_pkg.vhd.
### HDL TestBench generation complete.

```

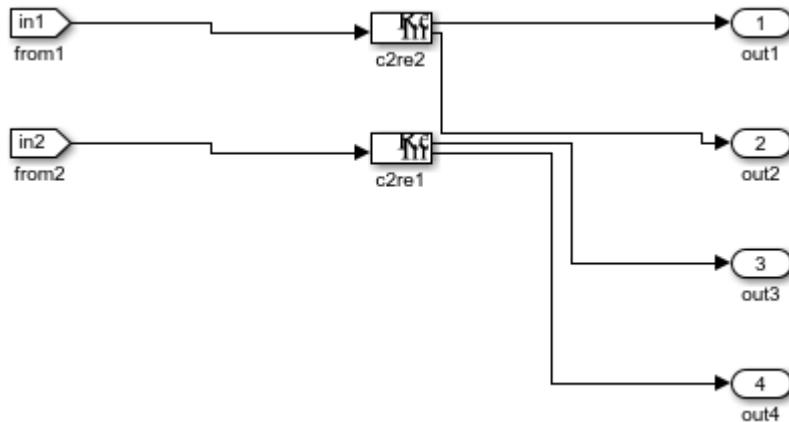




Generated Complex FromCosimSrc Subsystems

The input complex signal is automatically broken into real and imaginary pieces before driving the HDL Cosimulation block.

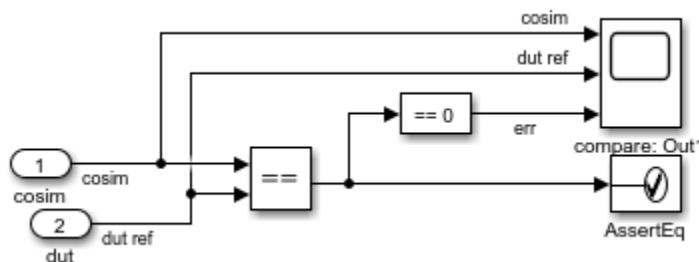
```
open_system('gm_hdl_cosim_demo2_mq/FromCosimSrc')
```



Generated Complex Compare Subsystems

The comparison section checks the results for real and imaginary parts of complex outputs separately.

```
open_system('gm_hdl_cosim_demo2_mq/Compare/Assert_Out1')
```

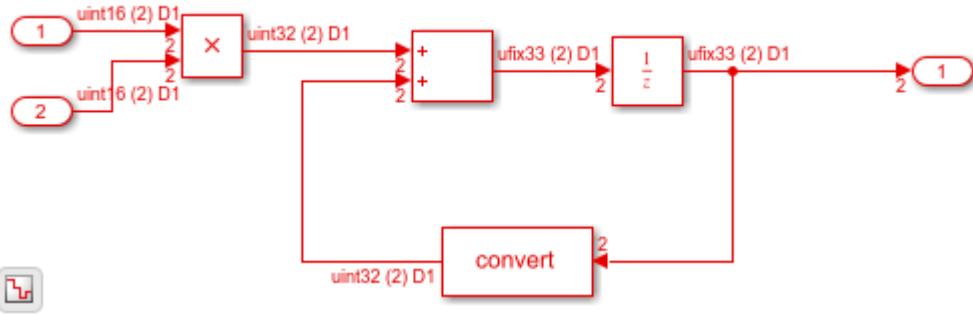


Support for Vector Signals

The model hdl_cosim_demo3 contains a Vector MAC subsystem;

```
bdclose all;
load_system('hdl_cosim_demo3');
open_system('hdl_cosim_demo3/Vector MAC');
makehdl('hdl_cosim_demo3/Vector MAC', 'targetlang', 've');

### Generating HDL for 'hdl_cosim_demo3/Vector MAC'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo3')".
### Starting HDL check.
### Begin Verilog Code Generation for 'hdl_cosim_demo3'.
### Working on hdl_cosim_demo3/Vector MAC as hdlsrc\hdl_cosim_demo3\Vector_MAC.v.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdl_cosim_demo3' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
```



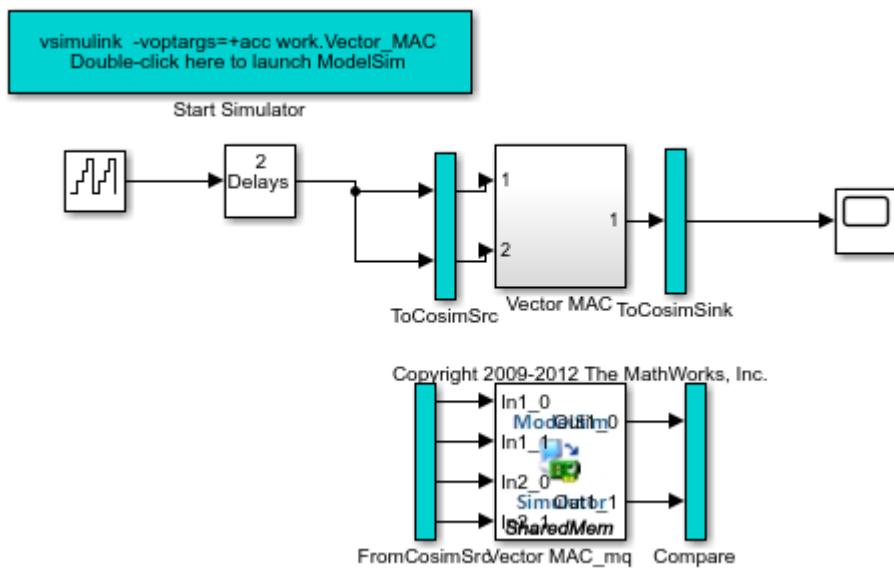
Cosimulation Model for Vector MAC

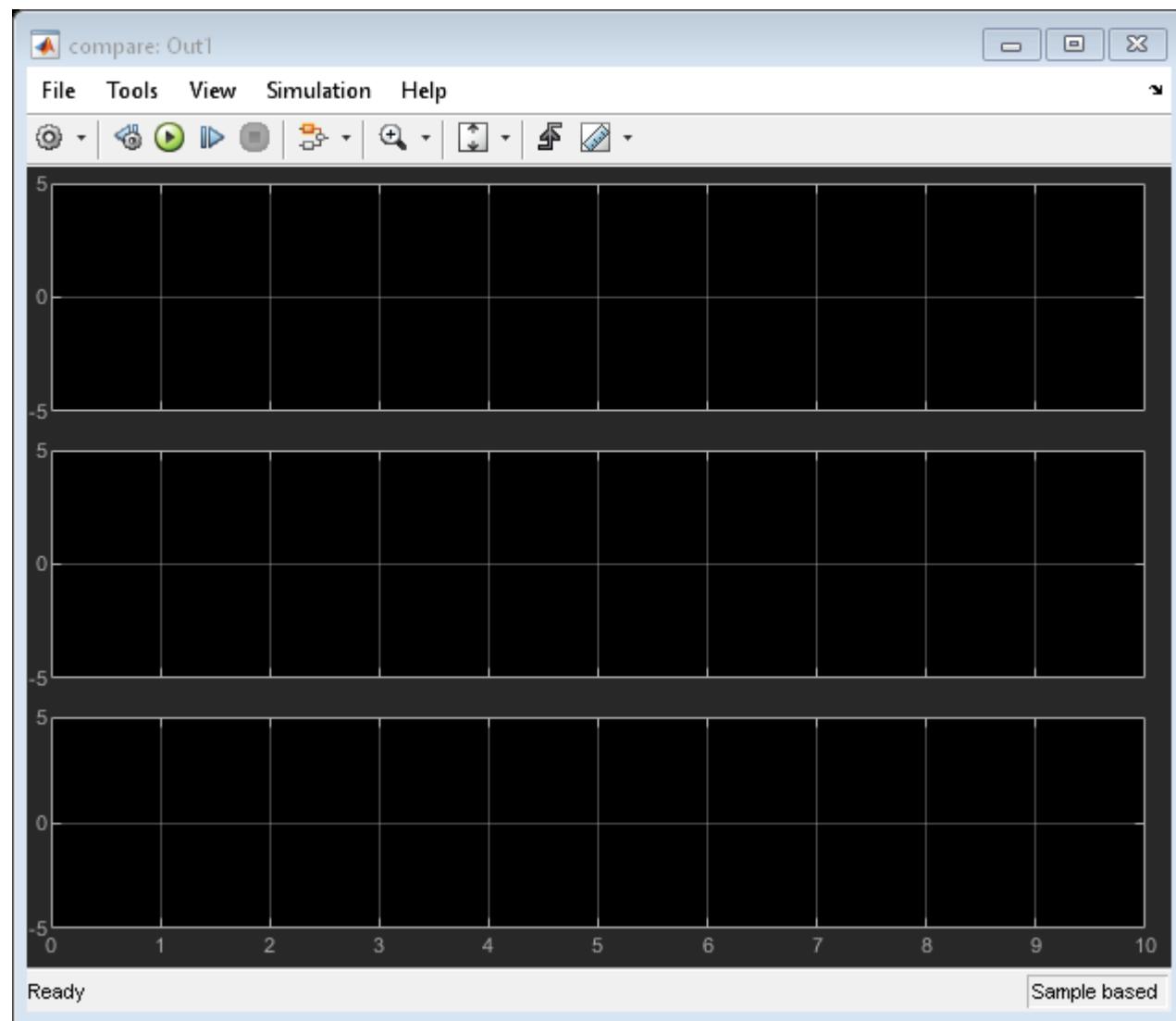
Let's generate cosimulation model as a part of test bench generation and observe the stimulus part of cosimulation model for vector signals in 'verilog' where we flatten the vector signals for code generation.

```
makehdltb('hdl_cosim_demo3/Vector MAC', 'targetlang', 've', 'GenerateCosimModel', 'ModelSim')

### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo3/Vector MAC'.
### Generating new cosimulation model: <a href="matlab:open_system('gm_hdl_cosim_demo3_mq')">gm_hdl_cosim_demo3_mq</a>
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo3\gm_hdl_cosim_demo3_mq_tcl.m.
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo3\gm_hdl_cosim_demo3_mq_batch_tcl.m.
### Note: Option 'Allow Direct Feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo3_mq/Vector MAC'
### Begin simulation of the model 'gm_hdl_cosim_demo3'...

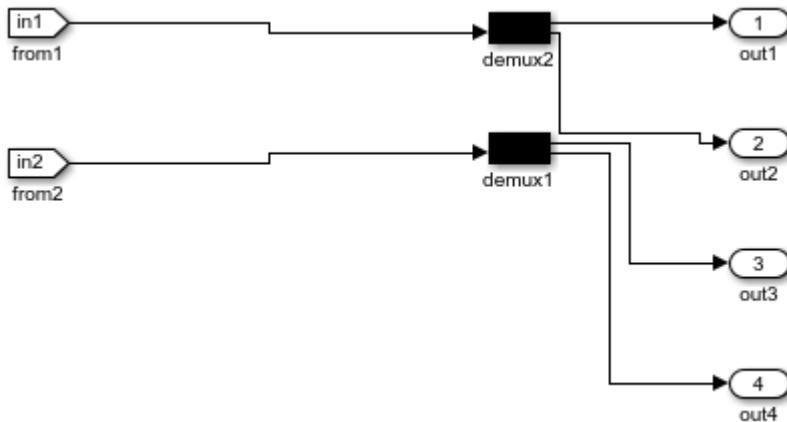
### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo3\In1_0.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo3\In1_1.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo3\Out1_0_0_expected.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo3\Out1_0_1_expected.dat.
### Working on Vector_MAC_tb as hdlsrc\hdl_cosim_demo3\Vector_MAC_tb.v.
### HDL TestBench generation complete.
```





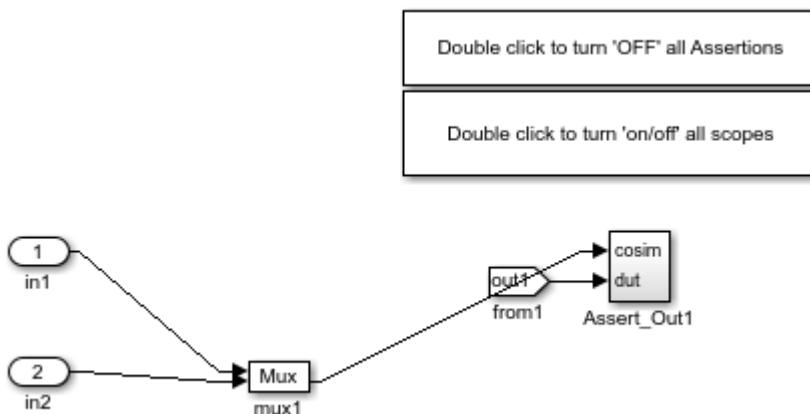
Generated Vector FromCosimSrc Subsystem

```
open_system('gm_hdl_cosim_demo3_mq/FromCosimSrc')
```



Generated Vector Compare Subsystem

```
open_system('gm_hdl_cosim_demo3_mq/Compare')
```



Support for Local Multi-Rate

The model hdl_cosim_demo4 contains a MAC subsystem with a Sum of Elements block that is configured with a Cascade implementation and requires overclocking as can be seen in the Code generation messages.

```

bdclose all;
load_system('hdl_cosim_demo4');
open_system('hdl_cosim_demo4/LocalMR MAC');
makehdl('hdl_cosim_demo4/LocalMR MAC', 'targetlang', 'vh');
makehdltb('hdl_cosim_demo4/LocalMR MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'ModelSim');

### Generating HDL for 'hdl_cosim_demo4/LocalMR MAC'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo4'
### Starting HDL check.
### 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 addit
### Output port 0: 1 cycles.
### Begin VHDL Code Generation for 'hdl_cosim_demo4'.

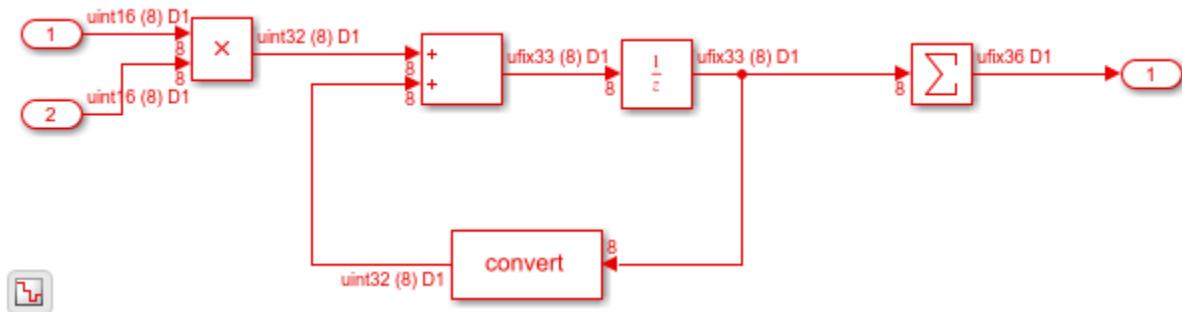
```

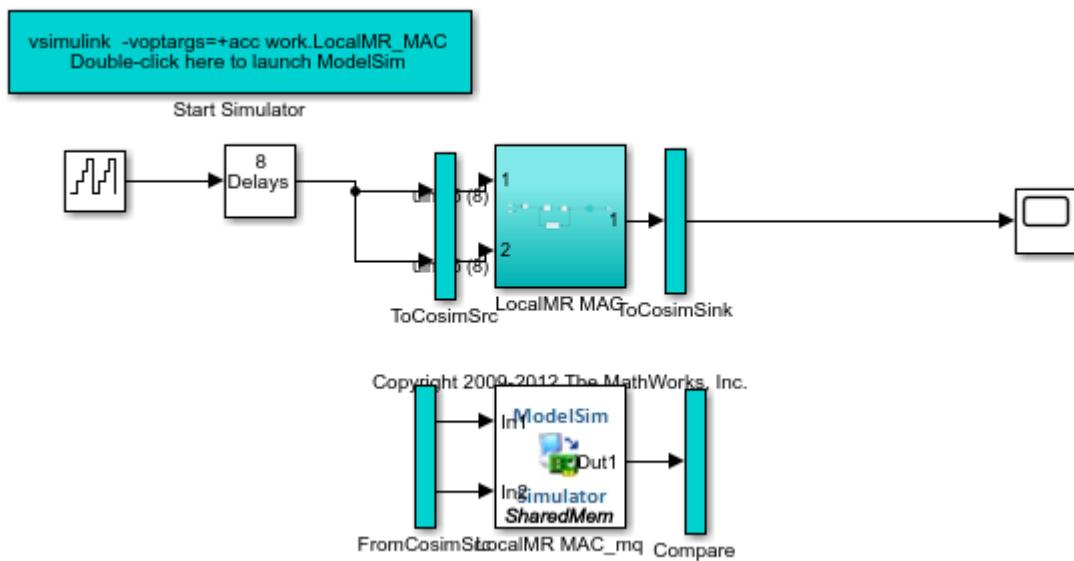
```

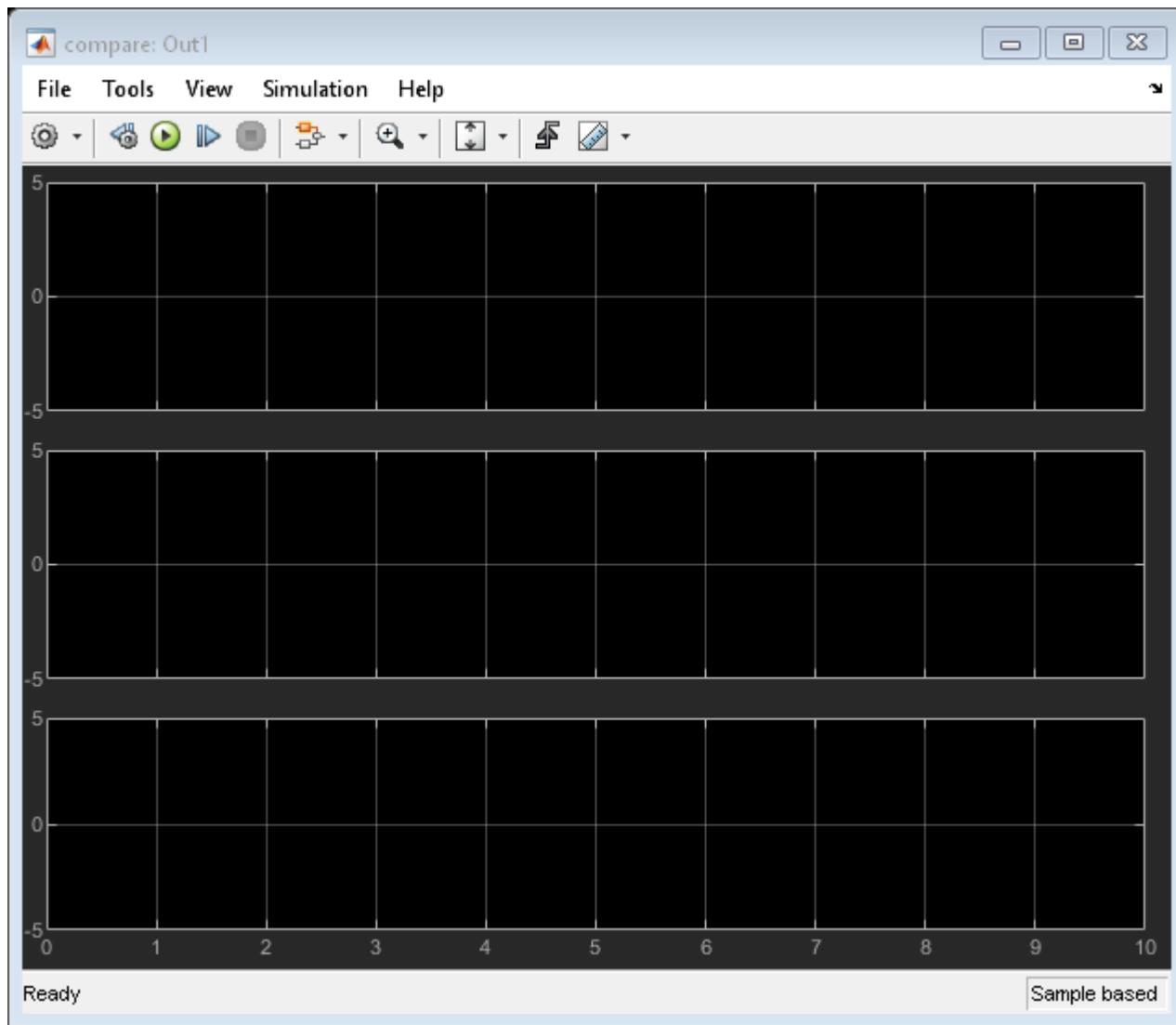
### MESSAGE: The design requires 5 times faster clock with respect to the base rate = 1.
### Working on hdl_cosim_demo4\LocalMR MAC/Sum of Elements/serial_sum_operation as hdlsrc\hdl_cosim_demo4\LocalMR_MAC\Sum_of_Elements.vhd.
### Working on hdl_cosim_demo4\LocalMR MAC/Sum of Elements as hdlsrc\hdl_cosim_demo4\Sum_of_Elements.vhd.
### Working on LocalMR MAC_tc as hdlsrc\hdl_cosim_demo4\LocalMR_MAC_tc.vhd.
### Working on hdl_cosim_demo4\LocalMR MAC as hdlsrc\hdl_cosim_demo4\LocalMR_MAC.vhd.
### Generating package file hdlsrc\hdl_cosim_demo4\LocalMR_MAC_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpt...
### HDL check for 'hdl_cosim_demo4' complete with 0 errors, 0 warnings, and 3 messages.
### HDL code generation complete.
### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo4\LocalMR MAC'.
### Generating new cosimulation model: <a href="matlab:open_system('gm_hdl_cosim_demo4_mq')">gm_hdl_cosim_demo4_mq</a>
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo4\gm_hdl_cosim_demo4_mq_tcl.m.
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo4\gm_hdl_cosim_demo4_mq_batch_tcl.m.
### Note: Option 'Allow Direct Feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo4_mq/LocalMR_MAC'.
### Begin simulation of the model 'gm_hdl_cosim_demo4' ...

### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo4\In1.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo4\Out1_expected.dat.
### Working on LocalMR_MAC_tb as hdlsrc\hdl_cosim_demo4\LocalMR_MAC_tb.vhd.
### Generating package file hdlsrc\hdl_cosim_demo4\LocalMR_MAC_tb_pkg.vhd.
### HDL TestBench generation complete.

```







Time-Scale Setting Updates to Offset Overclocking

The code generation messages show an overclocking that require a five times faster clock with respect to base rate of the model. This info is encapsulated in the cosimulation model as a part of the time scale setting as per the following message

```
N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
1 sec in Simulink corresponds to 50ns in the HDL Simulator(N = 50)
```

Support for Incisive

```
bdclose all;
load_system('hdl_cosim_demo1')
makehdl('hdl_cosim_demo1/MAC', 'targetlang', 'vh')
makehdltb('hdl_cosim_demo1/MAC', 'targetlang', 'vh', 'GenerateCosimModel', 'Incisive')
type hdlsrc/hdl_cosim_demo1/gm_hdl_cosim_demo1_in_tcl
bdclose all;
```

```
% close Modelsim
%tclHdlsim('after 1000 quit -f');

### Generating HDL for 'hdl_cosim_demo1/MAC'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdl_cosim_demo1')">.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdl_cosim_demo1'.
### Working on hdl_cosim_demo1/MAC as hdlsrc\hdl_cosim_demo1\MAC.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdl_cosim_demo1' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.
### Begin TestBench generation.
### Generating HDL TestBench for 'hdl_cosim_demo1/MAC'.
### Generating new cosimulation model: <a href="matlab:open_system('gm_hdl_cosim_demo1_in')">gm_hd...
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_in_tcl.m.
### Generating new cosimulation tcl script: hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_in_batch_t...
### Note: Option 'Allow Direct Feedthrough' has been set to 'on' on 'gm_hdl_cosim_demo1_in/MAC_in...
### Begin simulation of the model 'gm_hdl_cosim_demo1'...

### Collecting data...
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In1.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\In2.dat.
### Generating test bench data file: hdlsrc\hdl_cosim_demo1\Out1_expected.dat.
### Working on MAC_tb as hdlsrc\hdl_cosim_demo1\MAC_tb.vhd.
### Generating package file hdlsrc\hdl_cosim_demo1\MAC_tb_pkg.vhd.
### HDL TestBench generation complete.

%%%%%%%%%%%%%
% Auto generated cosimulation 'tclstart' script
%%%%%%%%%%%%%
% Source Model      : hdl_cosim_demo1
% Generated Model   : gm_hdl_cosim_demo1
% Cosimulation Model : gm_hdl_cosim_demo1_in
%
% Source DUT        : gm_hdl_cosim_demo1_in/MAC
% Cosimulation DUT   : gm_hdl_cosim_demo1_in/MAC_in
%
% File Location     : hdlsrc\hdl_cosim_demo1\gm_hdl_cosim_demo1_in_tcl.m
% Created           : 2020-11-10 13:51:42
%
% Generated by MATLAB 9.9 and HDL Coder 3.17
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% ClockName         : clk
% ResetName         : reset
% ClockEnableName   : clk_enable
%
% ClockLowTime      : 5ns
% ClockHighTime     : 5ns
% ClockPeriod       : 10ns
%
% ResetLength        : 20ns
% ClockEnableDelay   : 10ns
% HoldTime          : 2ns
%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%
% ModelBaseSampleTime : 1
% DutBaseSampleTime : 1
% OverClockFactor : 1
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% Mapping of DutBaseSampleTime to ClockPeriod
%
% N = (ClockPeriod / DutBaseSampleTime) * OverClockFactor
% 1 sec in Simulink corresponds to 10ns in the HDL Simulator(N = 10)
%
%%%%%%%%%%%%%

%%%%%%%%%%%%%
% ResetHighAt : (ClockLowTime + ResetLength + HoldTime)
% ResetRiseEdge : 27ns
% ResetType : async
% ResetAssertedLevel : 1
%
% ClockEnableHighAt : (ClockLowTime + ResetLength + ClockEnableDelay + HoldTime)
% ClockEnableRiseEdge : 37ns
%%%%%%%%%%%%%

function tclCmds = gm_hdl_cosim_demo1_in_tcl
tclCmds = {
    'exec ncvhdl -v93 MAC.vhd', ... % Compile the generated code
    'exec ncelab -access +wc MAC', ...
    ['hdlsimulink -gui MAC', ... %Comment: Initiate cosimulation
     '-input "{@simvision {set w \[waveform new\]}}"', ... % Add wave commands for chip input signals
     '-input "{@simvision {waveform add -using \$w -signals :clk}}"', ...
     '-input "{@probe -create -shm clk }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :reset}}"', ...
     '-input "{@probe -create -shm reset }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :clk_enable}}"', ...
     '-input "{@probe -create -shm clk_enable }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :In1}}"', ...
     '-input "{@probe -create -shm In1 }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :In2}}"', ...
     '-input "{@probe -create -shm In2 }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :ce_out}}"', ... % Add wave commands for chip output signals
     '-input "{@probe -create -shm ce_out }"', ...
     '-input "{@simvision {waveform add -using \$w -signals :Out1}}"', ...
     '-input "{@probe -create -shm Out1 }"', ...
     '-input "{@database -open waves -into waves.shm -default}"', ...
     '-input "{@puts \"\"}"', ...
     '-input "{@puts \"Ready for cosimulation...\"}"', ...
    ]
};

end
```

Verify HDL Design Using SystemVerilog DPI Test Bench

This example shows how to use SystemVerilog DPI test bench for verification of HDL code where a large data set is required.

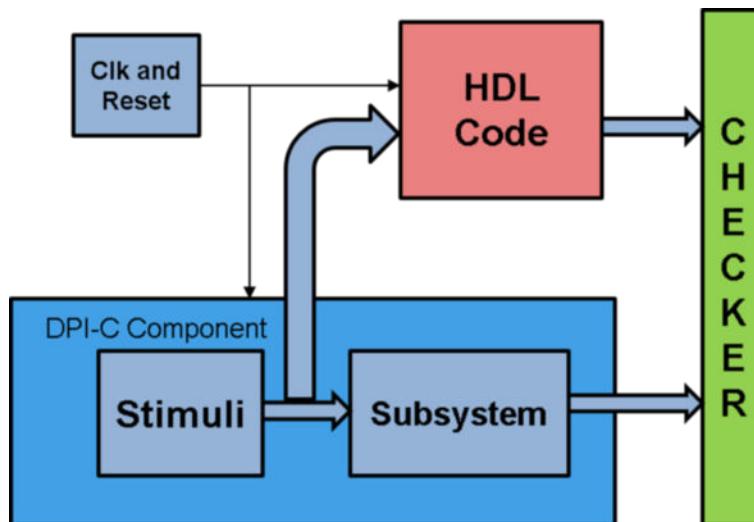
In certain applications, simulation of a large number of samples is required to verify the HDL code generated by HDL Coder™ for your algorithm. For instance, these applications require a large number of samples for algorithm verification :

- a) Calculation of radar astronomy frequency channels using a polyphase filter bank.
- b) Obtaining the Bit Error Rate (BER) from a Viterbi decoder in a communications system.
- c) Pixel-streaming video processing algorithms on high-resolution video.

Generating an HDL test bench to verify such a design is time consuming because the coder must simulate the model in Simulink to capture the test bench data.

A faster generated test bench alternative is the HDL Verifier™ SystemVerilog DPI test bench. The SystemVerilog DPI test bench does not require a Simulink simulation, so for large data sets it generates a test bench in a shorter time than the HDL test bench.

HDL Verifier™ SystemVerilog DPI test bench integrates with Simulink Coder™ to export a Simulink system as generated C code inside a SystemVerilog component with a Direct Programming Interface (DPI). Within the DPI-C component the stimuli is generated and applied to the C subsystem and also applied to the generated HDL code for the Simulink system. The test bench compares the output of the HDL simulation with the output of the DPI-C component to verify the HDL design.

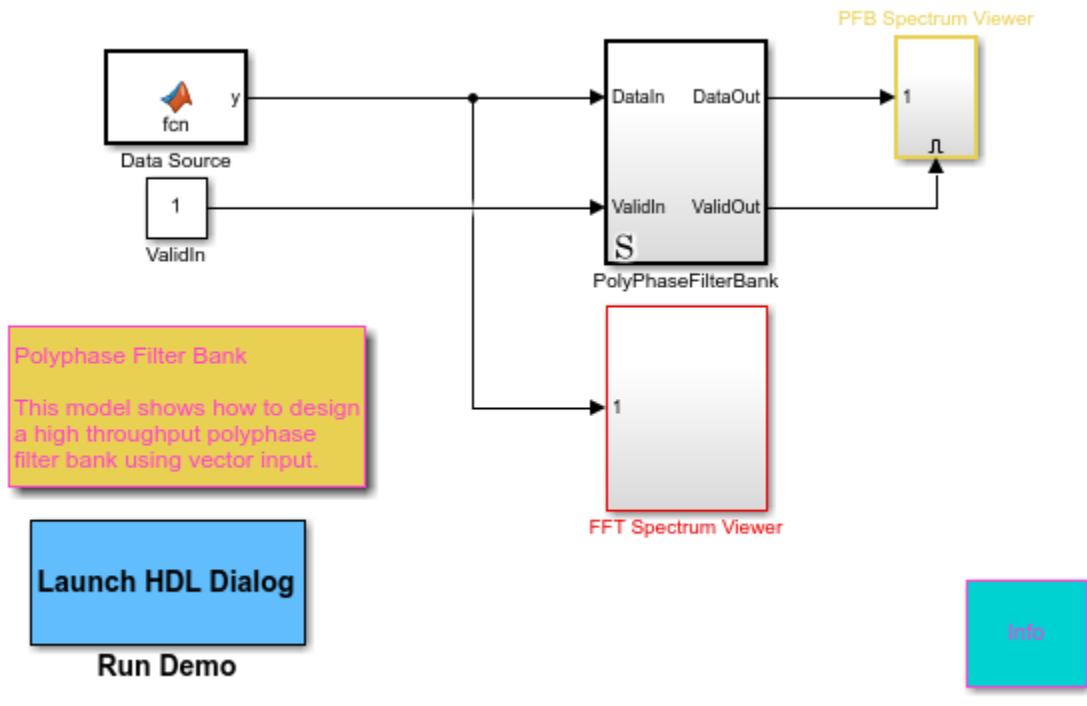


Polyphase Filter Bank

Polyphase filter bank is a widely used technique to reduce inaccuracy in FFT due to leakage and scalloping losses. Polyphase filter bank produces a flatter response as compared to a normal DFT by suppressing out-of-band signals significantly.

The model is a Polyphase Filter Bank which consists of a filter and an FFT that processes 16 samples at a time. For more information about the polyphase filter bank see "High Throughput Channelizer for FPGA" on page 11-15.

```
modelname = 'hdlcoder_DPIC_testbench';
open_system(modelname);
```



Copyright 2016 The MathWorks, Inc.

Set up the Model

The InitFcn callback(Model Properties > Callbacks > InitFcn) sets up the model. In this example, a 512-point FFT with a four tap filter for each band is used. The dsp.Channelizer object is used to generate the coefficients.

The algorithm requires 512 filters (one filter for each band). For a vector input of 16 samples the filter implementation shares 16 filters, 32 times. The input data consists of two sine waves, 200KHz and 250 KHz.

Generate HDL Code, HDL Test Bench, and SystemVerilog DPI Test Bench

Use a temporary directory for the generated files:

```
workingdir = tempname;
```

Check the PolyphaseFilterBank subsystem for HDL code generation compatibility:

```
checkhdl('hdlcoder_DPIC_testbench/PolyPhaseFilterBank','TargetDirectory',workingdir);
```

Run the following command to generate HDL code:

```
makehdl('hdlcoder_DPIC_testbench/PolyPhaseFilterBank','TargetDirectory',workingdir);
```

Run the following command to generate the test bench:

```
makehdltb('hdlcoder_DPIC_testbench/PolyPhaseFilterBank','TargetDirectory',workingdir);
```

This will generate an HDL test bench by simulating the model in Simulink and then capturing the test bench data.

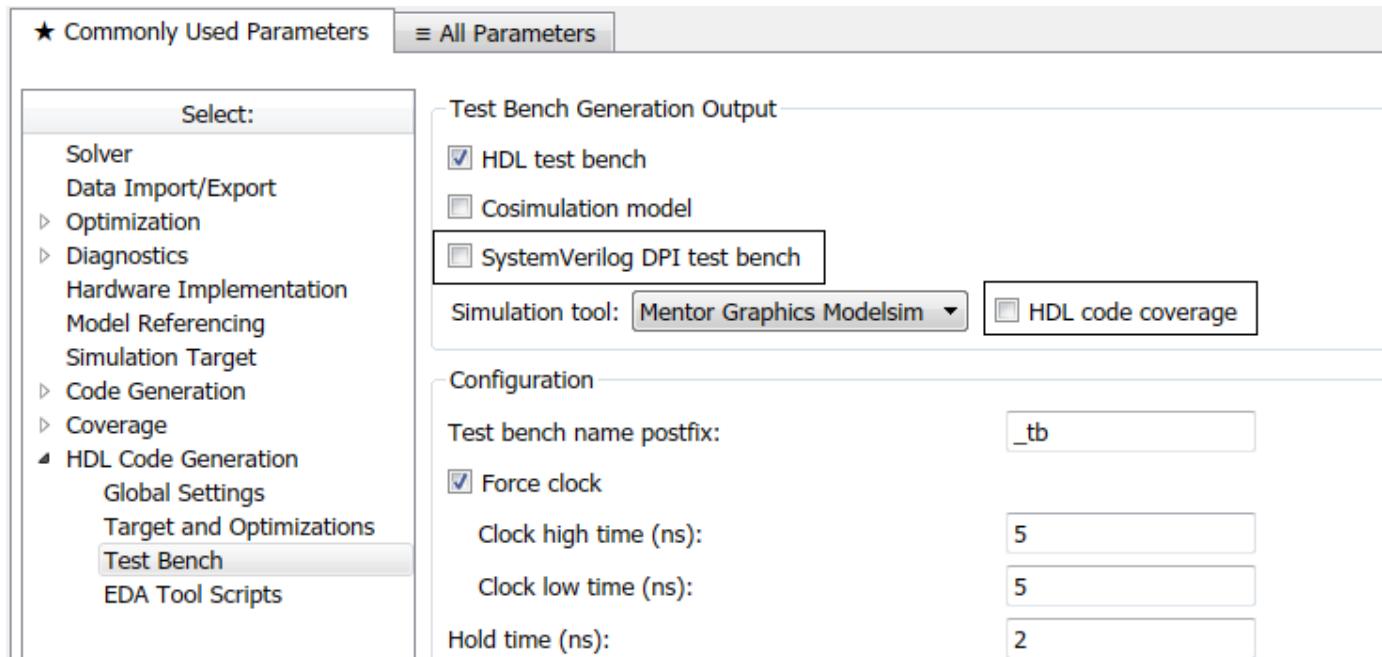
Run the following command to generate SystemVerilog DPI test bench:

```
HDLSimulator = 'ModelSim'; % Supported Simulator Options = 'ModelSim', 'Incisive', 'VCS', 'Vivado'
makehdltb('hdlcoder_DPI_C_testbench/PolyPhaseFilterBank', 'TargetDirectory', workingdir, 'GenerateSV')
```

This command generates a SystemVerilog test bench without running a Simulink simulation. Instead of a simulation, the code exports the Simulink system as generated C code inside a SystemVerilog component. The test bench verifies the output data by comparing it with the output of the HDL design. The makehdltb function also generates simulator-specific scripts for compilation and simulation.

SystemVerilog DPI test bench can be used to verify HDL designs of both target languages - VHDL and Verilog.

Alternatively, you can set SystemVerilog DPI test bench options on the 'HDL Code Generation > Test Bench' pane in Configuration Parameters.



Generated SystemVerilog DPI Test Bench Artifacts

When you request a SystemVerilog DPI test bench, the coder generates the following artifacts:

- PolyPhaseFilterBank_dpi_tb.sv - This is the SystemVerilog test bench that verifies the HDL code.
- PolyPhaseFilterBank_dpi_tb.do - This is the macro file that Mentor Graphics ModelSim® uses to compile the HDL code and run the test bench simulation.

Based on the selected simulator, the coder generates a different file for compilation and test bench simulation. For instance, if you select 'Incisive', the coder generates 'PolyPhaseFilterBank_dpi_tb.sh' for compilation and simulation on Cadence Incisive®.

(Optional) Generate HDL Code Coverage Report and Database

To instrument the HDL Simulator to generate a HDL code coverage report and database, either:

a.) On the 'HDL Code Generation > Test Bench' pane, select the check box labeled 'HDL code coverage'.

b.) When you call 'makehdltb', set 'HDLCodeCoverage' to 'on'. For example:

```
makehdltb('hdlcoder_DPICTestbench/PolyPhaseFilterBank', 'TargetDirectory', workingdir, 'GenerateS...
```

The HDL code coverage artifacts are generated in the source directory after the test bench is simulated.

Generation Time Comparison of HDL Test Bench and SystemVerilog DPI Test Bench

The simulation time of the model is set in the pre-load callback (Model Properties > Callbacks > PreLoadFcn)

```
simTime = 1000;
```

The sampling frequency is $2e+6$ Hz, which means that the simulation to generate the HDL testbench collects $2e+9$ samples.

For certain applications, it takes more samples to obtain the right frequency from the polyphase filter. An increase in required simTime would also increase the time required to generate an HDL test bench.

A solution for such applications is to use the SystemVerilog DPI test bench. The generation time for the test bench remains the same no matter how many samples your test scenario requires.

You can increase the Simulation Time by changing the 'simTime' variable. For instance to generate an HDL test bench for $2e+12$ samples, set:

```
simTime = 1000000;
```

The table shows a comparison of time taken (in seconds) for generation of HDL test bench and SystemVerilog DPI test bench for increasing numbers of samples (from $2e+9$ to $2e+15$) :

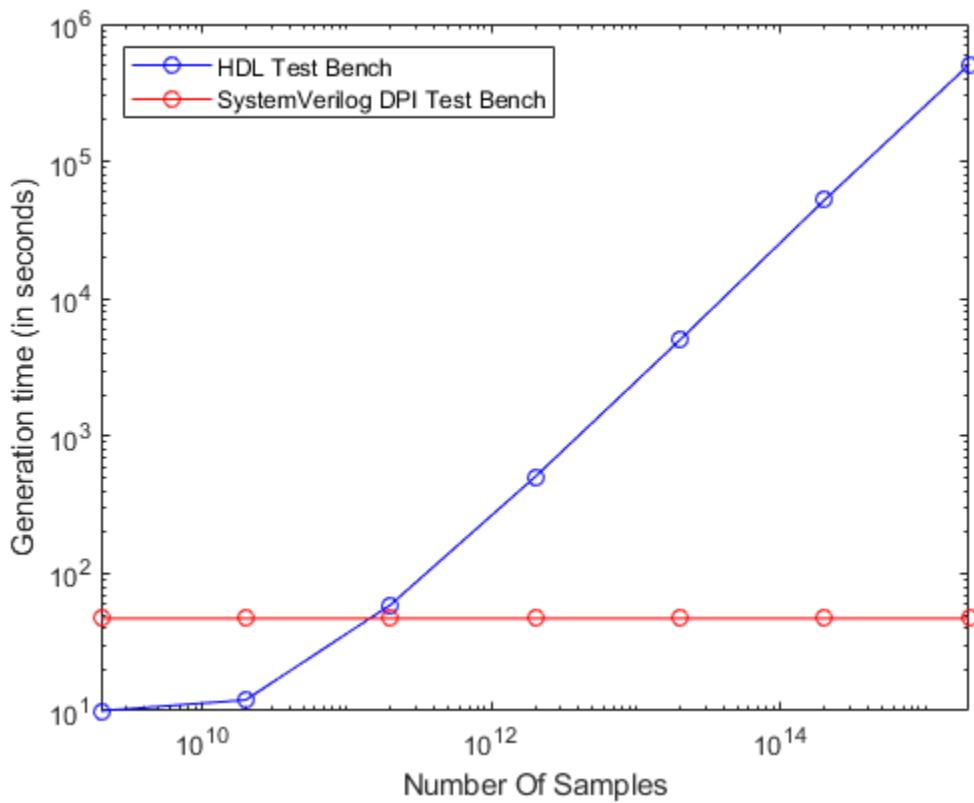
```
columns = {'NumberOfSamples'; 'GenerationTimeHDLTestBench'; 'GenerationTimeSystemVerilogDPITestben...
```

```
numSamples = [2e9; 2e10; 2e11; 2e12; 2e13; 2e14; 2e15];
HDLTBtime= [10; 12; 59; 504; 4994; 52200; 505506];
DPICTBtime=[47; 47; 47; 47; 47; 47];
CompareTestBenchTimes = table(numSamples,HDLTBtime,DPICTBtime,'VariableNames',columns);
disp(CompareTestBenchTimes);
```

NumberOfSamples	GenerationTimeHDLTestBench	GenerationTimeSystemVerilogDPITestbench
2e+09	10	47
2e+10	12	47
2e+11	59	47
2e+12	504	47
2e+13	4994	47
2e+14	52200	47
2e+15	5.0551e+05	47

A log plot of generation time for both these test bench types with respect to the Number of samples, shows that while HDL test bench requires more generation time with an increase in the number of samples, generation time for the SystemVerilog DPI test bench remains constant irrespective of the number of samples.

```
loglog(numSamples,HDLTBtime,'b-o', numSamples,DPICTBtime, 'r-o' );
xlin([2e09 2e15]);
legend('HDL Test Bench','SystemVerilog DPI Test Bench','Location','northwest');
xlabel('Number Of Samples');
ylabel('Generation time (in seconds)');
close_system(modelname,0);
```



Conclusion

While HDL test bench is very efficient for a small number of samples, if your test scenario requires a large number of samples, HDL Verifier™ SystemVerilog DPI test bench provides faster test bench generation.

Pass-Through and No-Op Implementations

HDL Coder provides a pass-through or no-op implementation for some blocks. A pass-through implementation generates a wire in the HDL; a no-op implementation omits code generation for the block or subsystem. These implementations are useful in cases where you need a block for simulation, but do not need the block or subsystem in your generated HDL code.

The pass-through and no-op implementations are summarized in the following table.

Implementation	Description
Pass-through implementations	<p>Provides a pass-through implementation in which the block's inputs are passed directly to its outputs. HDL Coder supports the following blocks with a pass-through implementation:</p> <ul style="list-style-type: none">• Convert 1-D to 2-D• Reshape• Signal Conversion• Signal Specification
No HDL	This implementation completely removes the block from the generated code. This enables you to use the block in simulation but treat it as a "no-op" in the HDL code. This implementation is used for many blocks (such as Scopes and Assertions) that are significant in simulation but are meaningless in HDL code.

Synchronous Subsystem Behavior with the State Control Block

In this section...

- “What Is a State Control Block?” on page 27-85
- “State Control Block Modes” on page 27-85
- “Synchronous Badge for Subsystems by Using Synchronous Mode” on page 27-86
- “Generate HDL Code with the State Control Block” on page 27-87
- “Enable and Reset Hardware Simulation Behavior” on page 27-89

What Is a State Control Block?

When you have blocks with state, and have enable or reset ports inside a subsystem, use the **Synchronous** mode of the State Control block to:

- Provide efficient enable and reset simulation behavior on hardware.
- Generate cleaner HDL code and use fewer resources on hardware.

You can add the State Control block to your Simulink model at any level in the model hierarchy. How you set the State Control block affects the simulation behavior of other blocks inside the subsystem that have state.

- For synchronous hardware simulation behavior, set **State control** to **Synchronous**.
- For default Simulink simulation behavior, set **State control** to **Classic**.

State Control Block Modes

Functionality	Synchronous mode	Classic mode
State Control block setting	Default block setting when you add the block from the HDL Subsystems block library.	The simulation behavior is the same as a subsystem that does not use the State Control block.
Simulink simulation behavior	<ul style="list-style-type: none"> • Initialize method: Initializes states. • Update method: Updates states. • Output method: Computes output values. <p>The update method only updates states. The output method computes the output values at each time step.</p> <p>For example, when you have enabled subsystems, the output value changes when the enable signal is low as it processes new input values. The output value matches the output from Classic mode when enable signal becomes high.</p>	<p>The update method updates states and computes the output values.</p> <p>For example, when you have enabled subsystems, the output value is held steady when the enable signal is low and changes only when the enable signal becomes high.</p>
HDL simulation behavior	More efficient on hardware.	Less efficient on hardware.

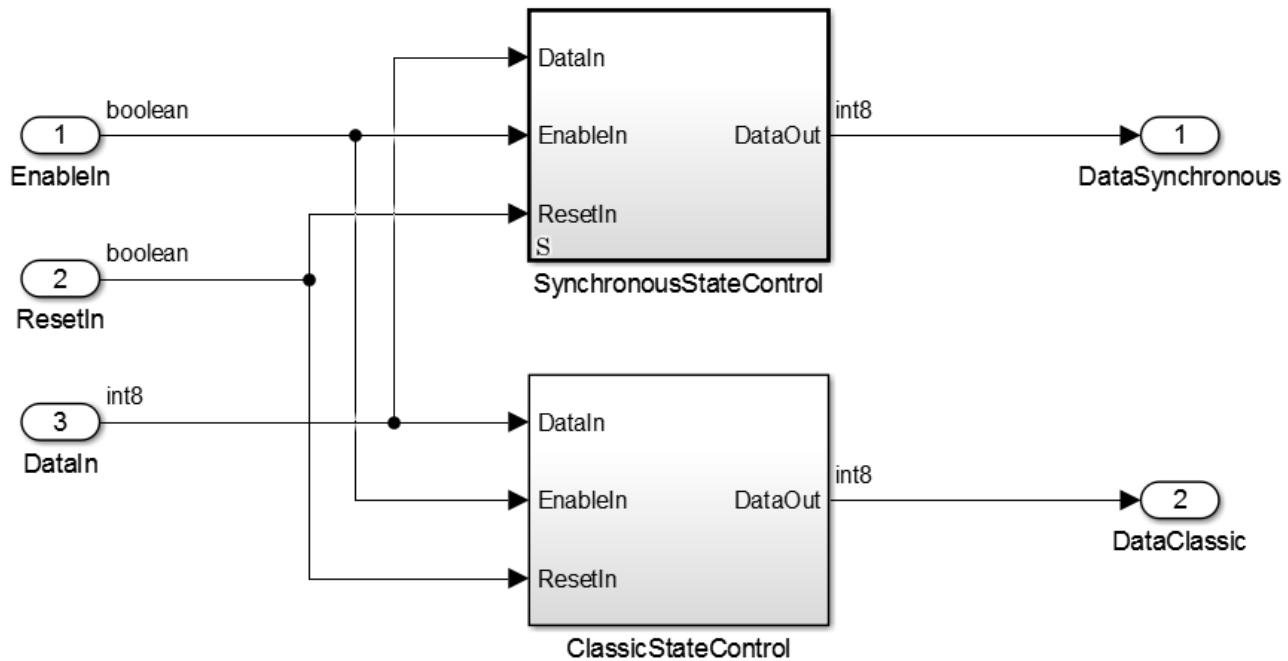
Functionality	Synchronous mode	Classic mode
HDL code generation behavior	<p>Generated HDL code is cleaner and uses fewer resources on hardware.</p> <p>For example, when you have enabled subsystems, HDL Coder does not generate bypass registers for each state update and uses fewer hardware resources.</p>	<p>Generated HDL code is not as clean and uses more hardware resources.</p> <p>For example, when you have enabled subsystems, HDL Coder generates bypass registers for each state update and uses more resources.</p>

To learn more about when you can use the State Control block, see State Control.

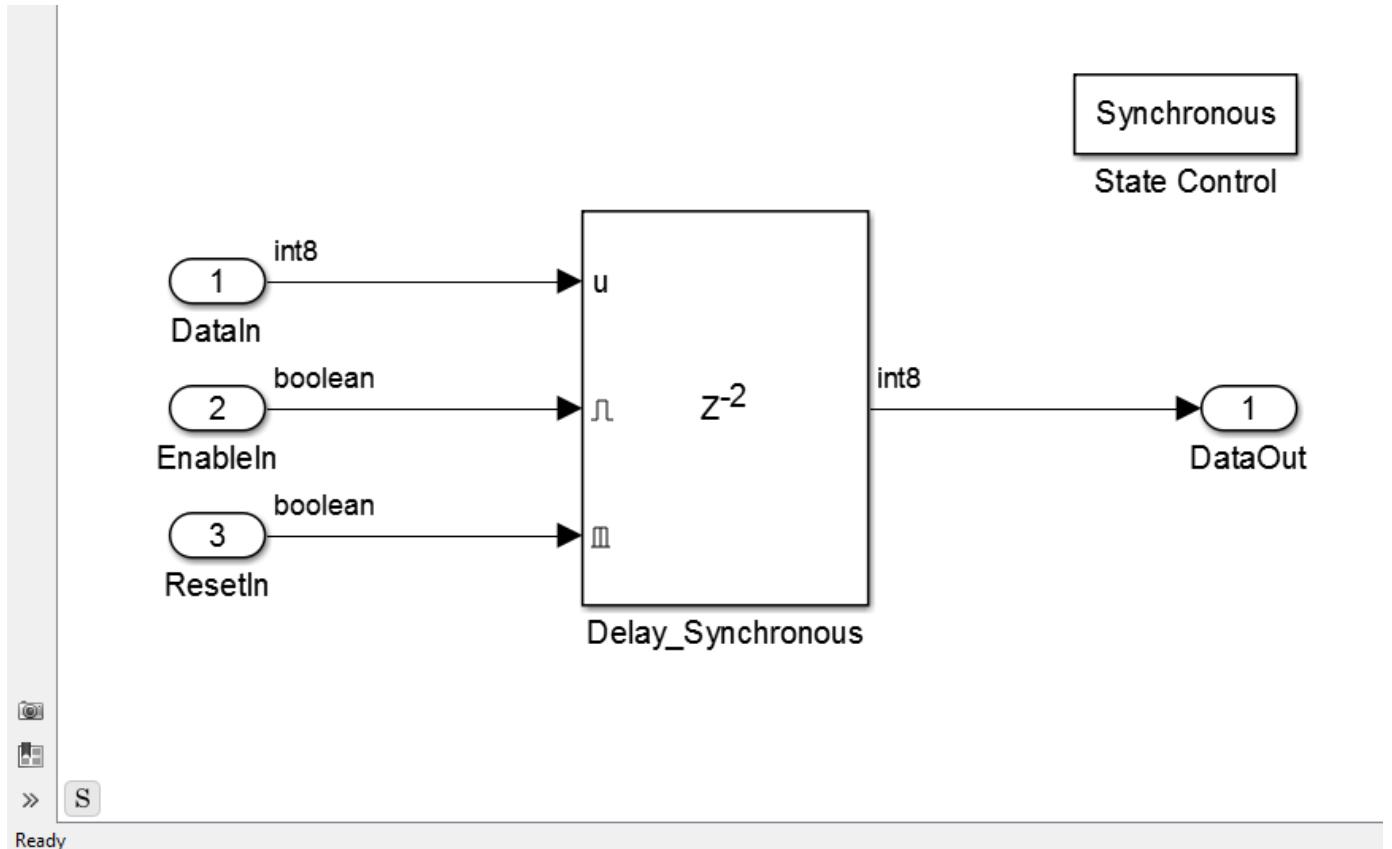
Synchronous Badge for Subsystems by Using Synchronous Mode

To see if a subsystem in your Simulink model uses synchronous semantics:

- A symbol **S** is displayed on the subsystem to indicate synchronous behavior.



- If you double-click the **SynchronousStateControl** subsystem, a badge **S** is displayed in the Simulink editor to indicate that blocks inside the subsystem are using synchronous hardware semantics.



The **SynchronousStateControl** and **ClassicStateControl** subsystems use a Delay block with an external reset and an enable port in Synchronous and Classic modes respectively.

Generate HDL Code with the State Control Block

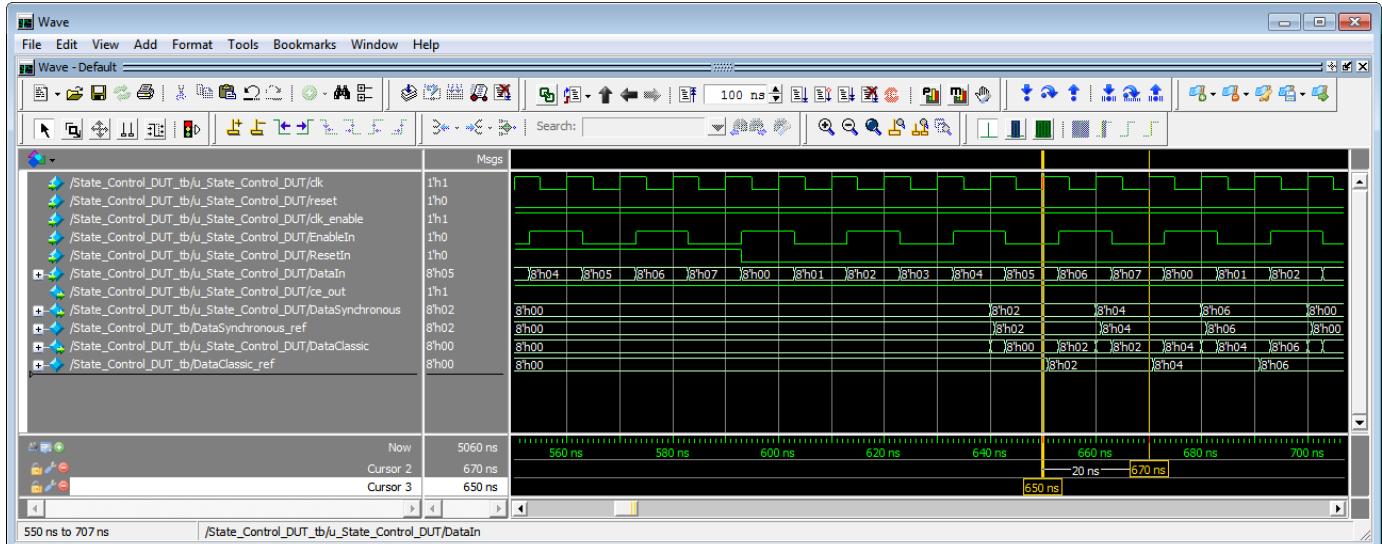
The following table shows a comparison of the HDL code generated from the Delay block for Classic and Synchronous modes of the State Control block.

Functionality	Synchronous mode	Classic mode
HDL code generation. Settings applied: • Language: Verilog • Reset type: Synchronous	<pre> `timescale 1 ns / 1 ns module SynchronousStateControl (clk, reset, enb, DataIn, EnableIn, ResetIn, DataOut); input clk; input reset; input enb; input signed [7:0] DataIn; // int8 input EnableIn; input ResetIn; output signed [7:0] DataOut; // int8 reg signed [7:0] Delay_Synchronous_reg[1:0]; // delay[2] synchronous_bypass; wire signed [7:0] Delay_Synchronous_reg_next[0:0]; // Delay_Synchronous_reg [0:0] wire signed [7:0] Delay_Synchronous_out1; // signed [7:0] Delay_Synchronous_bypass; always @(posedge clk) begin : Delay_Synchronous_process if (reset == 1'b1 ResetIn == 1'b1) begin Delay_Synchronous_reg[0] <= 8'b00000000; // delay_Synchronous_process Delay_Synchronous_reg[1] <= 8'b00000000; end else begin if (enb && EnableIn) begin Delay_Synchronous_reg[0] <= Delay_Synchronous_reg_next[0]; Delay_Synchronous_reg[1] <= Delay_Synchronous_reg_next[1]; end end end assign Delay_Synchronous_out1 = Delay_Synchronous_reg[1]; assign Delay_Synchronous_reg_next[0] = DataIn; assign Delay_Synchronous_reg_next[1] = DataOut; endmodule // SynchronousStateControl </pre>	<pre> `timescale 1 ns / 1 ns module ClassicStateControl (clk, reset, enb, DataIn, EnableIn, ResetIn, DataOut); input clk; input reset; input enb; input signed [7:0] DataIn; // int8 input EnableIn; input ResetIn; output signed [7:0] DataOut; // int8 wire signed [7:0] Delay_Synchronous_bypass; wire signed [7:0] Delay_Synchronous_reg_ne; wire signed [7:0] Delay_Synchronous_delay_ne; wire signed [7:0] Delay_Synchronous_out1; always @(posedge clk) begin : Delay_Synchronous_process if (reset == 1'b1 ResetIn == 1'b1) Delay_Synchronous_bypass <= 8'b00000000; Delay_Synchronous_reg[0] <= 8'b00000000; Delay_Synchronous_reg[1] <= 8'b00000000; if (enb && EnableIn) Delay_Synchronous_bypass <= Delay_Synchronous_reg[0]; Delay_Synchronous_reg[0] <= Delay_Synchronous_reg[1]; Delay_Synchronous_reg[1] <= Delay_Synchronous_bypass; end assign Delay_Synchronous_out1 = Delay_Synchronous_reg[1]; assign Delay_Synchronous_reg_next[0] = DataIn; assign Delay_Synchronous_reg_next[1] = DataOut; endmodule // ClassicStateControl </pre>

Functionality	Synchronous mode	Classic mode
	<ul style="list-style-type: none"> Generated HDL code is cleaner and requires fewer hardware resources as HDL Coder does not generate bypass registers. The update method only updates the states. 	<pre>assign DataOut = Delay_Synchronous_out1; endmodule // ClassicStateControl</pre> <ul style="list-style-type: none"> Generated HDL code is less cleaner and requires more hardware resources as HDL Coder generates bypass registers. The update method updates states and computes the output values.

Enable and Reset Hardware Simulation Behavior

Refer to the above Simulink model that shows a Delay block that uses **Classic** and **Synchronous** modes of the State Control block. The following diagram shows the ModelSim simulation behavior for the Delay block.



- When **ResetIn** signal is high, the **DataClassic** and **DataSynchronous** signals produce the same output.
- When both **ResetIn** and **EnableIn** signals are low, the **DataSynchronous** signal holds its value and changes only when the **EnableIn** signal becomes high at the next active clock edge. The **DataClassic** signal values change when the **EnableIn** signal is low as it processes new input values. The **DataClassic** signal values match the **DataSynchronous** signal values when the **EnableIn** becomes high.

For information about how to generate HDL code and simulate your design in ModelSim, see “Generate HDL Code from Simulink Model”.

See Also

State Control

Using the State Control block to generate more efficient code with HDL Coder™

This example shows how to use the State Control block to generate hardware-friendly HDL code using HDL Coder.

Introduction to the State Control block

The State Control block is a block that modifies the Simulink simulation behavior for its containing subsystem and all subsystems nested beneath it. Its purpose is to more closely model the synchronous behavior of clocked digital hardware, particularly with respect to blocks that have state and use explicit enable and reset signals.

When a State Control block is placed in a subsystem and has its parameter set to "Synchronous", the generated HDL code will be more hardware friendly. When a subsystem is in Synchronous mode it is marked with a graphic "S" in its lower left corner. A State Control block with its parameter set to "Classic" behaves identically to when there is no State Control block in the subsystem.

The simulation behavior difference between the two modes is small, but significant to generating efficient HDL code. The differences focus around the simulation behavior involving explicit reset and enable signals. For example, in Synchronous mode, the explicit block reset input has priority over the block enable input signal.

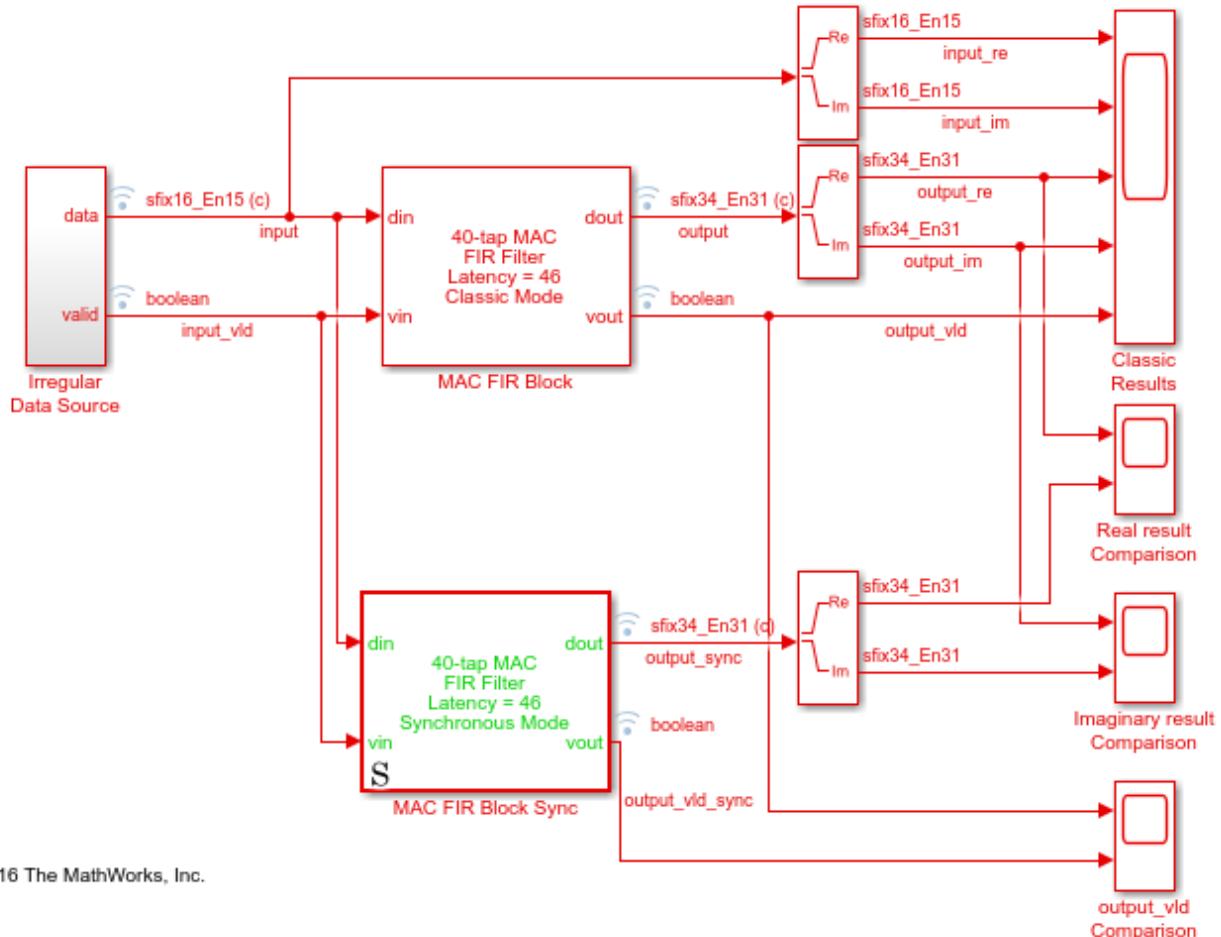
Classic mode behavior for Delay with explicit enable input port

HDL code generated by HDL Coder simulates identically to the model that it is generated from. In Classic State Control mode, the generated code for certain constructs implements sub-optimal hardware due to this requirement. For example, a Delay block with explicit enable input will generate a bypass register, comprised of a register and a multiplexer, in addition to the modeled register, to capture the Simulink Classic mode behavior.

Examine the contents of Enabled_Delay.vhd to observe the additional register signals and the bypass register.

```
load_system('hdlcoder_statecontrol_model');
open_system('hdlcoder_statecontrol_model');
set_param('hdlcoder_statecontrol_model', 'SimulationCommand', 'update');
makehdl('hdlcoder_statecontrol_model/MAC FIR Block', 'TargetDirectory', 'hdlsrc_classic');

### Generating HDL for 'hdlcoder_statecontrol_model/MAC FIR Block'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_statecontrol_model')">.
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_statecontrol_model'.
### Working on hdlcoder_statecontrol_model/MAC FIR Block/Coeff ROM as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block\Coeff ROM
### Working on hdlcoder_statecontrol_model/MAC FIR Block/Enabled_Delay as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block\Enabled_Delay
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line/circular buffer logic as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block\RAM delay line
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line/SimpleDualPortRAM_generic as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block\RAM delay line
### Working on hdlcoder_statecontrol_model/MAC FIR Block/RAM delay line as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block\RAM delay line
### Working on hdlcoder_statecontrol_model/MAC FIR Block as hdlsrc_classic\hdlcoder_statecontrol_model\MAC FIR Block
### Generating package file hdlsrc_classic\hdlcoder_statecontrol_model\MAC_FIR_Block_pkg.vhd.
### Creating HDL Code Generation Check Report file:/C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl
### HDL check for 'hdlcoder_statecontrol_model' complete with 0 errors, 0 warnings, and 0 messages
### HDL code generation complete.
```



Copyright 2016 The MathWorks, Inc.



```
type hdlsrc_classic/hdlcoder_statecontrol_model/Enabled_Delay.vhd
```

```
-- 
-- File Name: hdlsrc_classic\hdlcoder_statecontrol_model\Enabled_Delay.vhd
-- Created: 2020-11-10 14:03:19
-- 
-- Generated by MATLAB 9.9 and HDL Coder 3.17
-- 
-- 
-- 
-- Module: Enabled_Delay
-- Source Path: hdlcoder_statecontrol_model/MAC FIR Block/Enabled_Delay
-- Hierarchy Level: 1
-- 
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
```

```

ENTITY Enabled_Delay IS
  PORT( clk : IN std_logic;
        reset : IN std_logic;
        enb : IN std_logic;
        din_re : IN std_logic_vector(33 DOWNTO 0); -- sfix34_En31
        din_im : IN std_logic_vector(33 DOWNTO 0); -- sfix34_En31
        LocalEnable : IN std_logic;
        Out1_re : OUT std_logic_vector(33 DOWNTO 0); -- sfix34_En31
        Out1_im : OUT std_logic_vector(33 DOWNTO 0) -- sfix34_En31
      );
END Enabled_Delay;

ARCHITECTURE rtl OF Enabled_Delay IS

  -- Signals
  SIGNAL din_re_signed : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL din_im_signed : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL Enabled_Delay_bypass_delay_re : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL Enabled_Delay_bypass_delay_im : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL Enabled_Delay_reg_re : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL Enabled_Delay_reg_im : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL dout_re : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL dout_im : signed(33 DOWNTO 0); -- sfix34_En31

BEGIN
  din_re_signed <= signed(din_re);
  din_im_signed <= signed(din_im);

  Enabled_Delay_1_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      Enabled_Delay_bypass_delay_re <= to_signed(0, 34);
      Enabled_Delay_bypass_delay_im <= to_signed(0, 34);
      Enabled_Delay_reg_re <= to_signed(0, 34);
      Enabled_Delay_reg_im <= to_signed(0, 34);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' AND LocalEnable = '1' THEN
        Enabled_Delay_bypass_delay_im <= Enabled_Delay_reg_im;
        Enabled_Delay_reg_im <= din_im_signed;
        Enabled_Delay_bypass_delay_re <= Enabled_Delay_reg_re;
        Enabled_Delay_reg_re <= din_re_signed;
      END IF;
    END IF;
  END PROCESS Enabled_Delay_1_process;

  dout_re <= Enabled_Delay_reg_re WHEN LocalEnable = '1' ELSE
    Enabled_Delay_bypass_delay_re;

  dout_im <= Enabled_Delay_reg_im WHEN LocalEnable = '1' ELSE
    Enabled_Delay_bypass_delay_im;

  Out1_re <= std_logic_vector(dout_re);
  Out1_im <= std_logic_vector(dout_im);

```

```
END rtl;
```

Synchronous mode behavior for Delay with explicit enable input port

A Delay block with explicit enable in Synchronous State Control mode will generate HDL code that creates more efficient hardware. The implementation does not contain a bypass register.

Examine Enabled_Delay_Sync.vhd and note the improvement in the generated code as compared to the Classic mode output.

```
makehdl('hdlcoder_statecontrol_model/MAC FIR Block Sync', 'TargetDirectory', 'hdlsrc_sync');

### Generating HDL for 'hdlcoder_statecontrol_model/MAC FIR Block Sync'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_statecontrol_model')">
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_statecontrol_model'.
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/Coeff ROM as hdlsrc_sync\hdlcoder_statecontrol...
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/Enabled Delay Sync as hdlsrc_sync\hdlcoder...
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/circular buffer logic as hd...
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/SimpleDualPortRAM_gen as hd...
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line as hdlsrc_sync\hdlcoder...
### Working on hdlcoder_statecontrol_model/MAC FIR Block Sync as hdlsrc_sync\hdlcoder_statecontrol...
### Generating package file hdlsrc_sync\hdlcoder_statecontrol_model\MAC_FIR_Block_Sync_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpl...
### HDL check for 'hdlcoder_statecontrol_model' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

type hdlsrc_sync/hdlcoder_statecontrol_model/Enabled_Delay_Sync.vhd

-----
-- File Name: hdlsrc_sync\hdlcoder_statecontrol_model\Enabled_Delay_Sync.vhd
-- Created: 2020-11-10 14:03:28
-- Generated by MATLAB 9.9 and HDL Coder 3.17
-----

-----
-- Module: Enabled_Delay_Sync
-- Source Path: hdlcoder_statecontrol_model/MAC FIR Block Sync/Enabled Delay Sync
-- Hierarchy Level: 1
-- 

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY Enabled_Delay_Sync IS
PORT( clk : IN std_logic;
      reset : IN std_logic;
      enb : IN std_logic;
      din_re : IN std_logic_vector(33 DOWNTO 0); -- sfix34_Eng
      ... );
```

```

    din_im          : IN   std_logic_vector(33 DOWNTO 0); -- sfix34_En1
    LocalEnable     : IN   std_logic;
    Out1_re         : OUT  std_logic_vector(33 DOWNTO 0); -- sfix34_En1
    Out1_im         : OUT  std_logic_vector(33 DOWNTO 0)  -- sfix34_En31
  );
END Enabled_Delay_Sync;

ARCHITECTURE rtl OF Enabled_Delay_Sync IS

  -- Signals
  SIGNAL din_re_signed      : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL din_im_signed      : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL dout_re             : signed(33 DOWNTO 0); -- sfix34_En31
  SIGNAL dout_im             : signed(33 DOWNTO 0); -- sfix34_En31

BEGIN
  din_re_signed <= signed(din_re);

  din_im_signed <= signed(din_im);

  Enabled_Delay_process : PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      dout_re <= to_signed(0, 34);
      dout_im <= to_signed(0, 34);
    ELSIF clk'EVENT AND clk = '1' THEN
      IF enb = '1' AND LocalEnable = '1' THEN
        dout_re <= din_re_signed;
        dout_im <= din_im_signed;
      END IF;
    END IF;
  END PROCESS Enabled_Delay_process;

  Out1_re <= std_logic_vector(dout_re);
  Out1_im <= std_logic_vector(dout_im);

END rtl;

```

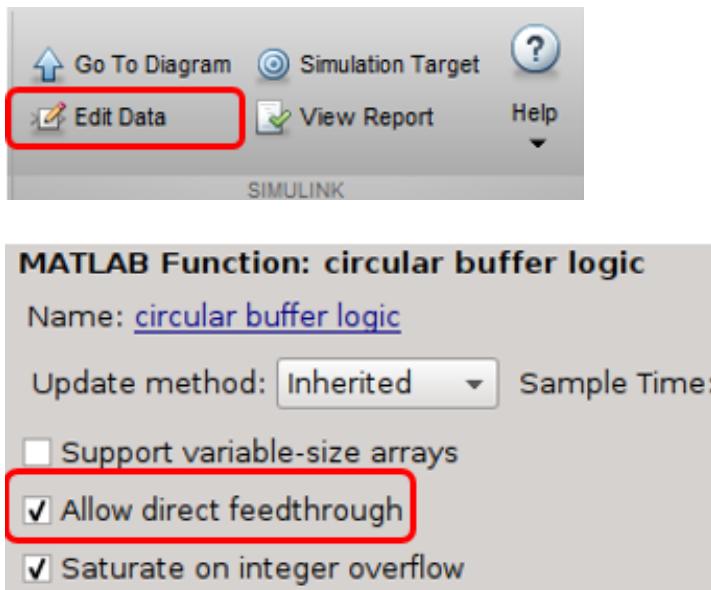
Enabled Subsystems

When a model has an Enabled subsystem in Synchronous mode, the code generated for it will also be improved. A Synchronous mode Enabled subsystem will no longer generate bypass registers on the subsystem outputs. In addition, any registers inside the Enabled subsystem that have explicit enable inputs will also show the same improvements as discussed previously.

MATLAB Function Blocks and Synchronous Mode

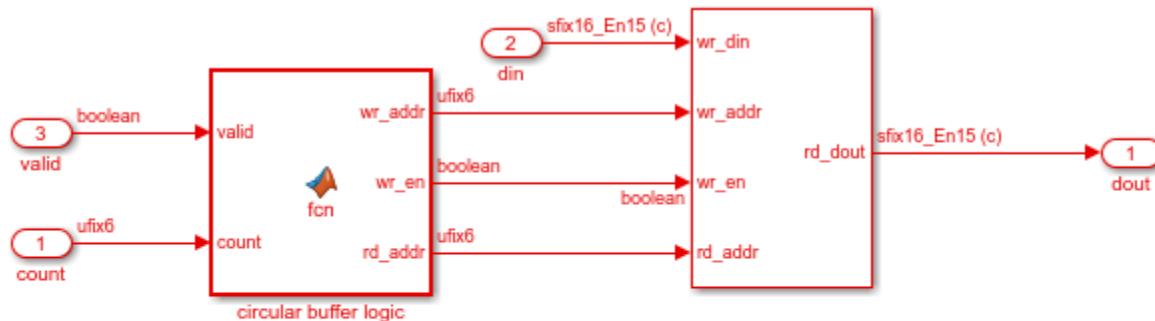
MATLAB Function Blocks require more precise configuration in order to be used in Synchronous mode. If the block contains a direct combinatorial path from block input to output, an additional setting must be enabled on the block.

The Edit data menu pick on the MATLAB toolbar opens the MATLAB Function block Ports and Data Manager. Each block containing a combinatorial output path must be marked as allowing direct feedthrough.



This setting allows code to be generated in Synchronous mode from a MATLAB Function block, when that block has both combinatorial and sequential paths in its code.

```
open_system('hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line')
open_system('hdlcoder_statecontrol_model/MAC FIR Block Sync/RAM delay line/circular buffer logic')
```



The RAM based delay line has 2 latency:
- 1 from count to rd_addr
- 1 from rd_addr to rd_dout



For additional information

To learn more about the State Control block, please refer to the block documentation.

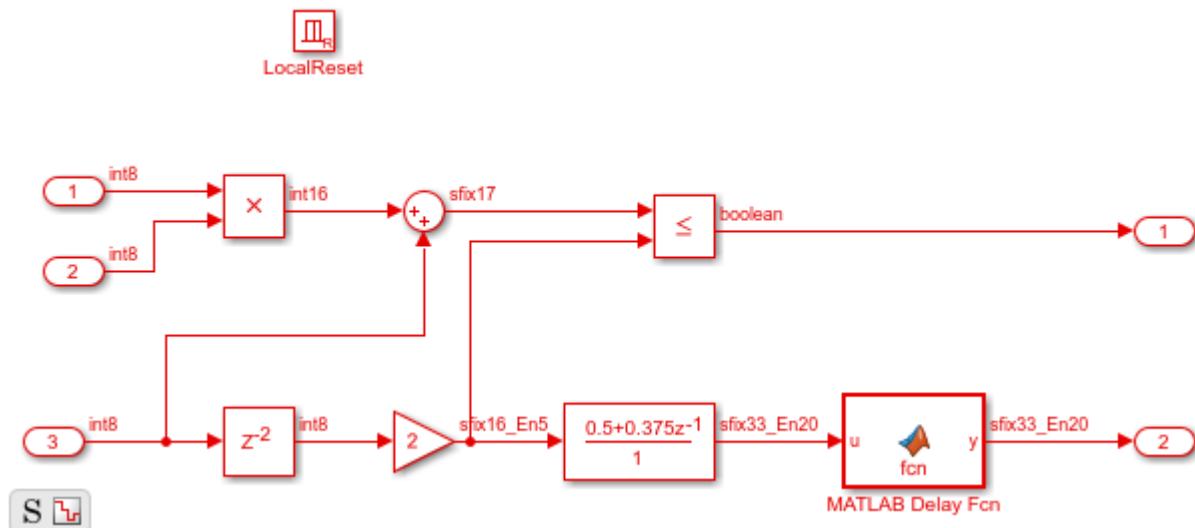
Resettable Subsystem Support in HDL Coder™

This example shows how to use Resettable Subsystems in HDL Coder.

Introduction to Resettable Subsystems

A Resettable Subsystem is a subsystem that will reset all states within the subsystem hierarchy based on a boolean control signal. It does this without requiring wiring the reset signal to each stateful block in Simulink. This feature allows resetting blocks such as the MATLAB Function Block, which does not have an available reset port. For support in HDL Coder, a Resettable Subsystem is supported only within a Synchronous StateControl region.

```
load_system('hdlcoder_resettable_subsystem');
open_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem');
set_param('hdlcoder_resettable_subsystem', 'SimulationCommand', 'update');
```



The Reset Block

A Resettable Subsystem looks similar to an Enabled Subsystem or any other Simulink conditionally executed subsystem in that it has a specialized Reset Port block inside it. This control port block has several trigger types available. HDL Coder supports the "level hold" trigger type.

```
open_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/LocalReset');
```

Resettable Subsystem Effects on Generated HDL code

Resettable Subsystems allow resetting the state of all blocks with state inside the subsystem to their initial value. In the generated HDL code, each design delay--a delay modeled explicitly in Simulink--will have a reset added. Hardware implementation delays such as pipeline delays are not reset. The reset signal is a synchronous signal and is entirely independent from the global reset signal.

```
close_system('hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/LocalReset');
makehdl('hdlcoder_resettable_subsystem/DUT');
type hdlsrc/hdlcoder_resettable_subsystem/DUT.vhd

### Generating HDL for 'hdlcoder_resettable_subsystem/DUT'.
### Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoder_rese...'
```

```
### Starting HDL check.
### Begin VHDL Code Generation for 'hdlcoder_resettable_subsystem'.
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/Discrete FIR Filter as hdlsrc
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/MATLAB Delay Fcn as hdlsrc
### Working on hdlcoder_resettable_subsystem/DUT/Resettable Subsystem as hdlsrc\hdlcoder_resettable
### Working on hdlcoder_resettable_subsystem/DUT as hdlsrc\hdlcoder_resettable_subsystem\DUT.vhd
### Generating package file hdlsrc\hdlcoder_resettable_subsystem\DUT_pkg.vhd.
### Creating HDL Code Generation Check Report file://C:\TEMP\Bdoc20b_1527579_10488\ib61345F\0\tpr
### HDL check for 'hdlcoder_resettable_subsystem' complete with 0 errors, 0 warnings, and 0 messages.
### HDL code generation complete.

-----
-- File Name: hdlsrc\hdlcoder_resettable_subsystem\DUT.vhd
-- Created: 2020-11-10 13:57:48
-- Generated by MATLAB 9.9 and HDL Coder 3.17
-- 
-- 
-- Rate and Clocking Details
-- Model base rate: 1
-- Target subsystem base rate: 1
-- 
-- Clock Enable Sample Time
-- ce_out      1
-- 
-- Output Signal          Clock Enable Sample Time
-- Out1                  ce_out      1
-- Out2                  ce_out      1
-- 
-- 
-- Module: DUT
-- Source Path: hdlcoder_resettable_subsystem/DUT
-- Hierarchy Level: 0
-- 
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY DUT IS
  PORT( clk           : IN  std_logic;
        reset         : IN  std_logic;
        clk_enable    : IN  std_logic;
        LocalReset   : IN  std_logic;
        In2          : IN  std_logic_vector(7 DOWNTO 0);  -- int8
```

```

In3           : IN    std_logic_vector(7 DOWNTO 0);  -- int8
In4           : IN    std_logic_vector(7 DOWNTO 0);  -- int8
ce_out        : OUT   std_logic;
Out1          : OUT   std_logic;
Out2          : OUT   std_logic_vector(32 DOWNTO 0)  -- sfix33_En20
);
END DUT;

ARCHITECTURE rtl OF DUT IS

-- Component Declarations
COMPONENT Resettable_Subsystem
PORT( clk      : IN    std_logic;
      reset   : IN    std_logic;
      enb     : IN    std_logic;
      In1     : IN    std_logic_vector(7 DOWNTO 0);  -- int8
      In2     : IN    std_logic_vector(7 DOWNTO 0);  -- int8
      In3     : IN    std_logic_vector(7 DOWNTO 0);  -- int8
      LocalReset : IN    std_logic;
      Out1    : OUT   std_logic;
      Out2    : OUT   std_logic_vector(32 DOWNTO 0)  -- sfix33_En20
);
END COMPONENT;

-- Component Configuration Statements
FOR ALL : Resettable_Subsystem
  USE ENTITY work.Resettable_Subsystem(rtl);

-- Signals
SIGNAL Resettable_Subsystem_out1      : std_logic;
SIGNAL Resettable_Subsystem_out2      : std_logic_vector(32 DOWNTO 0);  -- ufix33

BEGIN
  u_Resettable_Subsystem : Resettable_Subsystem
  PORT MAP( clk => clk,
            reset => reset,
            enb => clk_enable,
            In1 => In2,  -- int8
            In2 => In3,  -- int8
            In3 => In4,  -- int8
            LocalReset => LocalReset,
            Out1 => Resettable_Subsystem_out1,
            Out2 => Resettable_Subsystem_out2  -- sfix33_En20
  );
  ce_out <= clk_enable;
  Out1 <= Resettable_Subsystem_out1;
  Out2 <= Resettable_Subsystem_out2;
END rtl;

```

The MATLAB Function Block does not have support for an explicit reset port. When placed in a Resettable Subsystem, HDL Coder will generate a synchronous external reset signal to control the resetting of persistent variables inside the function.

```
function y = fcn(u)
persistent state;
if isempty(state)
    state = fi(0, 1, 33, 20);
end
y = state;
state = u;
end

type hdlsrc/hdlcoder_resettable_subsystem/MATLAB_Delay_Fcn.vhd

-----
-- File Name: hdlsrc\hdlcoder_resettable_subsystem\MATLAB_Delay_Fcn.vhd
-- Created: 2020-11-10 13:57:48
-- Generated by MATLAB 9.9 and HDL Coder 3.17
-- 

-----
-- Module: MATLAB_Delay_Fcn
-- Source Path: hdlcoder_resettable_subsystem/DUT/Resettable Subsystem/MATLAB Delay Fcn
-- Hierarchy Level: 2
-- 

-----
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY MATLAB_Delay_Fcn IS
    PORT( clk           : IN  std_logic;
          reset        : IN  std_logic;
          enb          : IN  std_logic;
          u            : IN  std_logic_vector(32 DOWNTO 0);  -- sfix33_En20
          LocalReset   : IN  std_logic;
          y            : OUT std_logic_vector(32 DOWNTO 0)  -- sfix33_En20
        );
END MATLAB_Delay_Fcn;

ARCHITECTURE rtl OF MATLAB_Delay_Fcn IS
    -- Signals
    SIGNAL u_signed      : signed(32 DOWNTO 0);  -- sfix33_En20
    SIGNAL y_tmp          : signed(32 DOWNTO 0);  -- sfix33_En20

BEGIN
    u_signed <= signed(u);
```

```
MATLAB_Delay_Fcn_1_process : PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        y_tmp <= to_signed(0, 33);
    ELSIF clk'EVENT AND clk = '1' THEN
        IF enb = '1' THEN
            IF LocalReset = '1' THEN
                y_tmp <= to_signed(0, 33);
            ELSE
                y_tmp <= u_signed;
            END IF;
        END IF;
    END IF;
END PROCESS MATLAB_Delay_Fcn_1_process;

y <= std_logic_vector(y_tmp);

END rtl;
```

A synchronous delay signal named "LocalDelay" has been added to the VHDL code generated for the delay implemented in the MATLAB Function block.

Stateflow HDL Code Generation Support

- “Introduction to Stateflow HDL Code Generation” on page 28-2
- “Hardware Realization of Stateflow Semantics” on page 28-6
- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-7
- “Design Patterns Using Advanced Chart Features” on page 28-14
- “Initialize Persistent Variables in MATLAB Functions” on page 28-22

Introduction to Stateflow HDL Code Generation

In this section...

- “Overview” on page 28-2
- “Comments” on page 28-2
- “Tunable Parameters” on page 28-2
- “Example” on page 28-2
- “Restrictions” on page 28-3

Overview

Stateflow charts provide concise descriptions of complex system behavior using hierarchical finite state machine (FSM) theory, flow diagram notation, and state-transition diagrams.

You use a chart to model a finite state machine or a complex control algorithm intended for realization as an ASIC or FPGA. When the model meets design requirements, you then generate HDL code (VHDL or Verilog) that implements the design embodied in the model. You can simulate and synthesize generated HDL code using industry standard tools, and then map your system designs into FPGAs and ASICs.

In general, generation of VHDL or Verilog code from a model containing a chart does not differ greatly from HDL code generation from other models. The HDL code generator is designed to

- Support the largest possible subset of chart semantics that is consistent with HDL. This broad subset lets you generate HDL code from existing models without significant remodeling effort.
- Generate bit-true, cycle-accurate HDL code that is fully compatible with Stateflow simulation semantics.

Comments

When your Simulink model contains a Stateflow Chart that uses comments, HDL Coder generates the comments in the HDL code.

When you generate Verilog code from the model, HDL Coder displays the comments in the Stateflow Chart inline beside the corresponding Stateflow object.

Tunable Parameters

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

For more information, see “Generate DUT Ports for Tunable Parameters” on page 10-17.

Example

The `hdlcoderfir` model shows how to generate HDL code for a subsystem that includes Stateflow charts.

To open the model, at the command line, enter:

```
hdlcoderfir
```

Restrictions

HDL Coder does not support Stateflow blocks that contain messages for HDL code generation.

Location of Charts in the Model

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

Data Types

The code generator supports a subset of MATLAB data types in charts that include:

- Signed and unsigned integer
- Fixed point
- Boolean
- Enumeration

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

If you use single and double data types, HDL Coder generates real data types in the HDL code. You can simulate and verify the code by using third-party simulators such as ModelSim.

However, real types are not synthesizable on the target FPGA device. The code generator does not support generation of HDL code for the Stateflow Chart in **Native Floating Point** mode. To generate synthesizable HDL code when you use floating-point data types, develop the algorithm by using MATLAB Function on page 10-106 blocks or other “Simulink Blocks Supported with Native Floating-Point” on page 10-120.

Chart Initialization

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

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

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

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

Note Active state data is not supported in charts that use classic or Mealy semantics when the chart property **Initialize outputs every time chart wakes up** is enabled. For more information, see “Monitor State Activity Through Active State Data” (Stateflow).

Imported Code

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

- Do not call MATLAB functions other than `min` or `max`.
- Do not use MATLAB System Objects in a Chart block.
- Do not use MATLAB workspace data.
- Do not call C math functions. HDL does not have a counterpart to the C math library.
- If the **Enable C-bit operations** property is disabled, do not use the exponentiation operator (`^`). The exponentiation operator is implemented with the C Math Library function `pow`.
- Do not include custom code. Information entered on the **Simulation Target > Custom Code** pane in the Configuration Parameters dialog box is ignored.
- Do not share data (via Data Store Memory blocks) between charts. HDL Coder does not map such global data to HDL because HDL does not support global data.

Vector of Tunable Parameters

Vector of Tunable Parameters as data types for Chart blocks are not supported.

Input and Output Events

HDL Coder supports the use of input and output events with Stateflow charts, subject to the following constraints:

- You can define and use only one input event per Stateflow chart. (There is no restriction on the number of output events that you can use.)
- The coder does not support HDL code generation for charts that have a single input event, and which also have nonzero initial values on the chart's output ports.
- All input and output events must be edge-triggered.

For detailed information on input and output events, see “Activate a Stateflow Chart by Sending Input Events” (Stateflow) and “Activate a Simulink Block by Sending Output Events” (Stateflow).

Messages

Stateflow messages are not supported for HDL code generation.

Loops

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

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

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

Other Restrictions

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

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

Do not use the following implicit events:

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

You can use the following implicit events:

- `wakeup`
- `tick`

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

- Do not use recursion through graphical functions. HDL Coder does not currently support recursion.
- Avoid unstructured code. Although charts allow unstructured code (through transition flow diagrams and graphical functions), this usage results in `goto` statements and multiple function return statements. HDL does not support either `goto` statements or multiple function return statements. Therefore, do not use unstructured flow diagrams.
- If you have not selected the **Initialize Outputs Every Time Chart Wakes Up** chart option, do not read from output ports.
- Do not use Data Store Memory objects.
- Do not use pointer (`&`) or indirection (`*`) operators. See “Pointer and Address Operations” (Stateflow).
- If a chart gets a run-time overflow error during simulation, it is possible to disable data range error checking and generate HDL code for the chart. However, in such cases, some results obtained from the generated HDL code might not be bit-true to results from the simulation. The recommended practice is to enable overflow checking and eliminate overflow conditions from the model during simulation.

See Also

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-7
- “Design Patterns Using Advanced Chart Features” on page 28-14

More About

- “Hardware Realization of Stateflow Semantics” on page 28-6

Hardware Realization of Stateflow Semantics

A mapping from Stateflow semantics to an HDL implementation has the following requirements:

- **Requirement 1:** Hardware designs require separability of output and state update functions.
- **Requirement 2:** HDL is a concurrent language. To achieve the goal of bit-true simulation, execution must be in order.

To meet Requirement 1, an FSM is coded in HDL as two concurrent blocks that execute under different conditions. One block evaluates the transition conditions, computes outputs and computes the next state variables. The other block updates the current state variables from the available next state and performs the actual state transitions. This second block is activated only on the trigger edge of the clock signal, or an asynchronous reset signal.

Stateflow sequential semantics map to HDL sequential statements, and local chart variables in function scope map to VHDL variables in process scope. In VHDL, variable assignment is sequential. Therefore, statements in a Stateflow function that uses local variables can map to statements in a VHDL process that uses corresponding variables. The VHDL assignments execute in the same order as the assignments in the Stateflow function.

See Also

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-7
- “Design Patterns Using Advanced Chart Features” on page 28-14

More About

- “Introduction to Stateflow HDL Code Generation” on page 28-2

Generate HDL for Mealy and Moore Finite State Machines

In this section...

["Overview" on page 28-7](#)

["Generating HDL Code for a Moore Finite State Machine" on page 28-7](#)

["Generating HDL for a Mealy Finite State Machine" on page 28-10](#)

Overview

Stateflow charts support modeling of three types of state machines:

- Classic (default)
- Mealy
- Moore

Mealy and Moore state machines differ in the following ways:

- The outputs of a Mealy state machine are a function of the current state and inputs.
- The outputs of a Moore state machine are a function of the current state only.

The principal advantages of using Mealy or Moore charts as an alternative to Classic charts are:

- Moore charts generate more efficient code than Classic charts.
- At compile time, Mealy and Moore charts are validated for conformance to their formal definitions and semantic rules, and violations are reported.

To learn more about HDL code generation guidelines for charts, see [Chart](#).

Open the `hdlcoder_fsm_mealy_moore` model for an example that shows how to model Mealy and Moore charts.

Generating HDL Code for a Moore Finite State Machine

When generating HDL code for a chart that models a Moore state machine:

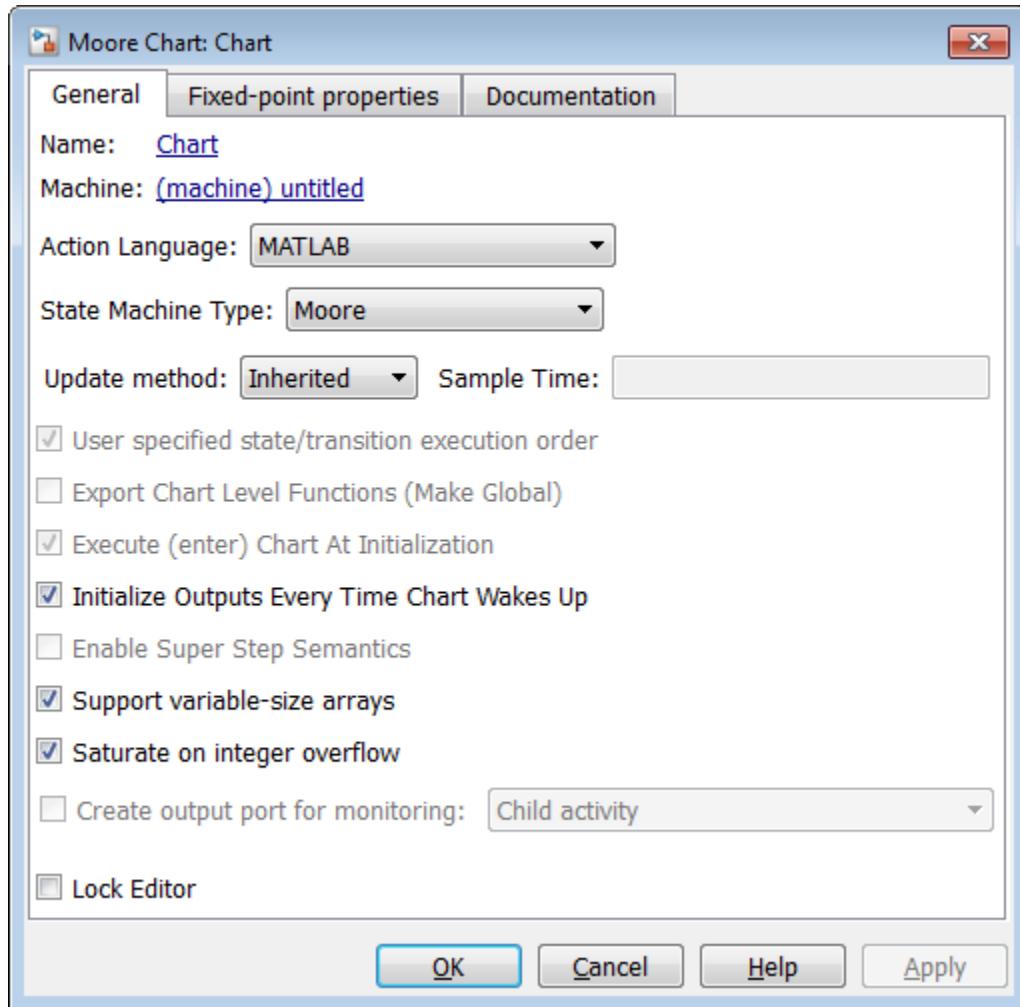
- The chart must meet the general code generation requirements as described in [Chart](#).
- Actions must occur in states only. These actions must be unlabeled.

Moore actions must be associated with states, because output computation must be dependent only on states, not input. The configuration of active states at time step t determines output. If state S is active when a chart wakes up at time t , it contributes to the output whether or not it remains active into time $t+1$.

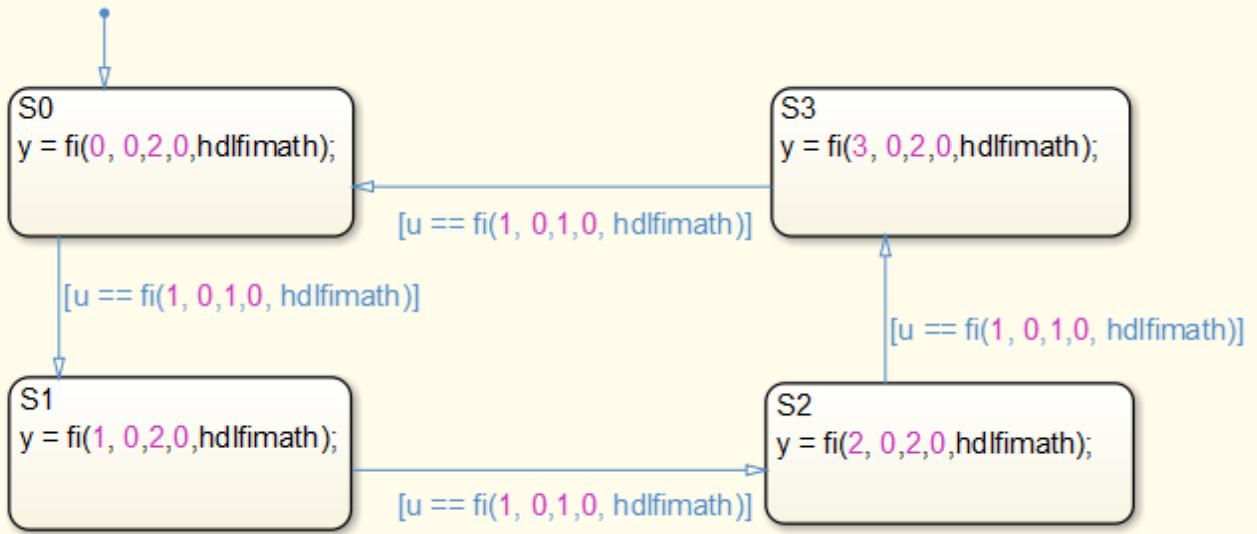
- Do not call Simulink functions.

This prevents output from depending on input in ways that would be difficult for the HDL code generator to verify.

- Make sure that you enable the chart property **Initialize Outputs Every Time Chart Wakes Up** as shown in the figure.



The following figure shows a Stateflow chart of a Moore state machine that uses MATLAB as the action language.



The Verilog code generated for the Moore chart:

```

always @(posedge clk or posedge reset)
begin : Moore_Chart_1_process
    if (reset == 1'b1) begin
        is_Moore_Chart <= is_Moore_Chart_IN_S0;
    end
    else begin
        if (enb) begin
            case ( is_Moore_Chart )
                is_Moore_Chart_IN_S0 :
                    begin
                        if (u == 8'sb00000001) begin
                            is_Moore_Chart <= is_Moore_Chart_IN_S1;
                        end
                    end
                is_Moore_Chart_IN_S1 :
                    begin
                        if (u == 8'sb00000001) begin
                            is_Moore_Chart <= is_Moore_Chart_IN_S2;
                        end
                    end
                is_Moore_Chart_IN_S2 :
                    begin
                        if (u == 8'sb00000001) begin
                            is_Moore_Chart <= is_Moore_Chart_IN_S3;
                        end
                    end
                default :
                    begin
                        if (u == 8'sb00000001) begin
                            is_Moore_Chart <= is_Moore_Chart_IN_S0;
                        end
                    end
            endcase
        end
    end
end

always @ (is_Moore_Chart) begin
    y_1 = 2'b00;
    case ( is_Moore_Chart )
        is_Moore_Chart_IN_S0 :
            begin
                y_1 = 2'b00;
            end
        is_Moore_Chart_IN_S1 :
            begin

```

```
    y_1 = 2'b01;
  end
is_Moore_Chart_IN_S2 :
begin
  y_1 = 2'b10;
end
default :
begin
  y_1 = 2'b11;
end
endcase
end

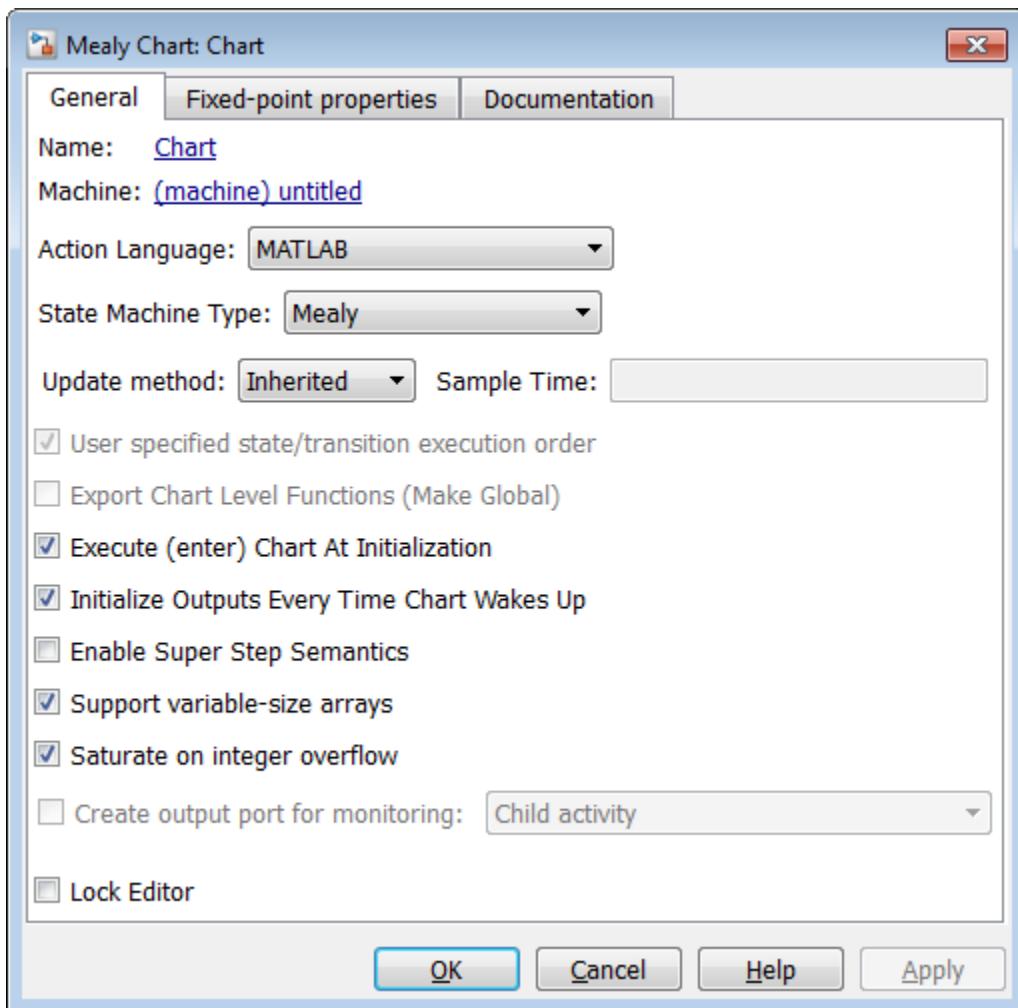
assign y = y_1;
```

For an example that shows Mealy and Moore state machines that are appropriate for HDL code generation, open the `hdlcoder_fsm_mealy_moore` model.

Generating HDL for a Mealy Finite State Machine

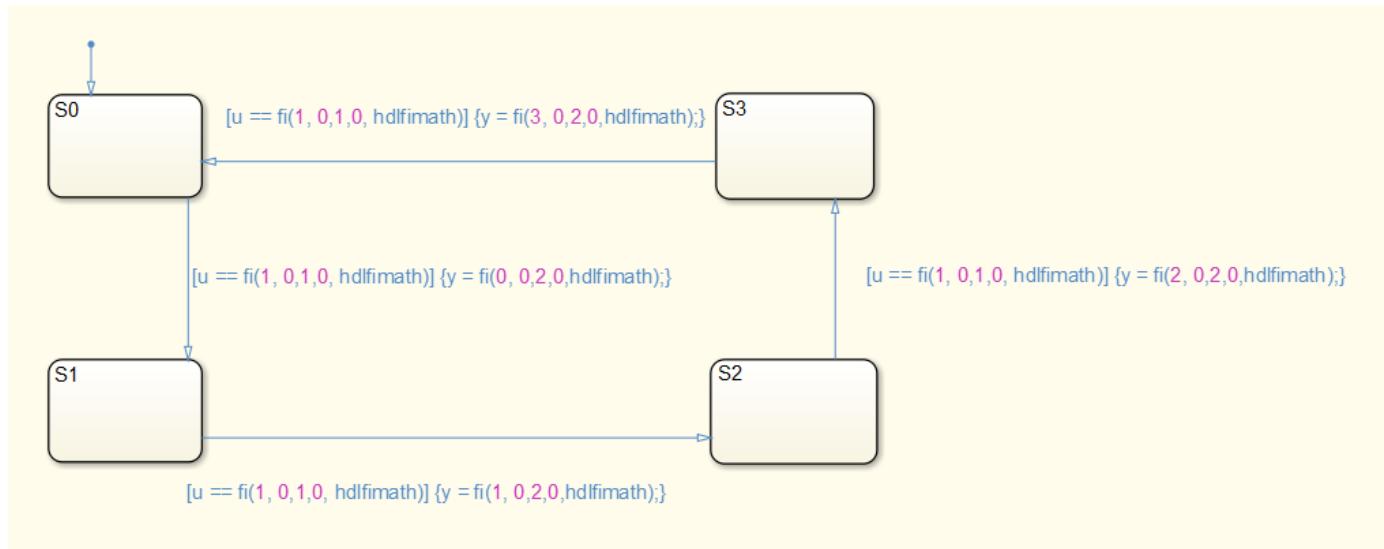
When generating HDL code for a chart that models a Mealy state machine:

- The chart must meet the general code generation requirements as described in Chart.
- Actions must be associated with inner and outer transitions only.
- For better synthesis results and more readable HDL code, we recommend enabling the chart property **Initialize Outputs Every Time Chart Wakes Up**, as shown in the following figure. If you disable **Initialize Outputs Every Time Chart Wakes Up**, the chart output is persistent, so the generated HDL code must internally register the output values.



Mealy actions are associated with transitions. In Mealy machines, output computation is expected to be driven by the change on inputs. In fact, the dependence of output on input is the fundamental distinguishing factor between the formal definitions of Mealy and Moore machines. The requirement that actions be given on transitions is to some degree stylistic, rather than required, to enforce Mealy semantics. However, it is natural that output computation follows input conditions on input, because transition conditions are primarily input conditions in any machine type.

The following figure shows an example of a chart that models a Mealy state machine using MATLAB as the action language.



The Verilog code generated for the Mealy chart:

```

always @(posedge clk or posedge reset)
begin : Mealy_Chart_1_process
  if (reset == 1'b1) begin
    is_Mealy_Chart <= is_Mealy_Chart_IN_S0;
  end
  else begin
    if (enb) begin
      is_Mealy_Chart <= is_Mealy_Chart_next;
    end
  end
end

always @ (is_Mealy_Chart, u) begin
  is_Mealy_Chart_next = is_Mealy_Chart;
  y_1 = 2'b00;
  case ( is_Mealy_Chart)
    is_Mealy_Chart_IN_S0 :
    begin
      if (u == 8'sb00000001) begin
        y_1 = 2'b00;
        is_Mealy_Chart_next = is_Mealy_Chart_IN_S1;
      end
    end
    is_Mealy_Chart_IN_S1 :
    begin
      if (u == 8'sb00000001) begin
        y_1 = 2'b01;
        is_Mealy_Chart_next = is_Mealy_Chart_IN_S2;
      end
    end
    is_Mealy_Chart_IN_S2 :
    begin
      if (u == 8'sb00000001) begin
        y_1 = 2'b10;
        is_Mealy_Chart_next = is_Mealy_Chart_IN_S3;
      end
    end
    default :
    begin
      if (u == 8'sb00000001) begin
        y_1 = 2'b11;
        is_Mealy_Chart_next = is_Mealy_Chart_IN_S0;
      end
    end
  endcase
end

assign y = y_1;

```

For an example that shows Mealy and Moore state machines that are appropriate for HDL code generation, open the `hdlcoder_fsm_mealy_moore` model.

See Also

Chart

More About

- “Design Patterns Using Advanced Chart Features” on page 28-14
- “Introduction to Stateflow HDL Code Generation” on page 28-2
- “Hardware Realization of Stateflow Semantics” on page 28-6

Design Patterns Using Advanced Chart Features

In this section...

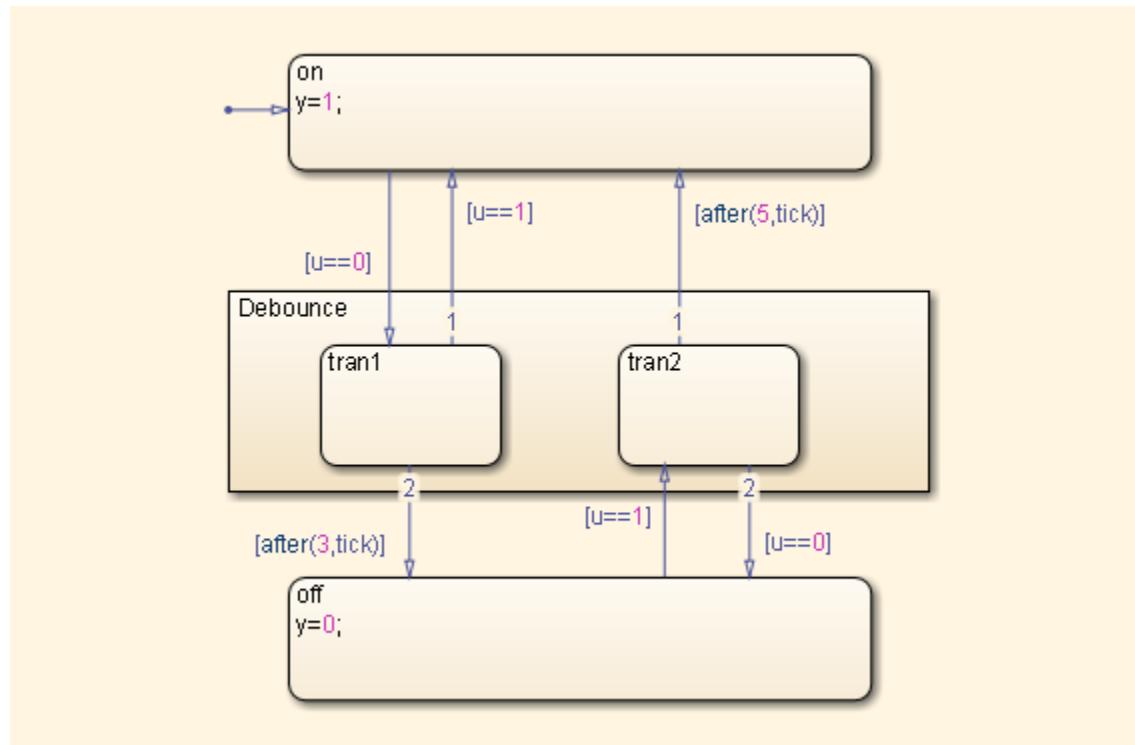
- “Temporal Logic” on page 28-14
- “Graphical Function” on page 28-15
- “Hierarchy and Parallelism” on page 28-16
- “Stateless Charts” on page 28-17
- “Truth Tables” on page 28-18

Temporal Logic

Stateflow temporal logic operators (such as `after`, `before`, or `every`) are Boolean operators that operate on recurrence counts of Stateflow events. Temporal logic operators can appear only in conditions on transitions that originate from states, and in state actions. Although temporal logic does not introduce new events into a Stateflow model, it is useful to think of the change of value of a temporal logic condition as an event. You can use temporal logic operators in many cases where a counter is required. A common use case would be to use temporal logic to implement a time-out counter.

For detailed information, see “Control Chart Execution by Using Temporal Logic” (Stateflow).

The chart shown in the following figure uses temporal logic in a design for a debouncer. Instead of instantaneously switching between on and off states, the chart uses two intermediate states and temporal logic to ignore transients. The transition is committed based on a time-out.



By default, states in a Stateflow Chart are ordered alphabetically. The ordering of states in the HDL code can potentially vary if you enable active state output port generation in the HDL code. To enable this setting, open the Chart properties and select the **Create output port for monitoring** check box. See also “Simplify Stateflow Charts by Incorporating Active State Output” (Stateflow).

When you generate VHDL code, the recently added state is selected as the OTHERS state in the HDL code. The following code excerpt shows VHDL code generated from this Chart.

```
Chart_1_output : PROCESS (is_Chart, u, temporalCounter_il1, y_reg)
  VARIABLE temporalCounter_il1_temp : unsigned(7 DOWNT0 0);
BEGIN
  temporalCounter_il1_temp := temporalCounter_il1;
  is_Chart_next <= is_Chart;
  y_reg_next <= y_reg;
  IF temporalCounter_il1 < 7 THEN
    temporalCounter_il1_temp := temporalCounter_il1 + 1;
  END IF;

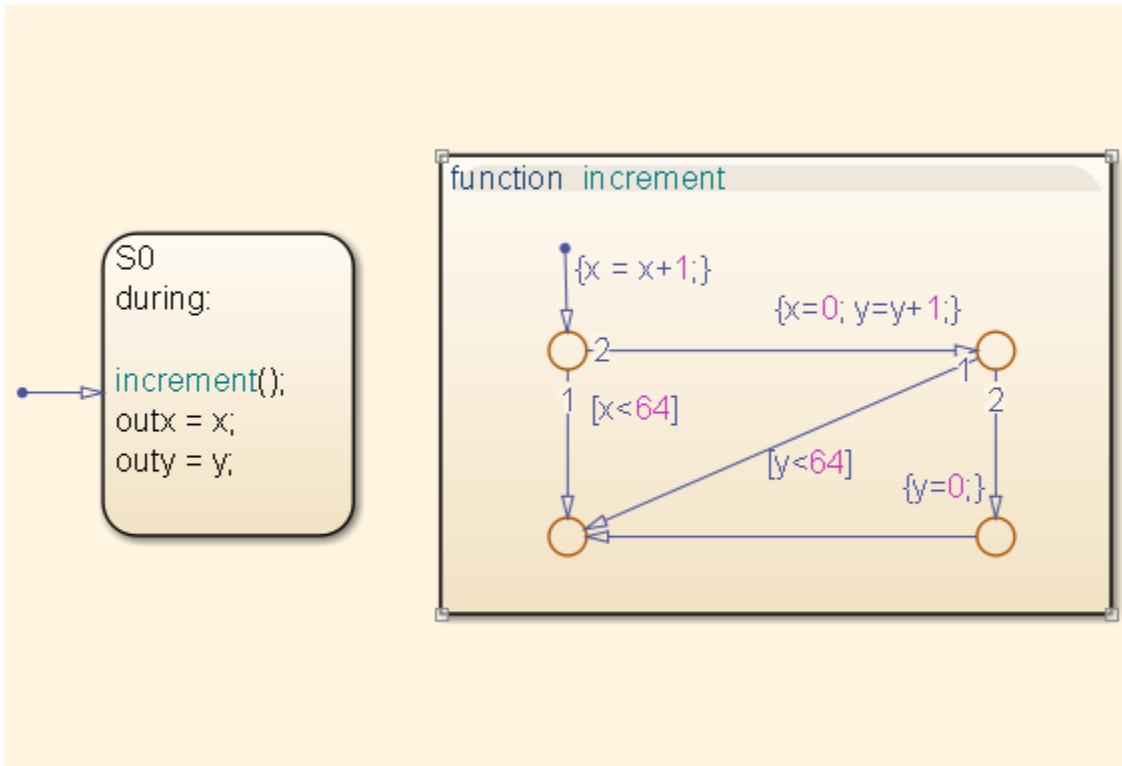
  CASE is_Chart IS
    WHEN IN_tran1 =>
      IF u = 1.0 THEN
        is_Chart_next <= IN_on;
        y_reg_next <= 1.0;
      ELSIF temporalCounter_il1_temp >= 3 THEN
        is_Chart_next <= IN_off;
        y_reg_next <= 0.0;
      END IF;
    WHEN IN_tran2 =>
      IF temporalCounter_il1_temp >= 5 THEN
        is_Chart_next <= IN_on;
        y_reg_next <= 1.0;
      ELSIF u = 0.0 THEN
        is_Chart_next <= IN_off;
        y_reg_next <= 0.0;
      END IF;
    WHEN IN_off =>
      IF u = 1.0 THEN
        is_Chart_next <= IN_tran2;
        temporalCounter_il1_temp := to_unsigned(0, 8);
      END IF;
    WHEN OTHERS =>
      IF u = 0.0 THEN
        is_Chart_next <= IN_tran1;
        temporalCounter_il1_temp := to_unsigned(0, 8);
      END IF;
  END CASE;

  temporalCounter_il1_next <= temporalCounter_il1_temp;
END PROCESS Chart_1_output;
```

Graphical Function

A graphical function is a function defined graphically by a flow diagram. Graphical functions reside in a chart along with the diagrams that invoke them. Like MATLAB functions and C functions, graphical functions can accept arguments and return results. Graphical functions can be invoked in transition and state actions.

The following figure shows a graphical function that implements a 64-by-64 counter.



The following code excerpt shows VHDL code generated for this graphical function.

```

x64_counter_sf : PROCESS (x, y, outx_reg, outy_reg)
    -- local variables
    VARIABLE x_temp : unsigned(7 DOWNTO 0);
    VARIABLE y_temp : unsigned(7 DOWNTO 0);
BEGIN
    outx_reg_next <= outx_reg;
    outy_reg_next <= outy_reg;
    x_temp := x;
    y_temp := y;
    x_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(x_temp, 9), 10)
        + tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

    IF x_temp < to_unsigned(64, 8) THEN
        NULL;
    ELSE
        x_temp := to_unsigned(0, 8);
        y_temp := tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(y_temp, 9), 10)
            + tmw_to_unsigned(to_unsigned(1, 9), 10), 8);

        IF y_temp < to_unsigned(64, 8) THEN
            NULL;
        ELSE
            y_temp := to_unsigned(0, 8);
        END IF;
    END IF;

    outx_reg_next <= x_temp;
    outy_reg_next <= y_temp;
    x_next <= x_temp;
    y_next <= y_temp;
END PROCESS x64_counter_sf;

```

Hierarchy and Parallelism

Stateflow charts support both hierarchy (states containing other states) and parallelism (multiple states that can be active simultaneously).

In Stateflow semantics, parallelism is not synonymous with concurrency. Parallel states can be active simultaneously, but they are executed sequentially according to their execution order. (Execution order is displayed on the upper right corner of a parallel state).

For detailed information on hierarchy and parallelism, see “Hierarchy of Stateflow Objects” (Stateflow) and “Execution Order for Parallel States” (Stateflow).

For HDL code generation, an entire chart maps to a single output computation process. Within the output computation process:

- The execution of parallel states proceeds sequentially.
- Nested hierarchical states map to nested CASE statements in the generated HDL code.

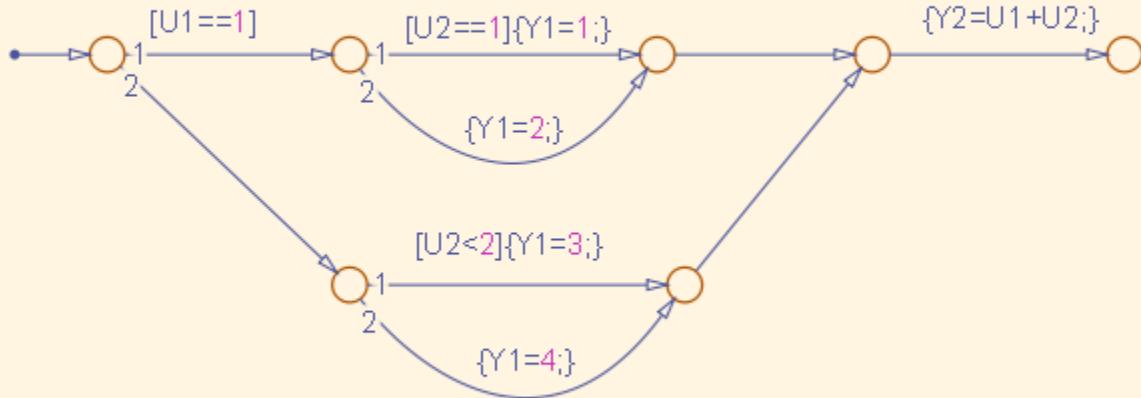
Stateless Charts

Charts consisting of pure flow diagrams (i.e., charts without states) are useful in capturing **if-then-else** constructs used in procedural languages like C.

As an example, consider the following logic, expressed in C-like pseudocode.

```
if(U1==1) {
    if(U2==1) {
        Y = 1;
    }else{
        Y = 2;
    }
}else{
    if(U2<2) {
        Y = 3;
    }else{
        Y = 4;
    }
}
```

The following figure shows the flow diagram that implements the **if-then-else** logic.



The following generated VHDL code excerpt shows the nested IF-ELSE statements obtained from the flow diagram.

```

Chart : PROCESS (Y1_reg, Y2_reg, U1, U2)
  -- local variables
  BEGIN
    Y1_reg_next <= Y1_reg;
    Y2_reg_next <= Y2_reg;

    IF unsigned(U1) = to_unsigned(1, 8) THEN
      IF unsigned(U2) = to_unsigned(1, 8) THEN
        Y1_reg_next <= to_unsigned(1, 8);
      ELSE
        Y1_reg_next <= to_unsigned(2, 8);
      END IF;
    ELSIF unsigned(U2) < to_unsigned(2, 8) THEN
      Y1_reg_next <= to_unsigned(3, 8);
    ELSE
      Y1_reg_next <= to_unsigned(4, 8);
    END IF;

    Y2_reg_next <= tmw_to_unsigned(tmw_to_unsigned(tmw_to_unsigned(unsigned(U1), 9), 10)
    + tmw_to_unsigned(tmw_to_unsigned(unsigned(U2), 9), 10), 8);
  END PROCESS Chart;
  
```

Truth Tables

HDL Coder supports HDL code generation for:

- Truth Table functions within a Stateflow chart
- Truth Table blocks in Simulink models

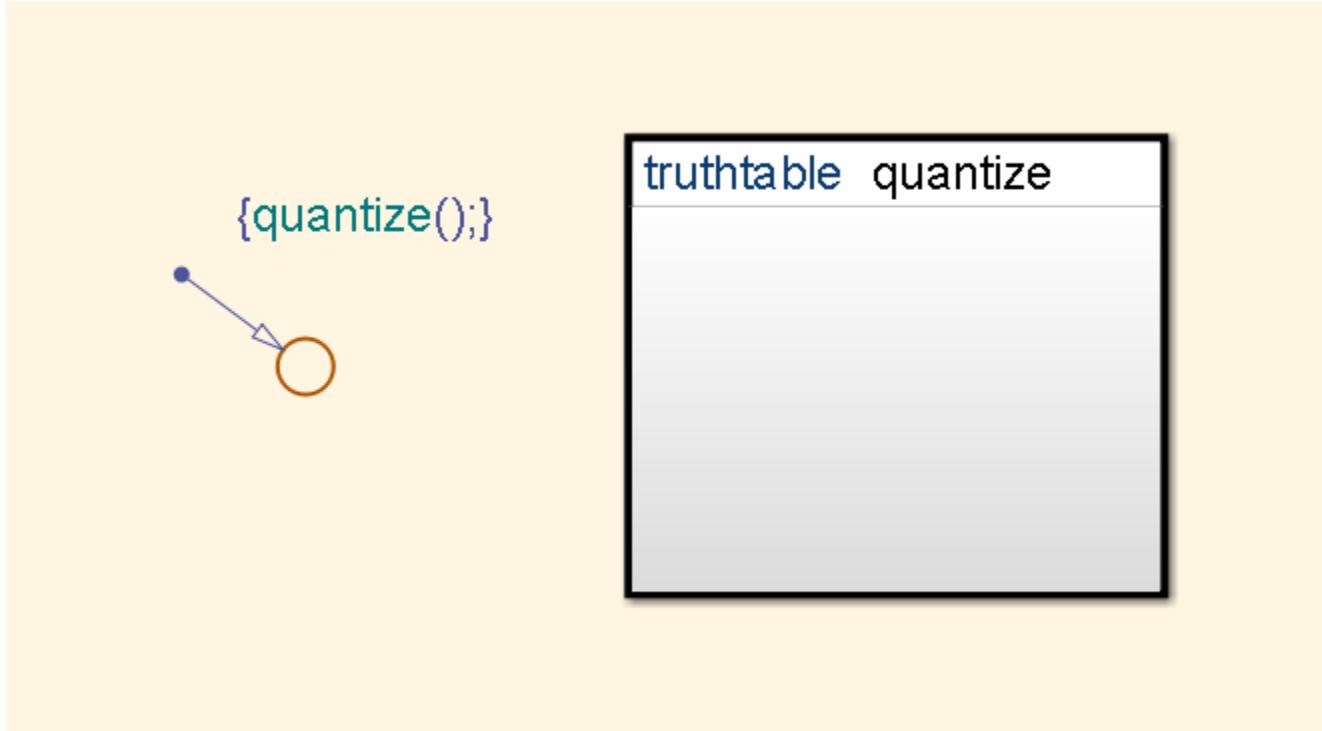
This section examines a Truth Table function in a chart, and the VHDL code generated for the chart.

Truth Tables are well-suited for implementing compact combinatorial logic. A typical application for Truth Tables is to implement nonlinear quantization or threshold logic. Consider the following logic:

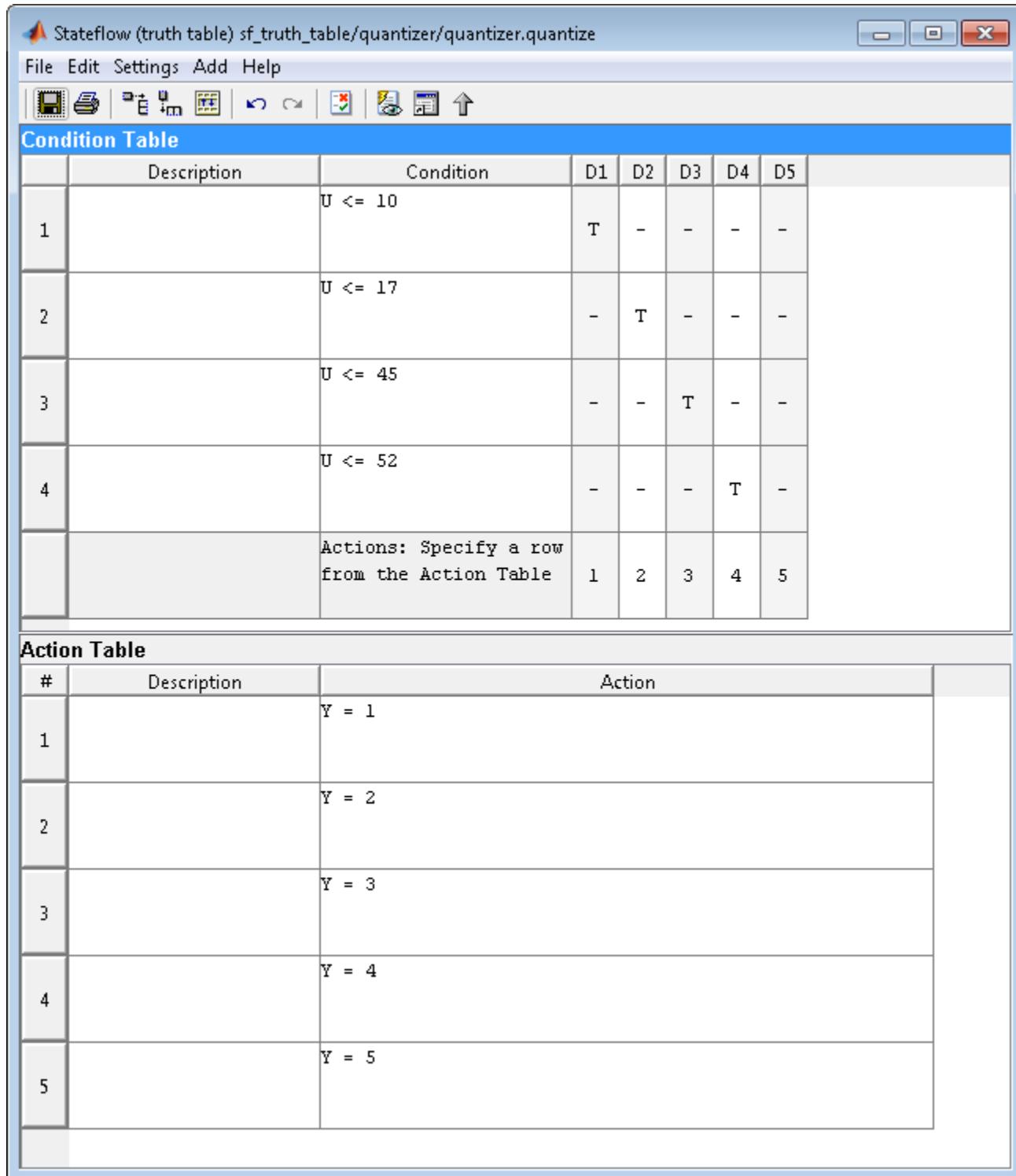
```
Y = 1 when 0 <= U <= 10
Y = 2 when 10 < U <= 17
Y = 3 when 17 < U <= 45
Y = 4 when 45 < U <= 52
Y = 5 when 52 < U
```

A stateless chart with a single call to a Truth Table function can represent this logic succinctly.

The following figure shows the `quantizer` chart, containing the Truth Table.



The following figure shows the threshold logic, as displayed in the Truth Table Editor.



The following code excerpt shows VHDL code generated for the quantizer chart.

```
quantizer : PROCESS (Y_reg, U)
  -- local variables
  VARIABLE aVarTruthTableCondition_1 : std_logic;
  VARIABLE aVarTruthTableCondition_2 : std_logic;
```

```

VARIABLE aVarTruthTableCondition_3 : std_logic;
VARIABLE aVarTruthTableCondition_4 : std_logic;
BEGIN
    Y_reg_next <= Y_reg;
    -- Condition #1
    aVarTruthTableCondition_1 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(10, 8));
    -- Condition #2
    aVarTruthTableCondition_2 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(17, 8));
    -- Condition #3
    aVarTruthTableCondition_3 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(45, 8));
    -- Condition #4
    aVarTruthTableCondition_4 := tmw_to_stdlogic(unsigned(U) <= to_unsigned(52, 8));

    IF tmw_to_boolean(aVarTruthTableCondition_1) THEN
        -- D1
        -- Action 1
        Y_reg_next <= to_unsigned(1, 8);
    ELSIF tmw_to_boolean(aVarTruthTableCondition_2) THEN
        -- D2
        -- Action 2
        Y_reg_next <= to_unsigned(2, 8);
    ELSIF tmw_to_boolean(aVarTruthTableCondition_3) THEN
        -- D3
        -- Action 3
        Y_reg_next <= to_unsigned(3, 8);
    ELSIF tmw_to_boolean(aVarTruthTableCondition_4) THEN
        -- D4
        -- Action 4
        Y_reg_next <= to_unsigned(4, 8);
    ELSE
        -- Default
        -- Action 5
        Y_reg_next <= to_unsigned(5, 8);
    END IF;

END PROCESS quantizer;

```

Note When generating code for a Truth Table block in a Simulink model, HDL Coder writes a separate entity/architecture file for the Truth Table code. The file is named `Truth_Table.vhd` (for VHDL) or `Truth_Table.v` (for Verilog).

See Also

[Sequence Viewer](#) | [State Transition Table](#) | [Truth Table](#)

Related Examples

- “Generate HDL for Mealy and Moore Finite State Machines” on page 28-7

More About

- “Hardware Realization of Stateflow Semantics” on page 28-6
- “Introduction to Stateflow HDL Code Generation” on page 28-2

Initialize Persistent Variables in MATLAB Functions

A persistent variable is a local variable in a MATLAB function that retains its value in memory between calls to the function. For code generation, functions must initialize a persistent variable if it is empty. For more information, see [persistent](#).

When programming MATLAB functions in these situations:

- MATLAB Function blocks with no direct feedthrough
- MATLAB Function blocks in models that contain State Control blocks in Synchronous mode
- MATLAB functions in Stateflow charts that implement Moore machine semantics

The specialized semantics impact how a function initializes its persistent data. Because the initialization must be independent of the input to the function, follow these guidelines:

- The function initializes its persistent variables only by accessing constants.
- The control flow of the function does not depend on whether the initialization occurs.

For example, this function has a persistent variable `n`.

```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = u;
        y = 1;
        return
    end

    y = n;
    n = n + u;
end
```

This type of initialization results in an error because the initial value of `n` depends on the input `u` and the `return` statement interrupts the normal control flow of the function.

To correct the error, initialize the persistent variable by setting it to a constant value and remove the `return` statement. For example, this function initializes the persistent variable without producing an error.

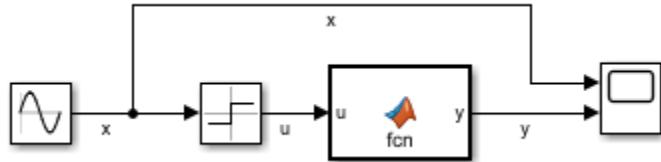
```
function y = fcn(u)
    persistent n

    if isempty(n)
        n = 1;
    end

    y = n;
    n = n + u;
end
```

MATLAB Function Block With No Direct Feedthrough

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input `u` is a square wave with values of 1 and -1.

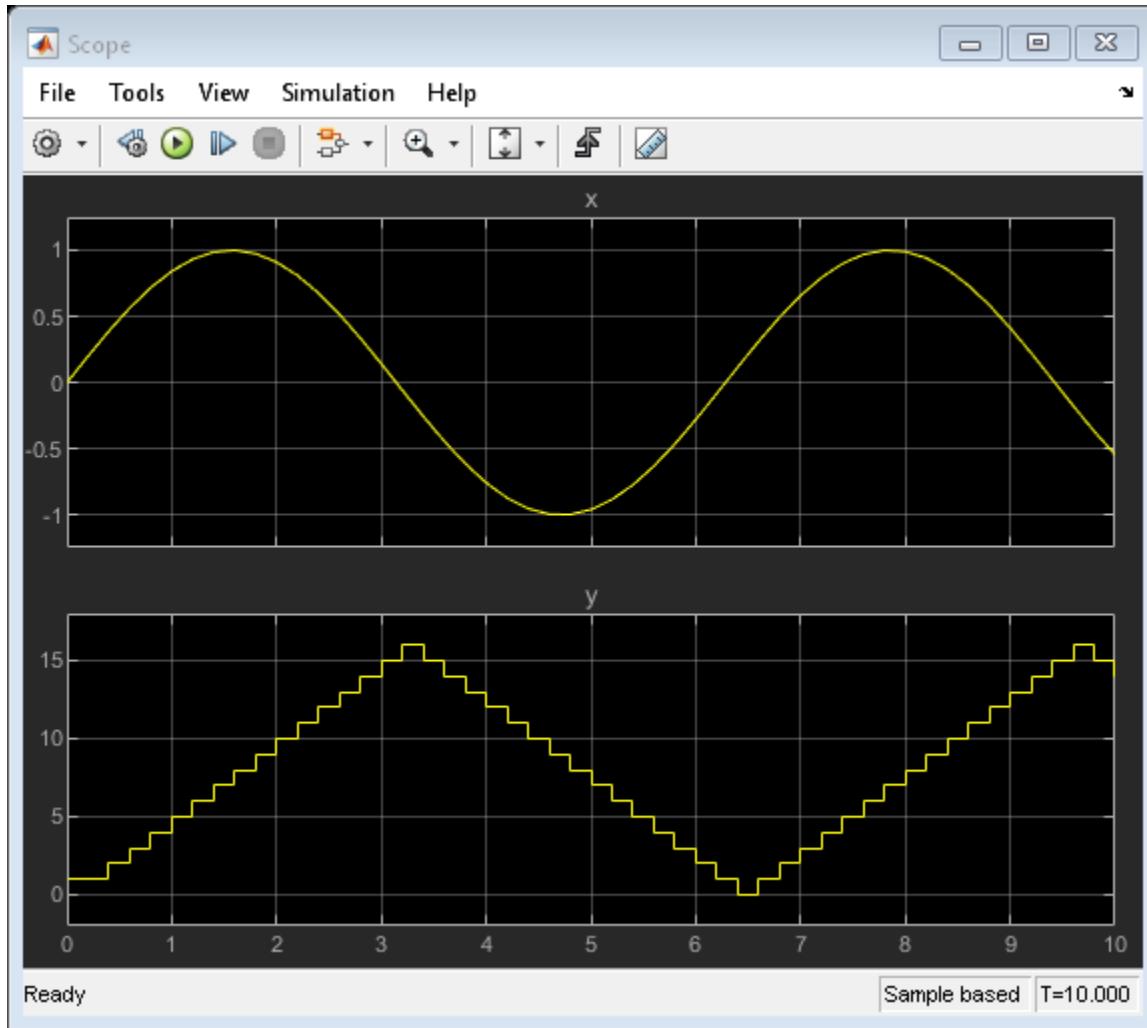


In the MATLAB function block:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

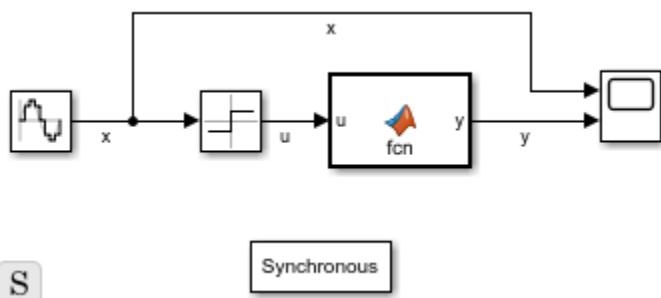
Because the **Allow direct feedthrough** check box is cleared, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



State Control Block in Synchronous Mode

This model contains a MATLAB function block that defines the function `fcn`, described previously. The input u is a square wave with values of 1 and -1.

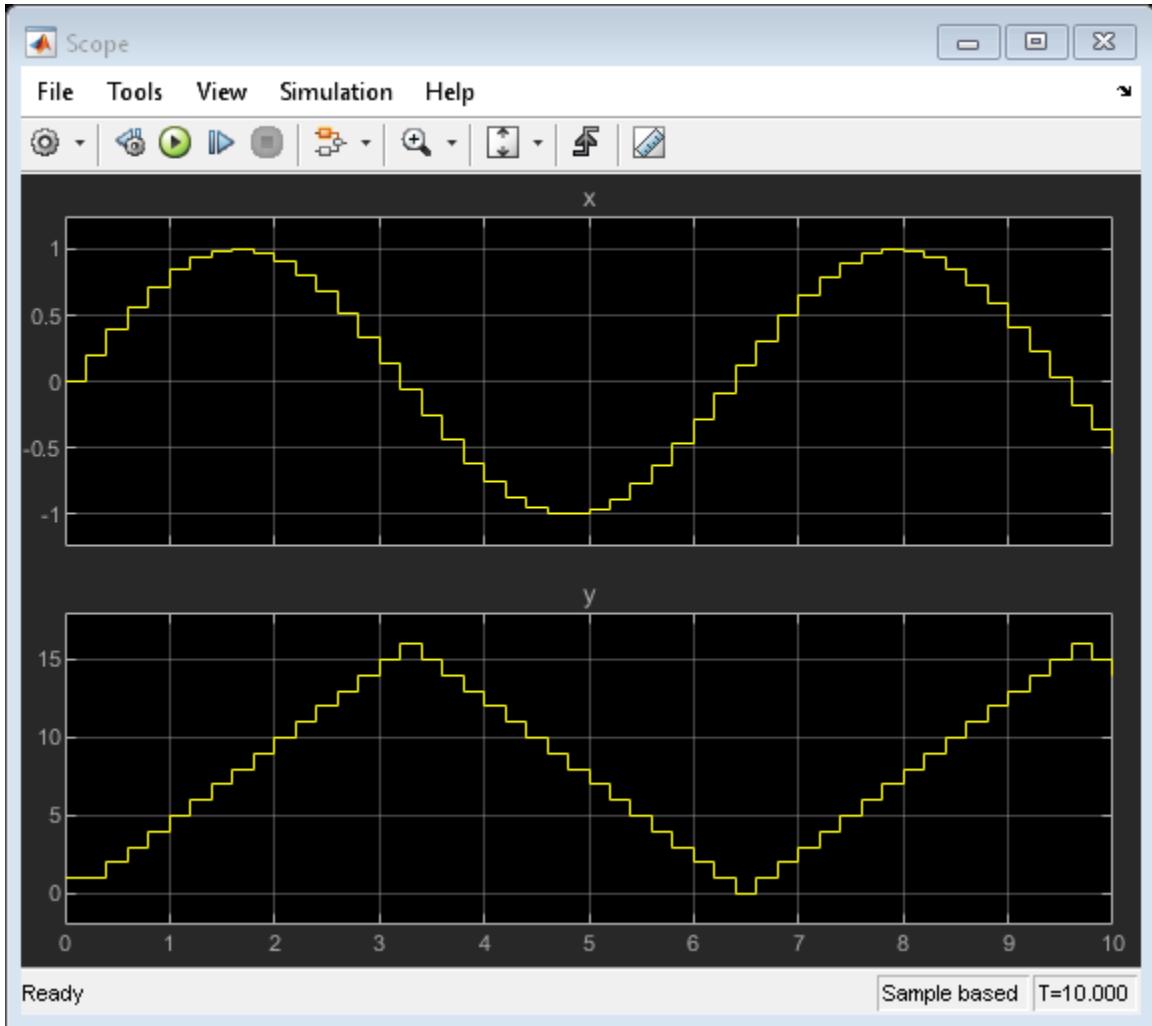


In the MATLAB function block:

- The initial value of the persistent variable n depends on the input u .
- The `return` statement interrupts the normal control flow of the function.

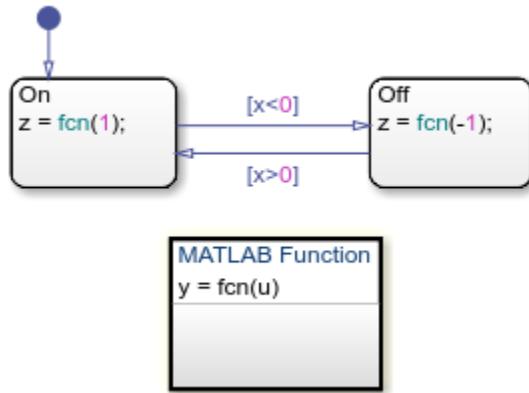
Because the model contains a State Control block in Synchronous mode, the initialization results in an error.

If you modify the function so it initializes n independently of the input, then you can simulate an error-free model.



Stateflow Chart Implementing Moore Semantics

This model contains a Stateflow chart that implements Moore machine semantics. The chart contains a MATLAB function that defines the function `fcn`, described previously. The input u has values of 1 and -1 that depend on the state of the chart.

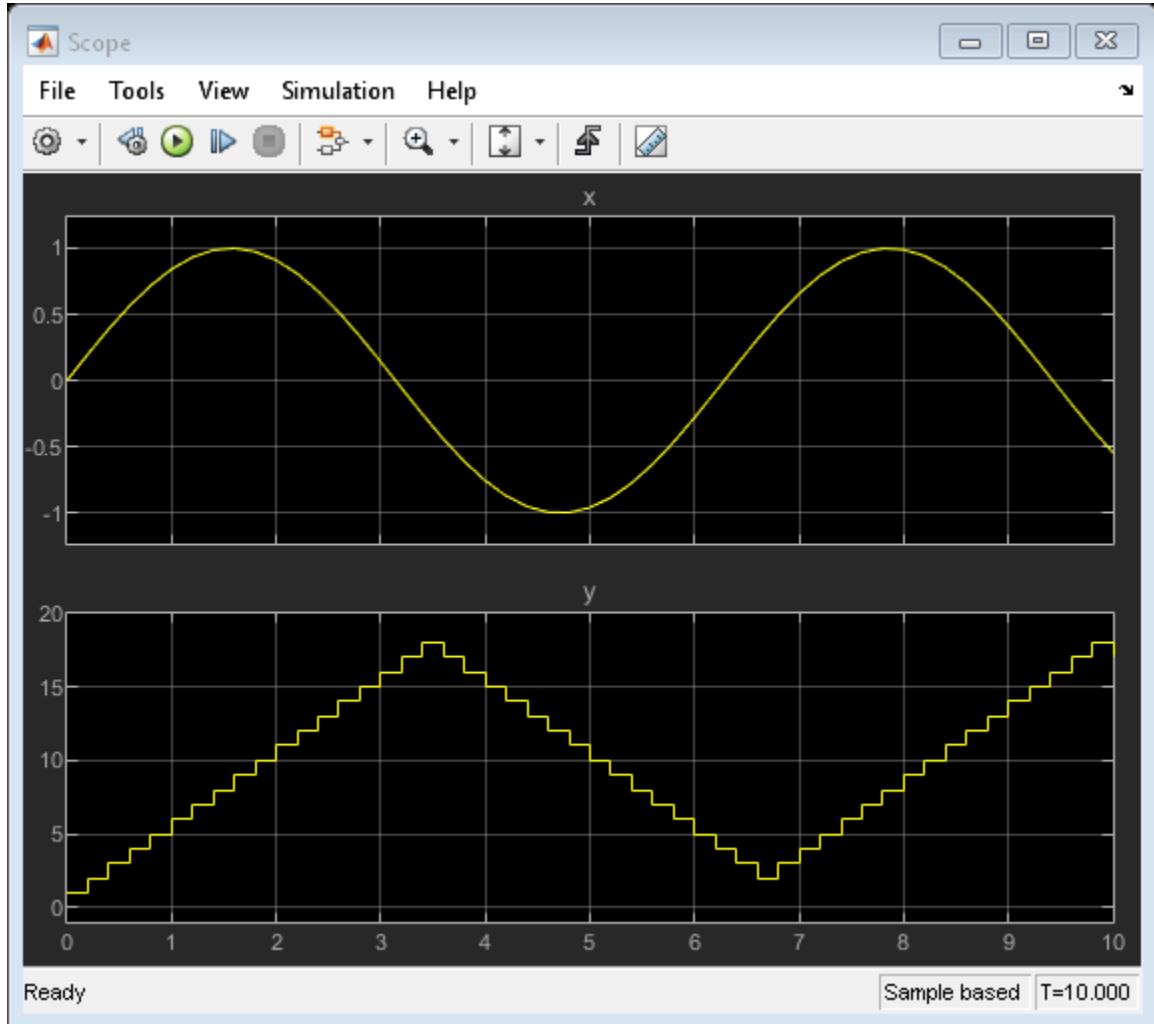


In the MATLAB function:

- The initial value of the persistent variable `n` depends on the input `u`.
- The `return` statement interrupts the normal control flow of the function.

Because the chart implements Moore semantics, the initialization results in an error.

If you modify the function so it initializes `n` independently of the input, then you can simulate an error-free model.



See Also

[Chart](#) | [MATLAB Function](#) | [State Control](#) | [persistent](#)

More About

- “Use Nondirect Feedthrough in a MATLAB Function Block”
- “Synchronous Subsystem Behavior with the State Control Block” on page 27-85
- “Design Considerations for Moore Charts” (Stateflow)

Generating HDL Code with the MATLAB Function Block

- “HDL Applications for the MATLAB Function Block” on page 29-2
- “Viterbi Decoder with the MATLAB Function Block” on page 29-4
- “Code Generation from a MATLAB Function Block” on page 29-5
- “Generate Instantiable Code for Functions” on page 29-17
- “MATLAB Function Block Design Patterns for HDL” on page 29-19
- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “CORDIC Algorithm Using the MATLAB® Function Block” on page 29-32
- “Hardware Design Patterns Using the MATLAB Function Block” on page 29-33
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37

HDL Applications for the MATLAB Function Block

In this section...

["Structure of Generated HDL Code" on page 29-2](#)

["HDL Applications" on page 29-2](#)

Structure of Generated HDL Code

The MATLAB Function block contains a MATLAB function in a model. The inputs and outputs of the function are represented by the ports on the block, which allow you to interface your model to the function code. When you generate HDL code for a MATLAB Function block, HDL Coder generates two HDL files:

- A file containing entity and architecture code that implement the actual algorithm or computations generated for the MATLAB Function block.
- A file containing an entity definition and RTL architecture that provide a black box interface to the algorithmic code generated for the MATLAB Function block.

The structure of these code files is analogous to the structure of the model, in which a subsystem provides an interface between the root model and the function in the MATLAB Function block.

HDL Applications

The MATLAB Function block supports a subset of the MATLAB language that is well-suited to HDL implementation of various DSP and telecommunications algorithms, such as:

- Sequence and pattern generators
- Encoders and decoders
- Interleavers and de-interleavers
- Modulators and demodulators
- Multipath channel models; impairment models
- Timing recovery algorithms
- Viterbi algorithm; Maximum Likelihood Sequence Estimation (MLSE)
- Adaptive equalizer algorithms

You can also use the MATLAB Function block in a wide variety of floating-point applications. Both `single` and `double` types are supported. To learn more, see ["HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture" on page 24-146](#).

See Also

["Check for MATLAB Function block settings" on page 38-19](#)

More About

- ["Design Guidelines for the MATLAB Function Block" on page 29-29](#)
- ["Code Generation from a MATLAB Function Block" on page 29-5](#)

- “MATLAB Function Block Design Patterns for HDL” on page 29-19
- “Generate DUT Ports for Tunable Parameters” on page 10-17

Viterbi Decoder with the MATLAB Function Block

`hdlcoderviterbi2` models a Viterbi decoder, incorporating a MATLAB Function block for use in simulation and HDL code generation. To open the model, type the following at the MATLAB command prompt:

```
hdlcoderviterbi2
```

See Also

["Check for MATLAB Function block settings" on page 38-19](#)

More About

- ["Design Guidelines for the MATLAB Function Block" on page 29-29](#)
- ["Code Generation from a MATLAB Function Block" on page 29-5](#)
- ["MATLAB Function Block Design Patterns for HDL" on page 29-19](#)
- ["Generate DUT Ports for Tunable Parameters" on page 10-17](#)

Code Generation from a MATLAB Function Block

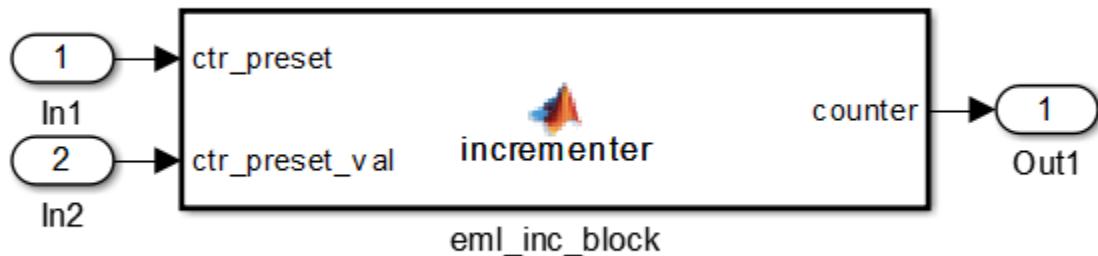
In this section...

- “Counter Model Using the MATLAB Function block” on page 29-5
- “Setting Up” on page 29-7
- “Creating the Model and Configuring General Model Settings” on page 29-7
- “Adding a MATLAB Function Block to the Model” on page 29-8
- “Set Fixed-Point Options for the MATLAB Function Block” on page 29-8
- “Programming the MATLAB Function Block” on page 29-10
- “Constructing and Connecting the DUT_eML_Block Subsystem” on page 29-11
- “Compiling the Model and Displaying Port Data Types” on page 29-13
- “Simulating the eml_hdl_incremter_tut Model” on page 29-13
- “Generating HDL Code” on page 29-14

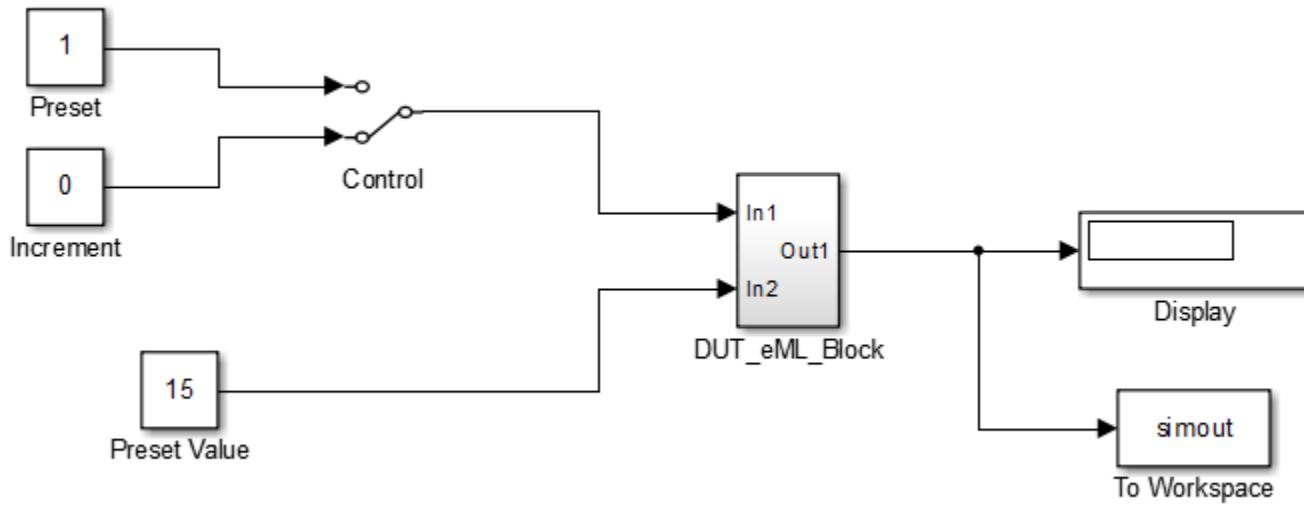
Counter Model Using the MATLAB Function block

In this tutorial, you construct and configure a simple model, `eml_hdl_incremter_tut`, and then generate VHDL code from the model. `eml_hdl_incremter_tut` includes a MATLAB Function block that implements a simple fixed-point counter function, `incremter`. The `incremter` function is invoked once during each sample period of the model. The function maintains a persistent variable `count`, which is either incremented or reinitialized to a preset value (`ctr_preset_val`), depending on the value passed in to the `ctr_preset` input of the MATLAB Function block. The function returns the counter value (`counter`) at the output of the MATLAB Function block.

The MATLAB Function block resides in a subsystem, `DUT_eML_Block`. The subsystem functions as the device under test (DUT) from which you generate HDL code.



The root-level model drives the subsystem and includes Display and To Workspace blocks for use in simulation. (The Display and To Workspace blocks do not generate HDL code.)



Tip If you do not want to construct the model step by step, or do not have time, you can open the completed model by entering the name at the command prompt:

`eml_hdl_incremener`

After you open the model, save a copy of it to your local folder as `eml_hdl_incremener_tut`.

The Incrementer Function Code

The following code listing gives the complete `incremener` function definition:

```

function counter = incremener(ctr_preset, ctr_preset_val)
% The function incremener implements a preset counter that counts
% how many times this block is called.
%
% This example function shows how to model memory with persistent variables,
% using fimath settings suitable for HDL. It also demonstrates MATLAB
% operators and other language features that HDL Coder supports
% for code generation from Embedded MATLAB Function block.
%
% On the first call, the result 'counter' is initialized to zero.
% The result 'counter' saturates if called more than 2^14-1 times.
% If the input ctr_preset receives a nonzero value, the counter is
% set to a preset value passed in to the ctr_preset_val input.

persistent current_count;
if isempty(current_count)
    % zero the counter on first call only
    current_count = uint32(0);
end

counter = getfi(current_count);

if ctr_preset
    % set counter to preset value if input preset signal is nonzero
    counter = ctr_preset_val;
else
    % otherwise count up
    inc = counter + getfi(1);
end

```

```

    counter = getfi(inc);
end

% store counter value for next iteration
current_count = uint32(counter);

function hdl_fi = getfi(val)

nt = numerictype(0,14,0);
fm = hdlfimath;
hdl_fi = fi(val, nt, fm);

```

Setting Up

Before you begin building the example model, set up a working folder for your model and generated code.

Setting Up a folder

- 1 Start MATLAB.
- 2 Create a folder named `eml_tut`, for example:

```
mkdir D:\work\eml_tut
```

The `eml_tut` folder stores the model you create, and also contains sub-folders and generated code. The location of the folder does not matter, except that it should not be within the MATLAB tree.

- 3 Make the `eml_tut` folder your working folder, for example:

```
cd D:\work\eml_tut
```

Creating the Model and Configuring General Model Settings

In this section, you create a model and set some parameters to values recommended for HDL code generation `hdlsetup` command. The `hdlsetup` command uses the `set_param` function to set up models for HDL code generation quickly and consistently. See “Customize `hdlsetup` Function Based on Target Application” on page 21-18 for further information about `hdlsetup`.

To set the model parameters:

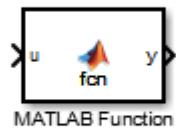
- 1 Create a new model.
- 2 Save the model as `eml_hdl_incremter_tut`.
- 3 At the MATLAB command prompt, type:

```
hdlsetup('eml_hdl_incremter_tut');
```

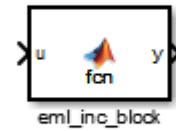
- 4 Open the Configuration Parameters dialog box.
- 5 Set the following **Solver** options, which are useful in simulating this model:
 - **Fixed step size:** 1
 - **Stop time:** 5
- 6 Click **OK** to save your changes and close the Configuration Parameters dialog box.
- 7 Save your model.

Adding a MATLAB Function Block to the Model

- 1 Open the Simulink Library Browser. Then, select the Simulink/User-Defined Functions library.
- 2 Select the MATLAB Function block from the library window and add it to the model.



- 3 Change the block label from MATLAB Function to eml_inc_block.



- 4 Save the model.
- 5 Close the Simulink Library Browser.

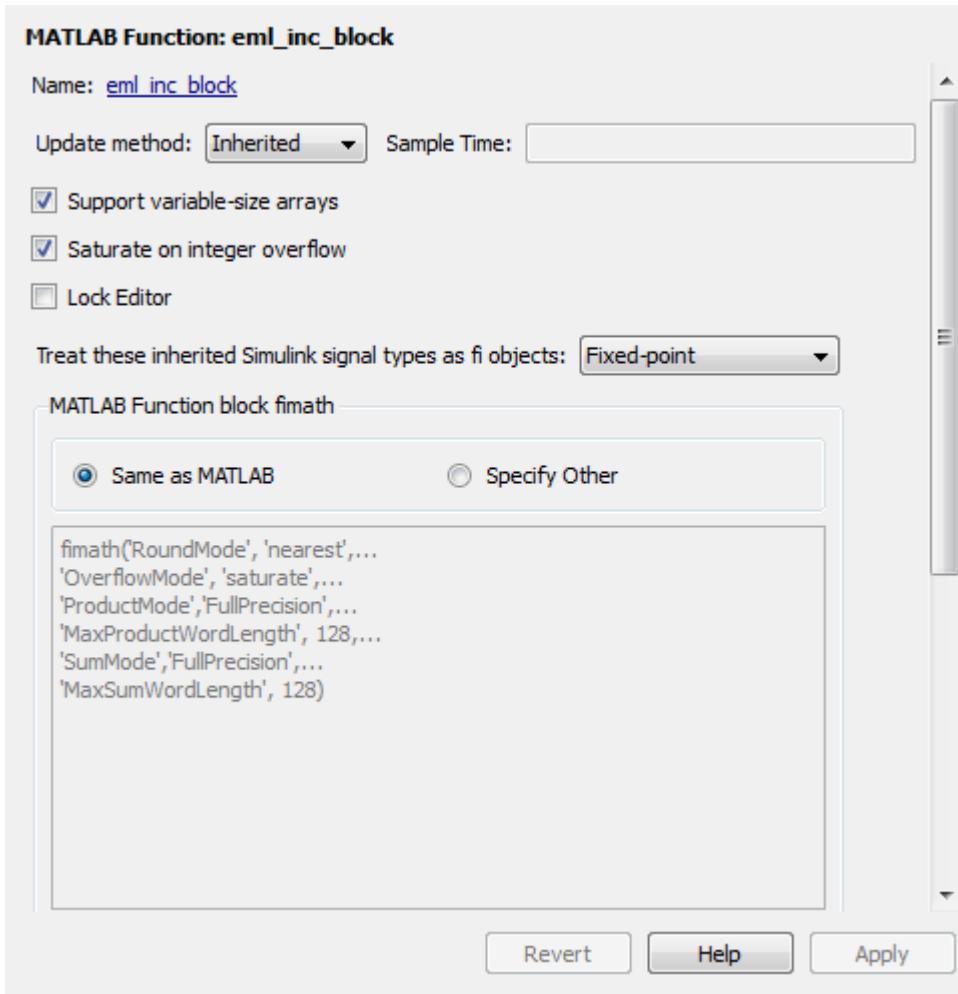
Set Fixed-Point Options for the MATLAB Function Block

This section describes how to set up the `fimath` specification and other fixed-point options that are recommended for efficient HDL code generation from the MATLAB Function block. The recommended settings are:

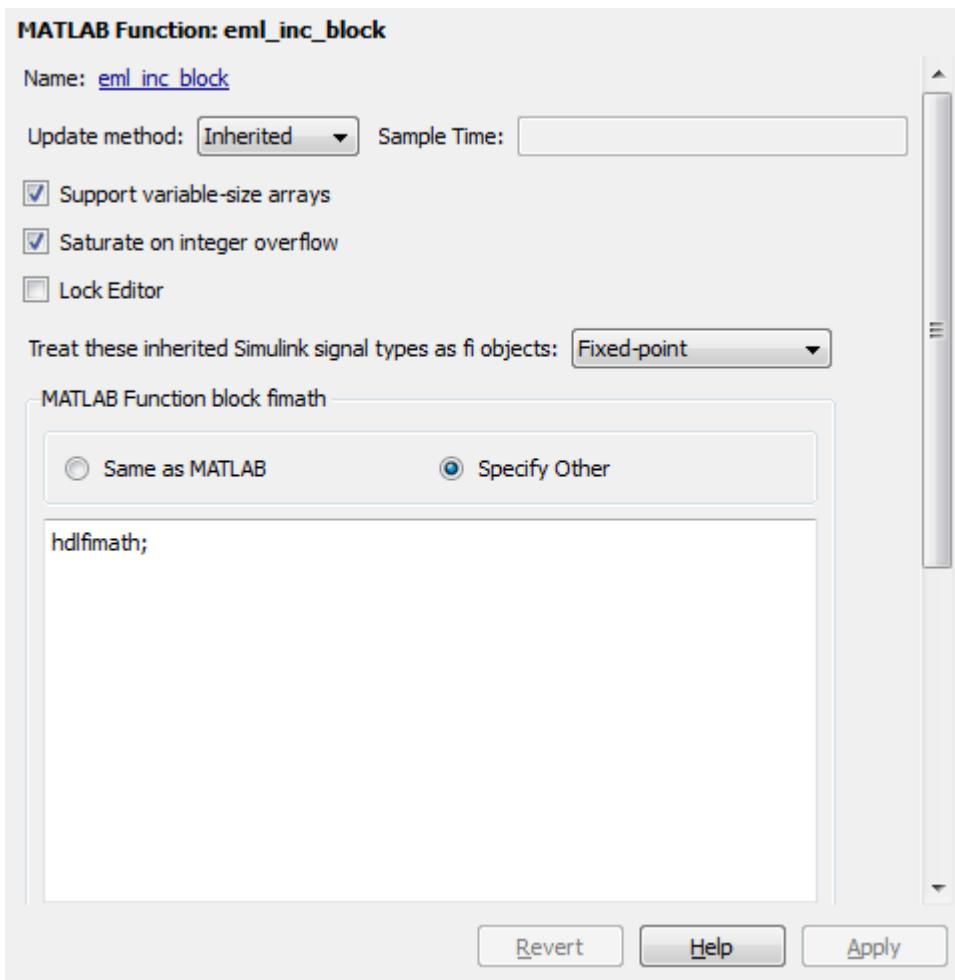
- ProductMode property of the `fimath` specification: 'FullPrecision'
- SumMode property of the `fimath` specification: 'FullPrecision'
- **Treat these inherited signal types as fi objects** option: Fixed-point (This is the default setting.)

Configure the options as follows:

- 1 Open the `eml_hdl_incremter_tut` model that you created in "Adding a MATLAB Function Block to the Model" on page 29-8.
- 2 Double-click the MATLAB Function block to open it for editing. The MATLAB Function Block Editor appears.
- 3 Click **Edit Data**. The Ports and Data Manager dialog box opens, displaying the default `fimath` specification and other properties for the MATLAB Function block.



- 4 Select **Specify Other**. Selecting this option enables the **MATLAB Function block fimath** text entry field.
 - 5 The `hdlfimath` function is a utility that defines a FIMATH specification that is optimized for HDL code generation. Replace the default **MATLAB Function block fimath** specification with a call to `hdlfimath` as follows:
- ```
hdlfimath;
```
- 6 Click **Apply**. The MATLAB Function block properties should now appear as shown in the following figure.



- 7 Close the Ports and Data Manager.
- 8 Save the model.

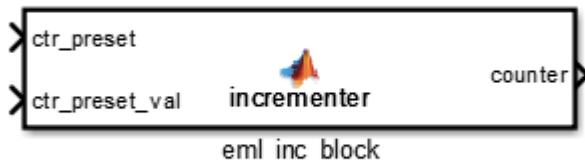
## Programming the MATLAB Function Block

The next step is add code to the MATLAB Function block to define the `incrementer` function, and then use diagnostics to check for errors.

- 1 Open the `eml_hdl_incremente_tut` model that you created in “Adding a MATLAB Function Block to the Model” on page 29-8.
- 2 Double-click the MATLAB Function block to open it for editing.
- 3 In the MATLAB Function Block Editor, delete the default code.
- 4 Copy the complete `incrementer` function definition from the listing given in “The Incrementer Function Code” on page 29-6, and paste it into the editor.
- 5 Save the model. Doing so updates the model window, redrawing the MATLAB Function block.

Changing the function header of the MATLAB Function block makes the following changes to the block icon:

- The function name in the middle of the block changes to `incrementer`.
  - The arguments `ctr_preset` and `ctr_preset_val` appear as input ports to the block.
  - The return value `counter` appears as an output port from the block.
- 6** Resize the block to make the port labels more legible.



- 7** Save the model again.

## Constructing and Connecting the DUT\_eML\_Block Subsystem

This section assumes that you have completed “Programming the MATLAB Function Block” on page 29-10 without encountering an error. In this section, you construct a subsystem containing the `incrementer` function block, to be used as the device under test (DUT) from which to generate HDL code. You then set the port data types and connect the subsystem ports to the model.

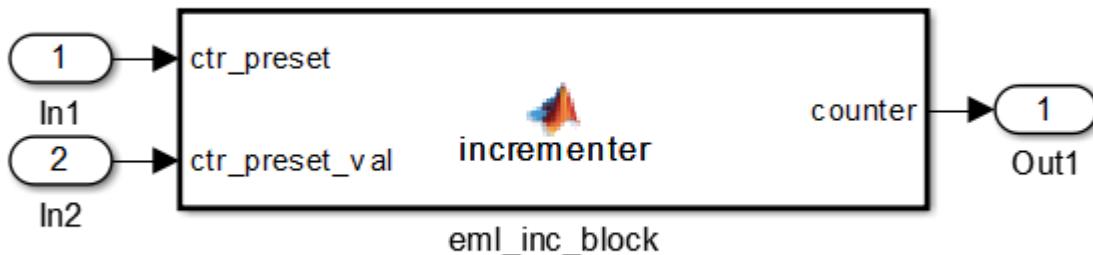
### Constructing the DUT\_eML\_Block Subsystem

Construct a subsystem containing the `incrementer` function block as follows:

- 1 Click the `incrementer` function block.
- 2 On the **Modeling** tab of the Simulink Toolstrip, select **Create Subsystem**. A subsystem, labeled `Subsystem`, is created in the model window.
- 3 Change the `Subsystem` label to `DUT_eML_Block`.

### Setting Port Data Types for the MATLAB Function Block

- 1 Double-click the subsystem to view its interior. As shown in the following figure, the subsystem contains the `incrementer` function block, with input and output ports connected.



- 2 Double-click the `incrementer` function block to open the MATLAB Function Block Editor.
- 3 In the editor, click **Edit Data** to open the Ports and Data Manager.
- 4 Select the `ctr_preset` entry in the port list on the left. Click the button labeled **>>** to display the Data Type Assistant. Set **Mode** for this port to `Built in`. Set **Data type** to `boolean`. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.
- 5 Select the `ctr_preset_val` entry in the port list on the left. Click the button labeled **>>** to display the Data Type Assistant. Set **Mode** for this port to `Fixed point`. Set **Signedness** to `Unsigned`. Set **Word length** to 14. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.
- 6 Select the `counter` entry in the port list on the left. Click the button labeled **>>** to display the Data Type Assistant. Verify that **Mode** for this port is set to `Inherit: Same as Simulink`. Click the button labeled **<<** to close the Data Type Assistant. Click **Apply**.
- 7 Close the Ports and Data Manager dialog box and the MATLAB Function Block Editor.
- 8 Save the model and close the `DUT_eML_Block` subsystem.

### Connecting Subsystem Ports to the Model

Next, connect the ports of the `DUT_eML_Block` subsystem to the model as follows:

- 1 From the Sources library, add a Constant block to the model. Set the value of the Constant block to 1, and the **Output data type** to `boolean`. Change the block label to `Preset`.
- 2 Make a copy of the Preset Constant block. Set its value to 0, and change its block label to `Increment`.
- 3 From the Signal Routing library, add a Manual Switch block to the model. Change its label to `Control`. Connect its output to the `In1` port of the `DUT_eML_Block` subsystem.
- 4 Connect the Preset Constant block to the upper input of the Control switch block. Connect the Increment Constant block to the lower input of the Control switch block.
- 5 Add a third Constant block to the model. Set the value of the Constant to 15, and the **Output data type** to `Inherit via back propagation`. Change the block label to `Preset Value`.
- 6 Connect the Preset Value Constant block to the `In2` port of the `DUT_eML_Block` subsystem.
- 7 From the Sinks library, add a Display block to the model. Connect it to the `Out1` port of the `DUT_eML_Block` subsystem.
- 8 From the Sinks library, add a To Workspace block to the model. Route the output signal from the `DUT_eML_Block` subsystem to the To Workspace block.
- 9 Save the model.

### Checking the Function for Errors

Use the built-in diagnostics of MATLAB Function blocks to test for syntax errors:

- 1 Open the `eml_hdl_incremter_tut` model.
- 2 Double-click the MATLAB Function block `incrementer` to open it for editing.
- 3 In the MATLAB Function Block Editor, select **Build Model > Build** to compile and build the MATLAB Function block code.

The build process displays some progress messages. These messages include some warnings, because the ports of the MATLAB Function block are not yet connected to signals. You can ignore these warnings.

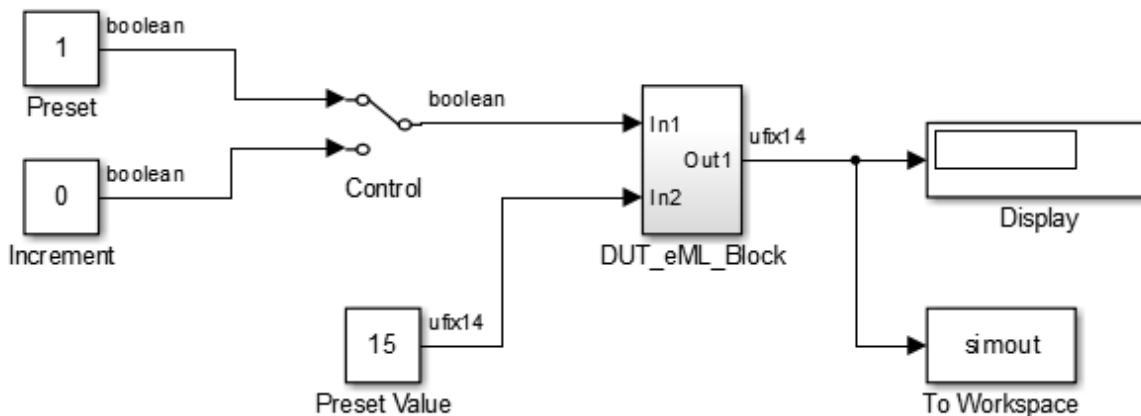
The build process builds an S-function for use in simulation. The build process includes generation of C code for the S-function. The code generation messages you see during the build process refer to generation of C code, not HDL code generation.

When the build concludes without encountering an error, a message window appears indicating that parsing was successful. If errors are found, the Diagnostics Manager lists them. See the MATLAB Function block documentation for information on debugging MATLAB Function block build errors.

## Compiling the Model and Displaying Port Data Types

In this section you enable the display of port data types and then compile the model. Model compilation verifies the model structure and settings, and updates the model display.

- 1 In the **Debug** tab of the Simulink Toolstrip, on the **Information Overlays > Ports** section, select **Base data types**.
- 2 Press **Ctrl+D** to compile and update the model. This triggers a rebuild of the code. After the model compiles, the block diagram updates to show the port data types.

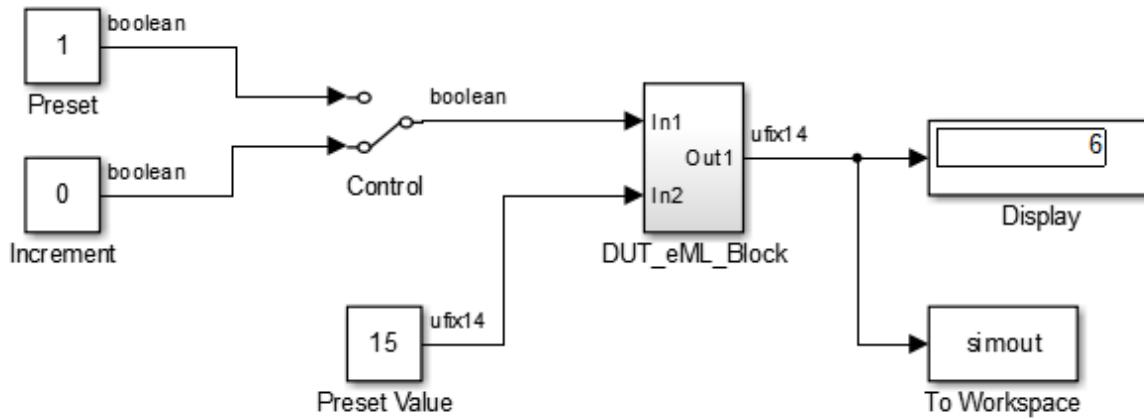


- 3 Save the model.

## Simulating the `eml_hdl_incremener_tut` Model

Start simulation. If required, the code rebuilds before the simulation starts.

After the simulation completes, the Display block shows the final output value returned by the `incrementer` function block. For example, given a **Start time** of 0, a **Stop time** of 5, and a zero value at the `ctr_preset` port, the simulation returns a value of 6:



You might want to experiment with the results of toggling the **Control** switch, changing the **Preset Value** constant, and changing the total simulation time. You might also want to examine the workspace variable **simout**, which is bound to the **To Workspace** block.

## Generating HDL Code

In this section, you select the **DUT\_eML\_Block** subsystem for HDL code generation, set basic code generation options, and then generate VHDL code for the subsystem.

### Selecting the Subsystem for Code Generation

Select the **DUT\_eML\_Block** subsystem for code generation:

- 1 Open the Configuration Parameters dialog box and click the **HDL Code Generation** pane.
- 2 Select **eml\_hdl\_incremter\_tut/DUT\_eML\_Block** from the **Generate HDL for** list.
- 3 Click **Apply**.

### Generating VHDL Code

In the Configuration Parameters dialog box, the top-level **HDL Code Generation** options should now be set as follows:

- The **Generate HDL for** field specifies the **eml\_hdl\_incremter\_tut/DUT\_eML\_Block** subsystem for code generation.
- The **Language** field specifies (by default) generation of VHDL code.
- The **Folder** field specifies (by default) that the code generation target folder is a subfolder of your working folder, named **hdlsrc**.

Before generating code, select **Current Folder** from the **Layout** menu in the MATLAB Command Window. This displays the Current Folder browser, which lets you easily access your working folder and the files that are generated within it.

To generate code:

- 1 Click the **Generate** button.

HDL Coder compiles the model before generating code. Depending on model display options (such as port data types), the appearance of the model might change after code generation.

- 2 As code generation proceeds, the coder displays progress messages. The process should complete with a message like the following:

```
HDL Code Generation Complete.
```

The names of generated VHDL files in the progress messages are hyperlinked. After code generation completes, you can click these hyperlinks to view the files in the MATLAB Editor.

- 3 A folder icon for the `hdlsrc` folder is now visible in the Current Folder browser. To view generated code and script files, double-click the `hdlsrc` folder icon.
- 4 Observe that two VHDL files were generated. The structure of HDL code generated for MATLAB Function blocks is similar to the structure of code generated for Stateflow charts and Digital Filter blocks. The VHDL files that were generated in the `hdlsrc` folder are:
  - `eml_inc_blk.vhd`: VHDL code. This file contains entity and architecture code implementing the actual computations generated for the MATLAB Function block.
  - `DUT_eML_Block.vhd`: VHDL code. This file contains an entity definition and RTL architecture that provide a black box interface to the code generated in `eml_inc_blk.vhd`.

The structure of these code files is analogous to the structure of the model, in which the `DUT_eML_Block` subsystem provides an interface between the root model and the `incrementer` function in the MATLAB Function block.

The other files generated in the `hdlsrc` folder are:

- `DUT_eML_Block_compile.do`: Mentor Graphics ModelSim compilation script (`vcom` command) to compile the VHDL code in the two `.vhd` files.
  - `DUT_eML_Block_synplify.tcl`: Synplify synthesis script.
  - `DUT_eML_Block_map.txt`: Mapping file. This report file maps generated entities (or modules) to the subsystems that generated them (see “Trace Code Using the Mapping File” on page 25-21).
- 5 To view the generated VHDL code in the MATLAB Editor, double-click the `DUT_eML_Block.vhd` or `eml_inc_blk.vhd` file icons in the Current Folder browser.

## See Also

“Check for MATLAB Function block settings” on page 38-19

## More About

- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “HDL Applications for the MATLAB Function Block” on page 29-2

- “MATLAB Function Block Design Patterns for HDL” on page 29-19
- “Generate DUT Ports for Tunable Parameters” on page 10-17

# Generate Instantiable Code for Functions

## In this section...

["How To Generate Instantiable Code for Functions" on page 29-17](#)

["Generate Code Inline for Specific Functions" on page 29-17](#)

["Limitations for Instantiable Code Generation for Functions" on page 29-17](#)

For the MATLAB Function block, you can use the **InstantiateFunctions** parameter to generate a VHDL entity or Verilog module for each function. HDL Coder generates code for each entity or module in a separate file.

The **InstantiateFunctions** options for the MATLAB Function block are listed in the following table.

| InstantiateFunctions Setting | Description                                                                                                    |
|------------------------------|----------------------------------------------------------------------------------------------------------------|
| 'off' (default)              | Generate code for functions inline.                                                                            |
| 'on'                         | Generate a VHDL entity or Verilog module for each function, and save each module or entity in a separate file. |

## How To Generate Instantiable Code for Functions

To set the **InstantiateFunctions** parameter using the HDL Block Properties dialog box:

- 1 Right-click the MATLAB Function block.
- 2 Select **HDL Code > HDL Block Properties**.
- 3 For **InstantiateFunctions**, select **on**.

To set the **InstantiateFunctions** parameter from the command line, use `hdlset_param`. For example, to generate instantiable code for functions in a MATLAB Function block, `myMatlabFcn`, in your DUT subsystem, `myDUT`, enter:

```
hdlset_param('my_DUT/my_MATLABFcnBlk', 'InstantiateFunctions', 'on')
```

## Generate Code Inline for Specific Functions

If you want to generate instantiable code for some functions but not others, enable the option to generate instantiable code for functions, and use `coder.inline`. See `coder.inline` for details.

## Limitations for Instantiable Code Generation for Functions

The software generates code inline when:

- Function calls are within conditional code or `for` loops.
- Any function is called with a nonconstant `struct` input.
- The function has state, such as a persistent variable, and is called multiple times.
- There is an enumeration anywhere in the design function.

**See Also**

"Check for MATLAB Function block settings" on page 38-19

**More About**

- "Design Guidelines for the MATLAB Function Block" on page 29-29
- "Code Generation from a MATLAB Function Block" on page 29-5
- "MATLAB Function Block Design Patterns for HDL" on page 29-19
- "Generate DUT Ports for Tunable Parameters" on page 10-17

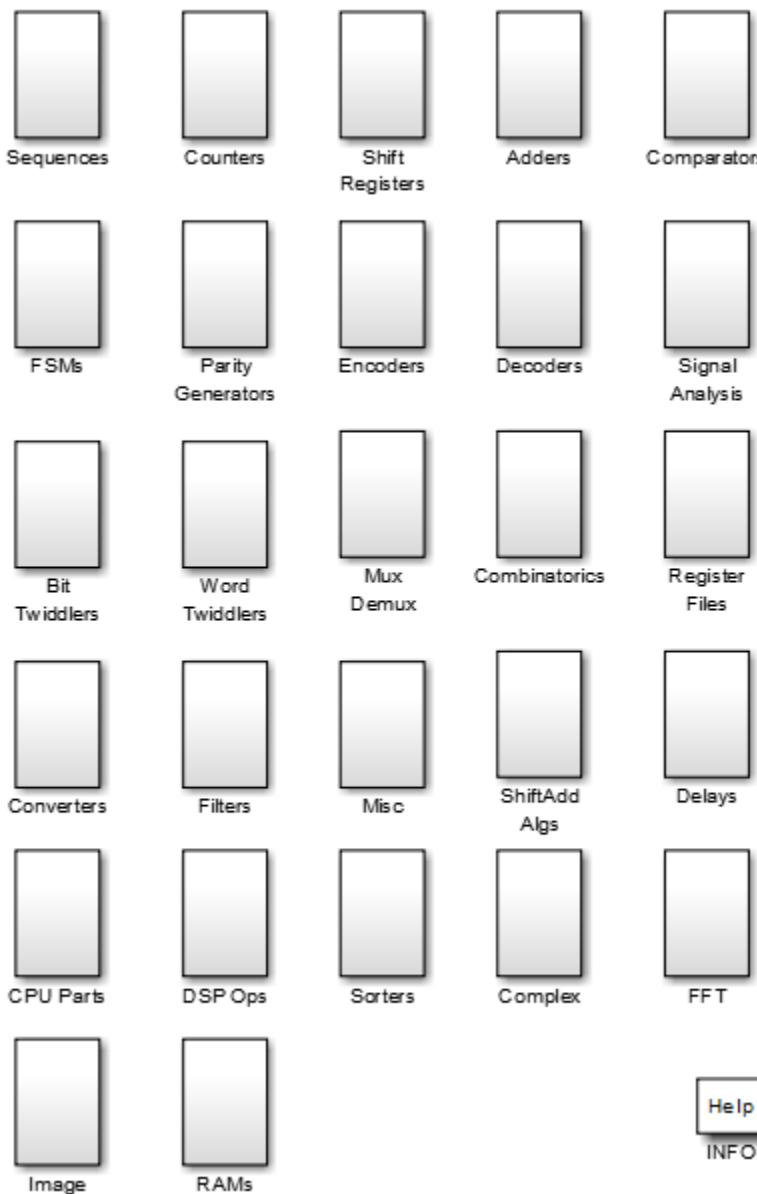
# MATLAB Function Block Design Patterns for HDL

## In this section...

- “The `eml_hdl_design_patterns` Library” on page 29-19
- “Efficient Fixed-Point Algorithms” on page 29-21
- “Model State Using Persistent Variables” on page 29-23
- “Creating Intellectual Property with the MATLAB Function Block” on page 29-24
- “Nontunable Parameter Arguments” on page 29-24
- “Modeling Control Logic and Simple Finite State Machines” on page 29-24
- “Modeling Counters” on page 29-26
- “Modeling Hardware Elements” on page 29-26
- “Decimal to Binary Conversion” on page 29-27

## The `eml_hdl_design_patterns` Library

The `eml_hdl_design_patterns` library is an extensive collection of examples demonstrating useful applications of the MATLAB Function block in HDL code generation.



To open the library, type the following command at the MATLAB prompt:

```
eml_hdl_design_patterns
```

You can use many blocks in the library as cookbook examples of various hardware elements, as follows:

- Copy a block from the library to your model and use it as a computational unit.
- Copy the code from the block and use it as a local function in an existing MATLAB Function block.

When you create custom blocks, you can control whether to inline or instantiate the HDL code generated from MATLAB Function blocks. Use the **Inline MATLAB Function block code** check box

in the **HDL Code Generation > Global Settings > Coding style** section of the Configuration Parameters dialog box. For more information, see “Inline MATLAB Function block code” on page 17-49.

---

**Note** Do not use the **Inline MATLAB Function block code** setting with the MATLAB Datapath architecture of the MATLAB Function block. Use **FlattenHierarchy** instead. For more information, see “HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture” on page 24-146.

---

## Efficient Fixed-Point Algorithms

The MATLAB Function block supports floating-point arithmetic and also fixed point arithmetic by using the Fixed-Point Designer `fi` function. This function supports rounding and saturation modes that are useful for coding algorithms that manipulate arbitrary word and fraction lengths. HDL Coder supports all `fi` rounding and overflow modes. HDL code generated from the MATLAB Function block is bit-true to MATLAB semantics. Generated code uses bit manipulation and bit access operators (for example, Slice, Extend, Reduce, Concat, etc.) that are native to VHDL and Verilog.

The following discussion shows how HDL code generated from the MATLAB Function block follows cast-before-sum semantics, in which addition and subtraction operands are cast to the result type before the addition or subtraction is performed.

Open the `eml_hdl_design_patterns` library and select the `Combinatorics/eml_expr` block. `eml_expr` implements a simple expression containing addition, subtraction, and multiplication operators with differing fixed point data types. The generated HDL code shows the conversion of this expression with fixed point operands. The MATLAB Function block uses the following code:

```
% fixpt arithmetic expression
expr = (a*b) - (a+b);

% cast the result to (sf7_En4) output type
y = fi(expr, 1, 7, 4);
```

The default `fimath` specification for the block determines the behavior of arithmetic expressions using fixed point operands inside the MATLAB Function block:

```
fimath(
 'RoundMode', 'ceil',...
 'OverflowMode', 'saturate',...
 'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
 'SumMode', 'FullPrecision', 'SumWordLength', 32,...
 'CastBeforeSum', true)
```

The data types of operands and output are as follows:

- a: (`sf5_En2`)
- b: (`sf5_En3`)
- y: (`sf7_En4`)

Before HDL code generation, this expression is broken down internally into many steps.

```
expr = (a*b) - (a+b);
```

The steps include:

```
1 tmul = a * b;
2 tadd = a + b;
3 tsub = tmul - tadd;
4 y = tsub;
```

Based on the `fimath` settings as described in “Design Guidelines for the MATLAB Function Block” on page 29-29, this expression is further broken down internally as follows:

- Based on the specified `ProductMode`, ‘`FullPrecision`’, the output type of `tmul` is computed as `(sfix10_En5)`.
- Since the `CastBeforeSum` property is set to ‘`true`’, step 2 is broken down as follows:

```
t1 = (sfix7_En3) a;
t2 = (sfix7_En3) b;
tadd = t1 + t2;
```

`sfix7_En3` is the result sum type after aligning binary points and accounting for an extra bit to account for possible overflow.

- Based on intermediate types of `tmul` (`sfix10_En5`) and `tadd` (`sfix7_En3`) the result type of the subtraction in step 3 is computed as `sfix11_En5`. Accordingly, step 3 is broken down as follows:

```
t3 = (sfix11_En5) tmul;
t4 = (sfix11_En5) tadd;
tsub = t3 - t4;
```

- Finally, the result is cast to a smaller type (`sfix7_En4`) leading to the following final expression statements:

```
tmul = a * b;
t1 = (sfix7_En3) a;
t2 = (sfix7_En3) b;
tadd = t1 + t2;
t3 = (sfix11_En5) tmul;
t4 = (sfix11_En5) tadd;
tsub = t3 - t4;
y = (sfix7_En4) tsub;
```

The following listings show the generated VHDL and Verilog code from the `eml_expr` block.

This is the VHDL code:

```
BEGIN
 --MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
 -- fixpt arithmetic expression
 -- '<S2>:1:4'
 mul_temp <= signed(a) * signed(b);
 sub_cast <= resize(mul_temp, 11);
 add_cast <= resize(signed(a & '0'), 7);
 add_cast_0 <= resize(signed(b), 7);
 add_temp <= add_cast + add_cast_0;
 sub_cast_0 <= resize(add_temp & '0' & '0', 11);
 expr <= sub_cast - sub_cast_0;
 -- cast the result to correct output type
 -- '<S2>:1:7'

 y <= "0111111" WHEN ((expr(10) = '0') AND (expr(9 DOWNTO 7) /= "000"))
 OR ((expr(10) = '0') AND (expr(7 DOWNTO 1) = "0111111"))
 ELSE
 "1000000" WHEN (expr(10) = '1') AND (expr(9 DOWNTO 7) /= "111")
 ELSE
 std_logic_vector(expr(7 DOWNTO 1) + ("0" & expr(0)));
END;
```

```
END fsm_SFHDl;
```

This is the Verilog code:

```
//MATLAB Function 'Subsystem/eml_expr': '<S2>:1'
// fixpt arithmetic expression
//<S2>:1:4'
assign mul_temp = a * b;
assign sub_cast = mul_temp;
assign add_cast = {a[4], {a, 1'b0}};
assign add_cast_0 = b;
assign add_temp = add_cast + add_cast_0;
assign sub_cast_0 = {{2{add_temp[6]}}, {add_temp, 2'b00}};
assign expr = sub_cast - sub_cast_0;
// cast the result to correct output type
//<S2>:1:7'
assign y = (((expr[10] == 0) && (expr[9:7] != 0))
 || ((expr[10] == 0) && (expr[7:1] == 63)) ? 7'sb0111111 :
 ((expr[10] == 1) && (expr[9:7] != 7) ? 7'sb1000000 :
 expr[7:1] + $signed({1'b0, expr[0]})));
```

These code excerpts show that the generated HDL code from the MATLAB Function block represents the bit-true behavior of fixed point arithmetic expressions using high-level HDL operators. The HDL code is generated using HDL coding rules like high level `bitselect` and `partselect` replication operators and explicit sign extension and resize operators.

## Model State Using Persistent Variables

In the MATLAB Function block programming model, state-holding elements are represented as persistent variables. A variable that is declared `persistent` retains its value across function calls in software, and across sample time steps during simulation.

Please note that your MATLAB code *must* read the persistent variable before it is written if you want HDL Coder to infer a register in the HDL code. The code generator displays a warning message if your code does not follow this rule.

The following example shows the `unit_delay` block, which delays the input sample, `u`, by one simulation time step. `u` is a fixed-point operand of type `sfix6`. `u_d` is a persistent variable that holds the input sample.

```
function y = fcn(u)
persistent u_d;
if isempty(u_d)
 u_d = fi(-1, numerictype(u), fimath(u));
end
% return delayed input from last sample time hit
y = u_d;
% store the current input to be used later
u_d = u;
```

Because this code intends for `u_d` to infer a register during HDL code generation, `u_d` is read in the assignment statement, `y = u_d`, before it is written in `u_d = u`.

HDL Coder generates the following HDL code for the `unit_delay` block.

```
ENTITY Unit_Delay IS
 PORT (
 clk : IN std_logic;
 clk_enable : IN std_logic;
 reset : IN std_logic;
 u : IN std_logic_vector(15 DOWNTO 0);
 y : OUT std_logic_vector(15 DOWNTO 0));
END Unit_Delay;
```

```
ARCHITECTURE fsm_SFHDl OF Unit_Delay IS

BEGIN
 initialize_Unit_Delay : PROCESS (clk, reset)
 BEGIN
 IF reset = '1' THEN
 y <= std_logic_vector(to_signed(0, 16));
 ELSIF clk'EVENT AND clk = '1' THEN
 IF clk_enable = '1' THEN
 y <= u;
 END IF;
 END IF;
 END PROCESS initialize_Unit_Delay;
```

Initialization of persistent variables is moved into the master reset region in the initialization process.

Refer to the `Delays` subsystem in the `eml_hdl_design_patterns` library to see how vectors of persistent variables can be used to model integer delay, tap delay, and tap delay vector blocks. These design patterns are useful in implementing sequential algorithms that carry state between executions of the MATLAB Function block in a model.

## Creating Intellectual Property with the MATLAB Function Block

The MATLAB Function block helps you author intellectual property and create alternate implementations of part of an algorithm. By using MATLAB Function blocks in this way, you can guide the detailed operation of the HDL code generator even while writing high-level algorithms.

For example, the subsystem `Comparators` in the `eml_hdl_design_patterns` library includes several alternate algorithms for finding the minimum value of a vector. The `Comparators/eml_linear_min` block finds the minimum of the vector in a linear mode serially. The `Comparators/eml_tree_min` block compares the elements in a tree structure. The tree implementation can achieve a higher clock frequency by adding pipeline registers between the  $\log_2(N)$  stages. (See `eml_hdl_design_patterns/Filters` for an example.)

Now consider replacing the simple comparison operation in the `Comparators` blocks with an arithmetic operation (for example, addition, subtraction, or multiplication) where intermediate results must be quantized. Using `fimath` rounding settings, you can fine tune intermediate value computations before intermediate values feed into the next stage. You can use this technique for tuning the generated hardware or customizing your algorithm.

## Nontunable Parameter Arguments

You can declare a nontunable parameter for a MATLAB Function block by setting its **Scope** to **Parameter** in the Ports and Data Manager GUI, and clearing the **Tunable** option.

A nontunable parameter does not appear as a signal port on the block. Parameter arguments for MATLAB Function blocks take their values from parameters defined in a parent Simulink masked subsystem or from variables defined in the MATLAB base workspace, not from signals in the Simulink model.

## Modeling Control Logic and Simple Finite State Machines

MATLAB Function block control constructs such as `switch/case` and `if-elseif-else`, coupled with fixed point arithmetic operations let you model control logic quickly.

The FSMs/mealy\_fsm\_blk andFSMs/moore\_fsm\_blk blocks in the eml\_hdl\_design\_patterns library provide example implementations of Mealy and Moore finite state machines in the MATLAB Function block.

The following listing implements a Moore state machine.

```
function Z = moore_fsm(A)

persistent moore_state_reg;
if isempty(moore_state_reg)
 moore_state_reg = fi(0, 0, 2, 0);
end

S1 = 0;
S2 = 1;
S3 = 2;
S4 = 3;

switch uint8(moore_state_reg)

 case S1,
 Z = true;
 if (~A)
 moore_state_reg(1) = S1;
 else
 moore_state_reg(1) = S2;
 end
 case S2,
 Z = false;
 if (~A)
 moore_state_reg(1) = S1;
 else
 moore_state_reg(1) = S2;
 end
 case S3,
 Z = false;
 if (~A)
 moore_state_reg(1) = S2;
 else
 moore_state_reg(1) = S3;
 end
 case S4,
 Z = true;
 if (~A)
 moore_state_reg(1) = S1;
 else
 moore_state_reg(1) = S3;
 end
 otherwise,
 Z = false;
end
```

In this example, a persistent variable (`moore_state_reg`) models state variables. The output depends only on the state variables, thus modeling a Moore machine.

The FSMs/mealy\_fsm\_blk block in the eml\_hdl\_design\_patterns library implements a Mealy state machine. A Mealy state machine differs from a Moore state machine in that the outputs depend on inputs as well as state variables.

The MATLAB Function block can quickly model simple state machines and other control-based hardware algorithms (such as pattern matchers or synchronization-related controllers) using control statements and persistent variables.

For modeling more complex and hierarchical state machines with complicated temporal logic, use a Stateflow chart to model the state machine.

## Modeling Counters

To implement arithmetic and control logic algorithms in MATLAB Function blocks intended for HDL code generation, there are some simple HDL related coding requirements:

- The top level MATLAB Function block must be called once per time step.
- It must be possible to fully unroll program loops.
- Persistent variables with reset values and update logic must be used to hold values across simulation time steps.
- Quantized data variables must be used inside loops.

The following script shows how to model a synchronous up/down counter with preset values and control inputs. The example provides both master reset control of persistent state variables and local reset control using block inputs (e.g. `presetClear`). The `isempty` condition enters the initialization process under the control of a synchronous reset. The `presetClear` section is implemented in the output section in the generated HDL code.

Both the up and down case statements implementing the count loop require that the values of the counter are quantized after addition or subtraction. By default, the MATLAB Function block automatically propagates fixed-point settings specified for the block. In this script, however, fixed-point settings for intermediate quantities and constants are explicitly specified.

```
function [Q, QN] = up_down_ctr(upDown, presetClear, loadData, presetData)

% up down result
% 'result' synthesizes into sequential element

result_nt = numerictype(0,4,0);
result_fm = fimath('OverflowMode', 'saturate', 'RoundMode', 'floor');

initVal = fi(0, result_nt, result_fm);

persistent count;
if isempty(count)
 count = initVal;
end

if presetClear
 count = initVal;
elseif loadData
 count = loadData;
elseif upDown
 inc = count + fi(1, result_nt, result_fm);
 -- quantization of output
 count = fi(inc, result_nt, result_fm);
else
 dec = count - fi(1, result_nt, result_fm);
 -- quantization of output
 count = fi(dec, result_nt, result_fm);
end

Q = count;
QN = bitcmp(count);
```

## Modeling Hardware Elements

The following code example shows how to model shift registers in MATLAB Function block code by using the `bitsliceget` and `bitconcat` functions. This function implements a serial input and output shifters with a 32-bit fixed-point operand input. See the `Shift Registers/shift_reg_1by32` block in the `eml_hdl_design_patterns` library for more details.

```
function sr_out = fcn(shift, sr_in)
%shift register 1 by 32
```

```

persistent sr;
if isempty(sr)
 sr = fi(0, 0, 32, 0, 'fimath', fimath(sr_in));
end

% return sr[31]
sr_out = getmsb(sr);

if (shift)
 % sr_new[32:1] = sr[31:1] & sr_in
 sr = bitconcat(bitsliceget(sr, 31, 1), sr_in);
end

```

The following code example shows VHDL process code generated for the `shift_reg_1by32` block.

```

shift_reg_1by32 : PROCESS (shift, sr_in, sr)
BEGIN
 sr_next <= sr;
 -- MATLAB Function Function 'Subsystem/shift_reg_1by32': '<S2>:1'
 --shift register 1 by 32
 --<S2>:1:1
 -- return sr[31]
 --'<S2>:1:10'
 sr_out <= sr(31);

 IF shift /= '0' THEN
 --<S2>:1:12'
 -- sr_new[32:1] = sr[31:1] & sr_in
 --'<S2>:1:14'
 sr_next <= sr(30 DOWNTO 0) & sr_in;
 END IF;
END PROCESS shift_reg_1by32;

```

The `Shift Registers/shift_reg_1by64` block shows a 64 bit shifter. In this case, the shifter uses two fixed point words, to represent the operand, overcoming the 32-bit word length limitation for fixed-point integers.

Browse the `eml_hdl_design_patterns` model for other useful hardware elements that can be easily implemented using the MATLAB Function block.

## Decimal to Binary Conversion

You can perform conversions from an integer type to generate a bit vector output and vice-versa. For an example model that shows how to perform this conversion, open the model `hdlcoder_int2bits_bits2int`.

```
open_system('hdlcoder_int2bits_bits2int')
```

The model uses the MATLAB Function blocks that are implemented in the `eml_hdl_design_patterns` library under the `Word Twiddlers` library.

## See Also

[“Check for MATLAB Function block settings” on page 38-19](#)

## More About

- “Design Guidelines for the MATLAB Function Block” on page 29-29
- “Code Generation from a MATLAB Function Block” on page 29-5
- “Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37

- “Generate DUT Ports for Tunable Parameters” on page 10-17

# Design Guidelines for the MATLAB Function Block

| In this section...                                                                   |
|--------------------------------------------------------------------------------------|
| "Use Compiled External Functions With MATLAB Function Blocks" on page 29-29          |
| "Build the MATLAB Function Block Code First" on page 29-29                           |
| "Use the <code>hdlfimath</code> Utility for Optimized FIMATH Settings" on page 29-29 |
| "Use Optimal Fixed-Point Option Settings" on page 29-30                              |
| "Set the Output Data Type of MATLAB Function Blocks Explicitly" on page 29-30        |
| "Using Tunable Parameters" on page 29-30                                             |
| "Run HDL Model Check for MATLAB Function Blocks" on page 29-30                       |
| "Use MATLAB Datapath Architecture for Enhanced HDL Optimizations" on page 29-30      |

## Use Compiled External Functions With MATLAB Function Blocks

The HDL Coder software supports HDL code generation from MATLAB Function blocks that include compiled external functions. This feature enables you to write reusable MATLAB code and call it from multiple MATLAB Function blocks.

Such functions must be defined in files that are on the MATLAB Function block path. Use the `%#codegen` compilation directive to indicate that the MATLAB code is suitable for code generation. See "Function Definition" for information on how to create, compile, and invoke external functions.

## Build the MATLAB Function Block Code First

Before generating HDL code for a subsystem containing a MATLAB Function block, build the MATLAB Function block code to check for errors. To build the code, click the **Build Model** button in the function editor.

## Use the `hdlfimath` Utility for Optimized FIMATH Settings

The `hdlfimath` function is a utility that defines a FIMATH specification that is optimized for HDL code generation.

The following listing shows the `fimath` setting defined by `hdlfimath`.

```
hdlfm = fimath(
 'RoundMode', 'floor',...
 'OverflowMode', 'wrap',...
 'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
 'SumMode', 'FullPrecision', 'SumWordLength', 32,...
 'CastBeforeSum', true);
```

The HDL division operator does not support 'floor' rounding mode. Use 'round' mode or change the signed integer division operations to unsigned integer division.

When the `fimath` `OverflowMode` property of the `fimath` specification is set to 'Saturate', HDL code generation is disallowed for the following cases:

- `SumMode` is set to 'SpecifyPrecision'
- `ProductMode` is set to 'SpecifyPrecision'

## Use Optimal Fixed-Point Option Settings

Use the default (Fixed-point) setting for the **Treat these inherited signal types as fi objects** option.

Clear the **Saturate on integer overflow** setting for the MATLAB Function block.

## Set the Output Data Type of MATLAB Function Blocks Explicitly

By setting the output data type of a MATLAB Function block explicitly, you enable optimizations for RAM mapping and pipelining. Avoid inheriting the output data type for a MATLAB Function block for which you want to enable optimizations.

## Using Tunable Parameters

HDL Coder supports both tunable and non-tunable parameters with the following data types:

- Scalar
- Vector
- Complex
- Structure
- Enumeration

When using tunable parameters with the MATLAB Function block:

- The tunable parameter should be a `Simulink.Parameter` object with the `StorageClass` set to `ExportedGlobal`.

```
x = Simulink.Parameter
x.Value = 1
x.CoderInfo.StorageClass = 'ExportedGlobal'
```

- In the Ports and Data Manager dialog box, select the **tunable** check box.

For details, see “Generate DUT Ports for Tunable Parameters” on page 10-17.

## Run HDL Model Check for MATLAB Function Blocks

When your design contains MATLAB Function blocks, before you generate HDL code, you can run the check “Check for MATLAB Function block settings” on page 38-19 in the HDL Code Advisor. This check verifies whether you use the recommended MATLAB Function blocks for HDL code generation. The settings include whether `fimath` settings are defined as per `hdlfimath` and **Saturate on integer overflow** check box is cleared.

See also “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

## Use MATLAB Datapath Architecture for Enhanced HDL Optimizations

In the HDL Block Properties dialog box for the MATLAB Function blocks, you can specify whether to use **MATLAB Function** or **MATLAB Datapath** as the HDL architecture. Floating-point models use the **MATLAB Datapath** architecture even if you specify the HDL architecture as **MATLAB Function** on the block. Fixed-point models use the **MATLAB Function** architecture by default.

To perform various speed and area optimizations such as sharing and distributed pipelining inside the MATLAB Function block and across the MATLAB Function block boundary with other Simulink blocks, use the **MATLAB Datapath** architecture. When you use this architecture, the code generator treats the MATLAB Function block like a regular Subsystem block. This capability enables you to optimize the algorithm inside the MATLAB Function block and across the MATLAB Function block with other blocks in your model.

See “[HDL Optimizations Across MATLAB Function Block Boundary Using MATLAB Datapath Architecture](#)” on page 24-146.

## See Also

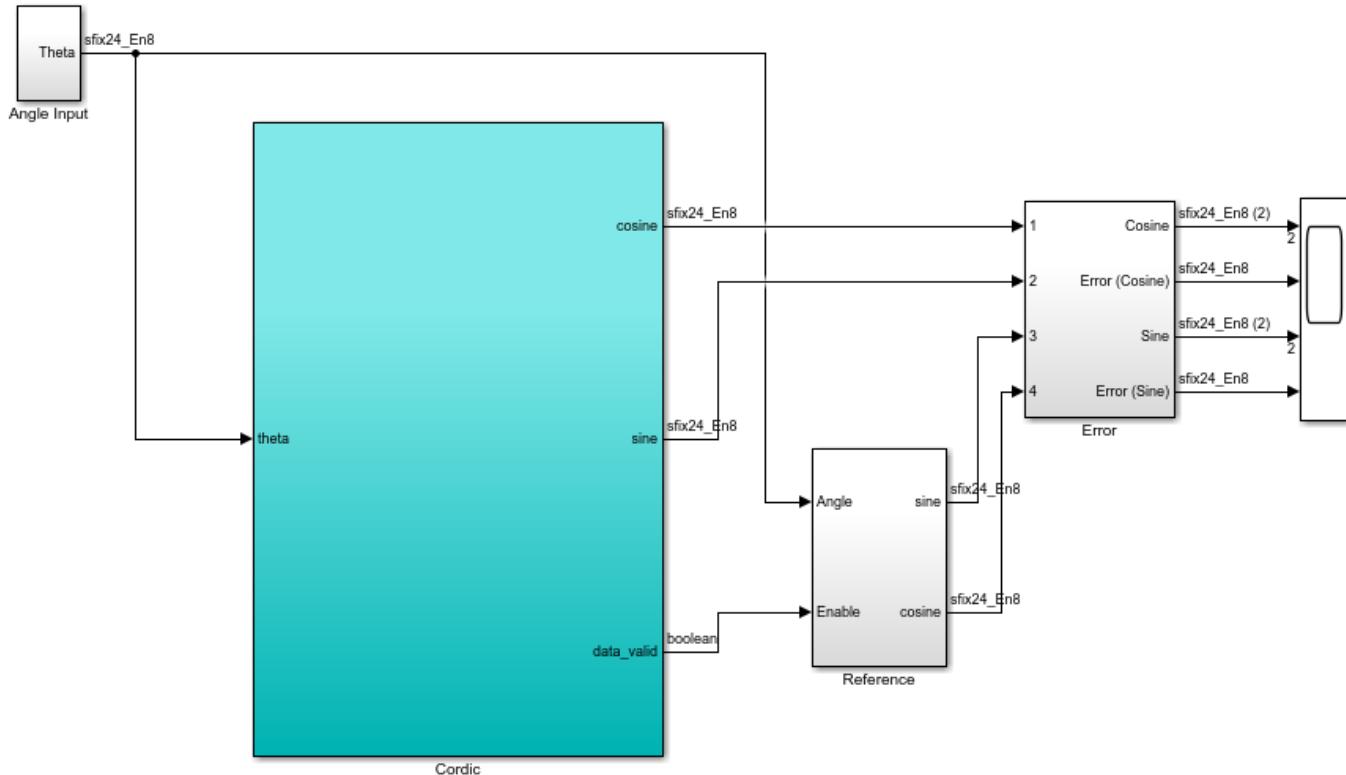
[“Check for MATLAB Function block settings” on page 38-19](#)

## More About

- [“Code Generation from a MATLAB Function Block” on page 29-5](#)
- [“MATLAB Function Block Design Patterns for HDL” on page 29-19](#)
- [“Distributed Pipeline Insertion for MATLAB Function Blocks” on page 29-37](#)
- [“Generate DUT Ports for Tunable Parameters” on page 10-17](#)

## CORDIC Algorithm Using the MATLAB® Function Block

This example shows how to use HDL Coder™ to check, generate and verify HDL for a fixed-point CORDIC model implementing sin and cos trigonometric functions using the MATLAB Function Block.



Double click here for information on this example

[Launch HDL Dialog](#)

[Run Example](#)

Copyright 2007-2012 The MathWorks, Inc.

# Hardware Design Patterns Using the MATLAB Function Block

This example shows how to effectively use the MATLAB Function block to model commonly used hardware algorithms using HDL Coder™. An HDL design patterns library is used to show the features of MATLAB Coder supported by HDL Coder.

## Quick Introduction

The MATLAB Function block supports simulation and code generation for a restricted subset of MATLAB® language. It provides a mechanism for algorithm implementation in Simulink® and Stateflow®. To get more information on how to use this block in Simulink type:

```
>> docsearch('About Code Generation from MATLAB Algorithms')
```

## Subset of Features Supported by HDL Coder

HDL Coder supports a powerful subset of MATLAB Function block features well suited for HDL implementations of various DSP and telecommunication applications such as sequence detectors, pattern generators, encoders, decoders etc., The following list briefly shows the features supported by MATLAB Function block in HDL Coder:

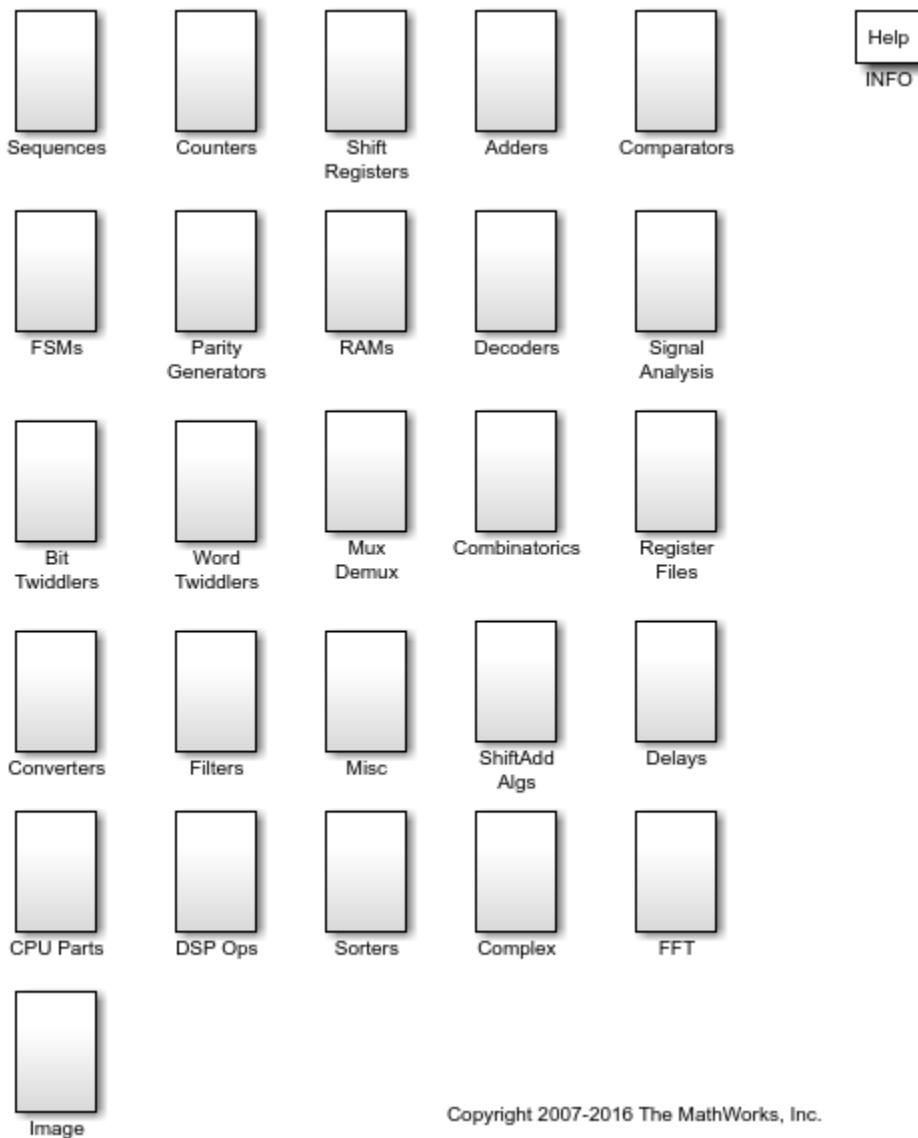
- Various Numeric Classes (int,uint,logical,single,double)
- Fixed-point arithmetic using 'fi' object
- Arithmetic, Logical, Relational and Bitwise operator support
- Full MATLAB expression support using above operators
- 1-D and 2-D Matrix Operations
- Matrix Subscripting
- Control flow using if, switch, for statements
- Sub functions
- Persistent variables to model state
- Fixed point and integer MATLAB library functions

## Modeling Hardware Algorithms

The MATLAB Function block provides a mechanism to model at a high level of abstraction with a concise and textual way of expressing behavior of a hardware algorithm. HDL Coder provides an easy path to implementation from such an algorithmic level representation. The following sections show how to use this block effectively in HDL Coder.

```
% To illustrate the use of the above features in hardware modeling a
% sample patterns library is created that shows how to model common
% HDL problems. To open this sample library model please type
% at the command prompt
```

```
open_system('eml_hdl_design_patterns')
```



### Using Blocks in this Library

To build some sample models using blocks in this model and observe the generated code please follow these steps:

- Create a new model.
- Copy the block of interest from the `eml_hdl_design_patterns` library to this model.
- Place it in a subsystem or a device under test.
- Run '`hdlsetup`' command.
- Run '`makehdl`' to generate code for the block
- Build valid test bench around it using Simulink sources and sinks.
- Run '`makehdltb`' command to generate testbench.

- Use the Modelsim '.do' script file to simulate the generated code.

see tutorial example model 'eml\_hdl\_incremener' for more details or type

```
>> docsearch('Tutorial Example: Incremener')
```

### **Modeling Arithmetic**

This section talks about some of the design considerations that help generate efficient HDL from the MATLAB Function block. Please note that for modeling arbitrary length arithmetic operations using signed and unsigned logic vectors use of fi object is recommended. The fi objects provide a powerful mechanism to model fixed point arithmetic. Here are a couple of things to note when using fi objects in the MATLAB Function block.

The fi function helps you to define a fixed-point object with customized 'numerictype' (that defines sign, wordlength, fractionlength) and 'fimath' (that defines rounding and saturation modes)

For HDL code generation, we recommend the fimath shown below. (with 'Floor', 'Wrap' and 'FullPrecision' modes) as shown below.

However when modeling algorithms requiring more complicated rounding and saturation logic you may use other fimath modes for rounding (floor,ceil,fix,nearest) and overflow (wrap,saturate)

```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
'SumMode', 'FullPrecision', 'SumWordLength', 32);
```

Examples: The following examples show the affect of fimath properties on the generated code.

```
open_system('eml_hdl_design_patterns/Adders/add_with_carry')
open_system('eml_hdl_design_patterns/Misc/eml_expr')
```

### **Modeling State Using Persistent Variables**

To model a complex control logic, the ability to model registers is a basic requirement. In the MATLAB Coder programming model, the state-holding elements are represented as persistent variables. A variable declared persistent retains its value across function calls in software, and across steps of Simulink sample times. State holding elements in hardware like registers and flip-flops also exhibit similar behavior. The following examples show how the values of persistent variables can be changed using global and local reset conditions.

Examples: open\_system('eml\_hdl\_design\_patterns/Delays') open\_system('eml\_hdl\_design\_patterns/Delays/unit delay') open\_system('eml\_hdl\_design\_patterns/Delays/integer delay') open\_system('eml\_hdl\_design\_patterns/Delays/tap delay') open\_system('eml\_hdl\_design\_patterns/Delays/tap delay vector')

### **Modeling Counters and FSMs**

The MATLAB Coder control-constructs such as switch/case and if-elseif-else, coupled with delay elements and fixed point arithmetic operations, let you model control logic. The examples in FSMs show how to use the switch-case and the if-elseif-end control statements. The counter show to model state and how to quantize data elements within loops.

Examples: open\_system('eml\_hdl\_design\_patterns/FSMs') open\_system('eml\_hdl\_design\_patterns/Counters')

## Modeling Bitwise Operations

The MATLAB Function block supports a variety of bitwise operations useful for hardware bit manipulation operations like bit concatenation, bit packing and unpacking, conversions between integer and bits, and pn-sequence generation and bit-scramblers.

Here is a quick list of bitwise functions that are supported by HDL Coder:

- bitget, bitsliceget, bitconcat, bitset, bitcmp
- bitand, bitor, bitxor
- bitandreduce, bitorreduce, bitxorreduce
- bitshift, bitsll, bitsrl, bitsra, bitrol, bitror

When modeling pure logic and no math operations in the MATLAB Function block the following settings on input operands are recommended.

- Prefer unsigned to signed input operands
- Use non saturating fimath options to generate less hardware.
- Prefer 'OverflowMode' to be 'wrap' and 'RoundMode' to be 'floor'

Examples: `open_system('eml_hdl_design_patterns/Bit Twiddlers/hdl_bit_ops')`  
`open_system('eml_hdl_design_patterns/Bit Twiddlers/signal_distance')`  
`open_system('eml_hdl_design_patterns/Word Twiddlers/nibble_swap_with_slice_concat')`

For an example that shows how to perform a conversion between integer and bits, open the model `hdlcoder_int2bits_bits2int`.

```
open_system('hdlcoder_int2bits_bits2int')
```

This model uses a MATLAB Function block that is implemented in the `Word Twiddlers` library.

```
open_system('eml_hdl_design_patterns/Word Twiddlers/Bits2Int')
open_system('eml_hdl_design_patterns/Word Twiddlers/Int2Bits')
open_system('eml_hdl_design_patterns/Word Twiddlers/Integer to Bits')
open_system('eml_hdl_design_patterns/Word Twiddlers/Bits to Integer')
```

## Modeling Hardware Elements

The MATLAB Function block can be used to model various hardware elements like barrel shifters, rotators, carry save adders using simple and concise MATLAB scripts.

Examples: `open_system('eml_hdl_design_patterns/Shift Registers/shift_reg_universal')`

## Conclusion

This example illustrates various opportunities that open up for hardware modeling through the use of the MATLAB Function block. A set of patterns to solve common hardware modeling problems using the MATLAB Function block are discussed. Please read the doc for more information on this library .

```
>> docsearch('The eml_hdl_design_patterns Library')
```

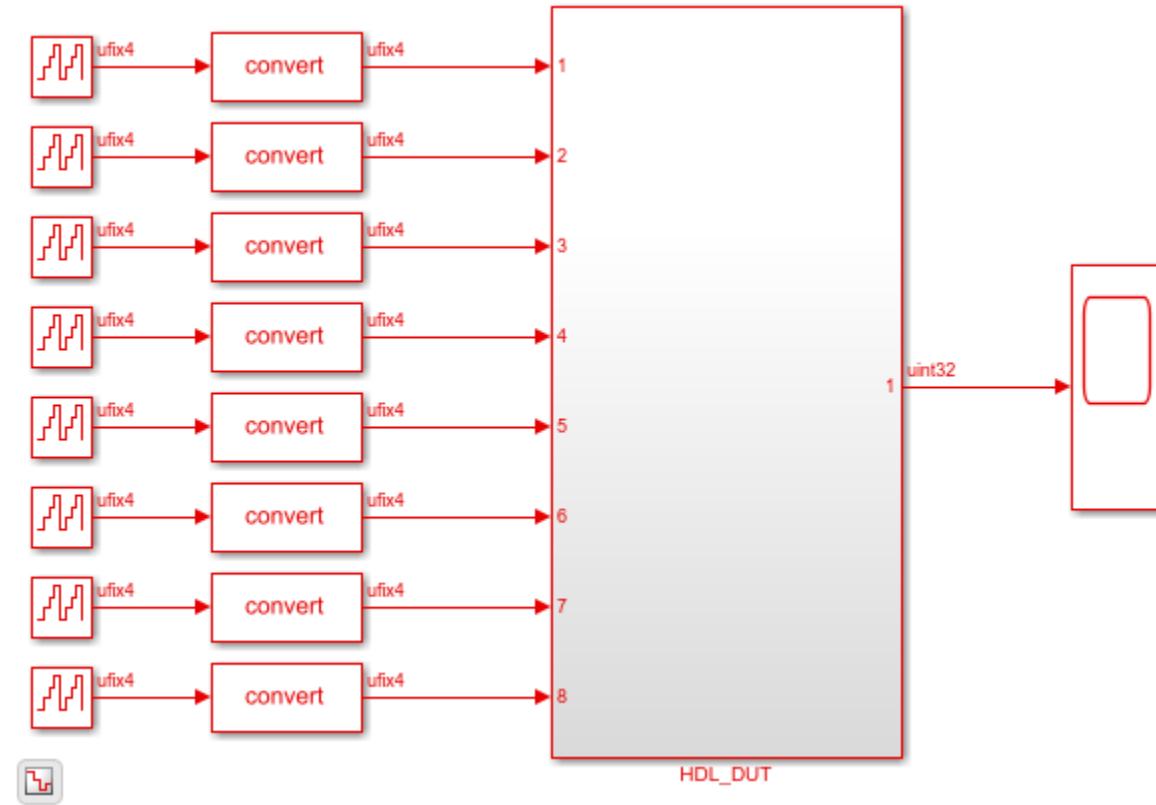
# Distributed Pipeline Insertion for MATLAB Function Blocks

This example shows how to optimize the generated HDL code for MATLAB Function blocks by using the distributed pipelining optimization. Distributed pipelining is an HDL Coder™ optimization that improves the generated HDL code from MATLAB Function blocks, Simulink® models, or Stateflow® charts. By using distributed pipelining, your design achieves higher clock rates on the FPGA device.

## Multiplier Chain Model

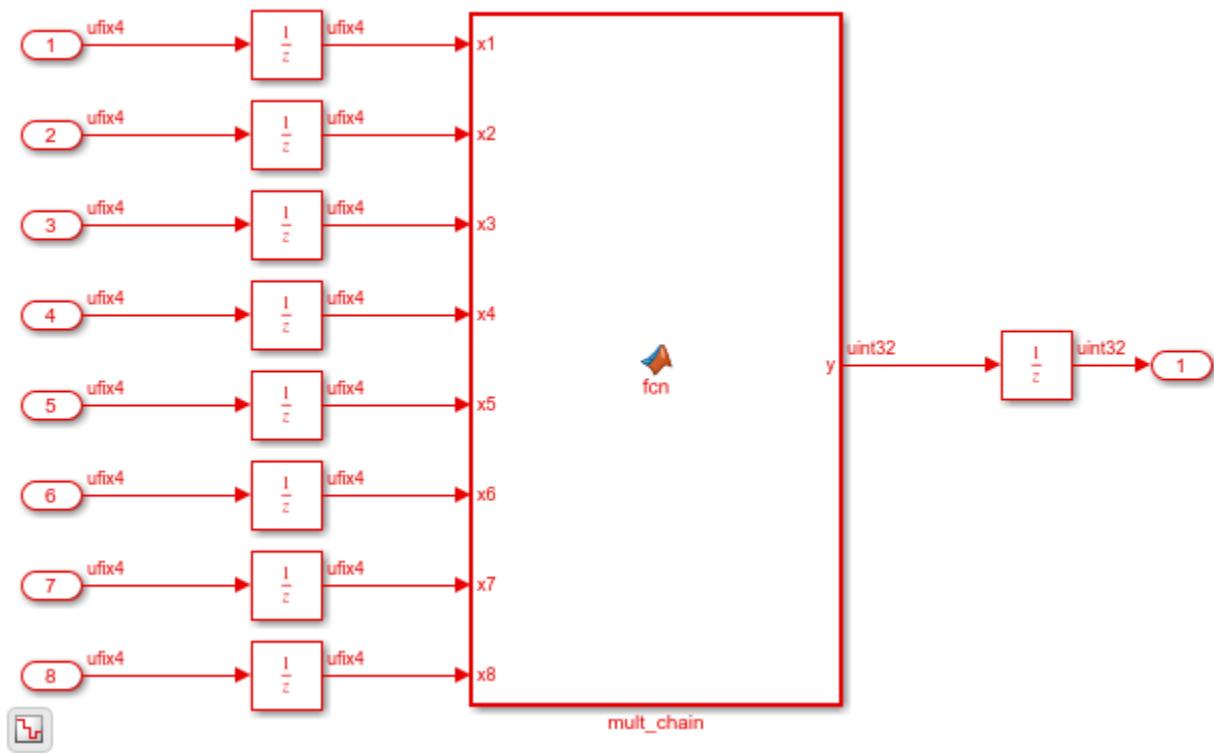
This example shows how to distribute pipeline registers in a simple model that chains five multiplications.

```
open_system('hdlcoder_distpipe_multiplier_chain')
set_param('hdlcoder_distpipe_multiplier_chain','SimulationCommand','Update')
```



The **HDL\_DUT** Subsystem is the DUT for which you want to generate HDL code. The subsystem drives a MATLAB Function block **mult\_chain**.

```
open_system('hdlcoder_distpipe_multiplier_chain/HDL_DUT')
```



To see the chain of multiplications, open the MATLAB Function block.

```
open_system('hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain')
```

```
function y = fcn(x1,x2,x3,x4,x5,x6,x7,x8)
% A chained multiplication:
% y = (x1*x2)*(x3*x4)*(x5*x6)*(x7*x8)

y1 = x1 * x2;
y2 = x3 * x4;
y3 = x5 * x6;
y4 = x7 * x8;

y5 = y1 * y2;
y6 = y3 * y4;

y = y5 * y6;
```

### Apply Distributed Pipelining Optimization

- Specify generation of two pipeline stages for the MATLAB Function block.

```
ml_subsys = 'hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain';
hdlset_param(ml_subsys, 'OutputPipeline', 2)
```

2. Specify the MATLAB Datapath architecture. This architecture treats the MATLAB Function block like a regular Subsystem. You can then apply various optimizations across the MATLAB Function blocks with other blocks in your Simulink® model.

```
hdlset_param(ml_subsys, 'architecture', 'MATLAB Datapath');
```

3. Enable the distributed pipelining optimization on the block. To see the results of the optimization, enable generation of the Optimization Report. To apply the optimization across hierarchies in your model, enable hierarchical distributed pipelining on the model and distributed pipelining on all subsystems.

```
hdlset_param('hdlcoder_distpipe_multiplier_chain', ...
 'HierarchicalDistPipe', 'on', 'OptimizationReport', 'on')
hdlset_param('hdlcoder_distpipe_multiplier_chain/HDL_DUT', 'DistributedPipelining', 'on');
hdlset_param(ml_subsys, 'DistributedPipelining', 'on');
```

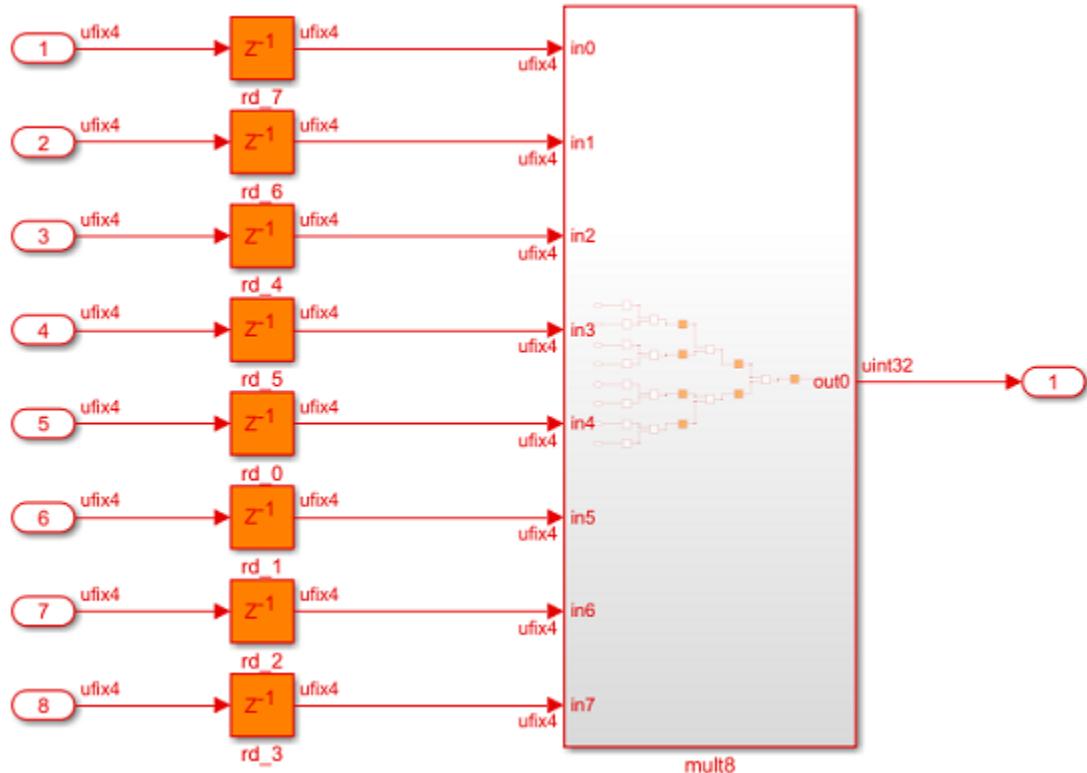
4. Generate HDL Code for the HDL\_DUT subsystem.

```
makehdl('hdlcoder_distpipe_multiplier_chain/HDL_DUT/mult_chain')
```

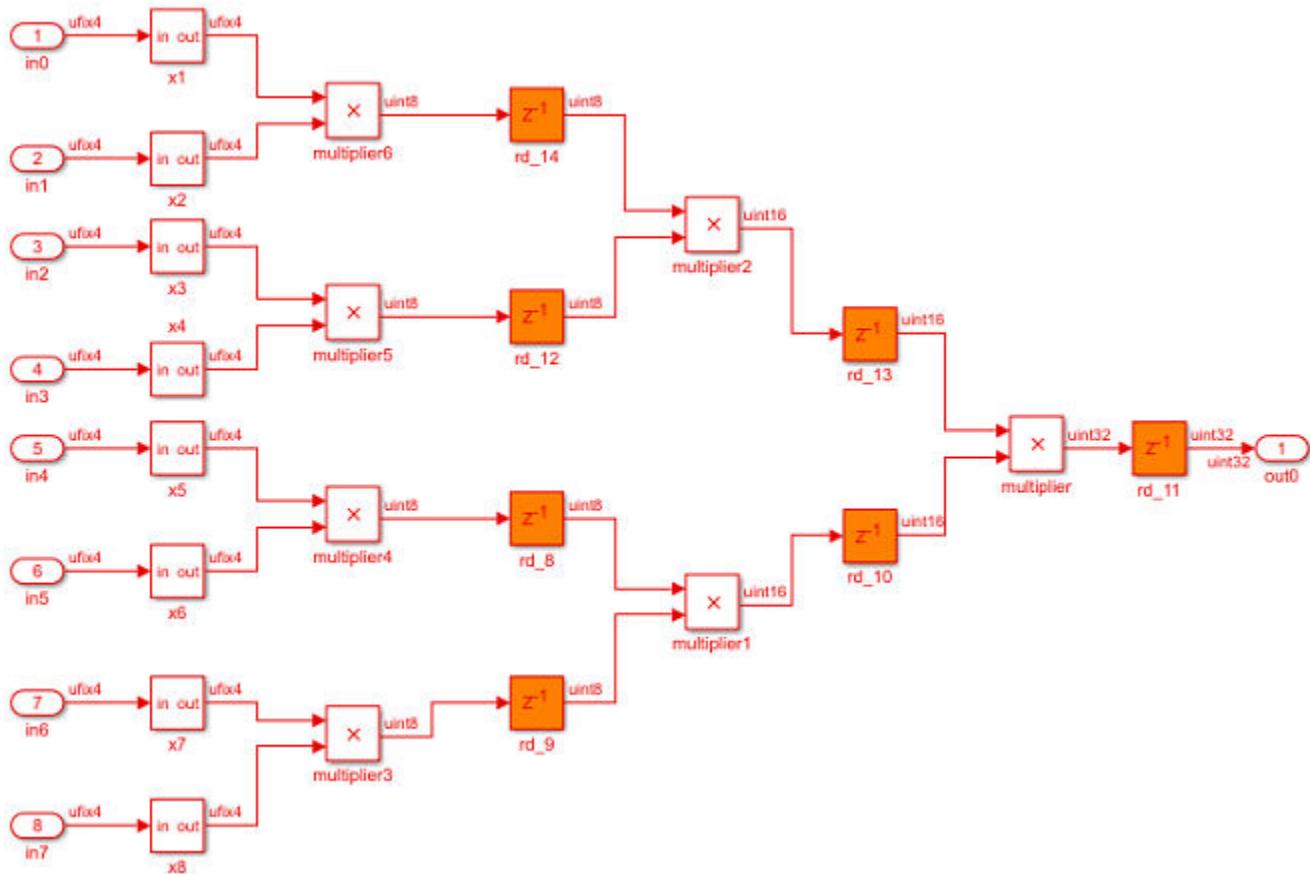
By default, HDL Coder generates VHDL code in the `hdlsrc` folder.

## Analyze Results of Optimization

In the Distributed Pipelining report, you see that the code generator moved the pipeline registers. To see the effects of the optimization, open the generated model `gm_hdlcoder_distpipe_multiplier_chain` and navigate to the HDL\_DUT Subsystem.



The MATLAB Datapath architecture creates a Subsystem in place of the MATLAB Function block. The optimization can then distribute the pipeline registers and the unit delay that you added inside the Subsystem to optimize the multiplier chain and improve timing. Open the `mult_chain` Subsystem.



## See Also

"Check for MATLAB Function block settings" on page 38-19

## More About

- "Design Guidelines for the MATLAB Function Block" on page 29-29
- "Code Generation from a MATLAB Function Block" on page 29-5
- "MATLAB Function Block Design Patterns for HDL" on page 29-19
- "Generate DUT Ports for Tunable Parameters" on page 10-17

# Generating Scripts for HDL Simulators and Synthesis Tools

---

- “Generate Scripts for Compilation, Simulation, and Synthesis” on page 30-2
- “Structure of Generated Script Files” on page 30-3
- “Properties for Controlling Script Generation” on page 30-4
- “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7
- “Add Synthesis Attributes” on page 30-14
- “Configure Synthesis Project Using Tcl Script” on page 30-15

## Generate Scripts for Compilation, Simulation, and Synthesis

You can enable or disable script generation and customize the names and content of generated script files using either of the following methods:

- Use the `makehdl` or `makehdltb` functions, and pass in property name/property value arguments, as described in “Properties for Controlling Script Generation” on page 30-4.
- Set script generation options in the **HDL Code Generation > EDA Tool Scripts** pane of the Configuration Parameters dialog box, as described in “Configure Compilation, Simulation, Synthesis, and Lint Scripts” on page 30-7.

## Structure of Generated Script Files

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

- 1 An initialization (`Init`) phase. The `Init` phase performs the required setup actions, such as creating a design library or a project file. Some arguments to the `Init` phase are implicit, for example, the top-level entity or module name.
- 2 A command-per-file phase (`Cmd`). This phase of the script is called iteratively, once per generated HDL file or once per signal. On each call, a different file or signal name is passed in.
- 3 A termination phase (`Term`). This is the final execution phase of the script. One application of this phase is to execute a simulation of HDL code that was compiled in the `Cmd` phase. The `Term` phase does not take arguments.

The HDL Coder software generates scripts by passing format strings to the `fprintf` function. Using the GUI options (or `makehdl` and `makehdltb` properties) summarized in the following sections, you can pass in customized format names to the script generator. Some of these format names take arguments, such as the top-level entity or module name, or the names of the VHDL or Verilog files in the design.

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

## Properties for Controlling Script Generation

This section describes how to set properties in the `makehdl` or `makehdltb` functions to enable or disable script generation and customize the names and content of generated script files.

### Enabling and Disabling Script Generation

The `EDAScriptGeneration` property controls the generation of script files. By default, `EDAScriptGeneration` is set on. To disable script generation, set `EDAScriptGeneration` to `off`, as in the following example.

```
makehdl('sfir_fixed/symmetric_fir','EDAScriptGeneration','off')
```

### Customizing Script Names

When you generate HDL code, HDL Coder appends a postfix string to the model or subsystem name `system` in the generated script name.

When you generate test bench code, HDL Coder appends a postfix string to the test bench name `testbench_tb`.

The postfix string depends on the type of script (compilation, simulation, or synthesis) being generated. The default postfix strings are shown in the following table. For each type of script, you can define your own postfix using the associated property.

| Script Type | Property                           | Default Value                                                                      |
|-------------|------------------------------------|------------------------------------------------------------------------------------|
| Compilation | <code>HDLCompileFilePostfix</code> | <code>_compile.do</code>                                                           |
| Simulation  | <code>HDLSimFilePostfix</code>     | <code>_sim.do</code>                                                               |
| Synthesis   | <code>HDLSynthFilePostfix</code>   | Depends on the selected synthesis tool. See “Choose synthesis tool” on page 20-11. |

The following command generates VHDL code for the subsystem `system`, specifying a custom postfix for the compilation script. The name of the generated compilation script will be `system_test_compilation.do`.

```
makehdl('mymodel/system', 'HDLCompileFilePostfix', '_test_compilation.do')
```

### Customizing Script Code

Using the property name/property value pairs summarized in the following table, you can pass in customized format names as character vectors to `makehdl` or `makehdltb`. The properties are named according to the following conventions:

- Properties that apply to the initialization (`Init`) phase are identified by the `Init` character vector in the property name.
- Properties that apply to the command-per-file phase (`Cmd`) are identified by the `Cmd` character vector in the property name.
- Properties that apply to the termination (`Term`) phase are identified by the `Term` character vector in the property name.

| Property Name and Default                                                                    | Description                                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Name: HDLCompileInit<br>Default: 'vlib %s\n'                                                 | Format name passed to <code>fprintf</code> to write the <code>Init</code> section of the compilation script. The implicit argument is the contents of the <code>VHDLLibraryName</code> property, which defaults to 'work'. You can override the default <code>Init</code> string ('vlib work\n') by changing the value of <code>VHDLLibraryName</code> .          |
| Name: HDLCompileVHDLCmd<br>Default: 'vcom %s %s\n'                                           | Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for VHDL files. The two implicit arguments are the contents of the <code>SimulatorFlags</code> property and the file name of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default).                     |
| Name: HDLCompileVerilogCmd<br>Default: 'vlog %s %s\n'                                        | Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the compilation script for Verilog files. The two implicit arguments are the contents of the <code>SimulatorFlags</code> property and the file name of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default).                  |
| Name: HDLCompileTerm<br>Default: ''                                                          | Format name passed to <code>fprintf</code> to write the termination portion of the compilation script.                                                                                                                                                                                                                                                            |
| Name: HDLSimInit<br>Default:<br><pre>[ 'onbreak resume\n',...<br/>'onerror resume\n' ]</pre> | Format name passed to <code>fprintf</code> to write the initialization section of the simulation script.                                                                                                                                                                                                                                                          |
| Name: HDLSimCmd<br>Default: 'vsim -voptargs+=acc %s.%s\n'                                    | Format name passed to <code>fprintf</code> to write the simulation command.<br><br>If your target language is VHDL, the first implicit argument is the value of the <code>VHDLLibraryName</code> property. If your target language is Verilog, the first implicit argument is 'work'.<br><br>The second implicit argument is the top-level module or entity name. |
| Name: HDLSimViewWaveCmd<br>Default: 'add wave sim:%s\n'                                      | Format name passed to <code>fprintf</code> to write the simulation script waveform viewing command. The implicit argument adds the signal paths for the DUT top-level input, output, and output reference signals.                                                                                                                                                |
| Name: HDLSimTerm<br>Default: 'run -all\n'                                                    | Format name passed to <code>fprintf</code> to write the <code>Term</code> portion of the simulation script. The string is a synthesis project creation command. The content of the string is specific to the selected synthesis tool. See "Choose synthesis tool" on page 20-11.                                                                                  |
| Name: HDLSynthInit                                                                           | Format name passed to <code>fprintf</code> to write the <code>Init</code> section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See "Choose synthesis tool" on page 20-11.                                                                                                                                  |

| Property Name and Default | Description                                                                                                                                                                                                                      |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Name: HDLSynthCmd         | Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See “Choose synthesis tool” on page 20-11.  |
| Name: HDLSynthTerm        | Format name passed to <code>fprintf</code> to write the <code>Term</code> section of the synthesis script. The content of the format name is specific to the selected synthesis tool. See “Choose synthesis tool” on page 20-11. |

## Examples

The following example specifies a custom VHDL library name for the Mentor Graphics ModelSim compilation script for code generated from the subsystem, `system`.

```
makehdl(system, 'VHDLlibraryName', 'mydesignlib')
```

The resultant script, `system_compile.do`, is:

```
vlib mydesignlib
vcom system.vhd
```

The following example specifies that HDL Coder generate a Xilinx ISE synthesis file for the subsystem `sfir_fixed/symmetric_fir`.

```
makehdl('sfir_fixed/symmetric_fir','HDLSynthTool', 'ISE')
```

The following listing shows the resultant script, `symmetric_fir_ise.tcl`.

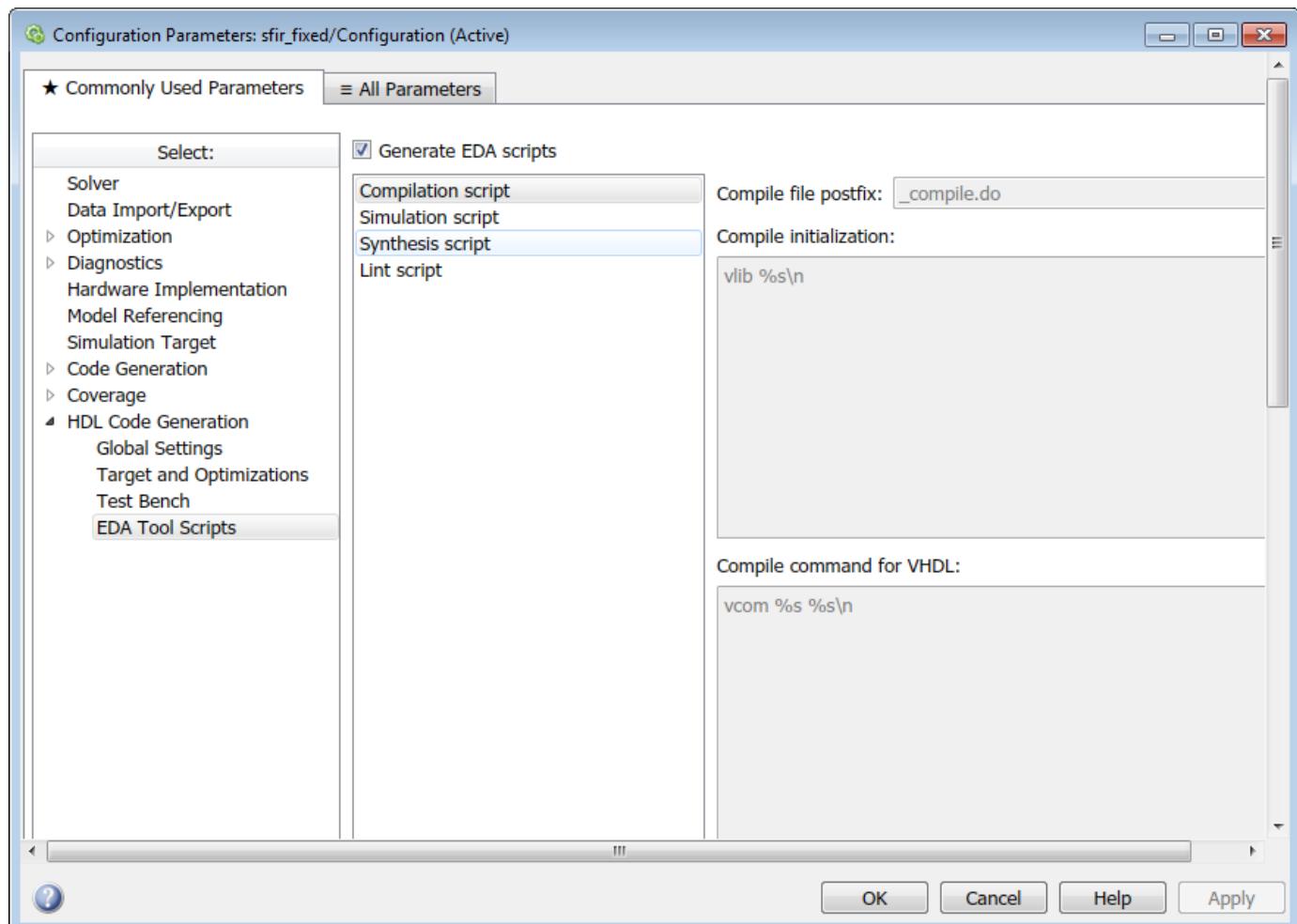
```
set src_dir "./hdlsrc"
set prj_dir "synprj"
file mkdir ../$prj_dir
cd ../$prj_dir
project new symmetric_fir.ise
xfile add ../$src_dir/symmetric_fir.vhd
project set family Virtex4
project set device xc4vsx35
project set package ff668
project set speed -10
process run "Synthesize - XST"
```

## Configure Compilation, Simulation, Synthesis, and Lint Scripts

You set options that configure script file generation on the **EDA Tool Scripts** pane. These options correspond to the properties described in “Properties for Controlling Script Generation” on page 30-4.

To view and set **EDA Tool Scripts** options:

- 1 Open the Configuration Parameters dialog box.
- 2 Select the **HDL Code Generation > EDA Tool Scripts** pane.



- 3 The **Generate EDA scripts** option controls the generation of script files. By default, this option is selected.

If you want to disable script generation, clear this check box and click **Apply**.

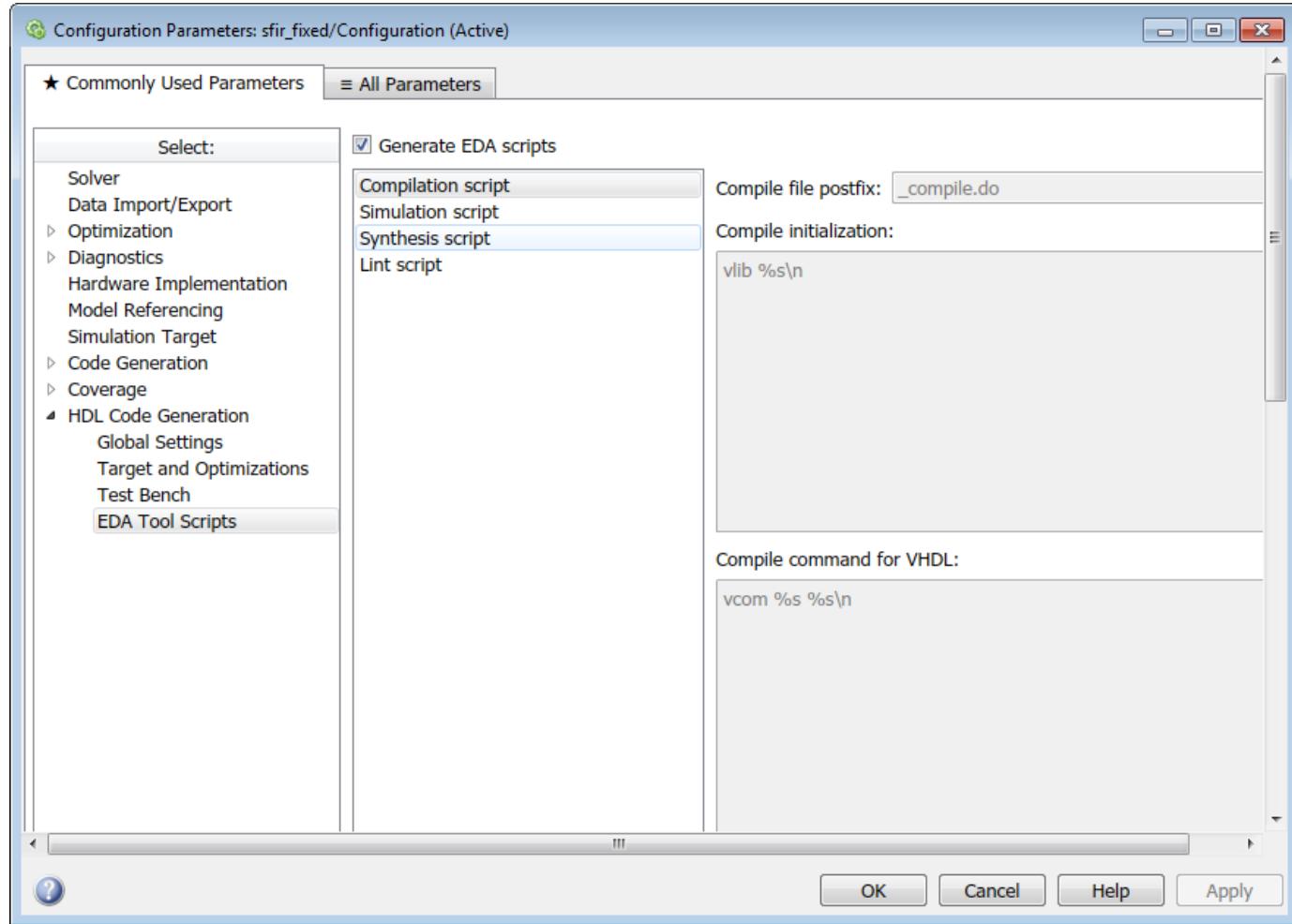
- 4 The list on the left of the **EDA Tool Scripts** pane lets you select from several categories of options. Select a category and set the options as desired. The categories are:

- **Compilation script**: Options related to customizing scripts for compilation of generated VHDL or Verilog code. See “Compilation Script Options” on page 30-8 for further information.

- **Simulation script:** Options related to customizing scripts for HDL simulators. See “Simulation Script Options” on page 30-9 for further information.
- **Synthesis script:** Options related to customizing scripts for synthesis tools. See “Synthesis Script Options” on page 30-11 for further information.

## Compilation Script Options

The following figure shows the **Compilation script** pane, with options set to their default values.



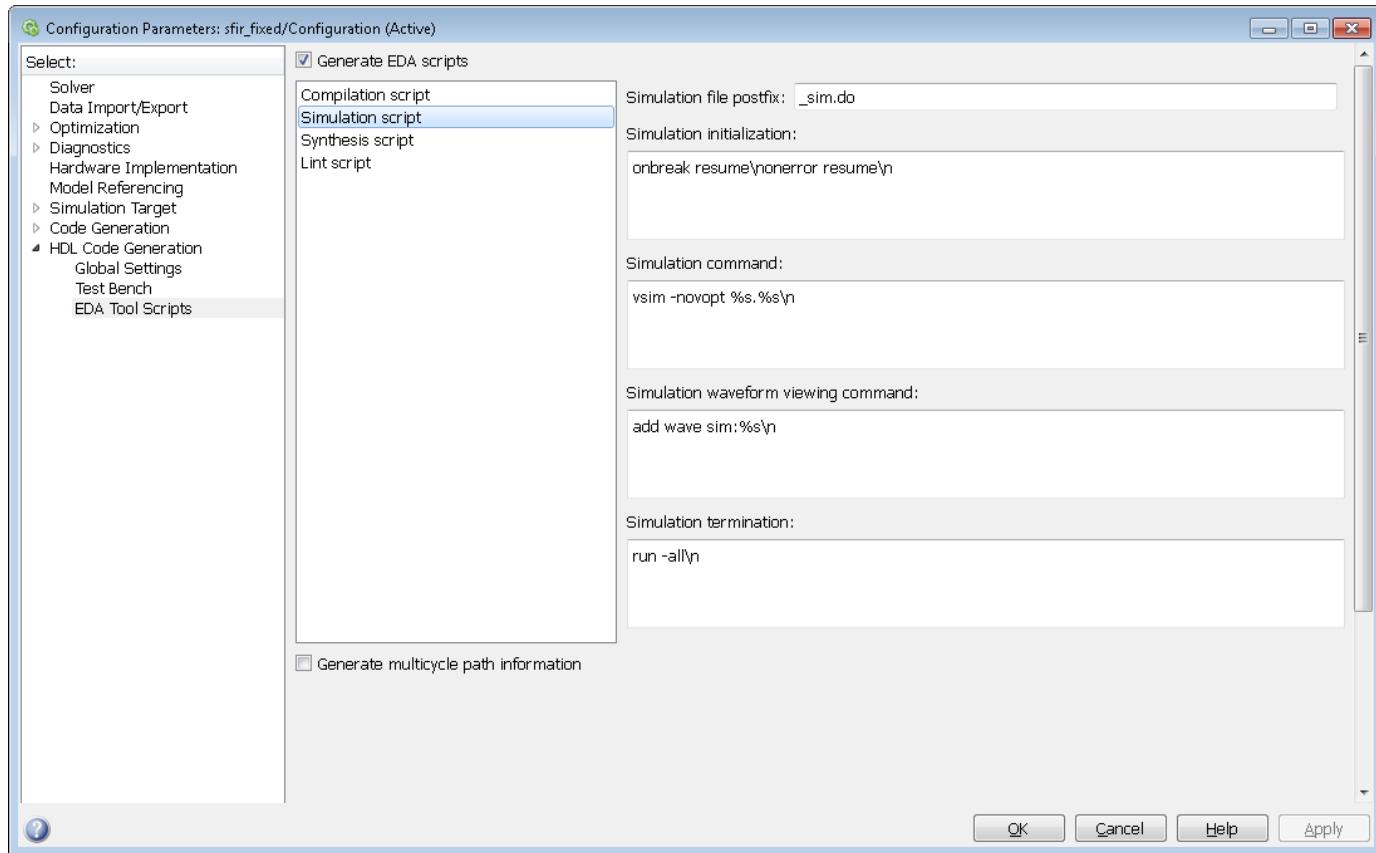
The following table summarizes the **Compilation script** options.

| Option and Default | Description                                                                       |
|--------------------|-----------------------------------------------------------------------------------|
| '_compile.do'      | Postfix appended to the DUT name or test bench name to form the script file name. |

| Option and Default                                                  | Description                                                                                                                                                                                                                                                                                                                   |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Name: Compile initialization</b><br>Default: 'vlib %s\n'         | Format name passed to <code>fprintf</code> to write the Init section of the compilation script. The argument is the contents of the <code>VHDLLibNameName</code> property, which defaults to 'work'. You can override the default Init 'vlib work\n' by changing the value of <code>VHDLLibNameName</code> .                  |
| <b>Name: Compile command for VHDL</b><br>Default: 'vcom %s %s\n'    | Format name passed to <code>fprintf</code> to write the Cmd section of the compilation script for VHDL files. The two arguments are the contents of the <code>SimulatorFlags</code> property option and the filename of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default). |
| <b>Name: Compile command for Verilog</b><br>Default: 'vlog %s %s\n' | Format name passed to <code>fprintf</code> to write the Cmd section of the compilation script for Verilog files. The two arguments are the contents of the <code>SimulatorFlags</code> property and the filename of the current entity or module. To omit the flags, set <code>SimulatorFlags</code> to '' (the default).     |
| <b>Name: Compile termination</b><br>Default: ''                     | Format name passed to <code>fprintf</code> to write the termination portion of the compilation script.                                                                                                                                                                                                                        |

## Simulation Script Options

The following figure shows the **Simulation script** pane, with options set to their default values.



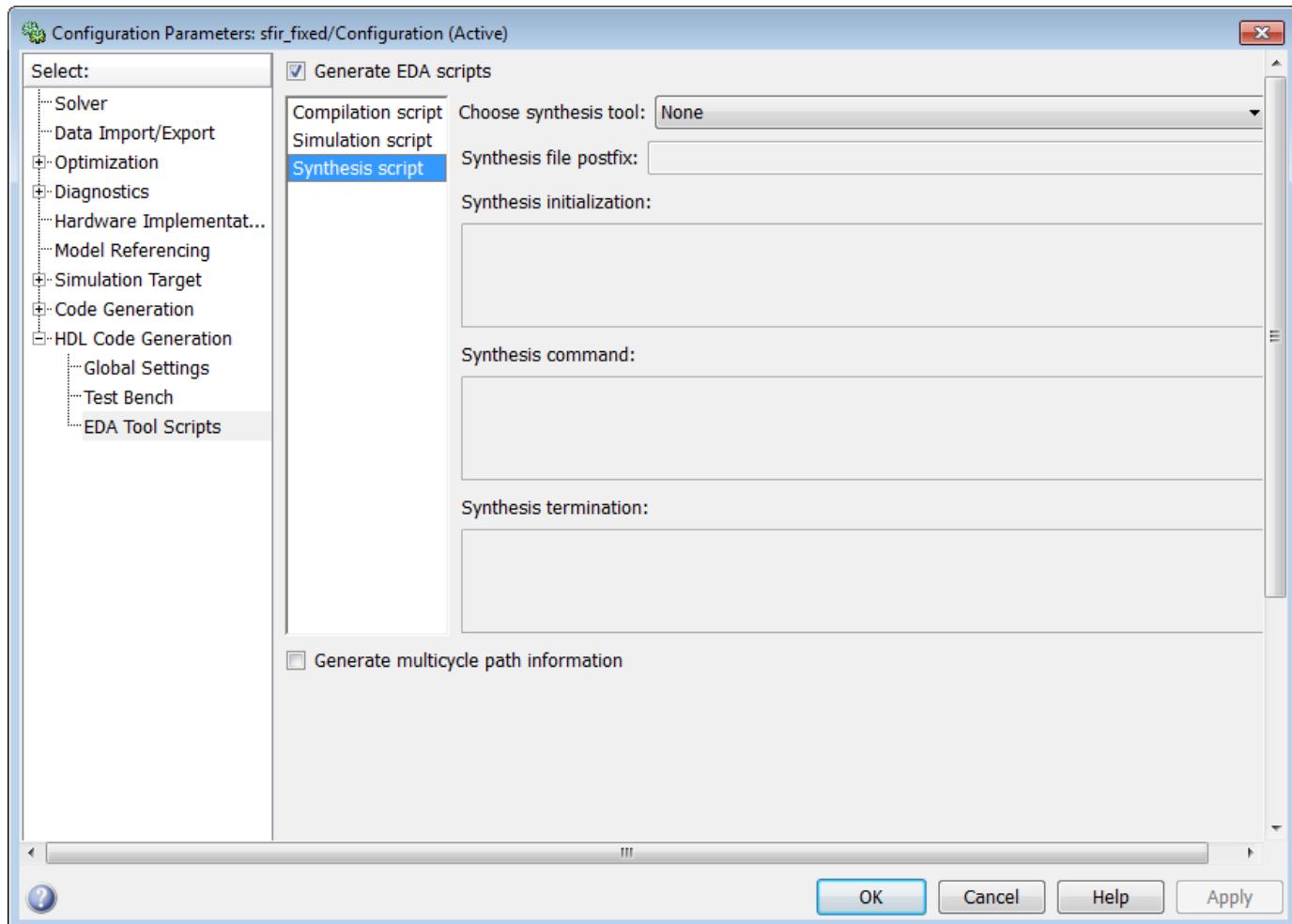
The following table summarizes the **Simulation script** options.

| Option and Default                                                                                 | Description                                                                                                                                                                                                                                                                                                                               |
|----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Simulation file postfix</b><br><code>'_sim.do'</code>                                           | Postfix appended to the model name or test bench name to form the simulation script file name.                                                                                                                                                                                                                                            |
| <b>Simulation initialization</b><br>Default:<br><code>['onbreak resume\nnonerror resume\n']</code> | Format name passed to <code>fprintf</code> to write the initialization section of the simulation script.                                                                                                                                                                                                                                  |
| <b>Simulation command</b><br>Default: <code>'vsim -voptargs+=acc %s.%s\n'</code>                   | Format name passed to <code>fprintf</code> to write the simulation command.<br><br>If your TargetLanguage is 'VHDL', the first implicit argument is the value of VHDLLibraryName. If your TargetLanguage is 'Verilog', the first implicit argument is 'work'.<br><br>The second implicit argument is the top-level module or entity name. |
| <b>Simulation waveform viewing command</b><br>Default: <code>'add wave sim:%s\n'</code>            | Format name passed to <code>fprintf</code> to write the simulation script waveform viewing command. The top-level module or entity signal names are implicit arguments.                                                                                                                                                                   |

| Option and Default                                     | Description                                                                                    |
|--------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <b>Simulation termination</b><br>Default: 'run -all\n' | Format name passed to <code>fprintf</code> to write the Term portion of the simulation script. |

## Synthesis Script Options

The following figure shows the **Synthesis script** pane, with options set to their default values. The **Choose synthesis tool** property defaults to None, which disables generation of a synthesis script.

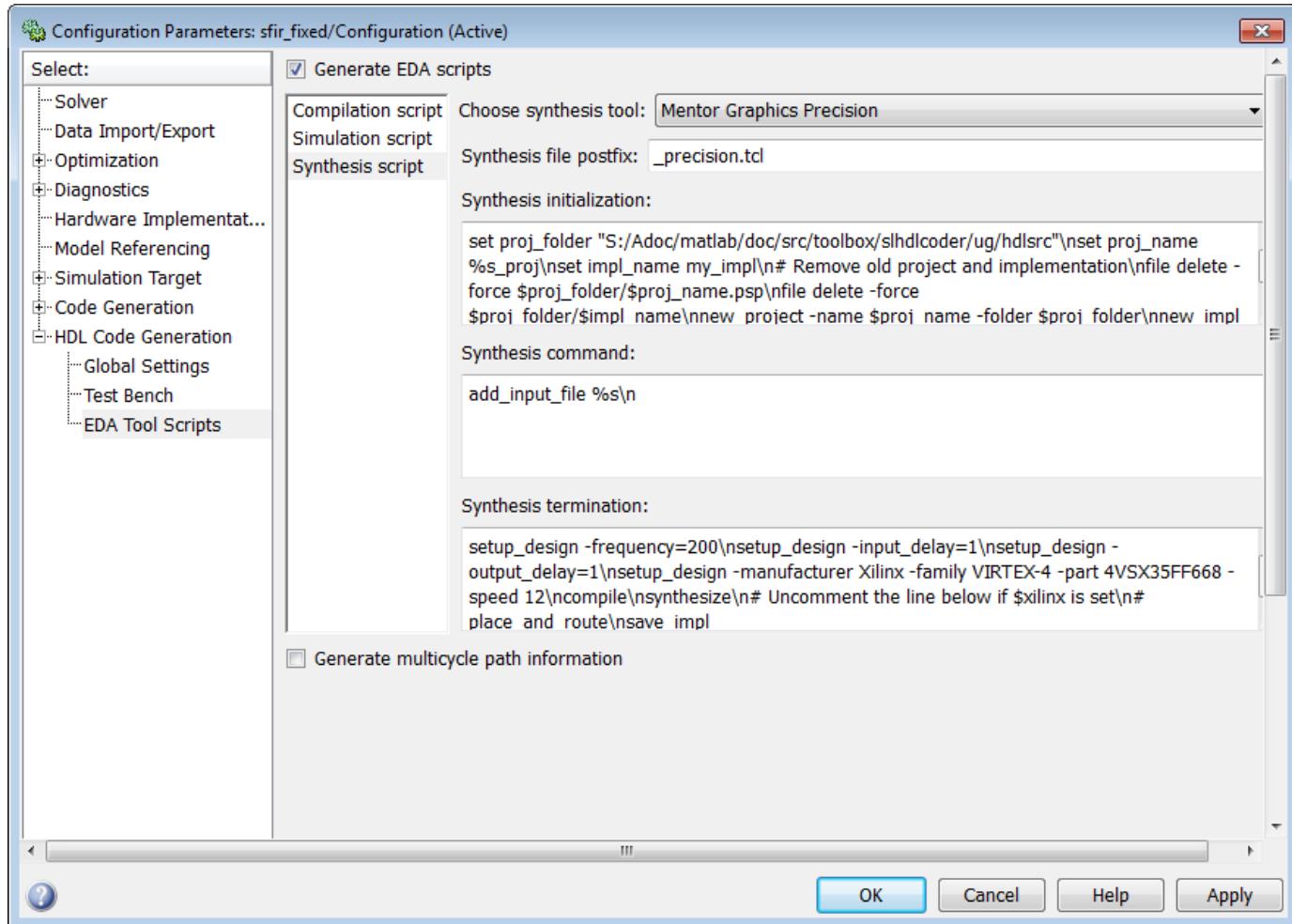


To enable synthesis script generation, select a synthesis tool from the **Choose synthesis tool** menu.

When you select a synthesis tool, HDL Coder:

- Enables synthesis script generation.
- Enters a file name postfix (specific to the chosen synthesis tool) into the **Synthesis file postfix** field.
- Enters strings (specific to the chosen synthesis tool) into the initialization, command, and termination fields.

The following figure shows the default option values entered for the Mentor Graphics Precision tool.



The following table summarizes the **Synthesis script** options.

| Option Name                  | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Choose synthesis tool</b> | None (default): do not generate a synthesis script<br>Xilinx ISE: generate a synthesis script for Xilinx ISE<br>Microsemi Libero: generate a synthesis script for Microsemi Libero<br>Mentor Graphics Precision: generate a synthesis script for Mentor Graphics Precision<br>Altera Quartus II: generate a synthesis script for Altera Quartus II<br>Synopsys Synplify Pro: generate a synthesis script for Synopsys Synplify Pro<br>Xilinx Vivado: generate a synthesis script for Xilinx Vivado<br>Custom: generate a custom synthesis script |

| <b>Option Name</b>              | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Synthesis file postfix</b>   | <p>Your choice of synthesis tool sets the postfix for generated synthesis file names to one of the following:</p> <ul style="list-style-type: none"> <li><code>_ise.tcl</code></li> <li><code>_libero.tcl</code></li> <li><code>_precision.tcl</code></li> <li><code>_quartus.tcl</code></li> <li><code>_synplify.tcl</code></li> <li><code>_vivado.tcl</code></li> <li><code>_custom.tcl</code></li> </ul> |
| <b>Synthesis initialization</b> | <p>Format name passed to <code>fprintf</code> to write the <code>Init</code> section of the synthesis script. The default string is a synthesis project creation command. The implicit argument is the top-level module or entity name.</p> <p>The content of the string is specific to the selected synthesis tool.</p>                                                                                    |
| <b>Synthesis command</b>        | <p>Format name passed to <code>fprintf</code> to write the <code>Cmd</code> section of the synthesis script. The implicit argument is the file name of the entity or module.</p> <p>The content of the string is specific to the selected synthesis tool.</p>                                                                                                                                               |
| <b>Synthesis termination</b>    | <p>Format name passed to <code>fprintf</code> to write the <code>Term</code> section of the synthesis script.</p> <p>The content of the string is specific to the selected synthesis tool.</p>                                                                                                                                                                                                              |

## Add Synthesis Attributes

To learn how to add synthesis attributes in the generated HDL code for multiplier mapping, see “DSPStyle” on page 22-10.

# Configure Synthesis Project Using Tcl Script

You can add a Tcl script that configures your synthesis project.

To configure your synthesis project using a Tcl script:

- 1 Create a Tcl script that contains commands to customize your synthesis project.

For example, to specify the finite state machine style:

- For Xilinx ISE, create a Tcl script that contains the following line:

```
project set "FSM Encoding Algorithm" "Gray" -process "Synthesize - XST"
```

- For Xilinx Vivado, create a Tcl script that contains the following line:

```
set_property STEPS.SYNTH_DESIGN.ARGS.FSM_EXTRACTION gray [get_runs synth_1]
```

- 2 In the HDL Workflow Advisor, in the **FPGA Synthesis and Analysis > Create Project** task, in the **Additional source files** field, enter the full path to the Tcl file manually, or by using the **Add** button.

When HDL Coder creates the project, the Tcl script is executed to apply the synthesis project settings.



# Using the HDL Workflow Advisor

---

- “Workflows in HDL Workflow Advisor” on page 31-2
- “Getting Started with the HDL Workflow Advisor” on page 31-6
- “Generate Code and Synthesize on FPGA Using HDL Workflow Advisor” on page 31-12
- “Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor” on page 31-17
- “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20
- “FPGA Floating-Point Library IP Mapping” on page 31-27
- “Customize Floating-Point IP Configuration” on page 31-39
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-47
- “Synthesis Objective to Tcl Command Mapping” on page 31-51
- “Run HDL Workflow with a Script” on page 31-53
- “Getting Started with the HDL Workflow Command-Line Interface” on page 31-65
- “Getting Started with FPGA Turnkey Workflow” on page 31-78

## Workflows in HDL Workflow Advisor

The HDL Workflow Advisor offers a workflow so that you can check your algorithm for HDL compatibility, generate HDL code, verify the code, and then deploy the code to your target platform.

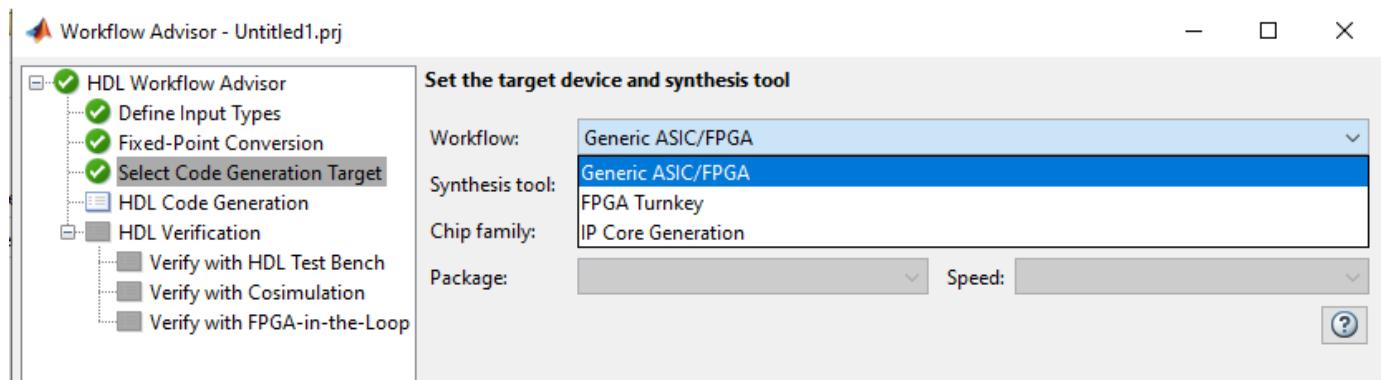
You can run the Workflow Advisor for your MATLAB algorithm or Simulink model. Before you deploy the code to a target hardware platform, install the synthesis tool and specify the path to that synthesis tool by using the `hdlsetuptoolpath` function. See “Tool Setup”.

### Set Up HDL Workflow Advisor in MATLAB

Before you specify the target workflow, when you run the Workflow Advisor from MATLAB, specify the design and test bench files, define the input types, and run fixed-point conversion.

To specify the target workflow:

- 1 On the MATLAB toolbar, from the **Apps** tab, select the **HDL Coder** app.
- 2 Select the MATLAB design and test bench files and click the **Workflow Advisor** button.
- 3 In the Workflow Advisor, on the **Select Code Generation Target** task, select the **Workflow**.



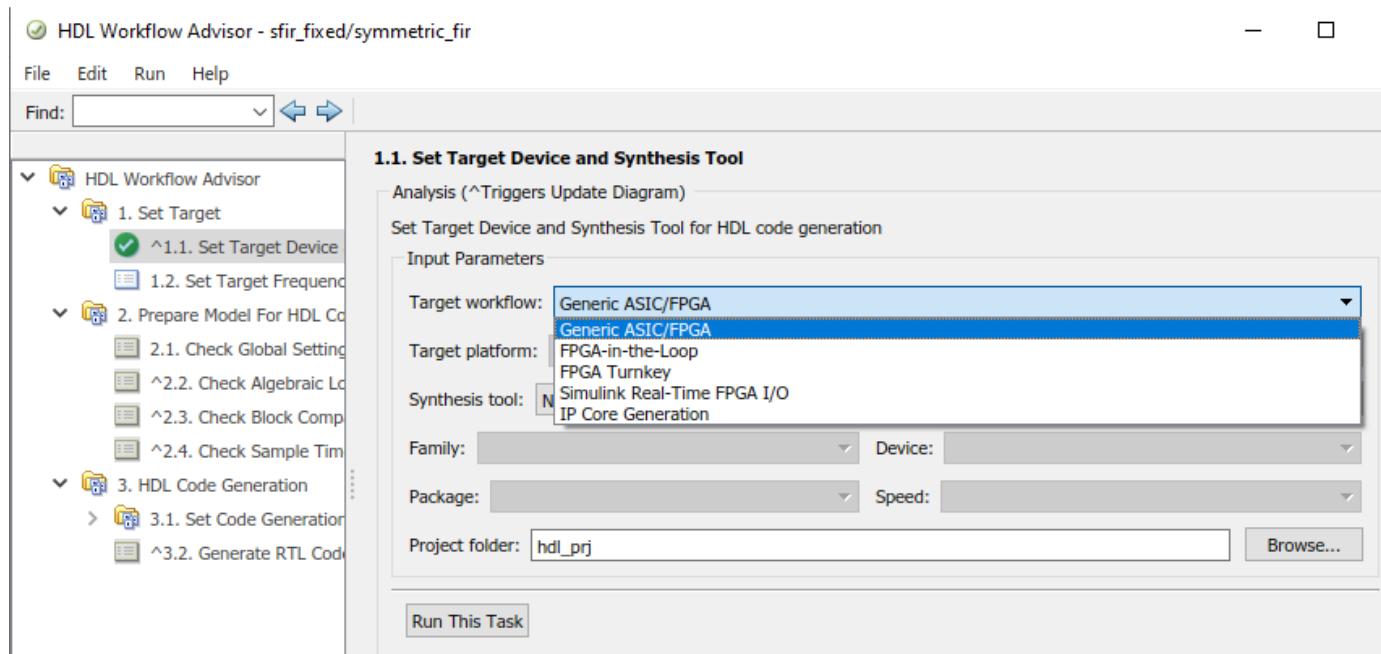
The steps after code generation target selection change depending on your target workflow.

### Set Up HDL Workflow Advisor in Simulink

When you run the Workflow Advisor from your Simulink model, irrespective of the target workflow, you run the steps to prepare the model for HDL code generation, and then generate code.

Open the Simulink model for which you want to run the workflow.

- 1 On the Simulink toolbar, from the **Apps** tab, select the **HDL Coder** app.
- 2 On the **HDL Code** tab, click the **Workflow Advisor** button.
- 3 In the HDL Workflow Advisor, on the **Set Target Device and Synthesis Tool** task, select the **Target workflow**.



The steps in the Workflow Advisor change depending on the **Target workflow**, **Target platform**, and **Synthesis tool**. The following sections describe more about each of these workflows.

## Generic ASIC/FPGA

Generate HDL code from your Simulink model or MATLAB algorithm, verify the HDL code, and deploy the code to a generic ASIC or FPGA device. You can select from a family of devices that belong to these synthesis tools as listed in “Generic ASIC/FPGA Hardware”.

By using this workflow, you can:

- Generate HDL code for your fixed-point MATLAB algorithm or your HDL-compatible Simulink model.
- Generate an HDL test bench and cosimulation test bench (requires HDL Verifier), and scripts to build and run the code and test bench. You can also generate a SystemVerilog DPI test benches and code coverage when running the Simulink HDL Workflow Advisor (requires HDL Verifier).
- Perform FPGA synthesis and timing analysis and rapidly prototype your design on generic FPGA platforms through integration with third-party synthesis tools.
- Back-annotate the model with critical path information and other information obtained during synthesis, and optimize your design for area and speed.

---

**Note** If you select Intel Quartus Pro or Microsemi Libero SoC as the **Synthesis tool**, the **Annotate Model with Synthesis Result** task is not available. To see the critical path, run the workflow to synthesis and then open the timing reports.

---

To learn more, see:

- “HDL Code Generation and FPGA Synthesis from Simulink Model”
- “Basic HDL Code Generation and FPGA Synthesis from MATLAB”

## FPGA Turnkey

Deploy your Simulink model or MATLAB algorithm onto standalone FPGA boards and SoC platforms. To use this workflow, you must select **VHDL** as the **Language**. You can select from one of these synthesis tools as listed in “FPGA Turnkey Hardware”.

Use this workflow to:

- Choose boards from the FPGA Board Manager that are **Turnkey Enabled** or create your own custom boards for deployment.
- Generate HDL code for the entire FPGA design, the DUT algorithm, and the FPGA wrapper top-level HDL code. You can also specify the pin mapping constraints.
- Perform FPGA synthesis and timing analysis and rapidly prototype your design on FPGA and SoC platforms through integration with third-party synthesis tools.

For an example, see “Getting Started with FPGA Turnkey Workflow” on page 31-78.

## IP Core Generation

Generate RTL code and a custom HDL IP core from your Simulink model or MATLAB algorithm. Before you run the workflow, partition your design into components that run on software and components that run on hardware. See “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2.

The IP core is a shareable and reusable HDL component that consists of IP core definition files, HDL code generated for your algorithm, C header file with the register address map, and the IP core report. See:

- “Custom IP Core Report” on page 40-13
- “Custom IP Core Generation” on page 40-10

You can select from one of these synthesis tools as listed in “IP Core Generation Hardware”.

Use this workflow to:

- Generate a generic board-independent Xilinx or Intel HDL IP core.
- Integrate the IP core into a reference design to target standalone FPGA boards or SoC platforms with Xilinx Vivado IP integrator or Intel Qsys.
- Communicate with the generated HDL IP core by using embedded ARM processor or from MATLAB by using the HDL Verifier MATLAB as AXI Master. See “Set Up for MATLAB AXI Master” (HDL Verifier).

You can integrate the HDL IP core into HDL Coder provided reference designs such as the **default system reference design** or into a reference design that you created. To learn more, see:

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

## Simulink Real-Time FPGA I/O

Generate HDL code from your Simulink model and deploy the code onto Speedgoat FPGA I/O modules. This workflow requires Xilinx Vivado and uses the IP Core Generation workflow infrastructure as mentioned in “Simulink Real-Time FPGA I/O: Speedgoat Target Hardware”.

To run the Simulink Real-Time FPGA I/O workflow, install the Speedgoat Library and the Speedgoat HDL Coder Integration Packages. After you install the integration packages, you can choose the **Target platform** and then run the workflow to:

- Generate a reusable and shareable IP core.
- Integrate the IP core into the Speedgoat reference design.
- Generate an FPGA bitstream and download the bitstream to the target hardware.
- Generate a Simulink Real-Time™ model. The model is an interface subsystem model that contains the blocks to program the FPGA and communicate with the board during real-time execution.

To learn more, see “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93.

## FPGA-in-the-Loop

Test your Simulink model or MATLAB algorithm on a target FPGA. This workflow requires HDL Verifier. You can select from one of these synthesis tools as listed in “FPGA-in-the-Loop Hardware”.

Use this workflow to:

- Choose boards from the FPGA Board Manager that are **FIL Enabled** or create your own custom boards for verification. See “FPGA Board Customization” on page 36-2.
- Generate HDL code for your fixed-point MATLAB algorithm or your HDL-compatible Simulink model.
- Perform FPGA implementation and connect to the target FPGA board using Ethernet, JTAG, or PCI Express for FIL simulation.

To learn more, see:

- “FIL Simulation with HDL Workflow Advisor for Simulink” (HDL Verifier)
- “FIL Simulation with HDL Workflow Advisor for MATLAB” (HDL Verifier)

## See Also

`hdladvisor` | `makehdl`

## More About

- “Getting Started with the HDL Workflow Advisor” on page 31-6
- “HDL Workflow Advisor Tasks” on page 37-2
- “Run HDL Workflow with a Script” on page 31-53
- “Getting Started with the HDL Workflow Command-Line Interface” on page 31-65

## Getting Started with the HDL Workflow Advisor

### In this section...

- “Open the HDL Workflow Advisor” on page 31-6
- “Run Tasks in the HDL Workflow Advisor” on page 31-7
- “Fix HDL Workflow Advisor Warnings or Failures” on page 31-8
- “Save and Restore the HDL Workflow Advisor State” on page 31-8
- “View and Save HDL Workflow Advisor Reports” on page 31-9

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

### Open the HDL Workflow Advisor

To start the HDL Workflow Advisor from a Simulink model:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Workflow Advisor**.

To start the HDL Workflow Advisor for a model from the command line, enter `hdladvisor(system)`. `system` is a handle or name of the model or subsystem that you want to check. For more information, see the `hdladvisor` function reference page.

For how to use the HDL Workflow Advisor to generate HDL code from a MATLAB script, see “Basic HDL Code Generation and FPGA Synthesis from MATLAB”.

In the HDL Workflow Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related tasks. Expanding the folders shows available tasks in each folder. From the left pane, you can select a folder or an individual task. The HDL Workflow Advisor displays information about the selected folder or task in the right pane. The contents of the right pane depends on the selected folder or task. For some tasks, the right pane contains simple controls for running the task and a display area for status messages and other task results. For other tasks that involve setting code or test bench generation parameters, the right pane displays several parameter and option settings.

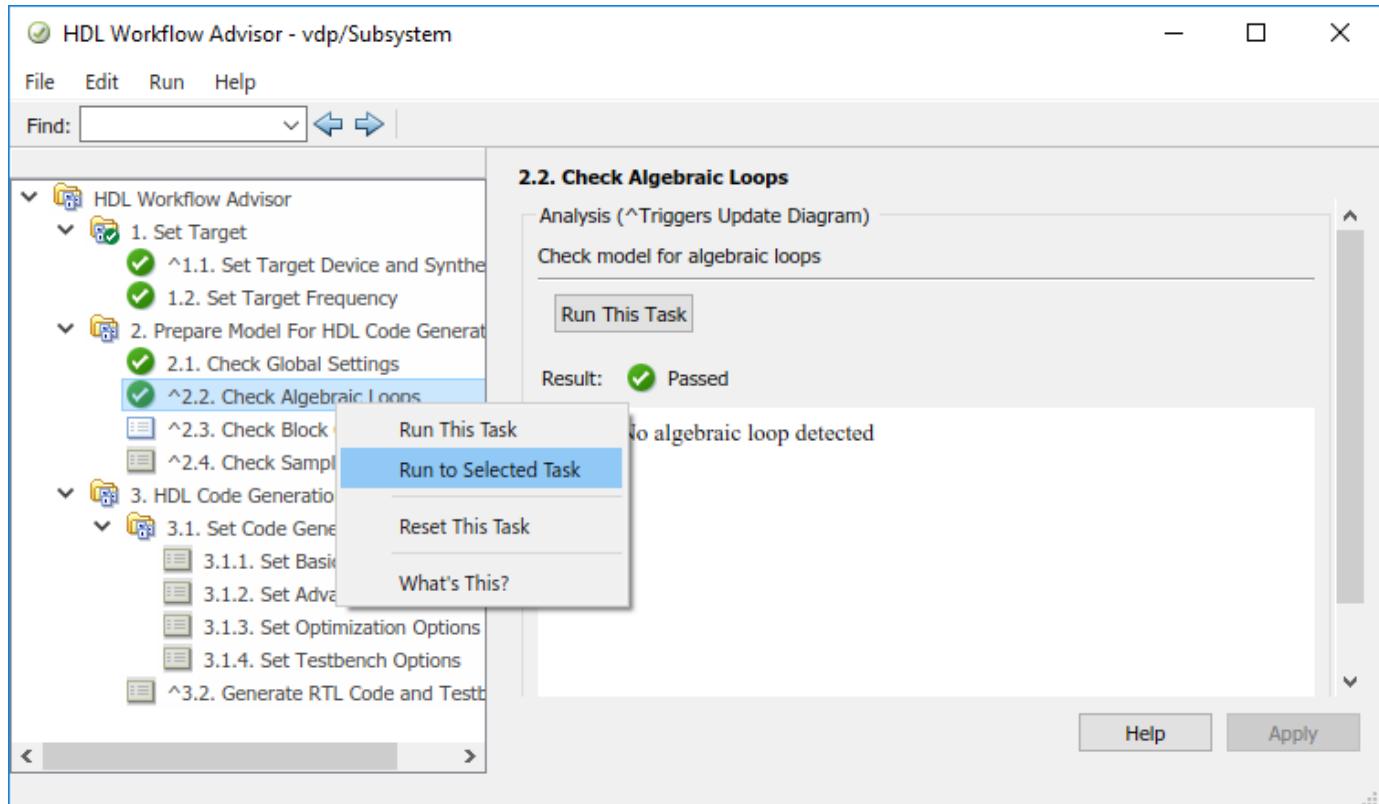
## Run Tasks in the HDL Workflow Advisor

In the HDL Workflow Advisor window, you can run individual tasks, a group of tasks, or all the tasks in the workflow. For example, before you generate HDL code, prepare the Simulink model for HDL code generation. In the **Set Target** folder, for each individual task, you can specify the target device settings and the target frequency. Then, select the task that you want to run and click **Run This Task**. To run a task, all tasks before it must have run successfully.

To learn more about each individual task, right-click that task, and select **What's This?**.



To generate HDL code, run the workflow to the **Generate RTL Code and Testbench** task. To run the workflow to a specific task inside a subfolder, expand that folder, and then right-click the task and select **Run To Selected Task**.

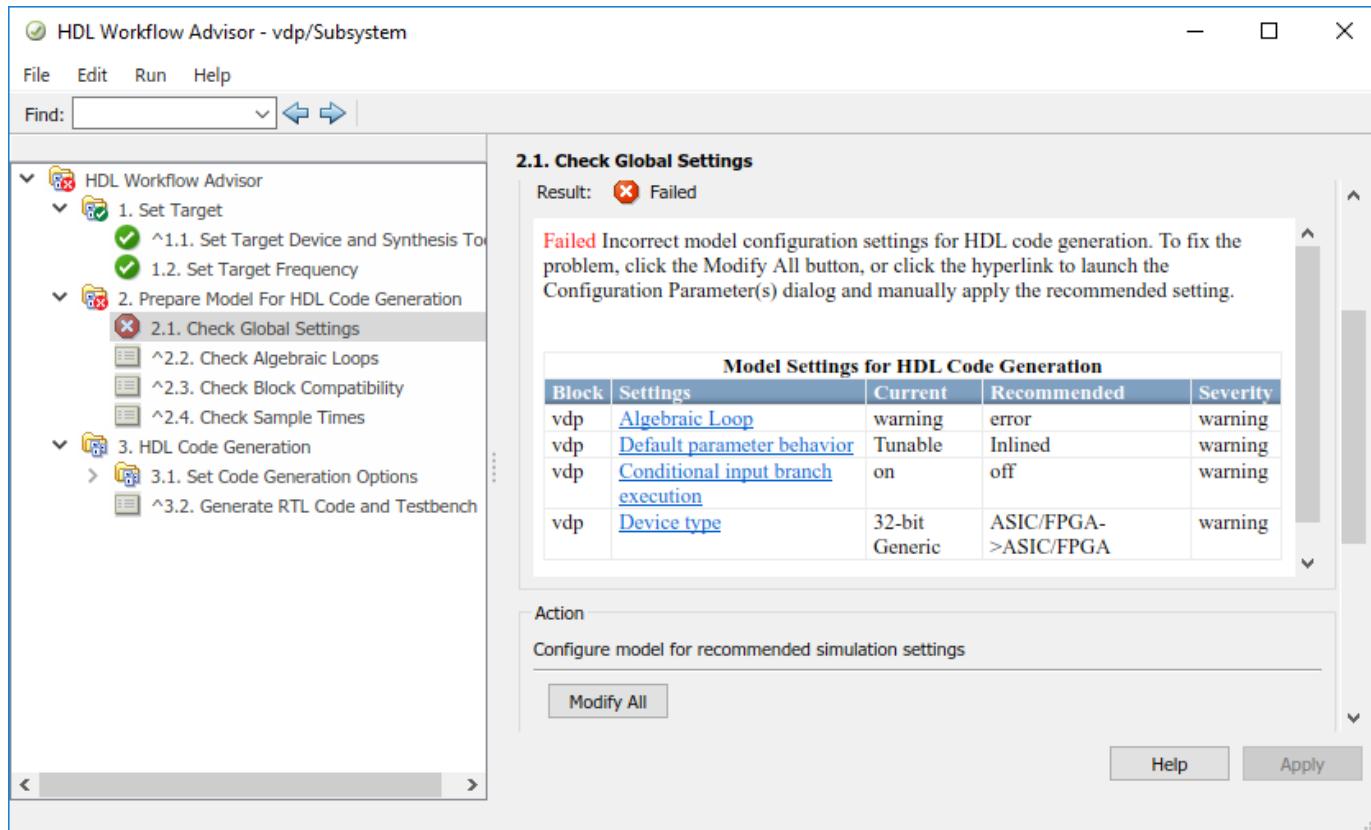


To rerun a task that you have already run, click **Reset This Task**. HDL Coder then resets the task and all tasks that follow it. For example, to customize the basic options for generating HDL code after running the **Generate RTL Code and Testbench** task, right-click the **Set Basic Options** task and select **Reset This Task**. You can then set the basic options on the model and click **Run This Task** to rerun the task.

To run all the tasks in the HDL Workflow Advisor with the default settings, in the HDL Workflow Advisor window, select **Run > Run All**. To run a group of tasks in a specific folder, select that folder and click **Run All**.

## Fix HDL Workflow Advisor Warnings or Failures

In the HDL Workflow Advisor, if a task terminates due to a warning or failure condition, the right pane shows the warning or failure information in a **Result** subpane. The **Result** subpane displays model settings that you can use to fix the warnings. For some tasks, use the **Action** subpane to apply those recommended actions.



For example, to apply the model configuration settings that the code generator reported in the **Result** subpane, click the **Modify All** button. After you click **Modify All**, the **Result** subpane reports the changes that were applied. and resets the task. You can now run this task.

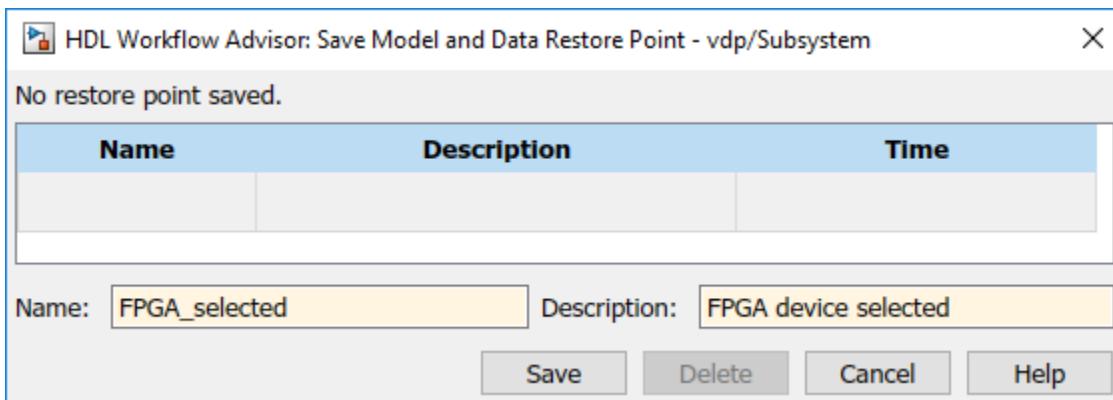
## Save and Restore the HDL Workflow Advisor State

By default, the HDL Coder software saves the state of the most recent HDL Workflow Advisor session. The next time that you activate the HDL Workflow Advisor, it returns to that state. You can also save the current settings of the HDL Workflow Advisor to a named restore point. Later, you can restore the same settings by loading the restore point data into the HDL Workflow Advisor.

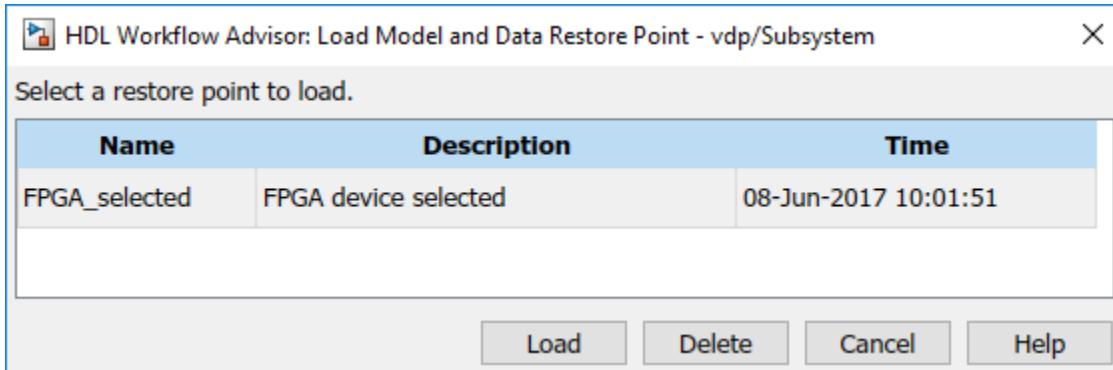
The save and restore process does not:

- Include operations that you perform outside the HDL Workflow Advisor.
- Save or restore the state of HDL Workflow Advisor tasks involving third-party tools.

To save the Workflow Advisor state, in the HDL Workflow Advisor Window, select **File > Save Restore Point As**. Enter a **Name** and **Description**, and then click **Save**. You can save more than one restore point.

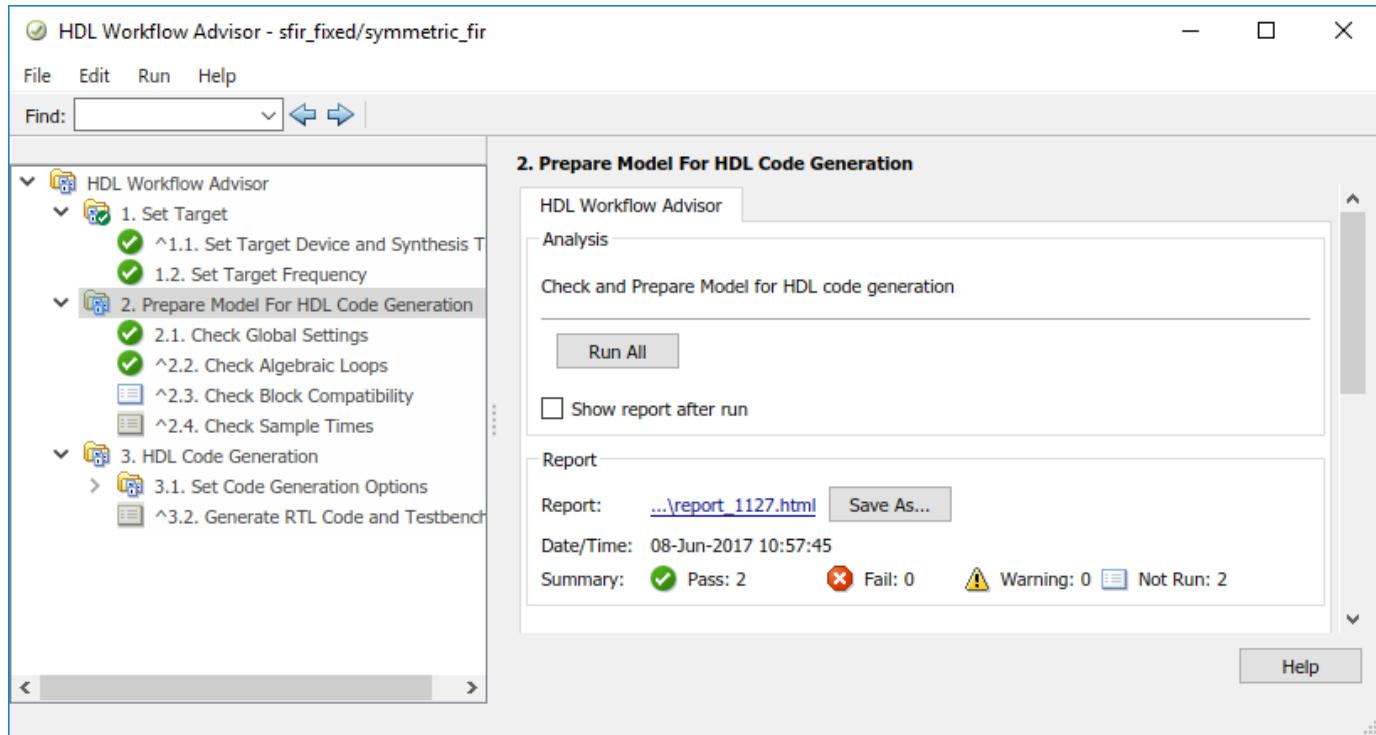


To restore a Workflow Advisor state, in the HDL Workflow Advisor, select **File > Load Restore Point**. Select the restore point that you want to load and click **Load**. When you load a restore point, the HDL Workflow Advisor warns that the restoration overwrites the current settings.



## View and Save HDL Workflow Advisor Reports

When you run tasks in the HDL Workflow Advisor, HDL Coder generates an HTML report of the task results. Each folder in the HDL Workflow Advisor contains a report for the checks within that folder and its subfolders. To access reports, select a folder, such as **Prepare Model for HDL Code Generation**, and in the **Report** subpane, click **Save As**. If you rerun the HDL Workflow Advisor, the report is updated in the working folder.



This report shows typical results after running the **Prepare Model For HDL Code Generation** tasks.

**Model Advisor Report - sfir\_fixed.mdl**

**Simulink version:** 9.0      **Model version:** 1.70  
**System:** sfir\_fixed/symmetric\_fir      **Current run:** 08-Jun-2017 10:57:45

**Treat as Referenced Model:** off

**Run Summary**

| Pass | Fail | Warning | Not Run | Total |
|------|------|---------|---------|-------|
| ✓ 2  | ✗ 0  | ⚠ 0     | ⌚ 2     | 4     |

**2. Prepare Model For HDL Code Generation**

**2.1. Check Global Settings** (08-Jun-2017 10:54:13)  
 Passed Correct Simulation settings for HDL code generation

**Input Parameters Selection**

| Name            | Value |
|-----------------|-------|
| Ignore warnings | false |

As you run checks, the HDL Workflow Advisor updates the reports with the latest information for each check in the folder. When you run the checks at different times, timestamps appear at the top right of the report to indicate when checks have been run. Checks that occurred during previous runs have a timestamp following the check name. You can filter checks in the report. For example, you can filter out tasks that are **Not Run** from the report, or you can filter the report to show tasks that **Passed**, and so on. To view the report for a folder each time the tasks in a folder are run, select **Show report after run**.

## See Also

### Functions

`clearAllTasks | hdlcoder.runWorkflow | setAllTasks`

### Classes

`hdlcoder.WorkflowConfig`

## Related Examples

- “HDL Workflow Advisor Tasks” on page 37-2
- “HDL Code Generation and FPGA Synthesis from Simulink Model”

## Generate Code and Synthesize on FPGA Using HDL Workflow Advisor

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

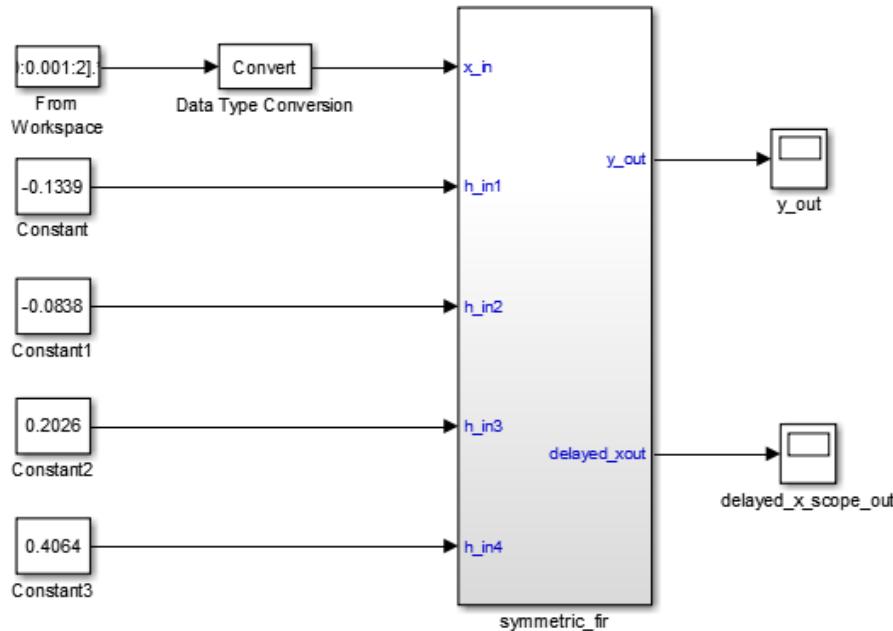
- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

### FIR Filter Model

This example illustrates how you can generate HDL code for the FIR filter model and synthesize the design on an FPGA device. Before you generate HDL code, the model must be compatible for HDL code generation. To check and update your model for HDL compatibility, see “Check HDL Compatibility of Simulink Model Using HDL Code Advisor” on page 39-2.

This example uses the Symmetric FIR filter model that is compatible for HDL code generation. To open this model at the command line, enter:

```
sfir_fixed
```



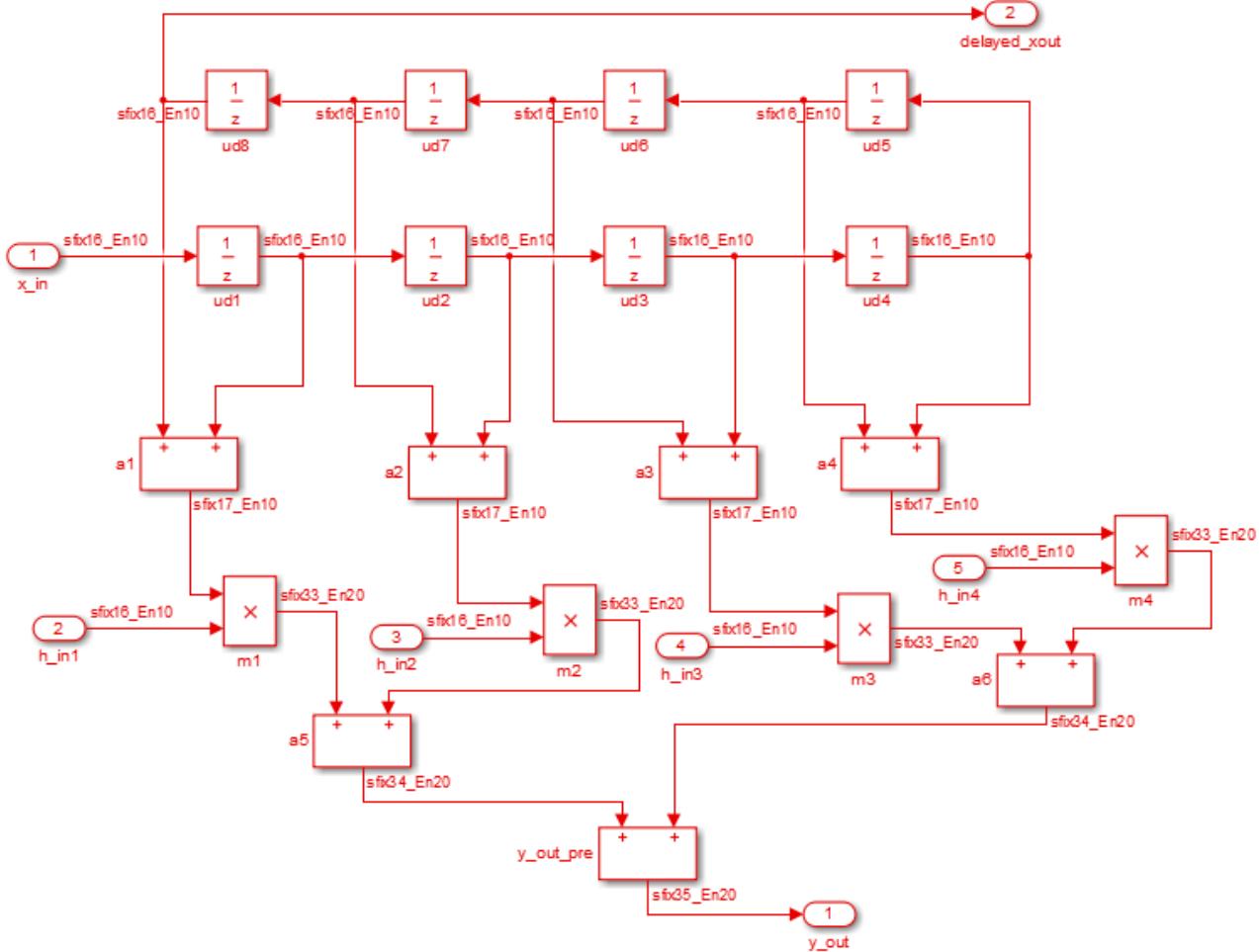
The model uses a division of labor that is suitable for HDL design.

- The `symmetric_fir` subsystem, which implements the filter algorithm, is the device under test (DUT). An HDL entity is generated from this subsystem.
- The top-level model components that drive the subsystem work as a test bench.

The top-level model generates 16-bit fixed-point input signals for the `symmetric_fir` subsystem. The Signal From Workspace block generates a test input (stimulus) signal for the filter. The four Constant blocks provide filter coefficients. The Scope blocks are used for simulation and are not used for HDL code generation.

To navigate to the `symmetric_fir` subsystem, enter:

```
open_system('sfir_fixed/symmetric_fir')
```



## Create a Folder and Copy Relevant Files

In MATLAB:

- 1 Create a folder named `sl_hdlcoder_work`, for example:

```
mkdir C:\work\sl_hdlcoder_work
```

`sl_hdlcoder_work` stores a local copy of the example model and folders and generated HDL code. Use a folder location that is not within the MATLAB folder tree.

- 2 Make the `sl_hdlcoder_work` folder your working folder, for example:

```
cd C:\work\sl_hdlcoder_work
```

- 3 Save a local copy of the `sfir_fixed` model to your current working folder. Leave the model open.

## Set Up Tool Path

If you do not want to synthesize your design, but want to generate HDL code, you do not have to set the tool path. In the HDL Workflow Advisor, on the **Set Target > Set Target Device and Synthesis Tool** step, leave the **Synthesis tool** setting to the default **No Synthesis Tool Specified**, and then run the workflow.

If you want to synthesize your design on a target platform, before you open the HDL Workflow Advisor and run the workflow, set up the path to your synthesis tool. This example uses Xilinx Vivado, so you must have already installed Xilinx Vivado. To set the tool path, use the `hdlsetuptoolpath` function to point to an installed Xilinx Vivado 2019.2 executable. Optionally, you can use a different synthesis tool of your choice and follow this example. To set the path to that synthesis tool, use `hdlsetuptoolpath`. To learn about the latest supported tools, see “HDL Language Support and Supported Third-Party Tools and Hardware”.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath',...
'C:\Xilinx\Vivado\2019.1\bin\vivado.bat');
```

## Open the HDL Workflow Advisor

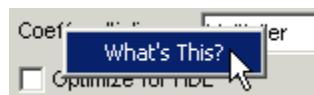
To start the HDL Workflow Advisor from a Simulink model,

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears.
- 2 Select the DUT Subsystem in your model, and make sure that this Subsystem name appears in the **Code for** option. To remember the selection, you can pin this option. Click **Workflow Advisor**.

When you open the HDL Workflow Advisor, the code generator might warn that the project folder is incompatible. To open the Advisor, select **Remove slprj and continue**.

In the HDL Workflow Advisor, the left pane lists the folders in the hierarchy. Each folder represents a group or category of related tasks. From the left pane, you can select a folder or an individual task. The HDL Workflow Advisor displays information about the selected folder or task in the right pane.

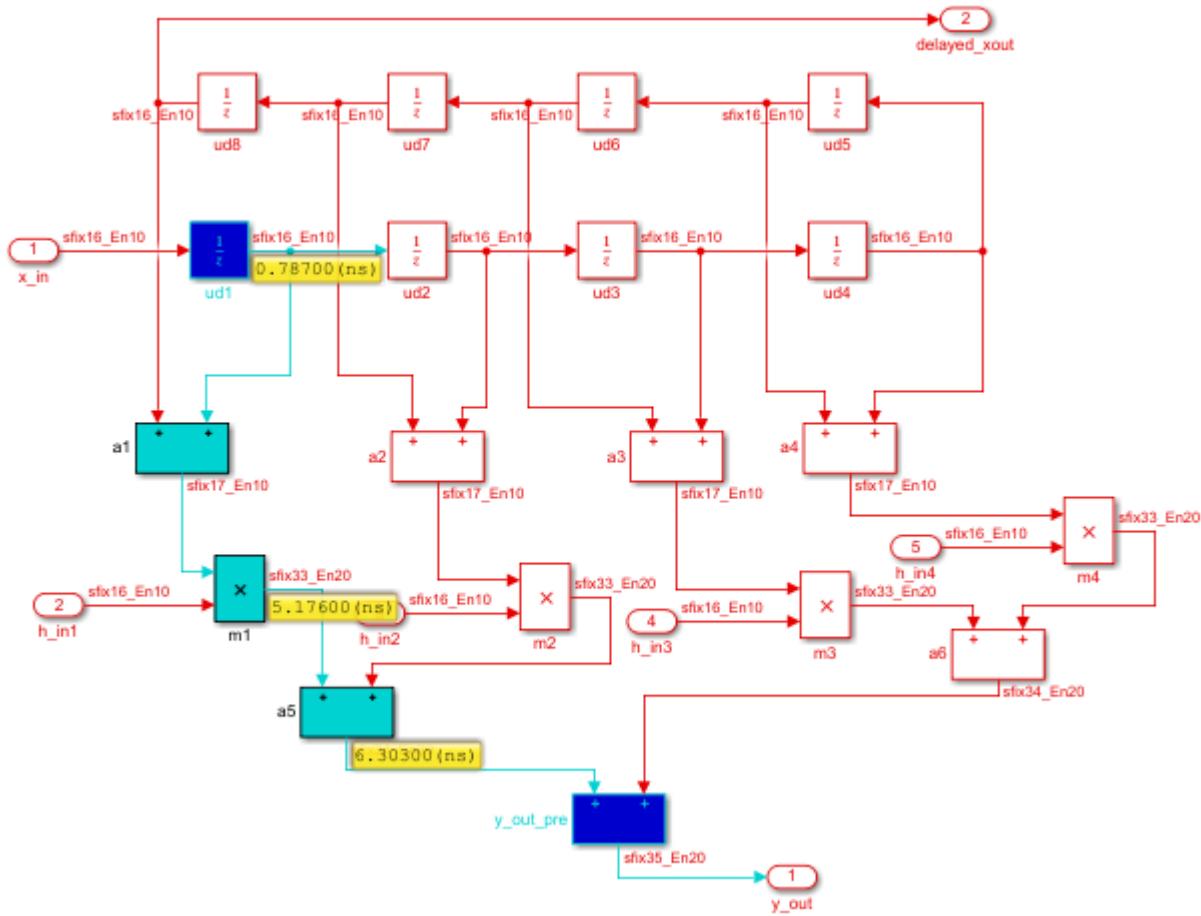
To learn more about each individual task, right-click that task, and select **What's This?**.



To learn more about the HDL Workflow Advisor window, see “Getting Started with the HDL Workflow Advisor” on page 31-6.

## Generate HDL Code and Synthesize on FPGA

- 1 In the **Set Target > Set Target Device and Synthesis Tool** step, for **Synthesis tool**, select **Xilinx Vivado** and select **Run This Task**.
- 2 To generate code, right-click the **Generate RTL Code and Testbench** task, and select **Run to Selected Task**.
- 3 In the **FPGA Synthesis and Analysis > Perform Synthesis and P/R > Run Implementation** task, clear **Skip this task** and click **Apply**.
- 4 Right-click the **Annotate Model with Synthesis Result** and select **Run to Selected Task**.



## Run Workflow at Command Line with a Script

To run the HDL workflow at a command line, you can export the Workflow Advisor settings to a script. To export to script, in the HDL Workflow Advisor window, select **File > Export to Script**. In the Export Workflow Configuration dialog box, enter a file name and save the script.

The script is a MATLAB file that you can run from the command line. You can modify the script directly or, import the script into the HDL Workflow Advisor, modify the tasks, and export the updated script. To learn more, see “Run HDL Workflow with a Script” on page 31-53.

**See Also**

`hdladvisor` | `makehdl`

**More About**

- “Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor” on page 31-17
- “Generate HDL Code from Simulink Model Using Configuration Parameters” on page 12-11
- “Generate HDL Code from Simulink Model from Command Line” on page 12-15

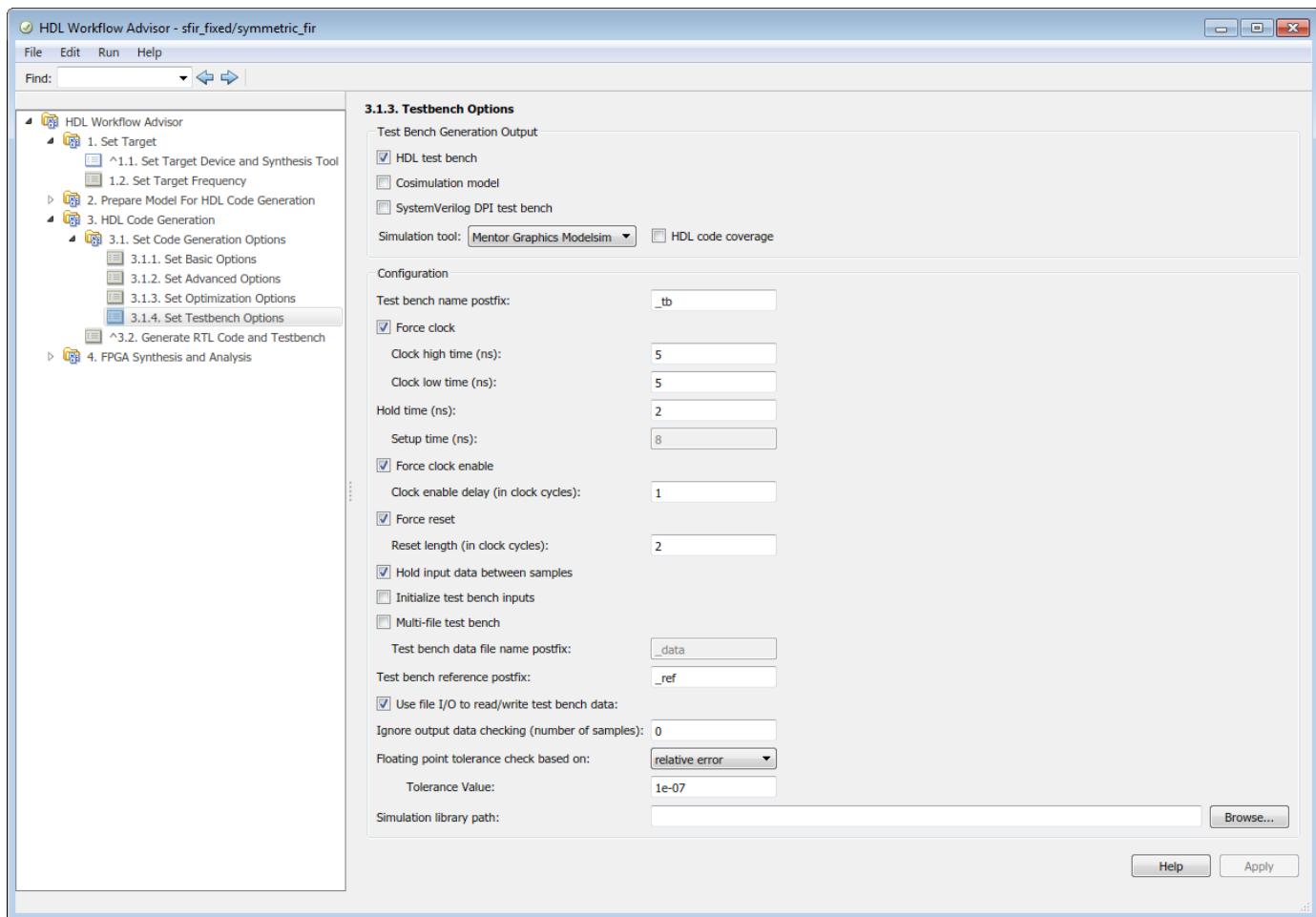
# Generate Test Bench and Enable Code Coverage Using the HDL Workflow Advisor

The HDL Workflow Advisor guides you through the stages of generating HDL code for a Simulink subsystem and the FPGA design process, such as:

- Checking the model for HDL code generation compatibility and automatically fixing incompatible settings.
- Generation of HDL code, a test bench, and scripts to build and run the code and test bench.
- Generation of cosimulation or SystemVerilog DPI test benches and code coverage (requires HDL Verifier).
- Synthesis and timing analysis through integration with third-party synthesis tools.
- Back-annotation of the model with critical path information and other information obtained during synthesis.
- Complete automated workflows for selected FPGA development target devices, including FPGA-in-the-loop simulation (requires HDL Verifier), and the Simulink Real-Time FPGA I/O workflow.

To select test bench and code coverage options for generating HDL code from a Simulink model using the HDL Workflow Advisor:

- 1 Perform the setup steps in “HDL Code Generation and FPGA Synthesis from Simulink Model”.
- 2 In Step 3.1.4 of the HDL Workflow Advisor, **Set Testbench Options**, select test bench and code coverage options from the **Test Bench Generation Output** section. The coder generates a build-and-run script for your test bench and the **Simulation tool** you specify. If you select multiple test bench options, the coder generates one test bench and script for each type of test bench selected. If you select **HDL code coverage**, the test bench scripts turn on code coverage for your generated HDL code. For more information about the different kinds of test benches, see “Choose a Test Bench for Generated HDL Code” on page 27-39. After you select your test bench options, click **Apply**.



- 3** In Step 3.2, **Generate RTL Code and Testbench**, select **Generate test bench**. Click **Apply**, and then click **Run This Task**. The coder generates HDL code for your subsystem, and the test benches and scripts you selected in step 3.1.3.

- If you selected **Cosimulation model**, then step 3.3, **Verify with HDL Cosimulation**, appears in the HDL Workflow Advisor. This step automatically runs the generated cosimulation model. The model compares the result of the HDL code running in your HDL simulator with the output of your Simulink subsystem.
- If you selected **HDL test bench**, the coder generates a compile script, `subsystemname_tb_compile`, and a run script, `subsystemname_tb_sim`. The script file extension depends on your selected simulator. For example, at the command line in the Mentor Graphics ModelSim simulator, change to the `hdl_prj/hdlsrc/modelname` folder and run these commands:

```
do symmetric_fir_compile.do
do symmetric_fir_tb_compile.do
do symmetric_fir_tb_sim.do
```

- If you selected **SystemVerilog DPI test bench**, the coder generates a script file, `subsystemname_dpi_tb`, that compiles the HDL code and runs the test bench simulation. The script file extension depends on your selected simulator. For example, at the command line in the Mentor Graphics ModelSim simulator, change to the `hdl_prj/hdlsrc/modelname` folder and run this command:

```
do symmetric_fir_dpi_tb.do
```

- If you selected **HDL code coverage**, the code coverage report from running any test bench, including the cosimulation model, is saved in `hdl_prj\hdlsrc\modelname\covhtmlreport`.

The screenshot shows a web browser window titled "Web Browser - Questa Coverage Report". The URL is "file:///C:/MATLAB/tb\_ex\_wfa/hdl\_prj/hdlsrc/sfir\_fixed/covhtmlreport/pages/\_frametop.htm". The main content is the "Questa Coverage Report".

**Testplan** [Design] DesUnits

**Design**

Number of tests run: 1

|          |   |
|----------|---|
| Passed:  | 0 |
| Warning: | 1 |
| Error:   | 0 |
| Fatal:   | 0 |

[List of tests included in report...](#)

[List of global attributes included in report...](#)

**Coverage Summary by Structure:**

| Design Scope                         | Coverage |
|--------------------------------------|----------|
| <a href="#">symmetric_fir_tb</a>     | 59.91%   |
| <a href="#">u_symmetric_fir</a>      | 97.03%   |
| <a href="#">symmetric_fir_tb_pkg</a> | 0.00%    |
| <a href="#">to_hex</a>               | 0.00%    |
| <a href="#">to_hex_1</a>             | 0.00%    |
| <a href="#">to_hex_2</a>             | 0.00%    |
| <a href="#">to_hex_3</a>             | 0.00%    |
| <a href="#">to_hex_4</a>             | 0.00%    |

**Coverage Summary by Type:**

| Total Coverage: | 73.69% | 53.71% |        |        |        |          |
|-----------------|--------|--------|--------|--------|--------|----------|
| Coverage Type   | Bins   | Hits   | Misses | Weight | % Hit  | Coverage |
| Statements      | 200    | 154    | 46     | 1      | 77.00% | 77.00%   |
| Branches        | 131    | 103    | 28     | 1      | 78.62% | 78.62%   |
| FEC Expressions | 19     | 12     | 7      | 1      | 63.15% | 63.15%   |
| FEC Conditions  | 10     | 3      | 7      | 1      | 30.00% | 30.00%   |
| Toggles         | 3076   | 2261   | 815    | 1      | 73.50% | 73.50%   |
| Assertions      | 1      | 0      | 1      | 1      | 0.00%  | 0.00%    |

## See Also

### More About

- "Choose a Test Bench for Generated HDL Code" on page 27-39

# Generate HDL Code for FPGA Floating-Point Target Libraries

## In this section...

- “Setup for FPGA Floating-Point Library Mapping” on page 31-20
- “Map to an FPGA Floating-Point Library” on page 31-20
- “View Code Generation Reports of Floating-Point Library Mapping” on page 31-22
- “Analyze Results of Floating-Point Library Mapping” on page 31-24

Mapping to a floating-point library enables you to synthesize your floating-point design without having to do floating-point to fixed-point conversion. Eliminating the floating-point to fixed-point conversion step reduces the loss of data precision, and enables you to model a wider dynamic range.

An FPGA floating-point library is a set of floating-point IP blocks that are optimized for synthesis on specific target hardware. Altera Megafunctions and Xilinx LogiCORE IP are examples of such libraries.

In the HDL Coder block library, a subset of Simulink blocks support floating-point library mapping. See “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-47.

## Setup for FPGA Floating-Point Library Mapping

To map your floating-point design to an Altera or Xilinx FPGA floating-point library:

- Set the target device options for your Altera or Xilinx FPGA synthesis tool using `hdlset_param`. For example, to set the synthesis tool as `Altera Quartus II` and chip family as `Arria10`:
 

```
hdlset_param(model, 'SynthesisToolChipFamily', 'Arria10', ...
 'SynthesisToolDeviceName', '10AS066H2F34E1SG', ...
 'SynthesisToolPackageName', '', ...
 'SynthesisToolSpeedValue', '')
```
- To set up the path to your synthesis tool executable file, use `hdlsetupoolpath`. For example, to set the path to the `Altera Quartus II` synthesis tool:
 

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

 See “[Synthesis Tool Path Setup](#)”.
- Set up your Altera or Xilinx FPGA floating-point simulation libraries. See “[FPGA Simulation Library Setup](#)”.

## Map to an FPGA Floating-Point Library

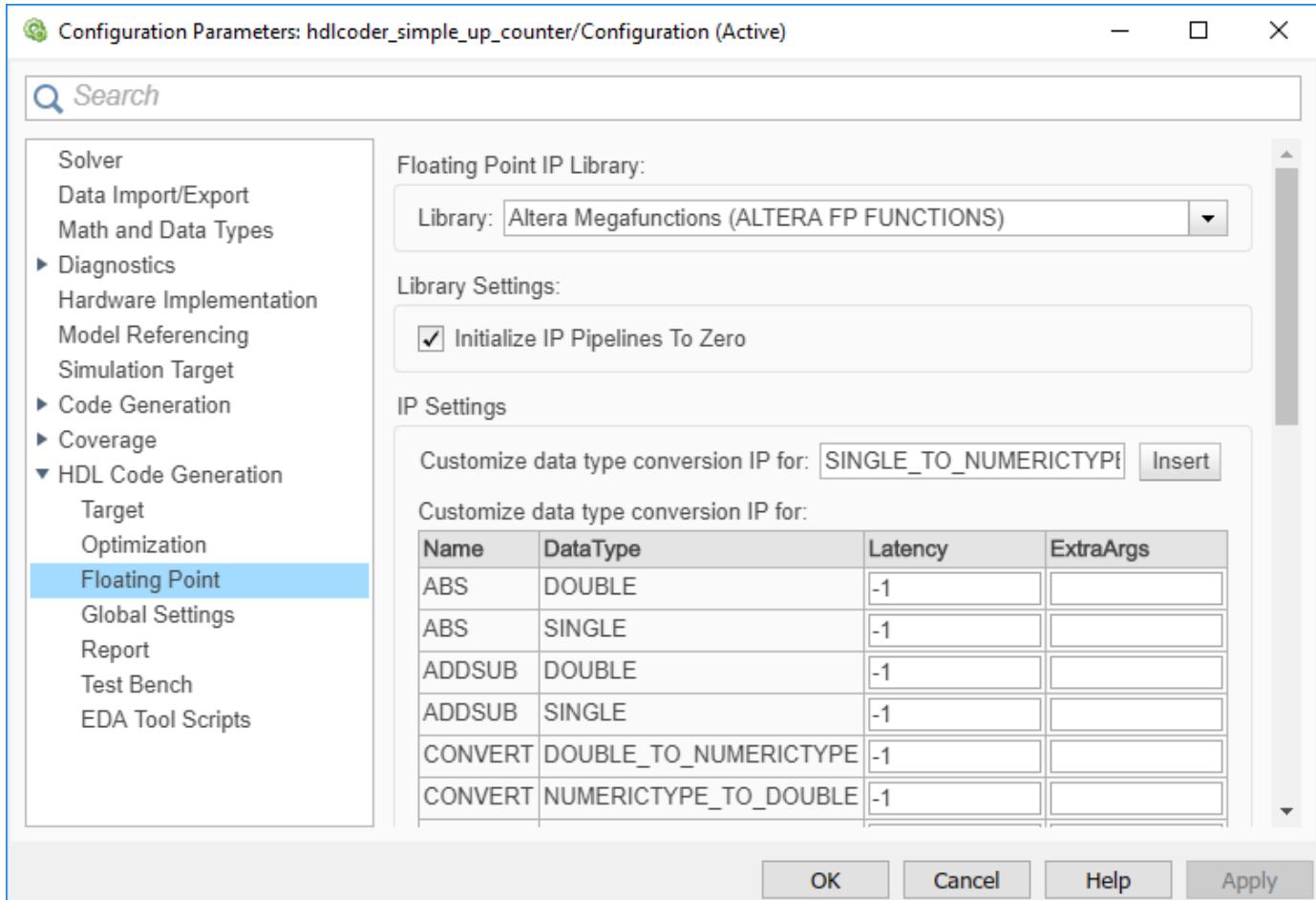
You can map your Simulink model to floating-point target libraries from the Configuration Parameters dialog box or from the command line.

### From the Configuration Parameters Dialog Box

To map to an FPGA floating-point library:

- 1 In the **Apps** tab, select **HDL Coder**. The **HDL Code** tab appears. Click **Settings**.

- 2 In the **HDL Code Generation > Floating Point Target** pane, select the floating-point IP library.



- 3 For Xilinx LogiCORE IP, select **XILINX LOGICORE** as the library. For Altera megafunction IP, you can select **ALTERA MEGAFUNCTION (ALTFP)** or **ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)** as the library.
- 4 If you choose **ALTERA MEGAFUNCTION (ALTERA FP FUNCTIONS)** as the library, the **Initialize IP Pipelines to Zero** option becomes available. Select the **Initialize IP Pipelines to Zero** option to initialize pipeline registers in the IP to zero. In the **Target and Optimizations** pane, enter the target frequency that you want the floating-point IP to map to.

**Note** When mapping to ALTERA FP FUNCTIONS, the target language must be set to VHDL.

When you choose the **ALTERA FP FUNCTIONS** library, an IP Configuration table appears. By using the data type table, you can customize the IP settings of the floating-point target library. For more information, see “Customize the IP Latency with Target Frequency” on page 31-40.

- 5 If you choose **XILINX LOGICORE** or **ALTERA MEGAFUNCTION (ALTFP)** as the library, select the **Latency Strategy** and **Objective** for the IP.

When you choose these libraries, an IP Configuration table appears. By using the data type table, you can customize the latency of the floating-point target IP. For more information, see “Customize the IP Latency with Latency Strategy” on page 31-43.

- 6 To share floating-point IP resources, on the **HDL Code Generation > Target and Optimizations > Resource Sharing** tab, make sure that **Floating-point IPs** is enabled. The number of floating-point IP blocks that get shared depends on the **SharingFactor** that you specify on the subsystem.
- 7 Click **Apply**. On the Simulink Toolstrip, click **Generate HDL Code**.

### From the Command-Line

To generate HDL code from the command line, you can use the `hdlcoder.createFloatingPointTargetConfig` function to create a floating-point IP configuration.

- 1 By using the `hdlcoder.createFloatingPointTargetConfig` function, create a `hdlcoder.FloatingPointTargetConfig` object for the floating-point library. Then, use `hdlset_param` to save the configuration on the model.

For example, to create a floating-point target configuration for the ALTERA FP FUNCTIONS library with the default settings:

```
fpconfig = hdlcoder.createFloatingPointTargetConfig('ALTERAFAULTFUNCTIONS');
hdlset_param('sfir_single', 'FloatingPointTargetConfiguration', fpconfig);
```

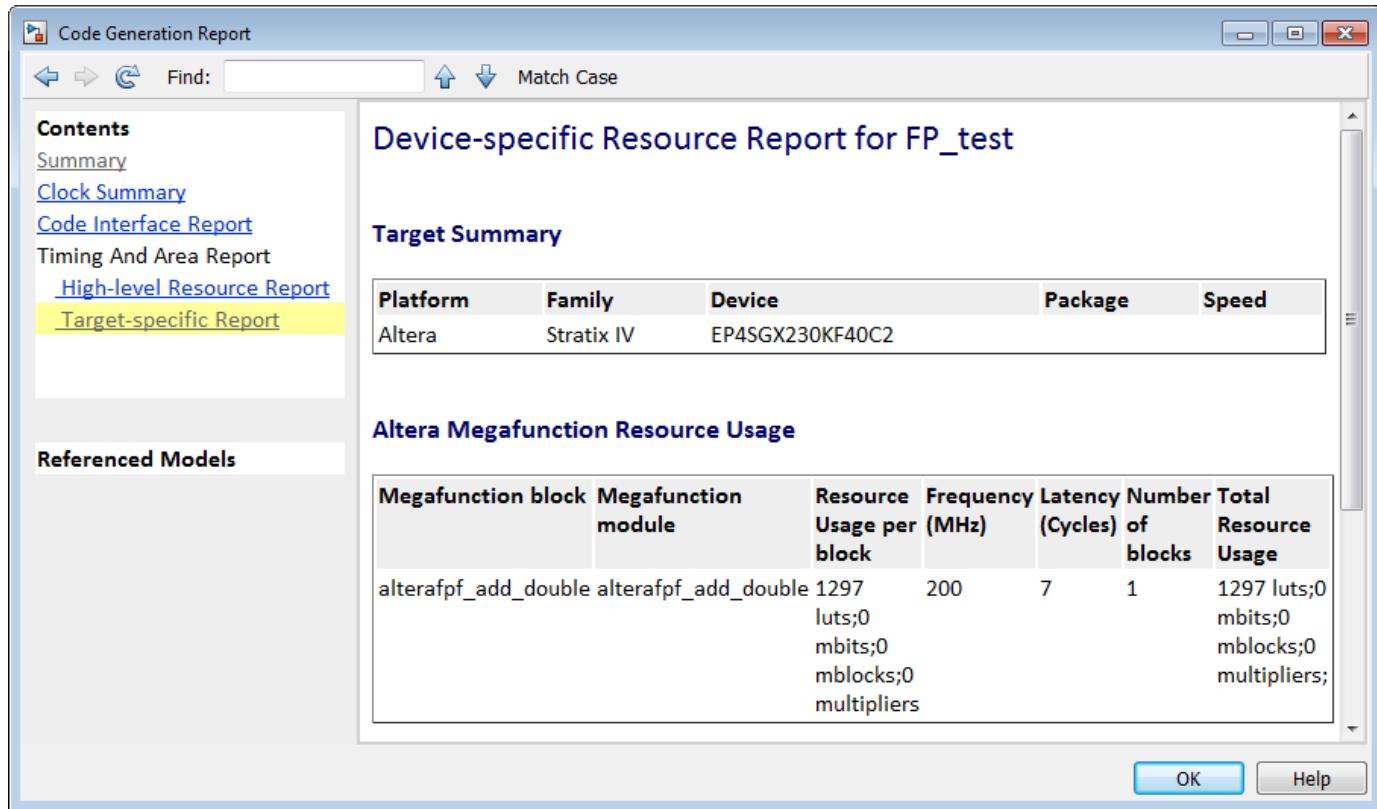
- 2 You can customize the IP settings based on the floating-point library that you specify. For more information, see “Customize Floating-Point IP Configuration” on page 31-39.
- 3 Use `makehdl` to generate HDL code from the subsystem.

## View Code Generation Reports of Floating-Point Library Mapping

To view the code generation reports of floating-point library mapping, before you begin code generation, enable generation of the Resource Utilization Report and Optimization Report. To learn how to generate these reports, see “Create and Use Code Generation Reports” on page 25-2.

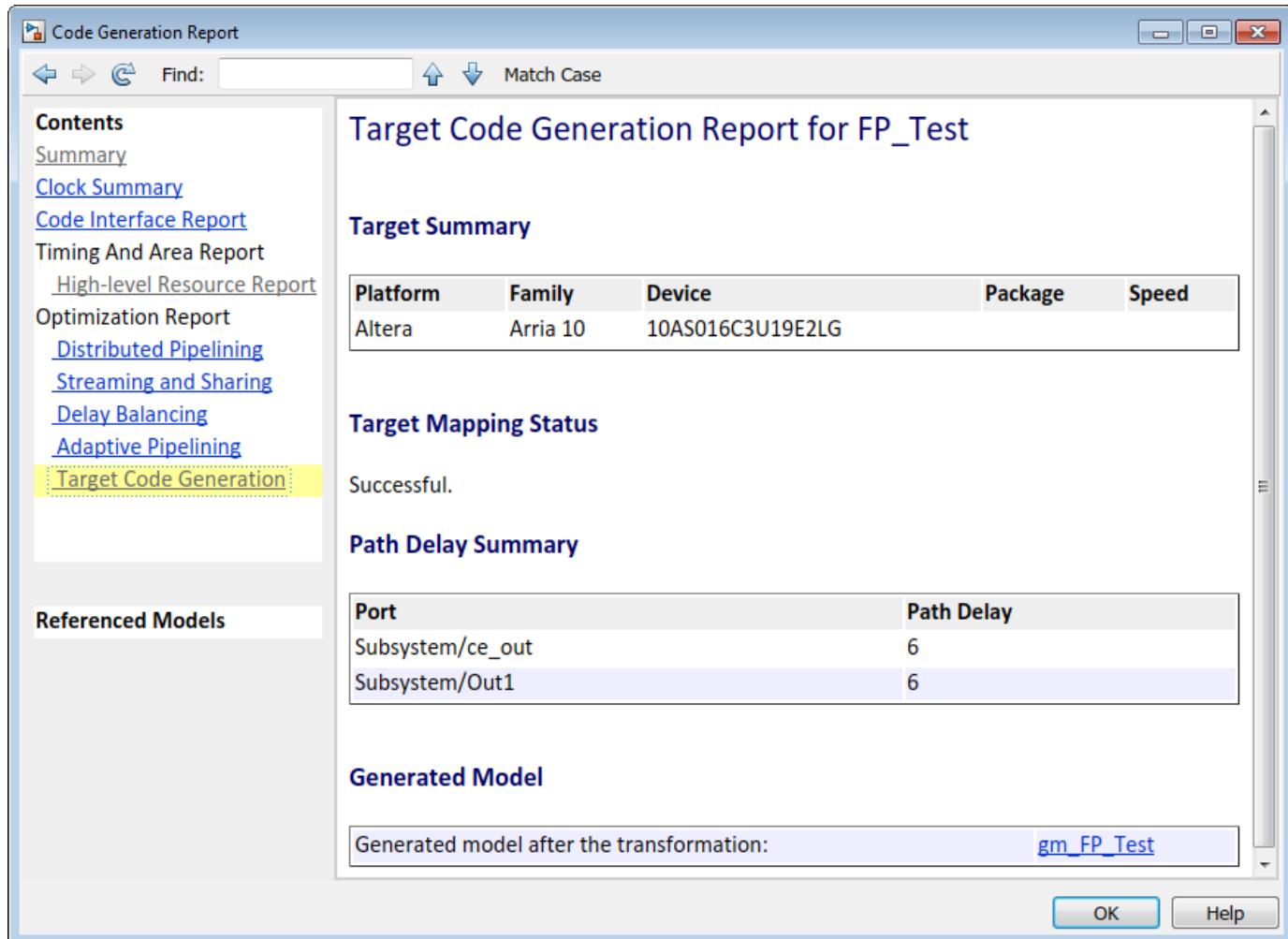
### Target-specific Report

To see the target floating-point block your design mapped to, the latency, and number of target-specific hardware resources, in the Code Generation Report, select **Target-specific Report**.



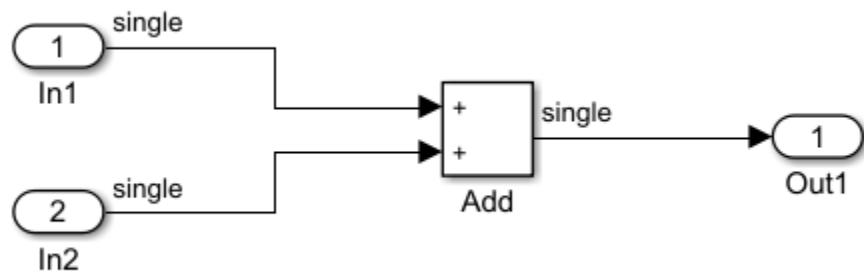
### Target Code Generation Report

In the Code Generation Report, the **Target Code Generation** section in the Optimization Report shows the status of optimization settings applied to the model. The report shows whether HDL Coder successfully generated floating-point target code.

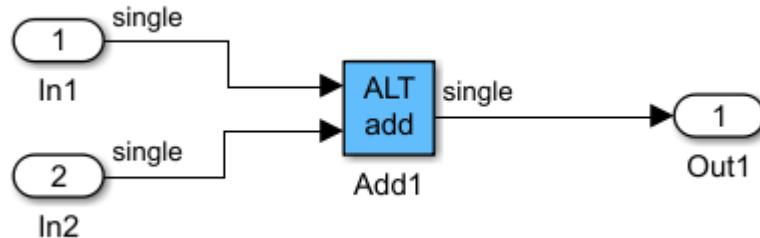


## Analyze Results of Floating-Point Library Mapping

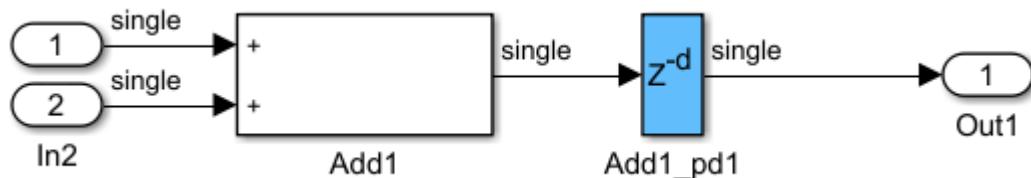
You can get the latency information of the floating-point target IP from the generated model after HDL code generation. For example, consider this add block in Simulink with inputs of double data type.



- After HDL code generation, the optimization report for target code generation displays a link to a generated model. To see the floating-point target library that your Simulink block mapped to, double-click the subsystem in the generated model.

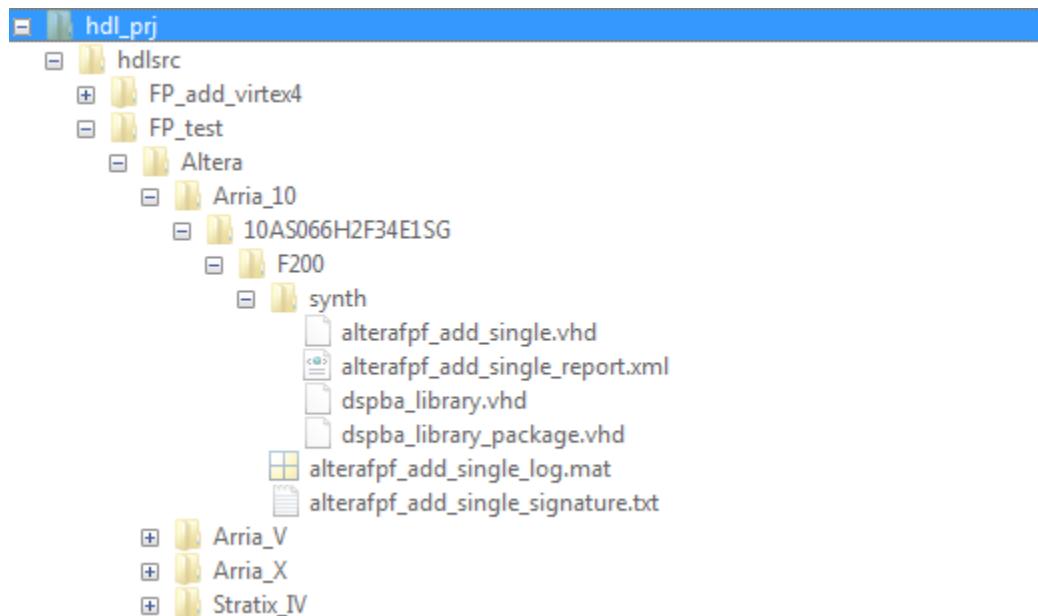


- Double-click the **ALT add** block. The length of the delay block is the latency of the floating-point target IP.



To learn more about the generated model, see “Generated Model and Validation Model” on page 24-10.

To see your FPGA floating-point library mapping results, you can view the IP core files generated after HDL code generation.



HDL Coder checks and reuses existing generated IP core files, taking less time when successively generating code for the same floating-point target IP.

## See Also

### Related Examples

- “FPGA Floating-Point Library IP Mapping” on page 31-27

### More About

- “Customize Floating-Point IP Configuration” on page 31-39
- “HDL Coder Support for FPGA Floating-Point Library Mapping” on page 31-47

# FPGA Floating-Point Library IP Mapping

This example illustrates the floating-point workflow that integrates IP libraries provided by vendors, such as Altera and Xilinx. For more information on how to map designs to floating-point libraries, see “Generate HDL Code for FPGA Floating-Point Target Libraries” on page 31-20.

## Introduction

Implementing designs with floating-point arithmetic enable you to model with higher precision and wider dynamic range and saves time by skipping floating-point to fixed-point conversion. This is particularly beneficial for model-based design, where high-level algorithms are modeled with floating-point and does not have implementation timing details, such as pipelining and timing constraints. However, they are necessary to map operations to floating-point IP modules. HDL Coder automatically optimizes and implements your designs with these timing details and provides interfaces for you to adjust them. Floating-point math is finally implemented by integrating with floating-point IP modules from vendor libraries.

## Mapping designs to floating-point IP libraries

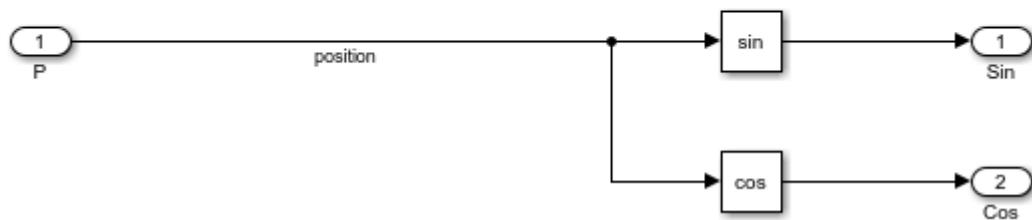
This Field-Oriented Control (FOC) algorithm example demonstrates the basic steps in this workflow to map designs to floating-point libraries. See the example “Field-Oriented Control of a Permanent Magnet Synchronous Machine” on page 10-55 for details about this application.

This model uses single-precision and contains blocks that perform basic math operators, such as adders, multipliers, comparators, and complex sin and cos functions.

Signal rates in this model are modeled at 20  $\mu\text{s}$  or 50 KHz only. Notice that this model contains only the numerical implementation, and doesn't have any FPGA implementation timing details, such as operation latencies. All numerical operations, including sin and cos functions, compute in a single sample time-step.

```
open_system('hdlcoderFocCurrentSingleTargetHdl');
open_system('hdlcoderFocCurrentSingleTargetHdl/FOC_Current_Control/Sine_Cosine');
```

## Sine Cosine Coefficients



## Choosing an IP library

In order to map to a vendor floating-point library, set the FPGA device.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SynthesisToolChipFamily', 'Arria 10');
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SynthesisTool', 'Altera Quartus II');
```

Setup target library tools.

```
hdlsetupoolpath('ToolName', 'Altera Quartus II', 'ToolPath', quartuspath);
hdlsetupoolpath('ToolName', 'XILINX ISE', 'ToolPath', iseopath);
```

Prepending following Altera Quartus II path(s) to the system path:

```
F:\hub\hub_share\share\apps\HDLTools\Altera\18.1-mw-0\Windows\quartus\bin64
Setting XILINX environment variable to:
```

```
F:\hub\hub_share\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\ISE
```

Setting XILINX\_EDK environment variable to:

```
F:\hub\hub_share\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\EDK
```

Setting XILINX\_PLANAHEAD environment variable to:

```
F:\hub\hub_share\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\PlanAhead
```

Prepending following XILINX ISE path(s) to the system path:

```
F:\hub\hub_share\share\apps\HDLTools\Xilinx_ISE\14.7-mw-0\Win\ISE_DS\ISE\bin\nt64;F:\hub\hub_sh
```

quartuspath and iseopath return synthesis tool paths in our environment. Refer to `hdlsetupoolpath` for how to setup tools in your environment.

The first step is to choose a vendor library. For Xilinx devices, you can use 'XILINXLOGICORE', and for Altera devices, you can select 'ALTERAFTPFUNCTIONS' or 'ALTFP'. Check library documentation for their supported devices.

Create a floating-point target configuration object for ALTERAFTPFUNCTIONS.

```
fc = hdlcoder.createFloatingPointTargetConfig('ALTERAFTPFUNCTIONS');
```

Set the configuration object on the model

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'FloatingPointTargetConfiguration', fc);
```

To compile and simulate the generated code with QuestaSim, you have to compile Altera simulation library and set its path on model with the SimulationLibPath parameter. Check "Tool Setup" for more information. alterasimulationlibpath returns the path to the compiled Altera simulation library in our environment.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'SimulationLibPath', alterasimulationlibpath);
```

Altera Megafunction (ALTERAFTPFUNCTIONS) library allows generating IP modules for a given target frequency. In this example, the target frequency is set to 250MHz.

```
hdlset_param('hdlcoderFocCurrentSingleTargetHdl', 'TargetFrequency', 250);
```

### Remodeling for IP mapping

Generate code

```
try
 makehdl('hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control');
catch me
 disp(me.message);
end
```

```
Generating HDL for 'hdlcoderFocCurrentSingleTargetHdl/F0C_Current_Control'.
```

```
Using the config set for model <a href="matlab:configset.showParameterGroup('hdlcoderFocCur
```